

操作系统-实验二

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、了解监控程序执行用户程序的主要工作
- 2、了解一种用户程序的格式与运行要求
- 3、加深对监控程序概念的理解
- 4、掌握加载用户程序方法
- 5、掌握几个BIOS调用和简单的磁盘空间管理

2.实验要求

- 1、知道引导扇区程序实现用户程序加载的意义
- 2、掌握COM/BIN等一种可执行的用户程序格式与运行要求
- 3、将自己实验一的引导扇区程序修改为3-4个不同版本的COM格式程序，每个程序缩小显示区域，在屏幕特定区域显示，用以测试监控程序，在 1.44MB 软驱映像中存储这些程序。
- 4、重写 1.44MB 软驱引导程序，利用BIOS调用，实现一个能执行COM格式用户程序的监控程序。
- 5、设计一种简单命令，实现用命令交互执行在 1.44MB 软驱映像中存储几个用户程序。
- 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) 将自己实验一的引导扇区程序修改为一个的COM格式程序，程序缩小显示区域，在屏幕第一个1/4区域显示，显示一些信息后，程序会结束退出，可以在DOS中运行。在 1.44MB 软驱映像中制定一个或多个扇区，存储这个用户程序a。

相似地、将自己实验一的引导扇区程序修改为第二、第三、第四个的COM格式程序，程序缩小显示区域，在屏幕第二、第三、第四个1/4区域显示，在 1.44MB 软驱映像中制定一个或多个扇区，存储用户程序b、用户程序c、用户程序d。

(2) 重写 1.44MB 软驱引导程序，利用BIOS调用，实现一个能执行COM格式用户程序的监控程序。程序可以按操作选择，执行一个或几个用户程序。解决加载用户程序和返回监控程序的问题，执行完一个用户程序后，可以执行下一个。

(3)设计一种命令，可以在一个命令中指定某种顺序执行若干个用户程序。可以反复接受命令。

(4)在映像盘上，设计一个表格，记录盘上有几个用户程序，放在那个位置等等信息，如果可以，让监控程序显示出表格信息。

(5)拓展自己的软件项目管理目录，管理实验项目相关文档

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) github

开源代码托管平台,用于存储管理编写的代码

3) 实验原理

1. BIOS 调用

BIOS是英文"Basic Input Output System"的缩略语, 直译过来后中文名称就是"基本输入输出系统"。其实, 它是一组固化到计算机内主板上一个ROM芯片上的程序, 它保存着计算机最重要的基本输入输出的程序、系统设置信息、开机后自检程序和系统自启动程序。其主要功能是为计算机提供最底层的、最直接的硬件设置和控制。


BIOS中断服务程序是微机系统软、硬件之间的一个可编程接口, 用于程序软件功能与微机硬件实现的衔接。DOS/Windows操作系统对软、硬盘、光驱与键盘、显示器等外围设备的管理即建立在系统BIOS的基础上。程序员也可以通过 对INT 5、INT 13等终端的访问直接调用BIOS终端例程。

2.常用BIOS调用

中断号	功能号	功能
10H	06H	插入空行上滚显示页窗口
10H	0EH	以电传方式显示单个字符
10H	13H	显示字符串
13H	00H	复位磁盘系统
13H	02H	读扇区
16H	00H	读下一个按键

4)实验步骤

0.总体实验拆分

- a)首先设计4个用户程序,功能与实验一的引导程序相似,分别在屏幕左上,左下,右上,右下方显示运动反射字符
- b)设计引导监控程序,将监控程序存入内存,并转到监控程序
- c)编译并整合各个程序,合并成文件,然后开机实验

各个程序名,以及其功能

程序类别	文件名	功能
引导程序		加载监控程序 and 用户程序，然后跳转到用户程序执行。
用户程序1		在左上角动态反射输出字符。
用户程序2		在右上角动态反射输出字符。
用户程序3		在左下角动态反射输出字符。
用户程序4		在右下角动态反射输出字符。
列表程序		用于存放打印用户程序信息

1.设计四个用户程序

因为四个程序非常的相似,因此只需要修改一下一些定义和代码就可以分别获得

程序名	功能	反射字符
	在左上角动态反射输出字符。	@
	在右上角动态反射输出字符。	#
	在左下角动态反射输出字符。	\$
	在右下角动态反射输出字符。	%

具体的反射代码由于在实验一中已经做过,在这里就不再赘述

由于子程序需要在接收到键盘后退回引导监控程序,所以在每次循环中新增代码

```
1 | mov ah, 01h
2 | int 16h
3 | cmp al, 27      ;是否按下ESC
4 | je QuitUsrProg  ;若按下ESC，退出用户程序
5 |
6 | QuitUsrProg:
7 | retf            ;退出用户程序
```

`int 16h` 是捕获键盘输入并放在 `al` 寄存器中,功能号 `ah=01h` ,不阻塞

因此比较寄存器与27(`ESC`),然后跳转回引导程序的地址

代码编写完成后,对代码进行编译

```

1 | nasm topleft.asm -o topleft.com
2 | nasm topright.asm -o topright.com
3 | nasm bottomleft.asm -o bottomleft.com
4 | nasm bottomright.asm -o bottomright.com

```

还有一个特别之处在于清屏函数,需要在一开始调用,因为只有一个显存,调用程序后,之前显示的东西还存在,因此需要清屏

```

1 | ClearScreen:      ;函数: 清屏
2 |     pusha
3 |     mov ax, 0003h
4 |     int 10h      ;中断调用, 清屏
5 |     popa
6 |     ret

```

使用 int 10h 的BIOS中断来进行

函数调用方式

```

1 | call ClearScreen

```

2.设计引导监控程序

引导监控程序名是 booter.asm

引导监控程序要实现以下功能:1.显示个人信息 2.感应键盘输入 3.将用户代码放进内存并跳转

a)显示个人信息

与实验一不一样,学习使用BIOS中断功能将字符串打印出来,由于字符串打印的代码经常使用,因此我将定义为宏

```

1 | %macro PRINT 4
2 |     pusha      ;保护现场
3 |     mov ax, cs ;置其他段寄存器值与CS相同
4 |     mov ds, ax ;数据段
5 |     mov bp, %1 ;BP=当前串的偏移地址
6 |     mov ax, ds ;ES:BP = 串地址
7 |     mov es, ax ;置ES=DS
8 |     mov cx, %2 ;CX = 串长 (=9)
9 |     mov ax, 1301h ;AH = 13h (功能号)、AL = 01h (光标置于串尾)
10 |    mov bx, 0009h ;页号为0(BH = 0) 黑底白字(BL = 07h)
11 |    mov dh, %3 ;行号=0
12 |    mov dl, %4 ;列号=0
13 |    int 10h ;BIOS的10h功能: 显示一行字符
14 |    popa      ;恢复现场
15 | %endmacro

```

该宏定义函数传入四个参数,字符串偏移地址,字符串长度,打印行号,打印列号

显示例子

```

1  mov bp, Message      ; BP=当前串的偏移地址
2  mov ax, ds            ; ES:BP = 串地址
3  mov es, ax            ; 置ES=DS
4  mov cx, msglen        ; CX = 串长 (=9)
5  mov ax, 1301h         ; AH = 13h (功能号)、AL = 01h (光标置于串尾)
6  mov bx, 0007h         ; 页号为0(BH = 0) 黑底白字(BL = 07h)
7  mov dh, 0             ; 行号=0
8  mov dl, 0             ; 列号=0
9  int 10h              ; BIOS的10h功能: 显示一行字符
10 ;要显示的字符串储存在内存中:
11     Message db 'Hello, MyOs is loading user program A.COM...'
12     msglen equ ($-Message)

```

b)感应键盘输入

```

1  mov ah, 00h
2  int 16h
3  cmp al, 'a'; 按下1

```

调用BIOS中断, int 16H,功能和 00h,阻塞,必须等待输入后才进行后续程序,与上文 01h 不一样

c)把软盘上的用户程序代码放进内存中,由于该代码也会重用多次,因此也使用宏

```

1  %macro LOADPRO 3      ;加载内存函数
2  pusha                ;保护现场
3  mov ax,cs            ;段地址;存放数据的内存基地址
4  mov es,ax            ;设置段地址 (不能直接mov es,段地址)
5  mov bx,%1            ;偏移地址;存放数据的内存偏移地址1
6  mov ah,2             ;功能号
7  mov al,%2            ;扇区数2
8  mov dl,0             ;驱动器号;软盘为0, 硬盘和U盘为80H
9  mov dh,0             ;磁头号;起始编号为0
10 mov ch,0             ;柱面号;起始编号为0
11 mov cl,%3            ;起始扇区号;起始编号为3
12 int 13H              ;调用读磁盘BIOS的13h功能
13 popa                 ;恢复现场
14 %endmacro

```

使用了 int 13H 功能号为2的BIOS调用,将指定地址代码放进内存,函数调用方式

```

1  LOADPRO offset_program1,1,2
2  ;放进内存
3  jmp offset_program1 ;跳转指令

```

以上代码的功能是,从2号扇区开始读2个扇区的大小(1024字节),将其放入内存地址 offset_program1 的位置。

d)由于是主引导程序,最后需要有记录

```

1  times 510-($-$$) db 0
2  db 0x55,0xaa

```

3.设计一个存储显示用户程序信息子程序

该程序名为 list.asm

程序里主要存放者用户的信息

```
1 title db 'name head cylinder start_sector size'
2 titlelen equ ($-title)
3 pro1 db 'pro1 0 0 1 512'
4 pro1len equ ($-pro1)
5 pro2 db 'pro2 0 0 2 512'
6 pro2len equ ($-pro2)
7 pro3 db 'pro3 0 0 3 512'
8 pro3len equ ($-pro3)
9 pro4 db 'pro4 0 0 4 512'
```

然后跳转到该程序后打印出来

```
1 PRINT title, titlelen, 0, 0
2 PRINT pro1, pro1len, 1, 0
3 PRINT pro2, pro2len, 2, 0
4 PRINT pro3, pro3len, 3, 0
5 PRINT pro4, pro4len, 4, 0
```

当键盘输入 `esc` 则返回

```
1 mov ah, 0; Bochs: 0000:a173
2 int 16h
3 cmp al, 27; 按下esc
4 je back
5
6 back:
7 jmp offset_booter
```

4.整合全部程序

使用 `dd` :

```
1 dd if=booter.bin of=myosv2.img bs=512 count=1 2>/dev/null
2 dd if=topleft.com of=myosv2.img bs=512 seek=1 count=1 2>/dev/null
3 dd if=topright.com of=myosv2.img bs=512 seek=2 count=1 2>/dev/null
4 dd if=bottomleft.com of=myosv2.img bs=512 seek=3 count=1 2>/dev/null
5 dd if=bottomright.com of=myosv2.img bs=512 seek=4 count=1 2>/dev/null
6 dd if=list.com of=myosv2.img bs=512 seek=5 count=2 2>/dev/null
```

按照自定义的软盘存放位置一个个 `dd` 进最终的 `myosv2.img` 文件当中

文件名	占用扇区数	扇区offset
booter.asm	1	1
toleft.asm	1	2
topright.asm	1	3
bottomleft.asm	1	4
bottomright.asm	1	5
list.asm	2	6

在实验调试的时候每次改动代码都需要重新编译一遍代码并且整合，但是这样使用的时间实在是太多，而且时不时会搞错文件名，因此编写了一个bash脚本简化过程，这样就能自动化快速地整合

```

1  #!/bin/bash
2  rm -f myosv2.img
3
4  nasm -f bin booter.asm -o booter.bin
5  cd usrpro
6  nasm -f bin toleft.asm -o ../toleft.com
7  nasm -f bin topright.asm -o ../topright.com
8  nasm -f bin bottomleft.asm -o ../bottomleft.com
9  nasm -f bin bottomright.asm -o ../bottomright.com
10 nasm -f bin list.asm -o ../list.com
11 cd ..
12 dd if=booter.bin of=myosv2.img bs=512 count=1 2>/dev/null
13 dd if=toleft.com of=myosv2.img bs=512 seek=1 count=1 2>/dev/null
14 dd if=topright.com of=myosv2.img bs=512 seek=2 count=1 2>/dev/null
15 dd if=bottomleft.com of=myosv2.img bs=512 seek=3 count=1 2>/dev/null
16 dd if=bottomright.com of=myosv2.img bs=512 seek=4 count=1 2>/dev/null
17 dd if=list.com of=myosv2.img bs=512 seek=5 count=2 2>/dev/null
18 rm *.bin
19 rm *.com

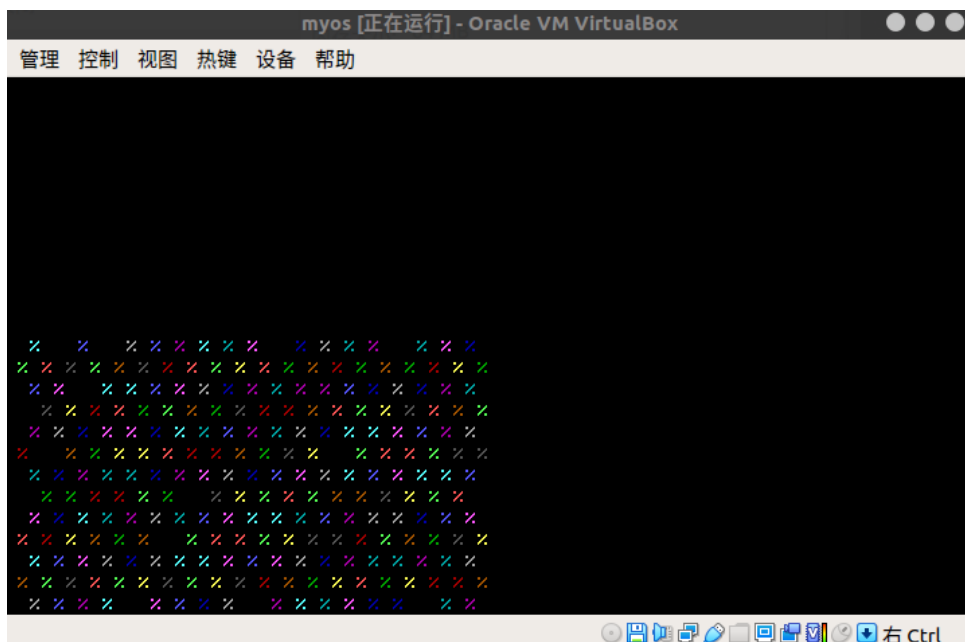
```

5.实验过程

1) 踩坑过程

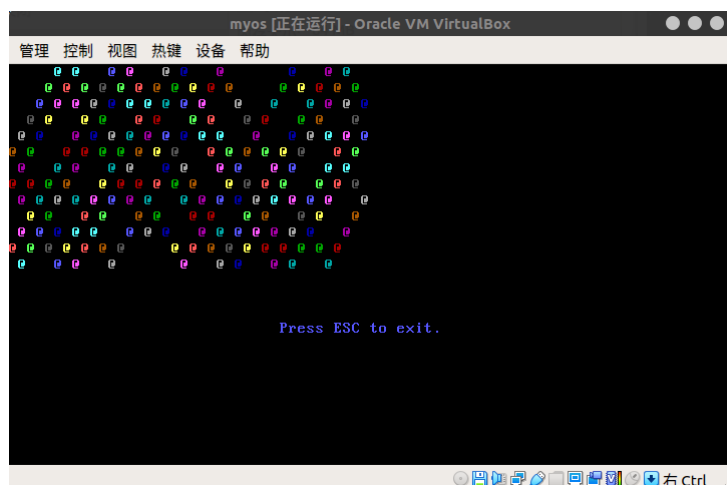
这次引导子程序的监控程序，卡住我最久的就是老师所说的坑 `org 100h`

起初的时候我在 `booter.asm` 使用 `jmp` 指令直接跳转到 `8100h`，但是发现跳转之后，反弹的字符可以运作，但是本来显示的字符串却不见了，而且返回不到监控程序



这个问题困扰了很久，之后老师在微信群里指引了一下我们怎么修改

然后我才知道这里的 `jmp` 需要 `jmp` 到 `800h:100h`，果然这样，我成功跳转了，并且字符串也是对的，但是我发现问题又来了，我按 `esc` 的时候，我回不到监控程序



我知道这里肯定是跳回来的指令出问题了，我原来跳回来的指令是 `jmp 7c00h`，正确的跳转应该是 `jmp 0:7c00h`。果然改正之后，跳转成功了。

这里背后的原因就是 `jmp` 指令会改变了 `cs` 寄存器的值，如果要回到监控程序，`cs` 的值要变回原来的 `0`（子程序是 `800h`）

原来这些错误都是因为寄存器改变引起的。

2.然后我尝试了一下 `call` 指令，这次我通过查资料了解 `call` 对寄存器的更改方式，很快就成功跳转跳转方式如下

```
1 | call 800h:100h    ;监控程序调用
2 |
3 | retf              ;返回监控程序
```

3.之后我发现不使用 `org 100h`，而使用 `org offset` 也可以成功跳转，跳转方式如下


```

1  jmp offset_program3 ;直接跳转到偏移地址
2
3  org offset_program3 ;子程序
4
5  jmp offset_booter   ;跳回监控程序方式

```

这样子跳转为什么会成功，是因为这样跳转是近跳转，cs的值是没有改变的，所以不需要 `org 100h`

4.同理call指令跳转也可以直接 `org offset`

```

1  call offset_program4 ; 跳转到偏移地址
2
3  org offset_program4 ;子程序
4
5  retf                  ;跳回监控程序方式
6

```

因此最终我代码里是使用四种不同的跳转方式跳到子程序的

跳转代码如下：

```

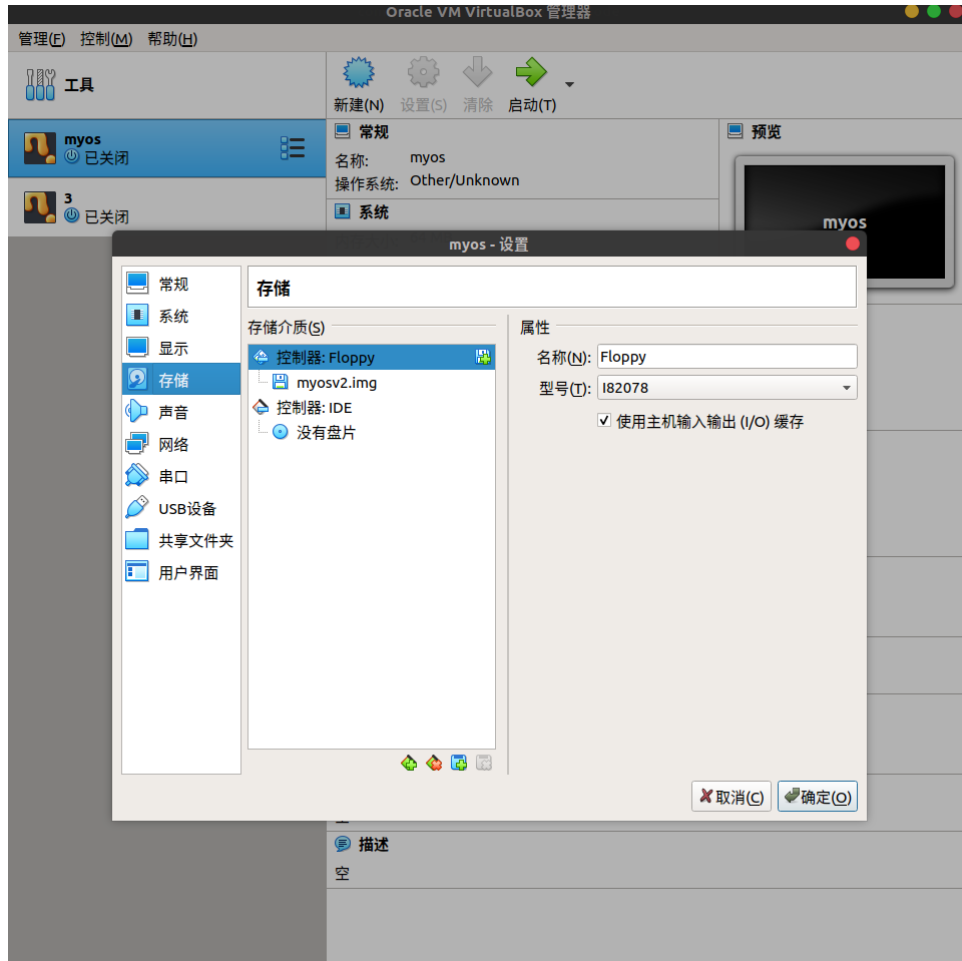
1  Keyboard:
2      mov ah, 0; Bochs: 0000:a173
3      int 16h
4      cmp al, 'a'; 按下a
5      jne noload1
6      mov ax, cs    ;置其他段寄存器值与CS相同
7      mov ds, ax    ;数据段
8      mov es, ax
9      mov ss, ax
10     LOADPRO offset_program1,1,2;加载program1到内存
11     jmp 800h:100h
12     ;jmp offset_program1 ;执行用户程序1
13 noload1:
14     cmp al, 'b'; 按下b
15     jne noload2
16     LOADPRO offset_program2,1,3;加载program2到内存
17     ;jmp offset_program2 ;执行用户程序2
18     call 800h:100h
19 noload2:
20     cmp al, 'c'; 按下c
21     jne noload3
22     LOADPRO offset_program3,1,4;加载program3到内存
23     jmp offset_program3 ;执行用户程序3
24     ;call 800h:100h
25 noload3:
26     cmp al, 'd'; 按下d
27     jne noload4
28     LOADPRO offset_program4,1,5;加载program3到内存
29     ;jmp offset_program4 ;执行用户程序4
30     call 0:8100h
31 noload4:
32     cmp al, 'l'; 按下6
33     jne noloadl
34     LOADPRO offset_list,2,6;加载list程序到内存
35     jmp offset_list

```

```
36 noloadl:
37     mov al,0
38     jmp start1
```

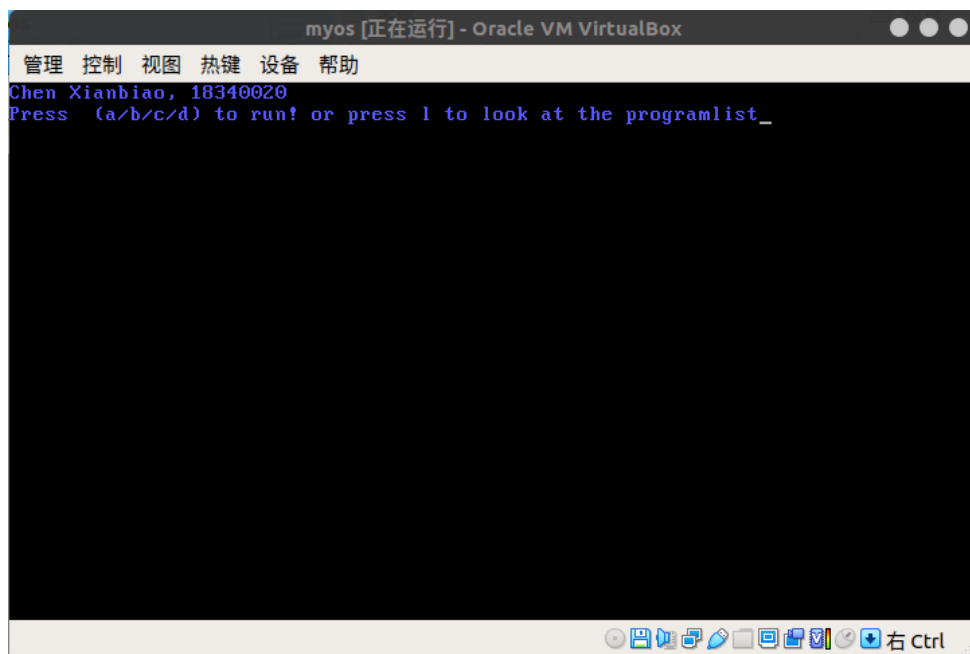
2) 最终成功的结果

1.加载 myosv2.img 进 VM Virtual Box 的xpc中

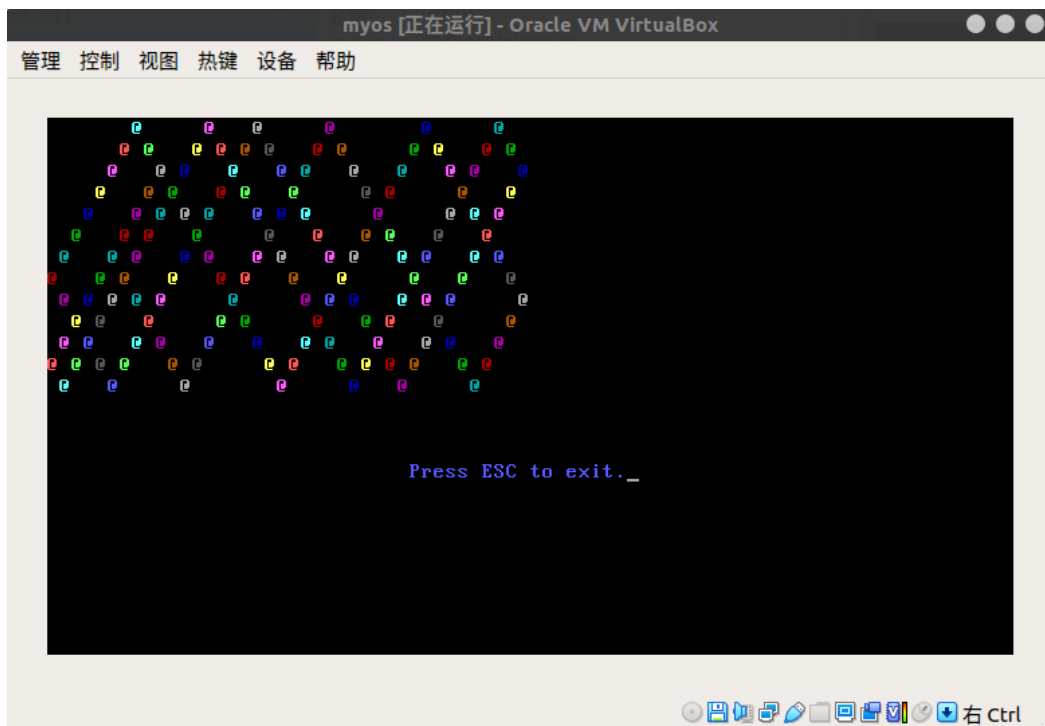


2.双击开启虚拟机

屏幕左上角显示出我的个人信息,并提示输入a/b/c/d来启动用户程序, 或者点击l来显示用户程序表

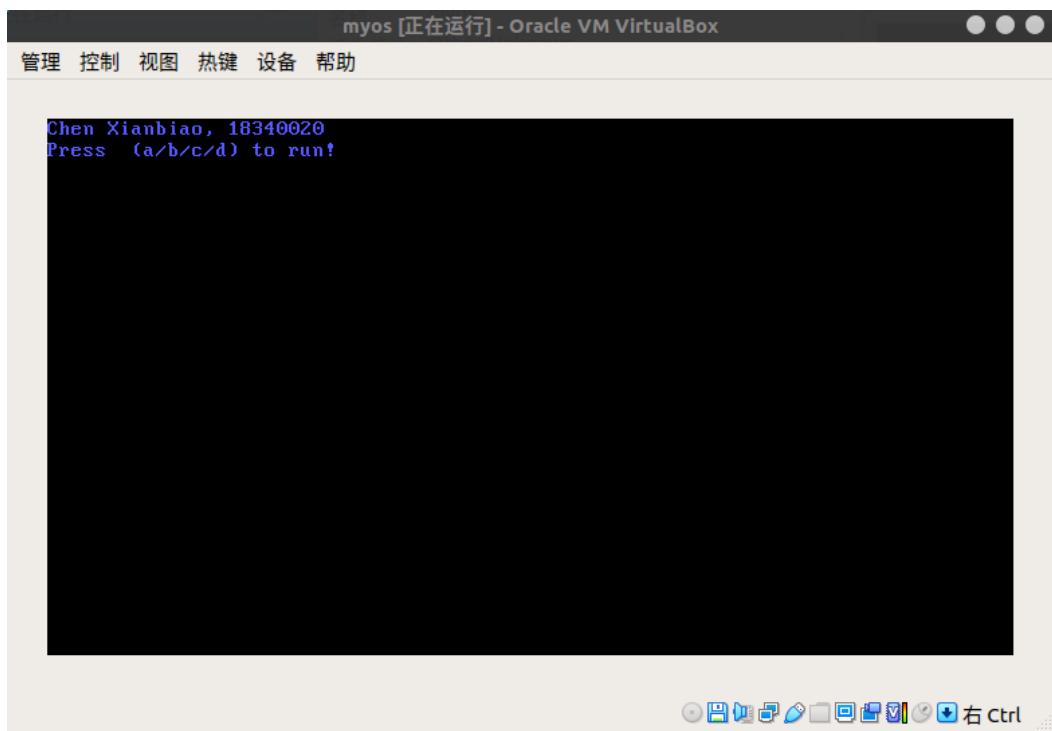


3.按下字母a,进入用户一程序,效果如下:

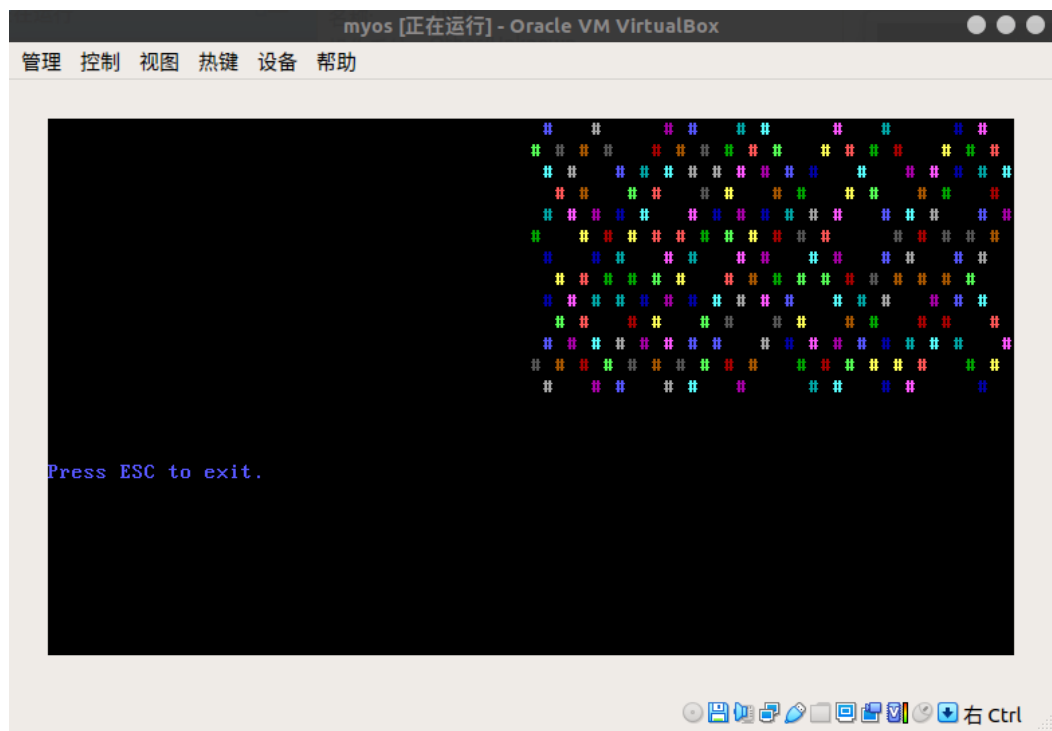


用户程序中提示按下exc来退出用户程序

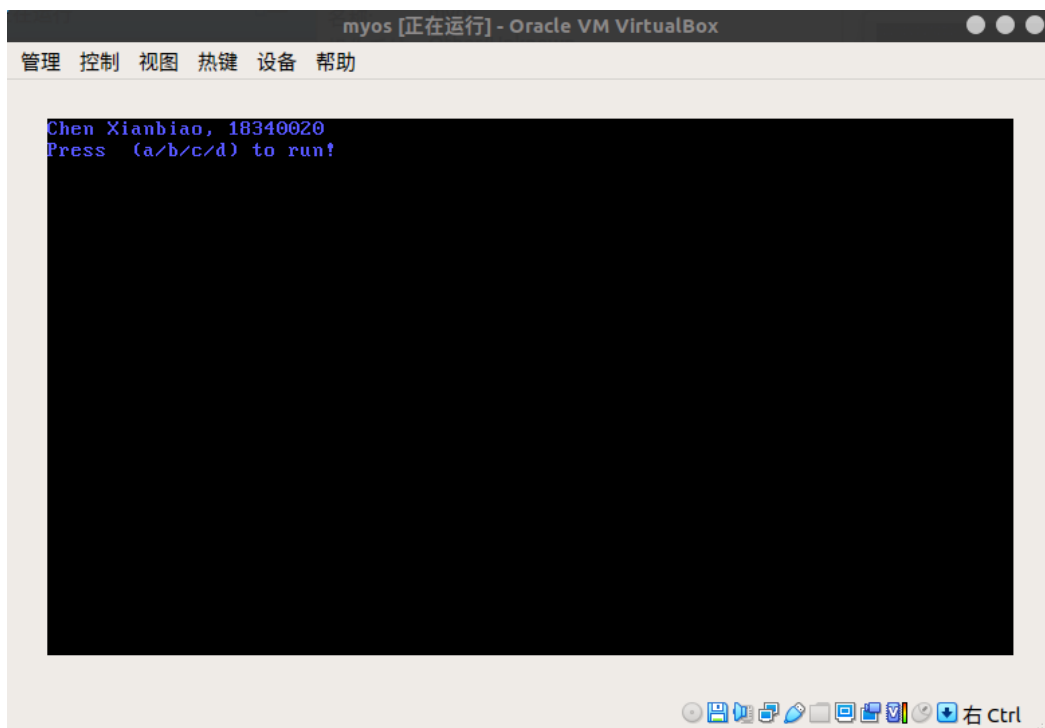
3.按下字母 **exc**,退回监控程序,效果如下:



4.按下b进入程序二,效果如下:



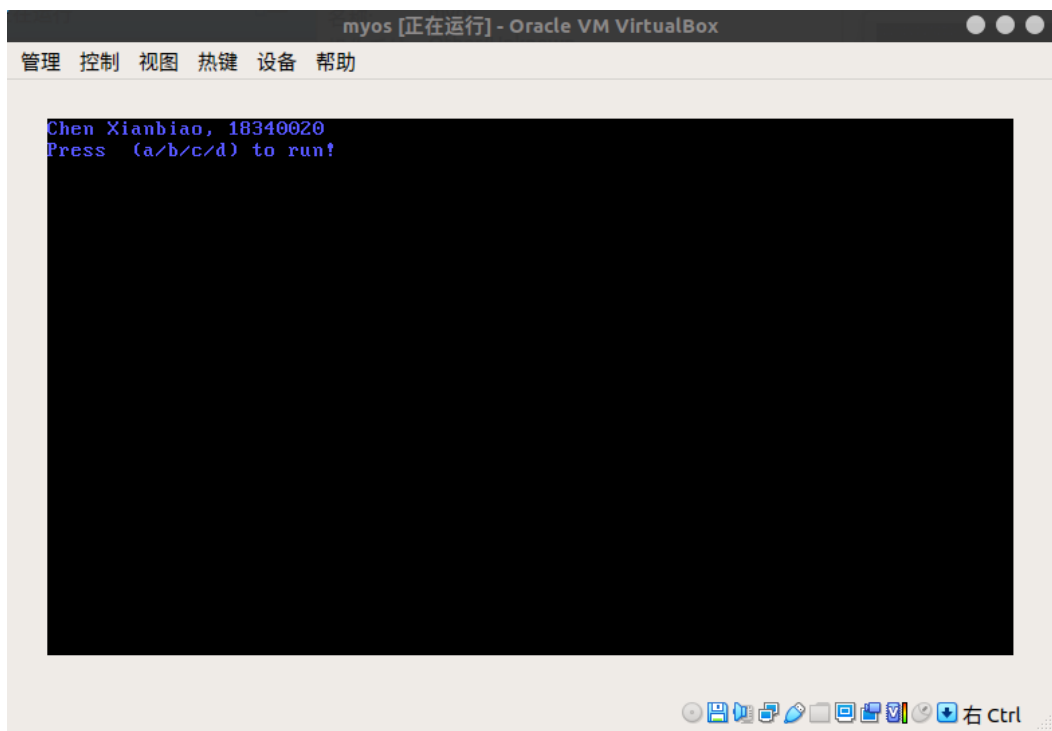
5.按下字母 **exc**, 退回监控程序,效果如下:



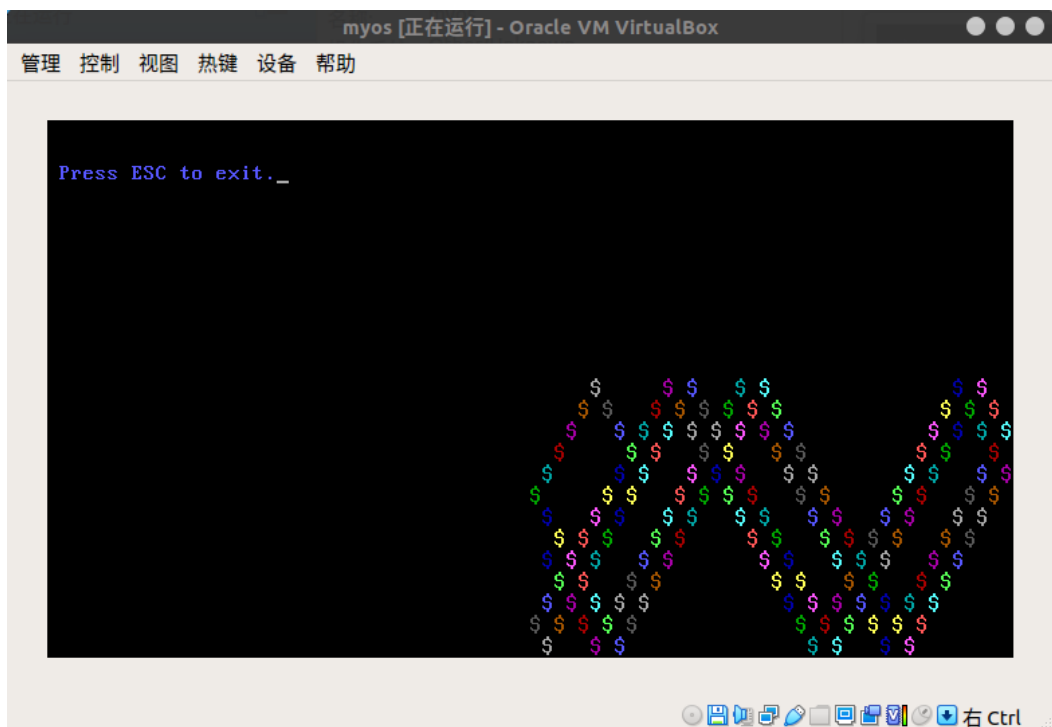
6.按下c,进入程序3,效果如下



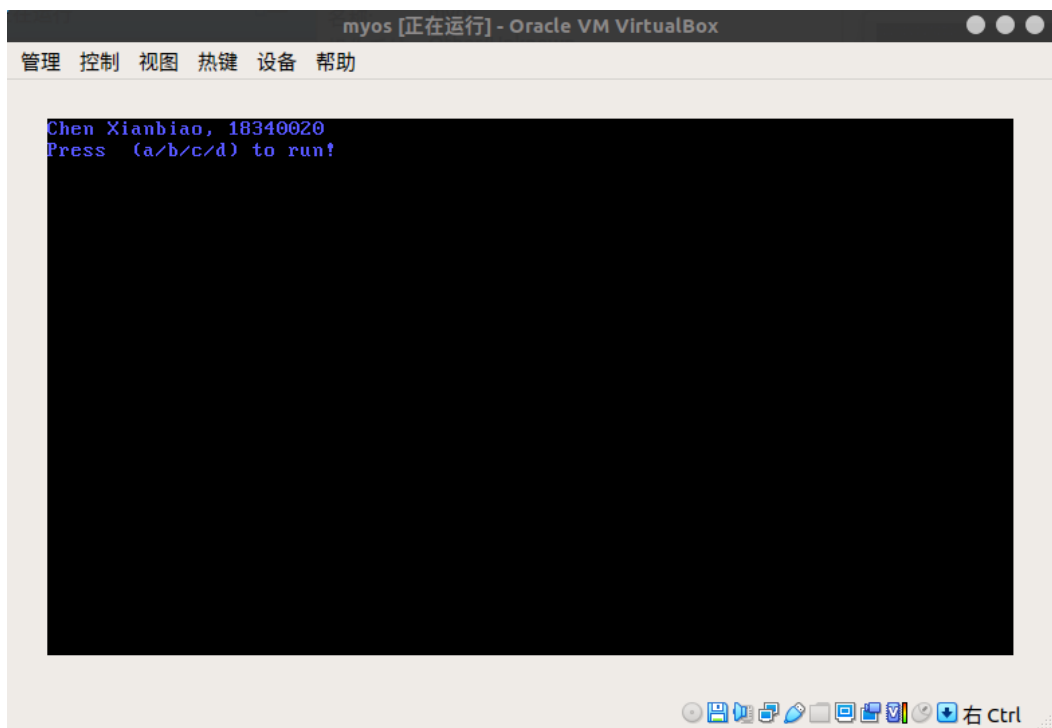
7.按下字母 **exc**, 退回监控程序,效果如下:



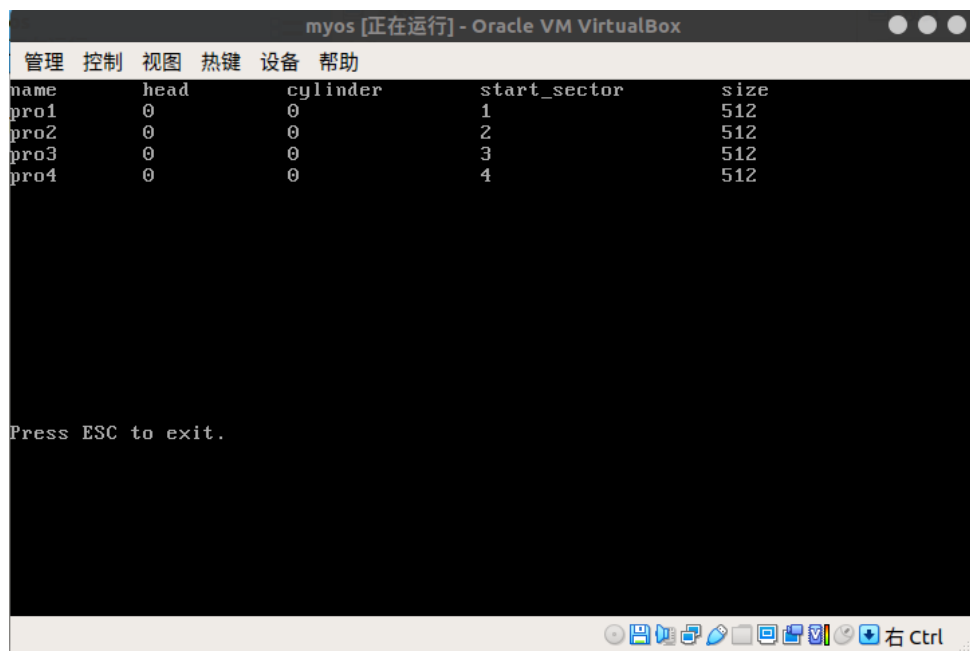
8.按下键盘d,进入程序4,效果如下:



9.按下字母 **exc**,退回监控程序,效果如下:



10.按下`l`查看用户程序表，效果如下



6.实验体会

总的来说，这次监控程序实验做起来还是比较折磨的。有了第一个引导程序实验的铺垫，我逐步地对汇编语言有了一定的了解，但是汇编里面的坑还是在这次实验里卡了我很久。譬如老师所说的 `org 100h`，这里隐含的坑就是汇编语言里跳转指令对段地址寄存器的修改问题。一开始我没有注意到这个问题，导致我不管怎么尝试总是跳转不到子程序里面。最后还是通过老师的一些指点，还有上网查资料最后顺利解决了这个问题。其实这只是一个很小的问题，但是这个问题却影响着整个程序的运行，可能这就是汇编语言的魅力吧。

这次实验我从 `windows` 的环境里转换的 `linux` 的环境下面，原因在于我第一次实验的时候发现使用软盘写入工具还有一些编译的时候，`windows` 需要打开很多不同的程序，导致效率不高，而在 `linux` 下，只需要终端指令就可以实现，省去了不必要的工作，还有一个就是使用 `bash` 可以快速地编译并合成软盘镜像，这样也能加快我的实验效率

懂得了如何将子程序的代码使用 BIOS 调用加载到内存里面，然后正确地跳转到子程序里面并能跳回监控程序。

还学会了如何使用宏定义 macro，对于一些重复次数比较多的代码，在汇编里面使用宏定义也让我效率高上不少，代码也更整洁

总体来说，第二次实验遇到的坑也还是不少的，不过经过不断地反复尝试，最后实现到实验目的的时候还是很有成就感的。

7.参考资料

- 1.保护模式下操作系统内核如何加载用户程序并运行:https://blog.csdn.net/qg_37375427/article/details/84996348
2. call、ret、retf 指令详解: <https://blog.csdn.net/longintchar/article/details/50989444>