实验七：同步互斥

遇到的问题

这次仍然是与lab6同样的问题，所以不再赘述，先在kern/mm/kmalloc.c改掉success，再在kern/process/proc.c中注释掉那两句assert，还有导致make qemu进入panic的那两句assert也先注释掉，虽然注释掉之后可能会引起一些问题，但是在这之后执行make qemu的结果与指导书上就相同了，所以我也没有更深入地去找这两句assert报错的原因。

第一次进行make grade的结果还是与lab6一样，check result都显示OK，check output都显示WRONG，根据error信息，可以发现报的都是同一个错：“missing 'page fault at 0x00000100: K/W [no page found].”，这里可能是之前的缺页处理没有做好，返回之前的实验修改了一下，方法还是与lab6相同。

实验目的

- 理解操作系统的同步互斥的设计实现；
- 理解底层支撑技术：禁用中断、定时器、等待队列；
- 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

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。

练习0：填写已有实验

把实验1/2/3/4/5/6的代码填入本实验中代码中有“LAB1”/“LAB2”/“LAB3”/“LAB4”/“LAB5”/“LAB6”的注释相应部分。并确保编译通过。为了能够正确执行lab7的测试应用程序，需对已完成的实验1/2/3/4/5/6的代码进行进一步改进。

这次需要修改 trap.c、vmm.c、default_pmm.c、pmm.c、proc.c、swap_fifo.c。

```
trap.c:
static void trap_dispatch(struct trapframe *tf) {
    /* LAB7 YOUR CODE */
    /* you should upate you lab6 code
    * IMPORTANT FUNCTIONS:
    * run_timer_list
```

```
    */

    ++ticks;
    /* 注销掉下面这一句 因为这一句被包含在了 run_timer_list()
        run_timer_list() 在之前的基础上 加入了对 timer 的支持 */
    // sched_class_proc_tick(current);
    run_timer_list();
}
```

定时器提供了基于时间事件的调度机制，lab7将时间片作为基本的调度和记时单位，实现基于时间长度的睡眠等待和唤醒机制。与lab6相比中断时，lab7调用`run_timer_list`来更新当前系统时间，遍历当前所有处在系统管理内的定时器，并找出所有到期的进程进行唤醒。

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）

给出内核级信号量的设计描述，并说明其大致执行流程。

在kern/sync/sem.h中定义了一个信号量的数据结构：

```
typedef struct {
    int value; // 计数值 用于 PV操作
    wait_queue_t wait_queue; // 进程等待队列
} semaphore_t;
```

在kern/sync/wait.h中定义了等待项，用于等待队列 存放了当前等待的线程PCB 和 唤醒原因 和 等待队列 和 用于还原结构体的等待队列标志：

```
typedef struct {
    struct proc_struct *proc;//标记当前等待元素对应的进程
    uint32_t wakeup_flags;
    wait_queue_t *wait_queue;//标识当前等待元素所在的等待队列wq
    list_entry_t wait_link;//用来建立当前等待元素与等待队列wq的链接。
} wait_t;
```

在kern/sync/sem.c中编写sem_init函数，初始化信号量中的计数值和等待队列：

```
void sem_init(semaphore_t *sem, int value) {
    sem->value = value;//将value设置为输入的值
    wait_queue_init(&(sem->wait_queue));//wq初始化为空链表,等待队列wq的每个元素均为wait_t结构
}
```

之后就是比较重要的P/V操作，在kern/sync/sem.c中：

```
/*
    P操作 要关闭中断并保存 eflag 寄存器的值 避免共享变量被多个线程同时修改
    判断 计数值是否大于 0 若大于 0 说明此时没有其他线程访问临界区 则直接将计数值 减 1 并 返回
    若 计数值小于 0 则 已经有其他线程访问临界区了 就将当前线程放入等待队列中 并调用调度函数
    等到进程被唤醒 再将当前进程从等待队列中 取出并删去 最后判断等待的线程是因为什么原因被唤醒
*/
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;
}

/*
    V操作 也要关闭中断 并保存 eflag 寄存器的值 防止共享变量同时被多个线程访问或修改
    先判断等待队列是否为空 若为空 则将计数值 加 1 并返回
    若不为空 则说明还有线程在等待 此时取出等待队列的第一个线程 并将其 唤醒 唤醒的过程中
    将其从等待队列中删除
*/
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);
}
```

给出给用户态进程/线程提供信号量机制的设计方案，并比较说明给内核级提供信号量机制的异同。

可以将内核级信号量包装成系统调用，供用户进程使用，但不能直接使用信号量结构体的指针作为参数，不能持有或访问内核的指针。可以在每个进程的PCB里维护一个信号量指针数组，供用户态进程及其进程下的多线程使用。不同的地方在于需要通过系统调用进入内核态进行操作。

基于内核级信号量的哲学家就餐问题的实现

1. 内核线程initproc执行init_main，init_main执行check_sync，check_sync内部测试哲学家就餐问题的解决方案。

2. check_sync分为两部分，初始化解决哲学家就餐问题需要用到的信号量和条件变量。现在只关注信号量部分。首先调用sem_init初始化信号量，将信号量的value设置为1、wq初始化为空，然后调用kernel_thread创建了5个使用信号量的内核线程。
3. 这5个使用信号量的内核线程的执行函数都是philosopher_using_semaphore。在philosopher_using_semaphore开头首先打印哲学家ID，然后进行4次循环，即进行4次思考和吃饭。首先哲学家思考10个ticks，这通过调用do_sleep来模拟。do_sleep首先将当前进程的state设置为SLEEPING，然后为进程创建一个expires为10 ticks的定时器，并添加到定时器列表中，最后调用schedule让出CPU给其他进程。
4. 10个ticks过去后，思考完毕，哲学家调用phi_take_forks_sema尝试拿起两把叉子。为了保护state_sema和s被互斥访问，设置了互斥量mutex。因此， phi_take_forks_sema对mutex执行down操作，进入临界区，然后将自己的state_sema标记为HUNGRY，接着调用phi_test_sema来检查是否能拿到两只叉子。若能拿到，则将自己的state_sema标记为EATING，然后离开临界区，开始吃饭，时间同样为10个ticks；若不能拿到，则通过对s[i]执行down操作而堵塞。
5. 10个ticks过去后，吃饭完毕，哲学家调用phi_put_forks_sema把两把叉子同时放回桌子。同样，为了保护state_sema和s被互斥访问，phi_put_forks_sema先对mutex执行down操作，进入临界区，然后将自己的state_sema标记为THINKING，接着两次调用phi_test_sema来检查左右邻居是否能进餐。若能就餐，则对s[i]执行up操作，这时会唤醒其他哲学家。等到哲学家离开临界区、进入下一次的思考后，会发生进程切换，使得其他哲学家可以就餐。

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）

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，并说明其大致执行流程。

这里实现的是 Hoare管程 因为 等待条件变量的进程的优先级更高。需要的数据结构：

```
// 管程数据结构 kern/sync/monitor.h
typedef struct monitor{
    semaphore_t mutex; // 二值信号量 用来互斥访问管程
    semaphore_t next; // 用于 条件同步 用于发出signal操作的进程等条件为真之前进入睡眠
    int next_count; // 记录睡在 signal 操作的进程数
    condvar_t *cv; // 条件变量
} monitor_t;

// 条件变量数据结构 kern/sync/monitor.h
typedef struct condvar{
    semaphore_t sem; // 用于条件同步 用于发出wait操作的进程等待条件为真之前进入睡眠
    int count; // 记录睡在 wait 操作的进程数(等待条件变量成真)
    monitor_t * owner; // 所属管程
} condvar_t;
```

初始化管程：

```
void monitor_init (monitor_t * mtp, size_t num_cv) {
    int i;
    assert(num_cv>0);
    mtp->next_count = 0; // 睡在signal进程数 初始化为0
    mtp->cv = NULL;
    sem_init(&(mtp->mutex), 1); // 二值信号量 保护管程 使进程访问管程操作为互斥的
    sem_init(&(mtp->next), 0); // 条件同步信号量
```

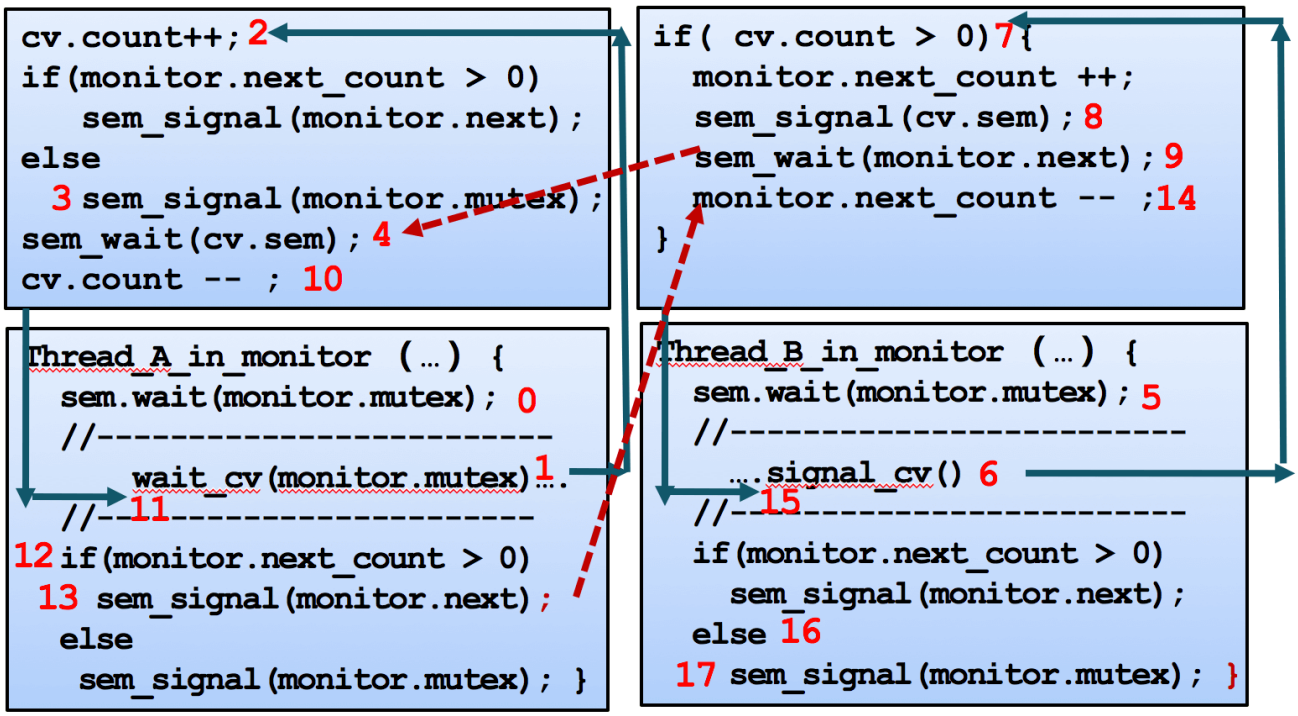
```
    mtp->cv =(condvar_t *) kmalloc(sizeof(condvar_t)*num_cv); // 获取一块内核空间 放置条件
变量
    assert(mtp->cv!=NULL);
    for(i=0; i<num_cv; i++){
        mtp->cv[i].count=0;
        sem_init(&(mtp->cv[i].sem),0);
        mtp->cv[i].owner=mtp;
    }
}
```

P/V操作，即wait和signal。wait用于进程因无法获取所需的资源时而将自己堵塞，signal用于另一个进程释放或生成相关资源后通知之前处于wait状态的进程而解除堵塞。

```
// 管程wait操作
/*
先将 因为条件不成立而睡眠的进程计数加1
分支1. 当 管程的 next_count 大于 0 说明 有进程睡在了 signal 操作上 我们将其唤醒
分支2. 当 管程的 next_count 小于 0 说明 当前没有进程睡在 signal操作数 只需要释放互斥体
然后 再将 自身阻塞 等待 条件变量的条件为真 被唤醒后 将条件不成立而睡眠的进程计数减1 因为现在成立了
*/
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++;
    if (cvp->owner->next_count > 0) {
        up(&(cvp->owner->next));
    } else {
        up(&(cvp->owner->mutex));
    }

    down(&(cvp->sem)); // 阻塞自己 等待条件成真
    cvp->count--;

    cprintf("cond_wait end:  cvp %x, cvp->count %d, cvp->owner->next_count %d\n", cvp,
cvp->count, cvp->owner->next_count);
}

// 管程signal操作
/*
分支1. 因为条件不成立而睡眠的进程计数小于等于0 时 说明 没有进程需要唤醒 则直接返回
分支2. 因为条件不成立而睡眠的进程计数大于0 说明有进程需要唤醒 就将其唤醒
同时设置 条件变量所属管程的 next_count 加1 以用来告诉 wait操作 有进程睡在了 signal操作上
然后自己将自己阻塞 等待条件同步 被唤醒 被唤醒后 睡在 signal 操作上的进程应该减少 故 next_count 应减
1
*/
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) { // 若存在因为当前条件变量而等待的进程的话
        up(&(cvp->sem));
        cvp->owner->next_count++; // 所属管程的 next 计数 加 1 表示当前进程会被等待者堵塞
```

```
        down(&(cvp->owner->next)); // 阻塞自己 等待条件同步
        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);
}
```

执行流程:



check_sync.c:

```
// 测试编号为i的哲学家是否能获得刀叉 如果能获得 则将状态改为正在吃 并且 尝试唤醒 因为wait操作睡眠的进
程
// cond_signal 还会阻塞自己 等被唤醒的进程唤醒自己
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]) ;
    }
}
```

编程部分:

```
// 拿刀叉
/*首先对管程的mutex执行down操作进入临界区，这是确保任何时候最多只有一个进程进入管程。然后将自己的state
设置为HUNGRY。接着测试是否满足就餐条件：自己的state是HUNGRY，而且左右的哲学家的state都不是EATING。
如果满足，则将自己的state设置为EATING，然后对管程的mutex执行up操作而退出临界区，从而完成拿叉子操作。
如果不满足就餐条件，则调用cond_wait将自己堵塞。*/
void phi_take_forks_condvar(int i) {
```

```
    down(&(mtp->mutex));    // P操作进入临界区
    state_condvar[i] = HUNGRY; // 饥饿状态 准备进食
    phi_test_condvar(i); // 测试当前是否能获得刀叉
    while (state_condvar[i] != EATING) {
        cond_wait(&mtp->cv[i]); // 若不能拿 则阻塞自己 等其它进程唤醒
    }
    if(mtp->next_count>0)
        up(&(mtp->next));
    else
        up(&(mtp->mutex));
}

// 放刀叉
/*首先对管程的mutex执行down操作进入临界区，然后将自己的state设置为THINKING。接下来测试左右哲学家是否满足就餐条件，若满足，则调用cond_signal将对应进程唤醒，而把自己堵塞。*/
void phi_put_forks_condvar(int i) {
    down(&(mtp->mutex)); // P操作进入临界区
    state_condvar[i] = THINKING; // 思考状态
    phi_test_condvar(LEFT); // 试试左右两边能否获得刀叉
    phi_test_condvar(RIGHT);
    if(mtp->next_count>0) // 有哲学家睡在 signal操作 则将其唤醒
        up(&(mtp->next));
    else
        up(&(mtp->mutex)); // 离开临界区
}
```

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。

与练习一的第二问相同，依然是通过系统调用的方式。

能否不用基于信号量机制来完成条件变量？如果不能，请给出理由，如果能，请给出设计说明和具体实现。

可以不用，使用自旋锁，条件变量维护进程的等待队列和进程的等待数目，用等待队列和锁机制同样可以实现。

执行make qemu：

```
++ setup timer interrupts
I am No.4 philosopher_condvar
Iter 1, No.4 philosopher_condvar is thinking
I am No.3 philosopher_condvar
Iter 1, No.3 philosopher_condvar is thinking
I am No.2 philosopher_condvar
Iter 1, No.2 philosopher_condvar is thinking
I am No.1 philosopher_condvar
Iter 1, No.1 philosopher_condvar is thinking
I am No.0 philosopher_condvar
Iter 1, No.0 philosopher_condvar is thinking
I am No.4 philosopher_sema
Iter 1, No.4 philosopher_sema is thinking
I am No.3 philosopher_sema
Iter 1, No.3 philosopher_sema is thinking
I am No.2 philosopher_sema
Iter 1, No.2 philosopher_sema is thinking
I am No.1 philosopher_sema
Iter 1, No.1 philosopher_sema is thinking
I am No.0 philosopher_sema
Iter 1, No.0 philosopher_sema is thinking
kernel_execve: pid = 2, name = "matrix".
fork ok.
pid 13 is running (1000 times)!.
pid 13 done!.
pid 15 is running (1100 times)!.
Iter 1, No.0 philosopher_sema is eating
Iter 1, No.2 philosopher_sema is eating
phi_test_condvar: state_condvar[0] will eating
```

```
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
Iter 4, No.1 philosopher_condvar is thinking
phi_take_forks_condvar: 2 didn't get fork and will wait
cond_wait begin:  cvp c03ca6b0, cvp->count 0, cvp->owner->next_count 0
pid 29 done!.
phi_test_condvar: state_condvar[2] will eating
phi_test_condvar: signal self_cv[2]
cond_signal begin: cvp c03ca6b0, cvp->count 1, cvp->owner->next_count 0
cond_wait end:  cvp c03ca6b0, cvp->count 0, cvp->owner->next_count 1
Iter 4, No.2 philosopher_condvar is eating
cond_signal end: cvp c03ca6b0, cvp->count 0, cvp->owner->next_count 0
Iter 4, No.3 philosopher_condvar is thinking
phi_take_forks_condvar: 4 didn't get fork and will wait
cond_wait begin:  cvp c03ca6d8, cvp->count 0, cvp->owner->next_count 0
phi_test_condvar: state_condvar[4] will eating
phi_test_condvar: signal self_cv[4]
cond_signal begin: cvp c03ca6d8, cvp->count 1, cvp->owner->next_count 0
cond_wait end:  cvp c03ca6d8, cvp->count 0, cvp->owner->next_count 1
Iter 4, No.4 philosopher_condvar is eating
pid 23 done!.
cond_signal end: cvp c03ca6d8, cvp->count 0, cvp->owner->next_count 0
No.0 philosopher_condvar quit
phi_take_forks_condvar: 1 didn't get fork and will wait
cond_wait begin:  cvp c03ca69c, cvp->count 0, cvp->owner->next_count 0
pid 20 done!.
matrix pass.
```

执行make grade：

```
exit:                   (2.9s)
  -check result:                OK
  -check output:                OK
spin:                   (2.9s)
  -check result:                OK
  -check output:                OK
waitkill:               (3.4s)
  -check result:                OK
  -check output:                OK
forktest:               (2.5s)
  -check result:                OK
  -check output:                OK
forktree:               (5.3s)
  -check result:                OK
  -check output:                OK
priority:               (15.3s)
  -check result:                OK
  -check output:                OK
sleep:                  (11.3s)
  -check result:                OK
  -check output:                OK
sleepkill:              (2.8s)
  -check result:                OK
  -check output:                OK
matrix:                 (10.1s)
  -check result:                OK
  -check output:                OK
Total Score: 190/190
pcy@ubuntu:~/Desktop/labcodes/lab7$ make qemu
```

ucore实验到这里就告一段落吧。还是有一些收获的。一个人做实验可能会比六个人做实验更快更有效率，但是一群人踩坑会比一个人踩坑更快更有勇气地走出来。