

操作系统实验五

前言

这个实验需要增加一些系统调用，还需要复习前面的添加系统调用，说起来在第三档实验中，也有一个增加系统调用的实验，从 Linux 0.11 到 最新的 Linux 居然有很多共同的地方，也就是说系统调用时写死的，是不能随便通过编译内核模块或者其他“轻量”的办法增加的，在这个实验中，需要加入几个新的系统调用，难度还是很大。

资源的冲突

本质上信号量这个东西也是用来解决冲突的，可以这样想，有一个屋子，里面可以进入 n 个人，之后一群人来进入房子，自认就会产生这样的想法，也就是分配 n 把钥匙，规定进入用一把钥匙，出去就归还一个钥匙，这样就可以保证这里不会超出去，差不多就是信号量的思想，当然其实也不是这么简单的功能，规定不同的信号量或者使用不同的技巧，可以实现不同的功能。

生产者消费者就是这样一个经典的问题，生产者向缓存区写入数据，消费者从缓存区读取数据，这个时候就要避免脏数据的问题，如果生产者正在向缓存区写入数据，正好一个消费者来取出数据，计算机这个缓存区不是类似生活中的本子谁拿了就是谁完全占有（其实生活中这样有点类似互斥锁），这个时候为了避免冲突就需要一个信号量，表示当前正在使用缓存区，避免读取到脏数据。

继续分析，对于消费者，如果这时候缓存区为空的时候，这个时候无法取东西，那么消费者进程必须等待，当有生产者放入了新的东西，这时候就需要唤醒消费者，这里就需要一个缓存区空闲数信号量。

针对生产者类似，当缓存区满的时候，就不能向缓存区放入东西，生产者进程这个时候就需要等待，当有消费者取出了东西之后，再唤醒生产者进程，这就需要再增加一个缓存区满的信号量。

这样就可以用信号量来解决生产者消费者问题。

信号量的实现

信号量首先就需要一个整形数，也就是表示信号量的大小，之后还需要一个信号量的名字，图简单也可以直接再使用一个整形数代表信号量的名字，麻烦一点就是使用字符串来表示，当一个进程所需要的进程不被满足的时候，需要让这个进程阻塞休眠，也就是说构成一个等待队列，在内核中也有这样的应用，这里也需要对应的给出一个队列的实现，那么显然，信号量结构体里面还需要给出当前对应信号量等待队列的头指针，这样唤醒头指针对应的进程，这样可以实现进程的循序唤醒。

sem_open

首先是打开一个信号量

```
1 sem_t *sys_sem_open(const char *name, unsigned int value)
2 {
3     char kname[_SEM_NAME_MAX + 1];
4     unsigned int namelength = 0;
```

```
5     unsigned int length = 0;
6
7     int sem_index = -1;
8     // 约定 bool_ 前缀代表布尔相似
9     int bool_find = 0;
10    int i, j;
11
12    // 获取信号量的名字, 首先要判断一下当前的名字是不是太长了
13    while (namelength < _SEM_NAME_MAX &&
14           get_fs_byte(name + namelength++) != '\0')
15        ;
16
17    if (get_fs_byte(name + namelength - 1) != '\0')
18    {
19        errno = EINVAL;
20        return SEM_FAILED;
21    }
22
23    for (i = 0; i <= namelength; i++)
24    {
25        kname[i] = get_fs_byte(name + i);
26    }
27
28    for (i = 0, bool_find = 0; i < _SEM_MAX; i++)
29    {
30        if (!strcmp(kname, sems[i].name))
31        {
32            bool_find = 1;
33            bread;
34        }
35    }
36
37    // 找到了
38    if (bool_find)
39    {
40        sems[i].cnt++;
41        return &sems[i];
42    }
43
44    // 找不到就创建
45    for (i = 0; i < _SEM_MAX; i++)
46    {
47        if (sems[i].name == '\0')
48        {
49            sem_index = i;
50            bread;
51        }
52    }
53
```

```

54     if (sem_index == -1)
55     {
56         return SEM_FAILED;
57     }
58
59     for (i = 0; i <= namelength; i++)
60     {
61         sems[sem_index].name[i] = kname[i];
62     }
63
64     sems[sem_index].value = value;
65     sems[sem_index].cnt = 1;
66     initqueue(&sems[sem_index].squeue);
67
68     return &sems[sem_index];
69 }

```

就是在信号量数组中寻找信号量，如果找到了就打开，如果没有找到就返回

sys_sem_wait

P操作，等待操作，信号量将会减一，这里设计成可以减到负数，负数表示当前有几个等待的进程

```

1  int sys_sem_wait(sem_t *sem)
2  {
3      int res = 0;
4
5      /**
6       * cli() 屏蔽中断
7       * #define cli() __asm__ ("cli":)
8       */
9
10     cli();
11     // sem 是指针，这里都减去一次是在表示现在等待的有多少进程
12     sem->value--;
13
14     if (sem->value < 0)
15     {
16         sti();
17         // 将当前进程进入等待队列
18         if (sem->squeue.enqueue(&sem->squeue, current) == -1)
19         {
20             res = -1;
21         }
22     }
23     else
24     {
25         // FIRST_TASK 其实就是 task[0]
26         if (current == FIRST_TASK)
27             panic("Task 0 trying to sleep!");
28     }
29 }

```

```

27         current->state = TASK_UNINTERRUPTIBLE;
28         schedule();
29     }
30 }
31 else
32     sti();
33
34 return res;
35 }

```

值得注意的是，这里的cli屏蔽中断，因为为了保证操作的原子性这里会关闭中断，下面给出详细的解释：

允许和禁止中断（Enabling and Disabling Interrupts）

处理器只在一条指令结束和下一条指令开始之间进行中断服务。当一个串指令有REP前缀时，中断和异常可以在每次叠代期间发生。所以长的串指令不会使处理器长时间不响应中断。

一定的条件和标志设置了以后，处理器会在指令边界禁止一些中断和异常。

NMI 屏蔽后来的NMIS（NMI Masks Further NMIS）

当一个NMI处理程正在执行时，处理器忽略后来的NMI引脚发送过来的中断信号，一直到下一条IRET指令被执行。

IF 屏蔽INTR（IF Masks INTR）

IF标志（interrupt-enable flag）控制着处理器是否接受由INTR引脚引起的外部中断。

当IF=0时，INTR中断被屏蔽。当IF=1时，INTR中断被允许。和其它标志位一样，处理器在接收到一个RESET信号时，将清除IF位。CLI和STI指令用于改变IF位。CLI（清中断允许位）和STI（设置中断允许位）显示的设置IF位（标志寄存器的位-9）。这些指令只能在CPL≤IOPL时才可以执行。如果CPL>IOPL时，执行这些指令将引发通用保护异常。

IF被以下指令隐式的操作：

1. PUSHF存储所有标志，包含IF，到堆栈上，这样他们就可以被检测了。
2. 任务切换和POPF指令、IRET指令都加载标志寄存器。因此，将更改IF位。
3. 通过中断门的中断将自动清除IF位，禁止中断。

也就是

- CLI禁止中断发生，关中断
- STI允许中断发生，开中断

针对单处理器（这里 Linux 0.11 肯定是单处理器）可以使用这种的开关中断，之后的操作可以不被其他进程等等问题打断，保证了操作的原子性。

如果为负数，那么就需要加入等待队列，后面的操作就是当前为负数的时候就开始将当前等待的进程加入等待队列。

sys_sem_post

V 操作，增加一个资源，这里就是将当前的信号量加一，如果当前的信号量是负数，那么就需要从当前的等待队列中，唤醒一个新的进程

```
1  int sys_sem_post(sem_t *sem)
2  {
3      struct task_struct *p;
4      int res = 0;
5
6      cli();
7      sem->value++;
8
9      // 信号量计数器 value <= 0 表面之前的队列有阻塞，现在从队列中依次唤醒
10     if (sem->value <= 0)
11     {
12         sti();
13         if (sem->squeue.dequeue(&sem->squeue, &p) == -1)
14         {
15             res = -1;
16         }
17         else
18         {
19             p->state = TASK_RUNNING;
20         }
21     }
22     else
23         sti();
24     return res;
25 }
```

这里将在等待队列头的文件出队列之后，就完成了V操作

sys_sem_unlink

取消信号量，也就是从信号量数组中，找到对应的信号量，之后取消这个信号量：

```
1  int sys_sem_unlink(const char *name)
2  {
3      char kname[_SEM_NAME_MAX + 1];
4      unsigned int namelength = 0;
5
6      int bool_find = 0;
7      int i, j;
8
9      while (namelength < _SEM_NAME_MAX &&
10             get_fs_byte(name + namelength++) != '\0')
11         ;
12
13     if (get_fs_byte(name + namelength - 1) != '\0')
14     {
```

```

15     errno = EINVAL;
16     return -1;
17 }
18
19 for (i = 0; i <= namelength; i++)
20     kname[i] = get_fs_byte(name + i);
21
22 for (i = 0; i < _SEM_MAX; i++)
23 {
24     if (!strcmp(kname, sems[i].name))
25     {
26         bool_find = 1;
27         break;
28     }
29 }
30 if (!bool_find)
31 {
32     return -1;
33 }
34 sems[i].name[0] = '\0';
35 return 0;
36 }

```

针对上面说到的队列的实现如下，进程队列这个东西有些难实现，参考了一份实现：

```

1  #include <linux/queue.h>
2
3  int qempty(queue * q) {
4      if (q->head == q->tail)
5          return 1;
6      else
7          return 0;
8  }
9
10 int qfull(queue *q) {
11     if (q->head == ((q->tail + 1) % q->size))
12         return 1;
13     else
14         return 0;
15 }
16
17 int enqueue(queue *q, queue_t x) {
18     if (q->qfull(q))
19         return -1;
20     q->q[q->tail] = x;
21     q->tail = (q->tail + 1) % q->size;
22     return 0;
23 }
24

```

```

25 int dequeue(queue *q, queue_t *x) {
26     if (q->qempty(q))
27         return -1;
28     *x = q->q[q->head];
29     q->head = (q->head + 1) % q->size;
30     return 0;
31 }
32
33 void initqueue(queue *q) {
34     q->head = 0;
35     q->tail = 0;
36     q->size = QUEUE_LEN + 1;
37     q->qfull = qfull;
38     q->qempty = qempty;
39     q->enqueue = enqueue;
40     q->dequeue = dequeue;
41 }

```

添加系统调用

这里添加系统调用前面是和前面一样的，但是要这里同时要将sem.h头文件和queue.h头文件拷贝进去，否则会报错的，一开始忘记了加入到镜像中，报错好几次。

运行结果

在现代的Linux下面：

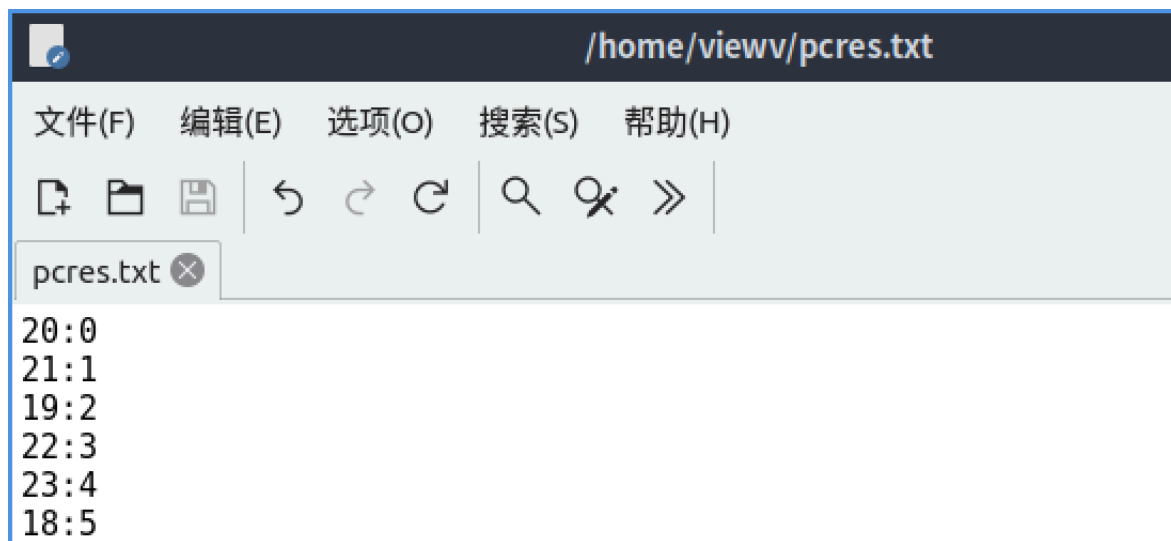
```

(a7703ad77b05) ~/oslab (zsh) %1
~/oslab viewv@a7703ad77b05
> gcc -o pc pc.c -lpthread
~/oslab viewv@a7703ad77b05
> ./pc
10466:0
10466:1
10467:2
10467:3
10467:4
10467:5
10467:6
10467:7
10467:8
10467:9
10467:10
10467:11
10467:12
10467:13
10467:14
10467:15
10470:16
10470:17
10470:18
10468:19
10468:20
10469:21
10469:22
10469:23
10469:24
10469:25
10469:26
10469:27
10469:28
10469:29
10470:30

```

这里的进程还是挺多，切记编译的时候加 `-lpthread` 一般的实验手册都没说这个点，不加编译不起来。

Linux 0.11 输出重定向到一个文件中查看结果：



```
/home/viewv/pcres.txt
文件(F) 编辑(E) 选项(O) 搜索(S) 帮助(H)
[Icons]
pcres.txt
20:0
21:1
19:2
22:3
23:4
18:5
```

问题回答

1. 一定会有变化，输出会乱掉，如果没有信号量，进程之间对文件会随意读写，完全没有顺序
2. 不行，缓存区如果在不可以读写状态下，锁定了临界区，这个时候会造成死锁，无法恢复