仔细阅读实验文档lab7同步互斥，完成以下练习。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？
2. 开关中断 根据操作系统原理的知识，我们知道如果没有在硬件级保证读内存-修改值-写回内存的原子性，我们只能通过复杂的软件来 实现同步互斥操作。但由于有开关中断和test\_and\_set\_bit等原子操作机器指令的存在，使得我们在实现同步互斥原语上可以 大大简化。在atomic.c文件中实现的test\_and\_set\_bit等原子操作。 在ucore中提供的底层机制包括中断开关控制和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。具体 调用关系为： 关中断：local\_intr\_save --> \_\_intr\_save --> intr\_disable --> cli 开中断：local\_intr\_restore--> \_\_intr\_restore --> intr\_enable --> sti 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。所以在单处理器情况下，可以通过开关中断实现对临界区的互斥保护，需要互 斥的临界区代码的一般写法为： local\_intr\_save(intr\_flag); { 临界区代码 }local\_intr\_restore(intr\_flag); …… 由于目前ucore只实现了对单处理器的支持，所以通过这种方式，就可简单地支撑互斥操作了。在多处理器情况下，这种方法 是无法实现互斥的，因为屏蔽了一个CPU的中断，只能阻止本CPU上的进程不会被中断或调度，并不意味着其他CPU上执行 的进程不能执行临界区的代码。所以，开关中断只对单处理器下的互斥操作起作用。在本实验中，开关中断机制是实现信号 量等高层同步互斥原语的底层支撑基础之一。
3. 等待队列 到目前为止，我们的实验中，用户进程或内核线程还没有睡眠的支持机制。在课程中提到用户进程或内核线程可以转入休眠 状态以等待某个特定事件，当该事件发生时这些进程能够被再次唤醒。内核实现这一功能的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），这是实现ucore中的信号量机制和条件变量机制的基础，进入wait queue的进程会被设为 睡眠状态，直到他们被唤醒。 typedef struct { struct proc\_struct \*proc; //等待进程的指针 uint32\_t wakeup\_flags; //进程被放入等待队列的原因标记 wait\_queue\_t \*wait\_queue; //指向此wait结构所属于的wait\_queue list\_entry\_t wait\_link; //用来组织wait\_queue中wait节点的连接 } wait\_t; typedef struct { list\_entry\_t wait\_head; //wait\_queue的队头 } wait\_queue\_t; le2wait(le, member) //实现wait\_t中成员的指针向wait\_t 指针的转化 与wait和wait queue相关的函数主要分为两层，底层函数是对wait queue的初始化、插入、删除和查找操作，相关函数如下：

void wait\_init(wait\_t \*wait, struct proc\_struct \*proc); //初始化wait结构 bool wait\_in\_queue(wait\_t \*wait); //wait是否在wait queue中 void wait\_queue\_init(wait\_queue\_t \*queue); //初始化wait\_queue结构 void wait\_queue\_add(wait\_queue\_t \*queue, wait\_t \*wait); //把wait前插到wait queue中 void wait\_queue\_del(wait\_queue\_t \*queue, wait\_t \*wait); //从wait queue中删除wait wait\_t \*wait\_queue\_next(wait\_queue\_t \*queue, wait\_t \*wait);//取得wait的后一个链接指针 wait\_t \*wait\_queue\_prev(wait\_queue\_t \*queue, wait\_t \*wait);//取得wait的前一个链接指针 wait\_t \*wait\_queue\_first(wait\_queue\_t \*queue); //取得wait queue的第一个wait wait\_t \*wait\_queue\_last(wait\_queue\_t \*queue); //取得wait queue的最后一个wait bool wait\_queue\_empty(wait\_queue\_t \*queue); //wait queue是否为空 高层函数基于底层函数实现了让进程进入等待队列，以及从等待队列中唤醒进程，相关函数如下： //让wait与进程关联，且让当前进程关联的wait进入等待队列queue，当前进程睡眠 void wait\_current\_set(wait\_queue\_t \*queue, wait\_t \*wait, uint32\_t wait\_state); //把与当前进程关联的wait从等待队列queue中删除 wait\_current\_del(queue, wait); //唤醒与wait关联的进程 void wakeup\_wait(wait\_queue\_t \*queue, wait\_t \*wait, uint32\_t wakeup\_flags, bool del); //唤醒等待队列上挂着的第一个wait所关联的进程 void wakeup\_first(wait\_queue\_t \*queue, uint32\_t wakeup\_flags, bool del); //唤醒等待队列上所有的等待的进程 void wakeup\_queue(wait\_queue\_t \*queue, uint32\_t wakeup\_flags, bool del);

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。

信号量是一种同步互斥机制的实现，普遍存在于现在的各种操作系统内核里。相对于spinlock 的应用对象，信号量的应用对 象是在临界区中运行的时间较长的进程。等待信号量的进程需要睡眠来减少占用 CPU 的开销。

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。

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。

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。另外，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

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。

\_\_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）

static \_\_noinline uint32\_t \_\_down(semaphore\_t \*sem, uint32\_t wait\_state) {

bool intr\_flag;

local\_intr\_save(intr\_flag);

if (sem->value > 0) {

sem->value --;

local\_intr\_restore(intr\_flag);

return 0;

}

wait\_t \_\_wait, \*wait = &\_\_wait;

wait\_current\_set(&(sem->wait\_queue), wait, wait\_state);

local\_intr\_restore(intr\_flag);

schedule();

local\_intr\_save(intr\_flag);

wait\_current\_del(&(sem->wait\_queue), wait);

local\_intr\_restore(intr\_flag);

if (wait->wakeup\_flags != wait\_state) {

return wait->wakeup\_flags;

}

return 0;

}

\_\_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。

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。另外，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

static \_\_noinline void \_\_up(semaphore\_t \*sem, uint32\_t wait\_state) {

bool intr\_flag;

local\_intr\_save(intr\_flag);

{

wait\_t \*wait;

if ((wait = wait\_queue\_first(&(sem->wait\_queue))) == NULL) {

sem->value ++;

}

else {

wakeup\_wait(&(sem->wait\_queue), wait, wait\_state, 1);

}

}

local\_intr\_restore(intr\_flag);

}

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。

①基于信号量实现完成条件变量实现

void cond\_signal (condvar\_t \*cvp) {

//LAB7 EXERCISE1: YOUR CODE

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: YOUR CODE

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);

}

1. 说明ucore中基于管程的哲学家就餐问题的实现机制。

用管程机制实现哲学家就餐问题的解决方案

void phi\_take\_forks\_condvar(int i) {

down(&(mtp->mutex));//进入临界区

// LAB7 EXERCISE1: YOUR CODE

// 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

}

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

// LAB7 EXERCISE1: YOUR CODE

// 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

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

}