实验4

## 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？**

进程同步是对多个相关进程在执行次序上进行协调，使并发执行的进程之间能按照一定的规则或时序共享系统资源，并且能很好的相互合作，从而使程序的执行具有可再现性。

而实现进程同步很重要的一个方面就是实现互斥。

有三种方法可以实现同步互斥：

软件方法：软件方法具有一定的难度，且具有局限性。所以现在很少使用。

硬件方法：

1.禁用中断：禁用中断后，进程停不下来，整个系统都会因为它停下来，产生饥饿现象。也可能产生其他严重的后果。而且这种方法也不适用多CPU的系统。

2.更高级的抽象方法：利用原子操作指令，数据结构锁等。（Test and Set，Swap）。利用这种方法，很难解决复杂的进程同步问题。

信号量机制：

P操作和V操作。P操作使sem值减一，如果值小于零，则等待。V操作使sem值加一，如果，sem值大于等于零，则唤醒进程。

管程机制：

管程是一种用于多线程互斥访问共享资源的程序结，对于请求访问共享资源的诸多并发进程，可以根据资源的情况接受或阻塞，确保每次仅有一个进程进入管程。执行这组过程，达到对共享资源所有访问的统一管理，有效的实现进程互斥。管程由零个或多个条件变量，和锁所组成。当某进程通过管城程请求获得临界资源，而未能满足时，管程便调用wait原语使该进程等待，并将其排在等队列上，仅当另一进程访问完毕后并释放该资源，管程才调用signal原语唤醒等待队列中的队首进程。

**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。**

wait.c是创建一个等待的队列，用于存放不能访问互斥资源的进程。某些函数如下：

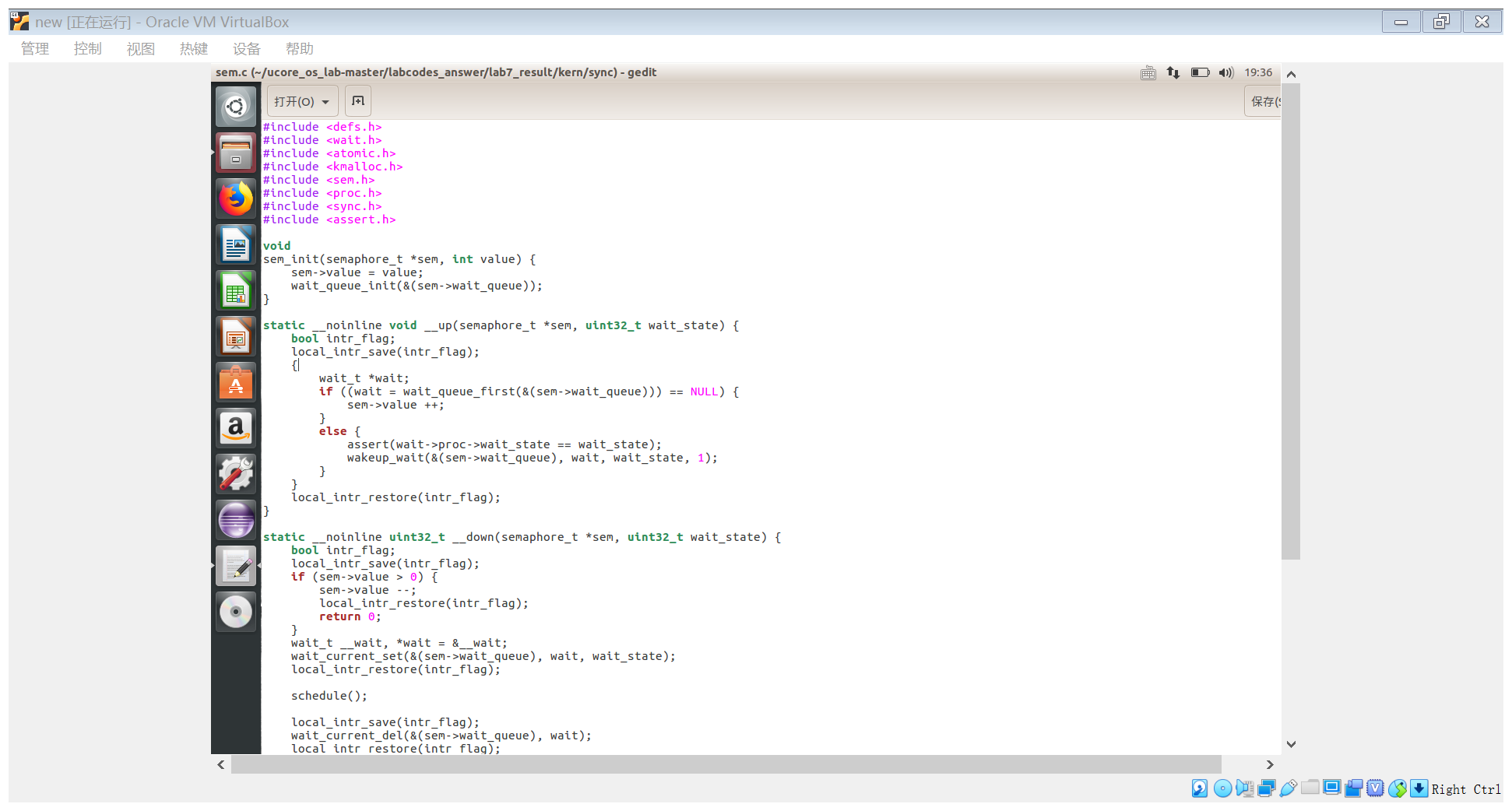
wait\_queue\_init(wait\_queue\_t \*queue)初始化等待队列

wait\_queue\_add(wait\_queue\_t \*queue, wait\_t \*wait)由于互斥资源被访问，当前进程便加入等待队列中

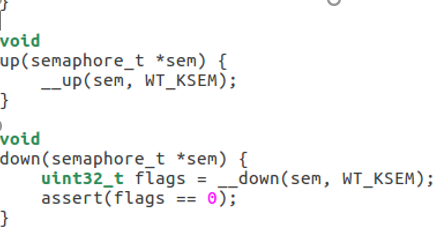
wait\_queue\_del(wait\_queue\_t \*queue, wait\_t \*wait)在等待队列中进程被唤醒，便执行此函数，而关于型号量的实现代码是在sem.c中的。

资源的数量由value的值给出，这是对互斥访问资源的初始化，对应原理课上的sem。

（1）



(2)



up对应的是V操作，down对应的是P操作，但是具体实现是看下里面的两个函数。

P操作对应的代码：



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）。

V操作的实现代码：

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。



**信号量的实现**

class Semaphore {

int sem;

WaitQueue q;

}

Semaphore::P() {

sem--;

if (sem < 0) {

add this progress p to q;

block(p);

}

}

Semaphore::V() {

sem++;

if (sem >= 0) {//sem为负数时说明有进程在等待资源，而sem自增后还不为正数说明有进程在等待资源

remove progress p from q;

wakeup(p);

}

}

信号量的使用

信号量可分为两种信号量：

二进制信号量：资源数目为0或者1

资源信号量：资源数目为任何非负值

两者等价，基于一个可以实现另一个。

## 练习2: 了解基于信号量和管程的哲学家就餐问题

首先掌握管程机制，然后基于信号量实现完成条件变量实现，然后用管程机制实现哲学家就餐问题的解决方案（基于条件变量）。给出内核级条件变量的设计描述，并说其大致执行流流程。给出给用户态进程/线程提供条件变量机制的设计方案，并比较说明给内核级提供条件变量机制的异同。

**1、原理：**

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（Condition Variables，简称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：

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：

typedef struct monitor{

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

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.

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

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。

管程中的条件变量的数据结构condvar\_t定义如下：

typedef struct condvar{

semaphore\_t sem; // the sem semaphore is used to down the waiting proc, and the signaling proc should up the waiting proc

int count; // the number of waiters on condvar

monitor\_t \* owner; // the owner(monitor) of this condvar

} 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。

理解了数据结构的含义后，我们就可以开始管程的实现了。ucore设计实现了条件变量wait\_cv操作和signal\_cv操作对应的具体函数，即cond\_wait函数和cond\_signal函数，此外还有cond\_init初始化函数。

cv.count++;//cond\_wait实现

if(monitor.next\_count > 0)

sem\_signal(monitor.next);

else

sem\_signal(monitor.mutex);

sem\_wait(cv.sem);

cv.count -- ;

if( cv.count > 0) {//cond\_signal实现

monitor.next\_count ++;

sem\_signal(cv.sem);

sem\_wait(monitor.next);

monitor.next\_count -- ;

}

简单分析一下cond\_wait函数的实现。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，可继续执行了！这里隐含这一个现象，即某进程A在时间顺序上先执行了signal\_cv，而另一个进程B后执行了wait\_cv，这会导致进程A没有起到唤醒进程B的作用。这里还隐藏这一个问题，在cond\_wait有sem\_signal(mutex)，但没有看到哪里sem\_wait(mutex)，这好像没有成对出现，是否是错误的？其实在管程中的每一个函数的入口处会有wait(mutex)，这样二者就配好对了。

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！

对照着再来看cond\_signal的实现。首先进程B判断cv.count，如果不大于0，则表示当前没有执行cond\_wait而睡眠的进程，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，这表示当前有执行cond\_wait而睡眠的进程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。

为了让整个管程正常运行，还需在管程中的每个函数的入口和出口增加相关操作，即：

function （…）

{

sem.wait(monitor.mutex);

the real body of function;

if(monitor.next\_count > 0)

sem\_signal(monitor.next);

else

sem\_signal(monitor.mutex);

}

这样做的作用有两个，（1）只有一个进程在执行管程中的函数。（2）避免由于执行了cond\_signal函数而睡眠的进程无法被唤醒。对于第二点，如果进程A由于执行了cond\_signal函数而睡眠（这会让monitor.next\_count大于0，且执行sem\_wait(monitor.next)），则其他进程在执行管程中的函数的出口，会判断monitor.next\_count是否大于0，如果大于0，则执行sem\_signal(monitor.next)，从而执行了cond\_signal函数而睡眠的进程被唤醒。上诉措施将使得管程正常执行。

**2、基于管程的条件变量的实现**

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);

/\*

\* cond\_signal(cv) {

\* if(cv.count>0) {

\* mt.next\_count ++;

\* signal(cv.sem);

\* wait(mt.next);

\* mt.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);

}

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);

/\*

\* cv.count ++;

\* if(mt.next\_count>0)

\* signal(mt.next)

\* else

\* signal(mt.mutex);

\* wait(cv.sem);

\* cv.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);

}

**3、基于管程的哲学家就餐问题**

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

down(&(mtp->mutex));//进入临界区

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

// LAB7 EXERCISE1: YOUR CODE

// I am hungry

// try to get fork

// I am hungry

state\_condvar[i]=HUNGRY; //记录下哲学家i饥饿的事实

// try to get fork

phi\_test\_condvar(i);

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

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

cond\_wait(&mtp->cv[i]);//如果得不到叉子就阻塞

}

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

if(mtp->next\_count>0)//如果阻塞则唤醒

up(&(mtp->next));

else

up(&(mtp->mutex));//离开临界区

}

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

down(&(mtp->mutex));//进入临界区

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

// LAB7 EXERCISE1: YOUR CODE

// I ate over

// test left and right neighbors

// I ate over

state\_condvar[i]=THINKING;//哲学家进餐结束

// test left and right neighbors

phi\_test\_condvar(LEFT);//看一下左邻居现在是否能进餐

phi\_test\_condvar(RIGHT);//看一下右邻居现在是否能进餐

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

if(mtp->next\_count>0)<span style="line-height: 25.6000003814697px; font-family: 'Open Sans', 'Clear Sans', 'Helvetica Neue', Helvetica, Arial, sans-serif;">//如果存在阻塞则唤醒</span>

up(&(mtp->next));

else

up(&(mtp->mutex));//离开临界区

