

操作系统-实验五

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、学习掌握PC系统的软中断指令
- 2、掌握操作系统内核对用户服务的系统调用程序设计方法
- 3、掌握C语言的库设计方法
- 4、掌握用户程序请求系统服务的方法

2.实验要求

- 1、了解PC系统的软中断指令的原理
- 2、掌握 x86 汇编语言软中断的响应处理编程方法
- 3、扩展实验四的内核程序，增加输入输出服务的系统调用。
- 4、C语言的库设计，实现 `putch()`、`getch()`、`printf()`、`scanf()` 等基本输入输出库过程。
- 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) 修改实验4的内核代码，先编写`save()`和`restart()`两个汇编过程，分别用于中断处理的现场保护和现场恢复，内核定义一个保护现场的数据结构，以后，处理程序的开头都调用`save()`保存中断现场，处理完后都用`restart()`恢复中断现场。
- (2) 内核增加 `int 20h`、`int 21h` 和 `int 22h` 软中断的处理程序，其中，`int 20h` 用于用户程序结束是返回内核准备接受命令的状态；`int 21h` 用于系统调用，并实现3-5个简单系统调用功能；`int 22h` 功能未定，先实现为屏幕某处显示 `int 22H`。
- (3) 保留无敌风火轮显示，取消触碰键盘显示OUCH!这样功能。
- (4) 进行C语言的库设计，实现 `putch()`、`getch()`、`gets()`、`puts()`、`printf()`、`scanf()` 等基本输入输出库过程，汇编产生 `libs.obj`。
- (5) 利用自己设计的C库 `libs.obj`，编写一个使用这些库函数的C语言用户程序，再编译,在与 `libs.obj` 一起链接，产生COM程序，增加内核命令执行这个程序。
- (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

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：

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
ouch.asm	ASM	封装了关于 ouch 键盘中断的代码
register.h	C	新增头文件，封装了一个包含所有寄存器的结构体和函数
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

以下是我已经实现的指令功能（新增了run 5 6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

(3) （新增） save()和restart()的编写

为保护中断前的寄存器，我首先在内核中构建了一个包含所有寄存器的结构体 r1 在 register.h

```
1 struct my_Register{
2     uint16_t ax; // 0
3     uint16_t cx; // 2
4     uint16_t dx; // 4
5     uint16_t bx; // 6
6     uint16_t sp; // 8
7     uint16_t bp; // 10
8     uint16_t si; // 12
9     uint16_t di; // 14
10    uint16_t ds; // 16
11    uint16_t es; // 18
12    uint16_t fs; // 20
13    uint16_t gs; // 22
14    uint16_t ss; // 24
15    uint16_t ip; // 26
```

```

16     uint16_t cs; // 28
17     uint16_t flags; // 30
18 };
19 struct my_Register r1;
20 struct my_Register* getr1()//取得结构体首地址的函数
21 {
22     return &r1;
23 }

```

构建一个结构体很简单，但是怎样才能保护到所有的寄存器呢。我使用的方法就是使用栈间接保存。

在进入中断函数的一开始，首先调用我定义的一个宏，将所有寄存器押进栈当中，这样可以防止寄存器的改变

```

1  %macro PUSHALL 0
2      push ss
3      push gs
4      push fs
5      push es
6      push ds
7      push di
8      push si
9      push bp
10     push sp
11     push bx
12     push dx
13     push cx
14     push ax
15 %endmacro

```

随后就可以调用 `save ()` 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

```

1  extern getr1
2  save:
3      pusha
4      mov bp, sp
5      add bp, 16+2      ; 参数首地址
6
7      call dword getr1  ; 获取结构体首地址
8      mov di, ax
9      mov ax, [bp]
10     mov [cs:di], ax
11     mov ax, [bp+2]      ; 最后压栈的ax，对应r1.ax
12     mov [cs:di+2], ax
13     mov ax, [bp+4]
14     mov [cs:di+4], ax
15     mov ax, [bp+6]
16     mov [cs:di+6], ax
17     mov ax, [bp+8]
18     mov [cs:di+8], ax
19     mov ax, [bp+10]
20     mov [cs:di+10], ax
21     mov ax, [bp+12]
22     mov [cs:di+12], ax

```

```

23     mov ax, [bp+14]
24     mov [cs:di+14], ax
25     mov ax, [bp+16]
26     mov [cs:di+16], ax
27     mov ax, [bp+18]
28     mov [cs:di+18], ax
29     mov ax, [bp+20]
30     mov [cs:di+20], ax
31     mov ax, [bp+22]
32     mov [cs:di+22], ax
33     mov ax, [bp+24]
34     mov [cs:di+24], ax
35     mov ax, [bp+26]           ;ip在栈中的位置
36     mov [cs:di+26], ax
37     mov ax, [bp+28]           ;cs在栈中的位置
38     mov [cs:di+28], ax
39     mov ax, [bp+30]           ;psw在栈中的位置
40     mov [cs:di+30], ax
41
42     popa
43     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。

```

1  %macro RESTART 0           ;宏：从PCB中恢复寄存器的值
2  call dword getr1
3  mov si, ax
4  mov ax, [cs:si+0]
5  mov cx, [cs:si+2]
6  mov dx, [cs:si+4]
7  mov bx, [cs:si+6]
8  mov sp, [cs:si+8]
9  mov bp, [cs:si+10]
10 mov di, [cs:si+14]
11 mov ds, [cs:si+16]
12 mov es, [cs:si+18]
13 mov fs, [cs:si+20]
14 mov gs, [cs:si+22]
15 mov ss, [cs:si+24]
16 add sp, 11*2               ;恢复正确的sp
17 push word[cs:si+30]        ;新进程flags
18 push word[cs:si+28]        ;新进程cs
19 push word[cs:si+26]        ;新进程ip
20 push word[cs:si+12]
21 pop si                     ;恢复si
22 %endmacro

```

(4) (新增) 系统调用的表

INT 21H :

功能号	输入参数	输出参数	作用
0	AH=0, es=串段址, dx =串首偏移	无	将 es:dx 位置的一个字符串中的小写变为大写
1	AH=1, es=串段址, dx =串首偏移	无	将 es:dx 位置的一个字符串中的大写变为小写
2	AH=2, es=串段址, dx =串首偏移	ax=整数	将 es:dx 位置的一个数字字符串转变对整数
3	AH=3, bx=整数, es=串段址, dx =串首偏移	无	将bx的数值转变对应的 es:dx 位置的一个数字字符串
4	AH=4, ch:行号cl:列号, dx =串首偏移	无	将 es:dx 位置的一个字符串显示在屏幕指定位置(ch:行号cl:列号)

INT 22H

功能号	输入参数	输出参数	作用
0	AH=0	无	屏幕某处显示 int 22H

(5) 系统调用的入口程序

由于系统调用的本质就是软中断的程序调用。因此我需要编写一个向量 21h 的入口程序，并且能够识别 ah 来进入不同的功能号不同的子程序。我实现的方式便是建一个向量表，根据ah的值的跳转进调用表上对应的函数。

注意：有一点便是ax的恢复问题，我会在踩坑过程中详细讲述

system_call 的实现在 kernel/kernel_a.asm

```
1  system_call:
2  PUSHALL
3  call save
4
5  push ds
6  push si
7  mov si,cs
8  mov ds,si
9  mov si,ax
10 shr si,8
11 add si,si
12 call [system_table+si]
13
14 mov cx,ax
15 push cx
```

```

16    call dword getr1
17    pop cx
18    mov si, ax
19    mov [cs:si+0], cx
20
21    pop si
22    pop ds
23
24    RESTART
25
26    iret      ;中断返回
27 system_table:
28    dw toupper_a, tolower_a, atoi_a, itoa_a
29    dw printlnpos_a

```

然后将该程序写进 21h 向量中

```

1 | WRITE_INT_VECTOR 21h, system_call

```

由于写向量的操作也会经常使用，因此我使用宏来进行

```

1  %macro WRITE_INT_VECTOR 2
2    push ax
3    push es
4    mov ax, 0
5    mov es, ax      ; ES = 0
6    mov word[es:%1*4], %2 ; 设置中断向量的偏移地址
7    mov ax, cs
8    mov word[es:%1*4+2], ax ; 设置中断向量的段地址=CS
9    pop es
10   pop ax
11 %endmacro

```

(6) INT21H ah=00h 的系统调用

功能是一个字符串中的小写变为小写，实现方式为汇编+c，汇编在 ./lib/system_a.asm 中，c语言在 ./lib/system_c.c

```

1  global toupper_a
2  extern toupper_c
3  toupper_a:
4    push es      ; 传递参数
5    push dx      ; 传递参数
6    call dword toupper_c
7    pop dx       ; 丢弃参数
8    pop es       ; 丢弃参数
9    ret

```

```

1 void toupper_c(char* str) {
2     int i=0;
3     while(str[i]) {
4         if (str[i] >= 'a' && str[i] <= 'z')
5             str[i] = str[i] - 'a' + 'A';
6         i++;
7     }
8 }

```

(7) INT21H ah=01h 的系统调用

功能是一个字符串中的大写变为小写，实现方式为汇编+c，汇编在 `./lib/system_a.asm` 中，c语言在 `./lib/system_c.c`

```

1 global tolower_a
2 extern tolower_c
3 tolower_a:
4     push es      ; 传递参数
5     push dx      ; 传递参数
6     call dword tolower_c
7     pop dx       ; 丢弃参数
8     pop es       ; 丢弃参数
9     ret

```

```

1 void tolower_c(char* str) {
2     int i=0;
3     while(str[i]) {
4         if (str[i] >= 'A' && str[i] <= 'Z')
5             str[i] = str[i] - 'A' + 'a';
6         i++;
7     }
8 }

```

(8) INT21H ah=02h 的系统调用

功能是将字符串转整数，实现方式为汇编+c，汇编在 `./lib/system_a.asm` 中，c语言在 `./lib/system_c.c`

```

1 global atoi_a
2 extern atoi_c
3 atoi_a:
4     push es      ; 传递参数;
5     push dx      ; 传递参数
6     call dword atoi_c
7     pop dx       ; 丢弃参数
8     pop es       ; 丢弃参数
9     ret

```



```

1 //字符串转数字
2 extern int strlen(char *);
3 int atoi_c(char *str) {
4     int res = 0; // Initialize result
5     int len=strlen(str);
6     for (int i = 0; i<len; ++i) {
7         res = res*10 + str[i] - '0';
8     }
9     // return result.
10    return res;
11 }

```

(9) INT21H ah=03h 的系统调用

功能是将整数转字符串，实现方式为汇编+c，汇编在 `./lib/system_a.asm` 中，c语言在 `./lib/system_c.c`

```

1 global itoa_a
2 extern itoa_c
3 itoa_a:
4     push es      ; 传递参数buf
5     push dx      ; 传递参数buf
6     mov ax, 0
7     push ax      ; 传递参数base
8     mov ax, 10   ; 10进制
9     push ax      ; 传递参数base
10    mov ax, 0
11    push ax      ; 传递参数val
12    push bx      ; 传递参数val
13    call dword itoa_c
14    pop bx       ; 丢弃参数
15    pop ax       ; 丢弃参数
16    pop ax       ; 丢弃参数
17    pop ax       ; 丢弃参数
18    pop dx       ; 丢弃参数
19    pop es       ; 丢弃参数
20    ret

```

```

1 char* itoa_c(int num, int base, char* str)
2 {
3     int i = 0;
4     int isNegative = 0;
5     if (num == 0) {
6         str[i] = '0';
7         str[i + 1] = '\0';
8         return str;
9     }
10    if (num < 0 && base == 10) {
11        isNegative = 1;
12        num = -num;
13    }
14    while (num != 0) {
15        int rem = num % base;
16        str[i++] = (rem > 9) ? (rem - 10) + 'A' : rem + '0';
17        num = num / base;
18    }
19    if (isNegative == 1) {

```

```

20     str[i++] = '-';
21 }
22     str[i] = '\0';
23     reverse(str, i);
24     return str;
25 }

```

(10) INT21H ah=04h 的系统调用

功能是在指定位置打印字符串，由于 bios 中已经有该功能，因此只需要使用，便可

```

1  extern strlen
2  printlnpos_a:
3      pusha
4      mov bp, dx      ; es:bp=串地址
5      push es        ; 传递参数
6      push bp        ; 传递参数
7      call dword strlen ; 返回值ax=串长
8      pop bp         ; 丢弃参数
9      pop es         ; 丢弃参数
10     mov bl, 07h     ; 颜色
11     mov dh, ch      ; 行号
12     mov dl, cl      ; 列号
13     mov cx, ax      ; 串长
14     mov bh, 0       ; 页码
15     mov al, 0       ; 光标不动
16     mov ah, 13h     ; BIOS功能号
17     int 10h
18     popa
19     ret

```

(11) INT22H ah=00h 的系统调用

功能是在屏幕上显示 int 22H。int22 属于另外一个向量上，而且和 int21 的模式基本相似，因此只需要重复一样的工作并写入向量表上便可。

```

1  global showint22
2  showint22:
3      pusha
4      PRINT strint22, strint22_len, 13, 20
5      popa
6      ret
7  data1:
8      strint22 db 'int 22H'
9      strint22_len equ ($-strint22)

```

(12) 系统调用测试程序

在写完所以的系统调用后，需要写一个测试程序来进行测试调用情况。由于该部分程序的编写相对比较简单，因此我就仅举一个测试例子。该测试程序可通过 run 5 来调用。

测试功能号 00h：小写变大写的系统调用

```

1  test_upper:
2      mov ax, cs
3      mov es, ax

```

```

4      mov dx,string      ;将需要操作的字符串首地址放进dx
5      mov ah,00h        ;功能号
6      int 21h           ;系统调用
7      PRINT string,string_len,3,15
8
9      mov ah,0
10     int 16h
11     cmp al,27          ;若按esc便退出测试程序
12     je quit
13
14     string db 'aaaBBBcccDDDeee'
15     string_len equ ($-string)

```

其他测试的方式都相似，具体的显示效果请看实验结果部分

(13) C语言的基本输入输出库的设计

实现 `putch()`、`getch()`、`gets()`、`puts()`、`printf()`、`scanf()` 等基本输入输出库过程

注意：由于在实验三的时候，我在**实验三c内核**的设计的时候已经封装实现过不少的基本库，但是函数名称不同，因此我直接重新调用原有函数名来定义便可。

该库的所有实现都在 `./usrpro/c_test/stdio.h` 中

- `putchar()` 的实现

该函数的实现需要通过汇编的 `uint 10h` 中断来实现，并且在内核中我已经实现过，因此我直接调用。

汇编中：

```

1      global putchar_color
2      putchar_color:      ;函数：在光标处打印一个彩色字符
3      pusha
4      push ds
5      push es
6      mov bx,0            ;页号=0
7      mov ah,03h          ;功能号：获取光标位置
8      int 10h             ;dh=行，dl=列
9      mov ax,cs
10     mov ds,ax           ;ds=cs
11     mov es,ax           ;es=cs
12     mov bp,sp
13     add bp,20+4         ;参数地址，es:bp指向要显示的字符
14     mov cx,1            ;显示1个字符
15     mov ax,1301h        ;AH=13h（功能号）、AL=01h（光标置于串尾）
16     mov bh,0           ;页号
17     mov bl,[bp+4]       ;颜色属性
18     int 10h            ;显示字符串（1个字符）
19     pop es
20     pop ds
21     popa
22     retf

```

c中实现

```

1 extern void putchar_color(char c,int color);//在光标处打印一个字符
2 void putchar(char c){
3     putchar_color(c, 0x07);
4 }

```

- `getch()` 的实现

`getch()` 的实现同样需要汇编的中断来实现

```

1 global getch
2 getch:          ; 读取一个字符
3     mov ah, 0    ; 功能号
4     int 16h      ; 读取字符, al=读到的字符
5     retf

```

- `gets()` 的实现

`gets()` 的实现，我在实验三中曾经写过一个 `void readcmd(char* *buffer,int *maxlen*)` 的函数，可以获取一整行的指定输入字数的输入，具体实现如下：

```

1 void readcmd(char* buffer,int maxlen)
2 {
3     int i=0;
4     while(1)
5     {
6         char tempc=getch();
7         if(!(tempc==0xD || tempc=='\b' || tempc>=32 && tempc<=127))
8         {
9             continue;
10        }
11        if(i>0&&i<maxlen-1)
12        {
13            if(tempc==0x0D)
14            {
15                break;//回车，停止读取
16            }
17            else if(tempc=='\b')//删除键
18            {
19                putchar("\b");
20                putchar(' ');
21                putchar("\b");
22                i--;
23            }
24            else
25            {
26                putchar_color(tempc,9);
27                buffer[i] = tempc;
28                ++i;
29            }
30        }
31        else if(i>=maxlen-1)//达到字符最大值，只能退课或回车
32        {
33            if(tempc == '\b')
34            { // 按下退格，则删除一个字符
35                putchar("\b");
36                putchar(' ');

```

```

37     putchar('\b');
38     i--;
39 }
40 else if(tempc == 0x0D)
41 {
42     break; // 按下回车，停止读取
43 }
44 }
45 else if(i<=0)
46 {
47     if(tempc == 0x0D) {
48         break; // 按下回车，停止读取
49     }
50     else if(tempc != '\b') {
51         putchar_color(tempc,9);
52         buffer[i] = tempc;
53         ++i;
54     }
55 }
56 }
57 putchar('\r'); putchar('\n');
58 buffer[i] = '\0';
59 }

```

因此：gets() 只需要直接调用便可

```

1 void gets(char* buffer)
2 {
3     readcmd( buffer,20);
4 }

```

- puts() 的实现

puts() 的实现也与我之前实验三中 print() 一样，打印一个字符串，具体实现如下：

```

1 void print(const char* str) {
2     int len=strlen(str);
3     for(int i = 0; i < len; i++) {
4         putchar(str[i]);
5     }
6 }

```

因此 puts(const char* str) 直接调用便可

```

1 void puts(const char* str)
2 {
3     print(str);
4 }

```

- my_printf() 的实现

my_printf() 和 my_scanf() 的实现的的核心部分就可变参数的函数的编程方式

可变参数实现原理：C调用约定下可使用 va_list 系列变参宏实现变参函数，用法如下：

```

1  #include <stdarg.h>
2  int VarArgFunc(int dwFixedArg, ...){ //以固定参数的地址为起点依次确定各变参的内存起始地址
3      va_list pArgs = NULL; //定义va_list类型的指针pArgs，用于存储参数地址
4      va_start(pArgs, dwFixedArg); //初始化pArgs指针，使其指向第一个可变参数。该宏第二个参数是变参
    列表的前一个参数，即最后一个固定参数
5      int dwVarArg = va_arg(pArgs, int); //该宏返回变参列表中的当前变参值并使pArgs指向列表中的下个变
    参。该宏第二个参数是要返回的当前变参类型
6      //若函数有多个可变参数，则依次调用va_arg宏获取各个变参
7      va_end(pArgs); //将指针pArgs置为无效，结束变参的获取
8      /* Code Block using variable arguments */
9  }

```

因此有了这个可变参数的处理，我就可以开始实现 `printf` 函数了，实现方式：循环遍历第一个字符串，每当遇到%，则根据后面字符的不同输出不同的字符，具体代码如下：

```

1  int my_printf(const char* fmt,...)
2  {
3      va_list arg_ptr;
4      char array[100];
5      char *str;
6      va_start(arg_ptr,fmt);
7      while((*fmt)!='\0')
8      {
9          if(*fmt=='\n')
10         {
11             putchar('\n');
12             putchar('\r');
13         }
14         else if(*fmt=='%')
15         {
16             fmt++;
17             switch(*fmt)
18             {
19                 case 'd':
20                 {
21                     itoa_c(va_arg(arg_ptr,int),10,array); //整数转字符串
22                     str=array;
23                     while(*str!='\0')
24                     {
25                         putchar(*str);
26                         str++;
27                     }
28                     }break;
29                 case 's':
30                 {
31                     str=va_arg(arg_ptr,char*);
32                     while(*str!='\0')
33                     {
34                         putchar(*str);
35                         str++;
36                     }
37                     }break;
38                 case 'c':
39                 {
40                     putchar(va_arg(arg_ptr,int));
41                     }break;

```

```

42     default:break;
43     }
44     }
45     else
46     {
47         putchar(*fmt);
48     }
49     fmt++;
50 }
51 va_end(arg_ptr);
52 str=(char*)0;
53 }

```

- my_scanf() 的实现

my_scanf() 函数的实现基本与 my_printf 的实现是类似的，因此直接上代码

```

1 void my_scanf(const char* fmt,...)
2 {
3     va_list arg_ptr;
4     int dec;
5     int *d_ptr;
6     char str[50];
7     char *s_ptr;
8     char *c;
9     va_start(arg_ptr,fmt);
10    while((*fmt]!='\0')
11    {
12        if(*fmt=='%')
13        {
14            fmt++;
15            switch(*fmt)
16            {
17                case 'd':
18                {
19                    readcmd(str,16);//读取一个数字字符串
20                    dec=atoi_c(str);//将字符串转为整数
21                    d_ptr=va_arg(arg_ptr,int*);
22                    *d_ptr=dec;//整数赋值
23                }break;
24                case 's':
25                {
26                    s_ptr=va_arg(arg_ptr,char*);
27                    readcmd(s_ptr,16);//读取一个数字字符串
28                }break;
29                case 'c':
30                {
31                    c=va_arg(arg_ptr,char*);
32                    readcmd(str,2);//读取一个字符（另外一个为回车）
33                    *c=str[0];
34                }break;
35                default:break;
36            }
37        }
38    }
39    else{
40

```

```

41     }
42     fmt++;
43 }
44 va_end(arg_ptr);
45 s_ptr=(char*)0;
46 d_ptr=(int*)0;
47 }

```

(14) c语言测试程序

编写完c语言的基本库后，需要编写一个调用该库的用户程序来进行测试。该用户程序仅需要 `#include "stdio.h"` 便可以直接调用里面的函数。我把该测试程序当成一个新加的用户程序，因此可以直接 `run 6` 就可以在虚拟机中调用。 具体的测试带代码如下：

```

1  #include "stdio.h"
2  #define BUFLen 16
3  #define NEXTLINE putchar('\r');putchar('\n')
4  int main()
5  {
6      char *prin="your str is:\n";
7      my_printf(prin);
8
9      char str[50];
10     int number;
11     char *scan="%s%d";
12     my_scanf(scan,str,&number);//测试my_scanf
13
14     char* temp="str:%s,number:%d\n";
15     my_printf(temp,str,number);//测试my_printf
16
17     char str2[50]="test gets and puts:\n\r";
18     puts(str2);//显示提示
19
20     gets(str2);//测试gets
21     puts(str2);//测试puts
22
23     putchar('q');
24     getch();
25 }

```

注意：用户程序的进入和退出是通过汇编来进行的，汇编的程序在 `./usrprog/main.asm`。整个bin程序需要像内核一样进行混合编译链接。

```

1  BITS 16
2  extern main
3  global _start
4  _start:
5      pusha
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      ;数据段
9      call dword main ;直接调用c语言中的main函数
10 quit:
11     popa
12     retf             ;调用完便退出

```


(15) 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 sys_test.asm main.c main.asm system_c.c system_a.asm，以及各个部分占用的扇区数以及偏移量已经改变

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
16 cd c_test
17 nasm -f elf32 main.asm -o ../../temp/main_a.o
18 gcc -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -mpreferred-stack-boundary=2 -lgcc
   -shared main.c -fno-pic -o ../../temp/main_c.o
19 ld -m elf_i386 -N -Ttext 0xB900 --oformat binary ../../temp/main_a.o ../../temp/main_c.o -o
   ../../temp/main.bin
20 cd ..
21
22 cd ..
23
24 cd lib
25 nasm -f elf32 system_a.asm -o ../temp/system_a.o
26 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
27 cd ..
28 cd kernel
29 nasm -f elf32 kernel.asm -o ../temp/kernel.o
30 nasm -f elf32 kernel_a.asm -o ../temp/kernel_a.o
31 nasm -f elf32 ouch.asm -o ../temp/ouch.o
32 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
33 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 -o ../temp/kernel.bin
34 cd ..
35 rm ./temp/*.o
36
37 dd if=./temp/booter.bin of=myosv4.img bs=512 count=1 2>/dev/null
38 dd if=./temp/kernel.bin of=myosv4.img bs=512 seek=1 count=35 2>/dev/null
39 dd if=./temp/topleft.com of=myosv4.img bs=512 seek=36 count=2 2>/dev/null
40 dd if=./temp/topright.bin of=myosv4.img bs=512 seek=38 count=2 2>/dev/null
41 dd if=./temp/bottomleft.bin of=myosv4.img bs=512 seek=40 count=2 2>/dev/null
42 dd if=./temp/bottomright.bin of=myosv4.img bs=512 seek=42 count=2 2>/dev/null
43 dd if=./temp/list.bin of=myosv4.img bs=512 seek=44 count=2 2>/dev/null
44 dd if=./temp/main.bin of=myosv4.img bs=512 seek=46 count=24 2>/dev/null
```

```
45 dd if=./temp/sys_test.bin of=myosv4.img bs=512 seek=70 count=2 2>/dev/null
46 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`

- 返回参数 `ax` 的保存

在系统调用（即中断）我都使用了`save`和`restart`来对寄存器进行保护，但是当我测试一个中断的时候发现数值并没有改变。这时我发现我的中断需要通过`ax`来将操作后的数值返回到用户程序当中。但是我的保护操作将`ax`寄存器原封不动的保护了起来，导致我的返回是没有用的。这时我想到一个办法就是，因为寄存器的`restart`是通过结构体上的值来恢复的，所以我就在`restart`操作之前将返回的`ax`值放进结构体上对应`ax`的位置上，这样`restart`的`ax`就是我需要的`ax`了

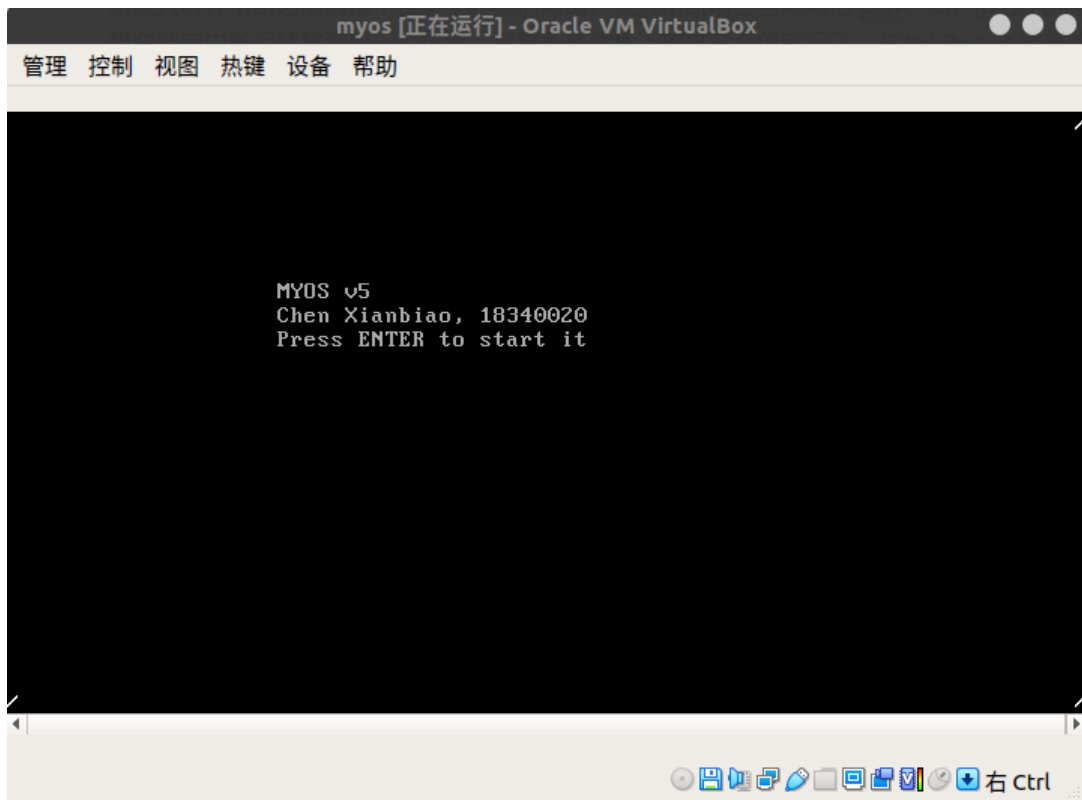
```
1 mov cx,ax
2 push cx
3 call dword getr1
4 pop cx
5 mov si, ax
6 mov [cs:si+0],cx
```

- 可变参数函数的编写

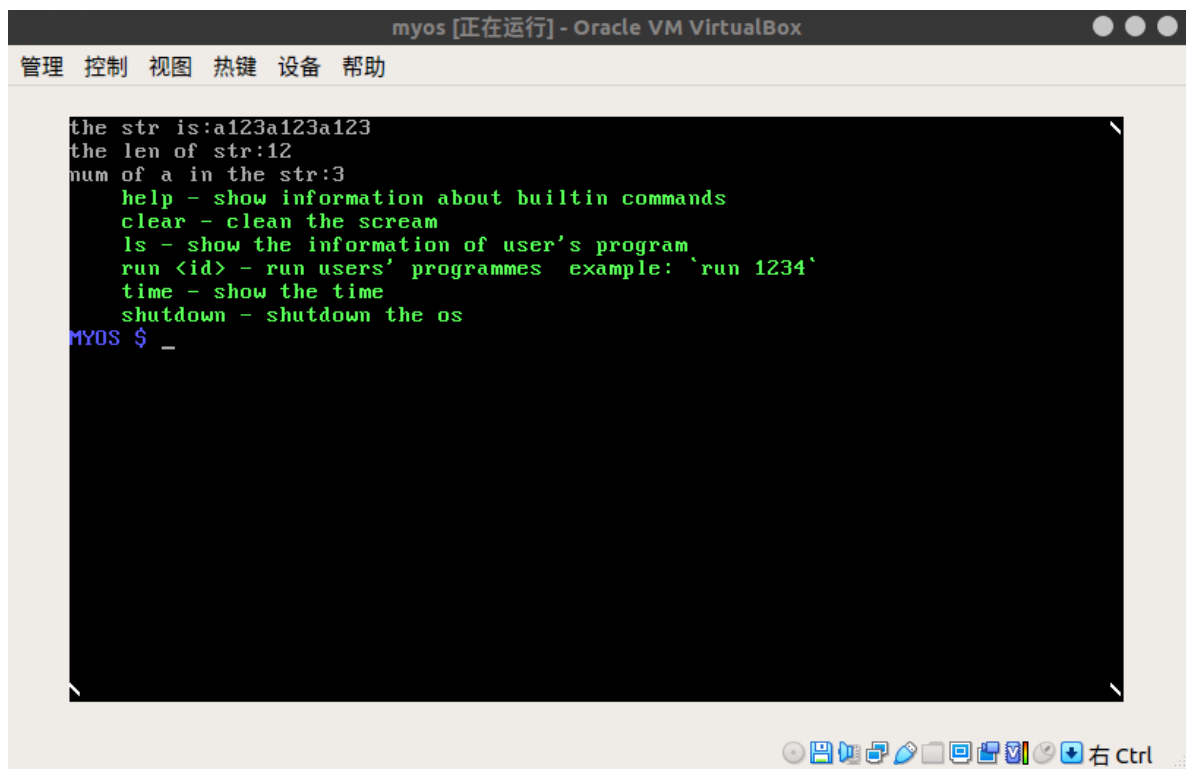
本次实验的另一个难点就是 `printf` 和 `scanf` 函数的编写，因为之前还没有接触过可变参数函数的编写，所以一开始是不知所措的，但是通过网上搜一些资料后就发现，这个实现只需要调用一个头文件及其函数就可以逐个参数访问。

2) 实验结果展示

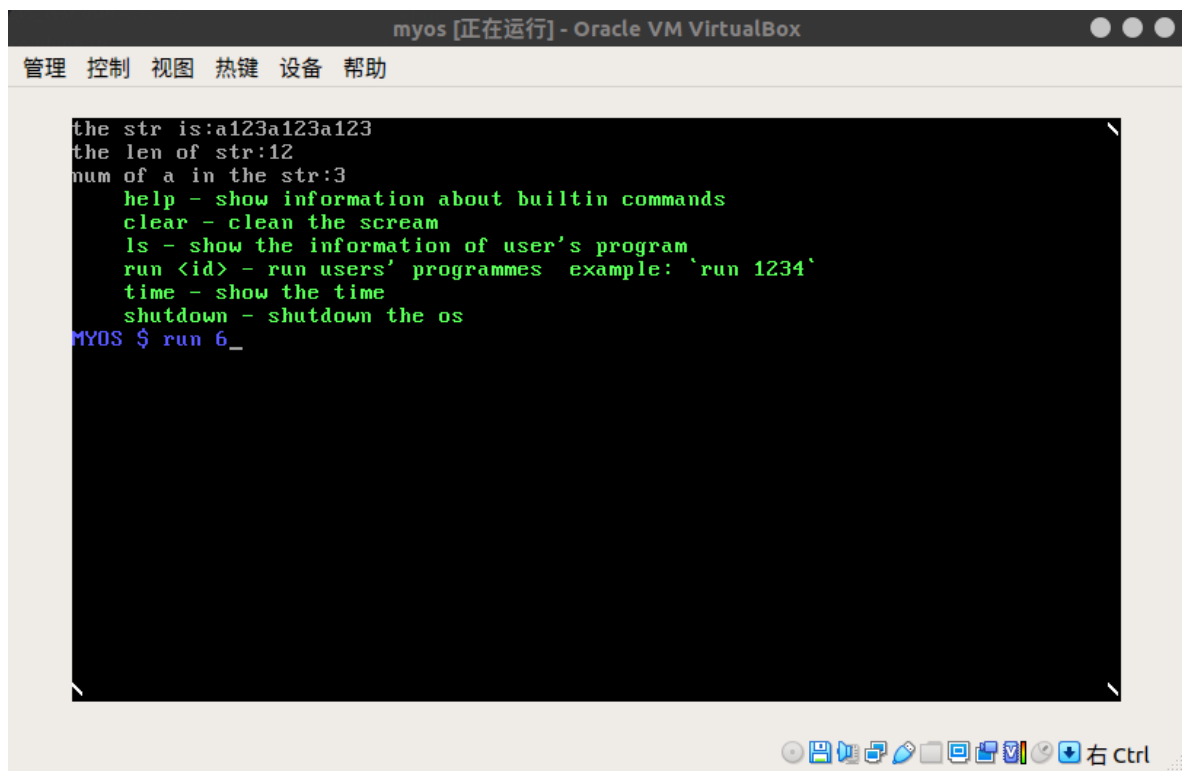
- 首先进入开机画面（风火轮还在）



- 点击 enter 进入命令行



- 输入 run1432 逐个运行用户程序，发现和之前的运行是一样的，这里就不再截图展示了
- 输入 run 6 来测试c基本库函数的编写



myos [正在运行] - Oracle VM VirtualBox

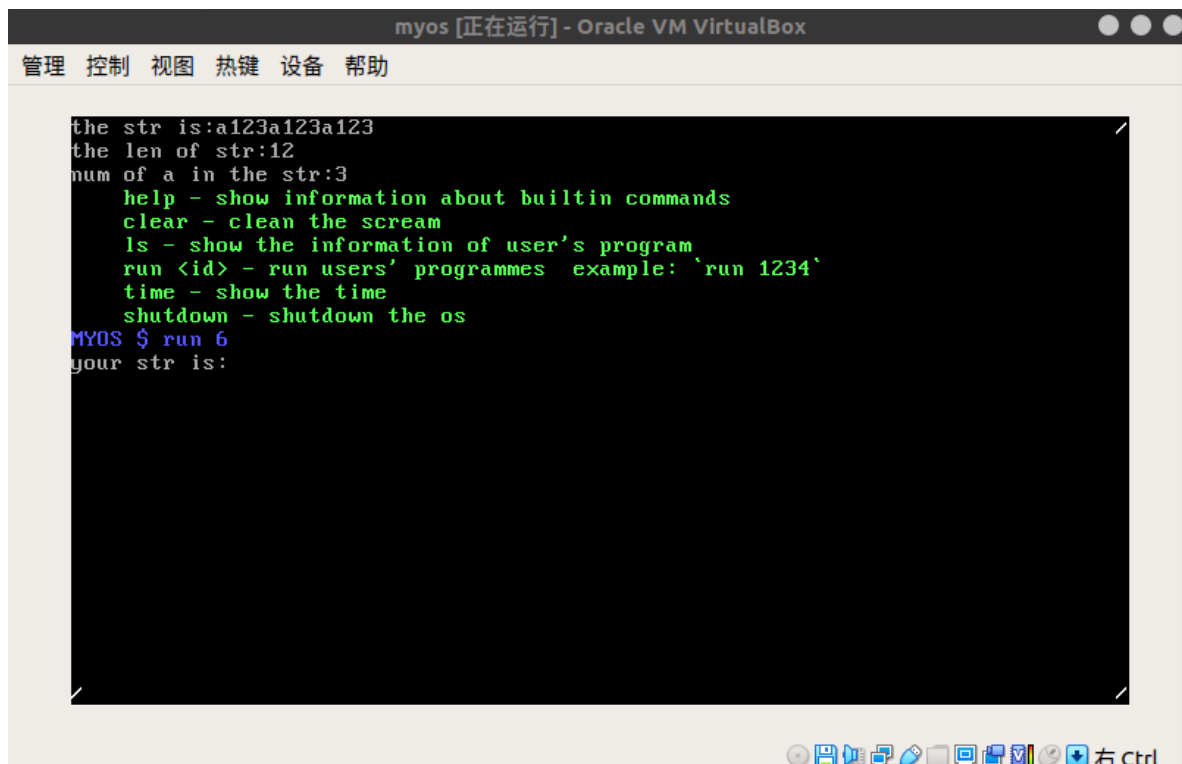
管理 控制 视图 热键 设备 帮助

```
the str is:a123a123a123
the len of str:12
num of a in the str:3
  help - show information about builtin commands
  clear - clean the screen
  ls - show the information of user's program
  run <id> - run users' programmes example: `run 1234`
  time - show the time
  shutdown - shutdown the os
MYOS $ run 6_
```

右 Ctrl

首先出现的是 `your str is:` 这个是通过下面代码输出的

```
1 char *prin="your str is:\n";
2 my_printf(prin);
```



myos [正在运行] - Oracle VM VirtualBox

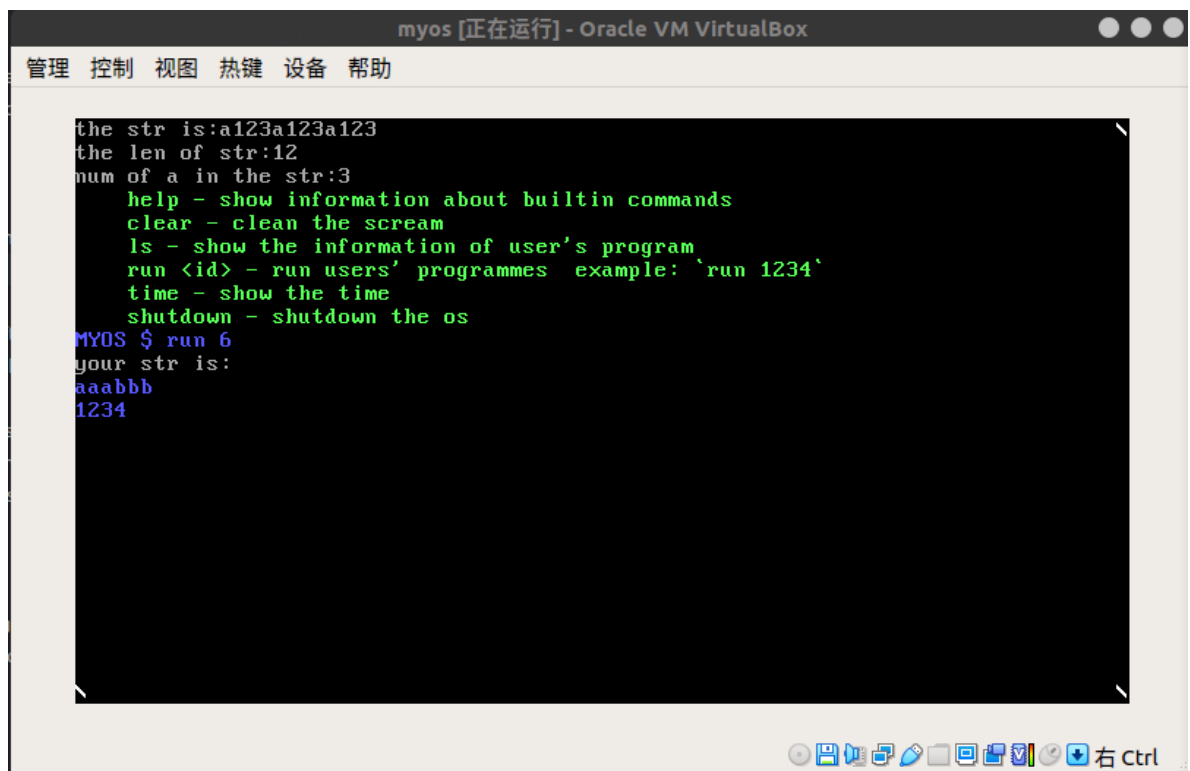
管理 控制 视图 热键 设备 帮助

```
the str is:a123a123a123
the len of str:12
num of a in the str:3
  help - show information about builtin commands
  clear - clean the screen
  ls - show the information of user's program
  run <id> - run users' programmes example: `run 1234`
  time - show the time
  shutdown - shutdown the os
MYOS $ run 6
your str is:
```

右 Ctrl

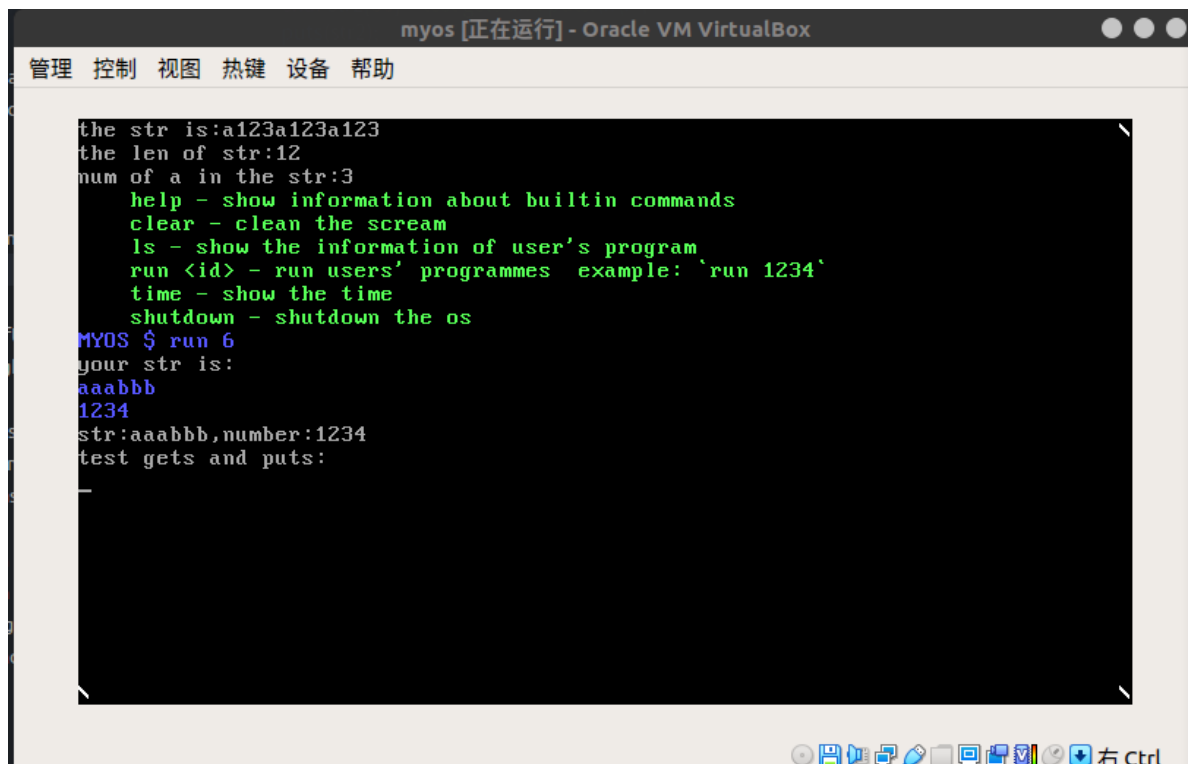
```
1 char str[50];
2 int number;
3 char *scan="%s%d";
4 my_scanf(scan,str,&number);//测试my_scanf
```

接下来便是 `my_scanf` 函数的输入，先输入一个字符串进入str，在输入数字进入number



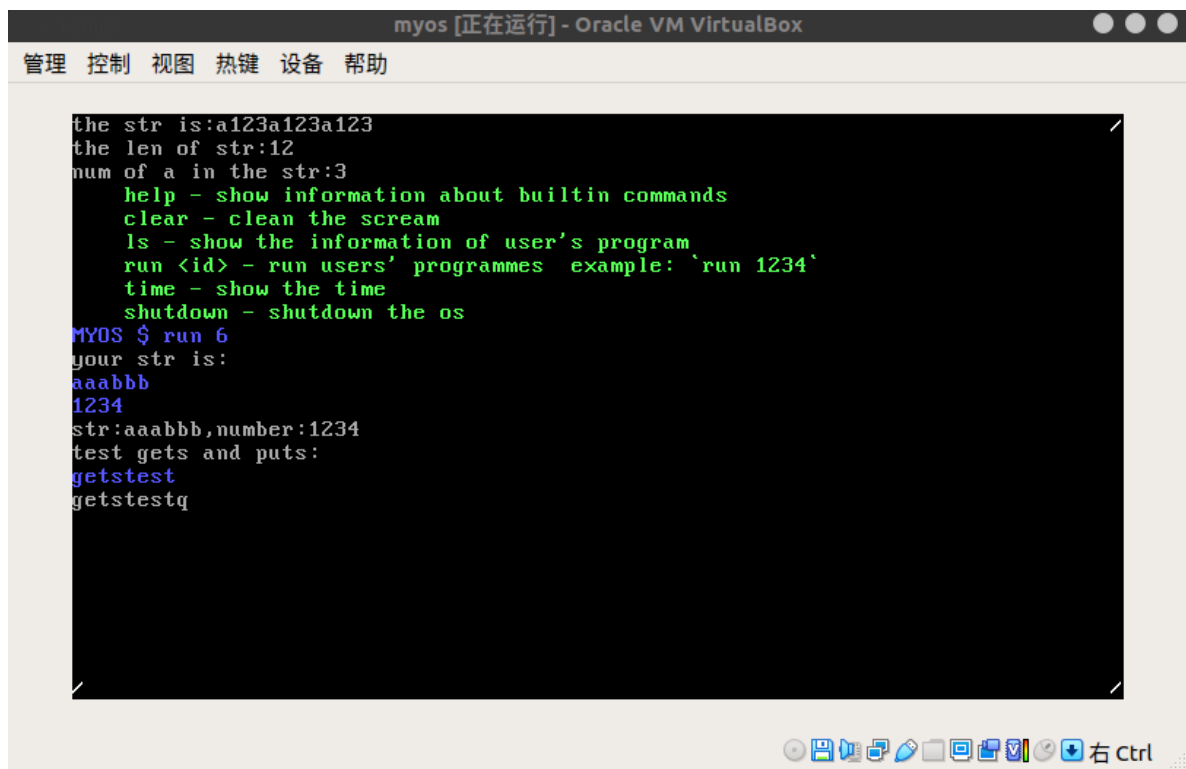
```
1 char* temp="str:%s,number:%d\n";
2 my_printf(temp,str,number);//测试my_printf
```

之后调用 my_printf 输出测试刚刚的输入是否成功



```
1 gets(str2);//测试gets
2 puts(str2);
3 putchar('q');
```

随后就是测试gets和puts

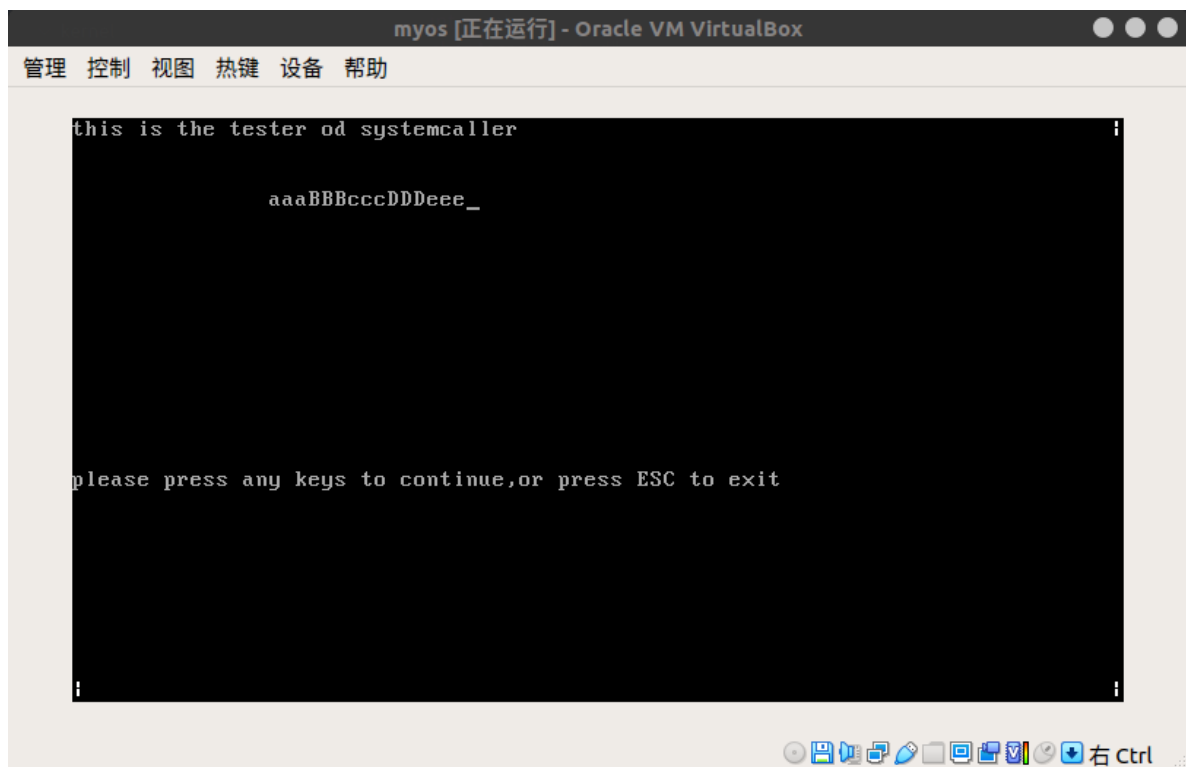


```
the str is:a123a123a123
the len of str:12
num of a in the str:3
help - show information about builtin commands
clear - clean the screen
ls - show the information of user's program
run <id> - run users' programmes example: `run 1234`
time - show the time
shutdown - shutdown the os
MYOS $ run 6
your str is:
aaabbb
1234
str:aaabbb,number:1234
test gets and puts:
getstest
getstestq
```

可以看出编写的库函数的输入输出结果是正确的

- 系统调用的测试

首先输入 `run 5` 进入测试程序,此时字符串为 `aaaBBBcccDDDeee`

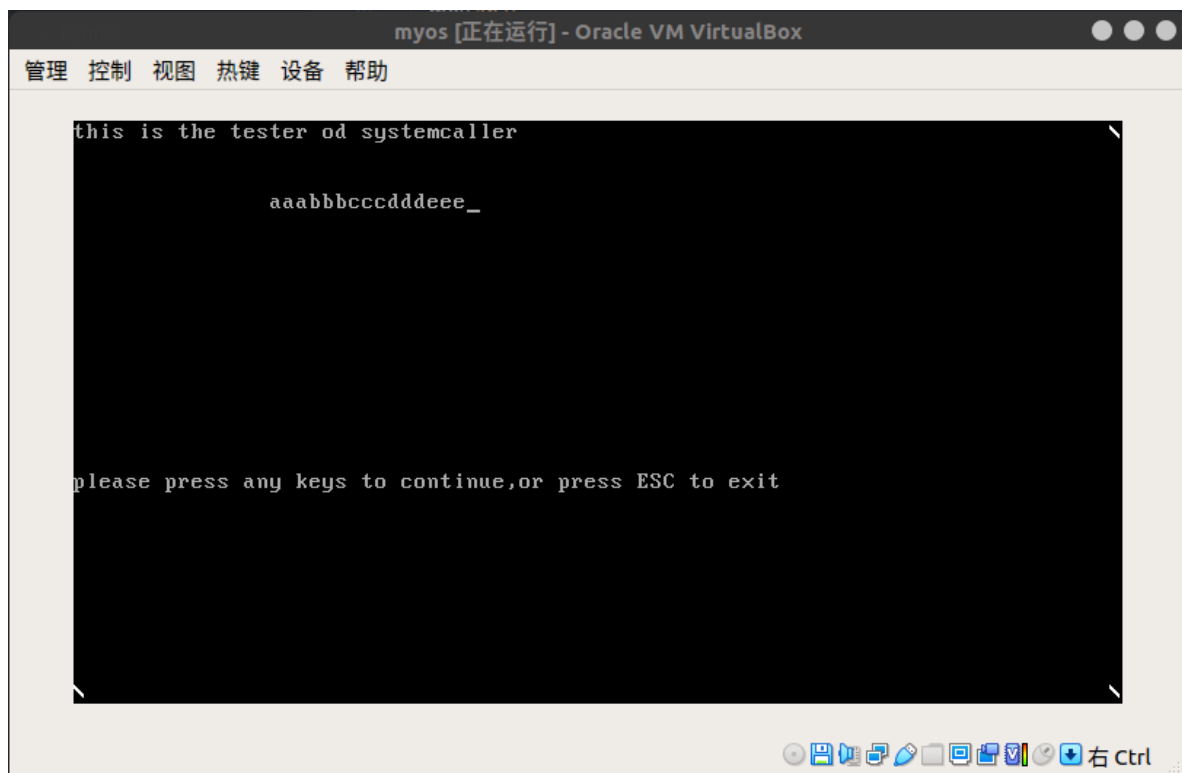


```
this is the tester od systemcaller

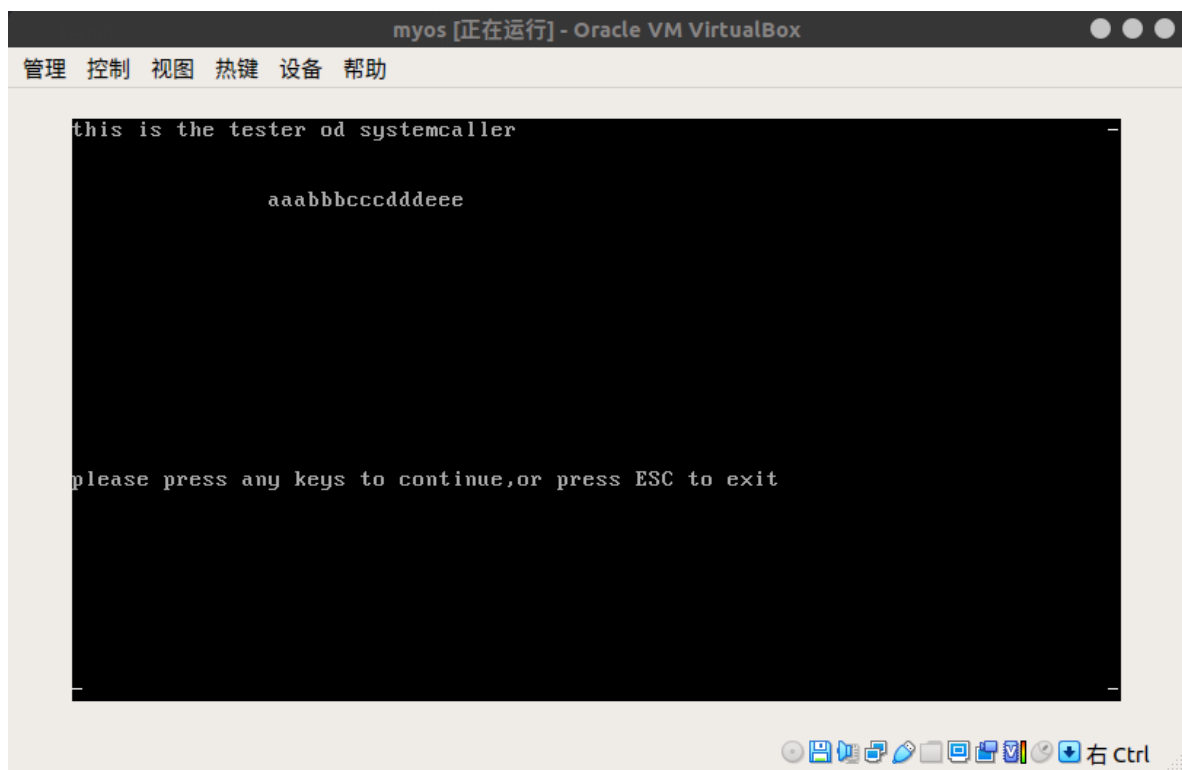
aaaBBBcccDDDeee_

please press any keys to continue,or press ESC to exit
```

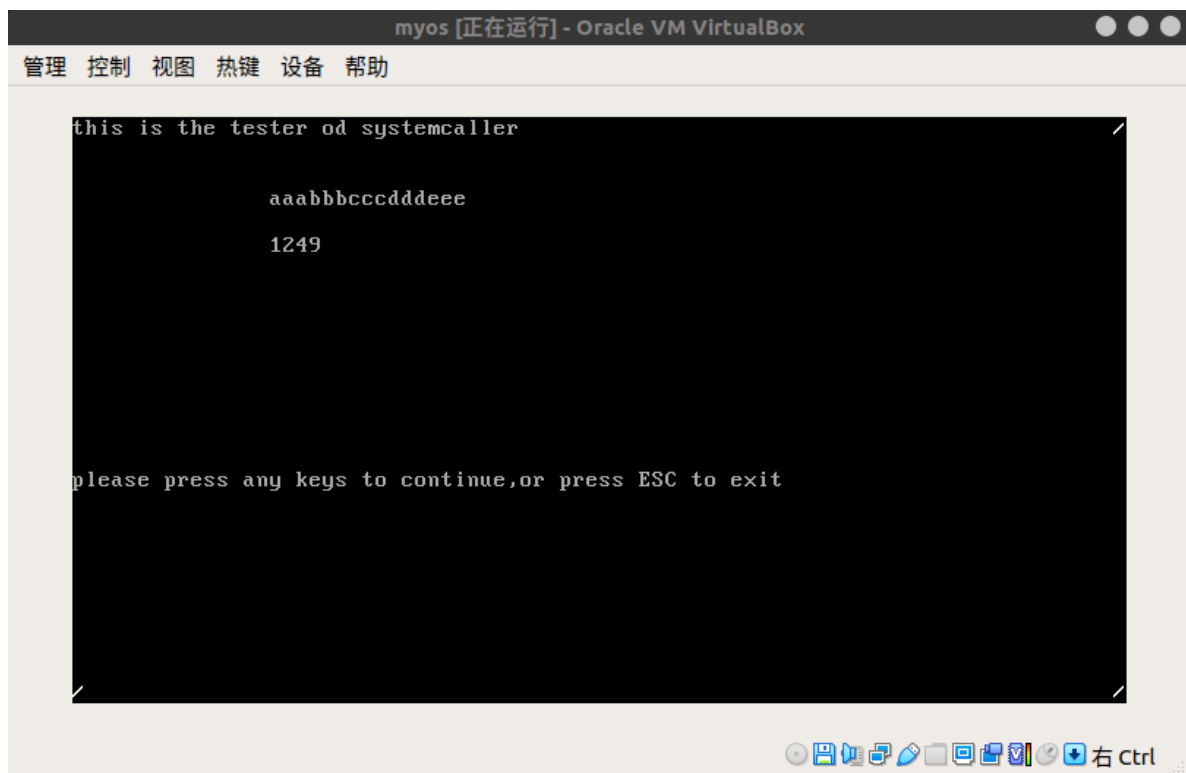
调用 `00h` 的变大写调用后（点击键盘任意键）：



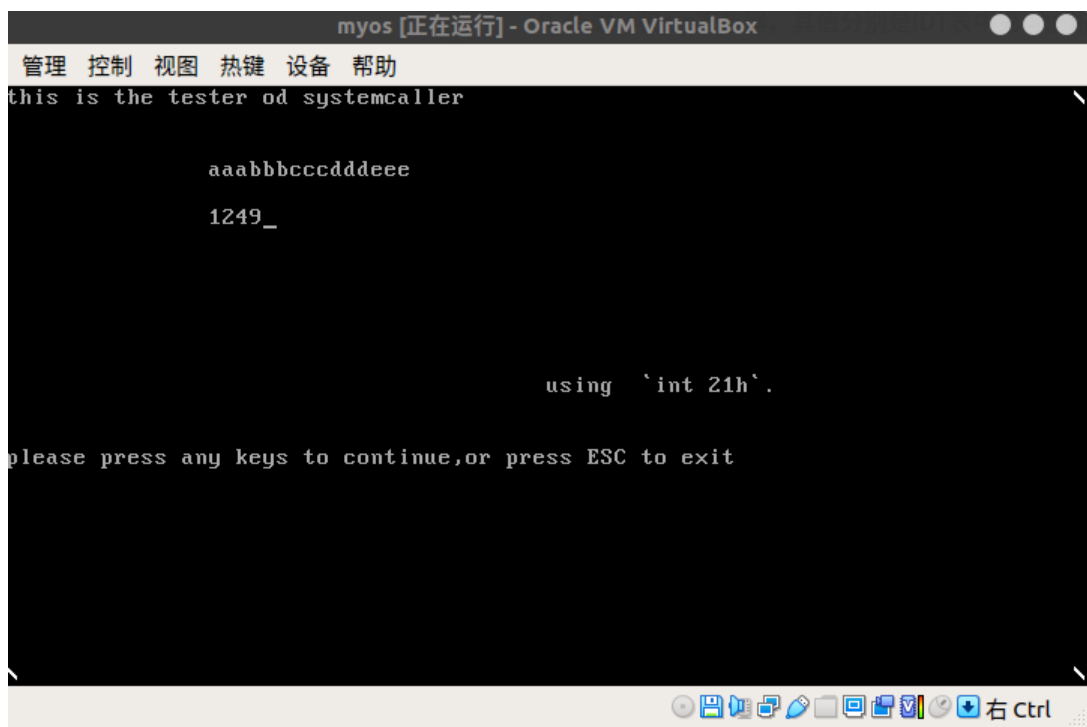
调用 01h 的变小写调用后（点击键盘任意键）：



随后出现一个数字的字符串1229，调用 02h 将字符串转成数字（该过程看不到），但是将该数字+20，然后调用 03h 将数字转成字符串输出就可以看到出现1249，证明两个调用是成功的：

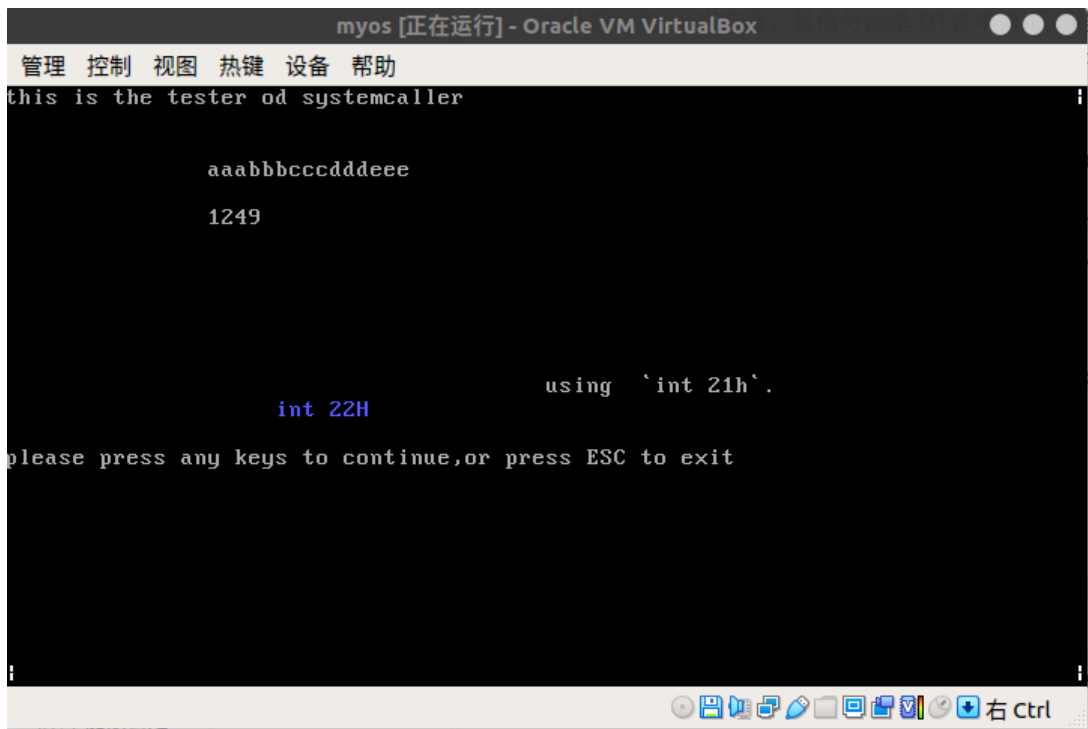


测试 04h 输出字符串调用（点击键盘任意键）：



可以看到出现字符串 using int21h

之后测试 int 22h ,在屏幕某处显示 int 22H （点击键盘任意键）：



- 随后退出测试程序，输入 shutdown 关机

5.实验体会

本次实验五相较于之前的实验工程量是比较大的，可以从我完成本次实验后文件夹文件数目可以看出这次实验的量是很大的，并且难度也比较大的。

懂得了系统调用的具体实现方法，因为内核的函数与用户程序是分开的，所以用户程序是不能直接访问内核中的函数的，所以需要专门指定一个中断号（21h）对应服务处理程序总入口，然后再将服务程序所有服务用功能号区分，并作为一个参数（通常是 ah）从用户中传递过来，程序再进行分支，进入相应的功能实现子程序。

也懂得了如何在进入和退出中断对所有寄存器进行保护和恢复的过程。这两个函数的实现是本次实验里最为困难的部分，虽然脑子里有一点思路去进行，但是每次都是去尝试发现总有一些寄存器是没有保护恰当，导致程序运行失败。这才深刻理解到老师上课所说的“一个不漏，准确不误”是多么难实现。特别是sp寄存器的保护，因为栈的操作很多，导致sp寄存器的变化也是很多，最后保护需要的理解也要很到位。

虽然在本次实验之前我已经封装过不少的c语言库函数，但这次的实现更加的具体化，也实现了 printf 和 scanf 两个最为常用的输入输出函数，最后测试成功的时候，成就感也是满满的

因为代码量也变得越来越多了，占用的软盘空间也变大，这次实验使用到了1号柱面，这让我更清楚了解到了软盘中柱面，磁头，扇区的对应关系。

总的来说，这次系统调用实验虽然困难险阻挺多的，但是在困难一个个解决过后，我对操作系统的理解也越来越深刻。

6.参考资料

- 系统调用的概念及原理：https://blog.csdn.net/qg_43646576/article/details/102841078
- 一个操作系统的实现——进程：<https://blog.csdn.net/fukai555/article/details/41625619>
- 可变参函数（my_printf可变参函数的实现）：https://blog.csdn.net/qg_39191122/article/details/79900720