操作系统-实验六

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、学习多道程序与CPU分时技术

2、掌握操作系统内核的二态进程模型设计与实现方法

3、掌握进程表示方法

4、掌握时间片轮转调度的实现

2.实验要求

1、了解操作系统内核的二态进程模型

2、扩展实验五的的内核程序，增加一条命令可同时创建多个进程分时运行，增加进程控制块和进程表数据结构。

4、修改时钟中断处理程序，调用时间片轮转调度算法。

5、设计实现时间片轮转调度算法，每次时钟中断，就切换进程，实现进程轮流运行。

5、修改save()和restart()两个汇编过程，利用进程控制块保存当前被中断进程的现场，并从进程控制块恢复下一个进程的现场。

6、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) 修改实验5的内核代码，定义进程控制块PCB类型，包括进程号、程序名、进程内存地址信息、CPU寄存器保存区、进程状态等必要数据项，再定义一个PCB数组，最大进程数为10个。

2) 扩展实验五的的内核程序，增加一条命令可同时执行多个用户程序，内核加载这些程序，创建多个进程，再实现分时运行

3) 修改时钟中断处理程序，保留无敌风火轮显示，而且增加调用进程调度过程

4) 内核增加进程调度过程：每次调度，将当前进程转入就绪状态，选择下一个进程运行，如此反复轮流运行。

5) 修改save()和restart()两个汇编过程，利用进程控制块保存当前被中断进程的现场，并从进程控制块恢复下一个进程的运行。

6) 实验5的内核其他功能，如果不必要，可暂时取消服务。

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	24	C语言用户程序
1	1	17	2	系统调用测试程序

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

（2）MY操作系统内核的设计

内核程序结构：

新增了关于进程调度的函数模块`pcb.asm`，`pcb.h`

程序名	代码形式	作用
`kernel.asm`	`ASM`	内核的入口，调用c中的函数，*新增运行风火轮代码*
`kernel_a.asm`	`ASM`	包含一些显示打印，IO借口，扇区加载的函数,*运行`ouch`代码*
`stdio.h`	C	包含一些对字符串处理的函数
`kernel_c.asm`	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）
`ouch.asm`	`ASM`	封装了关于`ouch`键盘中断的代码
`system_a.asm`	`ASM`	包含关于系统调用（汇编）的函数
`system_c.c`	C	包含关于系统调用（c语言）的函数
`pcb.asm`	`ASM`	封装了关于`PCB`保护和恢复的函数（新增）
`pcb.h`	C	封装了PCB的创建，初始化，进程调度，进程撤销的函数（新增）

以下是我已经实现的指令功能（**新增了**`runall`指令）

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

（3）进程控制块PCB的创建

为保护各个进程的上下文，我在`pcb.h`上创建了进程控制块，使用c语言结构体，定义如下

```
#define processnum 10
struct my_Register{
  uint16_t ax;   // 0
  uint16_t cx;   // 2
  uint16_t dx;   // 4
  uint16_t bx;   // 6
  uint16_t sp;   // 8
```

```
    uint16_t bp;    // 10
    uint16_t si;    // 12
    uint16_t di;    // 14
    uint16_t ds;    // 16
    uint16_t es;    // 18
    uint16_t fs;    // 20
    uint16_t gs;    // 22
    uint16_t ss;    // 24
    uint16_t ip;    // 26
    uint16_t cs;    // 28
    uint16_t flags; // 30
};
typedef struct PCB{
    struct my_Register reg;
    uint8_t id;
    uint8_t zhuangtai;
}PCB;
PCB pcb_table[processnum];
```

一个进程控制块PCB有三个部分，一个是id控制块的编号，一个是`zhuangtai`状态（现在是而状态模型，0表示未激活，1表示就绪，2表示正在运行），最后一个是reg存在进程的上下文寄存器

```
uint16_t current_process=0;
```

定义一个当前进程号的变量`current_process`，内核的进程号为0，其余的是1~9

为方便PCB在c语言模块和汇编模块的调用，我编写了一个获取当前进程的进程控制块函数

```
PCB* getcurrentpcb()
{
    return &pcb_table[current_process];
}
```

在汇编中调用只需要

```
extern getcurrentpcb
call dword getcurrentpcb
```

（4）（修改）save()和restart()的编写

save()和restart()的编写在实验5中已经实现过，但是当时是对于一个进程的保护，而不是对多个进程控制块的保护，因此在本次实验中，我进行了修改。

首先是进入中断，需要立即将所有寄存器压栈保护（使用宏），该部分在实验五已经写过，但是有一个坑很关键，会在后文踩坑

```
%macro  PUSHALLPCB 0
    push ss
    push gs
    push fs
    push es
    push ds
    push di
    push si
    push bp
```

```
    push sp
    push bx
    push dx
    push cx
    push ax

    mov ax,cs;重点在这里
    mov ds,ax;重点在这里
    mov es,ax;重点在这里
%endmacro
```

随后就可以调用 `pcbsave（）` 来将栈中的寄存器转移进结构体当中。注意：在进入时钟中断后，首先栈中会增加 `psw`、cs、 `ip` 共 3 个字。因此我之前只压栈了13个寄存器，但栈中已经有了16个值，可以看下面代码中的注释

```
extern getcurrentpcb
pcbsave:
    pusha
    mov bp, sp
    add bp, 16+2            ; 参数首地址

    call dword getcurrentpcb
    mov di, ax

    mov ax, [bp]
    mov [cs:di], ax
    mov ax, [bp+2]
    mov [cs:di+2], ax
    mov ax, [bp+4]
    mov [cs:di+4], ax
    mov ax, [bp+6]
    mov [cs:di+6], ax
    mov ax, [bp+8]
    mov [cs:di+8], ax
    mov ax, [bp+10]
    mov [cs:di+10], ax
    mov ax, [bp+12]
    mov [cs:di+12], ax
    mov ax, [bp+14]
    mov [cs:di+14], ax
    mov ax, [bp+16]
    mov [cs:di+16], ax
    mov ax, [bp+18]
    mov [cs:di+18], ax
    mov ax, [bp+20]
    mov [cs:di+20], ax
    mov ax, [bp+22]
    mov [cs:di+22], ax
    mov ax, [bp+24]
    mov [cs:di+24], ax
    mov ax, [bp+26]
    mov [cs:di+26], ax
    mov ax, [bp+28]
    mov [cs:di+28], ax
    mov ax, [bp+30]
    mov [cs:di+30], ax

```

```
    popa
    ret

```

通过压栈，并从栈中转移进结构体，能够完整地保存寄存器

`restart ()` 的实现。之后问题就是如何将结构体中的值在中断程序的最后放回原来的寄存器当中，我编写了一个宏来表示RESTART。

注意1：因为在汇编中总需要一个寄存器来进行操作，因此我选择 `si` 来进行，因此 `si` 的值的恢复需要在最后恢复

注意2：关于 `sp` 值的恢复，我们需要恢复的 `sp` 值是在进入中断前用户的 `sp` 值，但由于我在压栈 `sp` 之前栈中会增加 `psw、cs、ip` 共 3 个字，然后又压入了 `ss、gs、fs、es、ds、di、si、bp` 共 8 个字，加起来是11个字，因此最后的 `sp` 需要+22个字节，才能最终恢复。

在恢复完寄存器后，我再一次将 `psw、cs、ip` 压栈，因为在中断恢复需要这三个寄存器来回去。

```
%macro  RESTARTPCB 0        ;宏：从PCB中恢复寄存器的值
    call dword getcurrentpcb
    mov si, ax
    mov ax, [cs:si+0]
    mov cx, [cs:si+2]
    mov dx, [cs:si+4]
    mov bx, [cs:si+6]
    mov sp, [cs:si+8]
    mov bp, [cs:si+10]
    mov di, [cs:si+14]
    mov ds, [cs:si+16]
    mov es, [cs:si+18]
    mov fs, [cs:si+20]
    mov gs, [cs:si+22]
    mov ss, [cs:si+24]
    add sp, 11*2            ;恢复正确的sp
    push word[cs:si+30]       ;新进程flags
    push word[cs:si+28]       ;新进程cs
    push word[cs:si+26]       ;新进程ip
    push word[cs:si+12]
    pop si                  ;恢复si
%endmacro
```

（5）时钟中断（进程调度）的编写

二进程状态的模型就是其他等待，一个运行的两个状态，因此实现的思路是比较通俗易懂的：就是进入时钟中断，将当前运行状态进程的寄存器保护起来，然后将当前进程设为等待状态，然后从等待进程中取出一个进程，将其设为运行状态，然后将该进程的寄存器（上下文）恢复，然后退出时钟中断。

另外就是在正常情况下，我们不进行多状态运行，所以需要设置一个状态变量t_flag表示当前是否是执行多个进程，当该量为0时，时钟中断不进行进程调度，否则进行

此外还有一个退出多进程的设置，当输入 `esc` 时，重置所以PCB，然后退出回内核代码

在 `pcb.asm` 中定义，然后c语言使用 `extern uint16_t t_flag;`

```
global t_flag
t_flag dw 0
```

时钟中断伪代码：

```
pcbtimer:
   cmp word[cs:t_flag], 0;首先判断是否是多进程状态
   je timequit;若否则跳到最后

   PUSHALLPCB
   call pcbsave;进程上下文保护

checkesc:;判断是否输入esc，若输入则重置所有PCB，然后返回内核
   mov ah ,01h
   int 16h
   jz continuetimer
   mov ah,0
   int 16h
   cmp al,27
   jne continuetimer

  call dword reset;进程重置函数
   jmp restart
continuetimer:
   call dword schedule;进程调度函数

restart:
   RESTARTPCB;进程寄存器恢复

timequit:
   ......无敌风火轮的代码

   push ax
   mov al, 20h
   out 20h, al
   out 0A0h, al
   pop ax
   ;popa
   iret;中断退出
```

（6）进程调度函数

由于只是二状态模型，因此进程调度则是，首先将当前运行状态改为等待状态，然后遍历所有进程，将第一个等待进程转为运行状态，实现在 `./kernel/pcb.h`

```
void schedule()
{
   pcb_table[current_process].zhuangtai=1;
   current_process++;
   if(current_process>=processnum)current_process=1;
   while(pcb_table[current_process].zhuangtai!=1)
   {
      current_process++;
      if(current_process>=processnum)current_process=1;
   }
   pcb_table[current_process].zhuangtai=2;
}
```

该函数在时钟中断中被调用

```
extern schedule
```

（7）进程重置函数

当多进程时，时钟中断中检测到`esc`，则重置进程（将所有PCB（除0）变回初态，然后当前进程id设置回0（内核进程）），该函数实现在`./kernel/pcb.h`

```
void reset()
{
  for(int i=1;i<processnum;i++)
  {
    pcb_table[i].id=i;
    pcb_table[i].zhuangtai=0;

    pcb_table[i].reg.ax=0;
    pcb_table[i].reg.cx=0;
    pcb_table[i].reg.dx=0;
    pcb_table[i].reg.bx=0;
    pcb_table[i].reg.sp=0xFE00;
    pcb_table[i].reg.si=0;
    pcb_table[i].reg.di=0;
    pcb_table[i].reg.ds=0;
    pcb_table[i].reg.es=0;
    pcb_table[i].reg.fs=0;
    pcb_table[i].reg.gs=0xB800;
    pcb_table[i].reg.ss=0;
    pcb_table[i].reg.ip=0;
    pcb_table[i].reg.cs=0;
    pcb_table[i].reg.flags=512;
  }
  current_process=0;
  t_flag=0;
}
```

该函数在时钟中断中被调用

```
extern reset
```

（8）`runall`命令的编写

与之前的`run`指令不一样，`run`指令是依次执行用户程序，而`runall`指令是将所有要运行的用户程序放进内存，然后"同时"运行。

首先，使用`isnum124（）`函数判断要运行的程序编号是否是1~4，若不是则报错。

然后，使用`loadm()`函数将要运行的用户程序的写进内存，然后把对应PCB的状态`zhuangtai`设置为就绪状态，然后需要将该PCB的段寄存器设置为该用户程序的段地址。

最后，将`t_flag`设置为1（多进程运行状态），**注意**设置完之后要调用一个循环来进行延迟，该点会在后文踩坑过程提及。

```
else if(strcmp(first,commands[6])==0)//runall
        {

           int flag=1;
           int flag124[4];
           for(int i=0;i<4;i++)flag124[i]=0;
           for(int i=0;i<backlen;i++)
           {
              if(isnum124(back[i])!=1&&back[i]!=' '){flag=0;break;}

           }
           if(flag==1)
           {
              for(int i=0;i<backlen;i++)
               {
                 if(back[i]=='1'&&flag124[0]==0)
                 {
                    flag124[0]=1;
                   loadm(user[0].size, user[0].head,user[0].cylinder,user[0].sector,user[0].offset);
                   pcb_table[1].zhuangtai=1;
                   pcb_table[1].reg.cs=user[0].offset;//设置段寄存器为该用户程序的段地址
                   pcb_table[1].reg.ds=user[0].offset;
                   pcb_table[1].reg.es=user[0].offset;
                   pcb_table[1].reg.fs=user[0].offset;
                   pcb_table[1].reg.ss=user[0].offset;
                 }

                 else if(back[i]=='2'&&flag124[1]==0)
                 {
                    flag124[1]=1;
                    loadm(user[1].size,user[1].head,user[1].cylinder,user[1].sector,user[1].offset);
                   pcb_table[2].zhuangtai=1;
                   pcb_table[2].reg.cs=user[1].offset;
                   pcb_table[2].reg.ds=user[1].offset;
                   pcb_table[2].reg.es=user[1].offset;
                   pcb_table[2].reg.fs=user[1].offset;
                   pcb_table[2].reg.ss=user[1].offset;
                 }
                 else if(back[i]=='3'&&flag124[2]==0)
                 {
                    flag124[2]=1;
                      loadm(user[2].size,user[2].head,user[2].cylinder,user[2].sector,user[2].offset);
                   pcb_table[3].zhuangtai=1;
                   pcb_table[3].reg.cs=user[2].offset;
                   pcb_table[3].reg.ds=user[2].offset;
                   pcb_table[3].reg.es=user[2].offset;
                   pcb_table[3].reg.fs=user[2].offset;
                   pcb_table[3].reg.ss=user[2].offset;
                 }

                 else if(back[i]=='4'&&flag124[3]==0)
                 {
                    flag124[3]=1;
                      loadm(user[3].size,user[3].head,user[3].cylinder,user[3].sector,user[3].offset);
                    pcb_table[4].zhuangtai=1;
                   pcb_table[4].reg.cs=user[3].offset;
                   pcb_table[4].reg.ds=user[3].offset;
```

```
                pcb_table[4].reg.es=user[3].offset;
                pcb_table[4].reg.fs=user[3].offset;
                pcb_table[4].reg.ss=user[3].offset;
              }


              //timer_flag = 0; // 禁止时钟中断处理多进程
            }
            t_flag = 1; // 允许时钟中断处理多进程
            for( int i=0;i<5000;i++ )
              for(int  j=0;j<5000;j++ )
              {
                j++;
                j--;
                }
            t_flag = 0;
             clearScreen();
          }
          else
          {
            char* inf="wrong program name";
            print(inf);
            NEXTLINE;
          }
        }
```

（9）用户程序的修改，以及内存写入

由于之前用户程序是单独运行的，因此我是放在同一个段地址的，但是这次需要同时进行，所以要设置到不同的段地址

```
user[0].offset=0xB10;//用户程序1
user[1].offset=0xB50;//用户程序2
user[2].offset=0xB90;//用户程序3
user[3].offset=0xBD0;//用户程序4
```

然后由于原先用户程序是在程序当中检测是否有`esc`输入来进行退出的，但是进行多进程状态后，多个程序都检测输入显然是不妥的，于是可以判断是否是多进程状态（通过系统调用）选择跳过退出的环节，进入无限循环，而程序的退出通过时钟中断来进行

```
    mov ah, 05h        ; 功能号：获取t_flag
    int 21h          ; ax=t_flag
    cmp ax, 1
    je continue

    mov ah, 01h        ; 功能号：查询键盘缓冲区但不等待
    int 16h
    jz continue        ; 无键盘按下，继续
    mov ah, 0          ; 功能号：查询键盘输入
    int 16h
    cmp al, 27         ; 是否按下ESC
    je QuitUsrProg      ; 若按下ESC，退出用户程序

continue:
```

```
    jmp loop1

```

（10）获取t_flag的系统调用

由于在用户程序中无法直接获取内核中的t_flag的私有变量，因此需要系统调用来进行访问

```
extern t_flag
gettimeflag:
    mov ax,[cs:t_flag]
    ret
;随后将该函数写进实验5中systemcall的05h向量表中
```

调用方式为：

```
    mov ah, 05h         ; 功能号：获取t_flag
    int 21h             ; ax=t_flag
```

（11）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有pcb.asm pcb.h 。

combine.sh

```
#!/bin/bash
rm -rf temp
mkdir temp
rm *.img

nasm booter.asm -o ./temp/booter.bin

cd usrprog
nasm topleft.asm -o ../temp/topleft.com
nasm topright.asm -o ../temp/topright.bin
nasm bottomleft.asm -o ../temp/bottomleft.bin
nasm bottomright.asm -o ../temp/bottomright.bin
nasm list.asm -o ../temp/list.bin
nasm sys_test.asm -o ../temp/sys_test.bin

cd c_test
nasm -f elf32 main.asm -o ../../temp/main_a.o
gcc -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -mpreferred-stack-boundary=2 -lgcc -shared main.c -fno-pic  -o ../../temp/main_c.o
ld -m elf_i386 -N -Ttext 0xB900 --oformat binary ../../temp/main_a.o  ../../temp/main_c.o -o ../../temp/main.bin
cd ..

cd ..

cd lib
nasm -f elf32  system_a.asm -o ../temp/system_a.o
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
```

```
cd ..
cd kernel
nasm -f elf32 kernel.asm -o ../temp/kernel.o
nasm -f elf32 kernel_a.asm -o ../temp/kernel_a.o
nasm -f elf32 ouch.asm -o ../temp/ouch.o
nasm -f elf32 pcb.asm -o ../temp/pcb.o
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
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
cd ..
rm ./temp/*.o

dd if=./temp/booter.bin of=myosv6.img bs=512 count=1 2>/dev/null
dd if=./temp/kernel.bin of=myosv6.img bs=512 seek=1 count=35 2>/dev/null
dd if=./temp/topleft.com of=myosv6.img bs=512 seek=36 count=2 2>/dev/null
dd if=./temp/topright.bin of=myosv6.img bs=512 seek=38 count=2 2>/dev/null
dd if=./temp/bottomleft.bin of=myosv6.img bs=512 seek=40 count=2 2>/dev/null
dd if=./temp/bottomright.bin of=myosv6.img bs=512 seek=42 count=2 2>/dev/null
dd if=./temp/list.bin of=myosv6.img bs=512 seek=44 count=2 2>/dev/null
dd if=./temp/main.bin of=myosv6.img bs=512 seek=46 count=24 2>/dev/null
dd if=./temp/sys_test.bin of=myosv6.img bs=512 seek=70 count=2 2>/dev/null
echo "[+] Done."
```

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

- `save` `restart`的实现

这次实验无疑最难的就是这两个寄存器的保护机制。save的过程是进入中断后将所有寄存器存储保护到结构体当中，restart是在中断退出前将结构体中所有寄存器还原回去，然后回到调用的程序当中。这样听起来思路好像很清晰，但是寄存器本身的作用会影响到这两个过程。譬如在restart的过程中需要用到一个辅助寄存器来把值还原，但是这个辅助寄存器也需要还原，所以就需要一个恢复的策略来准确的将所有寄存器都一个不漏保护起来，然后一个不漏地恢复回去

- `sp`寄存器的处理

在寄存器保护恢复的过程中，最为特别的就是`sp`寄存器了，因为我是使用压栈的操作来先进行保护所有寄存器的。因此`sp`寄存器在压栈后会有许多的变化。而且**最重要**的一个点就是在在中断进入后系统会将`psw`、`cs`、`ip`三个寄存器压栈。然后我在进入中断后，在`sp`压栈之前已经将8个寄存器压栈了。所以在`restart`过程中取出的`sp`寄存器是偏移了8+3个字的，所以在恢复的过程中，`sp`还需要+22。才能完全回到原来用户程序的`sp`

- `t flag`用户程序的访问

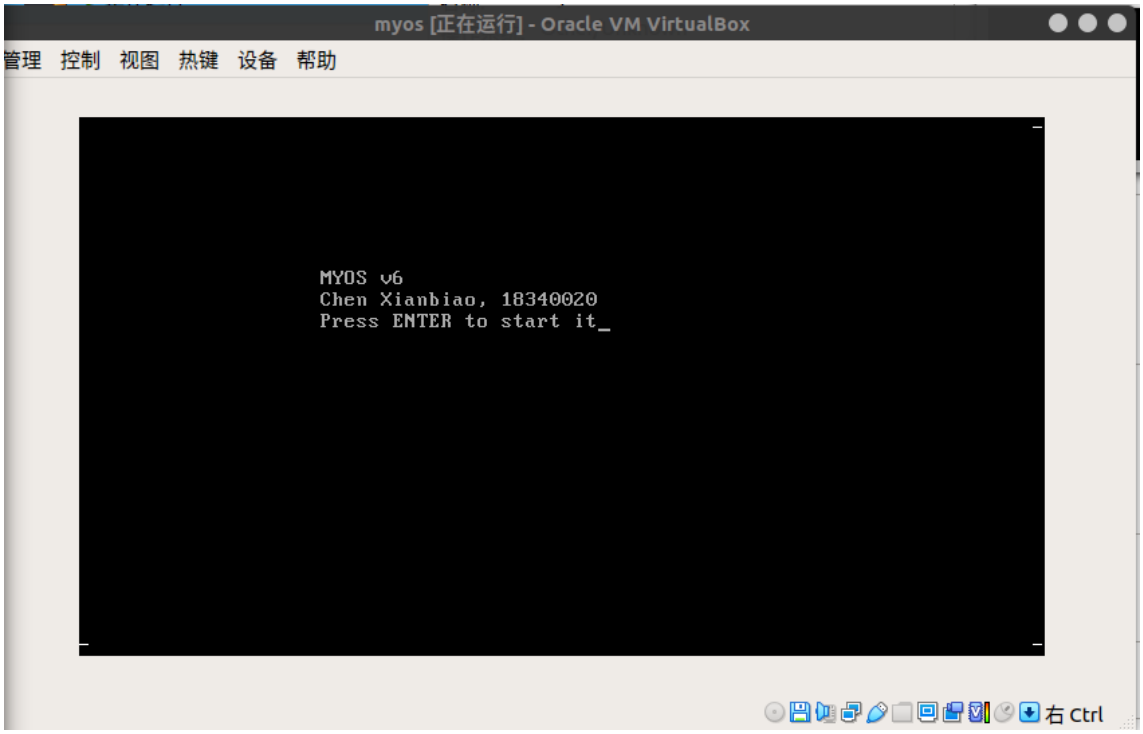
由于原来的用户程序是单独使用`exc`来退出的，但是当一起运行的时候就不能继续这样做，因此我需要判断当前运行状态是否是并行，然后使用中断来获取内核中的`t_flag`,随后分别处理

- 用户程序栈和内存的处理

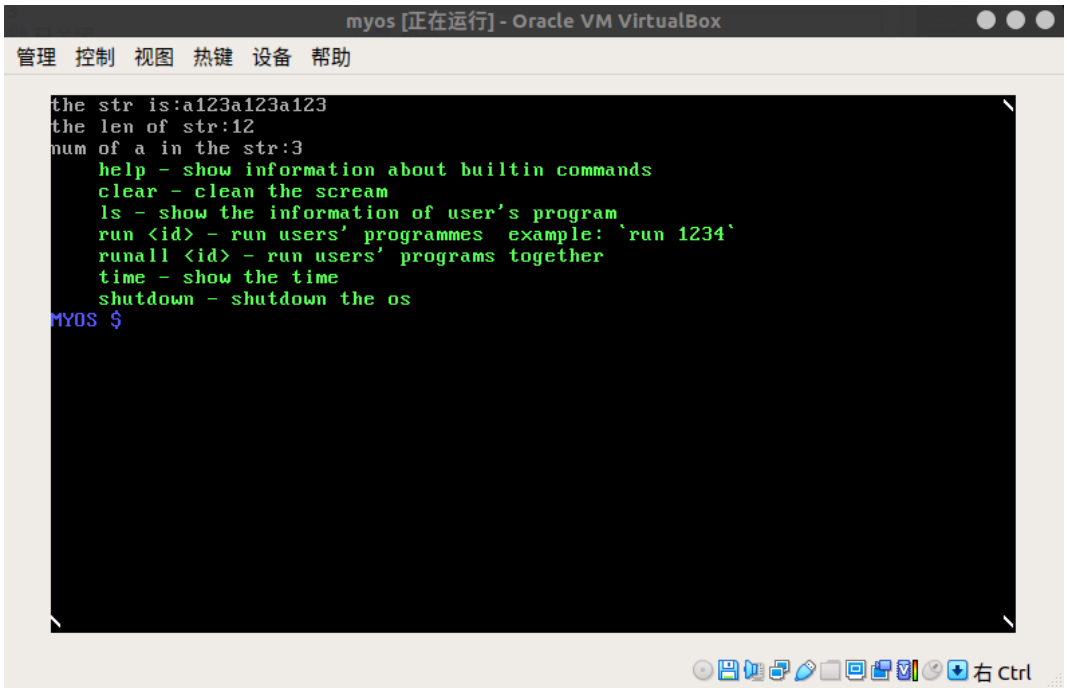
在原来的实验当中，由于用户程序都是单独运行的，所以我把所有用户程序都放在同一个段和内存当中，但是本次实验需要放在不同的位置。

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



- 回车进入命令行

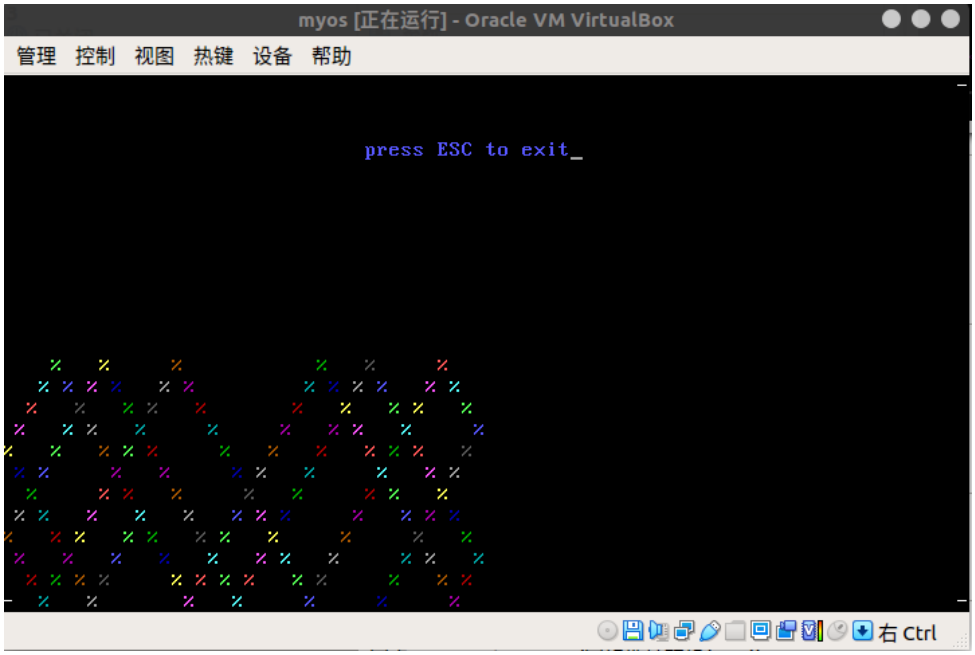


- 测试原有`run 1432`指令，单步运行

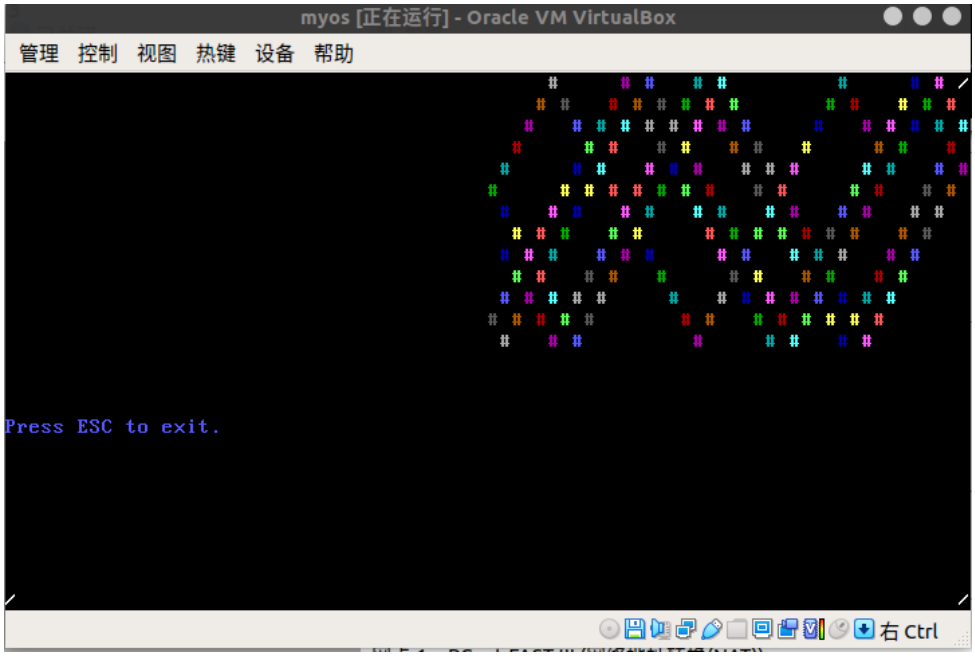
首先是用户程序1

点击 esc，进入用户程序4

点击 esc，进入用户程序3

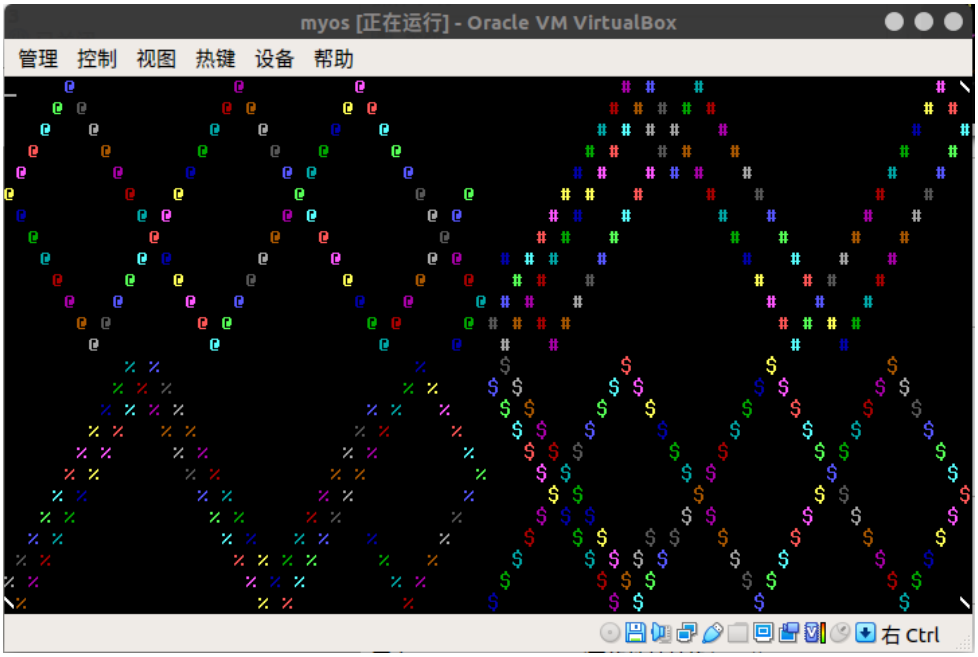


点击 esc，进入用户程序2

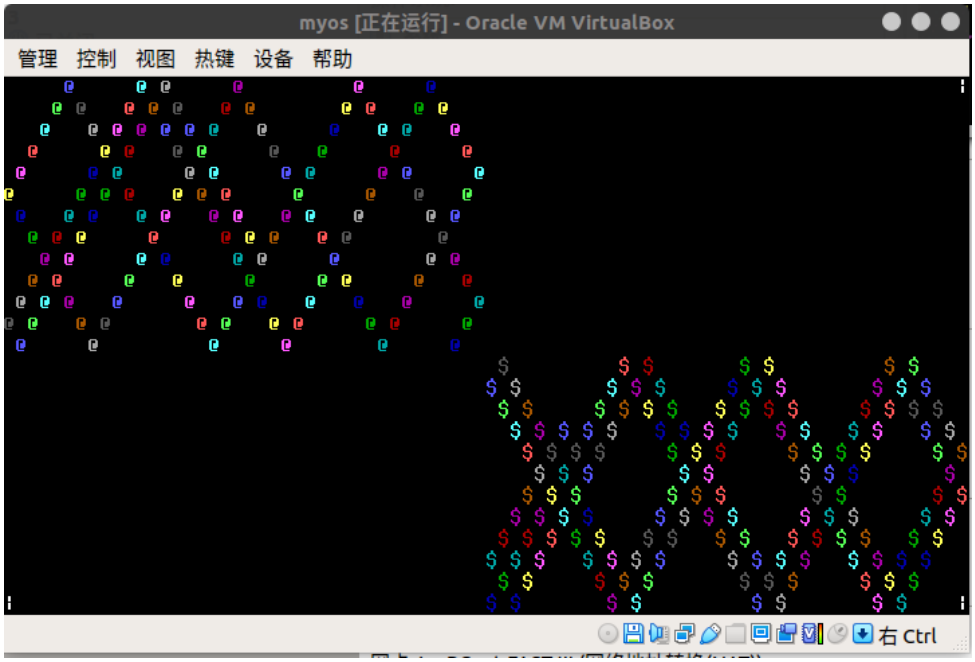


- 测试 runall 指令，并行运行

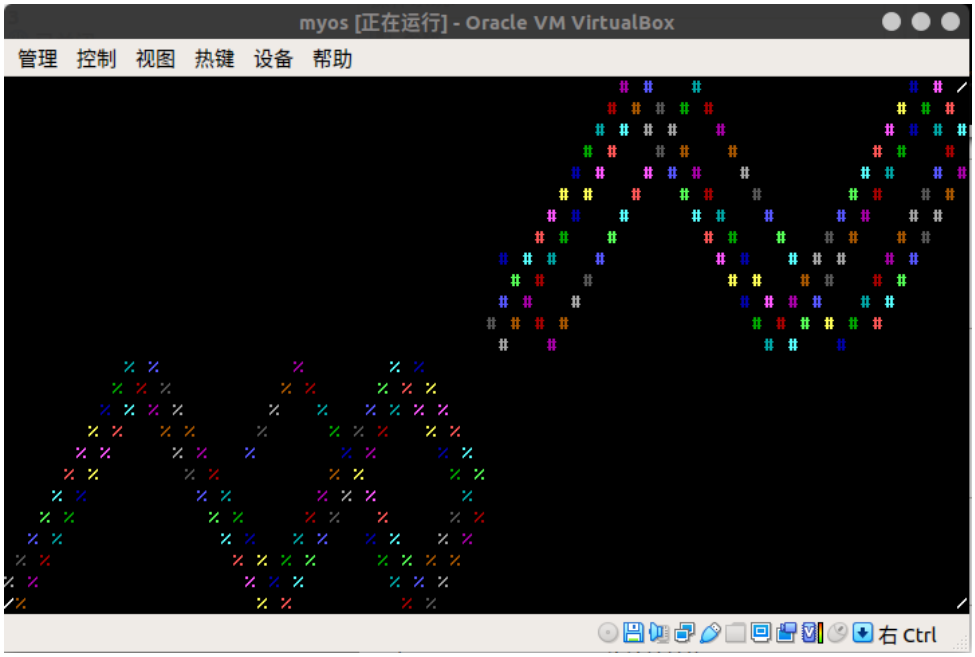
runall 1234 : 可以看到4个程序同时运行并且风火轮也在转动



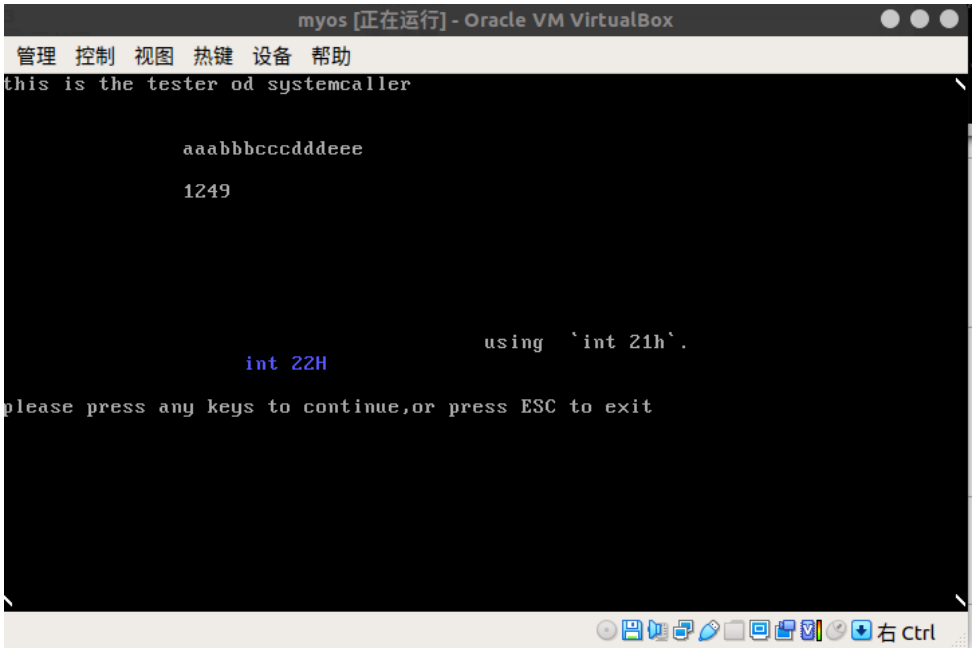
`runall 14`：可以看到程序1（左上角）和程序4（右下角）同时运行并且风火轮也在转动



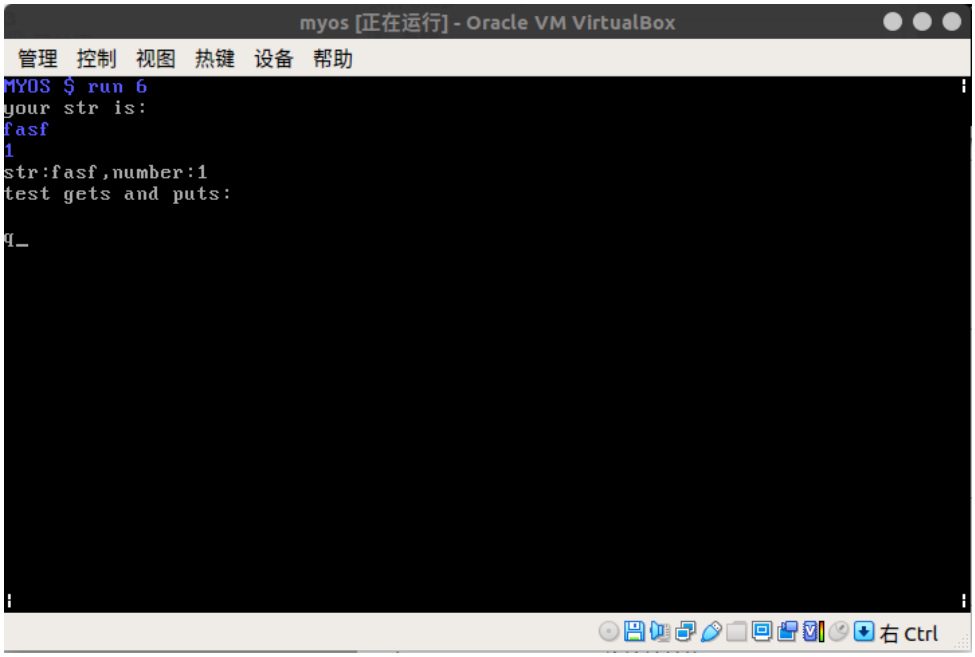
`runall 23`：可以看到程序2（右上角）和程序3（左下下角）同时运行并且风火轮也在转动



- run 5 测试实验五的中断测试程序，效果与原来相同



- 测试实验五中的c程序，效果与实验五相同



- 输入 shutdown 关机

5.实验体会

本次实验六相较于之前的实验工程量是挺大的。本次实验实现了二状态进程模型，进程用于运行态和就绪状态。

虽然在实验五中我已经初步实现了 save 和 restart，但是本次实验六仍然需要修改这两个过程。而且差别还是挺大的，由于实验五没有发送进程的转换，所以没有暴露出两个函数的问题，本次实验就暴露出来了，譬如 ds，es 段寄存器的问题，由于在每个用户程序下这几个寄存器的值是不一样的，因此我需要使用 cs 来重新初始化才能正确运行。

关于进程调度的问题，实验六的思路还是比较清晰的，但是实现起来是比较困难的。因为实验中涉及许多的细节问题，每一个细节都需要我仔细的考虑。在进程切换、创建进程这类过程中，需要频繁地保护寄存器、恢复寄存器，只要出现一点小错误就容易导致程序运行不了，使得原有用户程序失效。

最让我印象深刻的寄存器保护就是 sp 寄存器，因为我需要使用栈来保护，但是保护的寄存器中又有 sp 寄存器，因此该寄存器的保护也是调试了很久。

总的来说，从开学到现在，操作系统的实验已经做到了现在的实验六，文件的数目也从1个变到现在数不清的数目，虽然困难重重，但是操作系统的原理，以及实现的成就感也还是让我受益匪浅。

6.参考资料

- 操作系统——2.1-2进程的状态与转换：https://blog.csdn.net/qq_47664398/article/details/106070484
- 一个操作系统的实现——进程：https://blog.csdn.net/fukai555/article/details/41625619