

# 线程同步实习报告

姓名 张煌昭 学号 1400017707  
日期 2017.10.16

## 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.....	10
内容四：收获及感想.....	11
内容五：对课程的意见和建议.....	11
内容六：参考文献.....	11

## 内容一：总体概述

本次 LAB 针对 Nachos 的线程同步机制进行了研究，在信号量的基础之上实现了锁和条件变量，并利用三种同步机制解决生产者-消费者问题。之后再在此基础上，实现了 Barrier 线程同步机制和读写锁，并对 Nachos 上的 K\_FIFO 机制的移植可行性进行了讨论。

## 内容二：任务完成情况

### 任务完成列表 (Y/N)

Exercise	Exercise	Exercise	Exercise	Challenge	Challenge	Challenge
1	2	3	4	1	2	3
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。

参考 Linux 同步方法剖析. M. Tim Jones. Nov 19, 2007.<sup>[1]</sup>，对 Linux 的同步机制调研如下，其中大家所熟悉的，常见的机制将不再进行调研介绍。

##### 原子操作

临界段被包裹在 API 函数内部，API 内实现了锁定，因而并不需要额外的锁定操作。

使用原子操作，实现需要声明一个 `atomic_t` 类型的原子变量，之后使用符号常量或函数进行初始化。此外原子操作支持简单算术运算和 bitmask 操作。

##### 自旋锁

自旋锁使用忙等待来确保互斥。如果自旋锁可被获取，则获取自旋锁，执行互斥操作，然后释放自旋锁；如果自旋锁被占用而不可被获取，那么线程将在 CPU 上忙等待，直到自旋锁可用为止。一般情况下，使用自旋锁时，要求锁定时间或忙等待时间小于两个上下文切换时间。自旋锁具有完全锁和读写锁两种形式。

完全锁具有 `lock` 和 `unlock` 函数，可以对变量和操作进行互斥锁定，它会关闭中断以保证原子性。

读写锁满足了多数情况下，对数据的访问读极多写较少的情况。该锁允许多线程同时访问相同数据区，但同一时刻只能允许一个线程写入数据。若一写线程持有读写锁，则临界段不可被其他线程读取；若一读线程持有读写锁，那么多个读线程都可以进入临界段。

##### 内核互斥锁

内核中用来实现信号量行为的互斥锁，在原子 API 之上实现。同一时间内只能有一个任务持有互斥锁，并且当且仅当这个任务可以对互斥锁解锁，互斥锁不能进行递归锁定或递归解锁，并且互斥锁不可用于交换上下文。

互斥锁 API 提供了三个锁定函数，一个解锁函数，和一个用于测试的函数。

## 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` 和 `code/threads/synch.cc` 中实现了三个同步机制——信号量 Semaphore，锁 Lock，和条件变量 Condition。

信号量 Semaphore 类中具有私有成员变量 `value` 和队列 `queue`，并具有 P 和 V 两种方法，其中 `value` 必须为非负。每一次调用 P 将检查 `value` 是否为 0，若为 0 则将该线程入队并 Sleep 等待信号量；若大于 0，则将 `value` 减一后返回。每一次调用 V 将唤醒 `queue` 中的一个 Sleep 线程，并将 `value` 加一。Semaphore 中所有操作都是原子的，即中断关闭的。

锁 Lock 类和条件变量 Condition 类没有实现，将在 Exercise 3 中进行实现。

Lock 类只具有 FREE 和 BUSY 两个状态，使用 Acquire 方法将获取 FREE 状态的 Lock 并将其置为 BUSY，若 Lock 此时为 BUSY，则无法获取，Sleep 直至可以获取；Release 方法将获取的 BUSY 状态的锁状态设置为 FREE，并从 Sleep 的线程队列中唤醒一个线程。要求所有操作都是原子的。

Condition 类中没有值，但有一个线程队列，需要与一个外部的 Lock 一起使用。使用 Wait 方法将首先 Release Lock，之后将线程入队 Sleep，最后 Acquire Lock；使用 Signal 方法将从队列中唤醒一个线程；使用 Broadcast 方法将唤醒所有队列中 Sleep 的线程。所有操作都必须是原子的，并且要求当前线程获取了 Lock。

`code/threads/synchlist.h` 和 `code/threads/synchlist.cc` 中实现了同步链表 SynchList 类，该类使得每次都只能由一个进程对链表进行访问，并且如果链表内没有元素则会 Sleep 等待，直到其他进程放入元素。

## 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）。

### 使用 Semaphore 实现 Lock

首先考虑 Lock 类中需要包括哪些成员变量——使用 Semaphore 作为同步原语，因此必须使用一个信号量变量 `lock`；由于方法 `isHeldByCurrentThread` 的存在，必须有一个 Thread 指针 `held_by` 指明当前 Lock 被哪个线程获取。

下面考虑对成员函数的实现，构造函数、析构函数，以及 `isHeldByCurrentThread` 函数都比较简单，不进行详细的说明，按照规则即可，需要注意的是由于锁的特殊性质，`lock` 在构造时 `value` 应为 1。Acquire 函数中首先在开始处第一句关中断，在结束处最后一句关中断，保证操作的原子性；中间部分先 `lock->P()` 获取信号量，之后将 `held_by` 指示为 `currentThread`。Release 函数进行类似的操作，首先用开关中断保证操作原子性；中间部分先将 `held_by` 指示为 NULL，之后 `lock->V()` 释放信号量。由于信号量的特殊性质，所有 P 操作都应该靠前，所有 V 操作都应该靠后，当然对于特殊问题，还需要考虑死锁等情况。

### 使用 Lock 实现 Condition

由于 Condition 需要与外部的 Lock 同时使用，因为其内部不需要额外的 Lock，只需要添加一个 List 成员变量 waitLine 即可。

考虑成员函数的实现，构造函数和析构函数按照规则即可，下面主要说明 Wait, Signal 和 Broadcast 三个成员函数。所有函数都需要开关中断的操作以保证原子性，下面不进行特殊说明。Wait 函数首先断言检查 conditionLock 是否已经被当前线程获取，若为假则退出；之后按照步骤进行操作——第一步释放 conditionLock，第二步当前线程入队 waitLine 并 Sleep，第三步获取 conditionLock。Signal 函数 if 判断 conditionLock 是否被当前线程获取，若为否则直接退出；如果 waitLine 非空，则从中取出第一个线程并将之唤醒置入 READY 队列。Broadcast 函数同 Signal 类似，但区别在于 Broadcast 函数会唤醒 waitLine 中的所有 Sleep 线程。

### 实验证明

使用上述同步机制编写生产者-消费者问题实验，实验的具体说明和结果请见下一部分 Exercise 4。

## 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）

### 生产者-消费者问题

首先说明什么是生产者消费者问题。假设有一个仓库，其中可以存放若干件货物，有若干的生产者 Producer 和若干的消费者 Consumer 会使用这一仓库——Producer 不断生产货物，并将其不作位置区分地放入仓库空位中；Consumer 不断消费货物，从仓库中放了货物的位置取出货物。要求当仓库放满，即没有空位时，Producer 不再生产，即不能向仓库中存放任何货物；当仓库全空，即没有任何位置有货物时，Consumer 不再消费，也即不能从仓库中获得任何货物。

如果将每个生产者或者消费者视为一个线程，各个线程共享了一段存储区域（仓库），该问题实际上就是线程的同步问题。下面使用 Semaphore 和 Lock + Condition 两种方法解决这一问题。

为了方便下面使用，定义仓库大小上限为 100，用 production\_produced\_by 和 production\_name\_idx 两个数组进行表示。编写一系列函数对仓库进行操作，具体包括 production\_append 函数，将一个货物放到仓库里的一个空位；production\_get\_name 和 production\_get\_producer 函数，找到仓库里的第一个货物，返回其名称或制造者；production\_remove 函数，移除仓库里的第一个货物；production\_print 函数，打印仓库里存放的所有货物的信息；production\_occupied 函数，返回仓库里存放的货物数目。由于只在这一个问题内使用，因此没有将其包装成类，而是直接在 code/threads/threadtest.cc 内编写各个函数。

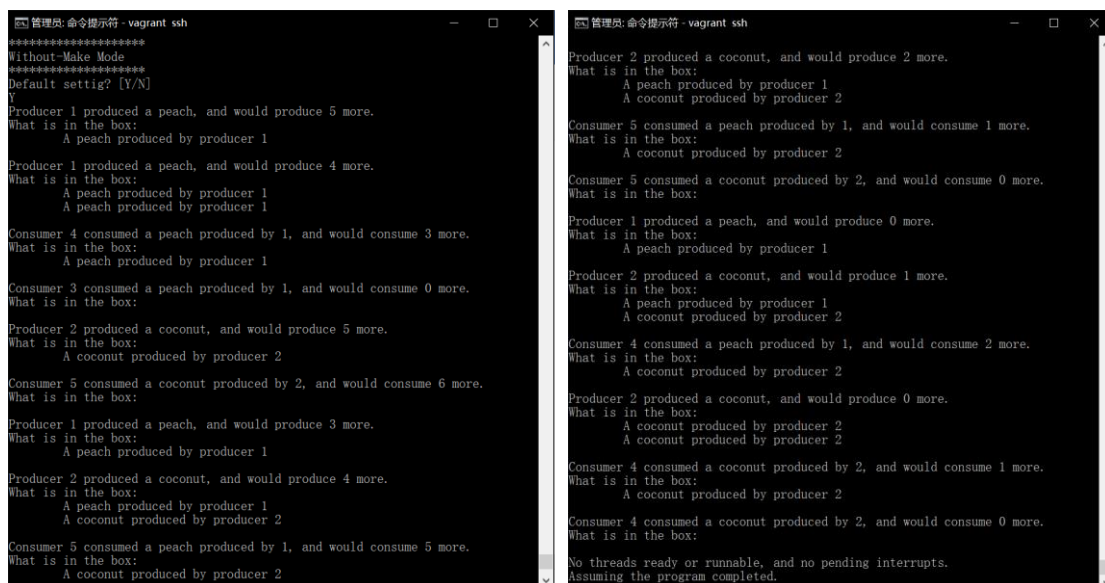
### 使用 Semaphore 解决生产者-消费者问题

使用 3 个信号量：semaphore\_mutex，semaphore\_full 和 semaphore\_empty。其中 semaphore\_mutex 初值为 1，作为访问“仓库”的互斥锁使用；semaphore\_full 初值为 0，表示仓库中有几个位置放了货物；semaphore\_empty 初值为 box\_cap（box\_cap 为一变量，表示仓库有几个位置），表示仓库中有几个位置是空的。

为 Producer 线程编写 semaphore\_producer 函数, 该函数具有一个参数 n, 表示该 Producer 一共生产多少个货物。循环 n 次, 并进行如下操作: 首先随机决定是否 Yield (为了方便测试), 之后 semaphore\_empty->P() 占用仓库中的一个空位, 再 semaphore\_mutex->P() 获取仓库的互斥锁以便进行操作; 调用 production\_append 函数将货物放入仓库并打印各个信息; semaphore\_mutex->V() 释放互斥锁, semaphore\_full->V() 添加一个仓库占位。需要说明的是两个信号量的 P 操作的顺序十分重要, 必须先占用空位再获取互斥锁, 否则将引起死锁。

类似地为 Consumer 线程编写 semaphore\_consumer 函数, 该函数具有一个参数 n, 表示该 Consumer 一共消费多少个货物。循环 n 次, 并进行如下操作: 首先随机决定是否 Yield, 之后 semaphore\_full->P() 占用一个占位, 之后 semaphore\_mutex->P() 获取互斥锁; 调用 production\_remove 函数移出一个货物并打印各个信息; semaphore\_mutex->V() 释放互斥锁, semaphore\_empty->V() 添加一个仓库空位。

编写测试函数 ThreadTest8, 创建两个生产者线程, 各生产 6 个货物; 创建三个消费者线程, 分别消费 1、4、7 个货物; 仓库大小设置 2。部分实验结果如下图。其中, 当仓库放满时所有生产者将等待仓库有空位; 当仓库完全空时, 所有消费者将等待仓库有货物。完全符合预期要求。



```
*****
Without-Make Mode
*****
Default settig? [Y/N]
Y
Producer 1 produced a peach, and would produce 5 more.
What is in the box:
A peach produced by producer 1
Producer 1 produced a peach, and would produce 4 more.
What is in the box:
A peach produced by producer 1
A peach produced by producer 1
Consumer 4 consumed a peach produced by 1, and would consume 3 more.
What is in the box:
A peach produced by producer 1
Consumer 3 consumed a peach produced by 1, and would consume 0 more.
What is in the box:
Producer 2 produced a coconut, and would produce 5 more.
What is in the box:
A coconut produced by producer 2
Consumer 5 consumed a coconut produced by 2, and would consume 6 more.
What is in the box:
Producer 1 produced a peach, and would produce 3 more.
What is in the box:
A peach produced by producer 1
Producer 2 produced a coconut, and would produce 4 more.
What is in the box:
A peach produced by producer 1
A coconut produced by producer 2
Consumer 5 consumed a peach produced by 1, and would consume 5 more.
What is in the box:
A coconut produced by producer 2

Producer 2 produced a coconut, and would produce 2 more.
What is in the box:
A peach produced by producer 1
A coconut produced by producer 2
Consumer 5 consumed a peach produced by 1, and would consume 1 more.
What is in the box:
A coconut produced by producer 2
Consumer 5 consumed a coconut produced by 2, and would consume 0 more.
What is in the box:
Producer 1 produced a peach, and would produce 0 more.
What is in the box:
A peach produced by producer 1
Producer 2 produced a coconut, and would produce 1 more.
What is in the box:
A peach produced by producer 1
A coconut produced by producer 2
Consumer 4 consumed a peach produced by 1, and would consume 2 more.
What is in the box:
A coconut produced by producer 2
Producer 2 produced a coconut, and would produce 0 more.
What is in the box:
A coconut produced by producer 2
A coconut produced by producer 2
Consumer 4 consumed a coconut produced by 2, and would consume 1 more.
What is in the box:
A coconut produced by producer 2
Consumer 4 consumed a coconut produced by 2, and would consume 0 more.
What is in the box:
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```

## 使用 Lock 和 Condition 解决生产者-消费者问题

使用一个 Lock cl\_lock 和两个 Condition cl\_pro\_wait 和 cl\_con\_wait。其中 cl\_pro\_wait 为处于等待状态的生产者队列, cl\_con\_wait 为处于等待状态的消费者队列。

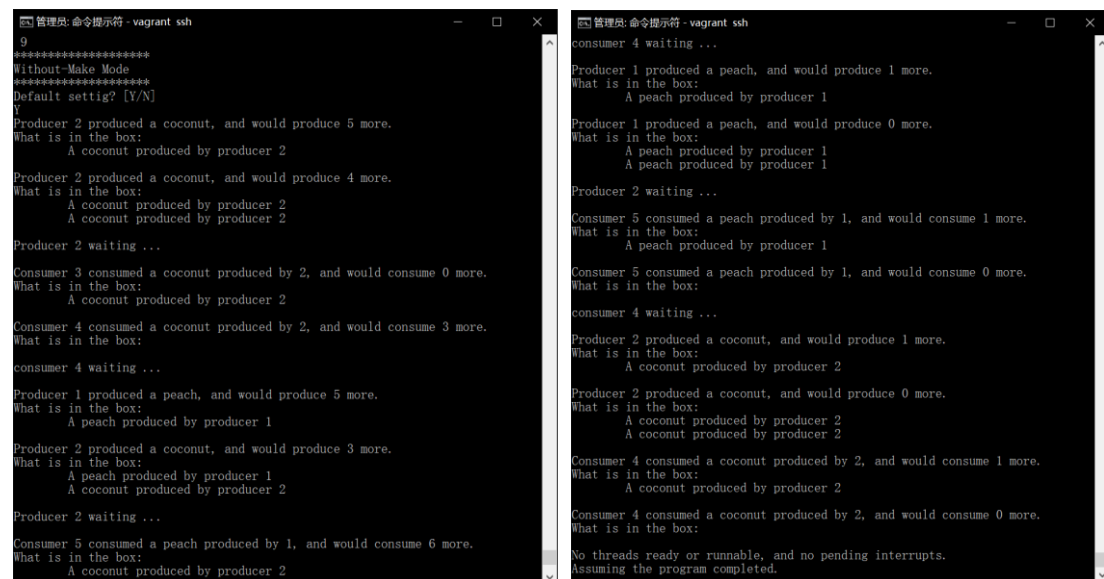
为 Producer 线程编写 condition\_lock\_producer 函数, 该函数具有一个参数 n, 表示该 Producer 一共生产多少个货物。循环 n 次, 并进行如下操作: 随机决定是否 Yield, 获取锁 cl\_lock; while 调用 production\_occupied 判断仓库里的货物数是否达到 box\_cap, 若是则使用 cl\_pro\_wait->Wait 函数将当前生产者加入等待队列, 并打印信息; 否则调用 production\_append 函数将货物放入仓库并打印信息, 完成后调用 cl\_con\_wait->Broadcast() 唤醒所有等待的 Consumer; 最后释放 cl\_lock。

类似地为 Consumer 线程编写 condition\_lock\_consumer 函数, 该函数具有一个参数 n, 表示该 Consumer 一共消费多少个货物。循环 n 次, 并进行如下操作: 随机决定是否 Yield, 获取锁 cl\_lock; while 调用 production\_occupied 判断仓库里的货物数是否到了 0, 若是则使用 cl\_con\_wait->Wait 函数将当前消费者加入等待队列, 并打印信息; 否则调用

`production_remove` 函数将货物移出仓库并打印信息，完成后调用 `cl_pro_wait->Broadcast()` 唤醒所有等待的生产者；最后释放 `cl_lock`。

需要注意的是，我使用了 `Broadcast` 函数唤醒所有等待的 `Producer` 或 `Consumer`，因此对应的在检查时需要使用 `while` 循环；如果使用 `Signal` 函数唤醒，则可以使用 `if` 判断进行检查。

编写测试函数 `ThreadTest9`，创建两个生产者线程，各生产 6 个货物；创建三个消费者线程，分别消费 1、4、7 个货物；仓库大小设置 2。部分实验结果如下图。其中，当仓库放满时所有生产者将等待仓库有空位；当仓库完全空时，所有消费者将等待仓库有货物。完全符合预期要求。



```
g
*****
Without-Make Mode
*****
Default settig? [Y/N]
Y
Producer 2 produced a coconut, and would produce 5 more.
What is in the box:
A coconut produced by producer 2
Producer 2 produced a coconut, and would produce 4 more.
What is in the box:
A coconut produced by producer 2
A coconut produced by producer 2
Producer 2 waiting ...
Consumer 3 consumed a coconut produced by 2, and would consume 0 more.
What is in the box:
A coconut produced by producer 2
Consumer 4 consumed a coconut produced by 2, and would consume 3 more.
What is in the box:
consumer 4 waiting ...
Producer 1 produced a peach, and would produce 5 more.
What is in the box:
A peach produced by producer 1
Producer 2 produced a coconut, and would produce 3 more.
What is in the box:
A peach produced by producer 1
A coconut produced by producer 2
Producer 2 waiting ...
Consumer 5 consumed a peach produced by 1, and would consume 6 more.
What is in the box:
A coconut produced by producer 2

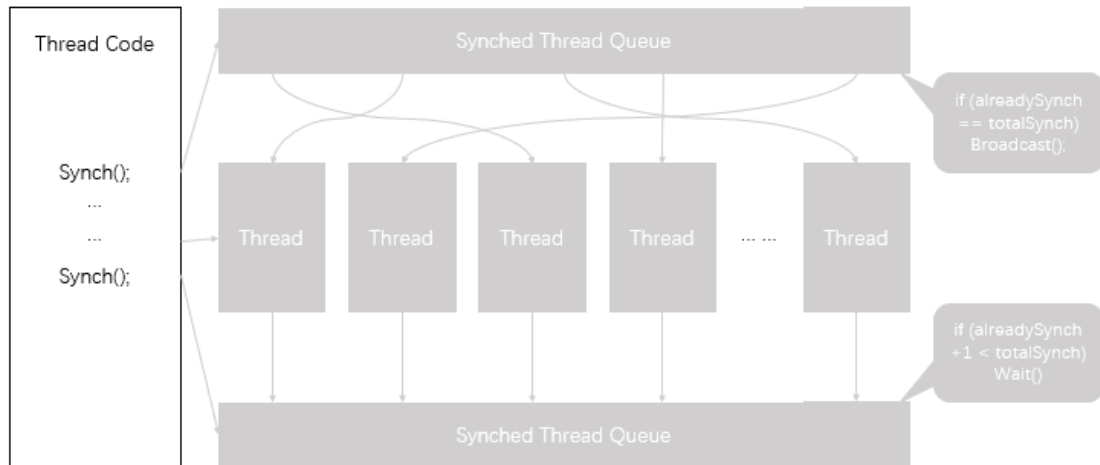
consumer 4 waiting ...
Producer 1 produced a peach, and would produce 1 more.
What is in the box:
A peach produced by producer 1
Producer 1 produced a peach, and would produce 0 more.
What is in the box:
A peach produced by producer 1
A peach produced by producer 1
Producer 2 waiting ...
Consumer 5 consumed a peach produced by 1, and would consume 1 more.
What is in the box:
A peach produced by producer 1
Consumer 5 consumed a peach produced by 1, and would consume 0 more.
What is in the box:
consumer 4 waiting ...
Producer 2 produced a coconut, and would produce 1 more.
What is in the box:
A coconut produced by producer 2
Producer 2 produced a coconut, and would produce 0 more.
What is in the box:
A coconut produced by producer 2
A coconut produced by producer 2
Consumer 4 consumed a coconut produced by 2, and would consume 1 more.
What is in the box:
A coconut produced by producer 2
Consumer 4 consumed a coconut produced by 2, and would consume 0 more.
What is in the box:
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
```

### Challenge 1 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。

考虑多线程的同步方式，各个线程任意运行至某个特定阶段后，等待其他需要同步的线程；当所有线程都运行到达目标位置后，唤醒所有待同步的进程，之后继续任意运行至下一个特定阶段。因此可以使用 `Lock` 和 `Condition`，以及在线程代码中插入同步点达到这一要求。

整个机制如下图所示。



仿照 SynchList 在 code/threads/synchlist.h 和 code/threads/synchlist.cc 中添加 Barrier 类。Barrier 中包含私有常量 totalSynch 和私有变量 alreadySynch，锁 lock，条件变量 condition；其中 totalSynch 为需要进行同步的线程总数，alreadySynch 为已经同步了的线程数目。除构造和析构函数外，Barrier 只有一个用于向线程代码插入同步点的函数 Synch。Synch 首先获取锁 lock，之后判断同步线程计数是否达到需要同步的线程总数 totalSynch，若未达到，则将同步线程计数加一并 condition->Wait 使该线程等待；否则将同步线程计数置为 0 并 condition->Broadcast 唤醒所有同步等待的线程；最后释放锁 lock。

编写同步线程函数 synchThread 和非同步线程函数 unsynchThread。synchThread 循环 n 次，并执行如下操作：随机决定是否 Yield，调用 Synch 函数同步，打印线程 TID 和循环轮数，调用 Synch 函数同步。unsynchThread 与 synchThread 操作相同，但不调用 Synch 同步。编写 ThreadTest10 测试同步，实验结果如下。同步线程循环打印，非同步线程乱序打印，符合预期要求。

```

管理: 命令提示符 - vagrant ssh
Cleaning up...
vagrant@precise32:/vagrant/nachos/nachos-3.4/code$ python onekeyrun.py -n -q 10
*****
Without-Make Mode
*****
SYNCHRONIZED SCENE:
thread 1, loop 0
thread 2, loop 0
thread 3, loop 0
thread 2, loop 1
thread 3, loop 1
thread 1, loop 1
thread 3, loop 2
thread 1, loop 2
thread 2, loop 2
UNSYNCHRONIZED SCENE:
thread 2, loop 0
thread 2, loop 1
thread 2, loop 2
thread 3, loop 0
thread 3, loop 1
thread 4, loop 0
thread 3, loop 2
thread 4, loop 1
thread 4, loop 2
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 824, idle 54, system 770, 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...
vagrant@precise32:/vagrant/nachos/nachos-3.4/code$

```

## 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。

假设需要使用 R/W Lock 的数据区为一维整形数组，将这样的数组包装在 RW\_Lock 类中。这样的数组，Read 函数不需要任何锁的操作，直接读取即可，Write 必须先获取 RW\_Lock，之后才能使用 Write 的函数。因而实现如下。

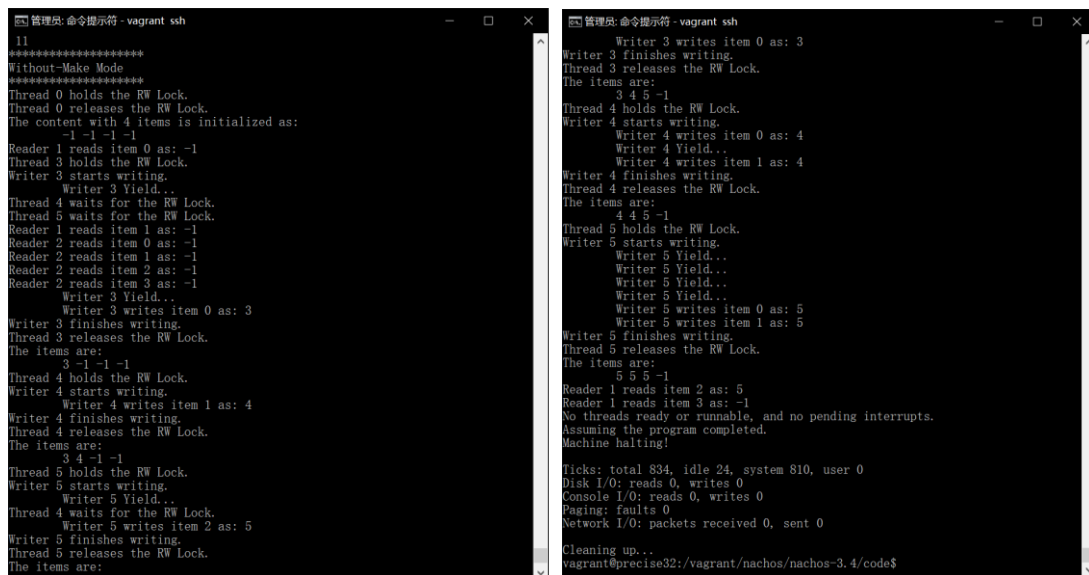
code/threads/synchlist.h 和 code/threads/synchlist.cc 中添加 RW\_Lock 类。RW\_Lock 包括私有常量数组大小 capacity，私有变量变动标记 change\_comp，整形数组 content，锁 lock，条件变量 condition，获取了 RW\_Lock 的线程 heldBy，RW\_Lock 锁值 used。成员函数除去构造和析构函数，还有获取写锁 lock\_acquire，释放写锁 lock\_release，带锁写 locked\_write，不带锁读 read，获取数组大小 get\_capacity，获取当前获得 R/W 锁的线程 get\_heldBy，获知当前 RW\_Lock 是否被 currentThread 获取。

lock\_acquire 函数首先 lock->Acquire()获取 lock；若 while (used > 0)满足，说明 RW\_Lock 已经被获取，使用 condition->Wait()将该线程挂起等待；否则将 used 加一并 heldBy 设为 currentThread 获取 RW\_Lock；最后 lock->Release()释放 lock。

类似的，lock\_release 函数首先 lock->Acquire()获取 lock；将 used 减一并 heldBy 设为 NULL，释放 RW\_Lock；之后 condition->Broadcast 唤醒所有等待线程（这其中至多有一个线程获取 RW\_Lock，而其他的会由于 while 条件继续满足而继续挂起）；最后 lock->Release()释放 lock。

locked\_write 函数将某个值写入数组某位置，只有当前线程获取了 RW\_Lock 时才允许写入，并返回 1 表示写入正常；否则返回-1 表示未获取锁而尝试写入。read 函数不进行锁的判断，直接读取数组某位置的值；但需要注意的是，如果在读的过程中，change\_comp 发生了变化（write 完成后会直接将 change\_comp 加一，说明数组内容发生了变化），则重新读取。

编写测试如下：创建两个读线程 rw\_read\_thread，不断尝试读取 rw\_lock 内容，创建三个写线程 rw\_write\_thread，不断尝试写 rw\_lock。部分实验结果如下，写线程互斥，读线程不受任何影响，符合预期要求。



```
ll
*****
Without-Make Mode
*****
Thread 0 holds the RW Lock.
Thread 0 releases the RW Lock.
The content with 4 items is initialized as:
-1 -1 -1 -1
Reader 1 reads item 0 as: -1
Thread 3 holds the RW Lock.
Writer 3 starts writing.
Writer 3 Yield...
Thread 4 waits for the RW Lock.
Thread 5 waits for the RW Lock.
Reader 1 reads item 1 as: -1
Reader 2 reads item 0 as: -1
Reader 2 reads item 2 as: -1
Reader 2 reads item 3 as: -1
Writer 3 Yield...
Writer 3 writes item 0 as: 3
Writer 3 finishes writing.
Thread 3 releases the RW Lock.
The items are:
3 -1 -1 -1
Thread 4 holds the RW Lock.
Writer 4 starts writing.
Writer 4 writes item 1 as: 4
Writer 4 finishes writing.
Thread 4 releases the RW Lock.
The items are:
3 4 -1 -1
Thread 5 holds the RW Lock.
Writer 5 starts writing.
Writer 5 Yield...
Thread 4 waits for the RW Lock.
Writer 5 writes item 2 as: 5
Writer 5 finishes writing.
Thread 5 releases the RW Lock.
The items are:
3 4 5 -1
Reader 1 reads item 2 as: 5
Reader 1 reads item 3 as: -1
No threads ready or runnable, and no pending interrupts.
Assuming the program completed.
Machine halting!

Ticks: total 834, idle 24, system 810, 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...
vagrant@precise32:/vagrant/nachos/nachos-3.4/code$
```

### Challenge 3 研究 kfifo

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

## KFIFO 机制

KFIFO 是一种 Linux 上的“环形数组”的同步机制，该机制的存储为一个一位数组，数组尾加一后返回数组头，从而实现“环形”；数组具有一个读指针，一个写指针，读操作与写操作不互斥，但读操作间互斥，写操作间互斥，即同时只能有一个进程读和一个进程写；此外，当读指针追上写指针时，数组为空无法再读，读进程挂起直至新内容写入被唤醒，写指针追上读指针是，数组满无法再写，写进程挂起直至有内容被读出。

总的来说，KFIFO 相当于生产者和消费者不互斥的生产者-消费者问题。

KFIFO 在 Linux 下采用自旋锁 SpinLock 完成读-读互斥和写-写互斥，这一点在目前的 Nachos 中并没有对应的机制，但并不意味着无法实现。

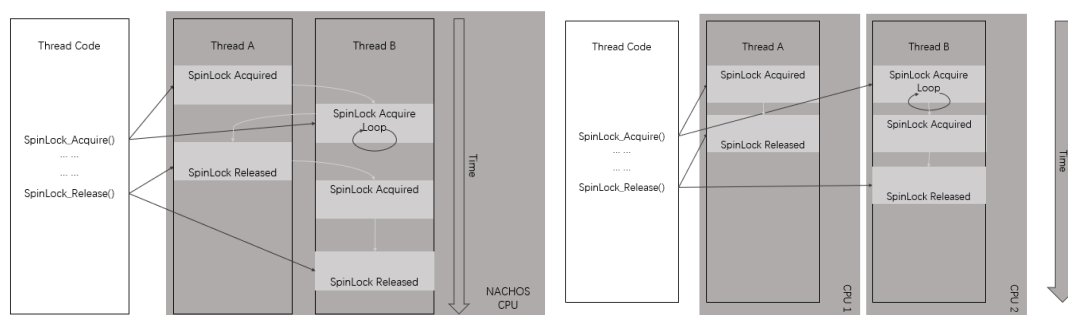
## Nachos 的 KFIFO 移植

我认为将 KFIFO 移植至 Nachos，不考虑代价等因素，仅仅考虑可行性的话是可行的一一环形数组非常容易实现，唯一的难点就在于自旋锁机制。下面说明自旋锁的实现方法。

自旋锁与 Nachos 中的 Lock 的区别在于，自旋锁无法获取时会在 CPU 上空转，而 Lock 无法获取时直接挂起，而 Lock 的挂起源自于 Semaphore 的 P 操作中的挂起。因此，可以向 Semaphore 添加一组新的操作 Spin\_P 和 Spin\_V，其中 Spin\_P 在无法获取信号量时不再挂起，而是不断循环，去尝试获取信号量。用这样一组新的操作，采用和 Lock 相同的方式，即可实现自旋锁 SpinLock。

下面考虑 KFIFO 移植在 Nachos 上的代价等不利因素。

Linux 系统中，SpinLock 用于对多 CPU 很短时间的操作进行同步，一般要求该时间小于两次 SWITCH 的时间。但目前的 Nachos 系统首先没有多核的概念，所有线程只能使用一个 CPU，因而自旋锁被获取后，不可能有另一个 CPU 也去尝试获取；此外，当某个线程获取了自旋锁但在释放前就用尽时间片的话，下一个需要获取自旋锁的线程必然会进入自旋态，从而必定空转整个时间片，这是极其浪费的——比起挂起，多耗了一个时间片，但最终的效果完全相同。因而在 Nachos 上使用自旋锁是没有意义的，可以直接用 Lock 替代。



由于仅仅研究该机制移植的可行性，我并没有进行代码实现。结论是可行，但我不认为有意义。

## 内容三：遇到的困难以及解决方法

由于本次 LAB 是关于同步互斥的，实验过程中经常会出现不可复现的 bug。最终使用 GDB 单步调试进行 debug，成功定位 bug 区域。

## 内容四：收获及感想

通过本次 LAB 对互斥、原子等概念有了进一步的理解，同时也利用 Nachos 现有的同步机制对几个同步问题进行了解决。

## 内容五：对课程的意见和建议

暂无。

## 内容六：参考文献

[1] M. Tim Jones. Linux 同步方法剖析 [DB/OL]

<https://www.ibm.com/developerworks/cn/linux/l-linux-synchronization.html>