**操作系统实验报告4**

171491221 姜世廷

**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。**

管程和条件变量

引入了管程是为了将对共享资源的所有访问及其所需要的同步操作集中并封装起来。Hansan为管程所下的定义：“一个管程定义了一个数据结构和能为并发进程所执行（在该数据结构上）的一组操作，这组操作能同步进程和改变管程中的数据”。有上述定义可知，管程由四部分组成：

​ 管程内部的共享变量；

​ 管程内部的条件变量；

​ 管程内部并发执行的进程；

​ 对局部于管程内部的共享数据设置初始值的语句。

局限在管程中的数据结构，只能被局限在管程的操作过程所访问，任何管程之外的操作过程都不能访问它；另一方面，局限在管程中的操作过程也主要访问管程内的数据结构。由此可见，管程相当于一个隔离区，它把共享变量和对它进行操作的若干个过程围了起来，所有进程要访问临界资源时，都必须经过管程才能进入，而管程每次只允许一个进程进入管程，从而需要确保进程之间互斥。

条件变量（CV）

一个条件变量CV可理解为一个进程的等待队列，队列中的进程正等待某个条件C变为真。每个条件变量关联着一个断言 “断言 (程序)”)Pc。当一个进程等待一个条件变量，该进程不算作占用了该管程，因而其它进程可以进入该管程执行，改变管程的状态，通知条件变量CV其关联的断言Pc在当前状态下为真。因此对条件变量CV有两种主要操作：

wait\_cv： 被一个进程调用，以等待断言Pc被满足后该进程可恢复执行. 进程挂在该条件变量上等待时，不被认为是占用了管程。

signal\_cv：被一个进程调用，以指出断言Pc现在为真，从而可以唤醒等待断言Pc被满足的进程继续执行。

管程的数据结构

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;

管程中的成员变量mutex是一个二值信号量，是实现每次只允许一个进程进入管程的关键元素，确保了互斥访问性质。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。管程中的成员变量信号量next和整形变量next\_count是配合进程对条件变量cv的操作而设置的，这是由于发出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。

condvar\_t的定义

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;

条件变量的定义中也包含了一系列的成员变量，信号量sem用于让发出wait\_cv操作的等待某个条件C为真的进程睡眠，而让发出signal\_cv操作的进程通过这个sem来唤醒睡眠的进程。count表示等在这个条件变量上的睡眠进程的个数。owner表示此条件变量的宿主是哪个管程。

monitor\_init的实现

void

monitor\_init (monitor\_t \* mtp, size\_t num\_cv) {

int i;

assert(num\_cv>0);

mtp->next\_count = 0; //设置next\_count为0

mtp->cv = NULL;

sem\_init(&(mtp->mutex), 1); //unlocked

sem\_init(&(mtp->next), 0);

mtp->cv =(condvar\_t \*) kmalloc(sizeof(condvar\_t)\*num\_cv);

assert(mtp->cv!=NULL);

for(i=0; i<num\_cv; i++){

mtp->cv[i].count=0; //设置cv的count为0

sem\_init(&(mtp->cv[i].sem),0);

mtp->cv[i].owner=mtp;

}

}

对条件变量进行初始化，设置next\_count为0，对mutex，next进行初始化， 并分配num个condvar\_t,设置cv的count为0，初始化cv的sem和owner。

cond\_wait的实现

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)); //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);

}

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！

cond\_signal的实现

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);

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);

}

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。

哲学家问题的数据结构

struct proc\_struct \*philosopher\_proc\_condvar[N]; //N个哲学家

int state\_condvar[N]; //哲学家的状态

monitor\_t mt, \*mtp=&mt; //管程

信号量的实现

semphore\_t mutex 临界区互斥信号量

semphore\_t s[N] 每个哲学家一个信号量

试图得到叉子

void phi\_test\_condvar (i) {

// 首先判断左右的状态是不是“eatting”

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

&&state\_condvar[RIGHT]!=EATING) {

cprintf("phi\_test\_condvar: state\_condvar[%d] will eating\n",i);

state\_condvar[i] = EATING ; //改变自己状态为eating

cprintf("phi\_test\_condvar: signal self\_cv[%d] \n",i);

cond\_signal(&mtp->cv[i]) ; //得到条件变量cv[i]

}

}

首先判断左右的状态不是“eatting”然后是自己的状态变成“EATING”。

拿叉子

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

down(&(mtp->mutex)); //通过P操作进入临界区

state\_condvar[i]=HUNGRY; //记录哲学家i是否饥饿

phi\_test\_condvar(i); //试图拿到叉子

if (state\_condvar[i] != EATING) {

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

cond\_wait(&mtp->cv[i]); //得不到叉子就睡眠

}

if(mtp->next\_count>0) //如果存在睡眠的进程则那么将之唤醒

up(&(mtp->next));

else

up(&(mtp->mutex));

}

放叉子

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

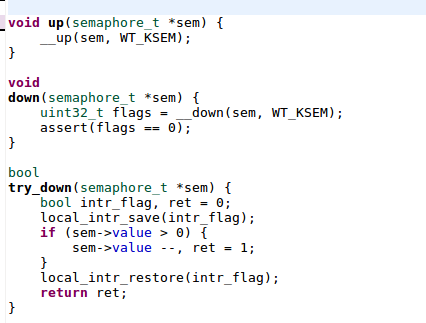
if(mtp->next\_count>0) //如果有哲学家睡眠就予以唤醒

up(&(mtp->next));

else

up(&(mtp->mutex)); //离开临界区

}



**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。**

**说明ucore中基于管程的哲学家就餐问题的实现机制。**

信号量

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。

struct semaphore {

int count;

queueType queue;

};

void semWait(semaphore s)

{

s.count--;

if (s.count < 0) {

/\* place this process in s.queue \*/;

/\* block this process \*/;

}

}

void semSignal(semaphore s)

{

s.count++;

if (s.count<= 0) {

/\* remove a process P from s.queue \*/;

/\* place process P on ready list \*/;

}

}

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。

信号量结构体

typedef struct {

int value;

wait\_queue\_t wait\_queue;

} semaphore\_t;

semaphore\_t是最基本的记录型信号量（record semaphore)结构，包含了用于计数的整数值value，和一个进程等待队列wait\_queue，一个等待的进程会挂在此等待队列上。

本次实验同步互斥的底层支撑

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);

……

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，二者的具体实现描述如下：

\_\_down(semaphore\_t \*sem, uint32\_t wait\_state, timer\_t \*timer)：

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

bool intr\_flag;

local\_intr\_save(intr\_flag); //关中断

if (sem->value > 0) { //可以获得信号量

sem->value --; //value减一

local\_intr\_restore(intr\_flag); //开中断

return 0;

}

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

wait\_current\_set(&(sem->wait\_queue), wait, wait\_state); //将当前进程加入等待队列

local\_intr\_restore(intr\_flag); //打开中断

schedule(); //调度下一进程

local\_intr\_save(intr\_flag); //被唤醒，关中断

wait\_current\_del(&(sem->wait\_queue),wait); //将当前进程移出等待队列

local\_intr\_restore(intr\_flag); //打开中断

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

return wait->wakeup\_flags;

}

return 0;

}

总结一下就是：

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）。

\_\_up(semaphore\_t \*sem, uint32\_t wait\_state)：

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

bool intr\_flag;

local\_intr\_save(intr\_flag); //关中断

{

wait\_t \*wait;

if ((wait = wait\_queue\_first(&(sem->wait\_queue))) == NULL) { //没有进程等待

sem->value ++; //value加一

}

else {

assert(wait->proc->wait\_state == wait\_state); //有进程等待

wakeup\_wait(&(sem->wait\_queue), wait, wait\_state, 1); //将第一个wait删除，唤醒进程。

}

}

local\_intr\_restore(intr\_flag); //打开中断

}

总结一下过程：

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

请在实验报告中给出内核级信号量的设计描述，并说其大致执行流流程。

对于V操作和P操作，分别用up和down函数来对应。

down()函数：

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）。

up()函数：

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。

请在实验报告中给出给用户态进程/线程提供信号量机制的设计方案，并比较说明给内核级提供信号量机制的异同。

用户态的进程/线程的信号量的数据结构和内核级的是一样的。 对于用户态的线程/进程使用信号量机制，应该首先通过系统调用进行sem的初始化，设置sem.value以及sem.wait\_queue，而在初始化之后，在使用这个信号量时，通过P操作与V操作，也是通过系统调用进入到内核中进行处理，简称是否等待或者释放资源。

不同： 在用户态使用信号量时，需要进行系统调用进入到内核态进行操作。

