实验5 同步互斥

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);**

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。

关于条件变量机制的实现主要位于monitor.c文件中的cond\_signal, cond\_wait两个函数中，这两个函数的含义分别表示提醒等待在这个条件变量上的进程恢复执行，以及等待在这个条件变量上，直到有其他进行将其唤醒位置；下文将对这两个函数的具体实现进行分析：

* 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，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，是否可能存在该锁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}**

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。

内核级信号量的实现主要包含信号量数据结构semaphore\_t和实现P操作的函数down以及实现V操作的函数up

* **semaphore\_t：信号量数据结构。value是一个计数器，wait\_queue是等待队列。**

**typedef struct {**

**int value;**

**wait\_queue\_t wait\_queue;} semaphore\_t;**

* down：完成了信号量中的P操作。该函数主要调用了\_\_down函数。\_\_down函数中，首先关掉中断，然后判断信号量的value值是否大于0，如果大于0说明资源未被占用，则将value值减一并退出。若value值小于或等于0，则说明资源已经被占用，因此该进程需要等待。将该进程加入到等待队列中，开中断，然后进行调度。如果之后被V操作唤醒，则先关中断，将该进程从等待队列中删除，再开中断。

**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：完成了信号量中的V操作。该函数主要调用了\_\_up函数。在\_\_up中，首先关中断，如果当前等待队列为空则直接将value值加一，否则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。

**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 {**

**assert(wait->proc->wait\_state == wait\_state);**

**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。

当多个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，信号量通过V、P操作来修改传送信号量。

程序的入口是check\_sync函数。首先初始化了mutex信号量和五个哲学家对应的信号量s[i]，然后针对五个哲学家创建了五个进程来运行philosopher\_using\_semaphore函数

**void check\_sync(void){**

**int i;**

**//check semaphore**

**sem\_init(&mutex, 1);**

**for(i=0;i<N;i++){**

**sem\_init(&s[i], 0);**

**int pid = kernel\_thread(philosopher\_using\_semaphore, (void \*)i, 0);**

**if (pid <= 0) {**

**panic("create No.%d philosopher\_using\_semaphore failed.\n");**

**}**

**philosopher\_proc\_sema[i] = find\_proc(pid);**

**set\_proc\_name(philosopher\_proc\_sema[i], "philosopher\_sema\_proc");**

**}**

**......}**

观察循环体里的内容可以发现，哲学家循环进行思考（第一次do\_sleep(SLEEP\_TIME)）、拿起两只叉子（或者被阻塞，phi\_take\_forks\_sema(i)）、进餐（第二次do\_sleep(SLEEP\_TIME)）、放回两只叉子（phi\_put\_forks\_sema(i)）这四个操作。

**int philosopher\_using\_semaphore(void \* arg) /\* i：哲学家号码，从0到N-1 \*/{**

**int i, iter=0;**

**i=(int)arg;**

**cprintf("I am No.%d philosopher\_sema\n",i);**

**while(iter++<TIMES)**

**{ /\* 无限循环 \*/**

**cprintf("Iter %d, No.%d philosopher\_sema is thinking\n",iter,i); /\* 哲学家正在思考 \*/**

**do\_sleep(SLEEP\_TIME);**

**phi\_take\_forks\_sema(i);**

**/\* 需要两只叉子，或者阻塞 \*/**

**cprintf("Iter %d, No.%d philosopher\_sema is eating\n",iter,i); /\* 进餐 \*/**

**do\_sleep(SLEEP\_TIME);**

**phi\_put\_forks\_sema(i);**

**/\* 把两把叉子同时放回桌子 \*/**

**}**

**cprintf("No.%d philosopher\_sema quit\n",i);**

**return 0; }**

phi\_take\_forks\_sema函数中，哲学家尝试拿起两个叉子。如果得到两只叉子则流程继续，否则阻塞（等待对应的信号量被释放）。

**void phi\_take\_forks\_sema(int i) /\* i：哲学家号码从0到N-1 \*/{**

**down(&mutex); /\* 进入临界区 \*/**

**state\_sema[i]=HUNGRY; /\* 记录下哲学家i饥饿的事实 \*/**

**phi\_test\_sema(i); /\* 试图得到两只叉子 \*/**

**up(&mutex); /\* 离开临界区 \*/**

**down(&s[i]); /\* 如果得不到叉子就阻塞 \*/}**

phi\_put\_forks\_sema函数中，哲学家放下两只叉子。

**void phi\_put\_forks\_sema(int i) /\* i：哲学家号码从0到N-1 \*/{**

**down(&mutex); /\* 进入临界区 \*/**

**state\_sema[i]=THINKING; /\* 哲学家进餐结束 \*/**

**phi\_test\_sema(LEFT); /\* 看一下左邻居现在是否能进餐 \*/**

**phi\_test\_sema(RIGHT); /\* 看一下右邻居现在是否能进餐 \*/**

**up(&mutex); /\* 离开临界区 \*/}**

phi\_test\_sema(i)函数，用来测试第i个哲学家的左右两边的叉子是否都是可以获得的，如果可以则对这个哲学家的V操作。

**void phi\_test\_sema(i) /\* i：哲学家号码从0到N-1 \*/{**

**if(state\_sema[i]==HUNGRY&&state\_sema[LEFT]!=EATING**

**&&state\_sema[RIGHT]!=EATING)**

**{**

**state\_sema[i]=EATING;**

**up(&s[i]); }}**

2）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：

**typedef struct monitor{**

**semaphore\_t mutex; // 二值信号量，只允许一个进程进入管程，初始化为1**

**semaphore\_t next; //配合cv，用于进程同步操作的信号量**

**int next\_count; // 睡眠的进程数量**

**condvar\_t \*cv; // 条件变量cv**

**} monitor\_t;**

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：

**typedef struct condvar{**

**semaphore\_t sem; //用于发出wait\_cv操作的等待某个条件C为真的进程睡眠**

**int count; // 在这个条件变量上的睡眠进程的个数**

**monitor\_t \* owner; // 此条件变量的宿主管程**

**} condvar\_t;**

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：

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

**cprintf("cond\_signal begin: cvp %x, cvp->count %d, cvp->owner->next\_count %d\n", cvp, cvp->count, cvp->owner->next\_count);**

**if(cvp->count>0) { //当前存在睡眠的进程**

**cvp->owner->next\_count ++;//睡眠的进程总数加一**

**up(&(cvp->sem));//唤醒等待在cv.sem上睡眠的进程**

**down(&(cvp->owner->next));//把自己睡眠**

**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);**

**}**

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减一。

同样，再来看看cond\_wait函数，实现如下：

**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);**

**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));//否则唤醒睡在monitor.mutex上的进程**

**down(&(cvp->sem));//将自己睡眠**

**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);**

**}**

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要加一。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减一，表示等待此条件的睡眠进程个数少了一个，可继续执行了！

这样我们就可以在此基础上继续完成哲学家就餐问题的解决了，主要是就是如下的两个函数：

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

**down(&(mtp->mutex)); //通过P操作进入临界区**

**state\_condvar[i]=HUNGRY; //记录下哲学家i是否饥饿，即处于等待状态拿叉子**

**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]);//如果得不到叉子就睡眠**

**}**

**//如果存在睡眠的进程则那么将之唤醒**

**if(mtp->next\_count>0)**

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

**else**

**up(&(mtp->mutex));**

**}**

**void phi\_put\_forks\_condvar(int i) {**

**down(&(mtp->mutex));//通过P操作进入临界区**

**state\_condvar[i]=THINKING;//记录进餐结束的状态**

**phi\_test\_condvar(LEFT);//看一下左边哲学家现在是否能进餐**

**phi\_test\_condvar(RIGHT);//看一下右边哲学家现在是否能进餐**

**//如果有哲学家睡眠就予以唤醒**

**if(mtp->next\_count>0)**

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

**else**

**up(&(mtp->mutex));**

**}**