# 实验5 同步互斥

## 常俊豪

## 171491108

## 练习1: 了解信号量和管程的实现机制

#### 同步互斥的底层支持是如何实现的？

定时器

在传统的操作系统中，定时器是其中一个基础而重要的功能.它提供了基于时间事件的调度机制。在ucore中，时钟中断给操作系统提供了有一定间隔的时间事件，操作系统将其作为基本的调度和计时单位。

屏蔽与使能中断

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。由于有开关中断的存在，使得我们在实现同步互斥原语上可以大大简化。

等待队列

用户进程或内核线程可以转入等待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。

#### 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。

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。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

#### 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：

typedef struct {

int value; wait\_queue\_t wait\_queue;

}

semaphore\_t;

在ucore中最重要的信号量操作是P操作函数)和V操作函数 。二者的具体实现描述 如下：

\_\_down：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：

\_\_up：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：

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。

## 练习2: 了解基于信号量和管程的哲学家就餐问题

#### 说明ucore中基于信号量的哲学家就餐问题的实现机制。

typedef struct {

int value;

wait\_queue\_t wait\_queue;

} semaphore\_t;

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);

}

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;

}

#### 说明ucore中基于管程的哲学家就餐问题的实现机制。

typedef struct condvar{

semaphore\_t sem;

int count;

monitor\_t \* owner;

typedef struct monitor{

semaphore\_t mutex;

semaphore\_t next;

int next\_count;

condvar\_t \*cv;

} monitor\_t;

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) {

cvp->owner->next\_count ++;

up(&(cvp->sem));

down(&(cvp->owner->next));

cvp->owner->next\_count --;

}

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

}

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);

cvp->count++;

if(cvp->owner->next\_count > 0)

up(&(cvp->owner->next));

else

up(&(cvp->owner->mutex));

down(&(cvp->sem));

cvp->count --;

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

}

function\_in\_monitor （…）

{

sem.wait(monitor.mutex);

function;

if(monitor.next\_count > 0)

sem\_signal(monitor.next);

else

sem\_signal(monitor.mutex);

}