lab2 实验报告

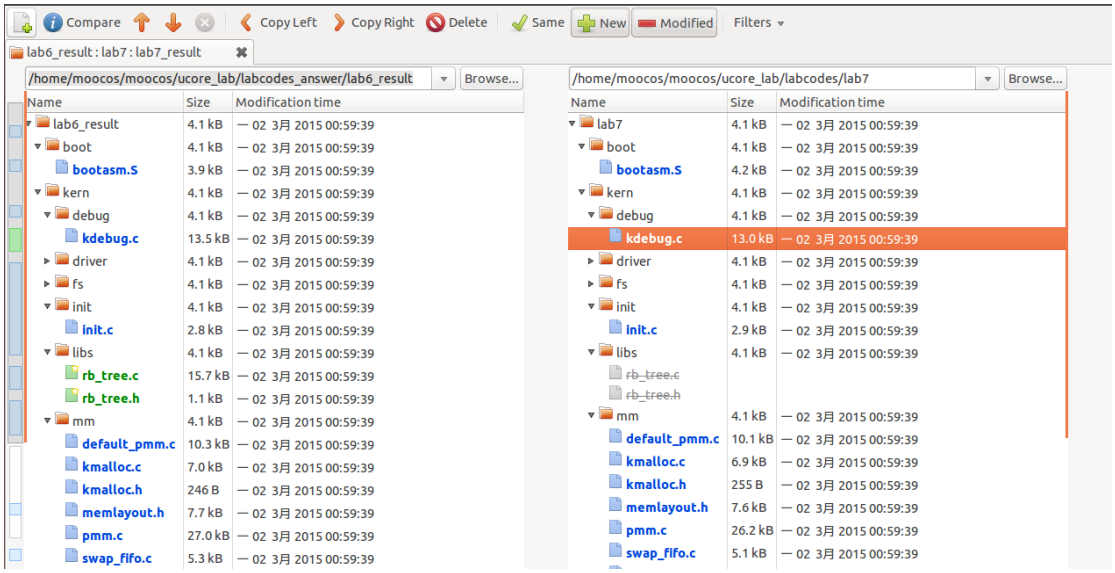
一、实验目的

1. 理解操作系统的同步互斥概念及各种实现方法
2. 理解信号量和管程机制，并应用于哲学家就餐问题和生产者-消费者问题的解决

二、实验内容

0）补全代码和概念理解

首先使用 meld 可视化比较工具对比 lab6_result 和 lab7



修改了 kdebug.c, trap.c, default_pmm.c, swap_fifo.c, vmm.c, proc.c, sche.c 等 8 个文件

通过看课本、看讲义了解了如下几个**概念**:

- **临界资源**

在操作系统中，进程是占有资源的最小单位。对于某些资源来说，其在同一时间只能被一个进程所占用。这些一次只能被一个进程所占用的资源就是所谓的临界资源

- **进程同步、互斥**

进程同步是进程之间直接制约关系，是为完成某种任务而建立的两个或多个线程，这个线程需要在某些位置上协调他们的工作次序而等待、传递信息所产生的制约关系。进程间的直接制约关系来源于他们之间的合作。比如进程 A 需要从缓冲区读取进程 B 产生的信息，当缓冲区为空时，进程 B 因为读取不到信息而被阻塞。而当进程 A 产生信息放入缓冲区时，进程 B 才会被唤醒。

进程互斥是进程之间的间接制约关系。当一个进程进入临界区使用临界资源时，另一个进程必须等待。只有当使用临界资源的进程退出临界区后，这个进程才会解除阻塞状态。比如进程 B 需要访问打印机，但此时进程 A 占有了打印机，进程 B 会被阻塞，直到进程 A 释放了打印机资源,进程 B 才可以继续执行。

实现进程同步互斥有多种方法：

- 严格轮换
  每个进程从头到尾执行，缺点是效率太低
- Peterson 解决方案
  turn 表示可以进入临界区的进程，flag 表示想要进入临界区的进程
- 硬件指令
  简单的硬件指令 test_and_set()和 compare_and_swap()
- 互斥锁
  mutex 进程进入临界区得到锁，退出时释放锁
- 定时器
  时钟(timer)中断给 OS 提供了一定间隔的时间事件，两次中断之间的间隔为一个时间片，基于该事件单位实现调度和睡眠唤醒等机制
- **屏蔽与使能中断**

进入临界区屏蔽中断、出临界区就打开中断，只适用于单处理器

实现开关中断的控制函数为

kern/sync.c 中的 **local_intr_save()**和 **local_intr_restore()**

```
27  #define local_intr_save(x)      do { x = __intr_save(); } while (0)
28  #define local_intr_restore(x)   __intr_restore(x);
```

调用了**_intr_save()**和**_intr_restore()**

```
11  static inline bool
12  __intr_save(void) {
13      if (read_eflags() & FL_IF) {
14          intr_disable();
15          return 1;
16      }
17      return 0;
18  }
19
20  static inline void
21  __intr_restore(bool flag) {
22      if (flag) {
23          intr_enable();
24      }
25  }
```

read_flags()在 x86.h 中定义

```
148 static inline uint32_t
149 read_eflags(void) {
150     uint32_t eflags;
151     asm volatile ("pushfl; popl %0" : "=r" (eflags));
152     return eflags;
153 }
```

FL_IF 是 ucore 中定义的中断标志寄存器

```
4   /* Eflags register */
5   #define FL_CF           0x00000001  // Carry Flag
6   #define FL_PF           0x00000004  // Parity Flag
7   #define FL_AF           0x00000010  // Auxiliary carry Flag
8   #define FL_ZF           0x00000040  // Zero Flag
9   #define FL_SF           0x00000080  // Sign Flag
10  #define FL_TF           0x00000100  // Trap Flag
11  #define FL_IF           0x00000200  // Interrupt Flag
```

调用了 intr.c 中的 **intr_disable()**和 **intr_enable()**

```
4   /* intr_enable - enable irq interrupt */
5   void
6   intr_enable(void) {
7       sti();
8   }
9
10  /* intr_disable - disable irq interrupt */
11  void
12  intr_disable(void) {
13      cli();
14  }
```

最后调用了 x86.h 中的 **cli()**和 **sti()**

```
133 static inline void
134 sti(void) {
135     asm volatile ("sti");
136 }
137
138 static inline void
139 cli(void) {
140     asm volatile ("cli" ::: "memory");
141 }
```

__asm__ __volatile__(""":::"memory")这行代码是内存屏障。
__asm__用于指示编译器在此插入汇编语句;
__volatile__用于告诉编译器，严禁将此处的汇编语句与其它的语句重组合优化。即按原来的样子处理这里的汇编;
memory 阻止了 cpu 又将 registers，cache 中的数据用于去优化指令，而避免去访问内存。

CLI(Clear Interrupt) 中断标志置 0 指令 使 IF = 0
STI(Set Interrupt) 中断标志置 1 指令 使 IF = 1
即：CLI 关闭中断，防止当前正在执行的控制流被打断，即内核运行当前进程无法被打断或重新调度，实现了对临界区的互斥操作；而 STI 打开中断

但是这种屏蔽与使能中断的方法只适用于单处理器情况，在多处理器情况下，屏蔽了一个 CPU 的中断，其它 CPU 上执行的进程仍可以中断或调度

- **等待队列**

睡眠-唤醒机制是实现 ucore 中信号量机制和条件变量机制的基础，它的底层就是等待队列 wait_queue。需要等待事件的进程在转入休眠状态(PROC_SLEEPING)后插入到等待队列中；当事件发生后，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（PROC_RUNNABLE）
ucore 在 kern/sync/wait.h 中实现了等待项 wait_t 结构和等待队列 wait_queue_t

```
6 typedef struct {
7     list_entry_t wait_head;
8 } wait_queue_t;
9
10 struct proc_struct;
11
```

wait_head 为队列的队头

```
typedef struct {
    struct proc_struct *proc;
    uint32_t wakeup_flags;
    wait_queue_t *wait_queue;
    list_entry_t wait_link;
} wait_t;
```

proc 是指向等待进程的指针；wakeup_flags 标记进程被放入等待队列的原因；wait_queue 指向该 wait_t 结构输入的队列；wait_link 用于连接所有的 wait_t 结构

相关函数在 wait.c 中，通过调用 list.h 中的函数来实现：
最基本的**初始化、查找、插入、删除**
wait_init()初始化 wait_t 结构、
wait_queue_init()初始化 wait_queue、
wait_queue_add()前插到队列中
wait_queue_del()删除节点
wait_queue_next()找到下一个节点
wait_queue_prev()找到前一个节点
wait_queue_first()应该是第一个有内容的节点
wait_queue_last()尾节点
wait_queue_empty()清空整个队列
wait_in_queue()判断某一 wait_t 结构是否在队列中

以及**将进程插入到等待队列**和**从等待队列中唤醒进程**
waitup_wait()通过调用 wakeup_proc()唤醒与当前 wait_t 相关联的进程

```
void
wakeup_proc(struct proc_struct *proc) {
    assert(proc->state != PROC_ZOMBIE);
    bool intr_flag;
    local_intr_save(intr_flag);
    {
        if (proc->state != PROC_RUNNABLE) {
            proc->state = PROC_RUNNABLE;
            proc->wait_state = 0;
            if (proc != current) {
                sched_class_enqueue(proc);
            }
        }
        else {
            warn("wakeup runnable process.\n");
        }
    }
    local_intr_restore(intr_flag);
}
```

wakeup_fist()唤醒队列上第一个 wait_t 相关联的进程
wakeup_queue()唤醒等待在整个队列上所有进程
wake_current_set()将当前进程设为 PROC_SLEEPING 状态，并插入到等待队列中

```
wait_current_set(wait_queue_t *queue, wait_t *wait, uint32_t wait_state) {
    assert(current != NULL);
    wait_init(wait, current);
    current->state = PROC_SLEEPING;
    current->wait_state = wait_state;
    wait_queue_add(queue, wait);
}
```

- **信号量**

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表明条件满足了的信号。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 传送和接受信号，信号量通过 V、P 操作来修改传送信号量。如果未发送，则进程被阻塞或睡眠

伪代码如下：

```
struct semaphore {
int count;
queueType queue;
};
void semWait(semaphore s)
{
s.count--;
if (s.count < 0) {
/* place this process in s.queue */;
/* block this process */;
}
}
void semSignal(semaphore s)
{
s.count++;
if (s.count<= 0) {
/* remove a process P from s.queue */;
/* place process P on ready list */;
}
}
```

在 ucore 中，通过 sem.h, sem.c 实现了信号量，数据结构定义如下：

```
 8 typedef struct {
 9     int value;
10     wait_queue_t wait_queue;
11 } 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 value 和一个进程等待队列 wait_queue

Value > 0,表示资源的空闲数

Value < 0,表示改信号量的等待队列里的进程数

Value = 0，表示等待队列为空

- **哲学家就餐问题：**

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。

1）下面通过分析代码框架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**
lab7 中 ucore 初始化过程，开始的执行流程与之前相同，直到执行到创建第二个内核线程 **init_main()**时，增加了 **check_sync()**函数的调用

```
838 static int
839 init_main(void *arg) {
840     size_t nr_free_pages_store = nr_free_pages();
841     size_t kernel_allocated_store = kallocated();
842
843     int pid = kernel_thread(user_main, NULL, 0);
844     if (pid <= 0) {
845         panic("create user_main failed.\n");
846     }
847  extern void check_sync(void);
848     check_sync();                // check philosopher sync problem
849
850     while (do_wait(0, NULL) == 0) {
851         schedule();
852     }
```

check_sync.c 中的 **check_sync()**函数如下：

```
169 void check_sync(void){
170
171     int i;
172
173     //check semaphore
174     sem_init(&mutex, 1);
175     for(i=0;i<N;i++){
176         sem_init(&s[i], 0);
177         int pid = kernel_thread(philosopher_using_semaphore, (void *)i, 0);
178         if (pid <= 0) {
179             panic("create No.%d philosopher_using_semaphore failed.\n");
180         }
181         philosopher_proc_sema[i] = find_proc(pid);
182         set_proc_name(philosopher_proc_sema[i], "philosopher_sema_proc");
183     }
184
185     //check condition variable
186     monitor_init(&mt, N);
187     for(i=0;i<N;i++){
188         state_condvar[i]=THINKING;
189         int pid = kernel_thread(philosopher_using_condvar, (void *)i, 0);
190         if (pid <= 0) {
191             panic("create No.%d philosopher_using_condvar failed.\n");
192         }
193         philosopher_proc_condvar[i] = find_proc(pid);
194         set_proc_name(philosopher_proc_condvar[i], "philosopher_condvar_proc");
195     }
196 }
```

可以看出该函数分为两部分：
第一部分实现基于信号量的哲学家问题，第二部分实现基于管程的哲学家问题，

先看前半部分，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。
下面看 check_sync()调用的 **philosopher_using_semaphore()**函数：

```c
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;
}
```

在一个循环中，一个信号量代表的哲学家初始是思考状态，经过一个 SLEEP_TIME 后，调用 phi_take_forks_sema()函数试图拿叉子吃饭，如果成功以后就进入进餐状态，再经过一个 SLEEP_TIME 以后调用 phi_put_forks_sema()函数把两把叉子同时放回桌子，开始下一次循环

根据注释理解 **phi_take_forks_sema()**函数和 **phi_put_forks_sema()**函数：

```c
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 */
}
```

其中 **phi_test_sema()**函数用于尝试获取两把叉子：
如果当前处于饥饿状态，且左右两边都不在进食，则可以拿起叉子进食

```c
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]);
    }
}
```

两个函数的核心是 **up()**和 **down()**两个函数，在 sem.c 中找到如下：
对应 ucore 中最重要的信号量操作 P 操作和 V 操作，具体实现靠_up()函数和_down()函数

```
56 void
57 up(semaphore_t *sem) {
58     __up(sem, WT_KSEM);
59 }
60
61 void
62 down(semaphore_t *sem) {
63     uint32_t flags = __down(sem, WT_KSEM);
64     assert(flags == 0);
65 }
```

先看**_up()函数**，实现信号量的 V 操作

```
16 static __noinline void __up(semaphore_t *sem, uint32_t wait_state) {
17     bool intr_flag;
18     local_intr_save(intr_flag);//关闭中断
19     {
20         wait_t *wait;
21         if ((wait = wait_queue_first(&(sem->wait_queue))) == NULL) {//该信号量对应wait_queue没有进程等待
22             sem->value ++;//信号量value加1
23         }
24         else {//有进程在等待
25             assert(wait->proc->wait_state == wait_state);
26             wakeup_wait(&(sem->wait_queue), wait, wait_state, 1);//唤醒queue中第一个wait_t关联的进程
27         }
28     }
29     local_intr_restore(intr_flag);//打开中断
30 }
```

首先关中断，如果信号量对应的 waitqueue 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，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。

再来看**_down()函数**，实现信号量的 P 操作

```
32 static __noinline uint32_t __down(semaphore_t *sem, uint32_t wait_state) {
33     bool intr_flag;
34     local_intr_save(intr_flag);//关闭中断
35     if (sem->value > 0) {//判断当前信号量value是否>0
36         sem->value --;
37         local_intr_restore(intr_flag);//如果是，value--，打开中断并直接返回
38         return 0;
39     }
40     wait_t __wait, *wait = &__wait;//当前信号量<=0，表明无法获得信号量
41     wait_current_set(&(sem->wait_queue), wait, wait_state);//将当前进程加入到等待队列中
42     local_intr_restore(intr_flag);//打开中断
43
44     schedule();//运行调度器选择其它进程执行
45
46     local_intr_save(intr_flag);//关闭中断
47     wait_current_del(&(sem->wait_queue), wait);//唤醒
48     local_intr_restore(intr_flag);//打开中断
49
50     if (wait->wakeup_flags != wait_state) {
51         return wait->wakeup_flags;
52     }
53     return 0;
54 }
```

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。是则表明可以获得信号量，故让 value 减一，并打开中断返回即可；不是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，然后调度另外一个进程执行。如果被 V 操作唤醒，则把自身关联的 wait_t 从等待队列中删除

sem.c 中还实现了两个函数：

sem_init()：信号量的初始化

```
10 void
11 sem_init(semaphore_t *sem, int value) {
12     sem->value = value;
13     wait_queue_init(&(sem->wait_queue));
14 }
```

try_down()：跟 down 功能类似，但是用 Understand 查看发现没有被调用过

至此，完成基于信号量的哲学家问题

- **管程、条件变量**

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。

为了防止因为忙等而出现的死锁现象，引入条件变量的概念。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

- signal_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

- 下面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

首先看管程 **monitor_t** 和条件变量 **condvar_t**

```
69 typedef struct condvar{
70     semaphore_t sem;        // the sem semaphore  is used to down the waiting proc, and the signaling
   proc should up the waiting proc
71     int count;              // the number of waiters on condvar
72     monitor_t * owner;      // the owner(monitor) of this condvar
73 } condvar_t;
74
75 typedef struct monitor{
76     semaphore_t mutex;      // the mutex lock for going into the routines in monitor, should be
   initialized to 1
77     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.
78     int next_count;         // the number of of sleeped signaling proc
79     condvar_t *cv;          // the condvars in monitor
80 } monitor_t;
```

- **mutex** 是一个二值信号量，只允许一个进程进入管程，初始化为 1，是实现互斥的关键；
- **条件变量 cv** 通过执行 wait_cv，会使得等待某个条件 Cond 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 Cond 为真并执行 signal_cv 时，能够让等待某个条件 Cond 为真的睡眠进程被唤醒，从而继续进入管程中执行。
- **成员变量信号量 next** 和**整型变量 next_count** 用于配合进程对条件变量 cv 的操作。发出 signal_cv 的进程 A 会唤醒由于 wait_cv 而睡眠的进程 B，由于管程中只允许一个进程运行，所以进程 B 执行会导致唤醒进程 B 的进程 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。
- **信号量 sem** 用于让发出 wait_cv 操作的等待某个条件 Cond 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

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 monitor.c 中的 cond_wait()函数和 cond_signal()函数，此外还有 monitor_init()初始化函数。

先看 **monitor_init()函数**:

```
// Initialize monitor.
void
monitor_init (monitor_t * mtp, size_t num_cv) {
    int i;
    assert(num_cv>0);
    mtp->next_count = 0;
    mtp->cv = NULL;
    sem_init(&(mtp->mutex), 1); //unlocked
    sem_init(&(mtp->next), 0);
    mtp->cv =(condvar_t *) kmalloc(sizeof(condvar_t)*num_cv);
    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;
    }
}
```

对条件变量进行初始化，设置 next_count 为 0，mutex 和 next 分别初始化为 1 和 0，然后分配 num_cv 个 condvar_t，设置 cv 的 count 为 0，初始化 cv 的 sem 和 owner

cond_signal()函数：唤醒睡在条件变量上的线程

```
cond_signal(cv) {
    if(cv.count>0) {
       mt.next_count ++;
       signal(cv.sem);
       wait(mt.next);
       mt.next_count--;
    }
 }
```

对照着可分析出 cond_signal 函数的具体执行过程。
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，monitor.next_count 减一

结合注释和讲义**补全函数**

如果 cv 的 count>0，说明有 proc 在等待，那么需要唤醒等待在 cv.sem 上的 proc，并使自己进行睡眠，同时 monitor.next_count++，在被唤醒后执行 monitor.next_count–；如果 cv 的 count == 0，说明没有 proc 在等待 cv.sem，直接返回函数。

```
25 // Unlock one of threads waiting on the condition variable.
26 void
27 cond_signal (condvar_t *cvp) {
28    //LAB7 EXERCISE1: YOUR CODE
29    cprintf("cond_signal begin: cvp %x, cvp->count %d, cvp->owner->next_count %d\n",
30   /*
31    *      cond_signal(cv) {
32    *          if(cv.count>0) {
33    *             mt.next_count ++;
34    *             signal(cv.sem);
35    *             wait(mt.next);
36    *             mt.next_count--;
37    *          }
38    *       }
39    */
40    if(cvp->count>0){//当前存在睡眠的进程
41       cvp->owner->next_count++;//睡眠进程总数++
42       up(&(cvp->sem));//唤醒等待在cvp->sem上的进程
43       down(&(cvp->owner->next));//使自己睡眠
44       cvp->owner->next_count--;//睡醒后等待此条件的进程总数--
45    }
46    cprintf("cond_signal end: cvp %x, cvp->count %d, cvp->owner->next_count %d\n", cv
47 }
```

同样看 **cond_wait()函数：**使线程睡在条件变量上

```
cv.count ++;
if(mt.next_count>0)
   signal(mt.next)
else
   signal(mt.mutex);
wait(cv.sem);
cv.count --;
```

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> 0，表示有大于等于 1 个进程执行 cond_signal 函数且睡着，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<=0，表示目前没有进程执行 cond_signal 函数且睡着，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

结合注释和讲义**补全函数**

先将 cv.count++,如果 monitor.next_count >0,说明有进程执行 cond_signal()函数并且睡着了，此时唤醒此 proc；否则的话，说明目前没有因为执行了 cond_signal()函数的 proc 睡着，此时唤醒因为互斥条件 mutex 无法进入管程的 proc。在这之后，使 A 在 cv.sem 上进行等待并进行调度，如果 A 睡醒了，则 cv.count–。

```
void
cond_wait (condvar_t *cvp) {
    //LAB7 EXERCISE1: YOUR CODE
    cprintf("cond_wait begin:  cvp %x, cvp->count %d, cvp->owner->next_count %
   /*
    *         cv.count ++;
    *         if(mt.next_count>0)
    *            signal(mt.next)
    *         else
    *            signal(mt.mutex);
    *         wait(cv.sem);
    *         cv.count --;
    */
    cvp->count++;//需要睡眠的进程数++
    if(cvp->owner->next_count>0)
        up(&(cvp->owner->next));//唤醒另一个给出条件变量（即进程链表中下一个）进程
    else
        up(&(cvp->owner->mutex));//释放mutex，使其它进程进入管城
    down(&cvp->sem);//使自己睡眠
    cvp->count--;//睡眠后进程数--
    cprintf("cond_wait end:  cvp %x, cvp->count %d, cvp->owner->next_count %d\
}
```

然后分析实现 **check_sync.c** 中用管程实现部分的结构和函数

数据结构

```
struct proc_struct *philosopher_proc_condvar[N]; // N philosopher
int state_condvar[N];                            // the philosopher's state: EATING, HUNGARY, THINKING
monitor_t mt, *mtp=&mt;                          // monitor
```

高层函数 **check_sync()中**管程部分

```
//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");
}
```

与之前信号量实现一致，这里调用了 **philosopher_using_condvar()函数**与之前也类似

```
//---------- 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;
}
```

可以看出主要函数是 phi_take_forks_condvar()和 phi_put_forks_condvar()
结合注释理解三个函数:

phi_test_condvar()：试图拿到叉子
与之前信号量相同，如果当前处于饥饿状态，并且左右两边都没有在进食，则可以得到条件变量 cv[i]，拿起叉子

```
110 void phi_test_condvar (i) {
111     if(state_condvar[i]==HUNGRY&&state_condvar[LEFT]!=EATING
112             &&state_condvar[RIGHT]!=EATING) {
113         cprintf("phi_test_condvar: state_condvar[%d] will eating\n",i);
114         state_condvar[i] = EATING ;
115         cprintf("phi_test_condvar: signal self_cv[%d] \n",i);
116         cond_signal(&mtp->cv[i]) ;
117     }
118 }
```

phi_take_forks_condvar()：拿叉子

```
121 void phi_take_forks_condvar(int i) {
122      down(&(mtp->mutex));
123 //--------into routine in monitor--------------
124      // LAB7 EXERCISE1: YOUR CODE
125      // I am hungry
126      state_condvar[i]=HUNGRY;
127      // try to get fork
128      phi_test_condvar(i);
129      while(state_condvar[i]!=EATING){
130          cprintf("phi_take_forks_condvar: %d failed to get forks\n",i);
131          cong_wait(&mtp->cv[i]);//得不到叉子就睡眠
132      }
133 //--------leave routine in monitor--------------
134       if(mtp->next_count>0)//唤醒睡眠的进程
135          up(&(mtp->next));
136       else
137          up(&(mtp->mutex));
138 }
```

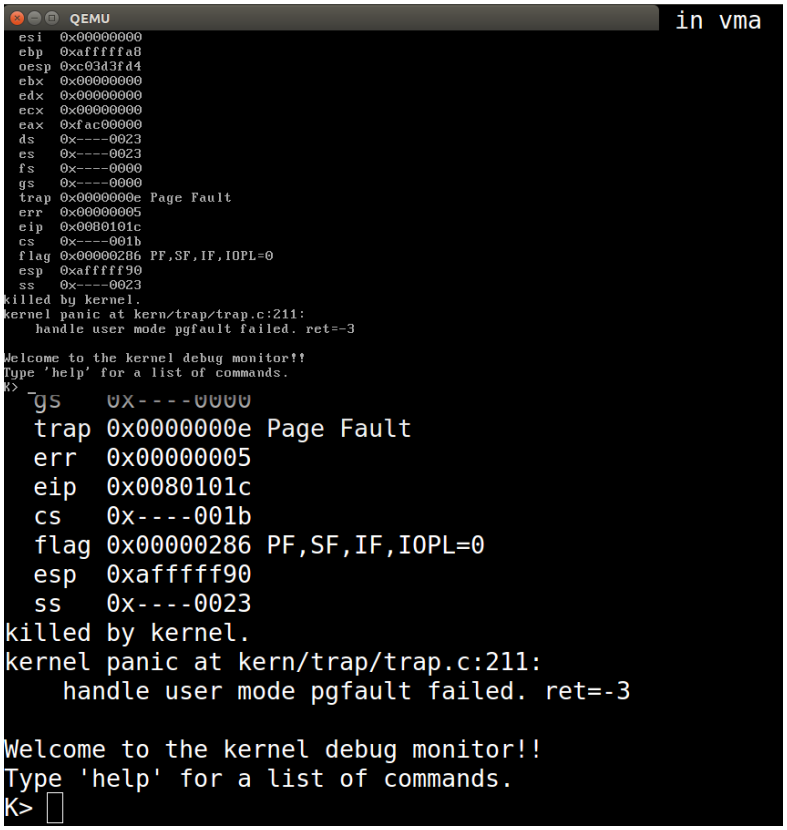
phi_put_forks_condvar()：放叉子

```
140 void phi_put_forks_condvar(int i) {
141      down(&(mtp->mutex));
142
143 //--------into routine in monitor--------------
144      // LAB7 EXERCISE1: YOUR CODE
145      // I ate over
146      state_condvar[i]=THINKING;
147      // test left and right neighbors
148      phi_test_condvar(LEFT);
149      phi_test_condvar(RIGHT);
150 //--------leave routine in monitor--------------
151      if(mtp->next_count>0)
152         up(&(mtp->next));
153      else
154         up(&(mtp->mutex));
155 }
```

make

```
'obj/bootblock.out' size: 434 bytes
build 512 bytes boot sector: 'bin/bootblock' success!
10000+0 records in
10000+0 records out
5120000 bytes (5.1 MB) copied, 0.105354 s, 48.6 MB/s
1+0 records in
1+0 records out
512 bytes (512 B) copied, 0.000109901 s, 4.7 MB/s
1605+1 records in
1605+1 records out
822207 bytes (822 kB) copied, 0.00525998 s, 156 MB/s
128+0 records in
128+0 records out
134217728 bytes (134 MB) copied, 0.290204 s, 462 MB/s
[~/moocos/ucore_lab/labcodes/lab7]
moocos->
```

make qemu

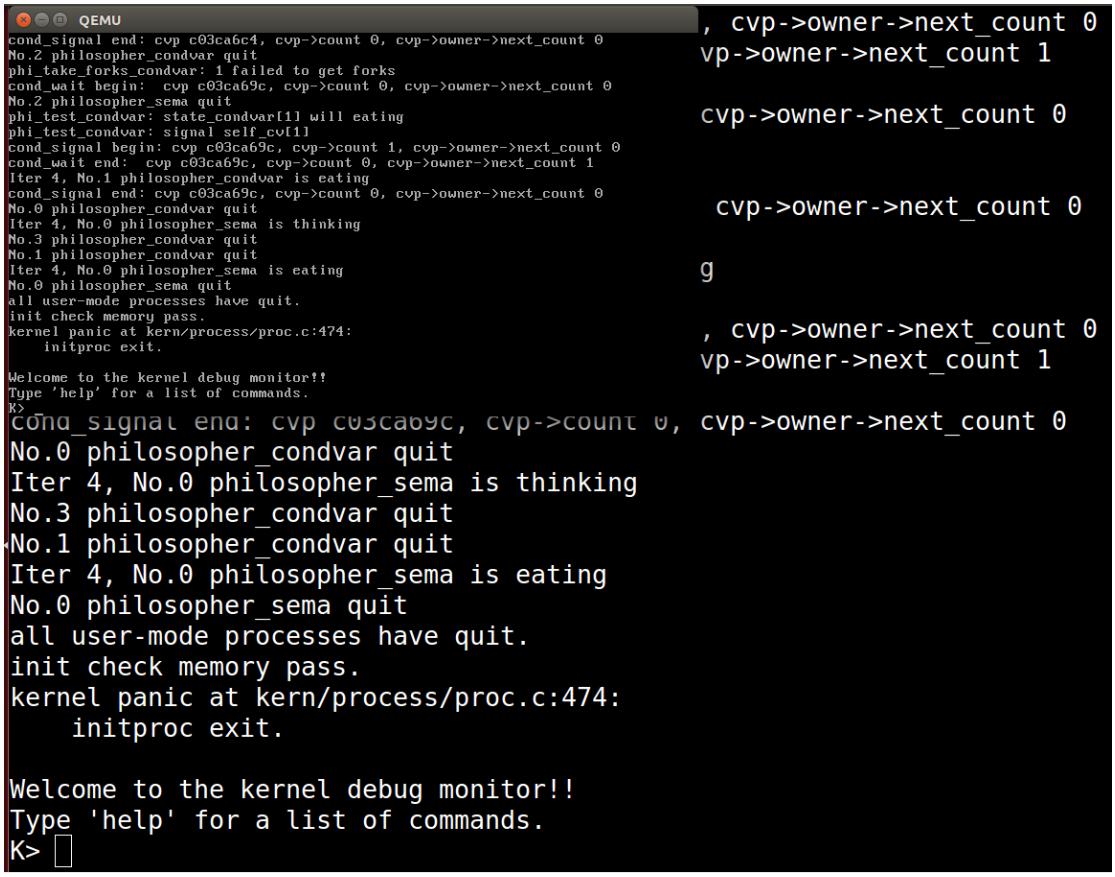


kernel panic 的位置与讲义上面不同，经过检查发现问题出在这里!

```
cvp->count++;//需要睡眠的进程数++
if(cvp->owner->next_count>0)
    up(&(cvp->owner->next));//唤醒另一
else
    up(&(cvp->owner->mutex));//释放mu
down(&(cvp->sem));//使自己睡眠
cvp->count--;//睡眠后进程数--
cprintf("cond_wait end:  cvp %x, cvp-
```

一开始 down(&(cvp->sem))没有写里面的括号，导致计算顺序错误!

修正以后的结果与讲义上面一致



- **生产者-消费者问题**

问题描述：2 个生产者进程和 2 个消费者进程共享一个初始为空、大小为 10 的缓冲区，只有缓冲区没满时，生产者才能把消息放入到缓冲区，否则必须等待；只有缓冲区不空时，消费者才能从中取出消息，否则必须等待。由于缓冲区是临界资源，它只允许一个生产者放入消息，或者一个消费者从中取出消息。

生产者和消费者对缓冲区互斥访问是互斥关系，同时生产者和消费者又是一个相互协作的关系，只有生产者生产之后，消费者才能消费，他们也是同步关系。

思路：仿照前面基于信号量解决哲学家就餐问题

首先定义**三个信号量：**

- **mutex** 作为互斥信号量，它用于控制互斥访问缓冲池，互斥信号量初值为 1；
- **full** 用于记录当前缓冲池中“满”缓冲区数，初值为 0；
- **empty** 用于记录当前缓冲池中“空”缓冲区数，初值为 n

在 kern/sync/check_sync.c 中定义如下：

```
#define M 2 //生产者消费者各自数目
#define Num 10 //缓冲区大小
int buff[Num]={0}; //缓冲初始化为0
int in=0;//生产者放置产品的位置
int out=0;//消费者取产品位置

semaphore_t mutex2; //互斥信号量
semaphore_t full; //记录满缓冲区个数
semaphore_t empty; //记录剩余空闲缓冲区个数
struct proc_struct *produce[M];
struct proc_struct *purchase[M];
```

```
28 void print(){
29     int i;
30     for(i=0;i!=Num;i++)
31         cprintf("%d",buff[i]);
32 }
33
```

produce_using_semaphore()

```
34 int produce_using_semaphore(void *arg){
35     int t=(int)arg;//生产者编号
36     int iter=0;
37     while(iter++<TIMES){
38         do_sleep(SLEEP_TIME);
39         down(&empty);
40         down(&mutex2); //进入临界区
41         buff[in]=1; //生产产品到缓冲区in
42         cprintf("Producer%d in%d.like:  ",t,in);
43         print();
44         cprintf("\n");
45         in=(++in)%Num;
46         up(&mutex2); //离开临界区
47         up(&full); //增加已用缓冲区数目
48     }
49     return 0;
50 }
```

其中 TIMES 利用前面哲学家吃饭次数的定义，值为 4
每次循环开始先进入 sleep_time 便于观察；然后空槽数目减 1，进入临界区，in 对应缓冲区设为 1；最后离开临界区，满槽数目加 1。如此循环四组

purchase_using_semaphore()

```
51 int purchase_using_semaphore(void *arg){
52     int t=(int)arg;//消费者编号
53     int iter=0;
54     while(iter++<TIMES){
55         do_sleep(SLEEP_TIME);
56         down(&full);
57         down(&mutex2); //进入临界区
58         buff[out]=0; //从缓冲区in取产品
59         cprintf("Consumer%d in%d.like:  ",t,out);
60         print();
61         cprintf("\n");
62         out=(++out)%Num;
63         up(&mutex2); //离开临界区
64         up(&empty); //增加剩余缓冲区数目
65     }
66     return 0;
67 }
```

与生产类似，先将满槽数目减 1，进入临界区后把 out 对应的缓冲区设为 0
离开临界区，空槽数加 1，循环 4 组

check_sync()中完成主函数

```
261     //check semaphore
262     sem_init(&mutex2, 1);
263     sem_init(&full, 0);
264     sem_init(&empty, Num);
265
```

```
266     for(i=0;i<M;i++){
267         int pid = kernel_thread(produce_using_semaphore, (void *)i, 0);
268         if (pid <= 0) {
269             panic("create No.%d produce_using_semaphore failed.\n");
270         }
271         produce[i] = find_proc(pid);
272         set_proc_name(produce[i], "produce");
273     }
274
275     for(i=0;i<M;i++){
276         int pid = kernel_thread(purchase_using_semaphore, (void *)i, 0);
277         if (pid <= 0) {
278             panic("create No.%d purchase_using_semaphore failed.\n");
279         }
280         purchase[i] = find_proc(pid);
281         set_proc_name(purchase[i], "purchase");
282     }
```

也是仿照之前哲学家问题，调用前面两个函数生产和消费，使用 kernel_thread()新建相应的四个进程（生产者消费者分别有两个）即可

make qemu 部分截图

```
kernel_execve: pid = 2, name = "exit".
I am the parent. Forking the child...
I am parent, fork a child pid 7
I am the parent, waiting now..
I am the child.
waitpid 7 ok.
exit pass.
Producer0 in0.like:  1000000000
Producer1 in1.like:  1100000000
Consumer0 in0.like:  0100000000
Consumer1 in1.like:  0000000000
Producer1 in2.like:  0010000000
Producer0 in3.like:  0011000000
Consumer0 in2.like:  0001000000
Consumer1 in3.like:  0000000000
Producer0 in4.like:  0000100000
```

```
Producer1 in5.like:  0000110000
Consumer1 in4.like:  0000010000
Consumer0 in5.like:  0000000000
Producer1 in6.like:  0000001000
Producer0 in7.like:  0000001100
Consumer0 in6.like:  0000000100
Consumer1 in7.like:  0000000000
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> qemu: terminating on signal 2
```

三、实验感悟

经过本次实验，深刻理解了操作系统的同步互斥概念，以及各种实现方法；同时了解了信号量和管程机制，并应用于经典同步问题哲学家就餐问题和生产者-消费者问题的解决。通过实践巩固了理论课知识。