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？

在操作系统原理中，知道了实现同步互斥机制的层次设计。处于底层的就是硬件支持，例如本实验应用的屏蔽/使能中断，还有诸如test\_and\_set\_bit等原子操作机器指令、LOCK机器指令的应用都在硬件底层为我们提供了支持。再往上就是软件层次的实现，例如多种锁机制。再往上就是更高级的抽象，用来实现系统中进程的同步互斥。

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。和自旋锁相比，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。以下是信号量的“抽象”定义：

//下面是信号量的类定义

struct semaphore {

int count; //信号量值

queueType queue; //等待队列

};

//在uCore中是下面这样定义的（利用了已定义好的类型）

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;

//以下是P操作的流程，或者说是信号量中的wait()函数

void semWait(semaphore s)

{

s.count--;

if (s.count < 0) { //注意此时判断条件是小于0

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

/\* block this process \*/;

}

}

//以下是V操作的流程，或者说是信号量中的signal()函数

void semSignal(semaphore s)

{

s.count++;

if (s.count<= 0) { //注意此时判断条件是小于等于0

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

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

}

}

下面看P操作，即wait()函数。在uCore中，这个函数的原型是down(semaphore\_t \*sem)，其进一步实现是\_\_down(semaphore\_t \*sem, uint32\_t wait\_state)，代码如下：

其执行逻辑是：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）。

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);

//以上是P操作的主要处理部分，当操作正常完成时，正常返回

//如果这个过程中进程被阻塞，则调用schedule()函数

schedule();

//当进程重新获得执行权，也即是被唤醒时，执行把自己从阻塞（等待）队列中删除的过程

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;

}

再看V操作，即signal()函数。在uCore中，这个函数的原型是up(semaphore\_t \*sem)，其进一步实现是\_\_up(semaphore\_t \*sem, uint32\_t wait\_state)，代码如下：

其执行逻辑是：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。

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 {

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

}

}

local\_intr\_restore(intr\_flag);

}

注意信号量中保证原子操作的机制，即关闭中断再恢复中断的过程。

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);}

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。

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]);

}}

1. 说明ucore中基于管程的哲学家就餐问题的实现机制。

原代码中已经提供哲学家就餐问题的大部分实现，再结合注释不难完成剩下的编码。无论是用管程还是信号量，此问题的框架是相同的：哲学家都是先思考一段时间，然后拿起叉子吃饭，吃了一段时间后，放下叉子继续思考，如此反复。思考和吃饭的过程都通过do\_sleep来实现，实际上就是利用定时器进行延时。因此，主要的差别在于拿叉子和放叉子两个功能的实现。

拿叉子（phi\_take\_forks\_condvar）：首先对管程的mutex执行down操作进入临界区，这是确保任何时候最多只有一个进程进入管程。然后将自己的state设置为HUNGRY。接着测试是否满足就餐条件：自己的state是HUNGRY，而且左右的哲学家的state都不是EATING。如果满足，则将自己的state设置为EATING，然后对管程的mutex执行up操作而退出临界区，从而完成拿叉子操作。如果不满足就餐条件，则调用cond\_wait将自己堵塞。

放叉子（phi\_put\_forks\_condvar）：同样，首先对管程的mutex执行down操作进入临界区，然后将自己的state设置为THINKING。接下来测试左右哲学家是否满足就餐条件，若满足，则调用cond\_signal将对应进程唤醒，而把自己堵塞。

内核线程initproc调用check\_sync检查使用管程来解决哲学家就餐问题的方案。check\_sync首先调用monitor\_init初始化管程，然后调用kernel\_thread创建5个内核线程，分别对应5位哲学家，并先将5位哲学家的state初始化为THINKING。此时无人占用叉子，RUNNABLE队列rq的元素依次为0,1,2,3,4，timer为空。

5个哲学家线程依次执行philosopher\_using\_condvar，打印自己的ID，然后开始思考（实际上是调用do\_sleep进行延时）。此时无人占用叉子，rq为空，timer依次是0,1,2,3,4.

哲学家0的延时最先结束，调用phi\_take\_forks\_condvar试图拿起2把叉子就餐，整个过程如下：对mtp->mutex执行down操作进入临界区（以保证互斥执行此函数），将自己的state设置为HUNGRY，调用phi\_test\_condvar拿到2把叉子，将自己的state改为EATING，调用cond\_signal唤醒之前由于执行cond\_wait而堵塞的进程（由于没有，此处啥也没做）。然后对mtp->mutex执行up操作而离开临界区，开始吃饭（实际上是调用do\_sleep来延时）。此时哲学家0占用叉子，rq为空，timer是1,2,3,4,0.

接着哲学家1延时结束，同样调用phi\_take\_forks\_condvar试图拿起2把叉子就餐，但由于左边的叉子正在被哲学家0占用，哲学家1只能调用cond\_wait进行等待。具体而言包括3步：将mtp->cv[1]->count加1，表示自己要等待该条件变量，然后对mtp->mutex执行up操作而离开临界区，最后对mtp->cv[1]->sem执行down操作而堵塞。此时哲学家0占用叉子，rq为空，timer是2,3,4,0.

哲学家2的执行过程与哲学家0相同，哲学家3、4的执行过程和哲学家1相同。最终，哲学家0和2占用叉子，rq为空，timer为0,2。

哲学家0延时结束，调用phi\_put\_forks\_condvar同时放下2把叉子。首先对mtp->mutex执行down操作而进入临界区，将自己的state修改为THINKING，然后调用phi\_test\_condvar检查左右的哲学家状态。首先检查到左边的哲学家4满足就餐条件，于是将哲学家4的state修改为EATING，并对mtp->cv[4]执行cond\_signal以唤醒哲学家4.具体而言包括3步：将mtp->next\_count加1，表示唤醒哲学家4后自己要进入等待状态，然后对mtp->cv[4]->sem执行up操作，这将唤醒哲学家4，最后对mtp->next执行down操作而堵塞。这时哲学家2占用叉子，rq为4，timer为2.

哲学家4被唤醒后，退出cond\_wait，对mtp->next执行up操作，从而将哲学家0唤醒。然后哲学家4开始吃饭。（这里可以看到管程的“临时退出临界区”的特点：上一步中哲学家0进入临界区，发现哲学家4满足就餐条件后，将其唤醒，让哲学家4临时进入临界区吃饭，最后哲学家4再把哲学家0唤醒，让哲学家0继续执行。）此时哲学家2和4占用叉子，rq为0，timer为2,4.

哲学家0继续检查右边的哲学家1，发现其不满足就餐条件，不作处理，然后对mtp->mutex执行up操作而退出临界区。最后哲学家0进入下一轮的思考。此时哲学家2和4占用叉子，rq为空，timer为2,4,0.

哲学家2延时结束，调用phi\_put\_forks\_condvar同时放下2把叉子。其执行流程与步骤6类似，只是哲学家2唤醒的是哲学家1.此时哲学家4占用叉子，rq为1，timer为4,0.

哲学家1被唤醒后的执行流程与步骤7类似：退出cond\_wait，对mtp->next执行up操作，从而将哲学家2唤醒。然后哲学家1开始吃饭。此时哲学家1和4占用叉子，rq为2，timer为4,0,1.

哲学家2被唤醒后的执行流程与步骤8类似：继续检查出右边的哲学家3，发现其不满足就餐条件，不作处理，然后对mtp->mutex执行up操作而退出临界区。最后哲学家2进入下一轮的思考。此时哲学家1和4占用叉子，rq为空，timer为4,0,1.

哲学家4延时结束，调用phi\_put\_forks\_condvar同时放下2把叉子，其执行流程与步骤6类似：首先检查到左边的哲学家3满足就餐条件，于是将其唤醒，并将自己堵塞。此时哲学家1占用叉子，rq为3，timer为0,1.

哲学家3被唤醒后的执行流程与步骤7类似：退出cond\_wait，对mtp->next执行up操作，从而将哲学家4唤醒。然后哲学家3开始吃饭。此时哲学家1和3占用叉子，rq为4，timer为0,1,3.