实验七：同步互斥

计31 张正 2013011418

一、实验目的

• 熟悉ucore中的进程同步机制，了解操作系统为进程同步提供的底层支持；

• 在ucore中理解信号量（semaphore）机制的具体实现；

• 理解管程机制，在ucore内核中增加基于管程（monitor）的条件变量（condition variable）的支持；

• 了解经典进程同步问题，并能使用同步机制解决进程同步问题。

二、实验内容

实验六完成了用户进程的调度框架和具体的调度算法，可调度运行多个进程。如果多个进程需要协同操作或访问共享资源，则存在如何同步和有序竞争的问题。本次实验，主要是熟悉ucore的进程同步机制—信号量（semaphore）机制，以及基于信号量的哲学家就餐问题解决方案。然后掌握管程的概念和原理，并参考信号量机制，实现基于管程的条件变量机制和基于条件变量来解决哲学家就餐问题。

练习0：填写已有实验

本实验依赖实验1/2/3/4/5/6。请把你做的实验1/2/3/4/5/6的代码填入本实验中代码中有“LAB1”/“LAB2”/“LAB3”/“LAB4”/“LAB5”/“LAB6”的注释相应部分。并确保编译通过。注意：为了能够正确执行lab7的测试应用程序，可能需对已完成的实验1/2/3/4/5/6的代码进行进一步改进。

合并并更新时钟中断

```
ticks ++;
assert(current != NULL);
run_timer_list();
```

练习1: 理解内核级信号量的实现和基于内核级信号量的哲学家就餐问题（不需要编码）

完成练习0后，建议大家比较一下（可用kdiff3等文件比较软件）个人完成的lab6和练习0完成后的刚修改的lab7之间的区别，分析了解lab7采用信号量的执行过程。执行make grade，大部分测试用例应该通过。

请在实验报告中给出内核级信号量的设计描述，并说其大致执行流流程。

请在实验报告中给出给用户态进程/线程提供信号量机制的设计方案，并比较说明给内核级提供信号量机制的异同。

```
struct semaphore {
   int count;
   queueType queue;
};
```

```
void semWait(semaphore s)
{
    s.count--;
    if (s.count < 0) {
    /* place this process in s.queue */;
    /* block this process */;
    }
}
void semSignal(semaphore s)
{
    s.count++;
    if (s.count<= 0) {
    /* remove a process P from s.queue */;
    /* place process P on ready list */;
    }
}
```

基于上诉信号量实现可以认为，当多个（>1）进程可以进行互斥或同步合作时，一个进程会由于无法满足信号量设置的某条件而在某一位置停止，直到它接收到一个特定的信号（表明条件满足了）。为了发信号，需要使用一个称作信号量的特殊变量。为通过信号量s传送信号，信号量的V操作采用进程可执行原语semSignal(s)；为通过信号量s接收信号，信号量的P操作采用进程可执行原语semWait(s)；如果相应的信号仍然没有发送，则进程被阻塞或睡眠，直到发送完为止。

ucore中信号量参照上述原理描述，建立在开关中断机制和wait queue的基础上进行了具体实现。信号量的数据结构定义如下：

```
typedef struct {
    int value;                      //信号量的当前值
    wait_queue_t wait_queue;    //信号量对应的等待队列
} semaphore_t;
```

在ucore中最重要的信号量操作是P操作函数down(semaphore_t *sem)和V操作函数 up(semaphore_t *sem)。但这两个函数的具体实现是__down(semaphore_t *sem, uint32_t wait_state) 函数和__up(semaphore_t *sem, uint32_t wait_state)函数，二者的具体实现描述如下：

① __down(semaphore_t *sem, uint32_t wait_state, timer_t *timer)：具体实现信号量的P操作，首先关掉中断，然后判断当前信号量的value是否大于0。如果是>0，则表明可以获得信号量，故让value减一，并打开中断返回即可；如果不是>0，则表明无法获得信号量，故需要将当前的进程加入到等待队列中，并打开中断，然后运行调度器选择另外一个进程执行。如果被V操作唤醒，则把自身关联的wait从等待队列中删除（此过程需要先关中断，完成后开中断）。

②__up(semaphore_t *sem, uint32_t wait_state)：具体实现信号量的V操作，首先关中断，如果信号量对应的wait queue中没有进程在等待，直接把信号量的value加一，然后开中断返回；如果有进程在等待且进程等待的原因是semophore设置的，则调用wakeup_wait函数将waitqueue中等待的第一个wait删除，且把此wait关联的进程唤醒，最后开中断返回。

我们可以看出信号量的计数器value具有有如下性质：

value>0，表示共享资源的空闲数

vlaue<0，表示该信号量的等待队列里的进程数

value=0，表示等待队列为空

练习2: 完成内核级条件变量和基于内核级条件变量的哲学家就餐问题（需要编码）

首先掌握管程机制，然后基于信号量实现完成条件变量实现，然后用管程机制实现哲学家就餐问题的解决方案

请在实验报告中给出内核级条件变量的设计描述，并说其大致执行流流程。

请在实验报告中给出给用户态进程/线程提供条件变量机制的设计方案，并比较说明给内核级提供条件变量机制的异同。

1、原理：

管程由四部分组成：

①管程内部的共享变量；

②管程内部的条件变量；

③管程内部并发执行的进程；

④对局部于管程内部的共享数据设置初始值的语句。

管程相当于一个隔离区，它把共享变量和对它进行操作的若干个过程围了起来，所有进程要访问临界资源时，都必须经过管程才能进入，而管程每次只允许一个进程进入管程，从而需要确保进程之间互斥。

但在管程中仅仅有互斥操作是不够用的。进程可能需要等待某个条件C为真才能继续执行。如果采用忙等(busy waiting)方式：

while not(C) do {}

在单处理器情况下，将会导致所有其它进程都无法进入临界区使得该条件C为真，该管程的执行将会发生死锁。为此，可引入条件变量（Condition Variables，简称CV）。一个条件变量CV可理解为一个进程的等待队列，队列中的进程正等待某个条件C变为真。每个条件变量关联着一个断言 "断言 (程序)")Pc。当一个进程等待一个条件变量，该进程不算作占用了该管程，因而其它进程可以进入该管程执行，改变管程的状态，通知条件变量CV其关联的断言Pc在当前状态下为真。因此对条件变量CV有两种主要操作：

①wait_cv： 被一个进程调用，以等待断言Pc被满足后该进程可恢复执行. 进程挂在该条件变量上等待时，不被认为是占用了管程。

②signal_cv：被一个进程调用，以指出断言Pc现在为真，从而可以唤醒等待断言Pc被满足的进程继续执行。

有了互斥和信号量支持的管程就可用用了解决各种同步互斥问题。

ucore中的管程机制是基于信号量和条件变量来实现的。ucore中的管程的数据结构monitor_t定义如下：

```
typedef struct monitor{
    semaphore_t mutex;      // the mutex lock for going into the routines in monitor, should be initialized to 1
    semaphore_t next;    // the next semaphore is used to down the signaling proc itself, and the other OR wakeuped
    //waiting proc should wake up the sleeped signaling proc.
    int next_count;         // the number of of sleeped signaling proc
    condvar_t *cv;          // the condvars in monitor
} monitor_t;
```

管程中的成员变量mutex是一个二值信号量，是实现每次只允许一个进程进入管程的关键元素，确保了互斥访问性质。管程中的条件变量cv通过执行wait_cv，会使得等待某个条件C为真的进程能够离开管程并睡眠，且让其他进程进入管程继续执行；而进入管程的某进程设置条件C为真并执行signal_cv时，能够让等待某个条件C为真的睡眠进程被唤醒，从而继续进入管程中执行。管程中的成员变量信号量next和整形变量next_count是配合进程对条件变量cv的操作而设置的，这是由于发出signal_cv的进程A会唤醒睡眠进程B，进程B执行会导致进程A睡眠，直到进程B离开管程，进程A才能继续执行，这个同步过程是通过信号量next完成的；而next_count表示了由于发出singal_cv而睡眠的进程个数。

管程中的条件变量的数据结构condvar_t定义如下：

```
typedef struct condvar{
    semaphore_t sem; // the sem semaphore is used to down the waiting proc, and the signaling proc should up the waiting proc
    int count;       // the number of waiters on condvar
    monitor_t * owner; // the owner(monitor) of this condvar
} condvar_t;
```

条件变量的定义中也包含了一系列的成员变量，信号量sem用于让发出wait_cv操作的等待某个条件C为真的进程睡眠，而让发出signal_cv操作的进程通过这个sem来唤醒睡眠的进程。count表示等在这个条件变量上的睡眠进程的个数。owner表示此条件变量的宿主是哪个管程。

基于管程的条件变量的实现

```
void
cond_signal (condvar_t *cvp) {
```

```
        //LAB7 EXERCISE1: 2013011418
        cprintf("cond_signal begin: cvp %x, cvp->count %d, cvp->owner->next_count
%d\n", cvp, cvp->count, cvp->owner->next_count);
        /*
         *      cond_signal(cv) {
         *          if(cv.count>0) {
         *             mt.next_count ++;
         *             signal(cv.sem);
         *             wait(mt.next);
         *             mt.next_count--;
         *          }
         *       }
         */
          if(cvp->count>0) {//当前存在执行cond_wait而睡眠的进程
            cvp->owner->next_count ++;//睡眠的进程总个数加一
            up(&(cvp->sem));//唤醒等待在cv.sem上睡眠的进程
            down(&(cvp->owner->next));//自己需要睡眠
            cvp->owner->next_count --;//睡醒后等待此条件的睡眠进程个数减一
          }
        cprintf("cond_signal end: cvp %x, cvp->count %d, cvp->owner->next_count
%d\n", cvp, cvp->count, cvp->owner->next_count);
      }
      void
      cond_wait (condvar_t *cvp) {
         //LAB7 EXERCISE1: 2013011418
         cprintf("cond_wait begin:  cvp %x, cvp->count %d, cvp->owner->next_count
%d\n", cvp, cvp->count, cvp->owner->next_count);
         /*
          *         cv.count ++;
          *         if(mt.next_count>0)
          *            signal(mt.next)
          *         else
          *            signal(mt.mutex);
          *         wait(cv.sem);
          *         cv.count --;
          */
           cvp->count++;//需要睡眠的进程个数加一
```

```
        if(cvp->owner->next_count > 0)
          up(&(cvp->owner->next));//唤醒进程链表中的下一个进程
        else
          up(&(cvp->owner->mutex));//唤醒睡在monitor.mutex上的进程
        down(&(cvp->sem));//将此进程等待
        cvp->count --;//睡醒后等待此条件的睡眠进程个数减一
       cprintf("cond_wait end:  cvp %x, cvp->count %d, cvp->owner->next_count
%d\n", cvp, cvp->count, cvp->owner->next_count);
    }
```

基于管程的哲学家就餐问题

```
    void phi_take_forks_condvar(int i) {
        down(&(mtp->mutex));//进入临界区
    //--------into routine in monitor--------------
        // LAB7 EXERCISE1: 2013011418
        // I am hungry
        // try to get fork
         // I am hungry
         state_condvar[i]=HUNGRY; //记录下哲学家i饥饿的事实
         // try to get fork
         phi_test_condvar(i);
         while (state_condvar[i] != EATING) {
             cprintf("phi_take_forks_condvar: %d didn't get fork and will wait\n",i);
             cond_wait(&mtp->cv[i]);//如果得不到叉子就阻塞
         }
    //--------leave routine in monitor--------------
         if(mtp->next_count>0)//如果阻塞则唤醒
           up(&(mtp->next));
         else
           up(&(mtp->mutex));//离开临界区
    }

    void phi_put_forks_condvar(int i) {
        down(&(mtp->mutex));//进入临界区

    //--------into routine in monitor--------------
        // LAB7 EXERCISE1: 2013011418
```

```
        // I ate over
        // test left and right neighbors
         // I ate over
         state_condvar[i]=THINKING;//哲学家进餐结束
         // test left and right neighbors
         phi_test_condvar(LEFT);//看一下左邻居现在是否能进餐
         phi_test_condvar(RIGHT);//看一下右邻居现在是否能进餐
    //--------leave routine in monitor---------------
        if(mtp->next_count>0)<span style="line-height: 25.6000003814697px; font-family: 'Open Sans', 'Clear Sans', 'Helvetica Neue', Helvetica, Arial, sans-serif;">//如果存在阻塞则唤醒</span>
          up(&(mtp->next));
        else
          up(&(mtp->mutex));//离开临界区
    }
```