

# **16 级计科 7 班: 操作系统原理实验 #8(保护模式)**

Due on Wednesday, June 21, 2018

凌应标 周一 9-10 节

**颜彬**

**16337269**

# Content

	Page
<b>1 实验目的</b>	<b>3</b>
<b>2 实验过程</b>	<b>3</b>
2.1 实现框架 . . . . .	3
2.2 P 操作的实现 . . . . .	4
2.3 V 操作 . . . . .	5
<b>3 实验结果</b>	<b>6</b>
3.1 测试样例 . . . . .	6
3.2 实验现象 . . . . .	6
<b>4 实验总结</b>	<b>7</b>
4.1 实验心得 . . . . .	7
4.2 BUG 汇总 . . . . .	8
4.2.1 没有初始化 . . . . .	8
4.2.2 数组长度忘记声明 . . . . .	8
<b>A 参考文献</b>	<b>8</b>

## 1 实验目的

在这个项目中, 我们完善进程模型, 使得多个进程能够利用计数信号量机制实现临界区互斥。合作进程在并发时, 利用计数信号量, 可以按规定的时序执行各自的操作, 实现复杂的同步, 确保进程并发的情况正确完成使命。

实现信号量机制, 即一个整数和一个指针组成的结构体, 在内核可以定义若干个信号量, 统一编号。

内核实现 do\_p() 原语, 在 c 语言中用 p(int sem\_id) 调用

内核实现 do\_v() 原语, 在 c 语言中用 v(int sem\_id) 调用

内核实现 do\_getsem() 原语, 在 c 语言中用 getsem(int ) 调用, 参数为信号量的初值

内核实现 do\_freeseem(int sem\_id), 在 c 语言中用 freeseem(int sem\_id) 调用

## 2 实验过程

### 2.1 实现框架

采用系统调用的框架完成本项目。本小节中, 用 getsem 为例子讨论如何应用系统调用的框架实现 PV 操作。

如代码1所示, 用户函数库中的 getsem, 仅仅只是产生一个中断。中断号是 06, 是 do\_getsem 对应的功能号。该系统调用接收一个参数, 作为信号量的初始值。该值被放在寄存器 ebx 中, 最终会成为 do\_getsem 的一个参数。

这里采用内嵌汇编的方式实现。“=r”会把汇编中 eax 寄存器的值传递给 ret。“b”(id) 会把 id 的值传递给 ebx 寄存器。当控制权最终转移给 do\_getsem 时, 其接收的参数的值恰好就是 ebx 中的值。中断

代码 1: 用户函数库中的 getsem

```
1 int freeseem(int id) {  
2     int ret;  
3     __asm__(  
4         "movl $0x06, %%eax\n"  
5         "int $0x80\n"  
6         : "=r"(ret)  
7         : "b"(id)  
8     );  
9     return ret;  
10 }
```

处理程序会把 ebx 等寄存器的值压入栈中, 供 do\_getsem 等函数作为参数。

go\_getsem 的返回值会存储在 eax 中, 最终在代码1中的"=r"(ret), 传递到变量 ret 中, 然后通过 return ret 语句返回。在 int \$0x80 这句代码中, 控制权会转移到系统调用处理程序中。如3所示。在进入中断服务程序之前首先会同步 ebx, ecx 和 edx(因为他们在前面的代码中发生了改变)。随后依次将 edx, ecx, ebx 压栈, 然后调用 [sys\_call\_table + 4 \* eax] 地址处的函数。eax 是索引, 4\*eax 恰好是每个服务程序的偏移量。

代码 2: do\_getsem 的实现

```
int do_getsem(int value) {  
2   for (int i = 0; i < NR_SEMAPHORE; ++i) {  
       if(semaphone_list[i].used == 0) {  
4           semaphone_list[i].used = 1;  
           semaphone_list[i].value = value;  
6           semaphone_list[i].bsize = 0;  
           return i;  
8       }  
       }  
10    return -1;  
}
```

若服务程序不接收参数, 这些压栈的寄存器不会被用到。若服务程序使用了寄存器, 服务程序会按照 ebx, ecx, edx 的顺序接收第一、第二、第三个参数 (如果有的话)

代码 3: 系统调用处理程序的部分实现

```
mov ebx, [esp + 10*4]  
2 mov ecx, [esp + 12*4]  
mov edx, [esp + 11*4]  
4  
push edx  
6 push ecx  
push ebx  
8  
call [sys_call_table + 4 * eax]  
10 add esp, 12
```

## 2.2 P 操作的实现

这里只叙述 do\_p 的实现。其实现如代码4所示。

首先进入前先关中断。在设置信号量和进程控制块的过程中被 schedule 会带来严重的问题。在把内核状态切换完全前, 不能被时钟中断。

这里没有采用链表, 而是采用数组的方式存储被阻塞的进程号。这在简化版的操作系统中, 可以让实现更加简单。

若信号量不小于零, 则开中断并返回。否则, 调用 wait 函数。wait 在上个实验中已经实现。它会阻塞当前进程, 并调度下一个就绪进程。

代码 4: P 操作的实现

```
int do_p(int id) {  
2   cli();  
   semaphore_list[id].value--;  
4   if (semaphore_list[id].value < 0) {  
       int size = semaphore_list[id].bsize;  
6       semaphore_list[id].block_processes[size] = current;  
       semaphore_list[id].bsize++;  
8       wait();  
   }  
10  sti();  
   return 0;  
12 }
```

## 2.3 V 操作

V 操作的实现如代码5所示。

同样首先关中断，然后把信号量的值加 1。如果其小于等于 0，则将队列出队，将出队的进程号标志为就绪。若信号量的值大于 0，则开中断并返回。

值得注意的是，此处采用数组实现队列。此处与用数组实现栈有所区别。如果以栈的方式工作，很可能会导致饥饿。只有以队列的方式实现，才能确保各个进程公平地被唤醒。故此处用了 for 循环将数组的元素全部向前平移一个位置，以实现队列的行为。

代码 5: V 操作的实现

```
int do_v(int id) {  
2   cli();  
   semaphore_list[id].value++;  
4   int val = semaphore_list[id].value;  
   if (semaphore_list[id].value <= 0) {  
6       int size = semaphore_list[id].bsize;  
       int block_process_id = semaphore_list[id].block_processes[0];  
8       PCB_List[block_process_id].state = TASK_INTERRUPTIBLE;  
       for (int i = 0; i < size-1; ++i) {  
10          semaphore_list[id].block_processes[i]  
              = semaphore_list[id].block_processes[i+1];  
12      }  
       semaphore_list[id].bsize--;  
14  }  
   sti();  
16  return 0;  
   }
```

## 3 实验结果

### 3.1 测试样例

此处采用了生产者消费者问题来进行测试。首先 fork 产生测试进程（测试不能运行在 0 号主进程，因为该进程的行为有所不同）。再在测试进程中 fork 处子进程 1 和 2。子进程 1 和 2 分别充当生产者和消费者，在有限长度缓冲区中做 push 和 pop 操作。

测试代码如代码6所示。代码中省略了输出到终端的代码。

生产者和消费者分别对 full 和 empty 这两个信号量作不同的 p 和 v 操作，以确保缓冲区不会发生上溢和下溢。

代码 6: 生产者消费者问题测试代码

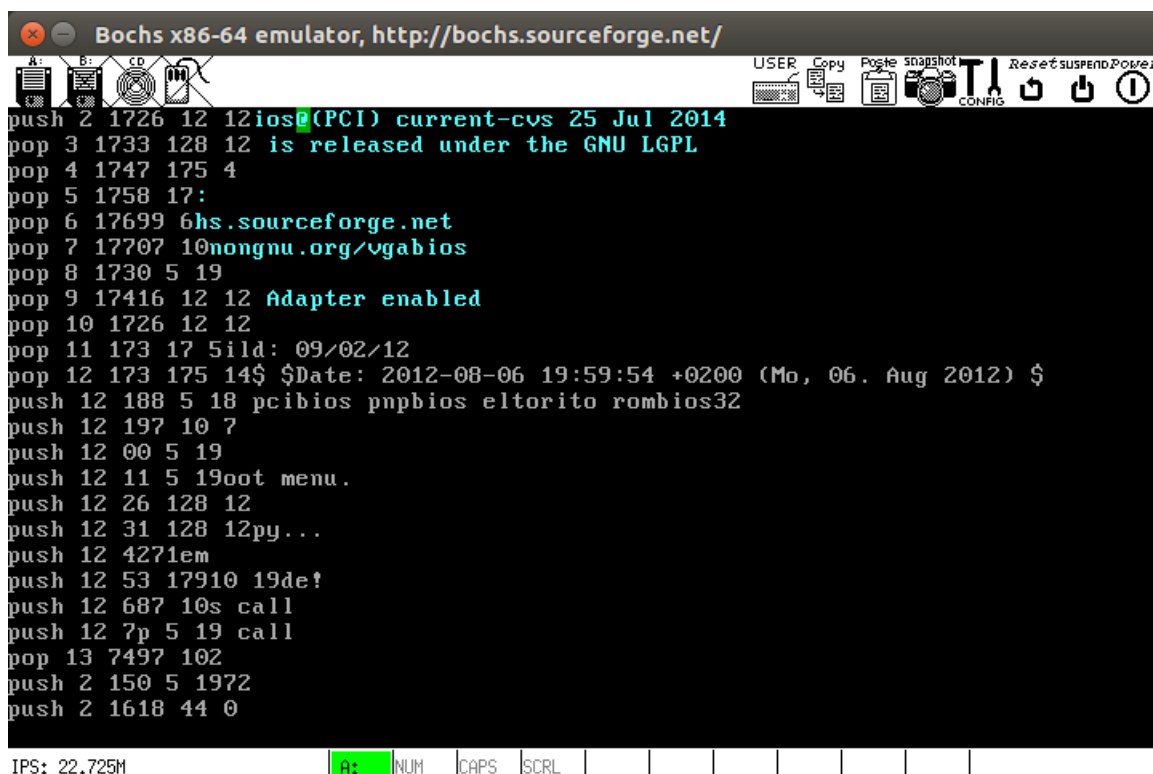
```
#define D 1000
2 void testPV() {
    full_lock = getsem(15);
4    empty_lock = getsem(0);
    puti(full_lock);
6    puti(empty_lock);
    beg = end = 0;
8    int id = fork();
    if (id == 1) {
10        while(1) {
            for (int i = 0; i < D; ++i) {}
12            p(empty_lock);
            push();
14            v(full_lock);
        }
16    } else {
        while(1) {
18            for (int i = 0; i < D; ++i) {}
            p(full_lock);
20            v(empty_lock);
            pop();
22        }
    }
24 }
```

### 3.2 实验现象

实验现象如图1所示。

由于进程切换是随机的，P 和 V 对进程挂起的触发也是随机的，故截图中打印出的 push 和 pop 的顺序和换行都是随机的。由于本项目没有作清屏操作，所以上一次打印遗留的数据还可能留在屏幕上，但从截图上看，P 和 V 正确工作，保证了队列不发生上溢和下溢。

可以修改宏定义 D 来改变 push 和 pop 的延时, 从而使得程序产生不同的效果。当 D 比较大时, 恰好 push 和 pop 相互间隔。这是因为延时大于时间片, 使得一个 (或多个) 时间片才能发生一次 push 和 pop。当 D 比较小时, 生产者会不断生产直到队列满后被挂起, 消费者会不断消费直到队列空后被挂起, 生产者和消费者交替工作。当 D 是某个合适的值时, push 和 pop 的行为就比较随机了。



```
Bochs x86-64 emulator, http://bochs.sourceforge.net/
push 2 1726 12 12ios(PC1) current-cvs 25 Jul 2014
pop 3 1733 128 12 is released under the GNU LGPL
pop 4 1747 175 4
pop 5 1758 17:
pop 6 17699 6hs.sourceforge.net
pop 7 17707 10nongnu.org/vgabios
pop 8 1730 5 19
pop 9 17416 12 12 Adapter enabled
pop 10 1726 12 12
pop 11 173 17 5ild: 09/02/12
pop 12 173 175 14$ $Date: 2012-08-06 19:59:54 +0200 (Mo, 06. Aug 2012) $
push 12 188 5 18 pcibios pnpbios eltorito rombios32
push 12 197 10 7
push 12 00 5 19
push 12 11 5 19oot menu.
push 12 26 128 12
push 12 31 128 12py...
push 12 4271em
push 12 53 17910 19de!
push 12 687 10s call
push 12 7p 5 19 call
pop 13 7497 102
push 2 150 5 1972
push 2 1618 44 0
IPS: 22,725M  A: NUM CAPS SCRL
```

图 1: 生产者消费者问题实验现象

## 4 实验总结

### 4.1 实验心得

本实验难度不大。在实验六和实验七都正常工作的前提下, 完成本实验花费的时间较少。

本实验在完成时依旧遇到了很严重的 bug, 见第4.2节的讨论。

由于一些历史遗留的原因, 以往的实验中的系统调用都没有涉及参数的传递。但本实验中, 内核的申请、释放信号量和 P、V 操作, 都涉及到参数的传递。故本实验又再次修改了中断发生时的逻辑, 在兼容以往实验的基础上, 允许中断服务程序接收参数。本实验所有的操作都最终在系统调用中实现。用户的库函数仅仅只是一个封装, 封装了中断的调用。如此即可将用户程序和内核代码解耦。确保用户程序无法直接修改和触碰到内核的变量。在以后开启特权级后, 即可做到真正的保护功能。

## 4.2 BUG 汇总

### 4.2.1 没有初始化

gcc 有时会初始化全局变量, 有时不会。在本实验中, 我仅仅修改了部分“无关紧要”的代码后, 发现全局变量没有被初始化了。这导致了全局变量被零初始化, 最终导致不断抛出常规保护错误的异常。

在一步一步跟踪到发生常规保护错误的代码后, 我就发现了变量的值很意外得为零。于是我猜想初始话没有发生。我试着写一个 init 函数, 在函数中显式得初始化全局变量, 并在内核初始化时调用 init 函数。如此即解决了这个问题。

### 4.2.2 数组长度忘记声明

由于忘记声明了数组长度, 我把数组的初始话写成了 `type arr[] = {}` 的形式。这导致了 gcc 实际上不在内存中为数组分配空间 (分配的空间大小为 0, 数组长度为 0)。

由于一些巧合的原因, 虽然数组没有被分配到空间, 但其前 8 个元素的值都“恰好”没有被改写。只有第 9 个元素和以后的元素被覆盖了。于是导致了前 8 个系统调用都正常工作, 但其后的系统调用都会产生常规保护异常。

调试时的截图如图2所示。在右侧把内存中的值打印出来时, 前面的值都是正确的。都与左侧的符号表中的值相符合。但是数组后面的值出错了。

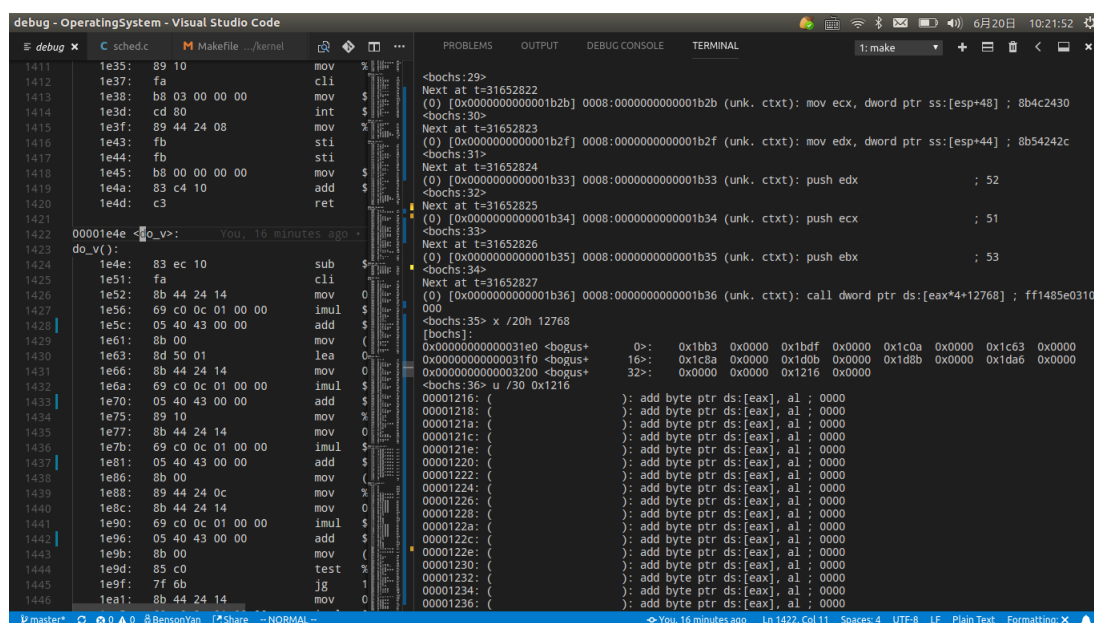


图 2: 调试时数组的值出错的截图

## 附录 A 参考文献

1. <https://blog.csdn.net/longintchar/article/details/50602851>  
16 位和 32 位汇编指令的不同 (尤其是 push 指令)



2. <https://www.ibiblio.org/gferg/ldp/GCC-Inline-Assembly-HOWTO.html#s1>  
GCC 内嵌汇编的书写。