**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？**
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。**
3. **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）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，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

内核实现等待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。

**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。

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) 。

**3）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。**

信号量的数据结构定义如下：

[cpp] view plaincopy

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;

P操作函数和V操作函数的具体实现是 \_\_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）。具体实现如下所示：

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); // 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

local\_intr\_restore(intr\_flag);

schedule();

local\_intr\_save(intr\_flag); // 当被唤醒时

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

local\_intr\_restore(intr\_flag);

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

return wait->wakeup\_flags;

}

return 0;

}

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。具体实现如下所示：

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) { // 取得 wait queue 的第一个 wait

sem->value ++;

}

else {

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

}

}

local\_intr\_restore(intr\_flag);

}

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: 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。**
2. **说明ucore中基于管程的哲学家就餐问题的实现机制。**
3. **说明ucore中基于信号量的哲学家就餐问题的实现机制。**

基于信号量实现完成条件变量实现

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);

/\*

\* 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) {

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

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

down(&(mtp->mutex)) ;// 进入临界区

//--------into routine in monitor--------------

// 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

}

//--------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: 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

//--------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 }