实验5 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？

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。

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);

2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。

**信号量机制即利用pv操作来对信号量进行处理。**

**信号量（*semaphore*）的数据结构为一个值和一个指针，指针指向等待该信号量的下一个进程。信号量的值与相应资源的使用情况有关。**

**当它的值大于*0*时，表示当前可用资源的数量；**

**当它的值小于*0*时，其绝对值表示等待使用该资源的进程个数。**

**注意，信号量的值仅能由*PV*操作来改变。**

**一般来说，信号量*S³0*时，*S*表示可用资源的数量。执行一次*P*操作意味着请求分配一个单位资源，因此*S*的值减*1*；当*S<0*时，表示已经没有可用资源，请求者必须等待别的进程释放该类资源，它才能运行下去。而执行一个*V*操作意味着释放一个单位资源，因此*S*的值加*1*；若*S£0*，表示有某些进程正在等待该资源，因此要唤醒一个等待状态的进程，使之运行下去。**

**p操作（wait）：申请一个单位资源，进程进入**

**p操作（wait）：申请一个单位资源，进程进入**

**v操作（signal）：释放一个单位资源，进程出来**

***PV操作的含义*：*PV*操作由*P*操作原语和*V*操作原语组成（原语是不可中断的过程），对信号量进行操作，具体定义如下： *P*（*S*）：①将信号量*S*的值减*1*，即*S=S-1*；②如果*S<=0*，则该进程继续执行；否则该进程置为等待状态，排入等待队列。 *V*（*S*）：①将信号量*S*的值加*1*，即*S=S+1*；②如果*S>0*，则该进程继续执行；否则释放队列中第一个等待信号量的进程。**

**PV操作的意义：我们用信号量及*PV*操作来实现进程的同步和互斥。*PV*操作属于进程的低级通信。**

**使用*PV*操作实现进程互斥时应该注意的是：（*1*）每个程序中用户实现互斥的*P*、*V*操作必须成对出现，先做*P*操作，进临界区，后做*V*操作，出临界区。若有多个分支，要认真检查其成对性。（*2*）*P*、*V*操作应分别紧靠临界区的头尾部，临界区的代码应尽可能短，不能有死循环。  
    （*3*）互斥信号量的初值一般为*1*。**

3Ucore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。

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。

内核级信号量的实现：

· 实现了内核级信号量机制的函数均定义在sem.c中，因此不妨对这些函数进行分析：

sem\_init: 对信号量进行初始化的函数，根据在原理课上学习到的内容，信号量包括了等待队列和一个整型数值变量，该函数只需要将该变量设置为指定的初始值，并且将等待队列初始化即可；

\_\_up: 对应到了原理课中提及到的V操作，表示释放了一个该信号量对应的资源，如果有等待在了这个信号量上的进程，则将其唤醒执行；结合函数的具体实现可以看到其采用了禁用中断的方式来保证操作的原子性，函数中操作的具体流程为：

查询等待队列是否为空，如果是空的话，给整型变量加1；

如果等待队列非空，取出其中的一个进程唤醒；

\_\_down: 同样对应到了原理课中提及的P操作，表示请求一个该信号量对应的资源，同样采用了禁用中断的方式来保证原子性，具体流程为：

查询整型变量来了解是否存在多余的可分配的资源，是的话取出资源（整型变量减1），之后当前进程便可以正常进行；

如果没有可用的资源，整型变量不是正数，当前进程的资源需求得不到满足，因此将其状态改为SLEEPING态，然后将其挂到对应信号量的等待队列中，调用schedule函数来让出CPU，在资源得到满足，重新被唤醒之后，将自身从等待队列上删除掉；

up, down：对\_\_up, \_\_down函数的简单封装；

try\_down: 不进入等待队列的P操作，即时是获取资源失败也不会堵塞当前进程；

2说明ucore中基于管程的哲学家就餐问题的实现机制。

cond\_signal: 将指定条件变量上等待队列中的一个线程进行唤醒，并且将控制权转交给这个进程；具体执行流程为：

判断当前的条件变量的等待队列上是否有正在等待的进程，如果没有则不需要进行任何操作；

如果由正在等待的进程，则将其中的一个唤醒，这里的等待队列是使用了一个信号量来进行实现的，由于信号量中已经包括了对等待队列的操作，因此要进行唤醒只需要对信号量执行up操作即可；

接下来当前进程为了将控制权转交给被唤醒的进程，将自己等待到了这个条件变量所述的管程的next信号量上，这样的话就可以切换到被唤醒的进程了；由于next信号量的实现，就带来了两个困惑：

等待在next信号量上的进程是否能够被唤醒？由于每一个next信号量上等待的进程的产生必定是因为存在了某个它需要唤醒的进程，而这个进程在结束cond\_wait函数之后返回到管程的函数，还会检查next信号量上是否存在等待着的进程，有的话将其唤醒，因此每一个next信号量上等待的进程最终必定会被唤醒；

在等待在next信号量上的时候，管程的mutex锁并没有被释放，是否可能存在该永远都被释放不了的情况？不会的。根据前一个问题得知所有next信号量上的等待进程一定会被唤醒，那么最后一个被唤醒的next进程就会将锁释放掉；

最终的具体代码实现如下所示：

if (cvp->count > 0) { // 判断条件变量的等待队列是否为空

cvp->owner->next\_count ++; // 修改next变量上等待进程计数，跟下一个语句不能交换位置，为了得到互斥访问的效果，关键在于访问共享变量的时候，管程中是否只有一个进程处于RUNNABLE的状态

up(&cvp->sem); // 唤醒等待队列中的某一个进程

down(&cvp->owner->next); // 把自己等待在next条件变量上

cvp->owner->next\_count --; // 当前进程被唤醒，恢复next上的等待进程计数}

接下来，当前进程被从next信号量上被唤醒的时候，首先将next count减一，然后离开cond\_signal函数，回到管程中的函数，检查是否应该释放管程的锁（取决于现在是否还有next信号量上等待的进程，有的话将其唤醒，完成其在函数中的操作，并且将释放锁的操作延迟给这个进程来进行），根据上述描述，我们可以知道在管程中能够运行的进程之间不会有互相有意料外的打断的过程（由于进程的切换时机都是固定好的，由当前的进程来唤醒另外某一个进程），因此实现了对共享变量访问的互斥性；

cond\_wait: 该函数的功能为将当前进程等待在指定信号量上，其操作过程为将等待队列的计数加1，然后释放管程的锁或者唤醒一个next上的进程来释放锁（否则会造成管程被锁死无法继续访问，同时这个操作不能和前面的等待队列计数加1的操作互换顺序，要不不能保证共享变量访问的互斥性），然后把自己等在条件变量的等待队列上，直到有signal信号将其唤醒，正常退出函数；

最终具体的代码实现如下所示：

cvp->count ++; // 修改等待在条件变量的等待队列上的进程计数if (cvp->owner->next\_count > 0) { // 释放锁

up(&cvp->owner->next);} else {

up(&cvp->owner->mutex);}down(&cvp->sem); // 将自己等待在条件变量上

cvp->count --; // 被唤醒，修正等待队列上的进程计数

接下来分析本实验中基于条件变量和管程的哲学家就餐问题的实现:

关于使用条件变量来完成哲学家就餐问题的实现中，总共有两个关键函数，以及使用到了N（哲学家数量）个条件变量，在管程中，还包括了一个限制管程访问的锁还有N个用于描述哲学家状态的变量（总共有EATING, THINKING, HUNGER）三种状态；

首先分析phi\_take\_forks\_condvar函数的实现，该函数表示指定的哲学家尝试获得自己所需要进餐的两把叉子，如果不能获得则阻塞，具体实现流程为：

给管程上锁；

将哲学家的状态修改为HUNGER；

判断当前哲学家是否有足够的资源进行就餐（相邻的哲学家是否正在进餐）；

如果能够进餐，将自己的状态修改成EATING，然后释放锁，离开管程即可；

如果不能进餐，等待在自己对应的条件变量上，等待相邻的哲学家释放资源的时候将自己唤醒；

最终具体的代码实现如下：

down(&(mtp->mutex)); // 获取管程的锁

state\_condvar[i] = HUNGRY; // 将自己设置为饥饿if (state\_condvar[(i + 4) % 5] != EATING && state\_condvar[(i + 1) % 5] != EATING) { // 判断当前叉子是否足够就餐

state\_condvar[i] = EATING; // 就餐} else {

cprintf("phi\_take\_forks\_condvar: %d didn’t get fork and will wait\n", i);

cond\_wait(mtp->cv + i); // 等待其他人释放资源}if(mtp->next\_count>0) // 释放管程的锁

up(&(mtp->next));else

up(&(mtp->mutex));

而phi\_put\_forks\_condvar函数则是释放当前哲学家占用的叉子，并且唤醒相邻的因为得不到资源而进入等待的哲学家：

首先获取管程的锁；

将自己的状态修改成THINKING；

检查相邻的哲学家是否在自己释放了叉子的占用之后满足了进餐的条件，如果满足，将其从等待中唤醒（使用cond\_signal）；

释放锁，离开管程；

最终的具体代码实现如下：

down(&(mtp->mutex)); // 获取管程的锁

state\_condvar[i] = THINKING; // 停止就餐cprintf("phi\_put\_forks\_condvar: %d finished eating\n", i); // 由于LAB7的评测脚本较弱，这是为了验证访问的互斥性而额外添加的注释性输出，可以用于my\_spj.py（本实验中额外完成的评测脚本）的测试phi\_test\_condvar((i + N - 1) % N); // 判断左右邻居的哲学家是否可以从等待就餐的状态中恢复过来phi\_test\_condvar((i + 1) % N);if(mtp->next\_count>0)

up(&(mtp->next));else

up(&(mtp->mutex));

由于限制了管程中在访问共享变量的时候处于RUNNABLE的进程只有一个，因此对进程的访问是互斥的；并且由于每个哲学家只可能占有所有需要的资源（叉子）或者干脆不占用资源，因此不会出现部分占有资源的现象，从而避免了死锁的产生；

根据上述分析，可知最终必定所有哲学将都能成功就餐；

扩展练习：了解java中同步互斥的实现机制，说明其与操作系统原理课的管程之间的关系，并用其实现写者优先的读者写者问题。