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？
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）同步底层支持

并发错误

多线程带来效率的提高，但是也带来了对原子操作的破坏。

例如

C代码

汇编代码

i++;

mov eax,[i]

inc eax

mov [i],eax

假设i=0; 两个线程执行后应该为2；

如果按照如下方式执行结果为1

线程1

线程2

mov eax,[i];

等待

等待

mov eax,[i];

等待

Inc eax;

等待

mov [i],eax;

inc eax;

等待

mov [i],eax;

等待

原子操作

如果是单处理器使用一条指令完成即可实现原子操作Inc [i];

如果是多处理器inc[i]; 依然不是原子的；使用lock 指令前缀锁定内存后才能保证指令的原子性 lock inc [i];

自旋锁中的原子操作

从上面可以看出，对数据的原子操作，直接是硬件支持的，如果操作的数据比较复杂要实现对这个复杂数据的原子操作就需要硬件和软件一起配合来完成。具体方法如下：

设有变量l 为0表示已经加锁，1 表示未加锁

加锁代码如下：

1：      lock    decb    [l]

jns   3

2:   rep      nop

      cmpb   0,   [l]

      jle 2

      jmp      1

3:   ….(排它性操作代码操作)

      Lock     incb      [l]

如果未加锁:的执行路径为： 执行完1 跳转到3继续执行后面的逻辑。

如果已经加锁的 执行路径为：执行完1，因跳转条件不成立，就执行2，并在2中不停的循环检查，直到锁释放(l为1)跳转到1重新执行，直到具备执行3中的逻辑，这样就保证了3中代码的原子性，在3中可以执行复杂的数据操作。

信号量中的原子操作

设有信号量sc初始为0。

申请操作如下：

Lock      decl [sc] // 对信号量减1操作

Js    2          // 信号量小于0

1:   ….(申请成功执行的代码)

2： lea eax,      [sc] //

      Call 申请信号量失败函数 //(进入内核态睡眠，有信号量从内核态返回)

      Jmp     1    //

释放操作如下：

      Lock incl [sc]          // 对信号量加1操作

      Jle 2                //小于有需要唤醒的线程

1:   信号量释放成功的代码逻辑

2： lea  eax,      [sc]

      Call      唤醒阻塞的线程（进入内核态）

      Jmp      1

互斥底层支持

并发错误

多线程带来效率的提高，但是也带来了对原子操作的破坏。

例如

C代码

汇编代码

i++;

mov eax,[i]

inc eax

mov [i],eax

假设i=0; 两个线程执行后应该为2；

如果按照如下方式执行结果为1

线程1

线程2

mov eax,[i];

等待

等待

mov eax,[i];

等待

Inc eax;

等待

mov [i],eax;

inc eax;

等待

mov [i],eax;

等待

原子操作

如果是单处理器使用一条指令完成即可实现原子操作Inc [i];

如果是多处理器inc[i]; 依然不是原子的；使用lock 指令前缀锁定内存后才能保证指令的原子性 lock inc [i];

自旋锁中的原子操作

从上面可以看出，对数据的原子操作，直接是硬件支持的，如果操作的数据比较复杂要实现对这个复杂数据的原子操作就需要硬件和软件一起配合来完成。具体方法如下：

设有变量l 为0表示已经加锁，1 表示未加锁

加锁代码如下：

1：      lock    decb    [l]

jns   3

2:   rep      nop

      cmpb   0,   [l]

      jle 2

      jmp      1

3:   ….(排它性操作代码操作)

      Lock     incb      [l]

如果未加锁:的执行路径为： 执行完1 跳转到3继续执行后面的逻辑。

如果已经加锁的 执行路径为：执行完1，因跳转条件不成立，就执行2，并在2中不停的循环检查，直到锁释放(l为1)跳转到1重新执行，直到具备执行3中的逻辑，这样就保证了3中代码的原子性，在3中可以执行复杂的数据操作。

信号量中的原子操作

设有信号量sc初始为0。

申请操作如下：

Lock      decl [sc] // 对信号量减1操作

Js    2          // 信号量小于0

1:   ….(申请成功执行的代码)

2： lea eax,      [sc] //

      Call 申请信号量失败函数 //(进入内核态睡眠，有信号量从内核态返回)

      Jmp     1    //

释放操作如下：

      Lock incl [sc]          // 对信号量加1操作

      Jle 2                //小于有需要唤醒的线程

1:   信号量释放成功的代码逻辑

2： lea  eax,      [sc]

      Call      唤醒阻塞的线程（进入内核态）

      Jmp      1

（2）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。

* count > 0，表示共享资源的空闲数
* count < 0，表示该信号量的等待队列里的进程数
* count = 0，表示等待队列为空

实现了内核级信号量机制的函数均定义在 sem.c 中，因此对上述这些函数分析总结如下：

* sem\_init：对信号量进行初始化的函数，根据在原理课上学习到的内容，信号量包括了等待队列和一个整型数值变量，该函数只需要将该变量设置为指定的初始值，并且将等待队列初始化即可；
* \_\_up：对应到了原理课中提及到的 V 操作，表示释放了一个该信号量对应的资源，如果有等待在了这个信号量上的进程，则将其唤醒执行；结合函数的具体实现可以看到其采用了禁用中断的方式来保证操作的原子性，函数中操作的具体流程为：
  + 查询等待队列是否为空，如果是空的话，给整型变量加 1；
  + 如果等待队列非空，取出其中的一个进程唤醒；
* \_\_down：同样对应到了原理课中提及的P操作，表示请求一个该信号量对应的资源，同样采用了禁用中断的方式来保证原子性，具体流程为：
  + 查询整型变量来了解是否存在多余的可分配的资源，是的话取出资源（整型变量减 1），之后当前进程便可以正常进行；
  + 如果没有可用的资源，整型变量不是正数，当前进程的资源需求得不到满足，因此将其状态改为 SLEEPING 态，然后将其挂到对应信号量的等待队列中，调用 schedule 函数来让出 CPU，在资源得到满足，重新被唤醒之后，将自身从等待队列上删除掉；
* up, down：对 \_\_up, \_\_down 函数的简单封装；
* try\_down：不进入等待队列的 P 操作，即时是获取资源失败也不会堵塞当前进程

（3）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。

管程主要由这四个部分组成：

* 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。

所谓条件变量，即将等待队列和睡眠条件包装在一起，就形成了一种新的同步机制，称为条件变量。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" 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。

因而条件变量两种操作如下：

* 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。如果条件不能满足，就需要等待。
* 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。如果条件可以满足，那么可以运行。

在 ucore 中，管程数据结构被定义在（kern/sync/monitor.h）中：

// 管程数据结构typedef struct monitor{

// 二值信号量，用来互斥访问管程，只允许一个进程进入管程，初始化为 1

semaphore\_t mutex; // 二值信号量 用来互斥访问管程

//用于进程同步操作的信号量

semaphore\_t next;// 用于条件同步（进程同步操作的信号量），发出 signal 操作的进程等条件为真之前进入睡眠

// 睡眠的进程数量

int next\_count;// 记录睡在 signal 操作的进程数

// 条件变量cv

condvar\_t \*cv;// 条件变量} 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：

// 条件变量数据结构typedef struct condvar{

// 用于条件同步 用于发出 wait 操作的进程等待条件为真之前进入睡眠

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

// 记录睡在 wait 操作的进程数(等待条件变量成真)

int count; //在这个条件变量上的睡眠进程的个数

// 所属管程

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。

我们对管程进行初始化操作：

// 初始化管程void monitor\_init (monitor\_t \* mtp, size\_t num\_cv) {

int i;

assert(num\_cv>0);

mtp->next\_count = 0; // 睡在 signal 进程数 初始化为 0

mtp->cv = NULL;

sem\_init(&(mtp->mutex), 1); // 二值信号量 保护管程 使进程访问管程操作为互斥的

sem\_init(&(mtp->next), 0); // 条件同步信号量

mtp->cv =(condvar\_t \*) kmalloc(sizeof(condvar\_t)\*num\_cv); // 获取一块内核空间 放置条件变量

assert(mtp->cv!=NULL);

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

mtp->cv[i].count=0;

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

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

}}

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。

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);}

在实验中，实现了应用信号量机制的哲学家问题。

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");

}

......}

philosopher\_using\_semaphore函数的内容如下。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; }

涉及到信号量的使用的主要是phi\_take\_forks\_sema和phi\_put\_forks\_sema两个函数。

在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 \*/}

在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 \*/}

在以上两个函数中，还调用了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。

内核级条件变量的哲学家就餐问题在check\_sync处实现。同信号量的测试相似，这里也是创建了5个内核进程表示5个哲学家的行为。

void check\_sync(void){

int i;

//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");

}}

实现了哲学家行为的函数philosopher\_using\_condvar也和信号量实现的philosopher\_using\_semaphore相似。哲学家尝试4次思考->拿叉子->吃饭->放下叉子。

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; }

拿叉子和放下叉子的函数phi\_take\_forks\_condvar和phi\_put\_forks\_condvar内容需要自己填写。

phi\_take\_forks\_condvar：首先进入管程，将哲学家状态改为HUNGRY，然后通过phi\_test\_condvar查看该哲学家对应的条件变量是否可以获得，如果不能则等待。最后退出管程。

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

down(&(mtp->mutex));

state\_condvar[i] = HUNGRY;

phi\_test\_condvar(i);

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

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

}

if(mtp->next\_count>0)

up(&(mtp->next));

else

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

phi\_puta\_forks\_condvar：首先进入管程，将哲学家状态改为THINKING，然后通过phi\_test\_condvar查看该哲学家左右两位是否可以同时获得两把叉子，如果能则唤醒左右两个条件变量。最后退出管程。

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

down(&(mtp->mutex));

state\_condvar[i] = THINKING;

phi\_test\_condvar(LEFT);

phi\_test\_condvar(RIGHT);

if(mtp->next\_count>0)

up(&(mtp->next));

else

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

条件变量用信号量来实现，在实验中，条件变量的wait和signal需要自己完成。

void

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

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);}

void

cond\_wait (condvar\_t \*cvp) {

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);}