实验七：同步互斥

1. 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。

2 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。

2.1 练习

练习0：填写已有实验

本实验依赖实验1/2/3/4/5/6。请把你做的实验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。注意：为了能够正确执行lab7的测试应用程序，可能需对已完成的实验1/2/3/4/5/6的代码进行进一步改进。

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）

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lab6和练习0完成后的刚修改的lab7之间的区别，分析了解lab7采用信号量的执行过程。执行make grade，大部分测试用例应该通过。

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）。

执行：make grade 。如果所显示的应用程序检测都输出ok，则基本正确。如果只是某程序过不去，比如matrix.c，则可执行 make run-matrix 命令来单独调试它。大致执行结果可看附录。（**使用的是qemu-1.0.1**）。

扩展练习Challenge：实现Linux的RCU

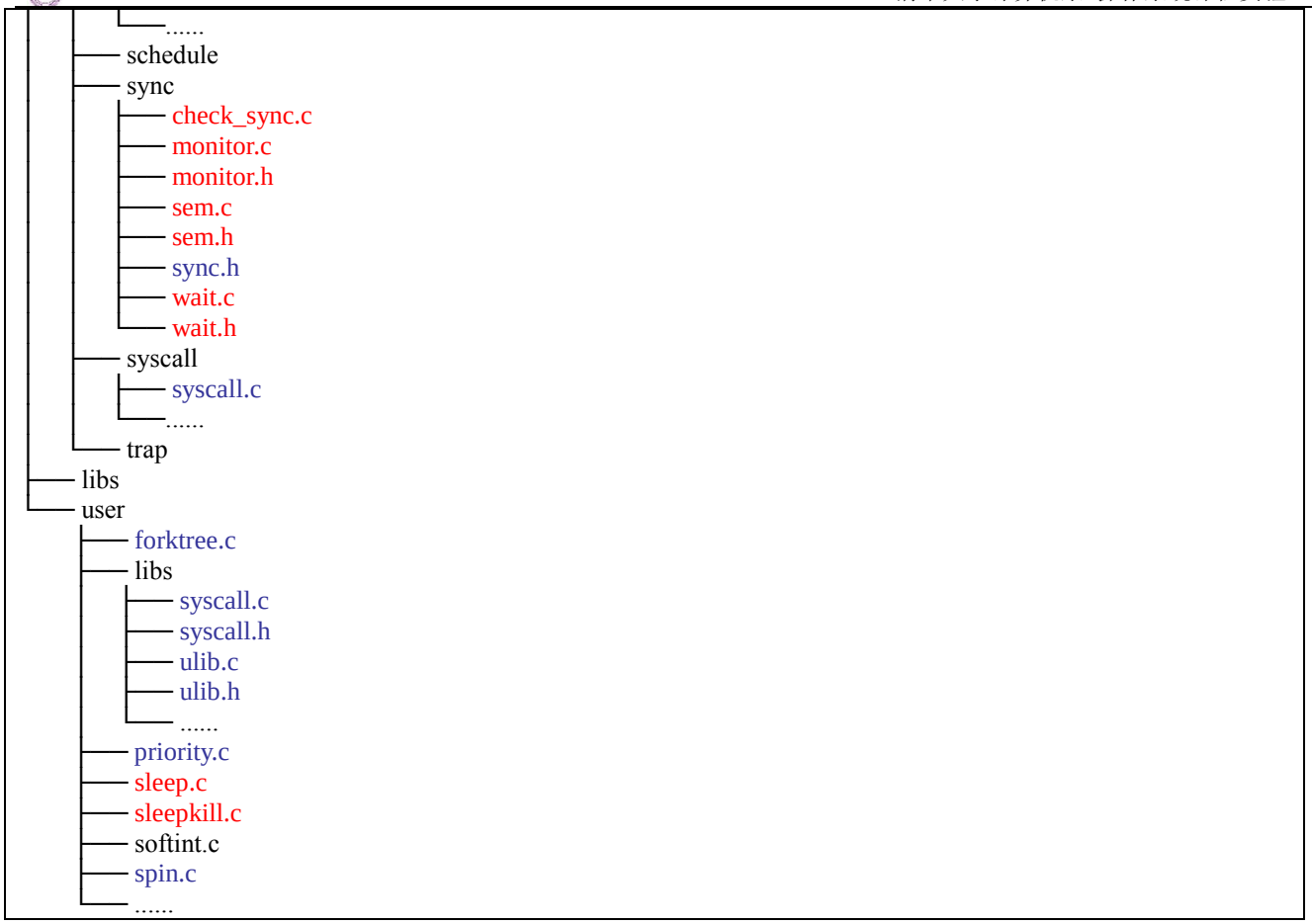
在ucore 下实现下Linux的RCU同步互斥机制。可阅读相关Linux内核书籍或查询网上资料，可了解RCU的细节，然后大致实现在ucore中。下面是一些参考资料：

- http://www.ibm.com/developerworks/cn/linux/l-rcu/
- http://www.diybl.com/course/6_system/linux/Linuxjs/20081117/151814.html

2.2 项目组成

此次实验中，主要有如下一些需要关注的文件：

```
.
├── boot
├── kern
│   ├── driver
│   ├── fs
│   ├── init
│   ├── libs
│   ├── mm
│   │   ├── ......
│   │   ├── vmm.c
│   │   └── vmm.h
│   ├── process
│   │   ├── proc.c
│   │   └── proc.h
```



简单说明如下：

- kern/sync/sync.h: 去除了lock实现（这对于不抢占内核没用）。
- kern/sync/wait.[ch]: 定了为wait结构和waitqueue结构以及在此之上的函数，这是ucore中的信号量semophore机制和条件变量机制的基础，在本次实验中你需要了解其实现。
- kern/sync/sem.[ch]:定义并实现了ucore中内核级信号量相关的数据结构和函数，本次试验中你需要了解其中的实现，并基于此完成内核级条件变量的设计与实现。
- user/ libs/ {syscall.[ch],ulib.[ch] }与kern/sync/syscall.c：实现了进程sleep相关的系统调用的参数传递和调用关系。
- user/{ sleep.c,sleepkill.c}: 进程睡眠相关的一些测试用户程序。
- kern/sync/monitor.[ch]: 基于管程的条件变量的实现程序，在本次实验中是练习的一部分，要求完成。
- kern/sync/check_sync.c：实现了基于管程的哲学家就餐问题，在本次实验中是练习的一部分，要求完成基于管程的哲学家就餐问题。
- kern/mm/vmm.[ch]：用信号量mm_sem取代mm_struct中原有的mm_lock。（本次实验不用管）

3 同步互斥的设计与实现

3.1 同步互斥的底层支撑

在ucore中提供的底层机制包括中断开关控制和原子操作。kern/sync.c中实现的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。在atomic.c文件中实现的test_and_set_bit等原子操作。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。

到目前为止，我们的实验中，用户进程或内核线程还没有睡眠的支持机制。在课程中提到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撑机制就是等待队列（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（runnable state），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，直到他们被唤醒。

```
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，相关函数如下：

```
void wait_init(wait_t *wait, struct proc_struct *proc);       //初始化 wait 结构
bool wait_in_queue(wait_t *wait);                                    //wait 是否在 wait queue 中
void wait_queue_init(wait_queue_t *queue);                          //初始化 wait_queue 结构
void wait_queue_add(wait_queue_t *queue, wait_t *wait);       //把 wait 前插到 wait queue 中
void wait_queue_del(wait_queue_t *queue, wait_t *wait);       //从 wait queue 中删除 wait
wait_t *wait_queue_next(wait_queue_t *queue, wait_t *wait);//取得 wait 的后一个链接指针
wait_t *wait_queue_prev(wait_queue_t *queue, wait_t *wait);//取得 wait 的前一个链接指针
wait_t *wait_queue_first(wait_queue_t *queue);                     //取得 wait queue 的第一个 wait
wait_t *wait_queue_last(wait_queue_t *queue);                      //取得 wait queue 的最后一个 wait
bool wait_queue_empty(wait_queue_t *queue);                        //wait queue 是否为空
```

高层函数基于底层函数实现了让进程进入等待队列，以及从等待队列中唤醒进程，相关函数如下：

```
//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);
//把与当前进程关联的 wait 从等待队列 queue 中删除
wait_current_del(queue, wait);
//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);
```

3.2 信号量

信号量是一种同步互斥机制的实现，普遍存在于现在的各种操作系统内核里。相对于spinlock 的应用对象，信号量的应用对象是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中信号量的实现是在Wait Queue的基础上来实现，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（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。

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)。但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, timer_t *timer)：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(semaphore_t *sem, uint32_t wait_state)：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。

在以上我们的分析的数据结构和函数实现的基础上，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

3.3 管程和条件变量

引入了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。Hansan为管程所下的定义：“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”。有上述定义可知，管程由四部分组成：

- 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。

局限在管程中的数据结构，只能被局限在管程的操作过程所访问，任何管程之外的操作过程都不能访问它；另一方面，局限在管程中的操作过程也主要访问管程内的数据结构。由此可见，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。ucore中的管程机制是基于信号量和条件变量来实现的。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;
```

ucore中的条件变量实现了用户进程同步的cond_wait、cond_signal函数，此外还有cond_init初始化函数（可直接看源码）。函数cond_wait(condvar_t *cvp, semaphore_t *mp)和cond_signal (condvar_t *cvp)的实现原理可参考《OS Concept》一书中的6.7.3小节“用信号量实现管程”的内容：

cond_wait 的原理参考 x.wait　　cond_signal 的原理参考 x.singnal

```
    x_count++;
    if (next_count > 0)
      signal(next);
    else
      signal(mutex);
    wait(x_sem);
    x_count--;
```

```
if (x_count > 0) {
  next_count++;
  signal(x_sem);
  wait(next);
  next_count--;
}
```

需要注意的是，上述只是原理描述，与具体描述相比，还有一定的差距。需要大家在完成练习时仔细设计和实现。

3.4 测试用例

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，自己实现基于管程（主要是灵活运用条件变量）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。需要注意，对于基于信号量的原理参考，请看课程的ppt和本实验的源码；对于基于管程的原理参考，请参考《OS Concept》一书中的6.7.2小节“用信号量解决哲学家就餐问题”的内容：

```
monitor dp
{
   enum {THINKING, HUNGRY, EATING}state[5];
   condition self[5];

   void pickup(int i) {
      state[i] = HUNGRY;
      test(i);
      if (state[i] != EATING)
         self[i].wait();
   }

   void putdown(int i) {
      state[i] = THINKING;
      test((i + 4) % 5);
      test((i + 1) % 5);
   }

   void test(int i) {
      if ((state[(i + 4) % 5] != EATING) &&
       (state[i] == HUNGRY) &&
       (state[(i + 1) % 5] != EATING)) {
         state[i] = EATING;
         self[i].signal();
      }
   }

   initialization_code() {
      for (int i = 0; i < 5; i++)
         state[i] = THINKING;
    }
}
```

Figure 6.19 A monitor solution to the dining-philosopher problem.

4 实验报告要求

从网站上下载 lab7.zip 后，解压得到本文档和代码目录 lab7，完成实验中的各个练习。完成代码编写并检查无误后，在对应目录下执行 make handin 任务，即会自动生成 lab7-handin.tar.gz。最后请一定提前或按时提交到网络学堂上。

注意有“LAB7”的注释，主要是修改 condvar.c 和 check_sync.c 中的内容。代码中所有需要完成的地方 challenge 除外）都有“LAB7”和“YOUR CODE”的注释，请在提交时特别注意保持注释，并将“YOUR CODE”替换为自己的学号，并且将所有标有对应注释的部分填上正确的代码。

附录：执行"make run-matrix"的大致的显示输出

```
(THU.CST) os is loading ...

Special kernel symbols:
    entry    0xc010002c (phys)
    etext    0xc0110a77 (phys)
    edata    0xc01cb309 (phys)
    end      0xc01ce8c4 (phys)
Kernel executable memory footprint: 827KB
memory management: default_pmm_manager
e820map:
    memory: 0009f400, [00000000, 0009f3ff], type = 1.
    memory: 00000c00, [0009f400, 0009ffff], type = 2.
    memory: 00010000, [000f0000, 000fffff], type = 2.
    memory: 07efd000, [00100000, 07ffcfff], type = 1.
    memory: 00003000, [07ffd000, 07ffffff], type = 2.
    memory: 00040000, [fffc0000, ffffffff], type = 2.
check_alloc_page() succeeded!
check_pgdir() succeeded!
check_boot_pgdir() succeeded!
-------------------- BEGIN --------------------
PDE(0e0) c0000000-f8000000 38000000 urw
  |-- PTE(38000) c0000000-f8000000 38000000 -rw
PDE(001) fac00000-fb000000 00400000 -rw
  |-- PTE(000e0) faf00000-fafe0000 000e0000 urw
  |-- PTE(00001) fafeb000-fafec000 00001000 -rw
--------------------- END ---------------------
check_slab() succeeded!
kmalloc_init() succeeded!
check_vma_struct() succeeded!
page fault at 0x00000100: K/W [no page found].
valid addr 100, and find it in vma range[0, 400000]
check_pgfault() succeeded!
check_vmm() succeeded.
sched class: stride_scheduler
ide 0:      10000(sectors), 'QEMU HARDDISK'.
ide 1:     262144(sectors), 'QEMU HARDDISK'.
SWAP: manager = fifo swap manager
BEGIN check_swap: count 31786, total 31786
setup Page Table for vaddr 0X1000, so alloc a page
setup Page Table vaddr 0~4MB OVER!
set up init env for check_swap begin!
page fault at 0x00001000: K/W [no page found].
valid addr 1000, and find it in vma range[1000, 6000]
page fault at 0x00002000: K/W [no page found].
valid addr 2000, and find it in vma range[1000, 6000]
page fault at 0x00003000: K/W [no page found].
valid addr 3000, and find it in vma range[1000, 6000]
page fault at 0x00004000: K/W [no page found].
valid addr 4000, and find it in vma range[1000, 6000]
set up init env for check_swap over!
write Virt Page c in fifo_check_swap
write Virt Page a in fifo_check_swap
write Virt Page d in fifo_check_swap
write Virt Page b in fifo_check_swap
write Virt Page e in fifo_check_swap
page fault at 0x00005000: K/W [no page found].
valid addr 5000, and find it in vma range[1000, 6000]
swap_out: i 0, store page in vaddr 0x1000 to disk swap entry 2
```

```
write Virt Page b in fifo_check_swap
write Virt Page a in fifo_check_swap
page fault at 0x00001000: K/W [no page found].
valid addr 1000, and find it in vma range[1000, 6000]
do pgfault: ptep c03d5004, pte 200
swap_out: i 0, store page in vaddr 0x2000 to disk swap entry 3
swap_in: load disk swap entry 2 with swap_page in vadr 0x1000 free_area.nr_free 0
write Virt Page b in fifo_check_swap
page fault at 0x00002000: K/W [no page found].
valid addr 2000, and find it in vma range[1000, 6000]
do pgfault: ptep c03d5008, pte 300
swap_out: i 0, store page in vaddr 0x3000 to disk swap entry 4
swap_in: load disk swap entry 3 with swap_page in vadr 0x2000 free_area.nr_free 0
write Virt Page c in fifo_check_swap
page fault at 0x00003000: K/W [no page found].
valid addr 3000, and find it in vma range[1000, 6000]
do pgfault: ptep c03d500c, pte 400
swap_out: i 0, store page in vaddr 0x4000 to disk swap entry 5
swap_in: load disk swap entry 4 with swap_page in vadr 0x3000 free_area.nr_free 0
write Virt Page d in fifo_check_swap
page fault at 0x00004000: K/W [no page found].
valid addr 4000, and find it in vma range[1000, 6000]
do pgfault: ptep c03d5010, pte 500
swap_out: i 0, store page in vaddr 0x5000 to disk swap entry 6
swap_in: load disk swap entry 5 with swap_page in vadr 0x4000 free_area.nr_free 0
count is 7, total is 7
check_swap() succeeded!
++ 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".
pid 14 is running (1000 times)!.
pid 13 is running (1000 times)!.
phi_test_condvar: state_condvar[4] will eating
phi_test_condvar: signal self_cv[4]
Iter 1, No.4 philosopher_condvar is eating
phi_take_forks_condvar: 3 didn't get fork and will wait
phi_test_condvar: state_condvar[2] will eating
phi_test_condvar: signal self_cv[2]
Iter 1, No.2 philosopher_condvar is eating
phi_take_forks_condvar: 1 didn't get fork and will wait
phi_take_forks_condvar: 0 didn't get fork and will wait
pid 14 done!.
pid 13 done!.
Iter 1, No.4 philosopher_sema is eating
Iter 1, No.2 philosopher_sema is eating
fork ok.
phi_test_condvar: state_condvar[0] will eating
phi_test_condvar: signal self_cv[0]
Iter 2, No.4 philosopher_condvar is thinking
phi_test_condvar: state_condvar[3] will eating
phi_test_condvar: signal self_cv[3]
Iter 2, No.2 philosopher_condvar is thinking
Iter 1, No.0 philosopher_condvar is eating
Iter 1, No.3 philosopher_condvar is eating
```

```
pid 32 is running (26600 times)!.
pid 33 is running (13100 times)!.
pid 31 is running (2600 times)!.
pid 30 is running (13100 times)!.
Iter 2, No.4 philosopher_sema is thinking
phi_take_forks_condvar: 4 didn't get fork and will wait
Iter 2, No.3 philosopher_condvar is thinking
Iter 2, No.2 philosopher_sema is thinking
Iter 1, No.3 philosopher_sema is eating
phi_test_condvar: state_condvar[2] will eating
phi_test_condvar: signal self_cv[2]
Iter 2, No.2 philosopher_condvar is eating
phi_test_condvar: state_condvar[4] will eating
phi_test_condvar: signal self_cv[4]
Iter 2, No.0 philosopher_condvar is thinking
Iter 2, No.4 philosopher_condvar is eating
Iter 1, No.0 philosopher_sema is eating
phi_take_forks_condvar: 3 didn't get fork and will wait
Iter 2, No.3 philosopher_sema is thinking
Iter 3, No.4 philosopher_condvar is thinking
phi_test_condvar: state_condvar[1] will eating
phi_test_condvar: signal self_cv[1]
phi_test_condvar: state_condvar[3] will eating
phi_test_condvar: signal self_cv[3]
Iter 2, No.2 philosopher_sema is eating
Iter 3, No.2 philosopher_condvar is thinking
Iter 1, No.1 philosopher_condvar is eating
Iter 2, No.3 philosopher_condvar is eating
phi_take_forks_condvar: 0 didn't get fork and will wait
Iter 2, No.0 philosopher_sema is thinking
Iter 2, No.4 philosopher_sema is eating
pid 30 done!.
phi_test_condvar: state_condvar[0] will eating
phi_test_condvar: signal self_cv[0]
Iter 3, No.4 philosopher_sema is thinking
Iter 3, No.2 philosopher_sema is thinking
Iter 1, No.1 philosopher_sema is eating
Iter 2, No.3 philosopher_sema is eating
Iter 2, No.1 philosopher_condvar is thinking
phi_take_forks_condvar: 4 didn't get fork and will wait
Iter 3, No.3 philosopher_condvar is thinking
phi_test_condvar: state_condvar[2] will eating
phi_test_condvar: signal self_cv[2]
Iter 3, No.2 philosopher_condvar is eating
Iter 2, No.0 philosopher_condvar is eating
pid 15 is running (1100 times)!.
pid 19 is running (20600 times)!.
phi_take_forks_condvar: 1 didn't get fork and will wait
phi_take_forks_condvar: 3 didn't get fork and will wait
Iter 2, No.1 philosopher_sema is thinking
phi_test_condvar: state_condvar[3] will eating
phi_test_condvar: signal self_cv[3]
Iter 3, No.3 philosopher_sema is thinking
Iter 3, No.2 philosopher_sema is eating
Iter 4, No.2 philosopher_condvar is thinking
Iter 3, No.3 philosopher_condvar is eating
phi_test_condvar: state_condvar[1] will eating
phi_test_condvar: signal self_cv[1]
Iter 3, No.0 philosopher_condvar is thinking
Iter 2, No.1 philosopher_condvar is eating
Iter 2, No.0 philosopher_sema is eating
Iter 3, No.1 philosopher_condvar is thinking
phi_test_condvar: state_condvar[4] will eating
phi_test_condvar: signal self_cv[4]
Iter 4, No.3 philosopher_condvar is thinking
Iter 3, No.4 philosopher_condvar is eating
phi_test_condvar: state_condvar[2] will eating
phi_test_condvar: signal self_cv[2]
Iter 4, No.2 philosopher_condvar is eating
Iter 4, No.2 philosopher_sema is thinking
Iter 3, No.3 philosopher_sema is eating
phi_take_forks_condvar: 0 didn't get fork and will wait
Iter 3, No.0 philosopher_sema is thinking
```

```
Iter 2, No.1 philosopher_sema is eating
phi_test_condvar: state_condvar[0] will eating
phi_test_condvar: signal self_cv[0]
Iter 4, No.3 philosopher_sema is thinking
Iter 3, No.4 philosopher_sema is eating
Iter 4, No.4 philosopher_condvar is thinking
phi_take_forks_condvar: 1 didn't get fork and will wait
phi_take_forks_condvar: 3 didn't get fork and will wait
phi_test_condvar: state_condvar[3] will eating
phi_test_condvar: signal self_cv[3]
No.2 philosopher_condvar quit
Iter 3, No.0 philosopher_condvar is eating
Iter 4, No.3 philosopher_condvar is eating
Iter 3, No.1 philosopher_sema is thinking
Iter 4, No.2 philosopher_sema is eating
pid 19 done!.
phi_take_forks_condvar: 4 didn't get fork and will wait
Iter 4, No.4 philosopher_sema is thinking
No.3 philosopher_condvar quit
phi_test_condvar: state_condvar[4] will eating
phi_test_condvar: signal self_cv[4]
phi_test_condvar: state_condvar[1] will eating
phi_test_condvar: signal self_cv[1]
No.2 philosopher_sema quit
Iter 4, No.3 philosopher_sema is eating
Iter 4, No.0 philosopher_condvar is thinking
Iter 4, No.4 philosopher_condvar is eating
Iter 3, No.1 philosopher_condvar is eating
Iter 3, No.0 philosopher_sema is eating
pid 23 is running (37100 times)!.
No.4 philosopher_condvar quit
Iter 4, No.1 philosopher_condvar is thinking
No.3 philosopher_sema quit
phi_test_condvar: state_condvar[0] will eating
phi_test_condvar: signal self_cv[0]
Iter 4, No.0 philosopher_condvar is eating
Iter 4, No.0 philosopher_sema is thinking
Iter 4, No.4 philosopher_sema is eating
Iter 3, No.1 philosopher_sema is eating
No.4 philosopher_sema quit
phi_take_forks_condvar: 1 didn't get fork and will wait
phi_test_condvar: state_condvar[1] will eating
phi_test_condvar: signal self_cv[1]
No.0 philosopher_condvar quit
Iter 4, No.1 philosopher_condvar is eating
Iter 4, No.1 philosopher_sema is thinking
Iter 4, No.0 philosopher_sema is eating
No.1 philosopher_condvar quit
No.0 philosopher_sema quit
Iter 4, No.1 philosopher_sema is eating
No.1 philosopher_sema quit
pid 27 is running (23500 times)!.
pid 16 is running (1900 times)!.
pid 18 is running (11000 times)!.
pid 17 is running (4600 times)!.
pid 21 is running (2600 times)!.
pid 21 done!.
pid 25 is running (23500 times)!.
pid 20 is running (37100 times)!.
pid 22 is running (13100 times)!.
pid 24 is running (4600 times)!.
pid 26 is running (2600 times)!.
pid 26 done!.
pid 28 is running (4600 times)!.
pid 29 is running (33400 times)!.
pid 28 done!.
pid 24 done!.
pid 17 done!.
pid 31 done!.
pid 16 done!.
pid 15 done!.
pid 18 done!.
pid 23 done!.
```

```
pid 22 done!.
pid 33 done!.
pid 27 done!.
pid 25 done!.
pid 32 done!.
pid 29 done!.
pid 20 done!.
matrix pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:426:
        initproc exit.

Welcome to the kernel debug monitor!!
Type 'help' for a list of commands.
K> qemu: terminating on signal 2
```