Lab7实验报告

实验目的

- 熟悉ucore中的进程同步机制，了解操作系统为进程同步提供的底层支持；
- 在ucore中理解信号量（semaphore）机制的具体实现；
- 理解管程机制，在ucore内核中增加基于管程（monitor）的条件变量（condition variable）的支持；
- 了解经典进程同步问题，并能使用同步机制解决进程同步问题。

实验内容

练习0：已有实验代码改进

1.trap_dispatch

计时器是操作系统基础而重要的功能。它提供了基于时间事件的调度机制。在计时器的基础上，操作系统可以实现基于时间长度的等待和唤醒机制。在每个时钟中断发生时，操作系统产生对应的时间事件。应用程序或者操作系统的其他组件可以以此来构建更复杂和高级的调度。sched.h, sched.c 定义了有关timer的各种相关接口来使用 timer 服务，包括定时器初始化，向系统添加定时器，删除定时器等。在每次时钟中断时，需要调用run_timer_list更新当前系统时间点，遍历当前所有处在系统管理内的定时器，找出所有应该激活的定时器，并激活它们。定时器的使用方式大致如下：

- timer_t 在某个位置被创建和初始化，并通过 add_timer加入系统管理列表中
- 系统时间被不断累加，直到 run_timer_list 发现该 timer_t到期。
- run_timer_list更改对应的进程状态，并从系统管理列表中移除该timer_t

在每次时钟中断时，添加run_timer_list的调用，原来的时间片-1和判断是否需要调度的工作在run_timer_list中完成。

```
static void
trap_dispatch(struct trapframe *tf) {
    ......
    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

1.同步互斥的底层支持

开关中断

在ucore中，为了实现同步互斥，提供了开关中断的机制。在kern/sync.c中实现了开关中断的控制函数local_intr_save(x)和local_intr_restore(x)，它们是基于kern/driver文件下的intr_enable()、intr_disable()函数实现的。具体调用关系为：

```
关中断: local_intr_save --> __intr_save --> intr_disable --> cli
开中断: local_intr_restore--> __intr_restore --> intr_enable --> sti
```

最终的cli和sti是x86的机器指令，实现了关中断和开中断，即设置了eflags寄存器中与中断相关的位。通过关闭中断，可以防止对当前执行的控制流被其他中断事件处理所打断。既然不能中断，那也就意味着在内核运行的当前进程无法被打断或被从新调度，即实现了对临界区的互斥操作。所以在单处理器情况下，可以通过开关中断实现对临界区的互斥保护，需要互斥的临界区代码的一般写法为：

```
local_intr_save(intr_flag);
{
  临界区代码
}
local_intr_restore(intr_flag);
......
```

通过这种方式，操作系统就可简单地支持互斥操作了，开关中断本身是最基本的互斥。在本实验中，使用开关中断机制实现信号量等高层同步互斥原语。

等待队列

用户进程或内核线程可以转入休眠状态以等待某个特定事件，当该事件发生时这些进程能够被再次唤醒。内核实现这一功能的一个底层支持机制就是等待队列（wait queue），等待队列和每一个事件（睡眠结束、时钟到达、任务完成、资源可用等）联系起来。需要等待事件的进程在转入休眠状态后插入到等待队列中。当事件发生之后，内核遍历相应等待队列，唤醒休眠的用户进程或内核线程，并设置其状态为就绪状态，并将该进程从等待队列中清除。ucore在kern/sync/{ wait.h, wait.c }中实现了wait结构和wait queue结构以及相关函数），这是实现ucore中的信号量机制和条件变量机制的基础，进入wait queue的进程会被设为睡眠状态，直到被唤醒。每个wait结构记录了一个休眠状态的进程。

```
typedef  struct {
    struct proc_struct *proc;              //等待进程的指针
    uint32_t wakeup_flags;                 //进程被放入等待队列的原因标记
    wait_queue_t *wait_queue;              //指向此wait结构所属于的wait_queue
    list_entry_t wait_link;                //用来组织wait_queue中wait节点的连接
} wait_t;
typedef struct {
    list_entry_t wait_head;                //wait_queue的队头
} wait_queue_t;
le2wait(le, member)                        //实现wait_t中成员的指针向wait_t 指针的转化
```

与wait和wait queue相关的函数主要分为两层，底层函数是对wait queue的初始化、插入、删除和查找操作。上层函数可以直接将进程加入等待队列，从等待队列中唤醒进程等。相关的函数如下：

```
//让wait与进程关联，且让当前进程关联的wait进入等待队列queue，当前进程睡眠
void wait_current_set(wait_queue_t *queue, wait_t *wait, uint32_t wait_state);
//把当前进程从等待队列queue中删除
wait_current_del(queue, wait)                                          //宏定义
//唤醒与wait关联的进程
void wakeup_wait(wait_queue_t *queue, wait_t *wait, uint32_t wakeup_flags, bool del);
//唤醒等待队列上挂着的第一个wait所关联的进程
void wakeup_first(wait_queue_t *queue, uint32_t wakeup_flags, bool del);
//唤醒等待队列上所有的等待的进程
void wakeup_queue(wait_queue_t *queue, uint32_t wakeup_flags, bool del);
```

2.内核级信号量实现

信号量是一种同步互斥机制的实现，通常用于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。

ucore中信号量建立在开关中断机制和wait queue的基础上进行了具体实现。信号量的数据结构定义如下：

```
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。在ucore中最重要的信号量操作是P操作函数down(semaphore_t *sem)和V操作函数up(semaphore_t *sem)。这两个函数的具体实现是__down(semaphore_t *sem, uint32_t wait_state) 函数和__up(semaphore_t *sem, uint32_t wait_state)函数，二者的具体实现描述如下：

__down(semaphore_t *sem, uint32_t wait_state)：实现信号量的P操作，首先关闭中断，然后判断当前信号量的value是否大于0。>0则表明可以获得信号量，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，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）。具体实现如下所示：

```
static __noinline uint32_t __down(semaphore_t *sem, uint32_t wait_state) {
    bool intr_flag;
    local_intr_save(intr_flag);                             //关中断
    if (sem->value > 0) {
        sem->value --;
        local_intr_restore(intr_flag);                      //开中断
        return 0;                                           //>0直接返回
    }
    wait_t __wait, *wait = &__wait;
    wait_current_set(&(sem->wait_queue), wait, wait_state); //将当前进程加入等待队列，设置睡眠状态
    local_intr_restore(intr_flag);                          //打开中断并调度其他进程
    schedule();
    /* 被唤醒 */
    local_intr_save(intr_flag);
    wait_current_del(&(sem->wait_queue), wait);             //从等待队列删除当前进程
```

```
    local_intr_restore(intr_flag);

    if (wait->wakeup_flags != wait_state) {
        return wait->wakeup_flags;                          //不是信号量导致的休眠则返回休眠原因
    }
    return 0;
}
```

__up(semaphore_t *sem, uint32_t wait_state)：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，然后打开中断返回；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。具体实现如下所示：

```
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);       //唤醒一个等待在该信号上的进程
        }
    }
    local_intr_restore(intr_flag);
}
```

这两个函数实现了信号量上进程的唤醒和睡眠操作，唤醒是将进程设置为可运行状态并加入运行队列，而睡眠则是将当前进程设置为睡眠状态，这样该进程将不会被调度，直到被唤醒。

3.基于内核级信号量的哲学家就餐问题

哲学家就餐问题

哲学家就餐问题的情景是这样的：有五个哲学家围着一个圆桌，每两个哲学家之间有一把餐叉。哲学家有时思考，不需要餐叉，有时就餐，需要得到左右两把餐叉。由餐叉带来的竞争和同步问题称为哲学家就餐问题。解决该问题的目标是保证没有死锁和饥饿的情况发生，且尽可能让更多哲学家吃到东西。本次实验需要分别使用信号量和条件变量解决这个问题，信号量的实现已经完成了，只需要理解。该问题的相关代码在check_sync.c中。

首先是check_sync函数，init_main中会调用该函数，哲学家就餐问题将在这个函数中进行测试和解决。先关注前半部分，即基于内核级信号量的哲学家就餐问题。该函数中调用kernel_thread创建了5个内核线程，对应5位哲学家，并初始化了一个信号量mutex为1，五个信号量为0。

```
#define N 5 /* 哲学家数目 */
void check_sync(void){
    int i;
    //check semaphore
    sem_init(&mutex, 1);
    for(i=0;i<N;i++){
```

```
        sem_init(&s[i], 0);                                             //使用5个信号量
        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");
    }
    ......
```

接下来分析哲学家就餐问题的主程序，philosopher_using_semaphore，参数为i，即哲学家编号。在这段程序中可以看出，每一位哲学家4次就餐，每次就餐的过程为：先进行思考，然后尝试拿起两只叉子，拿到叉子后就餐，最后再将叉子放回桌子。

```
#define TIMES  4 /* 吃4次饭 */
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;
}
```

其中就餐和思考的过程是通过do_sleep进行睡眠模拟的。do_sleep 会将当前进程的 state 设置为 SLEEPING，然后为进程创建一个定时器，并添加到定时器列表中，最后调用 schedule 让出 CPU 给其他进程。

```
int
do_sleep(unsigned int time) {
    if (time == 0) {
        return 0;
    }
    bool intr_flag;
    local_intr_save(intr_flag);
    timer_t __timer, *timer = timer_init(&__timer, current, time);
    current->state = PROC_SLEEPING;
    current->wait_state = WT_TIMER;
    add_timer(timer);                                                   //添加定时器
    local_intr_restore(intr_flag);
    schedule();                                                         //调度其他进程运行
    del_timer(timer);
    return 0;
```

```
}
```

哲学家就餐问题中主要需要解决的是餐叉的竞争，最关键的两个操作就是取得餐叉和放下餐叉。phi_take_forks_sema(i)实现了获取餐叉的过程。由于此处需要对哲学家的状态进行修改，而哲学家状态的列表是全局数据，因此需要使用mutex保护临界区。测试是否能拿到叉子调用了phi_test_sema，如果左右都不为就餐状态，说明叉子可获取，将状态设置为就餐状态，此时该哲学家状态确定了，会调用up让信号量+1，返回并离开临界区。然后将调用down，将信号量-1，如果刚才已经检查确认可以就餐，此时信号量值应该为0，可以进行就餐并返回，否则将阻塞在自己的信号量上，直到左右的哲学家确认他可以就餐，将他的信号量+1并将其唤醒。

```
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_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;       //将i状态修改，i拿起了叉子并可以就餐
        up(&s[i]);                  //信号量+1
    }
}
```

phi_put_forks_sema函数实现的则是放下叉子的过程。同样的，临界区需要使用mutex进行保护，进餐结束只需要将自己的状态修改为思考，然后测试左右的哲学家是否可以就餐，如果可以则将唤醒左右的哲学家。

```
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 */
}
```

4.用户态进程/线程信号量机制

**用户级信号量和内核级信号量的区别异同

用户级信号量和内核级信号量的区别在于内核级信号量是直接在内核态下使用的，因此信号量可以直接通过开关中断的方式实现。而用户态下，不能执行开关中断的特权指令，不可以直接使用内核级别的信号量，因此需要通过系统调用的方式实现信号量的使用。但具体的原理是相同的，都是通过__down和__up函数完成信号量的PV操作。

用户级信号量机制实现

为了使用户可以使用信号量，操作系统需要提供信号量相关的系统调用，用户进程通过系统调用进入内核态，然后调用内核级信号量的相关函数，完成信号量相关操作。为用户进程提供的系统调用至少有以下三个：

- sem_init：生成一个信号量，并初始化值。
- sem_post：V操作，可以直接调用up函数完成

- sem_wait：P操作，可以直接调用down函数完成

此外需要注意的是，用户级信号量应该存放在共享内存中，一个进程的线程都可以访问到信号量。

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

1.管程机制

管程是为了将对共享资源的所有访问及其所需要的同步操作集中并封装起来，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。一个管程由四部分构成：

- 管程内部的共享变量
- 管程内部的条件变量
- 管程内部并发执行的进程
- 对局部于管程内部的共享数据设置初始值的语句

管程的作用相当于将共享变量，涉及该共享变量的进程，互斥量及操作全部封装起来进行管理。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。很多情况下，进程需要等待某个条件满足后才会继续运行。为了实现这种等待，管程需要提供条件变量。最终一个管程的结构定义如下：

```
typedef struct monitor{
    semaphore_t mutex;      //保证对管程的访问是互斥的锁
    semaphore_t next;       //进程同步需要的信号量
    int next_count;         //睡眠进程数
    condvar_t *cv;          //条件变量，提供条件等待
} monitor_t;
```

管程中的条件变量cv通过执行cond_wait，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cond_signal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。信号量next和整形变量next_count用于配合cv操作。由于管程只有一个进程可以进入，当前进程cond_signal唤醒了等待状态睡眠的进程后，将睡眠在管程的next上，当有进程再次等待条件时，再将该进程唤醒。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cond_signal而睡眠的进程个数。

只考虑两个进程进入管程的情况，大致的过程如下：

进程1获取管程的锁->

条件不满足，进程1调用cond_wait释放锁在条件变量上等待(进程1在管程内)->

进程2获取管程的锁->

条件达成，进程2cond_signal唤醒进程1，进程2等待在管程的next上(进程2在管程内)->

进程1从cond_wait返回，完成工作后离开管程，发现next有进程等待，唤醒进程2，不释放锁(传递锁)->

进程2从cond_signal返回，完成工作后离开管程，next没有进程等待，释放管程锁

整个过程中，只有一个进程在管程内，对共享变量进行访问，并通过条件变量实现了条件等待。其中比较重要的部分是：发出cond_signal的进程，需要等待另一个进程醒来并离开管程才可以运行，因此等待在管程的信号量next上。这种等待与条件等待是不同的。进入管程时，是由mutex保证只有一个进程可以进入管程，但是如果一个进程在管程内cond_wait等待，释放了锁，就可能会有另一个进程进入管程，**两个进程不能同时运行，必须有一个在等待**，因此管程中才提供了信号量next，保证管程内只有一个进程运行。

2.条件变量

管程中的条件变量定义如下：

```
typedef struct condvar{
    semaphore_t sem;        //信号量实现条件变量
    int count;              //等待在该条件变量的进程数
    monitor_t * owner;      //该条件变量对应的管程
} condvar_t;
```

通过条件变量中的信号量，可以使等待某个特定条件的进程睡眠，而另一个完成该特定条件的进程可以将其唤醒。其中等待条件变量的操作通过调用cond_wait完成。在cond_wait中，首先要查看next上是否有进程在睡眠，这些进程因发出cond_signal而睡眠，当前进程需要唤醒其中一个进程。如果没有则唤醒一个唤醒因为互斥条件mutex无法进入管程的进程。然后自己等待在条件变量上，直到条件满足，被cond_signal唤醒。

```
void
cond_wait (condvar_t *cvp) {
    cprintf("cond_wait begin:  cvp %x, cvp->count %d, cvp->owner->next_count %d\n", cvp, cvp->count,
cvp->owner->next_count);
    cvp->count++;                                       //等待在条件变量上的进程数+1
    if(cvp->owner->next_count > 0)                      //如果有则唤醒next上的进程，锁传递
        up(&(cvp->owner->next));
    else
        up(&(cvp->owner->mutex));                       //没有则释放锁
    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);
}
```

cond_signal负责唤醒进程。调用up唤醒一个等待在条件变量上的进程，自己则调用down进行睡眠，等待唤醒。

```
void
cond_signal (condvar_t *cvp) {
   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){                                          //判断是否有进程等待
       cvp->owner->next_count ++;                             //next_count++，自己将要睡眠
       up(&(cvp->sem));                                       //唤醒一个在条件变量等待的进程
       down(&(cvp->owner->next));                             //自己睡眠，等待同步
       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);
}
```

3.条件变量实现的哲学家就餐问题

check_sync的后半部分为测试条件变量实现的哲学家就餐问题。与测试信号量实现的哲学家就餐问题相同，创建五个线程，调用对应的主函数。不同的地方在于与信号量实现中需要每个哲学家定义一个信号量相比，只需要创建并初始化一个管程，在管程中将保存需要用到的五个条件变量。

```
    ......
    //check condition variable
    monitor_init(&mt, N);                                     //使用管程mt完成哲学家就餐问题
    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");
    }
}
```

就餐过程与信号量实现的哲学家就餐问题基本相同，只有获取餐叉和放下餐叉使用的是条件变量实现的函数。

```
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);                                          //使用条件变量的拿起叉子
        /* 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);                                           //使用条件变量的放下叉子
        /* return two forks back*/
    }
    cprintf("No.%d philosopher_condvar quit\n",i);
```

```
    return 0;
}
```

拿起和放下叉子的两个函数是本练习的内容，首先完成拿起叉子的phi_take_forks_condvar函数。进出管程需要获取锁和释放锁。在持有锁的情况下调用phi_test_condvar尝试获取叉子，如果获取到了叉子，状态将被修改为就餐状态，否则状态不变，则需要通过cond_wait睡眠在对应的条件变量上。

```
//获取叉子，获取不到则将等待
void phi_take_forks_condvar(int i) {
     down(&(mtp->mutex));
//--------into routine in monitor--------------
     state_condvar[i] = HUNGRY;
     phi_test_condvar(i);
     if(state_condvar != EATING)               //不使用while也没有问题，因为状态只有自己能改回thinking
        cond_wait(&mtp->cv[i]);
//--------leave routine in monitor--------------
      if(mtp->next_count>0)
         up(&(mtp->next));
      else
         up(&(mtp->mutex));                    //释放锁
}
//尝试获取叉子，根据状态判断是否可以获取
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]) ;
    }
}
```

放下叉子调用phi_put_forks_condvar，将当前哲学家设置为THINKING状态，然后测试左右的哲学家是否可以获取叉子，如果可以获取，则将其使用cond_signal唤醒。

```
void phi_put_forks_condvar(int i) {
     down(&(mtp->mutex));
//--------into routine in monitor--------------
     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));
}
```

4.用户态进程/线程条件变量机制

与用户态下的信号量类似，用户态下的条件变量同样不可以在内核态下直接使用管程和条件变量实现。因为管程和条件变量也是依赖信号量，通过up和down函数中开关中断实现操作的原子性，在用户态下是不能使用的。因此，用户态下的条件变量机制也需要使用系统调用来完成。与信号量类似，至少需要三个系统调用接口：

- cond_init：创建并初始化条件变量
- cond_wait：使进程等待在条件变量上
- cond_signal：唤醒等待在条件变量的进程

类似于信号量，用户级条件变量应该存放在共享内存中，一个进程的线程都可以访问。

5.不基于信号量的条件变量机制

使用信号量来实现条件变量机制，实际上是使用了信号量的等待队列，并且使用了值来记录对资源的访问情况。由于与管程结合，直接使用信号量更加方便的实现了管程，使条件变量和锁配合使用。对于一个简单的，不与其他机制结合的条件变量，不需要复杂的实现，也不需要值来记录，只需要一个等待队列。即按照OSTEP书中的定义：条件变量是一个显示队列，当某些条件不满足时，线程或进程可以将自己加入队列，等待该条件；另外某个线程或进程，改变了上述条件时，就可以唤醒一个或多个等待线程。只需要一个等待队列，就可以实现一个简单的条件变量。而对条件变量的wait和signal操作，可以和信号量一样，使用开关中断的方式保证原子性。

实现简单的条件变量，首先进行定义，条件变量只需要作为一个显式队列，而与条件变量相关的操作有四个，cond_init对条件变量的队列进行初始化，cond_wait使当前进程等待，cond_signal唤醒一个等待进程，cond_broadcast唤醒所有等待进程。条件变量通常是与锁配合使用的，在进入临界区时会获取锁，如果调用cond_wait进行等待，则需要释放锁，因此cond_wait应该传入一个锁。但是由于实验中是使用开关中断实现了锁的功能，所以这里不传入锁，而是在传入标志中断开关的flag，从而打开中断。

```
//定义一个只有等待队列的条件变量cond_t
typedef struct {
    wait_queue_t wait_queue;
} cond_t;
//相关操作
void cond_init(cond_t *cond );
void cond_wait(cond_t *cond,  bool intr_flag);
void cond_signal(cond *cond);
bool cond_broadcast(cond *cond);
```

接下来对相关函数进行实现，cond_init只需要初始化等待队列，等待和唤醒，仿照信号量的实现，定义__wait和__signal完成。并且向这些函数额外传入一个等待状态。等待状态是在proc.h中定义的，可以仿照定义一个WT_KCOND表示因内核条件变量等待。

```
#define WT_KCOND 0x00001000         //仿照WT_KSEM在proc.h中定义
//条件变量初始化
void
cond_init(cond_t *cond) {
    wait_queue_init(&(cond->wait_queue));
```

```
}
//等待与唤醒
void
cond_wait(cond_t *cond) {
    __wait(sem, WT_KCOND);
}
void
cond_signal(cond_t *cond) {
    __signal(sem, WT_KCOND, 0);     //0表示不广播唤醒所有进程
}
void
cond_broadcast(cond_t *cond) {
    __signal(sem, WT_KCOND, 1);     //广播唤醒所有进程
}
```

__wait的实现只需要将当前进程加入等待队列，加入队列之前需要释放锁。

```
static __noinline uint32_t __wait(cond_t *cond, bool intr_flag, uint32_t wait_state) {
    //现在是关闭中断的状态
    wait_t __wait, *wait = &__wait;
    wait_current_set(&(cond->wait_queue), wait, wait_state);    //让当前进程等待
    local_intr_restore(intr_flag);                              //开中断
    schedule();                                                 //调度其他进程运行
    local_intr_save(intr_flag);                                 //重新打开中断
    wait_current_del(&(sem->wait_queue), wait);
    if (wait->wakeup_flags != wait_state) {
        return wait->wakeup_flags;
    }
    return 0;                                                   //以关中断状态返回
}
```

__signal和__broadcast将等待在条件变量的等待队列上的进程唤醒，区别在于唤醒进程的数量。

```
static __noinline void __signal(cond_t *cond, uint32_t wait_state,  bool broadcast) {
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        wait_t *wait;
        while ((wait = wait_queue_first(&(cond->wait_queue))) != NULL) {    //不为空
            assert(wait->proc->wait_state == wait_state);
            wakeup_wait(&(cond->wait_queue), wait, wait_state, 1);
            if(!broadcast) break;                                           //只唤醒一个
        }
    }
    local_intr_restore(intr_flag);
}
```

这样就实现了一个最简单的条件变量，配合开关中断就可以使用了。

```
//使用条件变量，假设有一个等待条件done
//等待条件的进程：
    bool intr_flag;
    local_intr_save(intr_flag);                                         //相当于上锁
    {
```

```
        while(done == 0){
            cond_wait(&cond, intr_flag);
        }
        等待条件成立后执行的代码...
    }
    local_intr_restore(intr_flag)
//改变等待条件的进程
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        done = 1;
        cond_signal(&cond);
    }
    local_intr_restore(intr_flag)
```

以上实现的条件变量只是根据书中给出的定义所完成的基本实现，仅仅作为等待队列使用，真正的实现可能要更加复杂，例如ucore中的管程和条件变量结合的实现。因此这个条件变量实现只是辅助对条件变量使用和概念的理解，不能使用。但是显然的是，只要有等待队列就可以实现条件变量，不一定需要信号量才可以实现条件变量。

实验总结

与参考答案的比较

练习1

不需要编写代码，主要是理解ucore中使用开关中断实现互斥的方式以及信号量的实现，并理解哲学家就餐问题以及基于信号量的解决方案。

练习2

与答案一致。主要是根据注释完成条件变量的实现。由于对条件变量与管程的结合实现并不熟悉，代码部分主要是依靠注释完成，然后通过实验指导和条件变量以及管程的实现代码理解条件变量是怎样实现的。尤其是对管程及管程中配合条件变量的next信号量进行理解。哲学家就餐问题的基于条件变量的解决方法，与信号量解决方法是一致的。

涉及知识点

- 并发问题
- 信号量的实现与使用
- 条件变量的实现与使用
- 哲学家就餐问题的解决
- 管程机制

未涉及的知识点

- 虚拟内存
- 进程及进程调度
- 文件系统