# lab7 report



## 练习0：填写已有实验

使用`meld`填写已有实验。

对之前实验代码的修改：

### `trap.c:trap_dispatch`

修改时钟中断的处理：调用`run_timer_list`

```
case IRQ_OFFSET + IRQ_TIMER:
    ticks ++;
    run_timer_list();
    break;
```

## 练习1: 理解内核级信号量的实现和基于内核级信号量的哲学家就 餐问题(不需要编码)

### lab6, lab7的区别

1. 在`sched.c`中，lab7加入了对定时器的支持。
2. 在`trap.c`中，lab7 将原来时钟中断时设置需要调度标准修改为调用`run_timer_list()`函

数。
3. 加入了对 sleep 系统调用的支持。
4. 加入了对 信号量和管程 的支持。

## 内核级信号量

信号量的数据结构定义如下:

**semaphore_t**

```
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。信号量的核心操作 `P()`，`V()` 对应 `__down()` 和 `__up()` 函数。

**__down()**

```
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;
}
```

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）。

**__up()**

```
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);
}
```

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。

对照信号量的原理性描述和具体实现，可以发现二者在流程上基本一致，只是具体实现采用了关中断的方式保证了对共享资源的互斥访问，通过等待队列让无法获得信号量的进程睡眠等待。另外，我们可以看出信号量的计数器value具有有如下性质：

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

### 给用户态进程/线程提供信号量机制

内核级信号量的实现中通过开关中断保证P V操作的原子性，而在用户态无法直接执行，需要通过系统调用。可将开关中断部分替换为相应的系统调用给用户态进程/线程提供信号量机制。相同点是机制相同，不同点是用户态需要系统调用。

## 练习2: 完成内核级条件变量和基于内核级条件变量的哲学家就餐问题(需要编码)

### 内核级条件变量

#### monitor_t

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。

**condvar_t**

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。

**cond_wait**

```
void
cond_wait (condvar_t *cvp) {
    //LAB7 EXERCISE1: 2014011336
    cprintf("cond_wait begin:  cvp %x, cvp->count %d, cvp->owner->next_count %d\n", cvp, cvp->count, cvp->owner->next_count);
    cvp->count++;
    if(cvp->owner->next_count>0){
        // wake up proc sleeping because signal cv
        up(&(cvp->owner->next));
    }else{
        // wake up proc wait for enter moniter
        up(&(cvp->owner->mutex));
    }
    // fall asleep
    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);
}
```

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)，但没有看到哪里有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！

**cond_signal**

```
void
cond_signal (condvar_t *cvp) {
   //LAB7 EXERCISE1: 2014011336
   cprintf("cond_signal begin: cvp %x, cvp->count %d, cvp->owner->next_count %d\n", cvp, cvp->count, cvp->owner->next_count);
    if(cvp->count>0){
        cvp->owner->next_count++;
        // wake up proc wait for cv
        up(&(cvp->sem));
        // sleep to yield moniter
        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);
}
```

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。

**function in monitor**

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);
//--------into routine in monitor--------------

  the real body of function;

//--------leave routine in monitor--------------
  if(monitor.next_count > 0)
     sem_signal(monitor.next);
  else
     sem_signal(monitor.mutex);
}
```

这样带来的作用有两个，（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。

## 哲学家就餐问题

基于管程实现哲学家就餐问题。每个哲学家对应一个条件变量。

### phi_test_condvar

由拿叉子和放叉子的函数调用，是一个简单的封装函数。测试第 i 个哲学家能否就餐，如果左右空闲且本身饥饿，则允许就餐，发出对应条件变量准备好了的信号并转入 eating 状态。

```
void phi_test_condvar (i) {
    if(state_condvar[i]==HUNGRY&&state_condvar[LEFT]!=EATING
            &&state_condvar[RIGHT]!=EATING) {
        cprintf("phi_test_condvar: state_condvar[%d] will eating\n",i);
        state_condvar[i] = EATING ;
        cprintf("phi_test_condvar: signal self_cv[%d] \n",i);
        cond_signal(&mtp->cv[i]) ;
    }
}
```

### phi_take_forks_condvar

拿叉子的函数。函数的入口和出口增加相关操作以保证管程正常执行。先设置状态为饥饿，看看能不能得到两把叉子(phi_test_condvar)，如果不能则进入阻塞状态。

```
void phi_take_forks_condvar(int i) {
     down(&(mtp->mutex));
//--------into routine in monitor--------------
     // LAB7 EXERCISE1: 2014011336
     // I am hungry
     // try to get fork
    state_condvar[i] = HUNGRY;
    phi_test_condvar(i);
    if(state_condvar[i]!=EATING){
        cond_wait(&mtp->cv[i]);
    }
//--------leave routine in monitor--------------
      if(mtp->next_count>0)
         up(&(mtp->next));
      else
         up(&(mtp->mutex));
}
```

### phi_put_forks_condvar

放叉子的函数。先设置状态为思考，然后通知左右的哲学家试着去就餐。如果可以就餐，该哲学家将从`phi_take_forks_condvar`函数中的`cond_wait`出来，等待进入管程执行。

```
void phi_put_forks_condvar(int i) {
     down(&(mtp->mutex));

//--------into routine in monitor--------------
     // LAB7 EXERCISE1: YOUR CODE
     // I ate over
     // test left and right neighbors
     state_condvar[i] = THINKING;
     phi_test_condvar(LEFT);
     phi_test_condvar(RIGHT);
//--------leave routine in monitor--------------
     if(mtp->next_count>0)
        up(&(mtp->next));
     else
        up(&(mtp->mutex));
}
```

### 给出给用户态进程/线程提供条件变量机制

与信号量类似，需要将信号量中的开关中断改为系统调用。

### 能否不用基于信号量机制来完成条件变量?

能，条件变量实际上就是等待队列和对应的一把锁。信号量只不过是一种封装。

## 与参考答案的区别

### 练习2

基本相同

## 知识点

### 列出你认为本实验中重要的知识点，以及与对应的OS原理中的知识点，并简要说明你对二者的含义，关系，差异等方面的理解（也可能出现实验中的知识点没有对应的原理知识点）

- 信号量的原理与实现。实验中有对等待队列的详细操作。
- 管程的原理与实现。条件变量的控制操作。基于信号量实现条件变量。
- 哲学家就餐问题的信号量和管程实现。

### 列出你认为OS原理中很重要，但在实验中没有对应上的知识点

- 信号量和管程的底层支撑，虽有介绍，但如果有实验内容就更好了。