

操作系统-实验三

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、加深理解操作系统内核概念
- 2、了解操作系统开发方法
- 3、掌握汇编语言与高级语言混合编程的方法
- 4、掌握独立内核的设计与加载方法
- 5、加强磁盘空间管理工作

2.实验要求

- 1、知道独立内核设计的需求
- 2、掌握一种 x86 汇编语言与一种C高级语言混合编程的规定和要求
- 3、设计一个程序，以汇编程序为主入口模块，调用一个C语言编写的函数处理汇编模块定义的数据，然后再由汇编模块完成屏幕输出数据，将程序生成COM格式程序，在DOS或虚拟环境运行。
- 4、汇编语言与高级语言混合编程的方法，重写和扩展实验二的的监控程序，从引导程序分离独立，生成一个COM格式程序的独立内核。
- 5、再设计新的引导程序，实现独立内核的加载引导，确保内核功能不比实验二的监控程序弱，展示原有功能或加强功能可以工作。
- 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) 寻找或认识一套匹配的汇编与c编译器组合。利用c编译器，将一个样板C程序进行编译，获得符号列表文档，分析全局变量、局部变量、变量初始化、函数调用、参数传递情况，确定一种匹配的汇编语言工具，在实验报告中描述这些工作。
- (2) 写一个汇编和c程序混合编程实例，展示你所用的这套组合环境的使用。汇编模块中定义一个字符串，调用C语言的函数，统计其中某个字符出现的次数（函数返回），汇编模块显示统计结果。执行程序可以在DOS中运行。
- (3) 重写实验二程序，实验二的的监控程序从引导程序分离独立，生成一个COM格式程序的独立内核，在 1.44MB 软盘映像中，保存到特定的几个扇区。利用汇编和c程序混合编程监控程序命令保留原有程序功能，如可以按操作选择，执行一个或几个用户程序、加载用户程序和返回监控程序；执行完一个用户程序后，可以执行下一个。
- (4) 利用汇编和c程序混合编程的优势，多用c语言扩展监控程序命令处理能力。
- (5) 重写引导程序，加载COM格式程序的独立内核。
- (6) 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) 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) c语言与汇编的链接方案

我使用的方式是 gcc + nasm + ld 的链接方式, 链接方式如下:

```
1 #gcc
2 gcc -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -mpreferred-stack-boundary=2 -lgcc -
  shared 源文件.c -o 目标文件.o
3
4 #nasm
5 nasm -f elf32 源文件.asm -o 目标文件.o
6
7 #ld
8 ld -m elf_i386 -N -Ttext 所在内存地址 --oformat binary 源文件1.o 源文件2.o ... -o 目标文件.bin
```

- 汇编头要加上

```
1 | BITS 16
```

- 汇编调用c函数

首先直接c语言文件中直接定义函数, 然后汇编程序使用前, 在程序上加上

```
1 extern 函数名;汇编头声明
2 ;调用使用
3 call dword 函数名
```

• c调用汇编函数

首先汇编头部加上

```
1 global 汇编函数名
```

c声明并调用

```
1 extern void 函数名()
```

若该函数需要传递参数，每个参数传递到栈当中，取出方式为；

```
1 uint16_t 函数名(uint8_t a, uint8_t b);
2 mov bp, sp
3 ;第一个参数
4 [bp]
5 ;第二个参数
6 [bp+4]
```

函数会先将b压栈，再将a压栈，栈地址是从高到低地址增长的。

(2) 总体程序架构

磁头号	扇区偏移	占用扇区数	功能
0	1	1	引导扇区
0	2	17	操作系统内核
1	1	2	userpro1
1	3	2	userpro2
1	5	2	userpro3
1	7	2	userpro4
1	9	2	用户程序表

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，

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。因此当内核已经占用到第18个扇区后，之后就轮到1号磁头1号扇区了

(3) 引导扇区的设计

上一次使用我将监控程序放进了引导程序的512字节里，当时是因为程序还不算大，所以就直接放进去了，但这次内核的程序比较大，所以引导扇区的作用是将内核加载进内存，然后跳转过去

```

1 | org 7c00h
2 | BITS 16
3 | offset_kernel equ 7E00h ;内核在内存中的地址

```

org 7c00h 引导程序内存偏移地址，BITS 16 16位实模式标志

```

1 | LoadOsKernel: ;加载操作系统内核
2 | mov ax,cs ;段地址;存放数据的内存基地址
3 | mov es,ax ;设置段地址 (不能直接mov es,段地址)
4 | mov bx, offset_kernel ;偏移地址;存放数据的内存偏移地址
5 | mov ah,2 ;功能号
6 | mov al,17 ;扇区数17个
7 | mov dl,0 ;驱动器号;软盘为0, 硬盘和U盘为80H
8 | mov dh,0 ;磁头号;起始编号为0
9 | mov ch,0 ;柱面号;起始编号为0
10 | mov cl,2 ;起始扇区号;起始编号为1
11 | int 13H ;调用读磁盘BIOS的13h功能
12 |
13 | jmp offset_kernel ;跳转到操作系统内核执行

```

加载内核然后跳转

```

1 | times 510-($-$$) db 0
2 | db 0x55,0xaa ;主引导记录标志

```

引导程序的标志

(4) 操作系统内核的设计

内核程序结构：

程序名	代码形式	作用
kernel.asm	ASM	内核的入口，调用c中的函数
kernel_a.asm	ASM	包含一些显示打印，IO借口，扇区加载的函数
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

我设计的内核中最主要的部分是cmd () 函数，进入内核后就会进入该函数的一个无限循环当中，一直等待接受指令，根据指令做一些操作

- kernel.asm

作用是初始界面，等待用户按下回车键进入

```

1 | global _start ;让c调用汇编的方式
2 | _start:
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 es,ax ;
6 | call dword startos

```

```

7
8 Keyboard:
9     mov ah, 0
10    int 16h
11    cmp al, 0dh ; 按下回车
12    jne Keyboard ; 无效按键, 重新等待用户按键
13    call dword cmd ; 进入命令行界面
14    jmp Keyboard ; 无限循环

```

调用 int 16h 的中断感应回车, 若回车则call进入 cmd 函数

- 指令的接收方式

getch () 函数 (asm) c语言调用汇编, 使用 BIOS 中断来接受键盘

```

1 getch: ; 函数: 读取一个字符到tempc
2     mov ah, 0 ; 功能号
3     int 16h ; 读取字符, al=读到的字符
4     mov ah, 0 ; 为返回值做准备
5     retf

```

```

1 extern char getch();

```

readcmd 函数在 stdio.h 中实现

接受并显示键盘输入, 直到遇到回车键则返回

- 指令处理

从 readcmd 函数后可以获得一个带空格的字符串, 因此需要对该字符串进行处理

指令最重要的部分便是指令头, 因此设计了一个 getFirstWord 函数获取指令头

```

1 void getFirstWord(const char* str, char* buf) // 获取指令头
2 {
3     int i = 0;
4     while(str[i] && str[i] == ' ')
5     {
6         i++;
7     }
8     int j=0;
9     while(str[i] && str[i] != ' ') {
10         buf[j] = str[i];
11         i++;
12         j++;
13     }
14     buf[j] = '\0'; // 字符串必须以空字符结尾
15 }

```

相对应的还有获取指令后部分的函数

```

1 void getbackWord(const char* str, char* buf)
2 {
3     buf[0] = '\0';
4     int i = 0;
5     while(str[i] && str[i] == ' ') {
6         i++;

```

```

7   }
8   while(str[i] && str[i] != ' '){
9       i++;
10  }
11  while(str[i] && str[i] == ' '){
12      i++;
13  }
14  int j = 0;
15  while(str[i]){
16      buf[j++] = str[i++];
17  }
18  buf[j] = '\0'; // 字符串必须以空字符结尾
19  }

```

- 字符串基础处理函数

由于不能调用 `string` 库，所以一些基本的字符串处理需要自己编写

字符串长度

```

1  int strlen(char *str) {
2      int count = 0;
3      while (str[count++] != '\0');
4      return count - 1;
5  }

```

字符串比较

```

1  int strcmp(const char* str1, const char* str2) {
2      int i = 0;
3      while (1) {
4          if(str1[i] == '\0' || str2[i] == '\0') { break; }
5          if(str1[i] != str2[i]) { break; }
6          ++i;
7      }
8      return str1[i] - str2[i];
9  }

```

以下是我已经实现的指令功能

指令名	功能
clear	清楚屏幕
ls	调用 <code>list</code> 程序显示用户程序的信息，需要按 <code>esc</code> 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
time	显示当前系统时间

- clear指令

该指令只需要调用汇编中断来进行

c中声明并调用

```
1 extern void clearScreen();
2
3 clearScreen();
```

汇编中编写

```
1 global clearScreen
2 clearScreen:      ;函数：清屏
3     push ax
4     mov ax,0003h
5     int 10h      ;中断调用，清屏
6     pop ax
7     retf
```

- ls指令

调用 `loadrun` 函数把list程序装入内存，然后跳转进去便可，该程序在实验二中已经实现过，所以在这里我只讲一下调用的方式

```
1 %macro LOADPRO 4      ;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,%3      ;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,%4      ;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;宏定义
```

```
1 global loadrun
2 loadrun:
3     pusha
4     mov bp,sp
5     add bp,20
6     LOADPRO offset_program1,2,[bp],[bp+4];宏定义，在实验2中实现过，作用是调用bios来装入内存
7     call dword offset_program1
8     popa
9     retf
```

该函数传递两个参数，一个是磁头号，一个是扇区号，list在前文提到过放在1号磁头，9号扇区

关键在于函数的参数是如何传递过去的，由于在汇编前面我加了 `pusha`，该指令会压栈，占用20，因此我取出 `sp` 后还会加20，`[bp]` 代表第一个参数head，`[bp+4]` 代表第二个参数sector

```
1 extern void loadrun(int head,int sector);
2 loadrun(1,9);
```

- run指令

run指令是这个实验里比较难的一部分，但是逻辑也很好理解

因为需要按照一定的顺序执行指令，因此我需要分析指令的后部分

```
1  else if(strcmp(first,commands[3])==0)//run
2      {
3
4          int flag=1;
5          for(int i=0;i<backlen;i++)
6          {
7              if(isnum124(back[i])!=1&&back[i]!=' '){flag=0;break;}
8          }
9          if(flag==1)
10         {
11             for(int i=0;i<backlen;i++)
12             {
13                 if(back[i]=='1')loadrun(user[0].head,user[0].sector);
14                 else if(back[i]=='2')loadrun(user[1].head,user[1].sector);
15                 else if(back[i]=='3')loadrun(user[2].head,user[2].sector);
16                 else if(back[i]=='4')loadrun(user[3].head,user[3].sector);
17             }
18             clearScreen();
19         }
20         else
21         {
22             char* inf="wrong program name";
23             print(inf);
24             NEWLINE;
25         }
26     }
```

首先先调用 `isnum124` 函数查出指令中是否全是字符1234，如果不全是则指令错误。

若全是字符1234，则遍历一般，当遇到数字时则调用 `loadrun` 函数来运行用户程序，`loadrun` 与ls中一样，这里就不再讲述

- help指令

help指令作用是给出指令的指示，因此只需要打印字符

```
1  void showHelp() {
2      char *help =
3          " help - show information about builtin commands\r\n"
4          " clear - clean the screem\r\n"
5          " ls - show the information of user's program\r\n"
6          " run <id> - run users' programmes example: `run 1234` \r\n"
7          " time - show the time\r\n"
8      ;
9      print(help);
10 }
```

- time指令

time指令作用是显示当前时间，实现方式是c调用汇编中的函数

在CMOS RAM中存放这当前时间，取出的方式是


```

1  mov al, offset
2  out 70h, al
3  in al, 71h

```

	秒	分	时	日	月	年
offset	0	2	4	7	8	9

因此分别些各自函数来取出时间

需要注意的是这些取出的数是以BCD码存放的，因此需要转换

cmd 中调用如下：

```

1  else if(strcmp(first,commands[4])==0)//time
2      {
3          putchar('2');
4          putchar('0');
5          print(itoa(bcd2dec(getDateYear()),10)); putchar(' ');
6          print(itoa(bcd2dec(getDateMonth()),10)); putchar(' ');
7          print(itoa(bcd2dec(getDateDay()),10)); putchar(' ');
8          print(itoa(bcd2dec(getDateHour()),10)); putchar(':');
9          print(itoa(bcd2dec(getDateMinute()),10)); putchar(':');
10         print(itoa(bcd2dec(getDateSecond()),10));
11         NEWLINE;
12     }

```

- cmd 整合

cmd 函数主要就是对指令的处理以及分配

该函数主要部分便是一个无限循环 while (1)

每次循环都接收键盘指令输入，然后对指令进行处理，然后输出对应指令的效果

(5) 4个用户程序以及list程序

这五个程序与实验2基本相同，因此在这里不再讲述

(6) 程序的编译与整合

与实验二一样，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

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 cd ..
15
16 cd kernel
17 nasm -f elf32 kernel.asm -o ../temp/kernel.o
18 nasm -f elf32 kernel_a.asm -o ../temp/kernel_a.o
19 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
20 ld -m elf_i386 -N -Ttext 0x7E00 --oformat binary ../temp/kernel.o ../temp/kernel_a.o
../temp/kernel_c.o -o ../temp/kernel.bin
21 cd ..
22 rm ../temp/*.o
23
24 dd if=../temp/booter.bin of=myosv3.img bs=512 count=1 2>/dev/null
25 dd if=../temp/kernel.com of=myosv3.img bs=512 seek=1 count=17 2>/dev/null
26 dd if=../temp/topleft.bin of=myosv3.img bs=512 seek=18 count=2 2>/dev/null
27 dd if=../temp/topright.bin of=myosv3.img bs=512 seek=20 count=2 2>/dev/null
28 dd if=../temp/bottomleft.bin of=myosv3.img bs=512 seek=22 count=2 2>/dev/null
29 dd if=../temp/bottomright.bin of=myosv3.img bs=512 seek=24 count=2 2>/dev/null
30 dd if=../temp/list.bin of=myosv3.img bs=512 seek=26 count=2 2>/dev/null
31 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

(1) 第一个坑也是最大的坑就是 `gcc+nasm` 的混合编译问题，`ppt` 上有一定的错误

`gcc` 指令中需要参数 `-m16` 而不是 `-m32`

`ld` 指令参数 `-m i386pe` 应该改成 `-m elf_i386`

```
ld: 无法于非 PE 输出文件 ../temp/kernel.bin 施行 PE 操作。
```

一开始进行混合汇编的时候，总是报出一些奇奇怪怪的错误，整了很久，幸好在微信群里同学们也遇到这些问题，最终问同学和老师就解决这个指令的问题

(2) 第二个问题就是在混合编译中，汇编的第一个函数名需要是 `_start` 并且必须 `global _start` 不然会出现错误

```
ld: 警告：无法找到项目符号 _start; 缺省为 00000000000007e00
```

(3) 第三个坑就是c函数调用问题

一开始我认为c中的函数想要调用就直接call便可以，但是由于c中的函数编译成汇编语言后，会把cs与ip出栈，因此我直接call指令调用该函数，就会出现错误

因为call指令只把 ip 入栈，因此就会从指令中跳不出来，一开始我只是调用 cmd 函数，不需要跳转出来，所以不会出现这个问题，但是当我调用子程序需要跳转出来的时候就卡死在那里了。



按下 esc 卡住

想要解决这个问题，可以自行入栈，然后 jmp 进入函数，也可以使用指令 call dword，该指令相当于把 cs, ip入栈，然后跳转。

(4) 关于c调用汇编字符串的问题

一开始听老师的讲法就是和函数调用一样，在汇编中global，然后在c中extern。但我发现我在c中 extern char* name 是不行的。这样子声明会导致打印的时候卡死，字符串的类型有问题。

当时我解决的办法就是因为我已经可以调用函数以及返回参数，因此我尝试调用函数然后将首地址当做返回参数，然后就成功获取该字符串。 *extern char* getzifuchuan();* 可以在 kernel_c.c 中看到该函数（被注释）

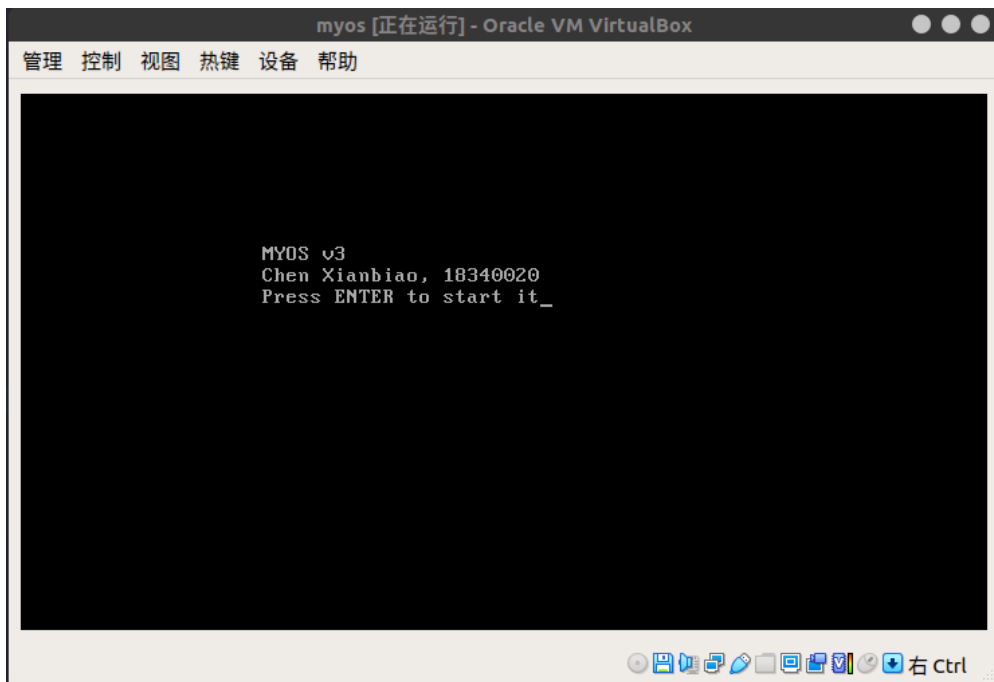
之后我发现当我这样声明 extern char name[] 也能成功返回字符串的首地址。实际操作在后面结果展示中解释。

(5) 参数保护问题

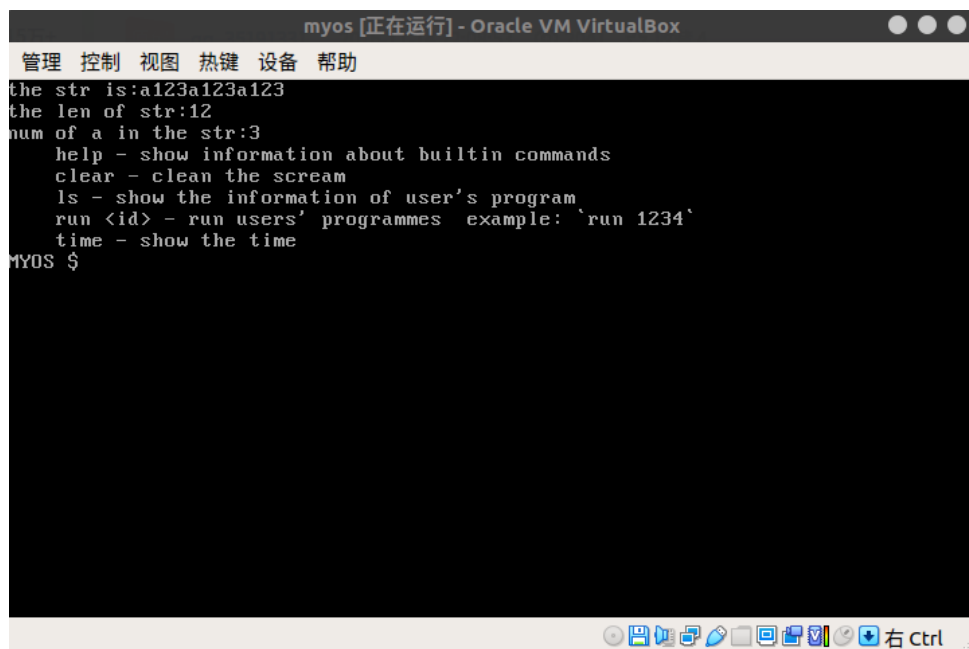
在调用汇编里的函数的时候，可能会改变某些寄存器的值，但是我在一开始编写函数的时候没有注意到这个问题，导致当我调用完这个函数之后，操作系统就运行不了了。后面细想才发现需要保护寄存器就是在函数开始的时候 pusha，函数退出是调用 popa

2) 实验结果展示

- 整合成软盘镜像后，加载进入 VM VirtualBox 然后运行



- 按下 ENTER 进入 cmd 命令行



注意：这里最前面三行是实现实验要求3，C语言编写的函数处理汇编模块定义的数据，然后再由汇编模块完成屏幕输出数据。

在汇编中的定义如下：

```
1 global name
2 DataArea:
3     name db 'a123a123a123'
4     times 16-($-name) db 0
```

在c中的定义如下

```
1 extern char name[16];
2 char *p=name;
3 print(p);
4 NEWLINE;
5 int len=strlen(p);//获取字符串长度
```

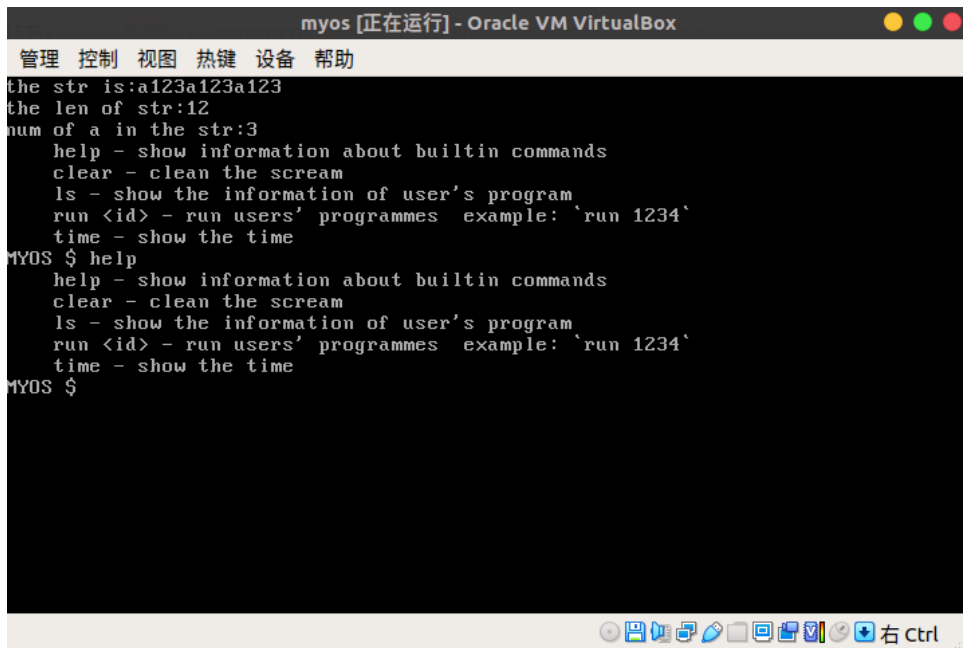
```

6 char* lll="the len of str: ";
7 print(lll);
8 char *q=itoa(len,10);将数字转为字符串
9 print(q);
10 NEWLINE;
11 int charnum=getnumofchar(p,'1');//获取该字符串中字符'1'的个数
12 q=itoa(charnum,10);
13 lll="num of a in the str: ";
14 print(lll);
15 print(q);//输出1的个数

```

之后第一次进入命令行会直接输出help，方便用户使用

- 键盘输入 `help`，回车运行，效果如下：

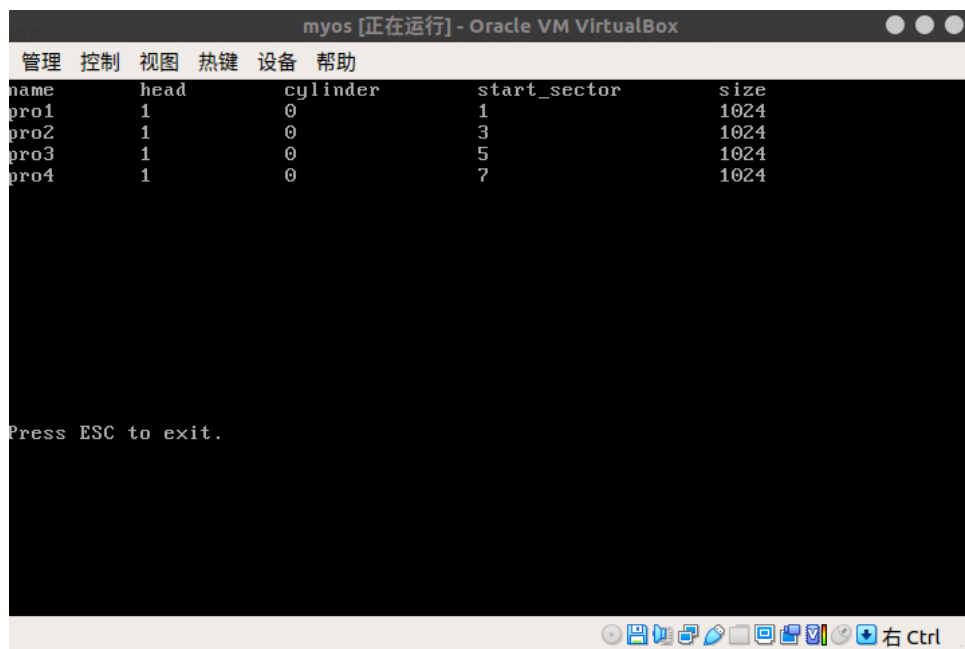


- 然后键盘输入 `clear`，回车运行，效果如下：



屏幕清屏

- 然后键盘输入 `ls`，进入 `list`



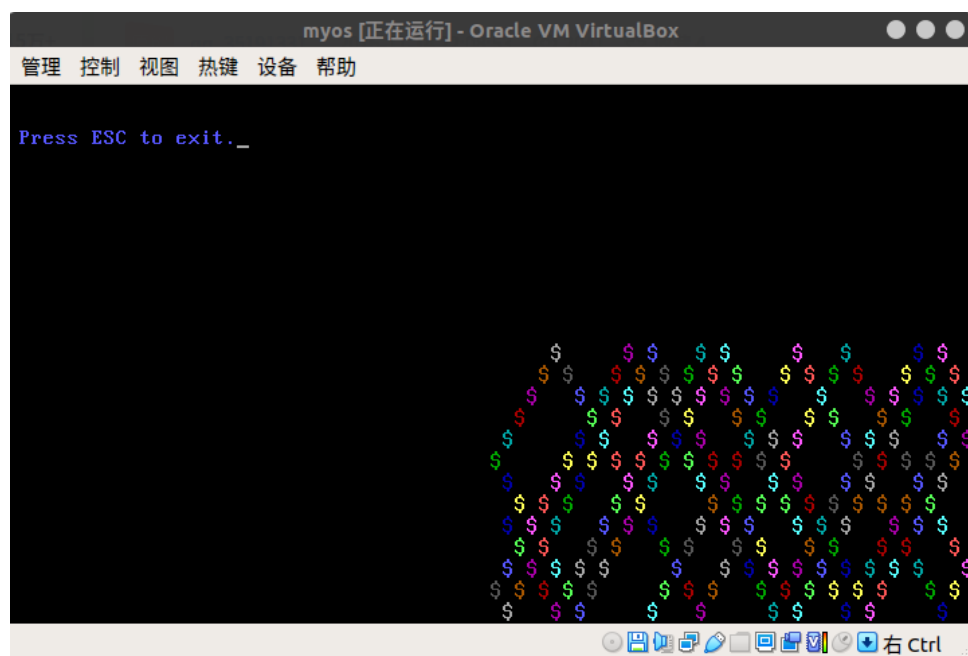
- 按 `esc`，退回 `cmd` 命令行



- 然后输入 `run 4213`，依次执行用户程序右下角，右上角，左上角，左下角



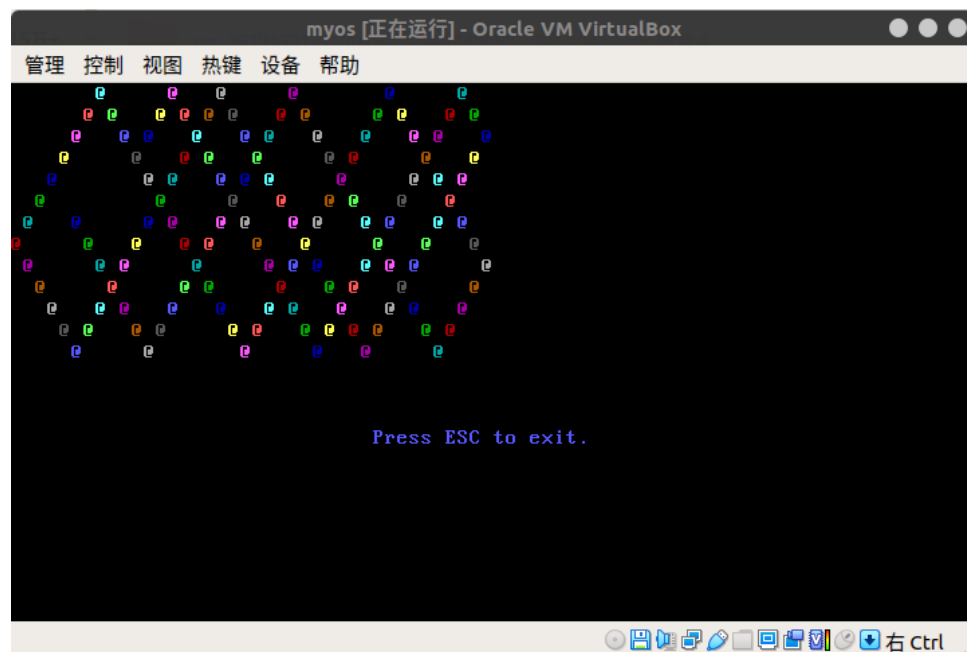
回车进入程序4，右下角：



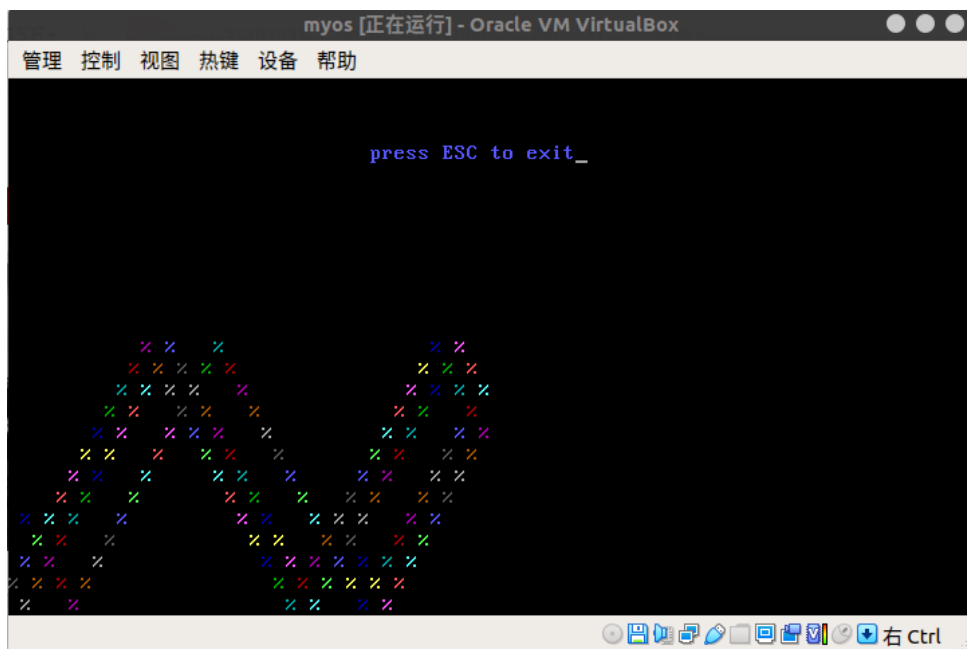
按下 `esc`，进入程序2，右上角：



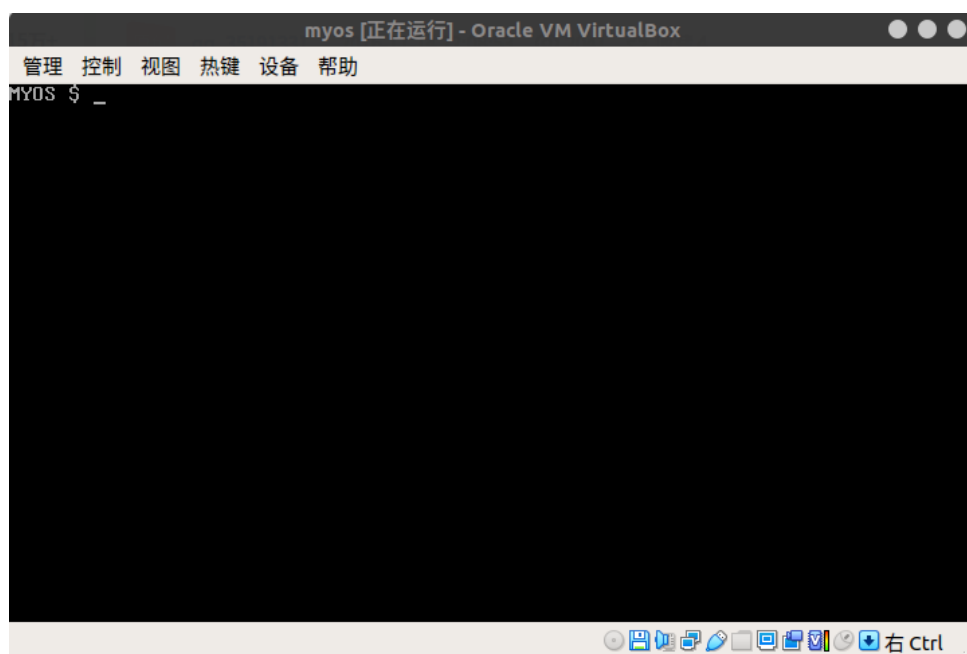
按下 `esc`，进入程序1，左上角：



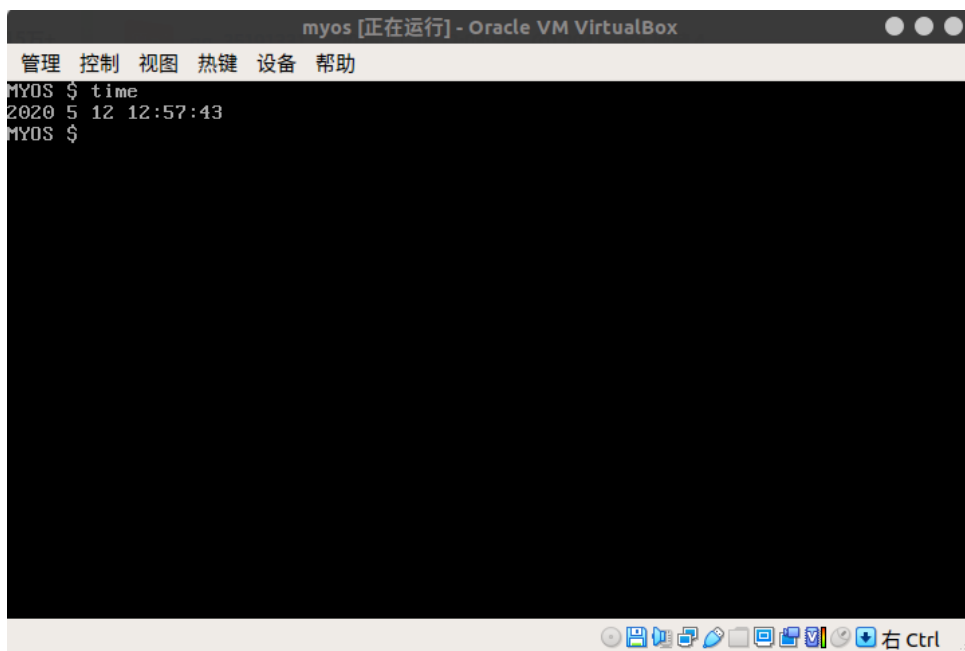
按下 `esc`，进入程序3，左下角：



最后按下 `esc`，退回命令行：



- 键盘输入 `time`，回车获取当前时间，效果如下：



- 若是输入错误命令，效果如下



5.实验体会

这次实验相比前两次实验用的时间多上不少，原因是成功找到汇编语言与c语言混合编译的正确打开方式实在是耗上不少时间。老师推荐的 `tcc+tasm` 因为我想要使用 `linux` 的环境，所以就不使用该混合方式了，因此我选择探讨 `nasm+gcc+ld` 的混合方式。为了混合，研究 `nasm` 和 `gcc` 编译参数、`ld` 的链接参数，要学习如何用 `nasm` 和 `gcc` 生成 16 位代码（或 16 位兼容代码），还要了解 `elf` 格式、`com` 格式、`bin` 格式的区别和它们各自的作用，要熟悉“源代码 - 对象文件 - 可执行文件”的流程。成功不报错混合后，还需要解决汇编与c函数参数互相调用的问题。

前期的整合尝试时间比起写c代码所用的时间要多上许多。当我成功搞清楚汇编与c混合的要求以及方式后，写代码就顺手多了，只要在汇编语言中实现好用户和程序的接口函数，IO函数，然后使用c语言调用这些函数，其他一些逻辑实现问题就全部使用c语言来实现。原本一些复杂的while，for循环在汇编里比较难以实现，现在只需要使用c语言来编写就简单多了。譬如我实现的run函数，本来在实验二中只能接收一个键盘输入，运行一个程序，而使用c实现，便很轻松地能根据输入指令按自己想要的顺序运行函数。

c中的比较困难的一个部分就是 `readcmd` 的编写，该函数需要很好地调用与汇编的接口的输入输出函数，譬如 `getchar` 获取键盘输入，然后就 `putchar` 到光标处，感应到回车便退出函数等,需要注意参数调用，参数传递的各种问题。

我还深入明白了寄存器保护的问题，在调用函数的时候寄存器一改变会导致程序无法进行，因此需要对寄存器进行压栈。

这次实验也我更加深入了解到软盘的结构，软盘结构其实老师在课堂上也有讲过，但这次实验当我超出18个扇区后需要换到第一磁道第一扇区，我才更加深刻地明白这个东西的意义在哪里。

总的来说，这次实验遇到的坑很多，但是我在解决这些困难的过程中学习到不少新知识。

6.参考资料

- 1.总结——gcc+nasm交叉编译在16位实模式互相引用的接口：<https://blog.csdn.net/laomd/article/details/80148121>
- 2.软盘结构及软盘数据的读取：<https://blog.csdn.net/smallmuou/article/details/6796867>