

操作系统-实验八

学号：18340020 姓名：陈贤彪 学院：数据科学与计算机学院

1.实验目的

- 1、理解基于信号量的进程/线程同步方法
- 2、掌握内核实现信号量的方法
- 3、掌握信号量的应用方法
- 4、实现C库封装信号量相关系统调用。

2.实验要求

- 1、学习信号量机制原理
- 2、扩展内核，设计实现一种计数信号量。提供信号量申请、初始化和p操作与v操作等功能的系统调用。
- 3、修改扩展C库，封装信号量相关的系统调用操作。
- 4、设计一个多线程同步应用的用户程序，展示你的多线程模型的应用效果。
- 5、编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性

3.实验内容

- 1、在没有互斥机制时，实际运行父子存款取款问题会产生竞态，设法截屏验证此事
- 2、修改内核代码，增加信号量结构类型并定义一组信号量，组成信号量数组等数据结构。提供信号量申请初始化、释放和p操作与v操作等功能的系统调用。
- 3、修改扩展C库，封装信号量相关的系统调用操作。
- 4、设计一个多线程同步应用的用户程序，展示你的多线程模型的应用效果。

4.实验方案

1) 实验环境

a)系统：Linux Ubuntu18.04

2) 实验工具

a) VM VirtualBox

虚拟机软件，用于模拟虚拟不同的操作系统，也可以创建多个虚拟软盘

b) NASM-2.13.02

汇编语言编译器，可以将写好的.asm文件编译成二进制文件.bin

c) gcc (Ubuntu 5.5.0-12ubuntu1) 5.5.0 20171010

c语言编译器

d)Visual Studio Code

代码编辑器,用于编辑 asm 代码

e) GNU bash version 4.4.20(1)-release (x86_64-pc-linux-gnu)

系统跟计算机硬件交互时使用的中间介质,用于简便对文件进行转换

f) GNU ld (GNU Binutils for Ubuntu) 2.30

链接器, 将汇编与c生成的.o文件链接在一起

f) github

开源代码托管平台,用于存储管理编写的代码

g) bochs 2.6.11

软盘调试工具

3) 实验原理

(1) 总体磁盘架构

柱面号	磁头号	扇区偏移	占用扇区数	功能
0	0	1	1	引导扇区程序
0	0~1	2	35	操作系统内核
1	0	1	2	userpro1
1	0	3	2	userpro2
1	0	5	2	userpro3
1	0	7	2	userpro4
1	0	9	2	用户程序表 list
1	0	11	18	C语言用户程序
1	1	11	2	系统调用测试程序
1	1	13	2	fork测试程序
1	1	15	2	pv 测试程序
1	1	17	2	pv 测试对比程序

相比于之前的代码量, 内核代码越来越多, 因此我直接将整个柱面的扇区分配给了内核

userpro1 对应左上角滚动字符, userpro2 对应又上角滚动字符, userpro3 对应左下角滚动字符, userpro4 对应右下角滚动字符.

c语言用户程序, 由于编写了库, 因此占用比较多的扇区数目

系统调用的测试程序放在最后面

在内核中, 我创建了一个结构体存储用户程序的信息

本次实验八新增了 `pv` 两个对比的测试程序

(2) MY操作系统内核的设计

内核程序结构：

新增：在进程调度的函数模块 `pcb.asm`，`pcb.h` 中加入了关于 `fork`，`wait`，`block` 的函数

程序名	代码形式	作用
<code>kernel.asm</code>	ASM	内核的入口，调用c中的函数，新增运行风火轮代码
<code>kernel_a.asm</code>	ASM	包含一些显示打印，IO借口，扇区加载的函数,运行 <code>ouch</code> 代码
<code>stdio.h</code>	C	包含一些对字符串处理的函数
<code>kernel_c.asm</code>	C	内核c代码，内核命令的主要部分（新增关于新指令的代码）
<code>ouch.asm</code>	ASM	封装了关于 <code>ouch</code> 键盘中断的代码
<code>system_a.asm</code>	ASM	包含关于系统调用（汇编）的函数
<code>system_c.c</code>	C	包含关于系统调用（c语言）的函数
<code>pcb.asm</code>	ASM	封装了关于 PCB 保护和恢复的函数（新增 <code>do_p,do_v</code> 等信号量函数）
<code>pcb.h</code>	C	封装了PCB的创建，初始化，进程调度，进程撤销的函数fork测试程序

以下是我已经实现的指令功能（新增了 `runall` 8,9指令: `pv` 两个测试程序）

指令名	功能
<code>clear</code>	清楚屏幕
<code>ls</code>	调用 <code>list</code> 程序显示用户程序的信息，需要 按 <code>esc</code> 退出
<code>help</code>	显示帮助的信息
<code>run</code>	可以按照自定义顺序运行1234程序，（新增） <code>run 5</code> 为系统调用的测试程序， <code>run 6</code> 为c程序测试程序
<code>time</code>	显示当前系统时间
<code>shutdown</code>	关机
<code>runall</code>	进入多进程状态，同时运行多个用户程序，如 <code>runall 1234</code> 即是同时运行四个用户程序

(3) 信号量初始化函数

首先我在 `pcb.h` 文件中定义有关信号量的结构体，如下：

```

1  #define NRsem 100
2  typedef struct semaphore {
3      int count;
4      int blocked[20];
5      int used;
6      int que;
7  }semaphore;
8  semaphore semlist[NRsem];

```

其中 `count` 变量表示该信号量的数量个数，`block` 数组则是记录被阻塞进程的队列，`used` 变量表示该信号量是否被使用，`que` 变量表示该信号队列的长度（用于队列操作）

我定义了系统中可生成最多的信号量为100。

随后我就开始编写信号量的生成函数，该函数定义在 `pcb.h`

```

1  int do_GetSema(int value) {
2      int i=0;
3      while(semlist[i++].used);
4      if (i< NRsem) {
5          semlist[i].used=1;
6          semlist[i].count=value;
7          semlist[i].que=0;
8          return(i);
9      }
10     else
11         return(-1);
12 }

```

函数的输入是一个 `value` 整型用于赋值该信号量的数量，输出是返回信号量的标号（用于找到该信号量）。

函数的逻辑如下：首先在信号量数组中从头开始检索，找到其中一个没有正在被使用的信号量（`used==0`），然后就使用该标号的信号量，然后将该信号量的`used`变成1，阻塞队列长度为0，数量为`value`，返回该信号量的标号。若全部信号量都被使用了，则返回-1。

由于信号量的申请是在用户程序中进行的，但是申请的函数是写在内核当中，因此我要将该函数变成系统调用，让用户程序来调用。系统调用函数封装在 `pcb.asm` 中

```

1  extern do_GetSema
2  global sys_getsema
3  sys_getsema:
4      PUSHALLPCB
5      call pcbsave
6      add sp,16*2
7      mov ax,0
8      push ax
9      push bx
10     call dword do_GetSema
11     pop bx
12     pop ax
13     mov cx,ax
14
15     push cx
16     call dword getcurrentpcb
17     pop cx

```

```

18 |     mov si, ax
19 |     mov [cs:si+0],cx
20 |
21 |     RESTARTPCB
22 |     iret

```

然后就在 `kernel.asm` 中写进27号向量表中

```

1 | extern sys_getsema
2 | WRITE_INT_VECTOR 27h, sys_getsema

```

(4) 信号量的释放

有信号量的初始化，必然就有其释放的过程，函数实现如下：

```

1 | void do_FreeSema(int s) {
2 |     semlist[s].used=0;
3 | }

```

函数的输入是需要释放的信号量标号，函数只需要将该标号的信号量变成未被使用（即used=0）

同理，信号量的释放也是在用户程序中进行，需要进行系统调用才能运行内核函数。

```

1 | extern do_FreeSema
2 | global sys_freeseema
3 | sys_freeseema:
4 |     PUSHALLPCB
5 |     call pcbsave
6 |     add sp,16*2
7 |     mov ax,0
8 |     push ax
9 |     push bx
10 |    call dword do_GetSema
11 |    pop bx
12 |    pop ax
13 |    RESTARTPCB
14 |
15 |    iret

```

在 `kernel.asm` 写进28号向量

```

1 | extern sys_freeseema
2 | WRITE_INT_VECTOR 28h, sys_freeseema

```

(5) P操作的实现

P操作的原理如下：

- 首先将信号量s的值减1， $s=s-1$
- 若 $s<0$ ，则需要将该进程转为阻塞状态，排进等待队列当中

具体代码实现(`pcb.h`)如下：

```

1 void do_P(int s) {
2     semlist[s].count--;
3     if (semlist[s].count<0) Blocked(s);
4 }
5 void Blocked(int s)
6 {
7     semlist[s].que++;
8     pcb_table[current_process].zhuangtai=BLOCKED;
9     semlist[s].blocked[semlist[s].que-1]=current_process;
10    schedulepcb();
11 }

```

函数输入为需要进行操作的信号量标号。

关于阻塞该进程，排进等待队列，需要将队列长度+1，然后将当前进程的id放进队列当中，然后转到其他可运行进程（schedulepcb()）

同理，P操作也需要使用系统调用的方式，使得用户程序能够调用内核的函数

```

1 global sys_p
2 extern do_P
3 sys_p:
4     PUSHALLPCB
5     call pcbsave
6     add sp,16*2
7     call dword do_P
8     RESTARTPCB
9
10    iret

```

然后将该系统调用函数写进25号向量中 kernel.asm

```

1 extern sys_p
2 WRITE_INT_VECTOR 25h, sys_p

```

（6）V操作的实现

V操作的原理如下：

- 首先将信号量s的值加1，s=s+1
- 若s<=0，则需要从阻塞队列中释放一个进程使其运行。

具体代码实现(pcb.h)如下：

```

1 void do_V() {
2     int s=0;
3     semlist[s].count++;
4     if (semlist[s].count<=0) WaitUp(s);
5 }
6 void WaitUp(int s)
7 {
8     pcb_table[semlist[s].blocked[0]].zhuangtai=READY;
9     for(int i=0;i<semlist[s].que;i++)
10    {

```

```

11     semlist[s].blocked[i]=semlist[s].blocked[i+1];
12 }
13 semlist[s].que--;
14 }

```

函数输入为需要进行操作的信号量标号。

关于恢复阻塞队列的进程，需要从队列中取出队列头的进程标号，然后将该进程改成READY状态，随后将队列全部往前移动。最后将队列长度-1。

同理，V操作也需要使用系统调用的方式，使得用户程序能够调用内核的函数

```

1  global sys_v
2  extern do_V
3
4  sys_v:
5      PUSHALLPCB
6      call pcbsave
7      add sp,16*2
8      call dword do_V
9      RESTARTPCB
10
11     iret

```

然后将该系统调用函数写进26号向量中 kernel.asm

```

1  extern sys_v
2  WRITE_INT_VECTOR 26h, sys_v

```

(7) pv 测试程序的编写

由于 getsema, freesema,P,V 四个函数需要使用系统调用来进行调用，所以我的测试程序是使用汇编语言来写的。该汇编函数为 pv_test.asm 以及 pv_test2.asm。由于汇编代码比较冗余，所以我使用c来表示该测试程序的思路逻辑。该程序可以在命令行中使用 runall 8 和 runall 9 来调用。

测试程序的思路如下：

首先定义存款共享变量 bankbalance 为50，还有两个记录变量 savemoney, drawmoney 分别记录存钱和取钱的数目。

程序开始，首先使用 getsema(1) 获取一个信号数目为1的信号量，随后fork出子进程。父进程复制存钱10次，子进程负责取款10次。每次存钱和取款都需要使用P，V操作来进行锁定,否则会出现RC问题。

程序的c代码如下：

```

1  int bankbalance= 50;
2  int savemoney=0;
3  int drawmoney=0;
4
5  void cmain()
6  {
7      clearScreen();
8      print("This is the `pv_test` user programme.\r\n");
9      int s=getsema(1);
10     int pid = fork();

```

```

11  if(pid < 0) {
12      print("error in fork!\r\n");
13  }
14  else if(pid > 0) { // 父进程
15      for(int i=0;i<10;i++)
16      {
17          //P(s);
18          int k=bankbalance;
19          k++;
20          bankbalance=k;
21          savemoney++;
22          printf("Parent : bankbalance=%d  totalsave=%d",bankbalance,savemoney);
23          //V(s);
24      }
25
26  }
27  else { // 子进程
28      for(int i=0;i<10;i++)
29      {
30          // P(s);
31          int k=bankbalance;
32          k--;
33          bankbalance=k;
34          frawmoney++;
35          printf("Child : bankbalance=%d  totaldraw=%d",bankbalance,drawmoney);
36          //V(s);
37      }
38  }
39  freesema(s);
40  }

```

为了看出P，V操作的用处，定义了两个程序，一个程序不包含信号量为 `pv_test2.asm`，运行便可以看到RC。另一个程序使用了P，V操作为 `pv_test.asm`，对比运行结果可以看出区别。

(8) 程序的编译与整合

由于程序的编译以及整合是一个大量重复工作，因此我使用bash脚本来快速进行编译与整合，本次实验加上的文件为 `pv_test.asm` 和 `pv_test2.asm`。

`combine.sh`

```

1  #!/bin/bash
2  rm -rf temp
3  mkdir temp
4  rm *.img
5
6  nasm booter.asm -o ./temp/booter.bin
7
8  cd usrprog
9  nasm topleft.asm -o ../temp/topleft.com
10 nasm topright.asm -o ../temp/topright.bin
11 nasm bottomleft.asm -o ../temp/bottomleft.bin
12 nasm bottomright.asm -o ../temp/bottomright.bin
13 nasm list.asm -o ../temp/list.bin
14 nasm sys_test.asm -o ../temp/sys_test.bin
15 nasm fork_test.asm -o ../temp/fork_test.bin
16 nasm pv_test.asm -o ../temp/pv_test.bin

```



```

17 nasm pv_test2.asm -o ../temp/pv_test2.bin
18
19 cd c_test
20 nasm -f elf32 main.asm -o ../temp/main_a.o
21 gcc -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -mpreferred-stack-boundary=2 -lgcc
-shared main.c -fno-pic -o ../temp/main_c.o
22 ld -m elf_i386 -N -Ttext 0xB900 --oformat binary ../temp/main_a.o ../temp/main_c.o -o
../temp/main.bin
23 cd ..
24
25 cd ..
26
27 cd lib
28 nasm -f elf32 system_a.asm -o ../temp/system_a.o
29 gcc -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -mpreferred-stack-boundary=2 -lgcc
-shared system_c.c -fno-pic -o ../temp/system_c.o
30 cd ..
31 cd kernel
32 nasm -f elf32 kernel.asm -o ../temp/kernel.o
33 nasm -f elf32 kernel_a.asm -o ../temp/kernel_a.o
34 nasm -f elf32 ouch.asm -o ../temp/ouch.o
35 nasm -f elf32 pcb.asm -o ../temp/pcb.o
36 gcc -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -mpreferred-stack-boundary=2 -lgcc
-shared kernel_c.c -fno-pic -o ../temp/kernel_c.o
37 ld -m elf_i386 -N -Ttext 0x7E00 --oformat binary ../temp/kernel.o ../temp/kernel_a.o
../temp/kernel_c.o ../temp/ouch.o ../temp/system_a.o ../temp/system_c.o ../temp/pcb.o -o
../temp/kernel.bin
38 cd ..
39 rm ./temp/*.o
40
41 dd if=../temp/booter.bin of=myosv7.img bs=512 count=1 2>/dev/null
42 dd if=../temp/kernel.bin of=myosv7.img bs=512 seek=1 count=35 2>/dev/null
43 dd if=../temp/topleft.com of=myosv7.img bs=512 seek=36 count=2 2>/dev/null
44 dd if=../temp/topright.bin of=myosv7.img bs=512 seek=38 count=2 2>/dev/null
45 dd if=../temp/bottomleft.bin of=myosv7.img bs=512 seek=40 count=2 2>/dev/null
46 dd if=../temp/bottomright.bin of=myosv7.img bs=512 seek=42 count=2 2>/dev/null
47 dd if=../temp/list.bin of=myosv7.img bs=512 seek=44 count=2 2>/dev/null
48 dd if=../temp/main.bin of=myosv7.img bs=512 seek=46 count=18 2>/dev/null
49
50 dd if=../temp/sys_test.bin of=myosv7.img bs=512 seek=64 count=2 2>/dev/null
51 dd if=../temp/fork_test.bin of=myosv7.img bs=512 seek=66 count=2 2>/dev/null
52 dd if=../temp/pv_test.bin of=myosv7.img bs=512 seek=68 count=2 2>/dev/null
53 dd if=../temp/pv_test2.bin of=myosv7.img bs=512 seek=70 count=2 2>/dev/null
54
55 echo "[+] Done."
56
57

```

该脚本需要严格对应磁盘的放置，譬如 `dd` 时的扇区号，以及 `ld` 中 `-Ttext 0x7E00` 需要严格对照内存放置情况，不然会导致错误。

5. 实验过程

1) 踩坑过程

- 软盘结构

一个 3.5 英寸的 1.44 MB 的软盘由 80 个磁道、18 个扇区构成，而且有 2 个柱面。首先使用的是 0 柱面、0 磁道的扇区，扇区编号从 1 到 18。再往后，是 0 柱面 1 磁道，扇区号又是从 1 到 18。由于现在文件数越来越多，占用的扇区数量也多了起来。虽然我对软盘的结构已经比较熟悉，但是一个稍不注意就错误了，导致找了很久的bug

- RC问题的检测

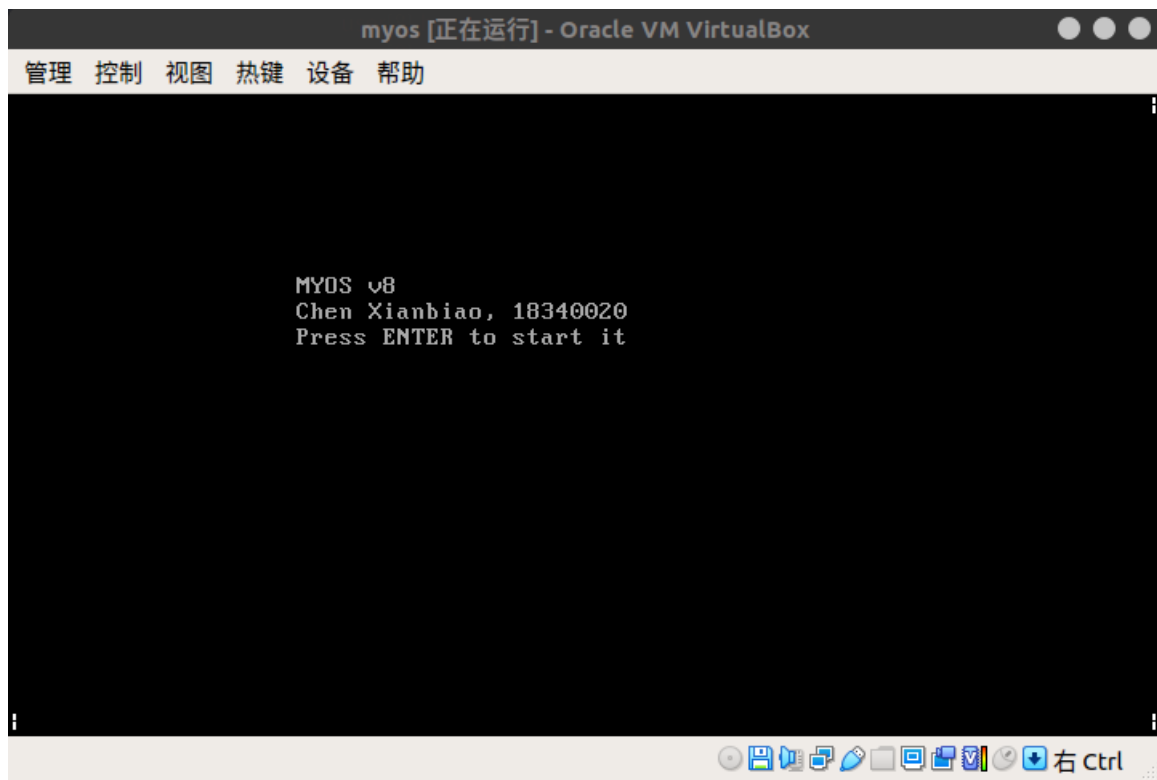
一开始写 `pv_test` 错误程序的时候，我发现程序并没有出现RC问题，原因是指令数目太少，导致出现的概率太低，因此我在会出现RC的指令中间加入了一个循环，并且加快时钟中断的频率，这样几乎每次运行我都能检测到RC问题。

- P操作的编写

P操作的时候，当检测到信号量 <0 时，需要阻塞进程，但是我一开始忘记阻塞之后调度，导致程序一直卡在某一个进程当中。

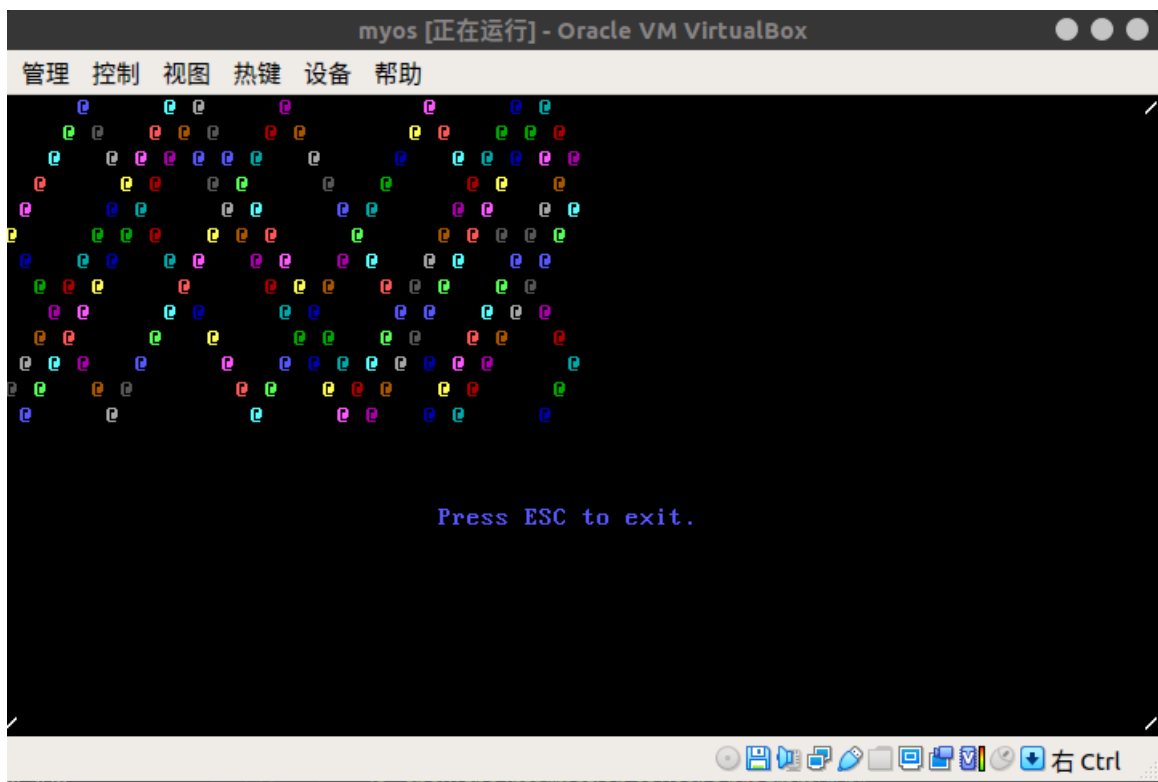
2) 实验结果展示

- 启动虚拟机，进入开始画面，可以看到风火轮仍然存在

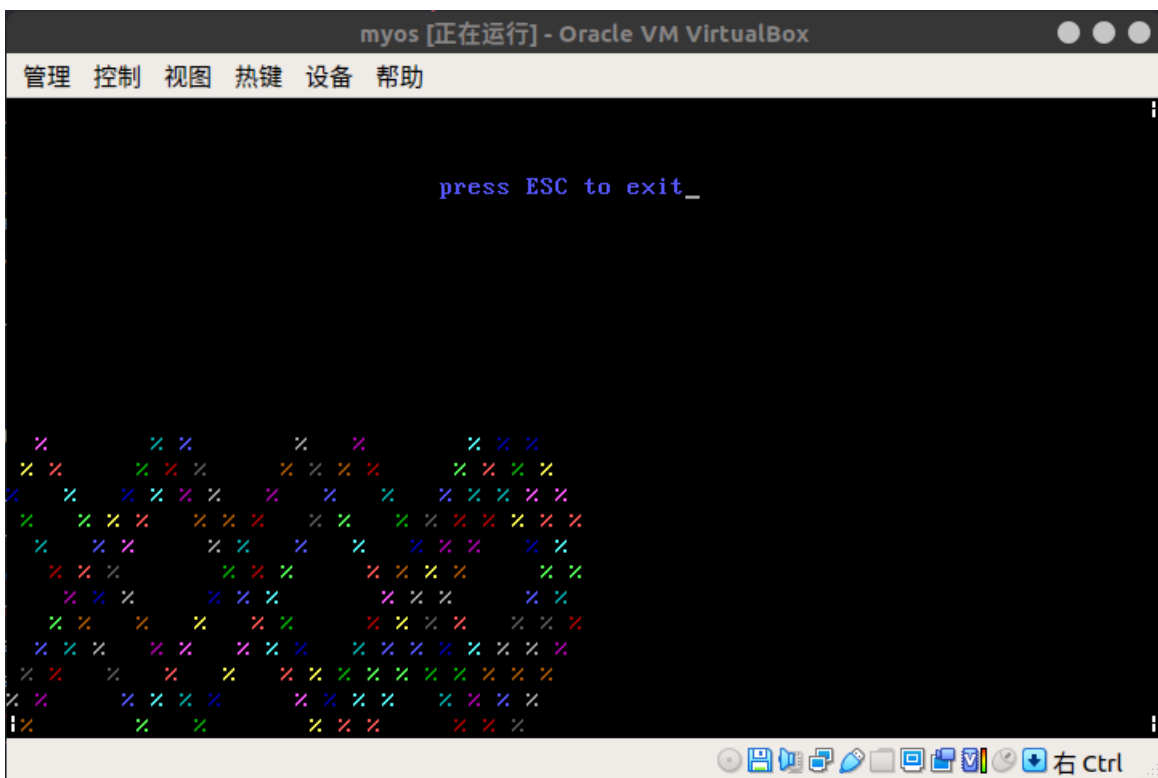


- 运行 `run 13` 测试之前的指令

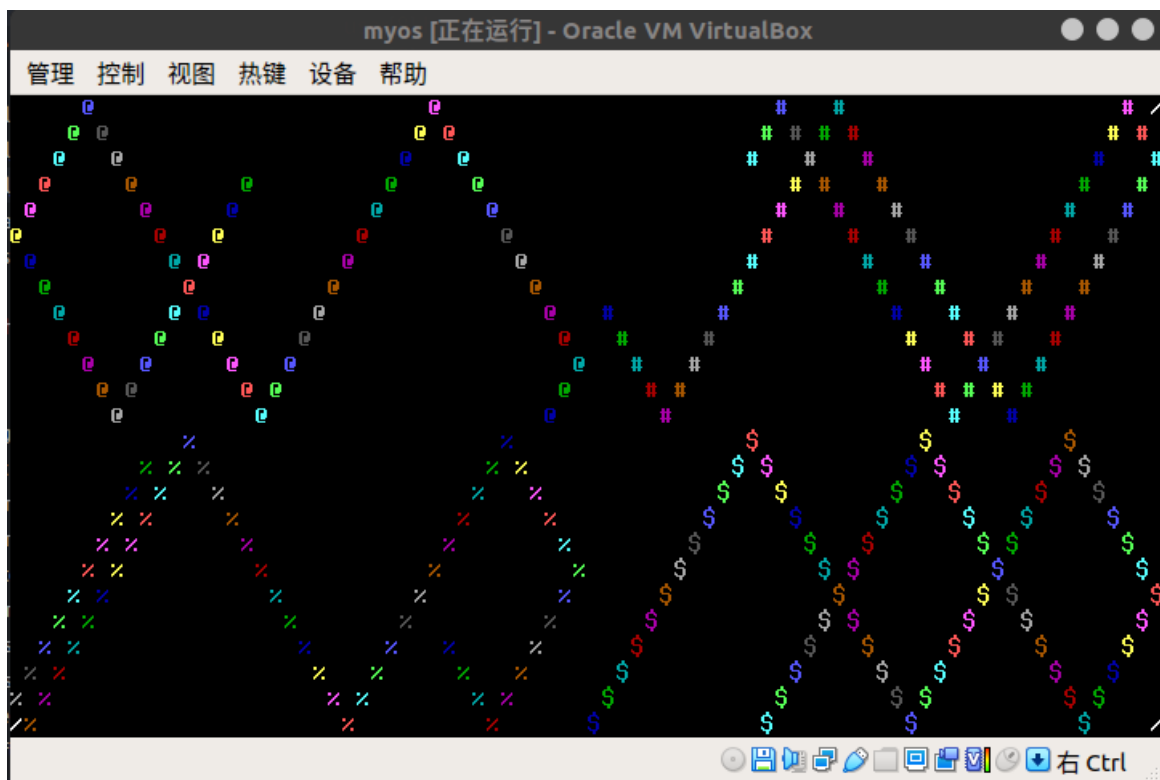
1: 左上角



按 esc，3：左下角



- 测试之前的多程序运行状态 `runall 1234`



- 测试有RC问题的 `pv_test` 程序，`runall 8`

```

myos [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
This is the 'pv_test' programme.
Parent : bankbalance=51      totalsave= 01
Parent : bankbalance=52      totalsave= 02
Parent : bankbalance=53      totalsave= 03
Parent : bankbalance=54      totalsave= 04
Child  : bankbalance=53      totaldraw= 01
Child  : bankbalance=52      totaldraw= 02
Child  : bankbalance=51      totaldraw= 03
Child  : bankbalance=50      totaldraw= 04
Child  : bankbalance=49      totaldraw= 05
Child  : bankbalance=48      totaldraw= 06
Child  : bankbalance=47      totaldraw= 07
Child  : bankbalance=46      totaldraw= 08
Child  : bankbalance=45      totaldraw= 09
Child  : bankbalance=44      totaldraw= 10
Parent : bankbalance=55      totalsave= 05
Parent : bankbalance=56      totalsave= 06
Parent : bankbalance=57      totalsave= 07
Parent : bankbalance=58      totalsave= 08
Parent : bankbalance=59      totalsave= 09
Parent : bankbalance=60      totalsave= 10_

```

可以看到运行中出现很多的 `rc` 问题，最容易看出来就是结果是错误的，本金是50元，存了10块，然后取了10块，那么结果应该是50元，而最后是60元，说明发生了RC问题。

我再运行一次，看还没有这个问题：

```
myos [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
This is the 'pv_test' programme.
Parent : bankbalance=51      totalsave= 01
Parent : bankbalance=52      totalsave= 02
Parent : bankbalance=53      totalsave= 03
Parent : bankbalance=54      totalsave= 04
Child  : bankbalance=53      totaldraw= 01
Child  : bankbalance=52      totaldraw= 02
Child  : bankbalance=51      totaldraw= 03
Child  : bankbalance=50      totaldraw= 04
Child  : bankbalance=49      totaldraw= 05
Child  : bankbalance=48      totaldraw= 06
Child  : bankbalance=47      totaldraw= 07
Parent : bankbalance=55      totalsave= 05
Parent : bankbalance=56      totalsave= 06
Parent : bankbalance=57      totalsave= 07
Parent : bankbalance=58      totalsave= 08
Parent : bankbalance=59      totalsave= 09
Parent : bankbalance=60      totalsave= 10
Child  : bankbalance=46      totaldraw= 08
Child  : bankbalance=45      totaldraw= 09
Child  : bankbalance=44      totaldraw= 10
```

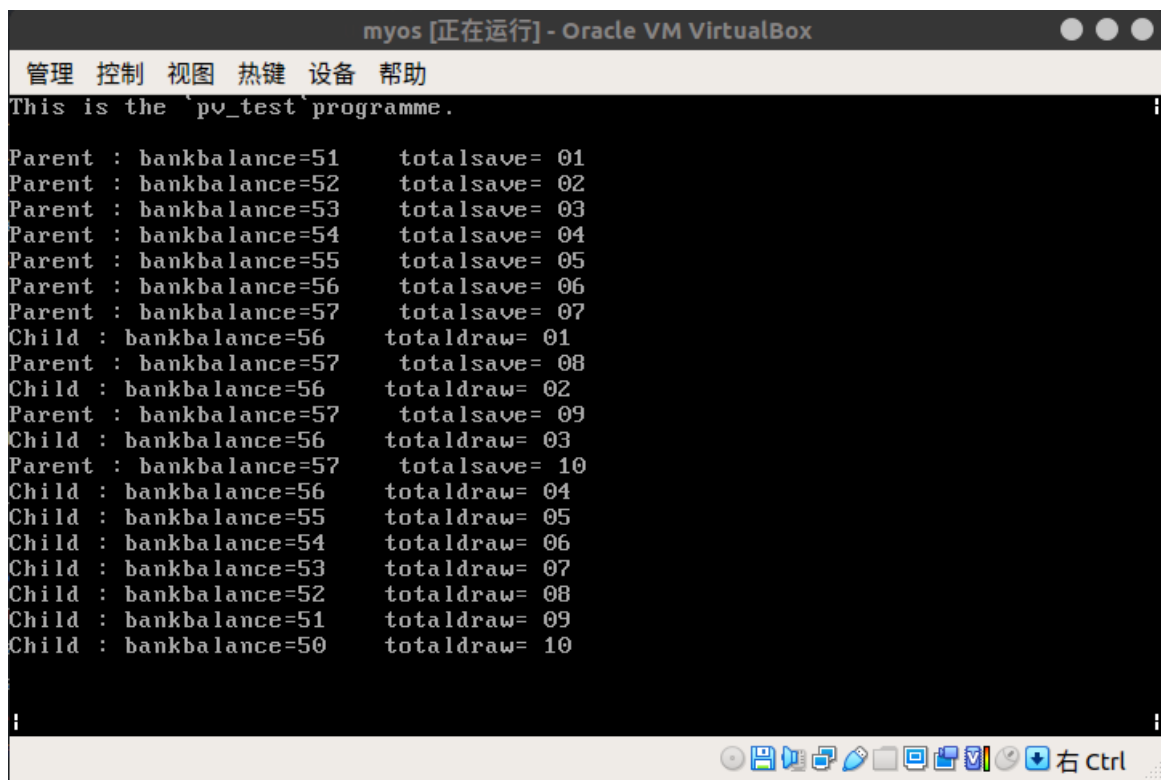
第二次运行还是出现了，最后的钱是44元

- 随后，运行使用P,V操作后的test程序，runall9

```
myos [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
This is the 'pv_test' programme.
Parent : bankbalance=51      totalsave= 01
Parent : bankbalance=52      totalsave= 02
Parent : bankbalance=53      totalsave= 03
Parent : bankbalance=54      totalsave= 04
Parent : bankbalance=55      totalsave= 05
Parent : bankbalance=56      totalsave= 06
Parent : bankbalance=57      totalsave= 07
Parent : bankbalance=58      totalsave= 08
Child  : bankbalance=57      totaldraw= 01
Parent : bankbalance=58      totalsave= 09
Child  : bankbalance=57      totaldraw= 02
Parent : bankbalance=58      totalsave= 10
Child  : bankbalance=57      totaldraw= 03
Child  : bankbalance=56      totaldraw= 04
Child  : bankbalance=55      totaldraw= 05
Child  : bankbalance=54      totaldraw= 06
Child  : bankbalance=53      totaldraw= 07
Child  : bankbalance=52      totaldraw= 08
Child  : bankbalance=51      totaldraw= 09
Child  : bankbalance=50      totaldraw= 10
```

可以看到，使用P,V操作后，程序不再出现RC问题，最后的钱是50.

我再一次运行来进行验证：



```
myos [正在运行] - Oracle VM VirtualBox
管理 控制 视图 热键 设备 帮助
This is the 'pv_test' programme.
Parent : bankbalance=51      totalsave= 01
Parent : bankbalance=52      totalsave= 02
Parent : bankbalance=53      totalsave= 03
Parent : bankbalance=54      totalsave= 04
Parent : bankbalance=55      totalsave= 05
Parent : bankbalance=56      totalsave= 06
Parent : bankbalance=57      totalsave= 07
Child  : bankbalance=56      totaldraw= 01
Parent : bankbalance=57      totalsave= 08
Child  : bankbalance=56      totaldraw= 02
Parent : bankbalance=57      totalsave= 09
Child  : bankbalance=56      totaldraw= 03
Parent : bankbalance=57      totalsave= 10
Child  : bankbalance=56      totaldraw= 04
Child  : bankbalance=55      totaldraw= 05
Child  : bankbalance=54      totaldraw= 06
Child  : bankbalance=53      totaldraw= 07
Child  : bankbalance=52      totaldraw= 08
Child  : bankbalance=51      totaldraw= 09
Child  : bankbalance=50      totaldraw= 10
```

可以看到这次运行，父子进程的次序虽然是不一样的，但是程序是正确的，没有出现RC问题。

- 输入 shutdown 关机

5.实验体会

本次实验八实现了信号量的编写方式。在原理课当中，老师和我们已经强调过很多次信号量的重要性，本次实验正是验证了这个重要性。之前在并行实验课程中使用过 openmp 并行编程来模拟过RC问题以及解决，但是本次实验是使用自己的操作系统的并行方式来进行，更加的有实感。

虽然信号量的几个函数实现起来代码量并不是很多，但是模拟RC问题的出现和解决才是本次使用的关键之处。由于调用信号量的是用户程序，而信号量是在内核当中，因此需要使用系统中断的方式才能成功使用，这也让我不断地巩固了前面实验所学的知识。

本次实验需要使用两个进程运行同一段的代码，使用到了实验7的fork程序，但是本次实验不同的是父子进程是需要同时运行的，因此本次实验不需要调用wait，和wakeup等函数，只需要使用fork。

本次实验，我加深理解了基于信号量的进程的同步方法，也掌握实现信号量及其应用的方式，收获满满。

6.参考资料

- PV操作：<https://blog.csdn.net/rocky1996/article/details/95172660>
- 操作系统基础-信号量机制的理解：https://blog.csdn.net/weixin_43616178/article/details/89415733