

实验 5：进程同步

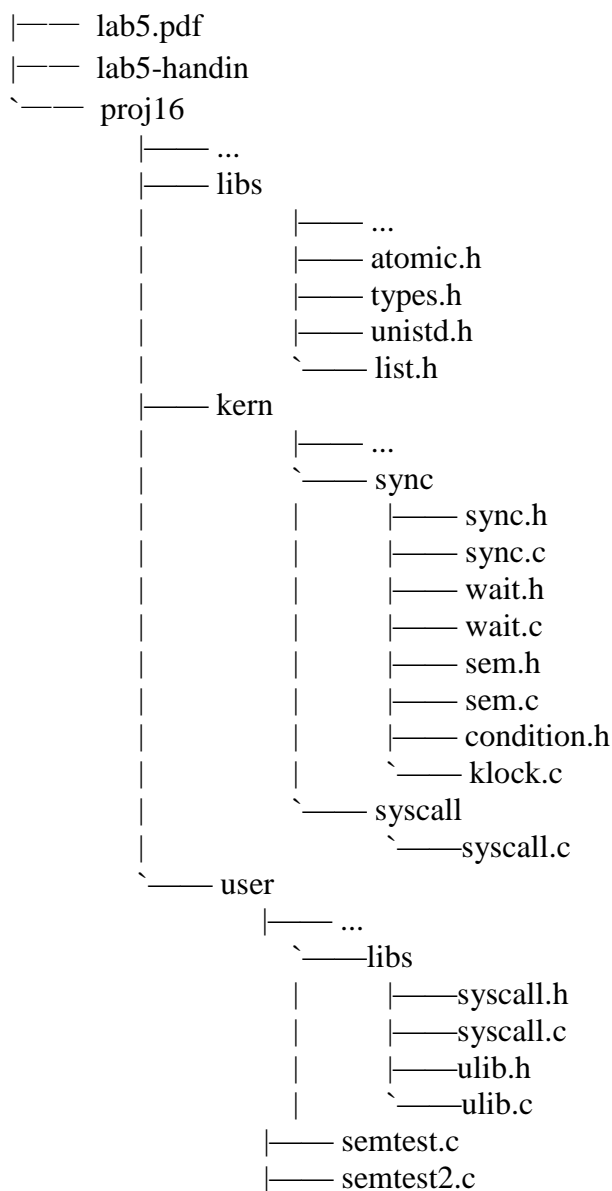
1. 实验目的

- 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 中实现信号量（semaphore）机制；
- 理解管程机制，在 ucore 中增加对条件变量（condition variable）的支持；
- 了解经典进程同步问题，并使用现有同步机制解决进程同步问题；

2. 程序清单

此次实验中，主要有如下一些需要关注的文件：

Lab5



```

|—— semtest3.c
|—— semtest4.c
|—— sem_rf.c
|—— sem_wf.c
|—— cdt_pc.c
|—— cdt_wf.c
\—— ...

```

简单说明如下：

- `libs/atomic`: 定义了一些原子操作，是进程同步的基础。
- `libs/types.h`: 提供了本次试验中需要使用数据类型的定义以及类型转换的参数。
- `libs/unistd.h`: 提供了目前系统中已经声明的系统调用号。
- `libs/list.h`: 提供了双向列表相关的定义和函数。
- `kern/sync/{ sync.h, sync.c }`: 提供了内核控制开关中断机制。
- `kern/sync/{ wait.h, wait.c }`: 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 `semaphore` 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.h, sem.c }`: 定义并实现了 `ucore` 中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，并补充其中空白的部分实现机制。
- `user/ libs/ {syscall.h, syscall.c, ulib.h , ulib.c }`与 `kern/sync/syscall.c`: 实现了进程相关的系统调用的参数传递和调用关系，与实验 3 中的系统调用类似。
- `user/{ semtest.c, semtest2.c, semtest3.c, semtest4.c, sem_rf.c, sem_wf.c }`: 信号量相关的一些测试用户程序，测试进程同步机制算法的正确性。
- `user/{ cdt_pc.c, cdt_wf.c }`: 管程相关的一些用户测试程序，并应用管程的同步机制实现写者优先的读者写者问题，在本次实验中要求补充其中空白的部分实现机制。

`user` 目录下包含但不限于以上程序。在完成实验过程中，建议阅读这些测试程序，以了解这些程序的行为，便于进行调试。

3. 准备知识

3.1 进程间的共享变量

在“内存管理”中我们已讨论过进程间的内存共享问题。其基本原理是，把一段内存区域同时映射到多个进程的地址空间中，使多个进程可以在自己的地址空间中访部相同的共享

内存区域。

在进程同步实验中需要在多个进程之间实现变量的共享，也就需要使用 ucore 中的内存共享机制。在 ucore 中为了支持不同进程之间的内存共享，定义了 shmem_struct 结构，在此结构上使用

shmem_create(), shmem_destroy(), shmem_get_entry(), shmem_insert_entry() 等函数结合内核态信号量机制实现对共享内存的管理。

在用户态 ucore 提供了 ulib 中的函数：

```
int shmem(uintptr_t *addr_store, size_t len, uint32_t mmap_flags);
```

该函数通过系统调用 sys_shmem(uintptr_t *addr_store, size_t len, uint32_t mmap_flags) 实现共享内存的申请，并返回一个指针指向不同进程可共享的一块内存区域的首地址。用户进程获得这个共享内存的首地址，就可通过普通的内存访问对共享内存进行读写操作。

3.2 同步互斥

任何为进程所占用的实体都可称为资源。资源可以是物理实体，比如 CPU、内存，也可以是 I/O 设备，还可以是逻辑实体，包括在内存中的全局变量等。可以被一个以上进程使用的资源叫做共享资源。为了防止数据被随意访问（特别是执行写操作），每个进程在与共享资源打交道时，需要独占该资源，这叫做互斥（Mutual exclusion）。需要互斥访问的共享资源称为临界资源。

如果多个进程在执行过程中需要遵守一定的逻辑时序，则需要采用一定的机制来保证进程执行过程中的先后顺序，这种机制称为同步（Synchronization）。如果两个以上进程对同一共享资源同时进行读写操作，且没有互斥或同步保证，则最后的结果是不可预测的，该结果取决于各个进程具体运行情况，这种状态被称为竞争状态（Race Condition）。我们把可能造成竞争状态出现的程序片断称为程序临界区。程序临界区在处理时不可以被中断，要保证其操作的原子性。为确保临界区程序执行过程中不被中断，在进入临界区之前要屏蔽中断，而临界区代码执行完以后要立即使能中断，以减少对中断处理延迟的影响。操作系统需要提供一整套的层次型的同步互斥的机制来解决上述问题，如图 1：

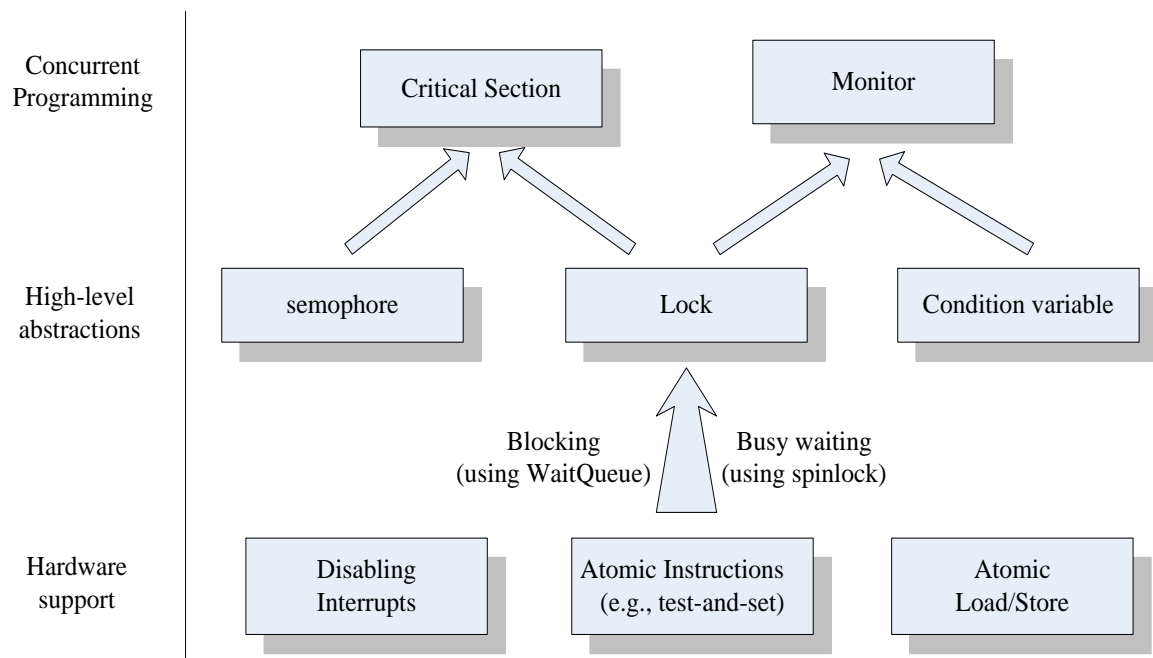


图 1 操作系统中的进程同步进程

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。

在此之上为了提供对上层高级抽象的支持，ucore 在 kern/sync/{ wait.h, wait.c } 中实现了 Wait 结构和 WaitQueue 结构以及相关函数（具体内容见下图），这是 ucore 中的信号量机制和条件变量机制的基础，进入 WaitQueue 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

```

由于 spinlock 在试图锁定的时候, 如果不成功会不断的尝试占用 CPU, 单 cpu 上的 spinlock 会影响其他进程运行, 所以在目前的 ucore 代码中没有实现高级抽象层之下 spinlock, 用户态的 lock 和内核态的 klock 都是基于 test_and_set_bit 等原子操作, 实现类似 spinlock 的工作。

3.3 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 中的信号量的实现是在 WaitQueue 的基础上来实现, 数据结构定义如下:

```
typedef struct {
    int value;                //记录型信号量的当前值
    bool valid;               //标记信号量是否有效
    atomic_t count;           //信号量的引用数量
    wait_queue_t wait_queue;  //信号量对应的等待队列
} semaphore_t;

typedef struct sem_undo {
    semaphore_t *sem;
    list_entry_t semu_link;
} sem_undo_t;

typedef struct sem_queue {
    semaphore_t sem;
    atomic_t count;
    list_entry_t semu_list;
} sem_queue_t;

struct proc_struct {
    ...

    uint32_t wait_state;      // 进程进入睡眠状态的原因
    sem_queue_t *sem_queue;   // 进程的信号量队列
    ...
}
```

在以上三个结构中，semaphore_t 是最基本的信号量结构，sem_undo_t 数据结构的定义主要是为了实现主要是系统稳定性的考虑，semaphore_t 型对象的处理都是在 sem_undo_t 对象之后来做的，就可以在 sem_undo_t 中保存相应的信息，semaphore_t 型对象操作失败时进行必要的回复，不过目前 ucore 中的实现很弱，主要的作用是帮助对 semaphore_t 链表进行组织。sem_queue_t 是 ucore 中信号量的链表组织，proc_struct 中的 sem_queue 指针指向进程实际申请的信号量，这样就是实现了信号量与进程的关联。

在 ucore 中最重要的信号量操作是：

- `__up(semaphore_t *sem, uint32_t wait_state)`：实现信号量的 V 操作，首先检查信号量的有效性，然后关中断，如果信号量对应的 Waitqueue 中没有进程在等待，直接修改信号量的 value 值即可，如果有进程在等待且进程等待的原因是 semaphore 设置的，则将 waitqueue 中等待的第一个进程唤醒并将 waitqueue 中对应的项删除，最后恢复中断设置。
- `__down(semaphore_t *sem, uint32_t wait_state, timer_t *timer)`：实现信号量的 P 操作，首先检查信号量的有效性，然后关掉中断，如果当前信号量的 value 大于 0，那么之间做 value 的修改并打开中断返回即可，否则需要将当前的进程加入到等待队列中，运行调度器选择另外一个进程执行。由于 P 操作中还有一个 timer 参数表示最长等待时间，所以在 __down 函数中还有一种可能是进程并非被 V 操作唤醒而是因为超时而发生的，所以在进程被唤醒重新执行时需要考虑这两种情况并作出相应的处理。

在以上我们的分析的数据结构和函数的基础上，我们可以看到从以上分析可以看出，信号量的 value 具有有如下性质：

- 计数器值为正，表示共享资源的空闲数
- 计数器值为负，表示该信号量的等待队列里的进程数
- 计数器为零，表示等待队列为空

ucore 中的信号量机制目前主要提供对外的函数接口包括如下内容：

```
int ipc_sem_init(int value);           //初始化
int ipc_sem_post(sem_t sem_id);        //signal
int ipc_sem_wait(sem_t sem_id, unsigned int timeout); //wait
int ipc_sem_free(sem_t sem_id);        //释放信号量
int ipc_sem_get_value(sem_t sem_id, int *value_store); //获取信号量 value 值
```

3.4 条件变量和管程

使用信号量和 wait、signal 操作实现同步时，对共享资源的管理分散在各个进程中，进程能够之间对共享变量进行修改，这不利于系统对于临界资源的管理，这不但容易造成程序设计的错误而且难以防止进程有意的违法同步操作，造成系统的安全隐患。所以引入了管程的概念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。Hanson 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。

局部于管程的数据结构，只能被局部于管程的过程所访问，任何管程之外的过程都不能访问它；反之，局部于管程的过程也只能访问管程内的数据结构。由此可见，管程相当于围墙，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，从而实现了进程的互斥。

ucore 中的管程是基于 klock_t 和条件变量 condition_t 来实现的。klock 的实现现在 3.2 节的最后已经说过，这里仅仅说明一些对外提供的函数接口：

- int sys_lock_init() //初始化函数
- int sys_lock_free(klock_t klock_id) //内核空闲释放函数
- int sys_lock(klock_t klock_id) //锁定函数
- int sys_unlock(klock_t klock_id) //解锁函数

ucore 中 condition 基于等待队列 WaitQueue 实现，定义 struct condition_t，为了使用户进程也能对 condition 进行操作，在 types.h 文件中还定义了 cdt_t 类型，两者的转换是 condition.h 中的 cdtid2cdt(cdt_id)、cdt2cdtid(cdt)实现的。

```
typedef struct {
    int numWaiting;           //等待的进程数
    int valid;                //标记变量是否有效
    wait_queue_t wait_queue;  //等待队列
}condition_t
```

<code>typedef uintptr_t cdt_t; //用户态对 condition 变量进行控制的指针</code>
--

ucore 中的条件变量实现了用户进程同步的 wait、signal 函数，此外还有初始化和释放函数。

- `int condition_wait(cdt_t cdt_id, klock_t kl_id)`: 将 `cdt_id`, `kl_id` 转化为 condition 变量和 `klock` 的内核态地址，然后将进程添加到等待队列中并释放目前获得的内核锁，最后还要运行调度器选择下一个要执行的进程，这一过程是不允许中断的。
- `int condition_signal(cdt_t cdt_id)`: 首先也是要将 `cdt_id` 转化为 condition 变量的内核态地址，如果目前的有进程在条件变量的等待队列之上的话，就从等待对类中取下一个进程，将这个进程唤醒，并修改 `numWaiting` 的值，这一过程也是不允许中断的。

3.5 读者写者问题

如果现在有多个进程都需要对内存中的同一块区域进行读写操作，则我们需要考虑同步互斥问题。我们把这些进程抽象成两类：

- 只进行读操作的进程，称之为“读者”
- 只进行写操作的进程，称之为“写者”

如何保证若干个读者和写者对共享内存的访问的同步互斥，就是读者写者问题。所有读者和写者都必须满足下面的限制条件：

- 多个读者可以同时进行读操作，这时写者只能等待；
- 任何写者在进行写操作时，其他读者和写者都必须等待。

在上述限制下，还有一些可选的限制条件。如下面的读者优先和写者优先。读者写者问题有多种，基本的是以下两种：

- 读者优先：当有读者正在进行读操作时，新到达的读者可以开始读操作，而不需要等待，即使在此之前已经有写者在等待；
- 写者优先：读者仍然可以同时读，但是当有写者处于等待状态时，还没有进入临界区的读者必须等待所有的写者完成写操作后才能开始读操作，即使某写请求的到达时间晚于自己；

3.6 测试代码

`user` 目录下定义了若干用户程序。提供的实验框架内，OS 会先创建 `idle` 进程，然后再加载指定的用户代码来创建用户进程。可以在跟目录下指定在本次实验中 OS 需要加载的用户程序。比如说，假设需要 OS 加载和执行用户程序 `matrix`，则在项目的根目录下使用

```
make build -matrix
```

则可以生成一个对应的操作系统镜像，该镜像会加载 `matrix.c` 作为初始用户程序。接下来就和之前的 LAB 一样，比如

```
make qemu
```

则可运行这个 OS 镜像。

`user` 目录中提供了测试你所实现的信号量是否正确的测试代码：

`sem_test.c` 文件中的 `sem_test(void)` 主要用来测试信号量的 `wait`、`signal` 操作对信号量计数值的修改是否成功，并检查子进程是否可以利用父进程申请的信号量。

`sem_test2.c` 文件中的 `test1(void)` 函数测试信号量实现的互斥操作。`Test2(void)` 函数用来测试进程之间的前驱后继关系。

`sem_test3.c` 文件中主要测试信号量 `wait` 操作中对于超时的处理。

`sem_test4.c` 文件中主要测试的信号量的 `free` 操作是否实现完善。

`sem_rf.c` 和 `sem_wf.c` 两个文件中用信号量分别实现了读者优先的读者写者问题和写者优先的读者写者问题，文件中分步测试全读者、全写者和读者写者同时存在的读者写者问题的测试。

`cdt_pc.c` 文件中利用条件变量管程机制解决生产者消费者问题。

`cdt_wf.c` 文件中利用条件变量管程机制解决写者优先的读者写者问题。

4. 实验任务及帮助

请按下面顺序依次进行本次实验。每一步的任务都对后续步骤的任务有重要影响。

4.1 代码分析实验

(1) 在 `sem.c` 文件中的 `__down` 函数中被 `block` 的进程是如何被唤醒的？被唤醒的方式可能有几种，他们在被唤醒的时刻是否还在等待队列 `WaitQueue` 中？

(2) 分析在现有的 `ucore` 中，`semaphore` 机制有几种可能的回收方式，他们分别涉及到那些函数，又是何时被调用的。

4.2 代码补全实验

(1) 补全文件“sem.c”中的__up、__down 两个函数的实现：这两个函数是信号量机制的核心，在前面的知识准备中，同学们可以认识到目前内核中提供的 WaitQueue 机制和中断开关是信号量机制的实现的基础。此外__down 函数中还涉及到 timer 的应用，这也是需要考虑的问题。

(2) 补全文件“condition.h”中的 condition_wait、condition_signal 两个函数的实现：ucore 中 condition 变量的机制也是基于 WaitQueue 来实现的，与信号量机制不同的是，执行 condition_wait 的进程不许用判定 numWaiting 值就直接加入等倒带队列中，condition 变量中的 numWaiting 标记等待的进程数，只能是正值。

(3) 补全文件“sem_wf.c”中写者优先读者（reader）和写者（writer）函数，并说明的实现思路；

(4) 补全文件“cdt_wf.c”中写者优先读者（reader）和写者（writer）相关的函数，并说明的实现思路；

文件 sem_rf.c 中的函数 init、reader 和 writer 给出读者优先的读者写者问题实现。cdt_pc.c 中利用条件变量和管程实现了生产者消费者问题，此外操作系统上课的课件上有一个管程实现读者写者问题，这些都可以作为以上代码补全任务的借鉴。

在已经存在的文件中，所有需要补完的代码位置可以通过如下命令找到：

```
grep "LAB5: YOUR CODE" * -ir --color
```

4.3 扩展实验 (optional)

(1) 目前 ucore 中锁机制的实现是基于 test_and_set 实现的，而锁机制的也可以基于信号量来实现，请以管程实现的写者优先的读者写者问题为例，说明这两类锁机制的实现在执行结果上是否可能不同，修改 ucore 中内核锁 klock 的实现方式并验证你的推测；

(2) 分析目前 ucore 中的条件变量与信号量回收机制的不同，借鉴目前信号量的回收机制改进条件变量的回收机制，说明需要修改了哪些数据结构和函数，利用并修改目前的用户程序验证你的工作。

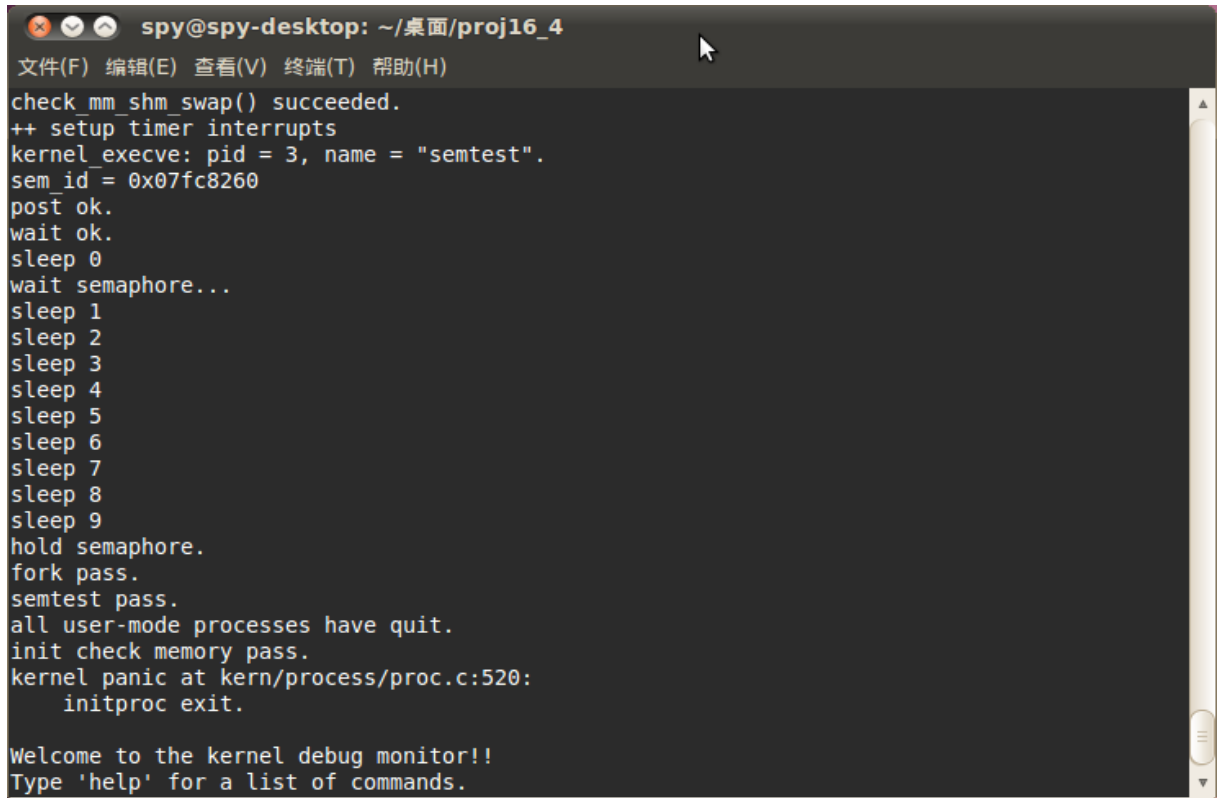
(3) 目前 ucore 中的信号量是匿名信号量，信号量在进程之间的传递和共享相比有名信号量实现较为困难，请在 ucore 信号量代码的基础上实现有名信号量机制，说明需要修改的数据结构和函数，写一个用户程序验证你的工作。

(4) 实现 hoare 机制的管程。

4.3 实验结果

ucore 中提供了多个测试程序，以下列出了一些测试程序的结果供参考。但是这里列出的某些测试结果并不是完全确定的，可能由于调度顺序变化而有一些小的变化。

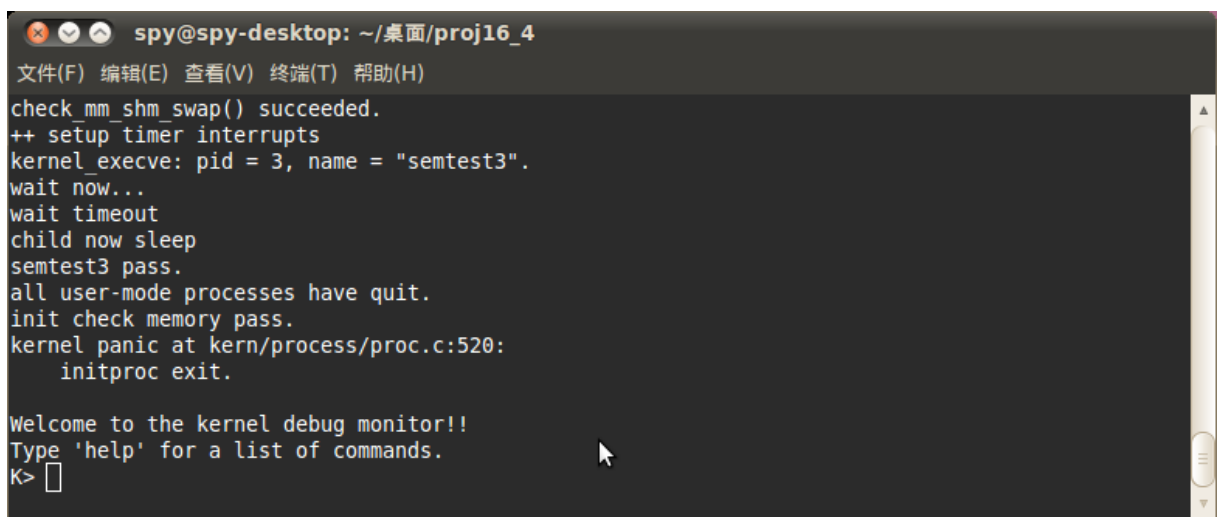
在完成补全代码任务（1）后可以使用 semtest 和 semtest3 测试信号量的基本实现是否完成，测试结果如下：



```
spy@spy-desktop: ~/桌面/proj16_4
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)
check_mm_shm_swap() succeeded.
++ setup timer interrupts
kernel_execve: pid = 3, name = "semtest".
sem_id = 0x07fc8260
post ok.
wait ok.
sleep 0
wait semaphore...
sleep 1
sleep 2
sleep 3
sleep 4
sleep 5
sleep 6
sleep 7
sleep 8
sleep 9
hold semaphore.
fork pass.
semtest pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:520:
  initproc exit.

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

semtest 结果

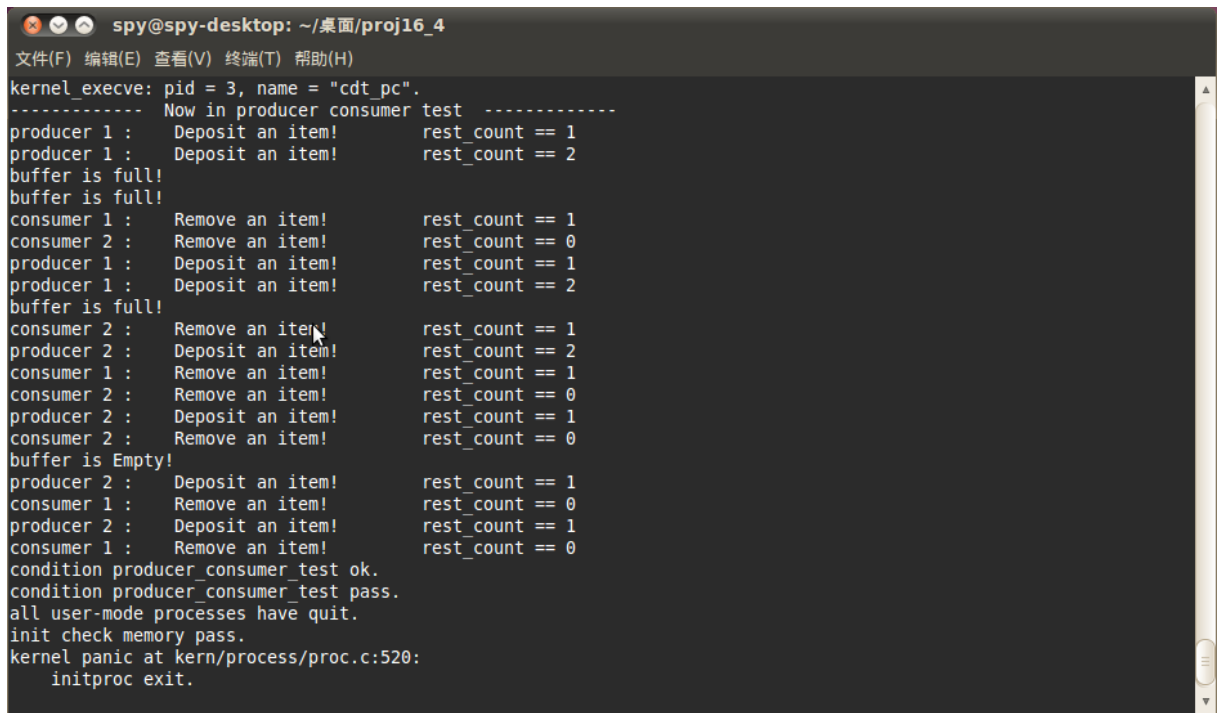


```
spy@spy-desktop: ~/桌面/proj16_4
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)
check_mm_shm_swap() succeeded.
++ setup timer interrupts
kernel_execve: pid = 3, name = "semtest3".
wait now...
wait timeout
child now sleep
semtest3 pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:520:
  initproc exit.

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

Semtest3 结果

在完成补全代码任务（2）后可以使用 cdt_pc 测试条件变量的基本实现是否完成，测试结果如下：



```
spy@spy-desktop: ~/桌面/proj16_4
文件(F) 编辑(E) 查看(V) 终端(T) 帮助(H)
kernel_execve: pid = 3, name = "cdt_pc".
----- Now in producer consumer test -----
producer 1 :   Deposit an item!      rest_count == 1
producer 1 :   Deposit an item!      rest_count == 2
buffer is full!
buffer is full!
consumer 1 :   Remove an item!       rest_count == 1
consumer 2 :   Remove an item!       rest_count == 0
producer 1 :   Deposit an item!      rest_count == 1
producer 1 :   Deposit an item!      rest_count == 2
buffer is full!
consumer 2 :   Remove an item!       rest_count == 1
producer 2 :   Deposit an item!      rest_count == 2
consumer 1 :   Remove an item!       rest_count == 1
consumer 2 :   Remove an item!       rest_count == 0
producer 2 :   Deposit an item!      rest_count == 1
consumer 2 :   Remove an item!       rest_count == 0
buffer is Empty!
producer 2 :   Deposit an item!      rest_count == 1
consumer 1 :   Remove an item!       rest_count == 0
producer 2 :   Deposit an item!      rest_count == 1
consumer 1 :   Remove an item!       rest_count == 0
condition producer_consumer_test ok.
condition producer_consumer_test pass.
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:520:
  initproc exit.
```

在完成补全代码任务（3）后可以使用 sem_wf 测试读者优先的读者写者问题实现是否正确，测试结果如下（结果中的顺序可能与实验指导书上的结果略有不同）：

```
kernel_execve: pid = 3, name = "sem_wf".
-----
reader 0: (pid:4) arrive
  reader_wf 0: (pid:4) start 100
reader 1: (pid:5) arrive
  reader_wf 1: (pid:5) start 110
reader 2: (pid:6) arrive
  reader_wf 2: (pid:6) start 110
reader 3: (pid:7) arrive
  reader_wf 3: (pid:7) start 100
reader 4: (pid:8) arrive
  reader_wf 4: (pid:8) start 100
reader 5: (pid:9) arrive
  reader_wf 5: (pid:9) start 120
reader 6: (pid:10) arrive
  reader_wf 6: (pid:10) start 120
reader 7: (pid:11) arrive
  reader_wf 7: (pid:11) start 100
reader 8: (pid:12) arrive
  reader_wf 8: (pid:12) start 120
reader 9: (pid:13) arrive
  reader_wf 9: (pid:13) start 110
reader_wf 0: (pid:4) end 100
reader_wf 3: (pid:7) end 100
reader_wf 4: (pid:8) end 100
reader_wf 7: (pid:11) end 100
```

```
    reader_wf 1: (pid:5) end 110
    reader_wf 2: (pid:6) end 110
    reader_wf 9: (pid:13) end 110
    reader_wf 5: (pid:9) end 120
    reader_wf 6: (pid:10) end 120
    reader_wf 8: (pid:12) end 120
read_test_wf ok.
```

```
-----
writer 0: (pid:14) arrive
    writer_wf 0: (pid:14) start 110
writer 1: (pid:15) arrive
writer 2: (pid:16) arrive
writer 3: (pid:17) arrive
writer 4: (pid:18) arrive
writer 5: (pid:19) arrive
writer 6: (pid:20) arrive
writer 7: (pid:21) arrive
writer 8: (pid:22) arrive
writer 9: (pid:23) arrive
    writer_wf 0: (pid:14) end 110
    writer_wf 1: (pid:15) start 120
    writer_wf 1: (pid:15) end 120
    writer_wf 2: (pid:16) start 110
    writer_wf 2: (pid:16) end 110
    writer_wf 3: (pid:17) start 120
    writer_wf 3: (pid:17) end 120
    writer_wf 4: (pid:18) start 110
    writer_wf 4: (pid:18) end 110
    writer_wf 5: (pid:19) start 100
    writer_wf 5: (pid:19) end 100
    writer_wf 6: (pid:20) start 100
    writer_wf 6: (pid:20) end 100
    writer_wf 7: (pid:21) start 110
    writer_wf 7: (pid:21) end 110
    writer_wf 8: (pid:22) start 110
    writer_wf 8: (pid:22) end 110
    writer_wf 9: (pid:23) start 100
    writer_wf 9: (pid:23) end 100
write_test_wf ok.
```

```
-----
reader 0: (pid:24) arrive
    reader_wf 0: (pid:24) start 120
writer 1: (pid:25) arrive
writer 2: (pid:26) arrive
reader 3: (pid:27) arrive
writer 4: (pid:28) arrive
writer 5: (pid:29) arrive
reader 6: (pid:30) arrive
writer 7: (pid:31) arrive
reader 8: (pid:32) arrive
reader 9: (pid:33) arrive
    reader_wf 0: (pid:24) end 120
    writer_wf 1: (pid:25) start 100
```

```
writer_wf 1: (pid:25) end 100
writer_wf 2: (pid:26) start 100
writer_wf 2: (pid:26) end 100
writer_wf 4: (pid:28) start 100
writer_wf 4: (pid:28) end 100
writer_wf 5: (pid:29) start 100
writer_wf 5: (pid:29) end 100
writer_wf 7: (pid:31) start 100
writer_wf 7: (pid:31) end 100
reader_wf 3: (pid:27) start 110
reader_wf 6: (pid:30) start 110
reader_wf 8: (pid:32) start 120
reader_wf 9: (pid:33) start 110
reader_wf 3: (pid:27) end 110
reader_wf 6: (pid:30) end 110
reader_wf 9: (pid:33) end 110
reader_wf 8: (pid:32) end 120
read_write_test_wf ok.
sem_wf pass..
all user-mode processes have quit.
init check memory pass.
```

在完成补全代码任务（4）后可以使用 cdt_wf 测试管程实现的写者优先的读者写者问题是否成功，测试结果如下（结果中的顺序可能与实验指导书上的结果略有不同）：

```
kernel_execve: pid = 3, name = "cdt_wf".
-----
reader 0: (pid:4) arrive
  reader_wf start 0: (pid:4) 100
reader 1: (pid:5) arrive
  reader_wf start 1: (pid:5) 110
reader 2: (pid:6) arrive
  reader_wf start 2: (pid:6) 110
reader 3: (pid:7) arrive
  reader_wf start 3: (pid:7) 100
reader 4: (pid:8) arrive
  reader_wf start 4: (pid:8) 100
reader 5: (pid:9) arrive
  reader_wf start 5: (pid:9) 120
reader 6: (pid:10) arrive
  reader_wf start 6: (pid:10) 120
reader 7: (pid:11) arrive
  reader_wf start 7: (pid:11) 100
reader 8: (pid:12) arrive
  reader_wf start 8: (pid:12) 120
reader 9: (pid:13) arrive
  reader_wf start 9: (pid:13) 110
  reader_wf end 0: (pid:4) 100
  reader_wf end 3: (pid:7) 100
  reader_wf end 4: (pid:8) 100
  reader_wf end 7: (pid:11) 100
```

```
    reader_wf end 1: (pid:5) 110
    reader_wf end 2: (pid:6) 110
    reader_wf end 9: (pid:13) 110
    reader_wf end 5: (pid:9) 120
    reader_wf end 6: (pid:10) 120
    reader_wf end 8: (pid:12) 120
read_test_wf ok.
```

```
-----
writer 0: (pid:14) arrive
    writer_wf start 0: (pid:14) 110
writer 1: (pid:15) arrive
writer 2: (pid:16) arrive
writer 3: (pid:17) arrive
writer 4: (pid:18) arrive
writer 5: (pid:19) arrive
writer 6: (pid:20) arrive
writer 7: (pid:21) arrive
writer 8: (pid:22) arrive
writer 9: (pid:23) arrive
    writer_wf end 0: (pid:14) 110
    writer_wf start 1: (pid:15) 120
    writer_wf end 1: (pid:15) 120
    writer_wf start 2: (pid:16) 110
    writer_wf end 2: (pid:16) 110
    writer_wf start 3: (pid:17) 120
    writer_wf end 3: (pid:17) 120
    writer_wf start 4: (pid:18) 110
    writer_wf end 4: (pid:18) 110
    writer_wf start 5: (pid:19) 100
    writer_wf end 5: (pid:19) 100
    writer_wf start 6: (pid:20) 100
    writer_wf end 6: (pid:20) 100
    writer_wf start 7: (pid:21) 110
    writer_wf end 7: (pid:21) 110
    writer_wf start 8: (pid:22) 110
    writer_wf end 8: (pid:22) 110
    writer_wf start 9: (pid:23) 100
    writer_wf end 9: (pid:23) 100
write_test_wf ok.
```

```
-----
reader 0: (pid:24) arrive
    reader_wf start 0: (pid:24) 120
writer 1: (pid:25) arrive
writer 2: (pid:26) arrive
reader 3: (pid:27) arrive
writer 4: (pid:28) arrive
writer 5: (pid:29) arrive
reader 6: (pid:30) arrive
writer 7: (pid:31) arrive
reader 8: (pid:32) arrive
reader 9: (pid:33) arrive
    reader_wf end 0: (pid:24) 120
    writer_wf start 1: (pid:25) 100
```

```
writer_wf end 1: (pid:25) 100
writer_wf start 2: (pid:26) 100
writer_wf end 2: (pid:26) 100
writer_wf start 4: (pid:28) 100
writer_wf end 4: (pid:28) 100
writer_wf start 5: (pid:29) 100
writer_wf end 5: (pid:29) 100
writer_wf start 7: (pid:31) 100
writer_wf end 7: (pid:31) 100
reader_wf start 3: (pid:27) 110
reader_wf start 6: (pid:30) 110
reader_wf start 8: (pid:32) 120
reader_wf start 9: (pid:33) 110
reader_wf end 3: (pid:27) 110
reader_wf end 6: (pid:30) 110
reader_wf end 9: (pid:33) 110
reader_wf end 8: (pid:32) 120
read_write_test_wf ok.
condition reader_writer_wf_test pass..
all user-mode processes have quit.
init check memory pass.
kernel panic at kern/process/proc.c:520:
initproc exit.
```

5. 实验报告要求

在代码目录 lab5 下完成实验。回答实验指导书中的问题，在实践中完成实验中的练习。在报告中简要描述如何完成练习，练习中碰到的问题/困难，已经如何解决的等。实验报告文档命名为 lab5-学生 ID.txt，在 proj16 目录下执行：

```
make handin
```

会得到 proj16-handin.tar.gz 的文件。将此文件和实验报告放在 lab5-handin 目录下，并在 Lab5 根目录下执行：

```
tar jcf lab5-StudentID.tar.bz2 lab5-handin
```

即可生成 lab5-StudentID.tar.bz2 压缩文件，请注意把 “Student ID” 替换成你的学号。比如你的学号是 2008011999，则对应的命令是：

```
tar jcf lab5-2008011999.tar.bz2 lab5-handin
```

如果你做了 Challenge，除了要在报告上写明外，还要在网络学堂提交窗口的作业内容

一栏注明。