同步机制实习报告

姓名　　马少鹏　　学号　1500012893
日期　　2018.3.24

目录

内容一：总体概述

本次实验通过调研 Linux 中采用的同步机制，理解 Linux 中的同步机制实现原理。通过阅读 Nachos 源代码，理解 Nachos 现有的同步机制。在此基础上，通过修改 Nachos 平台的源代码，达到“扩展同步机制，实现同步互斥实例”的目标。

内容二：任务完成情况

任务完成列表（Y/N）

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

具体 Exercise 的完成情况

第一部分

Exercise1 调研 Linux 中的同步机制

Linux 下线程的同步机制主要有：

信号量

POSIX 中定义了实现线程同步的信号量，使用的基本信号量函数：

初始化信号量 int sem_init()

信号量加一：int sem_post()

信号量减一：int sem_wait()

销毁信号量：int sem_destroy()

互斥锁

锁机制是同一时刻只允许一个线程执行一个关键部分的代码，实质就是一个二元信号量。基本的操作有：

初始化锁、阻塞加锁、非阻塞加锁、解锁、销毁锁。

条件变量

条件变量是利用线程间共享全局变量进行同步的一种机制。条件变量上的基本操作有：触发条件、等待条件并挂起线程直到其他线程触发条件。

条件变量必须和一个互斥锁配合，以防止多个线程同时请求。

除此之外，Linux 下实现线程同步还有其他方法。

Exercise2 源代码阅读

code/threads/synch.h 和 code/threads/synch.cc

synch.h 定义了 Nachos 实现线程同步的数据结构。

信号量：Semaphore

有 P()、V()操作。

互斥锁：Lock

有 Acquire() Release()操作。

条件变量 Condition()：

有 Wait() Signal()操作。

Synch.cc 中实现了 Semaphore 的成员函数，其他两个类需要我们自己完成。

code/threads/synchlist.h 和 code/threads/synchlist.cc

定义了一个可以同步增删元素、操作元素的链表。

通过使用互斥锁和条件变量对 append 操作、remove 操作和 mapcar 操作的保护，在非同步访问的链表 list 的基础上实现支持同步访问的链表。

Exercise3 实现锁和条件变量

实现锁机制：

在 Lock 类中添加成员变量：

Semaphore *mutex; // 在构造函数中初始化为二元信号量，即一个互斥锁

Thread *owner; // 占用者指针，指向占有该锁的线程

成员函数的具体实现：

构造函数 Lock()：调用 Semaphore 的构造函数初始化 mutex，初始化 owner 为 NULL。

析构函数：调用 Semaphore 的析构函数。

Acquire()函数：因为我的实现使用的是 Semaphore 原语，已经关中断，所以不需要编写关中断的代码。调用 mutex->P()，并令 owner = currentThread，即占用者为当前线程。

Release()函数：首先判断占有该锁的线程是否为当前线程，因为只有对互斥锁加锁的线程才能解锁该互斥锁。然后调用 mutex->V()，并令 owner = NULL。

isHeldByCurrentThread()函数：如果当前线程占有该锁，则返回 true，否则返回 false。

总的来说，Lock 类就是对一个二元信号量的封装。

实现条件变量：

在 Condition 类中添加成员变量：

List *queue; //保存被该条件变量阻塞的线程。

成员函数的具体实现：

构造函数：新建一个 queue = List 链表。

析构函数：delete queue。

Wait()函数：wait 操作的功能是判断条件，如果不符，则挂起当前线程直到其他线程通过 signal 操作改变条件。首先关中断。然后判断互斥锁的占有者是否为当前线程。然后解锁，把当前线程添加到被该锁阻塞的队列中，然后调用 currentThread->Sleep()挂起当前线程，等待其他线程做出 Signal 操作。等到之后重新加锁。最后开中断。

Signal()函数：signal 的功能是改变条件变量，然后从阻塞队列中唤醒一个线程。首先关中断。然后判断互斥锁的占有者是否为当前线程。如果被该互斥锁阻塞的队列不为空的化，唤醒一个线程。最后开中断。

Broadcast()函数：大致与 Signal(0 函数相同，只需要把唤醒一个线程改为通过 while 循环唤醒所有线程即可。

测试：

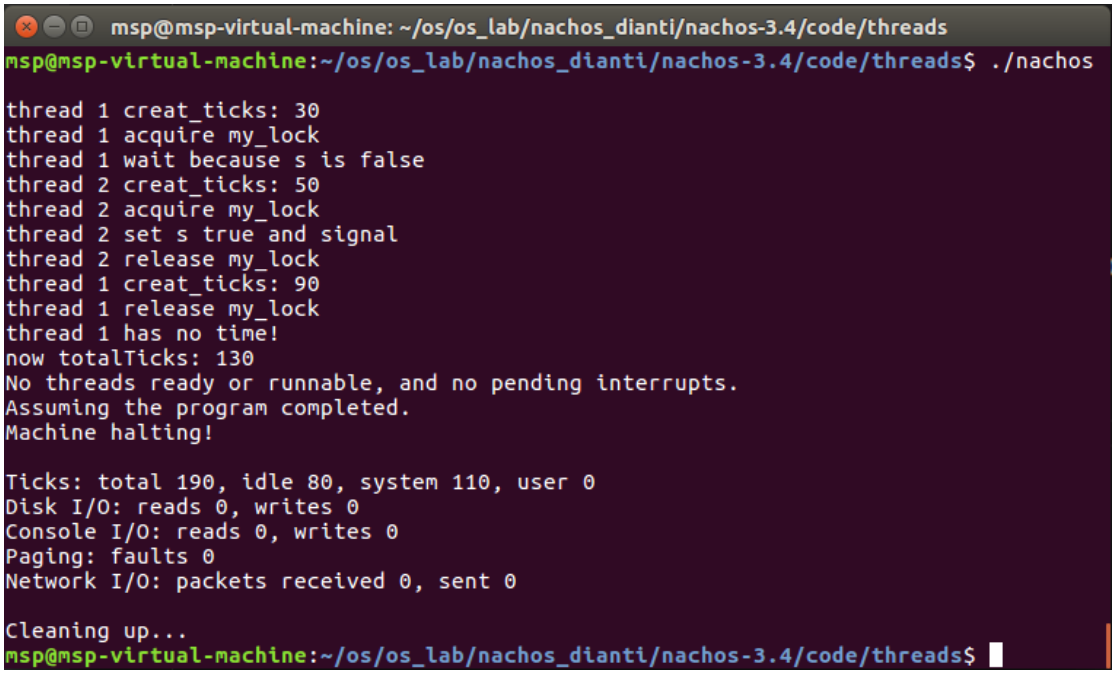
定义一个 bool 型的全局变量 s，互斥锁 my_lock，条件变量 my_condition。创建两个线程，第一个线程判断 s，如果为 true，则执行，如果为 false，则执行 my_lock 的 Wait 操作挂起等待。第二个线程设置 s 为 true，并执行 my_lock 的 Signal 操作。在两个线程中，对 my_condition 的操作用互斥锁 my_lock 保护起来。

测试线程：

```
void
SimpleThread5()
{
    printf("thread %d acquire my_lock\n",currentThread->get_thread_id());
    my_lock->Acquire();
    while(s != true)
    {
        printf("thread %d wait because s is false\n",currentThread->get_thread_id());
        my_condition->Wait(my_lock);
    }
    printf("thread %d release my_lock\n",currentThread->get_thread_id());
    my_lock->Release();
}

void
SimpleThread6()
{
    printf("thread %d acquire my_lock\n",currentThread->get_thread_id());
    my_lock->Acquire();
    s = true;
    printf("thread %d set s true and signal\n",currentThread->get_thread_id());
    my_condition->Signal(my_lock);
     printf("thread %d release my_lock\n",currentThread->get_thread_id());
    my_lock->Release();
}
```

运行结果：



可以看出：线程 1 判断 s 为 false 后，此时线程 1 的时间片并没有耗尽，但是挂起等待。切换到线程 2 后，线程 2 将 s 置为 true，并执行 Signal 操作，将线程 1 唤醒，添加到 readylist 队列中。然后执行线程 1。说明锁机制和条件变量实现成功。

Exercise4 实现同步互斥实例（直接在 threadtest 中编写代码）

生产者-消费者问题需要互斥的访问缓冲区。并且调度对缓冲区的访问：如果缓冲区是满的，那么生产者必须等待直到有一个槽位变为可用。与之相似，如果缓冲区是空的，那么消费者必须等待直到有一个项目变为可用。

利用信号量实现“生产者-消费者问题”：

定义缓冲区槽位数目为 3。

定义三个信号量：

Semaphore *mutex; // 一个二元信号量，也就是互斥锁，保护访问缓冲区的代码，提供对缓冲区的互斥访问。初始化为 1.

Semaphore *slots; // 记录空槽位的数目，初始化为 3。

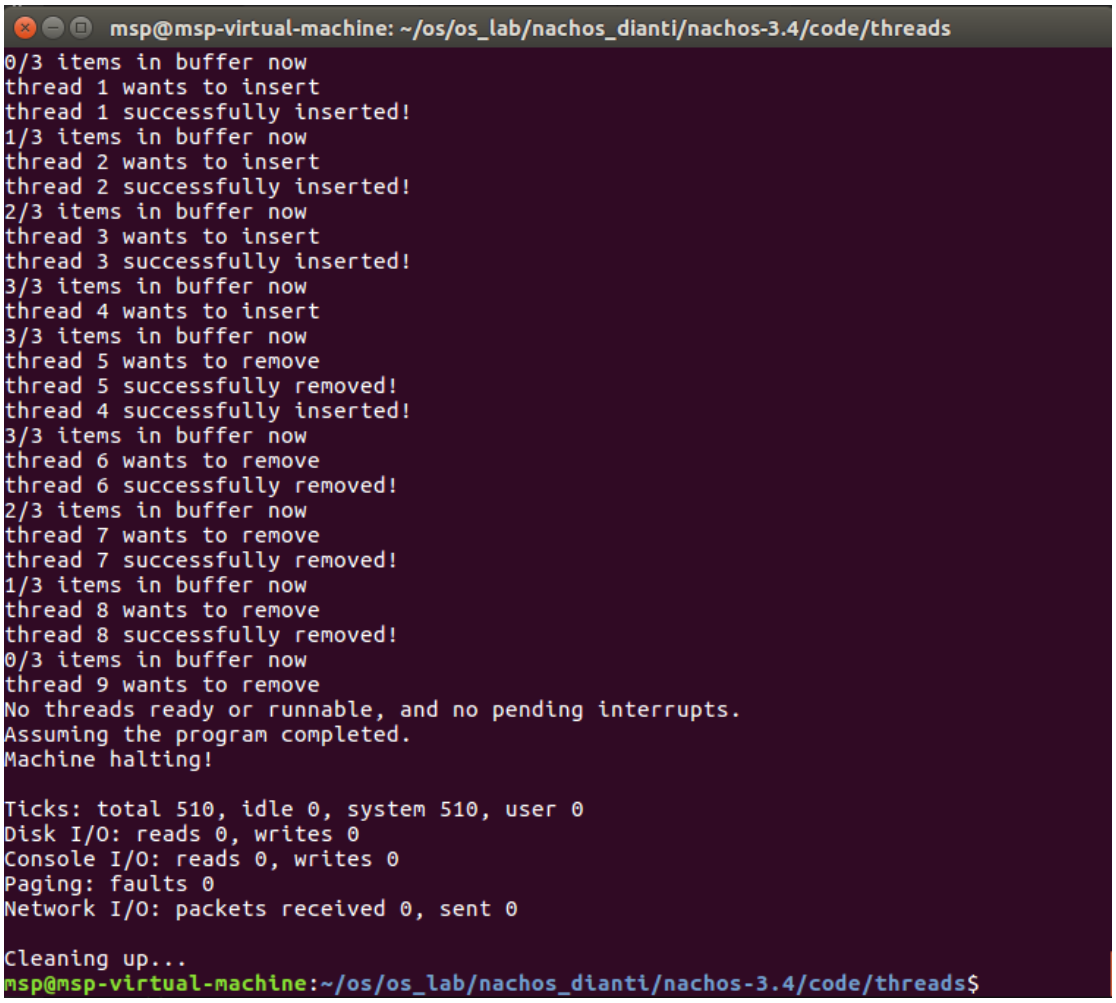
Semaphore *items; // 记录缓冲区中项目的数目，初始化为 0。

定义对缓冲区的两个操作：

插入项目的函数 insert。首先调用 slots 的 P 操作，判断是否还有空槽位，没有则挂起线程。然后调用 mutex 的 P 操作，对缓冲区的访问进行加锁。然后执行插入项目的操作。然后调用 mutex 的 V 操作解锁对缓冲区的访问。最后调用 items 的 V 操作更改缓冲区中项目数。

去除项目的函数 remove。首先调用 items 的 P 操作，判断缓冲区中是否有项目可以取出。然后调用 mutex 的 P 操作，对对缓冲区的访问加锁。然后执行取出项目的操作。然后调用 mutex 的 V 操作，解锁对缓冲区的访问。最后调用 slots 的 V 操作，更改缓冲区中空槽位的数目。

运行结果：



如图：连续三个 insert 操作，缓冲区满。第四个 insert 操作被阻塞，thread4 没有 insert 成功。然后 thread5 做出 remove 操作，slots->V()唤醒被 slots->P()阻塞的 thread4，thread4 成功 insert。然后连续三个线程做出 remove 操作，缓冲区空。线程 9 仍然想 remove 被 items->P()阻塞。

利用条件变量实现生产者-消费者问题：

定义一个互斥锁：Lock *my_lock; // 用于保护对条件变量的互斥访问，防止出现同步错误。

定义两个个条件变量：

Condition *slots; // 记录缓冲区是否有空槽位可以插入项目

Condition *items; // 记录缓冲区是否有项目可以取出

定义全局变量 item_num; // 记录缓冲区中的项目数目。

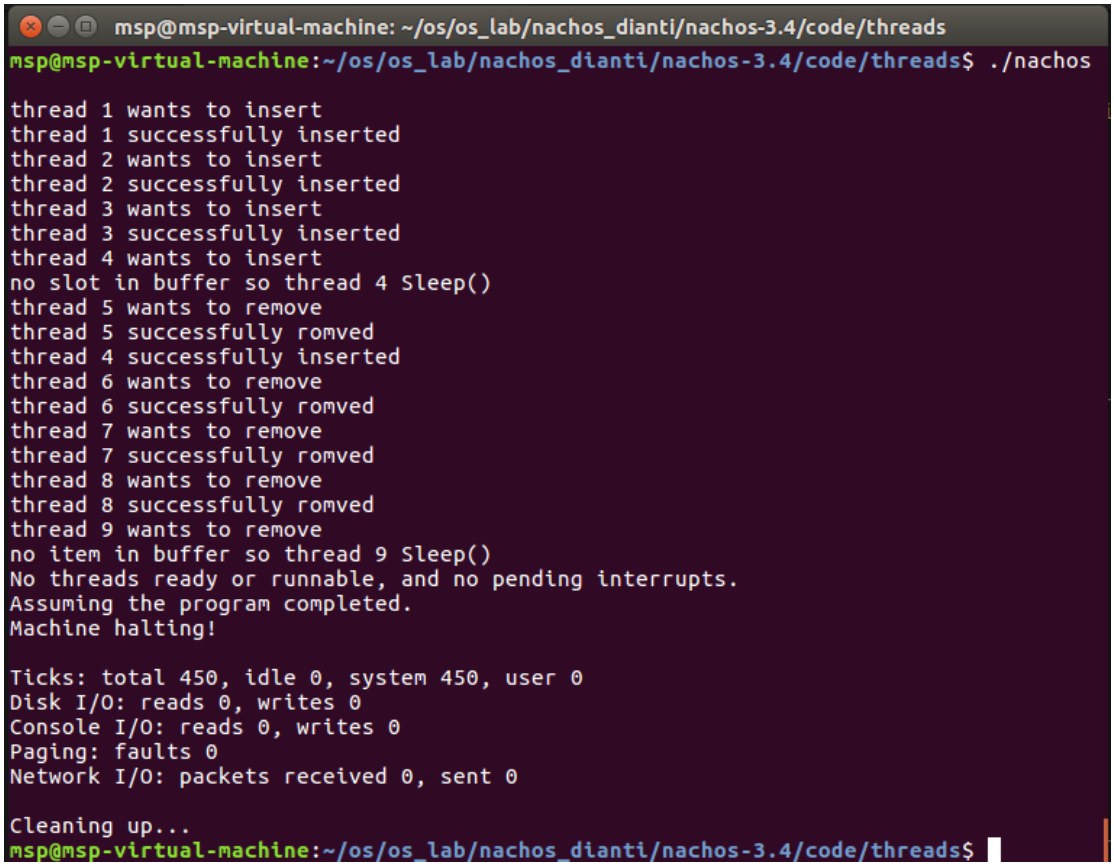
定义对缓冲区的两个操作：

插入操作 insert：首先调用 my_lock->Acquire 加锁。然后循环判断缓冲区中是否有空槽位可以插入，若没有，则调用 slots->Wait()，挂起当前线程，等待缓冲区中有空槽位的信号。若有空槽位，则直接插入，并调用 items->Signal()发送缓冲区中有项目的信号。最后解锁。

取出操作 remove：首先调用 my_lock->Acquire()加锁。然后循环判断缓冲区中是否有项目可以取出，若没有，则调用 items->Wait()，挂起当前线程，等待缓冲区中有项目的信号。若有项目可以取出，则直接取出，并调用 slots->Signal()发送缓冲

区中有空槽位的信号。最后解锁。

测试数据与信号量实现方式相同，运行结果：



如图：发现结果与用信号量实现方式相同，说明利用条件变量也实现成功。

Challenge 实现 barrier

利用条件变量实现 barrier，即当若干线程同时到达某一点，才可以继续执行。

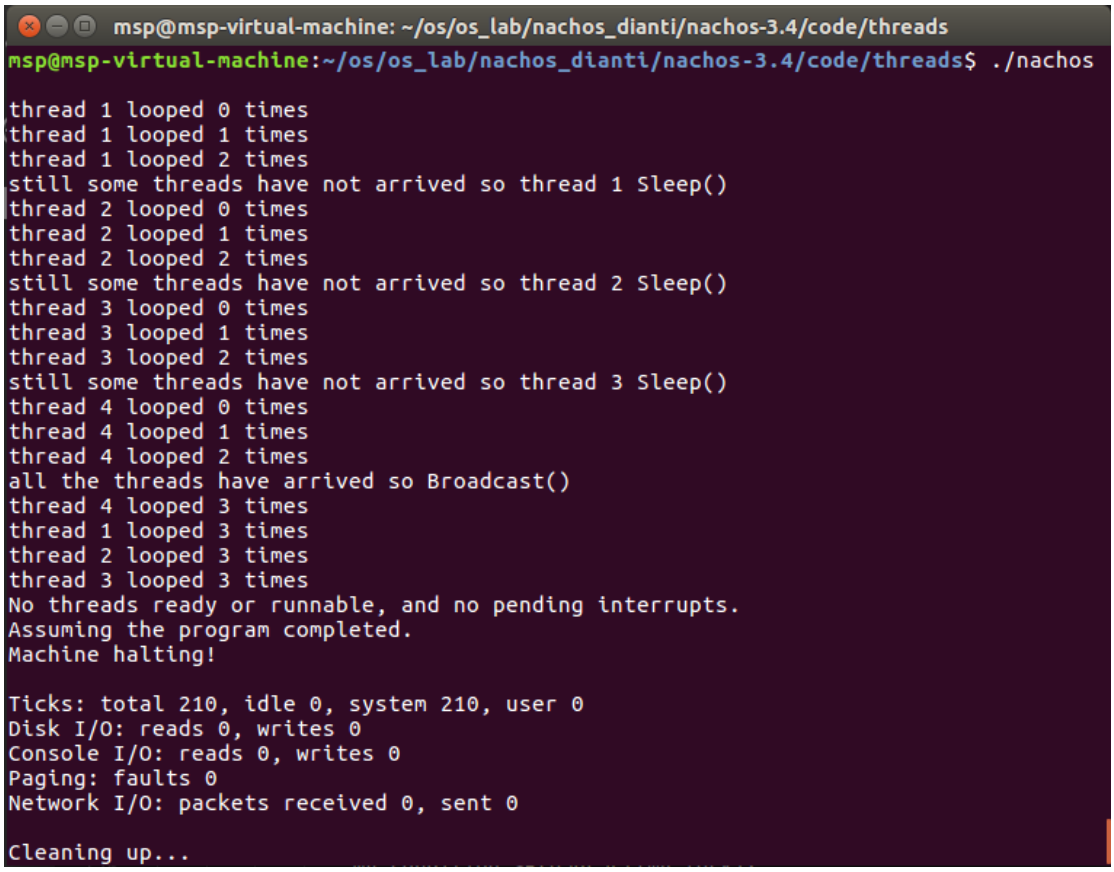
创建四个线程，每个线程循环 4 次，只有当这四个线程同时执行到第三次循环时，才可以继续执行，继续第四次执行。

定义一个互斥锁：Lock *my_lock;

定义一个条件变量：Condition *my_condition; // 记录是否所有线程达到第二次循环。

每个线程首先加锁，然后进入循环，如果是第三次循环，记录一下。并判断是否所有线程已经达到第三次循环，如果没有，则调用 my_condition->Wait()，挂起当前线程。否则唤醒所有被 my_condition 挂起的线程。最后解锁。

运行结果：



如图：结果符合预期，说明 barrier 实现成功。

内容三：遇到的困难以及解决方法

困难 1

条件变量 Condition 的实现。刚开始没有理解 Wait()的具体操作。后来仔细阅读了 Nachos 的注释，才搞清楚。

内容四：收获及感想

通过同步机制实习，我深入理解了 Linux 和 Nachos 实现进程/线程同步机制的方法。并自己着手实现了其中的一部分，锻炼了我的动手能力。

内容五：对课程的意见和建议

目前感觉课程的设置以及老师助教的讲解都很好，没有意见。

内容六：参考文献

[1] Andrew S. Tanenbaum 著．陈向群 马洪兵 译 .现代操作系统［M］. 北京：机械工业出版社，2011：47-95