**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？**

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：

1. 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。
2. 硬件方法：
3. 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。
4. 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。
5. 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。

1. 管程机制：

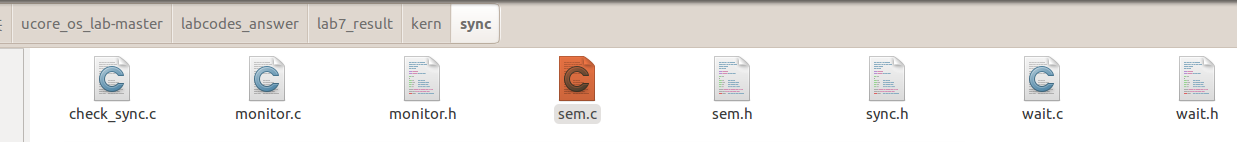
管程是一种用于多线程互斥访问共享资源的程序结，对于请求访问共享资源的诸多并发进程，可以根据资源的情况接受或阻塞，确保每次仅有一个进程进入管程。执行这组过程，达到对共享资源所有访问的统一管理，有效的实现进程互斥。

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。

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。**

信号量实现在这个文件里。其中c , wait.c与它有关。

****

其中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：

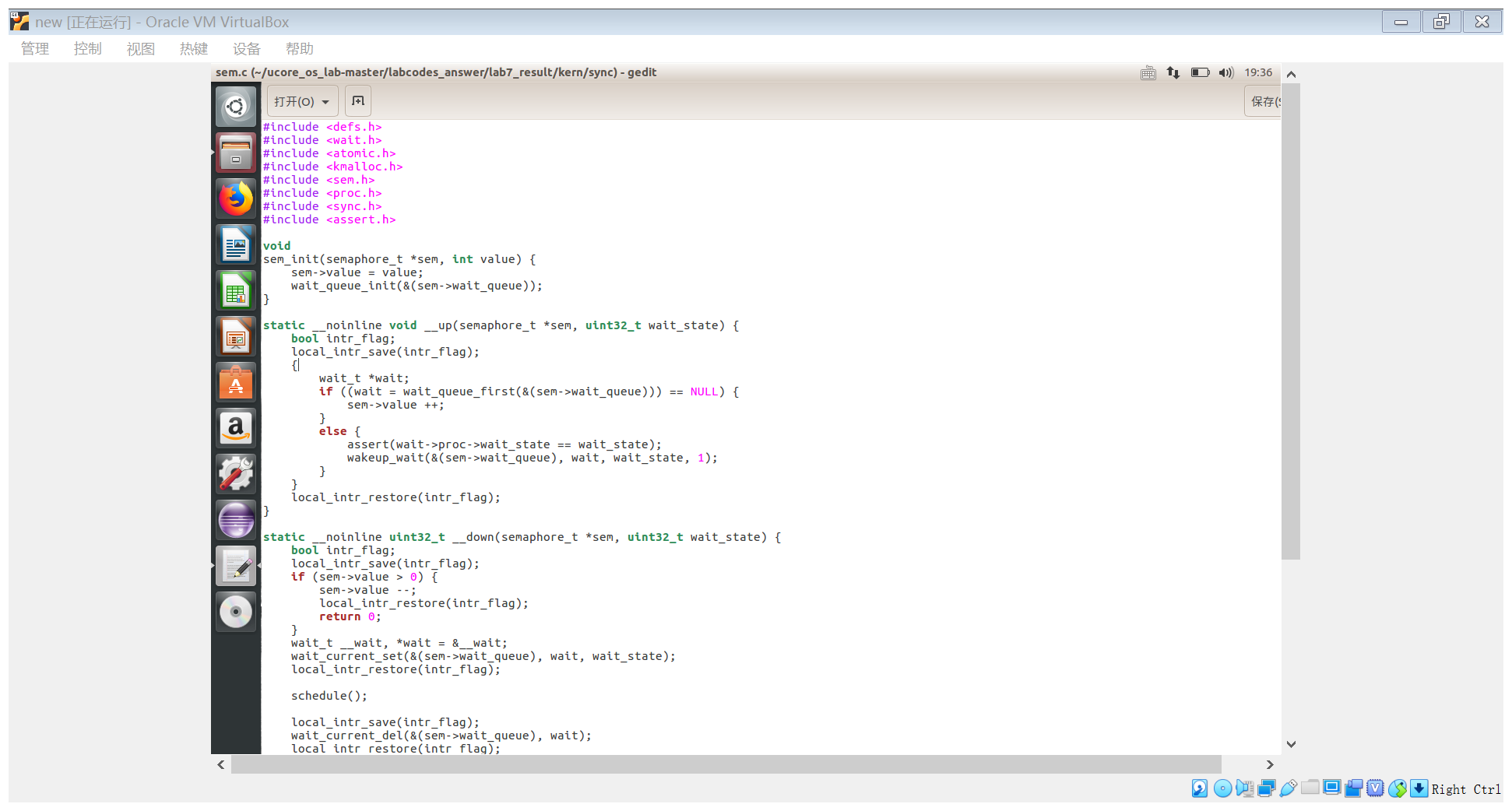
wait\_queue\_init(wait\_queue\_t \*queue)初始化等待队列

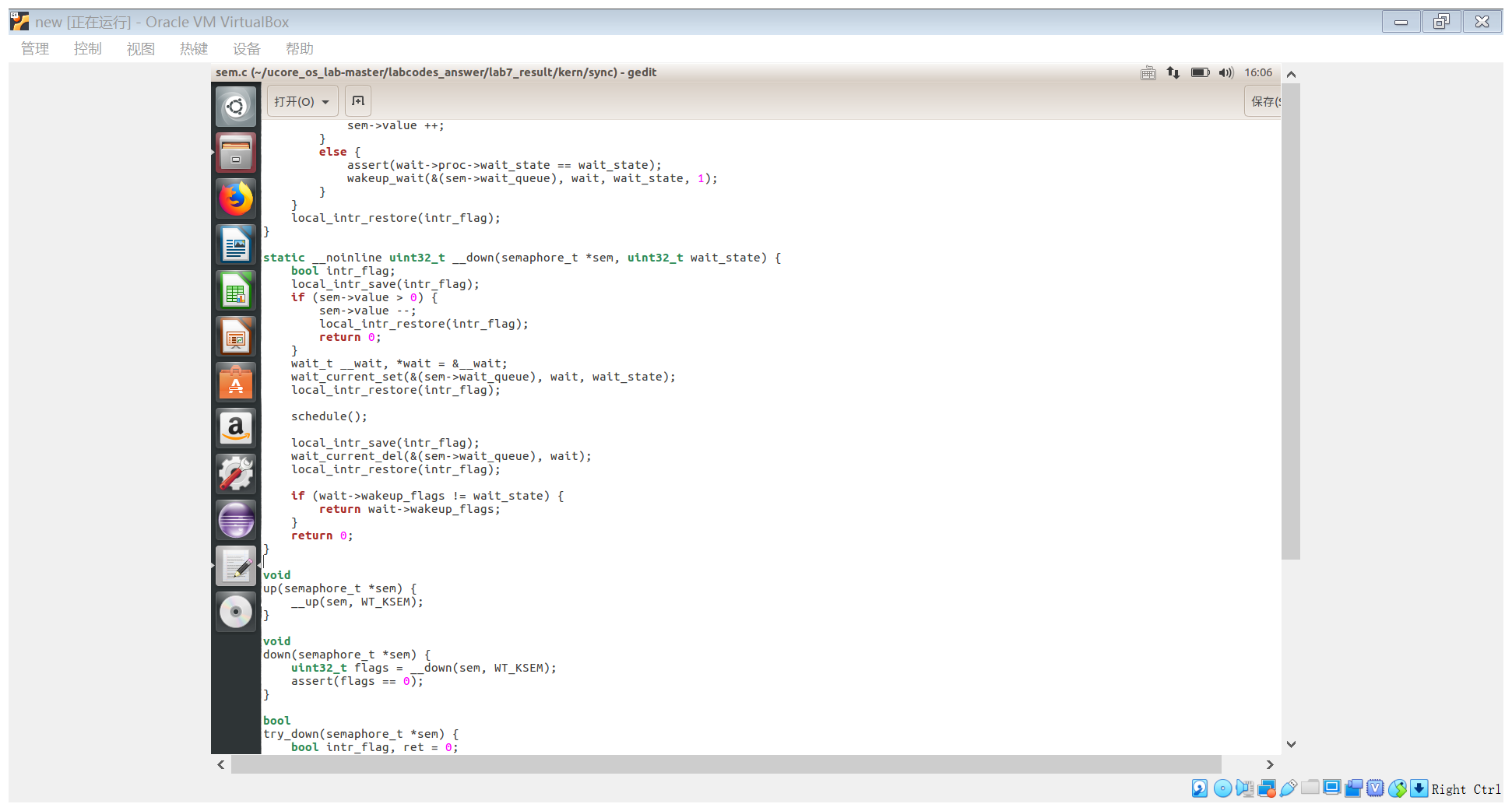
wait\_queue\_add(wait\_queue\_t \*queue, wait\_t \*wait)由于互斥资源被访问，当前进程便加入等待队列中

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。

首先，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。





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）。

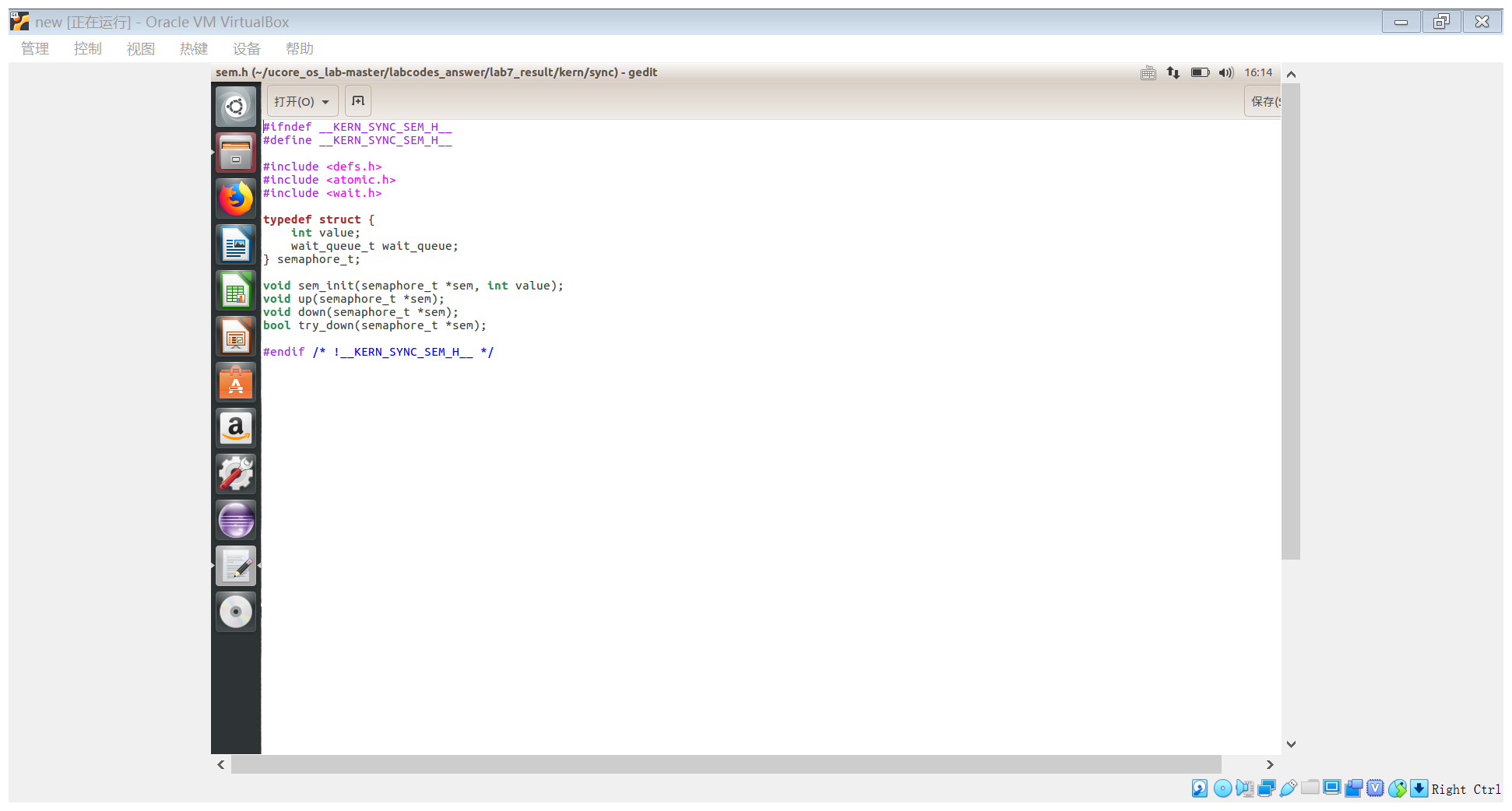
V操作的实现代码：

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。



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。**

在sem.h文件中，定义了信号量的数据结构，如图：



函数的设计方法见本实验报告练习1第二小问。

**练习2: 了解基于信号量和管程的哲学家就餐问题**

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，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。

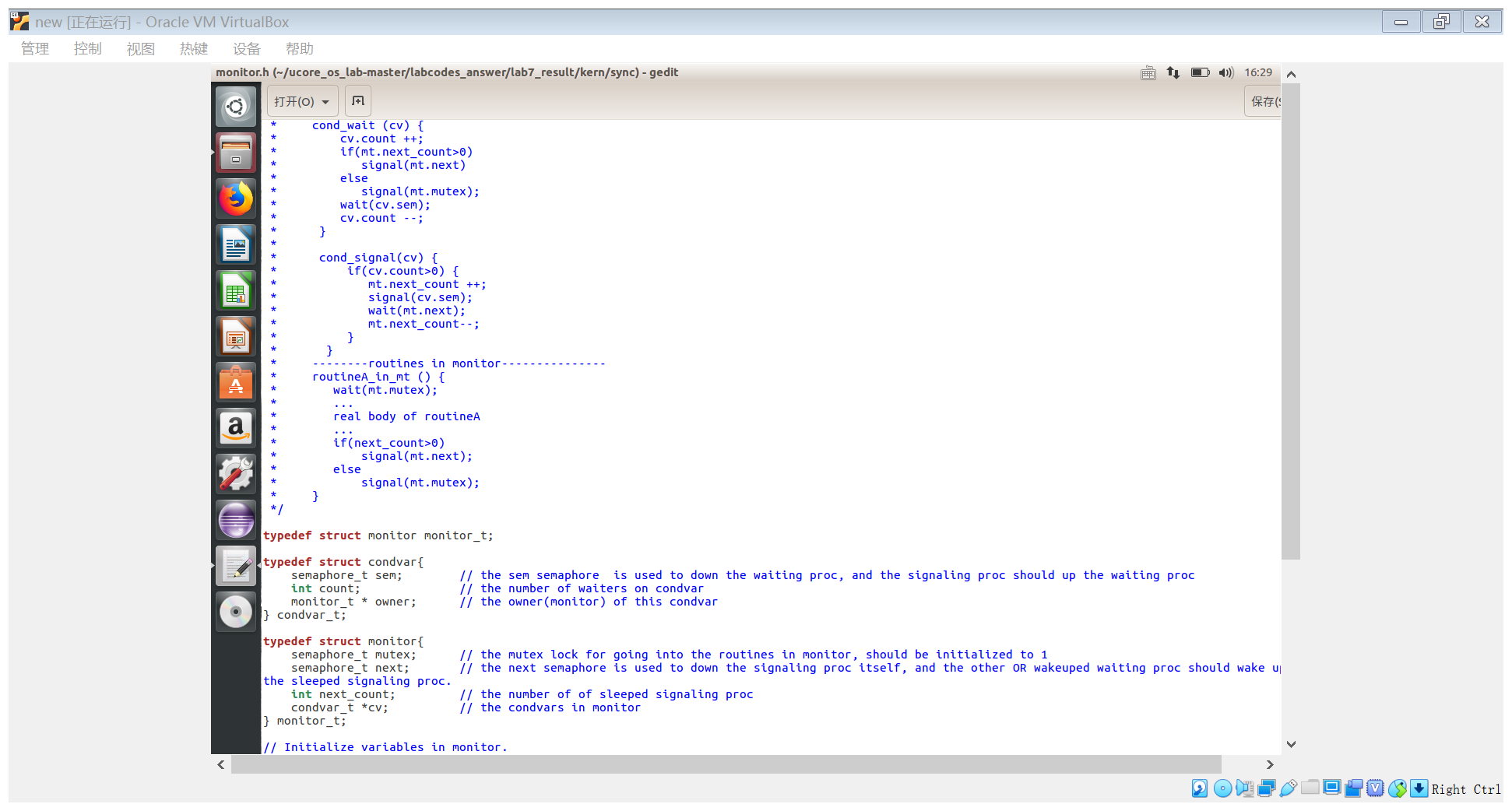
但在管程中仅仅有互斥操作是不够用的。进程可能需要等待某个条件C为真才能继续执行。可引入条件变量CV。因此对条件变量CV有两种主要操作：

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：



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。条件变量的数据结构是condvar\_t。

1. **说明ucore中基于信号量的哲学家就餐问题的实现机制。**

/---------- philosophers problem using semaphore ----------------------

int state\_sema[N]; /\* 记录每个人状态的数组 \*/

/\* 信号量是一个特殊的整型变量 \*/

semaphore\_t mutex; /\* 临界区互斥 \*/

semaphore\_t s[N]; /\* 每个哲学家一个信号量 \*/

struct proc\_struct \*philosopher\_proc\_sema[N];

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]);

}

}

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 \*/

}

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;

}

1. **说明ucore中基于管程的哲学家就餐问题的实现机制。**

/---------- philosophers using monitor (condition variable) ----------------------

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;

}

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");

}

//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");

}

}