**171491121 硬件一班 王倩倩**

**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？**

互斥是当一个进程在临界区访问共享资源时，其他进程不能进入该临界区访问任何共享资源。

互斥的实现有两个方面：硬件支持，软件设计

## **硬件支持**

## **禁用中断**，指的是在线程执行临界区代码过程中不可中断，即不会把CPU的时间片交出去，直到执行完成。这种方式在多处理器的系统中不能保证互斥，且即使在单核系统中能实现互斥，代价也是很大的。

## **专用的机器指令** 包括比较和交换指令和交换指令； 在硬件级别上，对存储单元的访问排斥对相同单元的其他访问。基于这一点，处理器的设计者提出了一些机器指令，用于保证两个动作的原子性，如在一个取指令周期中对一个存储器单元的读和写或者读和测试。在该指令执行的过程中，其他指令访问内存将被阻止。而且这些动作在一个指令周期中完成。

## 程序设计语言机制

1. 互斥量（基于二元信号量）
2. 管程（基于互斥量的结构）

**信号量**

信号量是多线程协作的一个共享变量，是进程（线程）间传递信号的一个整数值。它有三种原子操作：初始化、递减（P）和递增（V）。  
递减操作可以用来阻塞一个线程，递增操作可以用来唤醒一个线程。

**管程**

管程是编程语言提供的一种抽象数据结构，用于多线程互斥访问共享资源。首先，是互斥访问，即任一时刻只有一个线程在执行管程代码；第二，正在管程内的线程可以放弃对管程的控制权，等待某些条件发生再继续执行。

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。**

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。

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。

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。

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。**

信号量的数据结构定义如下：

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;

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

在ucore中最重要的信号量操作是P操作函数down和V操作函数 up。但这两个函数的具体实现是\_\_down 函数和\_\_up函数，二者的具体实现描述 如下：

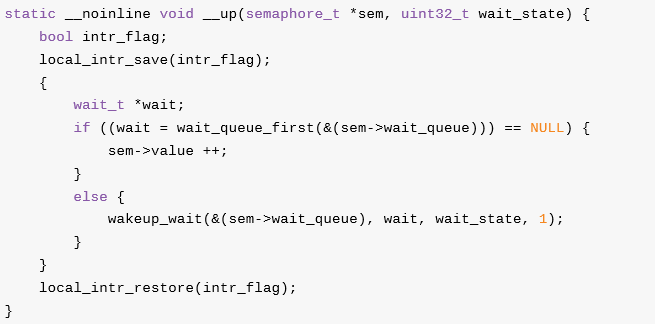
● \_\_down：具体实现信号量的P操 作，首先关掉中断，然后判断当前信号量的value是否大于0。如果是>0，则表明可以获得信 号量，故让value减一，并打开中断返回即可；如果不是>0，则表明无法获得信号量，故需要 将当前的进程加入到等待队列中，并打开中断，然后运行调度器选择另外一个进程执行。如 果被V操作唤醒，则把自身关联的wait从等待队列中删除（此过程需要先关中断，完成后开中 断）。具体实现如下所示：



与 \_\_down 相关的调用和被调用函数关系图如下所示：



● \_\_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



与 \_\_up 相关的调用和被调用函数关系图如下所示：



**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。**

描述的信号量的实现方法：

wait(semaphore \*S){

S->value--;

if(S->value <0) {

add this process to S->list;

block();

}

}

signal(semaphore \*S){

S->value++;

if(S->value <=0) {

remove a process P from S->list;

wakeup(P);

}

}

具体的信号量数据结构被定义在（kern/sync/sem.h）中：

typedef struct {

int value;

wait\_queue\_t wait\_queue;

} semaphore\_t;

找到相关函数init\_main（kern/process/proc.c，838）：

static int

init\_main(void \*arg) {

size\_t nr\_free\_pages\_store = nr\_free\_pages();

size\_t kernel\_allocated\_store = kallocated();

int pid = kernel\_thread(user\_main, NULL, 0);

if (pid <= 0) {

panic("create user\_main failed.\n");

}

extern void check\_sync(void);

check\_sync(); // check philosopher sync problem

while (do\_wait(0, NULL) == 0) {

schedule();

}

cprintf("all user-mode processes have quit.\n");

assert(initproc->cptr == NULL && initproc->yptr == NULL && initproc->optr == NULL);

assert(nr\_process == 2);

assert(list\_next(&proc\_list) == &(initproc->list\_link));

assert(list\_prev(&proc\_list) == &(initproc->list\_link));

assert(nr\_free\_pages\_store == nr\_free\_pages());

assert(kernel\_allocated\_store == kallocated());

cprintf("init check memory pass.\n");

return 0;

}

它调用了check\_sync()这个函数去执行了哲学家就餐问题，分析check\_sync函数（kern/sync/check\_sync.c，182+行）：

void check\_sync(void)

{

int i;

//check semaphore

sem\_init(&mutex, 1);

for(i=0;i<N;i++) { //N是哲学家的数量

sem\_init(&s[i], 0); //初始化信号量

int pid = kernel\_thread(philosopher\_using\_semaphore, (void \*)i, 0);//线程需要执行的函数名、哲学家编号、0表示共享内存

//创建哲学家就餐问题的内核线程

if (pid <= 0) { //创建失败的报错

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");

}

//check condition variable

monitor\_init(&mt, N);

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

state\_condvar[i]=THINKING;

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

if (pid <= 0) {

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

}

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

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

}

}

首先观察到利用kernel\_thread函数创建了一个哲学家就餐问题的内核线程（kern/process/proc.c，270）

简单的来说，这个函数需要传入三个参数：

第一个fn是一个函数，代表这个创建的内核线程中所需要执行的函数；第二个arg是相关参数，这里传入的是哲学家编号i；第三部分是共享内存的标记位，内核线程之间内存是共享的，因此应该设置为0。其余地方则是设置一些寄存器的值，保留需要执行的函数开始执行的地址，以便创建了新的内核线程之后，函数能够在内核线程中找到入口地址，执行函数功能。接下来，让我们来分析需要创建的内核线程去执行的目标函数philosopher\_using\_semaphore（kern/sync/check\_sync.c，52）：

int philosopher\_using\_semaphore(void \* arg)

{

int i, iter=0;

i=(int)arg; //传入的参数转为int型，代表哲学家的编号

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

while(iter++<TIMES) //TIMES=4

{

cprintf("Iter %d, No.%d philosopher\_sema is thinking\n",iter,i); do\_sleep(SLEEP\_TIME);//等待

phi\_take\_forks\_sema(i);

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

do\_sleep(SLEEP\_TIME);

phi\_put\_forks\_sema(i);

} //哲学家思考一段时间，吃一段时间饭

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

return 0;

}

参数及其分析：

传入参数\*arg，代表在上一个函数中“参数”部分定义的(void \*)i，是哲学家的编号。iter++<TIMES，表示循环4次，目的在于模拟多次试验情况。从这个函数，我们看到，哲学家需要思考一段时间，然后吃一段时间的饭，这里面的“一段时间”就是通过系统调用sleep实现的，内核线程调用sleep，然后这个线程休眠指定的时间，从某种方面模拟了吃饭和思考的过程。

目前看来，最关键的函数是phi\_take\_forks\_sema(i)，和phi\_take\_forks\_sema(i);

phi\_take\_forks\_sema；phi\_take\_forks\_sema：（kern/sync/check\_sync,c，34——50行）：

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 \*/

}

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 \*/

}

其中，mutex的数据类型是“信号量结构体，up函数的作用是：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。

down函数的作用是：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）。test函数：

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 \*/

该函数被定义在（kern/sync/check\_sync.c，86——94行）：

void phi\_test\_sema(i)

{

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

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

{

state\_sema[i]=EATING;

up(&s[i]);

}

}

在试图获得筷子的时候，函数的传入参数为i，即为哲学家编号，此时，他自己为HUNGRY，而且试图检查旁边两位是否都在吃。如果都不在吃，那么可以获得EATING的状态。在从吃的状态返回回到思考状态的时候，需要调用两次该函数，传入的参数为当前哲学家左边和右边的哲学家编号，因为他试图唤醒左右邻居，如果左右邻居满足条件，那么就可以将他们设置为EATING状态。其中，LEFT和RIGHT的定义如下：

#define LEFT (i-1+N)%N

#define RIGHT (i+1)%N

由于哲学家坐圆桌，因此可以使用余数直接获取左右编号。

**（2）说明ucore中基于管程的哲学家就餐问题的实现机制。**

管程由四部分组成：  
①管程内部的共享变量；  
②管程内部的条件变量；  
③管程内部并发执行的进程；  
④对局部于管程内部的共享数据设置初始值的语句。

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。

但在管程中仅仅有互斥操作是不够用的。进程可能需要等待某个条件C为真才能继续执行。如果采用忙等(busy waiting)方式：

while not( C ) do {}

在单处理器情况下，将会导致所有其它进程都无法进入临界区使得该条件C为真，该管程的执行将会发生死锁。为此，可引入条件变量。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：

①wait\_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。

②signal\_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。

有了互斥和信号量支持的管程就可用用了解决各种同步互斥问题。

ucore中的管程机制是基于信号量和条件变量来实现的。ucore中的管程的数据结构monitor\_t定义如下：（kern/sync/monitor.h）

typedef struct monitor{

// 二值信号量，只允许一个进程进入管程，初始化为1

semaphore\_t mutex; // the mutex lock for going into the routines in monitor, should be initialized to 1

//用于进程同步操作的信号量

semaphore\_t next; // the next semaphore is used to down the signaling proc itself, and the other OR wakeuped waiting proc should wake up the sleeped signaling proc.

// 睡眠的进程数量

int next\_count; // the number of of sleeped signaling proc

// 条件变量cv

condvar\_t \*cv; // the condvars in monitor

} 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。

其中，条件变量cv的数据结构也被定义在同一个位置下：

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;

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。

其实本来条件变量中需要有等待队列的成员，以表示有多少线程因为当前条件得不到满足而等待，但这里，直接采用了信号量替代，因为信号量数据结构中也含有等待队列。那么现在开始解决哲学家就餐问题，使用管程，它的实现在（kern/sync/check\_sync，199+行）

monitor\_init(&mt, N); //初始化管程

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

state\_condvar[i]=THINKING;

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

if (pid <= 0) {

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

}

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

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

我们发现，这个实现过程和使用信号量无差别，不同之处在于，各个线程所执行的函数不同，此处执行的为philosopher\_using\_condvar函数：

int philosopher\_using\_condvar(void \* arg) { /\* arg is the No. of philosopher 0~N-1\*/

int i, iter=0;

i=(int)arg;

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

while(iter++<TIMES)

{ /\* iterate\*/

cprintf("Iter %d, No.%d philosopher\_condvar is thinking\n",iter,i); /\* thinking\*/

do\_sleep(SLEEP\_TIME);

phi\_take\_forks\_condvar(i);

/\* need two forks, maybe blocked \*/

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

do\_sleep(SLEEP\_TIME);

phi\_put\_forks\_condvar(i);

/\* return two forks back\*/

}

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

return 0;

}

我们发现这里和用信号量还是没有本质的差别，不同之处在于，获取筷子和放下都使用了不同的，配套管程使用的函数phi\_take\_forks\_condvar和phi\_put\_forks\_condvar。

phi\_take\_forks\_condvar和phi\_put\_forks\_condvar被定义在（kern/sync/check\_sync，121——159行）

其中，mtp为一个管程，声明于同一文件下的第108行，state\_convader数组记录哲学家的状态，声明于第107行。

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

down(&(mtp->mutex)); //保证互斥操作

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

// LAB7 EXERCISE1: YOUR CODE

// I am hungry

// try to get fork

// I am hungry

state\_condvar[i]=HUNGRY;

// try to get fork

phi\_test\_condvar(i); //测试哲学家是否能拿到筷子

while (state\_condvar[i] != EATING) { //没拿到，需要等待，调用wait函数

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

cond\_wait(&mtp->cv[i]);

}

//--------leave routine in monitor--------------

if(mtp->next\_count>0)

up(&(mtp->next));

else

up(&(mtp->mutex));

}

最主要的部分在于管程的signal和wait操作，ucore操作系统中对于signal和wait操作的实现是有专门的函数的：

他们是cond\_signal和cond\_wait（kern/sync/monitor.c，26）：

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); //这是一个输出信息的语句，可以不管

if(cvp->count>0) {

cvp->owner->next\_count ++;//管程中睡眠的数量

up(&(cvp->sem)); //唤醒在条件变量里睡眠的进程

down(&(cvp->owner->next)); //将在管程中的进程睡眠

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);}

再来看wait函数：

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++; //条件变量中睡眠的进程数量加加

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

up(&(cvp->owner->next)); //如果当前有进程正在等待，且睡在宿主管程的信号量上，此时需要唤醒，让该调用了wait的睡，此时就唤醒了，对应上面讨论的情况。这是一个同步问题。

else

up(&(cvp->owner->mutex)); //如果没有进程睡眠，那么当前进程无法进入管程的原因就是互斥条件的限制。因此唤醒mutex互斥锁，代表现在互斥锁被占用，此时，再让进程睡在宿主管程的信号量上，如果睡醒了，count--，谁唤醒的呢？就是前面的signal啦，这其实是一个对应关系。

down(&(cvp->sem)); //因为条件不满足，所以主动调用wait的进程，会睡在条件变量cvp的信号量上，是条件不满足的问题；而因为调用signal唤醒其他进程而导致自身互斥睡眠，会睡在宿主管程cvp->owner的信号量上，是同步的问题。两个有区别，不要混了，超级重要鸭！！！

cvp->count --;

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

}**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。**