**实验4 同步互斥**

**仔细阅读实验文档lab7同步互斥，完成以下练习（不做实验文档中的题目）。扩展练习选做，有能力者完成。**

**练习1: 了解信号量和管程的实现机制**

1. **同步互斥的底层支持是如何实现的？**

（1）开关中断

内核实现这一功能的一个底层支撑机制就是中断开关控制和test\_and\_set相关原子操作机器指令。kern/sync.c中实现的开关中断的控制函数local\_intr\_save(x)和local\_intr\_restore(x)，它们是基于kern/driver文件下的intr\_enable()、intr\_disable()函数实现的。最终的cli和sti是x86的机器指令，最终实现了关中断和开中断，即设置了eflags寄存器中与中断相关的位。通过关闭中断，可以防止对当前执行的控制流被其他中断事件处理所打断。既然不能中断，那也就意味着在内核运行的当前进程无法被打断或被从新调度，即实现了对临界区的互斥操作。

（2）等待队列

内核实现这一功能的另一个底层支撑机制就是等待队列（wait queue），等待队列和每一个事件（睡眠结束、时钟到达、任务完成、资源可用等）联系起来。需要等待事件的进程在转入休眠状态后插入到等待队列中。当事件发生之后，内核遍历相应等待队列，唤醒休眠的用户进程或内核线程，并设置其状态为就绪状态（runnable state），并将该进程从等待队列中清除。ucore在kern/sync/{ wait.h, wait.c }中实现了wait结构和wait queue结构以及相关函数）。

1. **对比原理课上学到的信号量和p，v操作，说明Ucore中信号量机制的实现。**

对照信号量的原理性描述和具体实现，可以发现二者在流程上基本一致，只是具体实现采用了关中断的方式保证了对共享资源的互斥访问，通过等待队列让无法获得信号量的进程睡眠等待。

1. **Ucore中的信号量是基于信号量和条件变量实现的，请说明其中的数据结构和函数方法的设计。**

（1）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;

（2）函数方法设计：

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)。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，

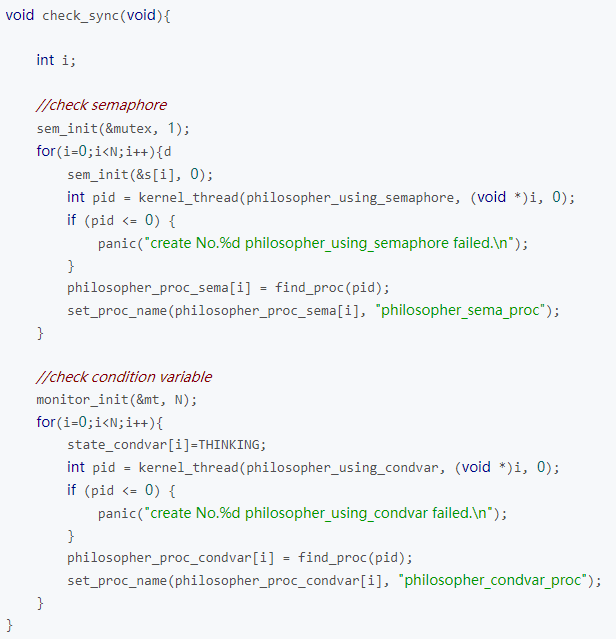
二者的具体实现描述如下：

● \_\_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1，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）。

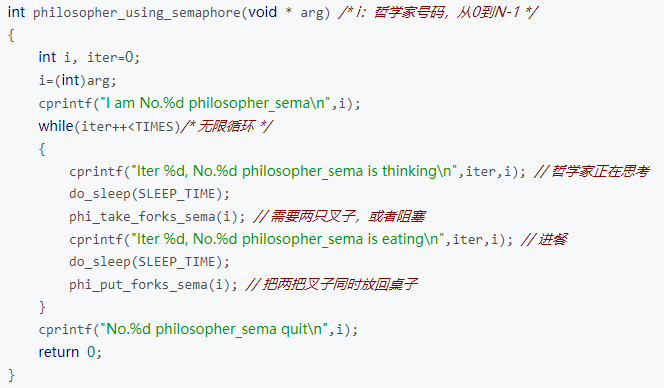
● \_\_up(semaphore\_t \*sem, uint32\_t wait\_state)：具体实现信号量的V操作，首先关中断，如果信号量对应的wait queue中没有进程在等待，直接把信号量的value加1，然后开中断返回；如果有进程在等待且进程等待的原因是semophore设置的，则调用wakeup\_wait函数将waitqueue中等待的第一个wait删除，且把此wait关联的进程唤醒，最后开中断返回。

**练习2: 了解基于信号量和管程的哲学家就餐问题**

1. **说明ucore中基于信号量的哲学家就餐问题的实现机制。**



该函数分为了两个部分，第一部分是实现基于信号量的哲学家问题,第二部分是实现基于管程的哲学家问题。首先实现初始化了一个互斥信号量，然后创建了对应5个哲学家行为的5个信号量，并创建5个内核线程代表5个哲学家，每个内核线程完成了基于信号量的哲学家吃饭睡觉思考行为实现。现在我们继续跟进philosopher\_using\_semaphore函数观察它的具体实现。



看到核心就是phi\_take\_forks\_sema和phi\_put\_forks\_sema两个函数，具体的函数注释如下：



而这里到了信号量的核心部分，就是上述代码中的up和down函数就分别调用了\_\_up函数和\_\_down函数，而这两个函数分别对应着信号量的V，P操作。   
先看\_\_up函数，它实现了信号量的的V操作:



首先通过local\_intr\_save函数关闭中断，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，然后通过local\_intr\_restore函数开中断返回。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wait\_queue中等待的第一个wait删除，且把此wait关联的进程唤醒，最后通过local\_intr\_restore函数开中断返回。

再来看看\_\_down函数，它实现了信号量的P操作:

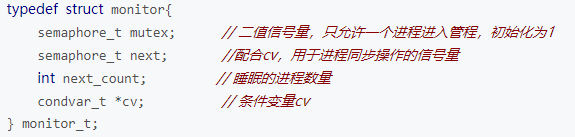


首先关掉中断，然后判断当前信号量的value是否大于0。如果是大于0，则表明可以获得信号量，故让value减1，并打开中断返回即可；如果小于0，则表明无法获得信号量，故需要将当前的进程加入到等待队列中，并打开中断，然后运行调度器选择另外一个进程执行。如果被V操作唤醒，则把自身关联的wait从等待队列中删除（此过程需要先关中断，完成后开中断）

至此，基于信号量的哲学家问题的解决就分析完毕了。

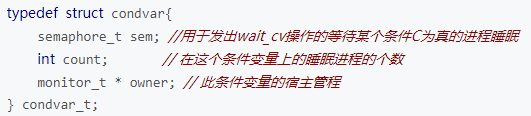
1. **说明ucore中基于管程的哲学家就餐问题的实现机制。**

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

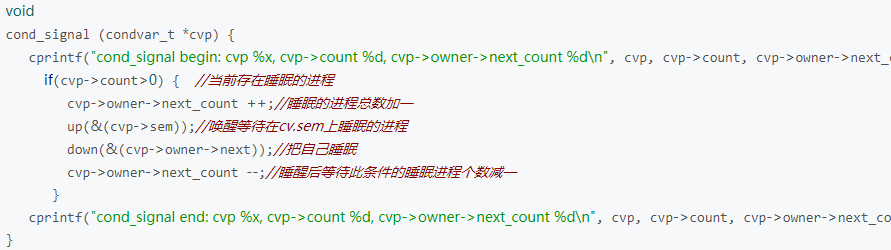


管程中的条件变量cv通过执行wait\_cv，会使得等待某个条件C为真的进程能够离开管程并睡眠，且让其他进程进入管程继续执行；而进入管程的某进程设置条件C为真并执行signal\_cv时，能够让等待某个条件C为真的睡眠进程被唤醒，从而继续进入管程中执行。发出signal\_cv的进程A会唤醒睡眠进程B，进程B执行会导致进程A睡眠，直到进程B离开管程，进程A才能继续执行，这个同步过程是通过信号量next完成的；而next\_count表示了由于发出singal\_cv而睡眠的进程个数。

条件变量condvar\_t的数据结构如下：

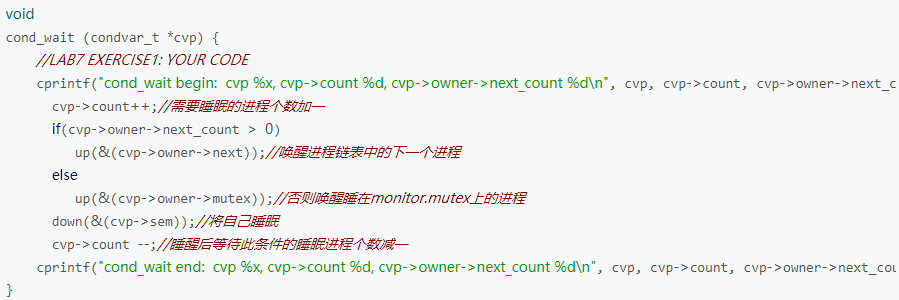


分析完数据结构之后，我们开始分析管程的实现。   
ucore设计实现了条件变量wait\_cv操作和signal\_cv操作对应的具体函数，即cond\_wait函数和cond\_signal函数，此外还有cond\_init初始化函数。   
先看看cond\_signal函数，实现如下：



首先进程B判断cv.count，如果不大于0，则表示当前没有睡眠的进程，因此就没有被唤醒的对象了，直接函数返回即可；如果大于0，这表示当前有睡眠的进程A，因此需要唤醒等待在cv.sem上睡眠的进程A。由于只允许一个进程在管程中执行，所以一旦进程B唤醒了别人（进程A），那么自己就需要睡眠。故让monitor.next\_count加一，且让自己（进程B）睡在信号量monitor.next上。如果睡醒了，这让monitor.next\_count减1。

同样，再来看看cond\_wait函数，实现如下：



可以看出如果进程A执行了cond\_wait函数，表示此进程等待某个条件C不为真，需要睡眠。因此表示等待此条件的睡眠进程个数cv.count要加1。接下来会出现两种情况。

情况一：如果monitor.next\_count如果大于0，表示有大于等于1个进程执行cond\_signal函数且睡着了，就睡在了monitor.next信号量上。假定这些进程形成S进程链表。因此需要唤醒S进程链表中的一个进程B。然后进程A睡在cv.sem上，如果睡醒了，则让cv.count减一，表示等待此条件的睡眠进程个数少了一个，可继续执行。

情况二：如果monitor.next\_count如果小于等于0，表示目前没有进程执行cond\_signal函数且睡着了，那需要唤醒的是由于互斥条件限制而无法进入管程的进程，所以要唤醒睡在monitor.mutex上的进程。然后进程A睡在cv.sem上，如果睡醒了，则让cv.count减1，表示等待此条件的睡眠进程个数少了一个，可继续执行了！

