

uCore实验七报告

黄郭斌 计科1502班 201507010206

一. 内容

实验六完成了用户进程的调度框架和具体的调度算法，可调度运行多个进程。如果多个进程需要协同操作或访问共享资源，则存在如何同步和有序竞争的问题。本次实验，主要是熟悉ucore的进程同步机制—信号量（semaphore）机制，以及基于信号量的哲学家就餐问题解决方案。然后掌握管程的概念和原理，并参考信号量机制，实现基于管程的条件变量机制和基于条件变量来解决哲学家就餐问题。在本次实验中，在kern/sync/check_sync.c中提供了一个基于信号量的哲学家就餐问题解法。同时还需完成练习，即实现基于管程（主要是灵活运用条件变量和互斥信号量）的哲学家就餐问题解法。哲学家就餐问题描述如下：有五个哲学家，他们的生活方式是交替地进行思考和进餐。哲学家们公用一张圆桌，周围放有五把椅子，每人坐一把。在圆桌上有五个碗和五根筷子，当一个哲学家思考时，他不与其他人交谈，饥饿时便试图取用其左、右最靠近他的筷子，但他可能一根都拿不到。只有在他拿到两根筷子时，方能进餐，进餐完后，放下筷子又继续思考。

二. 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。

三. 实验分析及结果

0. 练习零(填写已有实验)

练习0:填写已有实验

这里把需要填充的文件罗列如下：

```
proc.c
default_pmm.c
pmm.c
swap_fifo.c
vmm.c
trap.c
sche.c1234567
```

另外需要说明的是，上述需要填充的部分，不需要在以前的基础上再额外加以修改，直接讲lab1-lab6的代码根据对比复制过来即可。

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

在完成本练习之前，先说明下什么是哲学家就餐问题

哲学家就餐问题，即有五个哲学家，他们的生活方式是交替地进行思考和进餐。哲学家们公用一张圆桌，周围放有五把椅子，每人坐一把。在圆桌上有五个碗和五根筷子，当一个哲学家思考时，他不与其他人交谈，饥饿时便试图取用其左、右最靠近他的筷子，但他可能一根都拿不到。只有在他拿到两根筷子时，方能进餐，进餐完后，放下筷子又继续思考。

在分析之前先对信号量进行简介，直接看信号量的伪代码如下

```
struct semaphore {
    int count;
    queueType queue;
};

void P(semaphore S){
    S.count-- ;
    if (S.count<0) {
        把进程置为睡眠态；
        将进程的PCB插入到S.queue的队尾；
        调度，让出CPU；
    }
}

void V(semaphore S){
    S.count++ ;
    if (S.count≤0) {
        唤醒在S.queue上等待的第一个进程；
    }
}
```

基于上述信号量实现可以认为，当多个进程可以进行互斥或同步合作时，一个进程会由于无法满足信号量设置的某条件而在某一位置停止，直到它接收到一个特定的信号（表明条件满足了）。为了发信号，需要使用一个称作信号量的特殊变量。为通过信号量s传送信号，信号量通过V、P操作来修改传送信号量。接下来进入代码的分析。lab7和之前的lab6的总体步骤基本没有多大的变化，开始的执行流程都与实验六相同，而二者的差异主要是从，而我们跟着代码继续往下看，一直到创建第二个内核线程init_main时，我们可以看到，init_main的内容有一定的修改，函数在开始执行调度之前多执行了一个check_sync函数，check_sync函数如下：

```
void check_sync(void){

    int i;

    //check semaphore
    sem_init(&mutex, 1);
    for(i=0;i<N;i++){d
        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");
}
}

```

根据注释可以看到，该函数分为了两个部分，第一部分是实现基于信号量的哲学家问题，第二部分是实现基于管程的哲学家问题。练习1要求分析基于信号量的哲学家问题，这里我们先只看该函数的前半部分。首先实现初始化了一个互斥信号量，然后创建了对应5个哲学家行为的5个信号量，并创建5个内核线程代表5个哲学家，每个内核线程完成了基于信号量的哲学家吃饭睡觉思考行为实现。现在我们继续跟进philosopher_using_semaphore函数观察它的具体实现。

```

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;
}

```

看到核心就是phi_take_forks_sema和phi_put_forks_sema两个函数，具体的函数注释如下：

```

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 */
    }1234567891011121314151617

```

而这里到了信号量的核心部分，就是上述代码中的up和down函数就分别调用了__up函数和__down函数，而这两个函数分别对应着信号量的V，P操作。先看__up函数，它实现了信号量的V操作

```

static __noinline void __up(semaphore_t *sem, uint32_t wait_state) {
    bool intr_flag;
    local_intr_save(intr_flag); //关闭中断

    {
        wait_t *wait;
        if ((wait = wait_queue_first(&(sem->wait_queue))) == NULL) { //没有进程等待
            sem->value ++; //信号量的value加一
        }
        else { //有进程在等待
            assert(wait->proc->wait_state == wait_state);
            wakeup_wait(&(sem->wait_queue), wait, wait_state, 1); //将`wait_queue`中等待的第一个
            wait删除，并将该进程唤醒
        }
    }
    local_intr_restore(intr_flag); //开启中断返回
}12345678910111213141516

```

首先通过local_intr_save函数关闭中断，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，然后通过local_intr_restore函数开中断返回。如果有进程在等待且进程等待的原因是semaphore设置的，则调用wakeup_wait函数将wait_queue中等待的第一个wait删除，且把此wait关联的进程唤醒，最后通过local_intr_restore函数开中断返回。

再来看看__down函数，它实现了信号量的P操作

```

static __noinline uint32_t __down(semaphore_t *sem, uint32_t wait_state) {
    bool intr_flag;
    local_intr_save(intr_flag); //关掉中断
    if (sem->value > 0) { //当前信号量value大于0
        sem->value --; //直接让value减一
        local_intr_restore(intr_flag); //开中断返回
        return 0;
    }
    //当前信号量value小于等于0，表明无法获得信号量
    wait_t __wait, *wait = &__wait;
    wait_current_set(&(sem->wait_queue), wait, wait_state); //将当前的进程加入到等待队列中
    local_intr_restore(intr_flag); //打开中断

    schedule(); //运行调度器选择其他进程执行

    local_intr_save(intr_flag); //关中断
    wait_current_del(&(sem->wait_queue), wait); //被V操作唤醒，从等待队列移除

    local_intr_restore(intr_flag); //开中断
}

```

```
    if (wait->wakeup_flags != wait_state) {
        return wait->wakeup_flags;
    }
    return 0;
}123456789101112131415161718192021222324
```

首先关掉中断，然后判断当前信号量的value是否大于0。如果是大于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；如果小于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）

至此，基于信号量的哲学家问题的解决就分析完毕了。

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

即要求首先掌握管程机制,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。

管程，即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。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。管程主要由这四个部分组成

- 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。

所谓条件变量，即将等待队列和睡眠条件包装在一起，就形成了一种新的同步机制，称为条件变量。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 `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。

因而条件变量两种操作如下：- wait_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。- 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。

大概了解了原理之后，接下来我们开始分析具体的代码。ucore中的管程机制是基于信号量和条件变量来实现的。管程的数据结构monitor_t如下：

```
typedef struct monitor{
    semaphore_t mutex;        // 二值信号量，只允许一个进程进入管程，初始化为1
    semaphore_t next;         // 配合cv，用于进程同步操作的信号量
    int next_count;           // 睡眠的进程数量
    condvar_t *cv;           // 条件变量cv
} monitor_t;123456
```

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。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`signal_cv`而睡眠的进程个数。

条件变量`condvar_t`的数据结构如下：

```
typedef struct condvar{
    semaphore_t sem; //用于发出wait_cv操作的等待某个条件C为真的进程睡眠
    int count;        // 在这个条件变量上的睡眠进程的个数
    monitor_t * owner; // 此条件变量的宿主管程
} condvar_t;12345
```

分析完数据结构之后，我们开始分析管程的实现。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。先看看`cond_signal`函数，实现如下：

```
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) { //当前存在睡眠的进程
        cvp->owner->next_count ++; //睡眠的进程总数加一
        up(&(cvp->sem)); //唤醒等待在cv.sem上睡眠的进程
        down(&(cvp->owner->next)); //把自己睡眠
        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);
}1234567891011
```

首先进程`B`判断`cv.count`，如果不大于0，则表示当前没有睡眠的进程，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。

同样，再来看看`cond_wait`函数，实现如下：

```

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);
    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));//否则唤醒睡在monitor.mutex上的进程
    down(&(cvp->sem));//将自己睡眠
    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);
}12345678910111213

```

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。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！

这样我们就可以在此基础上继续完成哲学家就餐问题的解决了，主要就是就是如下的两个函数：

```

void phi_take_forks_condvar(int i) {
    down(&(mtp->mutex)); //通过P操作进入临界区
    state_condvar[i]=HUNGRY; //记录下哲学家i是否饥饿，即处于等待状态拿叉子
    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]);//如果得不到叉子就睡眠
    }
    //如果存在睡眠的进程则那么将之唤醒
    if(mtp->next_count>0)
        up(&(mtp->next));
    else
        up(&(mtp->mutex));
}

void phi_put_forks_condvar(int i) {
    down(&(mtp->mutex));//通过P操作进入临界区

    state_condvar[i]=THINKING;//记录进餐结束的状态
    phi_test_condvar(LEFT);//看一下左边哲学家现在是否能进餐
    phi_test_condvar(RIGHT);//看一下右边哲学家现在是否能进餐
    //如果有哲学家睡眠就予以唤醒
    if(mtp->next_count>0)
        up(&(mtp->next));
    else

```

```
up(&(mtp->mutex));
}123456789101112131415161718192021222324252627
```

至此，基于条件变量的哲学家就餐问题也得以解决。

1. 练习一(使用Round Robin调度算法)

完成练习0后，建议大家比较一下（可用kdiff3等文件比较软件）个人完成的lab5和练习0完成后的刚修改的lab6之间的区别，分析了解lab6采用RR调度算法后的执行过程。执行make grade，大部分测试用例应该通过。但执行priority.c应该过不去。请在实验报告中完成：请理解并分析sched_calss中各个函数指针的用法，并接合Round Robin 调度算法描述ucore的调度执行过程 请在实验报告中简要说明如何设计实现“多级反馈队列调度算法”，给出概要设计，鼓励给出详细设计

Round Robin 调度算法的调度思想是让所有 runnable 态的进程分时轮流使用 CPU 时间。Round Robin 调度器维护当前 runnable进程的有序运行队列。当前进程的时间片用完之后,调度器将当前进程放置到运行队列的尾部，再从其头部取出进程进行调度。

在这个理解的基础上，我们来分析算法的具体实现。这里 Round Robin 调度算法的主要实现在 default_sched.c 之中，源码如下：

```
static void
RR_init(struct run_queue *rq) {
    list_init(&(rq->run_list));
    rq->proc_num = 0;
}

static void
RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link));
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice;
    }
    proc->rq = rq;
    rq->proc_num ++;
}

static void
RR_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
    list_del_init(&(proc->run_link));
    rq->proc_num --;
}

static struct proc_struct *
RR_pick_next(struct run_queue *rq) {
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) {
        return le2proc(le, run_link);
    }
    return NULL;
}
```



```
static void
RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice --;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}

struct sched_class default_sched_class = {
    .name = "RR_scheduler",
    .init = RR_init,
    .enqueue = RR_enqueue,
    .dequeue = RR_dequeue,
    .pick_next = RR_pick_next,
    .proc_tick = RR_proc_tick,
};
```

现在我们来逐个函数的分析，从而了解 Round Robin 调度算法的原理。首先是 `RR_init` 函数，函数比较简单，不再罗列，完成了对进程队列的初始化。

然后是 `RR_enqueue` 函数，

```
static void RR_enqueue(struct run_queue *rq, struct proc_struct *proc) {
    assert(list_empty(&(proc->run_link)));
    list_add_before(&(rq->run_list), &(proc->run_link));
    if (proc->time_slice == 0 || proc->time_slice > rq->max_time_slice) {
        proc->time_slice = rq->max_time_slice;
    }
    proc->rq = rq;
    rq->proc_num ++;
}
```

看代码，首先，它把进程的进程控制块指针放入到rq队列末尾，且如果进程控制块的时间片为0，则需要把它重置为 `max_time_slice`。这表示如果进程在当前的执行时间片已经用完，需要等到下一次有机会运行时，才能再执行一段时间。然后在依次调整rq和rq的进程数目加一。

然后是 `RR_dequeue` 函数

```
static void
RR_dequeue(struct run_queue *rq, struct proc_struct *proc) {
    assert(!list_empty(&(proc->run_link)) && proc->rq == rq);
    list_del_init(&(proc->run_link));
    rq->proc_num --;
}
```

即简单的把就绪进程队列rq的进程控制块指针的队列元素删除，然后使就绪进程个数的proc_num减一。

接下来是 `RR_pick_next` 函数。

```
static struct proc_struct *RR_pick_next(struct run_queue *rq) {
    list_entry_t *le = list_next(&(rq->run_list));
    if (le != &(rq->run_list)) {
        return le2proc(le, run_link);
    }
    return NULL;
}
```

选取函数，即选取就绪进程队列rq中的队头队列元素，并把队列元素转换成进程控制块指针，即置为当前占用CPU的程序。

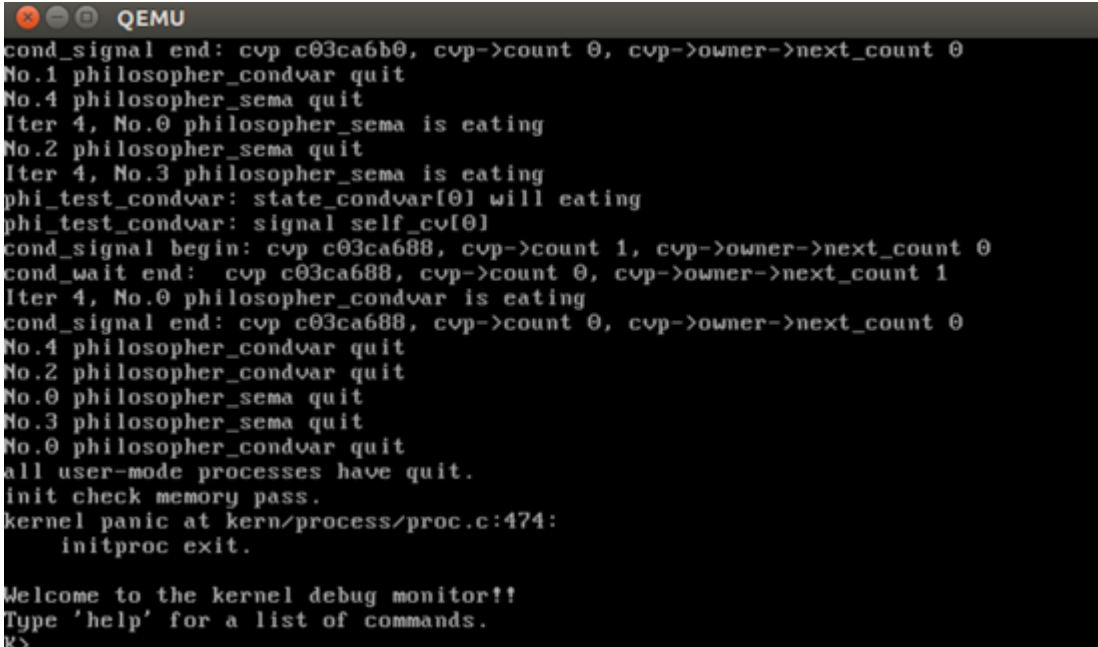
最后是

```
static void RR_proc_tick(struct run_queue *rq, struct proc_struct *proc) {
    if (proc->time_slice > 0) {
        proc->time_slice --;
    }
    if (proc->time_slice == 0) {
        proc->need_resched = 1;
    }
}
```

观察代码，即每一次时间片到时的时候，当前执行进程的时间片 `time_slice` 便减一。如果 `time_slice` 降到零，则设置此进程成员变量 `need_resched` 标识为1，这样在下次中断来后执行trap函数时，会由于当前进程成员变量 `need_resched` 标识为1而执行schedule函数，从而把当前执行进程放回就绪队列末尾，而从就绪队列头取出在就绪队列上等待时间最久的那个就绪进程执行。之后是一个对象化，提供一个类的实现，不是重点，这里不做赘述。

3. 运行结果

Make qemu之后的结果:



```
QEMU
cond_signal end: cvp c03ca6b0, cvp->count 0, cvp->owner->next_count 0
No.1 philosopher_condvar quit
No.4 philosopher_sema quit
Iter 4, No.0 philosopher_sema is eating
No.2 philosopher_sema quit
Iter 4, No.3 philosopher_sema is eating
phi_test_condvar: state_condvar[0] will eating
phi_test_condvar: signal self_cv[0]
cond_signal begin: cvp c03ca688, cvp->count 1, cvp->owner->next_count 0
cond_wait end: cvp c03ca688, cvp->count 0, cvp->owner->next_count 1
Iter 4, No.0 philosopher_condvar is eating
cond_signal end: cvp c03ca688, cvp->count 0, cvp->owner->next_count 0
No.4 philosopher_condvar quit
No.2 philosopher_condvar quit
No.0 philosopher_sema quit
No.3 philosopher_sema quit
No.0 philosopher_condvar quit
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:474:
initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K>
```

Make grade之后的结果:

```
mooos-> make grade
badsegment:          (6.8s)
-check result:              OK
-check output:             WRONG
!! error: missing 'check_slab() succeeded!'

divzero:              (3.2s)
-check result:              OK
-check output:             WRONG
!! error: missing 'check_slab() succeeded!'

softint:              (2.7s)
-check result:              OK
-check output:             WRONG
!! error: missing 'check_slab() succeeded!'

faultread:            (2.9s)
-check result:              OK
-check output:             WRONG
!! error: missing 'check_slab() succeeded!'

faultreadkernel:      (1.6s)
-check result:              OK
-check output:             WRONG
!! error: missing 'check_slab() succeeded!'

hello:                (3.0s)
-check result:              OK
-check output:             WRONG
!! error: missing 'check_slab() succeeded!'

testbss:              (1.6s)
-check result:              OK
-check output:             WRONG
!! error: missing 'check_slab() succeeded!'

pgdir:                (3.1s)
-check result:              OK
-check output:             WRONG
!! error: missing 'check_slab() succeeded!'

yield:                (3.1s)
-check result:              OK
-check output:             WRONG
!! error: missing 'check_slab() succeeded!'

badarg:               (3.6s)
-check result:              OK
-check output:             WRONG
!! error: missing 'check_slab() succeeded!'

exit:                 (3.1s)
-check result:              OK
-check output:             WRONG
!! error: missing 'check_slab() succeeded!'

spin:                 (3.8s)
-check result:              OK
-check output:             WRONG
!! error: missing 'check_slab() succeeded!'

waitkill:             (4.8s)
-check result:              OK
-check output:             WRONG
!! error: missing 'check_slab() succeeded!'

forktest:             (3.1s)
-check result:              WRONG
!! error: missing 'init check memory pass.'

-check output:             WRONG
!! error: missing 'check_slab() succeeded!'
```

四. 实验体会

通过本次实验，让我了解了操作系统底层是如何设计并实现信号量和管程，并通过分别基于信号量和管程来实现一个哲学家就餐算法，让我比较了这两种实现直接的差别以及各自的优缺点，从而提高了对线程同步方面的理解和操作系统对同步方面的支持的理解。