

# 实验二

## 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

## 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

## 实验方案

---

### 实验环境

硬件：个人计算机

操作系统：Windows 10

虚拟机软件：VirtualBox、Bochs

# 实验开发工具

语言工具：16位x86汇编语言

汇编器：nasm

汇编调试工具：Bochs

磁盘映像文件浏览编辑工具：WinHex、Winimage

代码编辑器：Visual Studio Code

# 程序设计

## 磁盘存储设计

本实验总共有5个程序（1个监控程序和4个用户程序），而这五个程序都需要放到实验一创建的软盘映像文件。

根据实验的要求，我设计了一个自定义的磁盘存储组织：



首扇区毫无疑问是存放监控程序，以便开机立刻启动。第二个扇区对应实验内容的第四条，用于记录盘区里的用户程序，方便监控程序调用。具体的表格信息设计如下：

00000200	31 20 20 20 20 20 20 20	43 4F 4D 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 30 30 33	1	COM	0003
00000220	32 20 20 20 20 20 20 20	43 4F 4D 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 30 30 34	2	COM	0004
00000240	33 20 20 20 20 20 20 20	43 4F 4D 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 30 30 35	3	COM	0005
00000260	34 20 20 20 20 20 20 20	43 4F 4D 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 30 30 36	4	COM	0006
00000280	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00			
000002A0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00			
000002C0	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00			

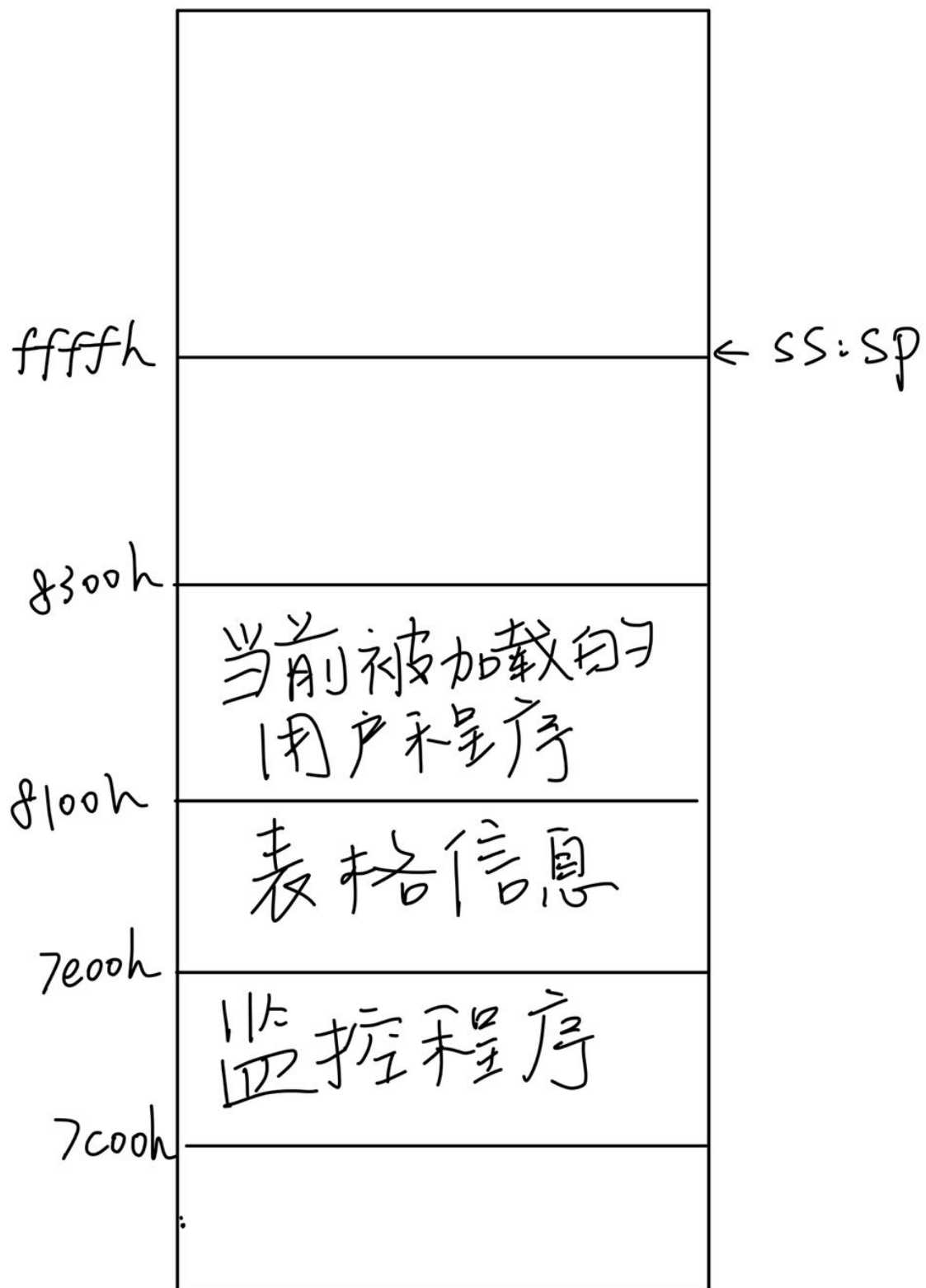
与fat12文件系统类似，前11个字节（8个字节用于文件名，3个字节用于文件扩展名）。而最后的4个字节则是用户程序所在的扇区号。

剩余的扇区就是存放我修改实验一的程序写出来的用户程序的com格式二进制格式数据。

## 内存安排

BIOS装入引导扇区的的时候，会把首扇区内容放到内存的0000H: 7C00H处。根据这个设计，在加载表格信息的时候，我把表格信息放到了0000H: 7e00h处。

同时，参考老师给出的资料，我将用户程序加载到内存的800H: 100H处。具体内存安排如下：



值得强调的一点是，由于现在的原型操作系统还是单道处理系统。因此，监控程序一次只会加载一个用户程序，且加载的位置是800H:100H（即08100H）处。

不仅如此，相比于实验一，我还显式地初始化了堆栈空间（从0FFFFH开始）。这样的话，我的汇编程序就可以使用堆栈操作（例如call、push、pop）

实验中使用的BIOS调用

功能	中断号	功能号
在当前光标处按原有属性显示字符	10H	0AH
显示字符串	10H	13H
获取当前状态和光标位置	10H	04H
用文本坐标下设置光标位置	10H	02H
读下一个按键	16H	00H
读扇区	13H	02H

10、功能 0AH

功能描述：在当前光标处按原有属性显示字符

入口参数：AH = 0AH

AL = 字符

BH = 显示页码

BL = 颜色 (图形模式，仅适用于PCjr)

CX = 重复输出字符的次数

出口参数：无

## 5、功能04H

功能描述：获取当前状态和光笔位置

入口参数：AH = 04H

出口参 数：AH = 00h——光笔未按下/未触发，01h——光笔已按下/已触发

BX = 像素列(图形X坐标)

CH = 像素行(图形Y坐标，显示模 式：04H~06H)

CX = 像素行(图形Y坐标，显示模式：0DH~10H)

DH = 字符行(文本Y坐标)

DL = 字符列(文本X坐 标)

## 3、 功能02H

功能描述：用文本坐标下设置光标位置

入口参数：AH = 02H

BH = 显示页码

DH = 行(Y坐标)

DL = 列(X坐标)

出口参数：无

读扇区↵	13H↵	02H↵	AL: 扇区数(1~255)↵ DL: 驱动器号(0 和 1 表示软盘, 80H 和 81H 等表示硬盘或 U 盘) ↵ DH: 磁头号(0~15)↵ CH: 柱面号的低 8 位↵ CL: 0~5 位为起始扇区号(1~63), 6~7 位为硬盘柱面号的高 2 位(总共 10 位柱面号, 取值 0~1023)↵ ES:BX: 读入数据在内存中的存储地址↵	返回值: ↵ ■ 操作完成后 ES:BX 指向数据区域的起始地址↵ ■ 出错时置进位标志 CF=1, 错误代码存放在寄存器 AH 中↵ ■ 成功时 CF=0、AL=0↵
显示字符串↵	10H↵	13H↵	AL: 放置光标的方式↵ BL: 字符属性字节↵ BH: 显示页(0~3)↵ DH: 行位置(0~24)↵ DL: 列位置(0~79)↵ CX: 字符串的字节数↵ ES:BP: 字符串的起始地址↵	AL=0/2 光标留在串头↵ AL=1/3 光标放到串尾↵ AL=0/1 串中只有字符↵ AL=2/3 串中字符和属性字节交替存储↵ BL: 位 7 为 1 闪烁↵ 位 6~4 为背景色 RGB↵ 位 3 为 1 前景色高亮↵ 位 2~0 为前景色 RGB↵

## 加载用户程序和返回监控程序的具体实现方式

### 加载用户程序

加载用户程序主要有两部分组成:

- 将用户程序从磁盘读入内存
- 将CPU控制权转交给用户程序

具体实现代码:

```
LoadnEx:
;读软盘或硬盘上的若干物理扇区到内存的ES:BX处:
mov ax,cs                ;段地址 ; 存放数据的内存基地址
mov es,ax                ;设置段地址 (不能直接mov es,段地址)
mov bx,8100h             ;偏移地址; 存放数据的内存偏移地址 0:8100h和800:100h相同
mov ah,2                 ;功能号
mov al,1                 ;扇区数
mov dl,0                 ;驱动器号 ; 软盘为0, 硬盘和U盘为80H
mov dh,0                 ;磁头号 ; 起始编号为0
mov ch,0                 ;柱面号 ; 起始编号为0
```

```
mov cl,3           ;起始扇区号 起始编号为1 2用来存放用户程序信息
int 13H           ;调用读磁盘BIOS的13h功能
; 用户程序已加载到指定内存区域中
call 800h:0100h    ;将控制权转交给用户程序
```

第一部分的实现就是上面说的使用BIOS调用。先把指定的用户程序所存放的扇区号放到cl寄存器里，然后按照读扇区这个bios调用的格式（上面原理有个截图）填充相应的寄存器即可。

而对于转交控制权，我使用了长距离直接转移Call语句 `call 800h:0100h`。由于是自己实现的监控程序，加载的用户程序位置是确定的，所以我使用了直接转移语句。同时，为了保持com文件的格式（即org 100h），我特地使用 `800h:100h`，确保用户程序能够正常运行。

不仅如此，Call语句还会将此时的CS和IP寄存器的值压入栈中。回顾一下我前面对于内存的设计，这部分操作时正确的，因为我已经正确地初始化了堆栈寄存器值和堆栈指针寄存器值。而这也为用户程序把控制权还给监控程序提供了便利。

## 返回监控程序

### 返回dos系统

加载部分，dos系统版本和我自己写的原型操作系统版本比较类似，而返回dos系统部分有些细微差别。

```
mov ax, 4c00h
int 21h
```

在dos操作系统里，我直接调用了dos中断程序服务，利用4CH功能号直接把控制权交还给dos系统。

### 返回监控程序

这部分也相对容易。

```
retf      ; 长距离返回语句
```

有了监控程序的长距离Call语句，监控程序的CS寄存器与IP寄存器已经在栈顶了。此时，调用retf语句，可以直接恢复CS和IP的值，相当于把控制权返回给监控程序了。

我个人认为，这样设计既简单，又能保证程序的正确性。

## 程序流程

这里主要介绍的是最终版监控程序的实验流程（根据实验内容，我实际上写了2个版本的监控程序 详情参见压缩包里的代码）

## 初始化程序

首先是利用伪指令初始化一些常量

```
; 用户程序存放位置（磁盘号）
com_1 equ 3
com_2 equ 4
com_3 equ 5
com_4 equ 6
NumOfProgram equ 4      ; 盘区里有的用户程序数目
```



```

org 7c00h ; 引导区程序 需要在0:7c00h上执行

Start:
mov ax, cs ; 置其他段寄存器值与cs相同
mov ds, ax ; 数据段
mov es, ax
mov ss, ax ; 堆栈段
mov ax, 0B800h ; 文本窗口显存起始地址
mov gs, ax ; GS = B800h
mov sp, 0 ; 初始化堆栈地址

```

这些常量指明了用户程序在软盘映像文件里的存放扇区号，便于监控程序直接调用。同时，使用伪指令赋值可以随时修改它们存放的扇区。

Start部分就是初始化四个段寄存器的值。由于com格式大小是64k，所以cs、ds、es、ss四个段都是指向同一个段地址的

## 设置窗口背景颜色

原理很简单，就是给整个大小为25 \* 80的显存，填充背景颜色为青色，字符为空格（即空白字符）。这样就可以形成一个空白的青色背景。

```

SetBackGround:
mov cx, 0x7d0 ; 等于25 * 80 即显存的大小
mov bx, 0x0 ; 用于显存偏移量
mov ax, 0x3020 ; 颜色为青色的空格字符

ls:
mov [gs:bx], ax ; 给gs:bx地址填充颜色
inc bx
inc bx
loop ls

```

这段代码，我就是利用一个循环语句，给显存的每个位置都填充青色空格字符，就可以实现将背景颜色设为青色。

值得注意的是，由于一个字符由两个字节决定，所以我在移动偏移地址的时候，使用了两个inc语句，即加2，保证正确。

## 欢迎语句

这里用作交互功能，主要是介绍本监控程序可以调用的用户程序有哪些。具体实现用的是上面提到的BIOS调用

```

Welcoming:
    mov bp, WelcomingMessage      ; BP=当前串的偏移地址
    mov ax, ds                    ; ES:BP = 串地址
    mov es, ax                    ; 置ES=DS
    mov cx, WelcomingMessageLength ; CX = 串长 (=9)
    mov ax, 1301h                 ; AH = 13h (功能号)、AL = 01h (光标置于串尾)
    mov bx, 0031h                 ; 页号为0 (BH = 0)
    mov dh, 0                     ; 行号=0
    mov dl, 0                     ; 列号=0
    int 10h                       ; BIOS的10h功能: 显示一行字符

    WelcomingMessage db " Hello, this is ouyhlán's monitor program! The following is
the user programs which you can use: ", 0dh, 0ah
    WelcomingMessageLength equ ($-WelcomingMessage)

```

欢迎语句是预先定义好的，放在引导扇区里（见上面代码的最后两行）。它会随着引导扇区代码，直接加载到内存里。而我的监控程序就会按照这个内存地址进行读取。

## 显示盘区表格信息

这部分与刚才的部分稍微有点不同，表格信息是独立于监控程序的，他单独存放在第二个扇区。所以我需要先加载后读取。

考虑到引导扇区已经占据了内存的0x7c00-0x7dff，我选择把这段信息加载到0x7e00-0x7fff。

加载后，由于我只有内存位置信息，这里我用了一个小技巧。我把es设置成0x7c00，这样我的偏移量bp就从0开始。

之后，我利用在初始化阶段定义的用户程序数目作为循环的计数值，赋值给si。同时，按照之前设计的，一个表格项共占32个字节，在设计的时候，我就按照ascii码进行记录，可以直接作为字符串进行输出。字符串的输出输出则利用了bios调用。效果见下一个板块的截图。

```

DisplayInfo:
    ; 读扇区, 将磁盘信息加载到0:7e00h
    mov bx, 7e00h
    mov ah, 2                ; 功能号
    mov al, 1                ; 扇区数
    mov dl, 0                ; 驱动器号 ; 软盘为0, 硬盘和U盘为80H
    mov dh, 0                ; 磁头号 ; 起始编号为0
    mov ch, 0                ; 柱面号 ; 起始编号为0
    mov cl, 2                ; 第2个扇区存放了磁盘信息
    int 13h                  ; 调用读磁盘BIOS的13h功能

    ; 显示表格信息
    mov ax, 7e0h
    mov es, ax                ; es = 7e0h
    mov bp, 0h                ; 逐一显示程序信息
    mov si, NumOfProgram      ; 要显示的程序信息数目
    mov dh, 2                 ; 设置从第3行开始显示
    mov dl, 0                 ; 列号为0

    ; 利用循环逐行显示表格信息
L1:

```

```

mov cx, 32                ; 要显示的信息长度为32
mov ax, 1301h             ; AH = 13h (功能号)、AL = 01h (光标置于串尾)
mov bx, 0031h             ; 页号=0, 字体=青底蓝字
int 10h                   ; 显示一行信息
add bp, 32
inc dh                    ; 换行
dec si
cmp si, 0
jz Input
jmp L1

```

按设计，首先需要把磁盘里的表格信息加载到对应的内存地址，之后重复调用显示字符串中断服务程序显示所有的表格信息。

## 等待用户输入

这部分模块的设计核心有两个：

- 用户输入时，监控程序提供回显功能（即可以在屏幕里看见自己的输入信息）
- 监控程序能保存用户输入，回车键停止读取输入值

由于希望交互界面可以美观一点，我先调用功能号为03h的中断10h来获取当前光标。按照CR+LF的换行方式修改了横坐标和纵坐标。

之后，我参考了dos系统和linux的shell里一般都会提供 `user>` 这样东西提示用户进行输入。所以我在这部分代码加入了显示字符串的过程就是在显示 `user>` 这个提示符。

紧接着，我利用 `inputbuf` 这个内存位置来存放用户的输入，以便后面进行使用。

当用户每输入一个字符时，我立刻调用中断对其进行输出，这就是所谓的回显功能Echo。

```

Input:                    ; 输入函数 选择执行的程序
    mov ah, 03h
    int 10h               ; 获取光标位置
    inc dh                ; 换行CR
    mov dl, 0             ; LF

    mov ax, 0
    mov es, ax            ; es = 0000h
    mov cx, UserMessageLength
    mov ax, 1301h         ; AH = 13h (功能号)、AL = 01h (光标置于串尾)
    mov bx, 0031h         ; 页号=0, 字体=青底蓝字
    mov bp, UserMessage
    int 10h
    mov bx, 0030h

    mov bp, 0             ; inputbuf[0]
I1:
    mov ah, 0
    int 16h               ; 等待按键输入
    cmp al, 0dh           ; 回车进入选择阶段
    jz Select

```

```

    mov [inputbuf + bp], al    ; inputbuf[bp] = al
    inc bp
Echo:
    mov cx, 1
    mov ah, 0ah                ; 设置显示一个字符
    int 10h                    ; 同步回显
    mov ah, 03h                ; 移动光标
    int 10h
    inc dl                      ; 更新光标位置
    mov ah, 02h
    int 10h                    ; 输出字符
    jmp I1

```

## 功能选择

根据前面用户的输入，监控程序会执行相应的程序。

这部分的思路，是以cl作为入口参数，不同的cl值代表加载不同的扇区到内存里（不同扇区即是不同的用户程序）。这样就可以根据用户输入的不同的命令,执行不同的程序了。

选择功能的实现就是C语言里的Switch-Case语句，逐一对比，满足即进行跳转。

```

Select:
    mov al, [inputbuf]
    cmp al, 'q'
    jz Exit

    cmp al, '1'
    mov cl, com_1
    jz LoadnEx

    cmp al, '2'
    mov cl, com_2
    jz LoadnEx

    cmp al, '3'
    mov cl, com_3
    jz LoadnEx

    cmp al, '4'
    mov cl, com_4
    jz LoadnEx

```

## 加载用户程序，转交控制权

加载主要就是填写读扇区bios调用的入口参数，cl寄存器的值由上一个选择模块决定，所以这部分代码打了注释表示不执行。

当加载完成后，调用call语句正式将控制权交给用户程序。

```
LoadnEx:
;读软盘或硬盘上的若干物理扇区到内存的ES:BX处:
mov ax,cs                ;段地址 ; 存放数据的内存基地址
mov es,ax                ;设置段地址 (不能直接mov es,段地址)
mov bx,8100h             ;偏移地址; 存放数据的内存偏移地址 0:8100h和800:100h相同
mov ah,2                 ; 功能号
mov al,1                 ; 扇区数
mov dl,0                 ; 驱动器号 ; 软盘为0, 硬盘和U盘为80H
mov dh,0                 ; 磁头号 ; 起始编号为0
mov ch,0                 ; 柱面号 ; 起始编号为0
;mov cl,3                ; 起始扇区号 起始编号为1 2用来存放用户程序信息
int 13H ;                调用读磁盘BIOS的13h功能
; 用户程序已加载到指定内存区域中
call 800h:0100h          ; 将控制权交给用户程序
jmp Start                ; 回到监控程序, 继续执行
```

最后的一条语句 `jmp Start` 保证了监控程序可以在用户程序返回后继续调用用户程序，实现了重复调用用户程序的功能。

## 实验过程和结果

### (1) 修改实验一的程序并在dos系统运行

具体代码见压缩里的文件夹1

为了突出四个小程序的不同，同时又适当地简化原有的反弹程序，我给这4个程序做了不同的反弹方向以及反弹形状。

1.com是从左边发射画一个w形状，2.com是从右边发射画一个w形状，3.com是从左边发射画一个m形状，4.com是从右边发射画一个m形状。

### 修改代码

修改主要有三个地方：

- 修改显示的区域
- 引入计数器使其在执行一定次数后自动停止
- 代码执行后，把控制权交回给dos系统

下面我就展示一下修改的部分

## 初始化代码

考虑到显示区域是25 \* 80，为了对称，我空出第一行，所以是24 \* 80。经过计算，第一部分显示区域：横坐标（1-12），纵坐标（0-39）。

```
Lmax equ 0
Rmax equ 39
Umax equ 1
Dmax equ 12
```

## 循环执行代码

在原来show模块的末尾，我添加了这三个语句，`looptime`就是小球总共移动的次数，当他等于0，程序终止，进入返回dos系统模块；不等于0，重复弹射操作。

```
dec byte[looptime]
jz end
jmp loop1
```

## 返回dos系统

这部分就是简单的调用一下dos中断使用4ch作为功能号

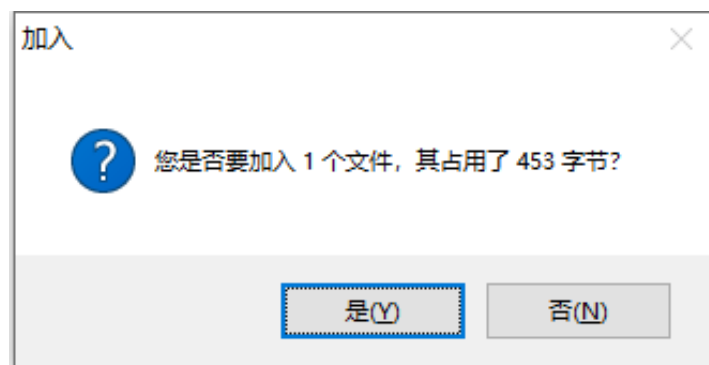
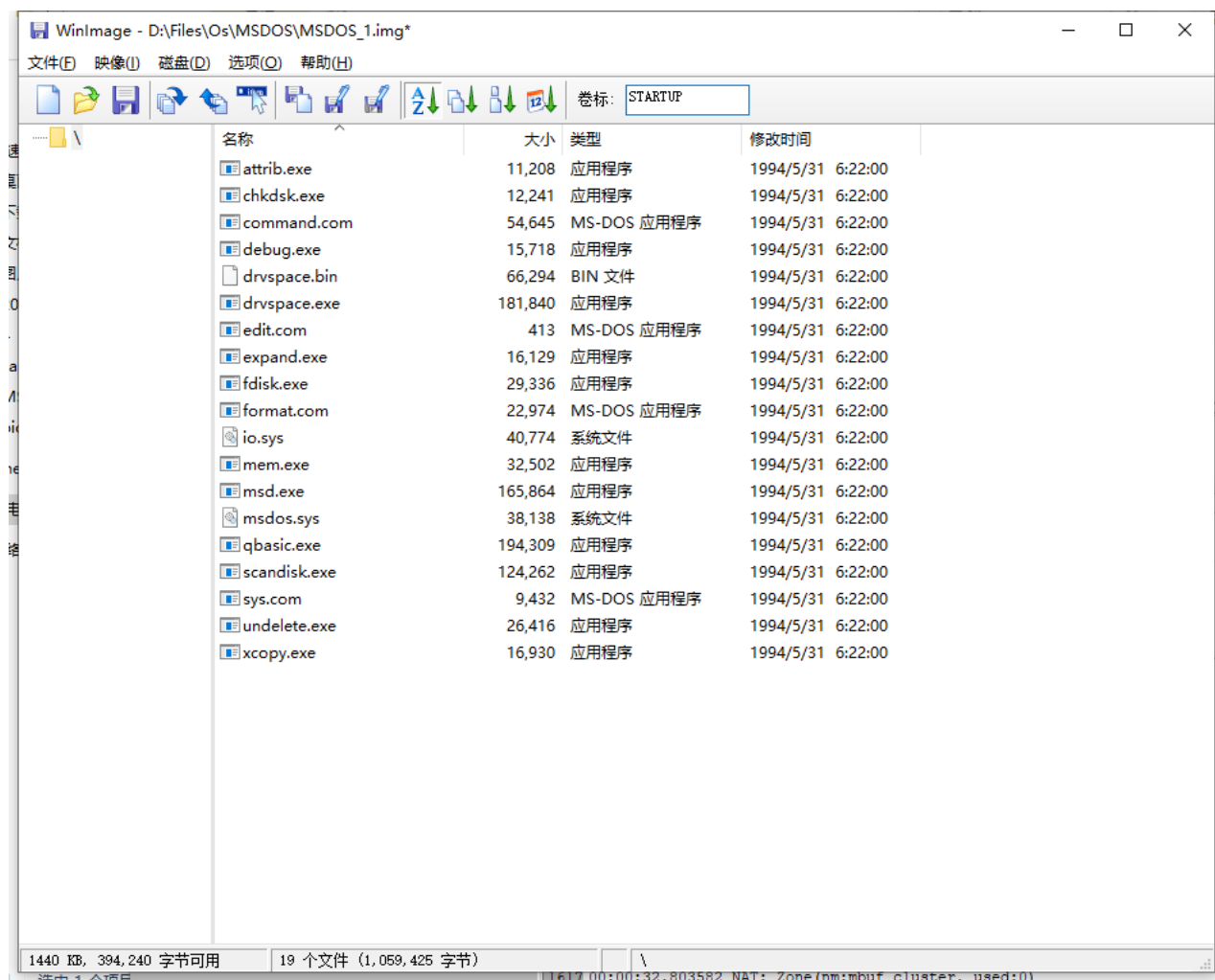
```
end:
mov ax, 4c00h
int 21h
```

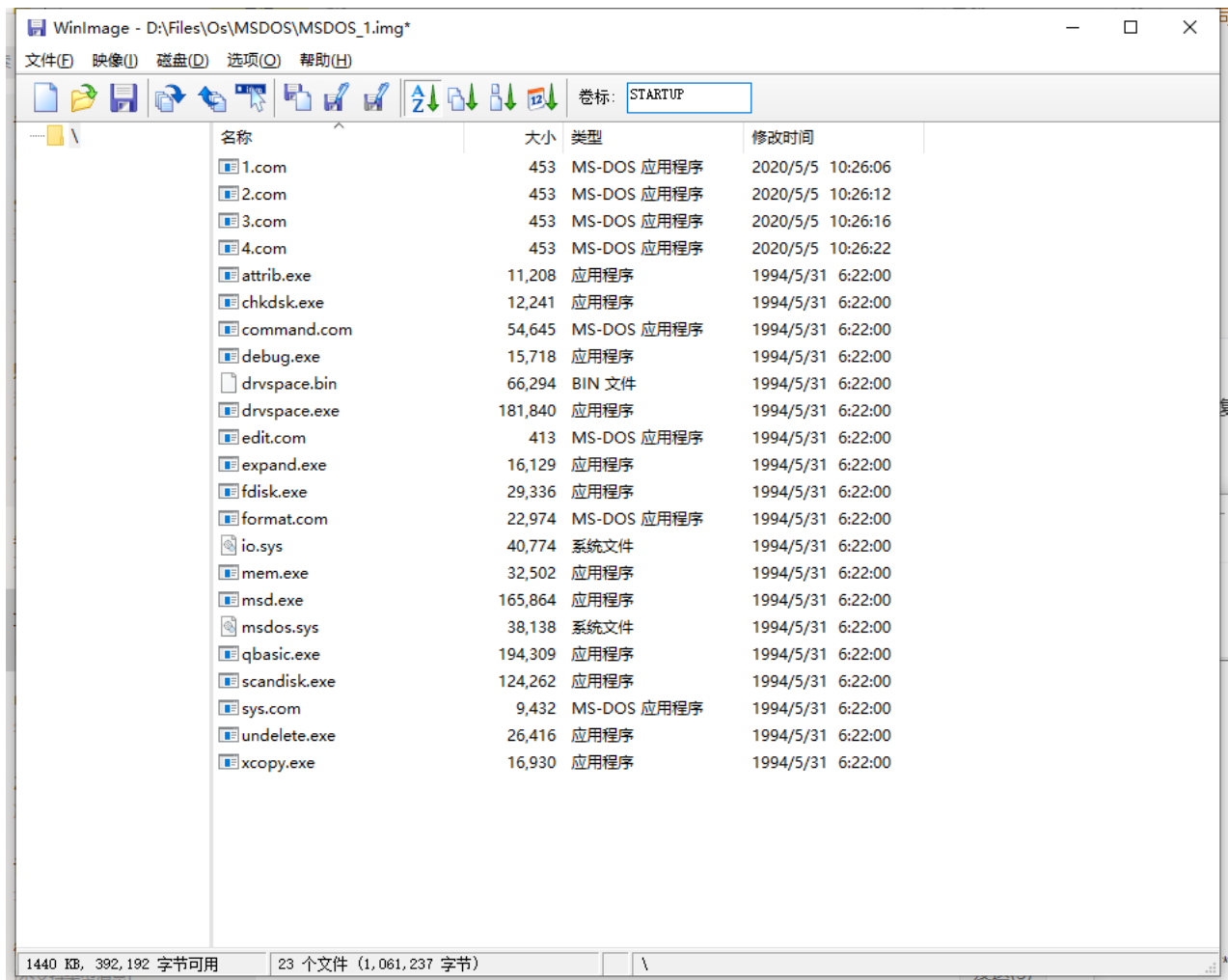
## 运行代码

首先，利用nasm编译4个源代码程序

```
PS D:\Files\Os\Lab2\src\1> nasm -f bin 1.asm -o 1.com
PS D:\Files\Os\Lab2\src\1> nasm -f bin 2.asm -o 2.com
PS D:\Files\Os\Lab2\src\1> nasm -f bin 3.asm -o 3.com
PS D:\Files\Os\Lab2\src\1> nasm -f bin 4.asm -o 4.com
```

紧接着，利用winimage软件把这4个程序导入dos系统。





进入dos系统查看

```

COMMAND  COM      54,645  05-31-94   6:22a
DRUSPACE BIN      66,294  05-31-94   6:22a
ATTRIB   EXE      11,208  05-31-94   6:22a
CHKDSK   EXE      12,241  05-31-94   6:22a
DEBUG    EXE      15,718  05-31-94   6:22a
EXPAND   EXE      16,129  05-31-94   6:22a
FDISK    EXE      29,336  05-31-94   6:22a
FORMAT   COM      22,974  05-31-94   6:22a
MEM       EXE      32,502  05-31-94   6:22a
SYS       COM        9,432  05-31-94   6:22a
EDIT     COM         413  05-31-94   6:22a
QBASIC   EXE     194,309  05-31-94   6:22a
SCANDISK EXE     124,262  05-31-94   6:22a
XCOPY    EXE      16,930  05-31-94   6:22a
DRUSPACE EXE     181,840  05-31-94   6:22a
MSD      EXE     165,864  05-31-94   6:22a
UNDELETE EXE      26,416  05-31-94   6:22a
1        COM         453  05-05-20  10:26a
2        COM         453  05-05-20  10:26a
3        COM         453  05-05-20  10:26a
4        COM         453  05-05-20  10:26a
      21 file(s)      982,325 bytes
                        392,192 bytes free

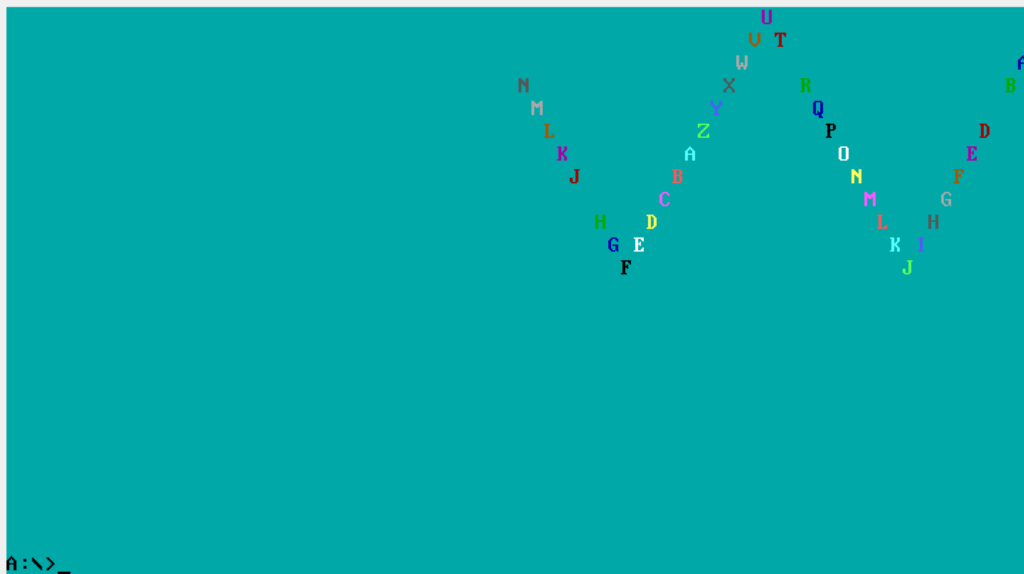
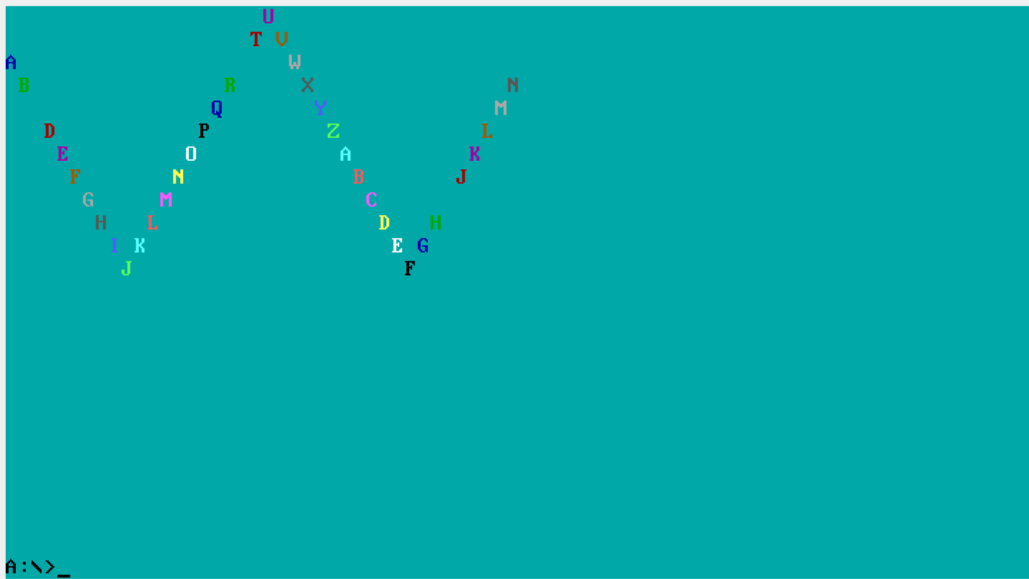
A:\>

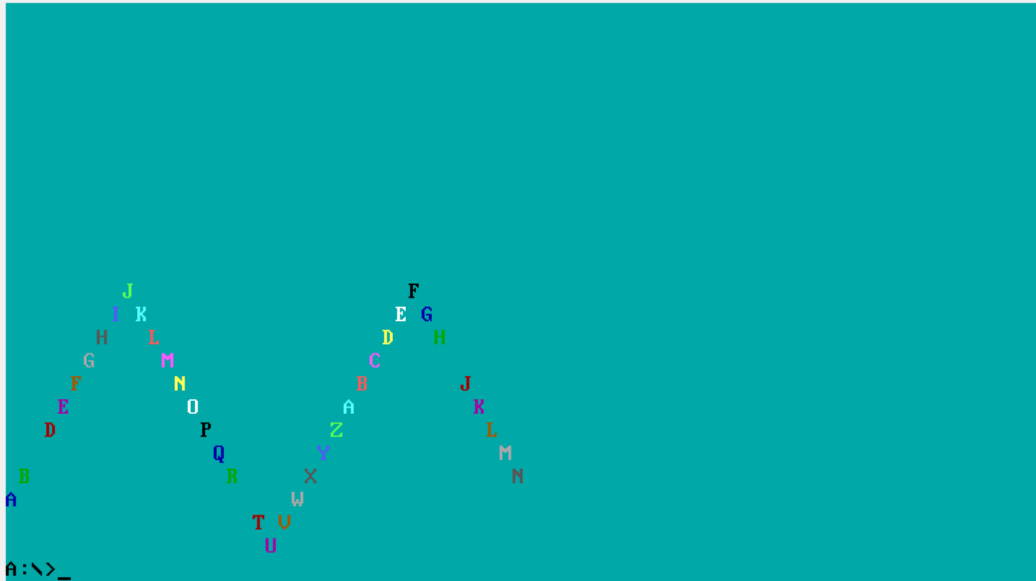
```

从上图可以看到，4个用户程序已经成功导入dos系统了。



## 4个程序运行截图





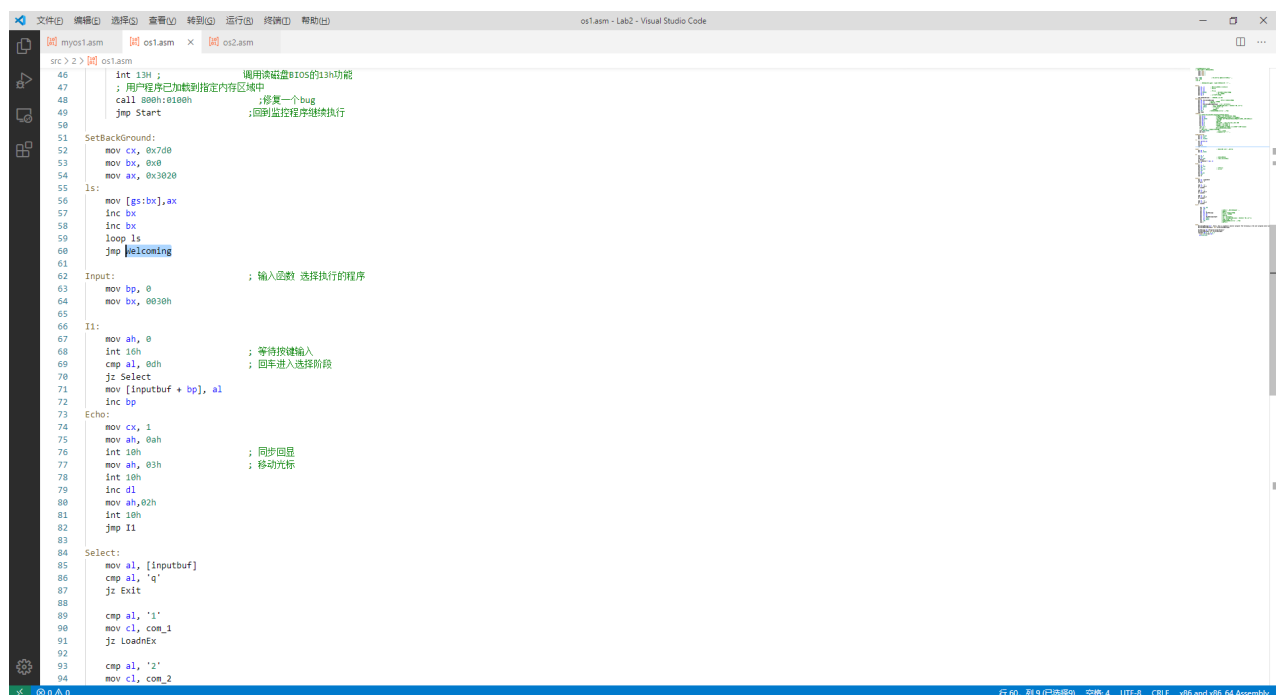
同时，从上面可以看出，用户程序在执行完以后，成功地把控制交回给dos系统了。

## (2) 可以加载用户程序的监控程序

具体代码见压缩里的文件夹2

## 代码的编写

这里我采用的是Visual Studio Code



## 代码的编译

```
nasm -f <format> <filename> [-o output]
```

```
PS D:\Files\0s\Lab2\src\2> nasm -f bin os1.asm -o os1.img
```

## 将监控程序复制到软盘映像文件

由于要在软盘里运行监控程序，我选择使用winhex软件，将刚刚编译成功的文件复制到实验一制作好的第三个空的软盘映像文件里。

Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	10	11	12	13	14	15	16	17	18	19	1A	1B	1C	1D	1E	1F	ANSI ASCII	
00000000	8C	C8	8E	D8	8E	C0	8E	D0	B8	00	B8	8E	8E	BC	00	00	EB	32	B0	7F	7C	8C	D8	8E	C0	B9	89	00	B8	01	13	BB	GEZA2D, Ze4 .GEZA4h, »	
00000020	31	00	B6	00	B2	00	CD	10	EB	2C	8C	8E	C0	BB	00	00	81	B4	02	B0	01	B2	00	B6	00	B5	00	CD	13	9A	00	01	1 q ' i e,GEZA' : ' q p i s	
00000040	00	08	EB	BC	B9	D0	07	BB	00	00	B8	20	00	B5	89	07	43	43	E2	F9	FE	BC	BD	00	00	BB	30	00	B4	00	CD	16	e4'D » , Oek CCAe4h »O ' i	
00000060	3C	0D	74	18	88	8E	4F	7D	45	B9	01	00	B4	0A	CD	10	B4	03	00	10	FE	BC	B4	02	CD	10	EB	B0	A0	4F	7D	3C	< t 'tO'E' ' i ' i pA' i eà O)<	
00000080	71	74	18	3C	31	B1	03	74	A1	3C	32	B1	04	74	9B	3C	33	B1	05	74	95	3C	34	B1	06	74	8F	B4	03	CD	10	FE	< t< i t; <2i t><3i t> <4i t> i p	
000000A0	C6	B2	00	BD	40	7D	8C	D8	8E	C0	89	0F	00	B8	01	13	BB	31	00	CD	10	EB	FE	20	72	20	48	65	6C	63	FD	2C	20	è4@GEZA' , »I i eþ Hello,
000000C0	74	68	69	73	54	69	73	20	6F	75	79	68	6C	61	6E	27	73	20	6D	6F	6E	69	74	6F	72	70	70	72	6F	67	72	61	this is ouyhan's monitor progra	
000000E0	6D	21	20	54	68	65	20	66	6F	75	79	68	6F	77	69	6E	67	20	69	73	20	74	68	65	20	75	73	65	72	20	72	6F	m! The following is the user pro	
00000100	67	72	61	6D	73	20	77	68	69	63	68	20	79	6F	75	20	63	61	6E	20	75	73	65	6A	20	0D	0A	31	7E	63	6F	6D	grams which you can use: 1.ccm	
00000120	20	20	20	32	2E	63	6F	6D	20	20	20	33	2E	63	6F	6D	20	20	20	34	2E	63	6F	6D	0A	00	0A	0F	7E	31	3E	20	2.ccm 3.ccm 4.ccm osl>	
00000140	5B	45	78	69	74	65	64	20	4E	6F	72	6D	6C	79	5D	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	[Exited Normly]	
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000180	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
000001A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
000001C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
000001E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA	U*	
00000200	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000220	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000240	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000260	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000280	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
000002A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
000002C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
000002E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000300	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000320	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000340	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000360	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000380	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
000003A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
000003C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
000003E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00		
00000400	8C	C8	8E	C0	8E	D8	8E	C0	BB	00	B8	8E	8E	C6	C6	BD	02	41	B4	02	BA	00	18	CD	10	E9	84	01	FF	0E	B4	02	GEZA2DZA, Ze4 4 A' ' i é, y ' uG' PÄY q uG' q D' :	
00000420	75	FA	C7	06	B4	02	50	C3	F8	0E	B6	02	75	EE	C7	06	B4	02	53	C7	06	B6	02	44	02	B0	01	PA	06	B8	02	00	t' : , tV' : , '	

第一个扇区是监控程序，第三个扇区是用户程序。

复制的方式比较手工，首先打开要复制的内容，全选文件，再按Ctrl-C进行复制

[illegible]

打开软盘运行文件，选择要复制的首地址，再按Ctrl-B写入

[illegible]

## 将用户程序复制到软盘映像文件

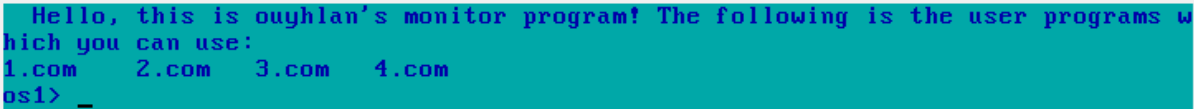
方式如上

[illegible]

4个用户程序都按这种办法做。

## 使用虚拟机运行

开始是一些提示信息，接下来有个输入提示符



输入1.com运行第一个用户程序

```
Hello, this is oughlan's monitor program! The following is the user programs w  
hich you can use:  
1.com 2.com 3.com 4.com  
os1> 1.com
```



剩下的3个程序运行效果类似，我将这个过程放到了程序运行截屏文件里（1.mp3）。

系统的退出（q）：

```
Hello, this is oughlan's monitor program! The following is the user programs w  
hich you can use:  
1.com    2.com    3.com    4.com  
os1> q  
[Exited Normly]
```

### (3) 反复接受命令执行用户程序

按照平时我们运行程序习惯，我设计的命令就是输入程序名字，即可运行对应的程序。

如同（2）中的截图，在完成第一个程序以后，监控程序可以执行第二个用户程序。但比较不直观一点在于，每次执行完一个程序以后，我会刷新一下显存，把刚刚执行的痕迹刷新掉。这样做，就很难看出是在按一定顺序执行不同的程序，详情可以看实验录屏文件。

```
Hello, this is oughlan's monitor program! The following is the user programs w  
hich you can use:  
1.com    2.com    3.com    4.com  
os1> _
```

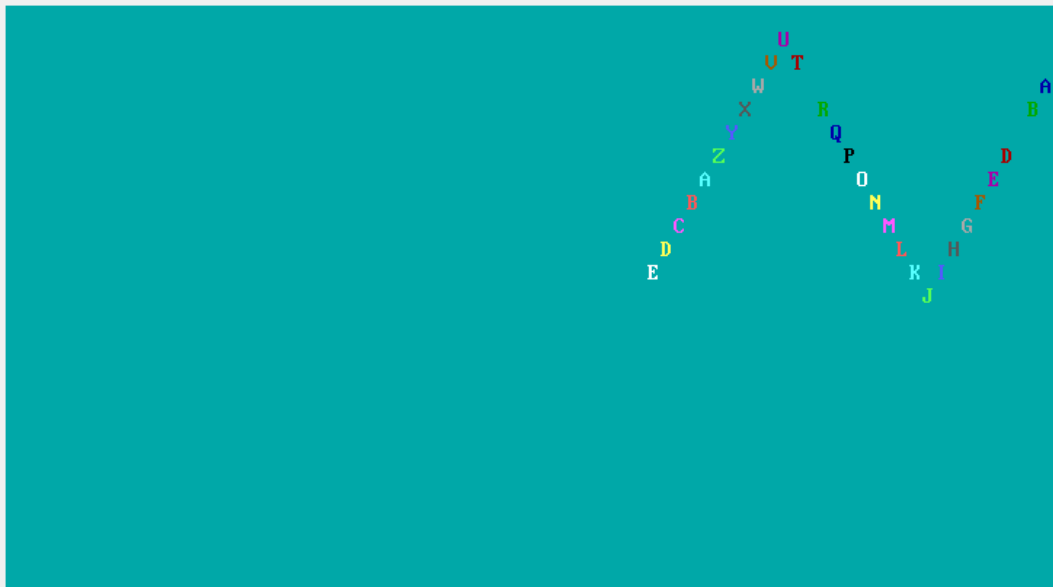
```
Hello, this is oughlan's monitor program! The following is the user programs w  
hich you can use:  
1.com 2.com 3.com 4.com  
os1> 1.com
```





```
    Hello, this is oughlan's monitor program! The following is the user programs w  
hich you can use:  
1.com    2.com    3.com    4.com  
os1> _
```

```
    Hello, this is oughlan's monitor program! The following is the user programs w  
hich you can use:  
1.com    2.com    3.com    4.com  
os1> 2.com_  
_
```



```
Hello, this is oughlan's monitor program! The following is the user programs w  
hich you can use:  
1.com 2.com 3.com 4.com  
os1> _
```

```

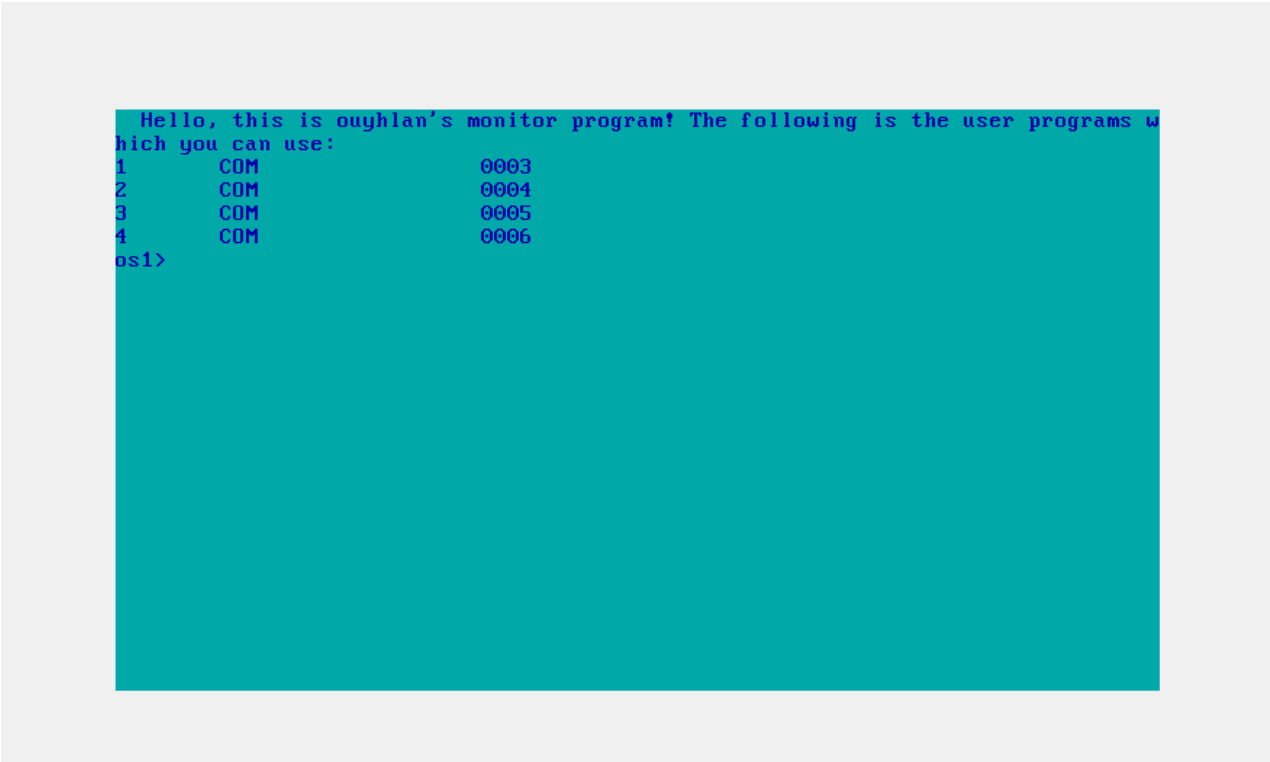
Hello, this is oughlan's monitor program! The following is the user programs w
hich you can use:
1.com    2.com    3.com    4.com
os1> 3.com_

```

[illegible]

## 显示表格信息

与（3）不同，监控程序在开机的时候就会显示表格里面的所有信息



前11个字符是程序名（8个字符是程序的名字，最后3个字符是程序的后缀名）。最后4个字节，则是程序的扇区号。

## （5）完善自己的软件项目管理目录

```
PS D:\Files\Os> tree
卷 新加卷 的文件夹 PATH 列表
卷序列号为 2461-717C
D:.
|-- Lab1
|   |-- 18340133_欧阳浩岚_实验一_v0
|   |   |-- src
|   |   |-- video
|   |-- PC1
|   |   |-- Logs
|   |-- pic
|   |-- src
|   |-- test
|-- Lab2
|   |-- ini
|   |-- pic
|   |-- src
|   |   |-- 1
|   |   |-- 2
|   |   |-- 4
|   |-- test2
|-- MSDOS
|   |-- Logs
|-- Os1
|   |-- Logs
```

这是我的实验文件夹的目录树结构。Lab1和Lab2文件夹分别是我两次实验的文件夹。MSDOS是dos系统虚拟机文件夹，Os1则是我用来调试原型操作系统所采用的软盘映像文件管理文件夹。而在Lab2文件夹下，ini文件存放一些虚拟机配置文件，例如bochs的bochsrc文件，pic文件管理实验过程截图，src则是源代码管理文件夹，test2文件夹用于写一些小代码测试效果。

总体上看，整个软件项目管理目录已经成型了。

## 实验中遇到的问题

### (1) 老师给的例程里存在有问题

老师给出的监控程序开头是org 100h，但是监控程序实际上是放在引导扇区里的，所以应该是org 7c00h。

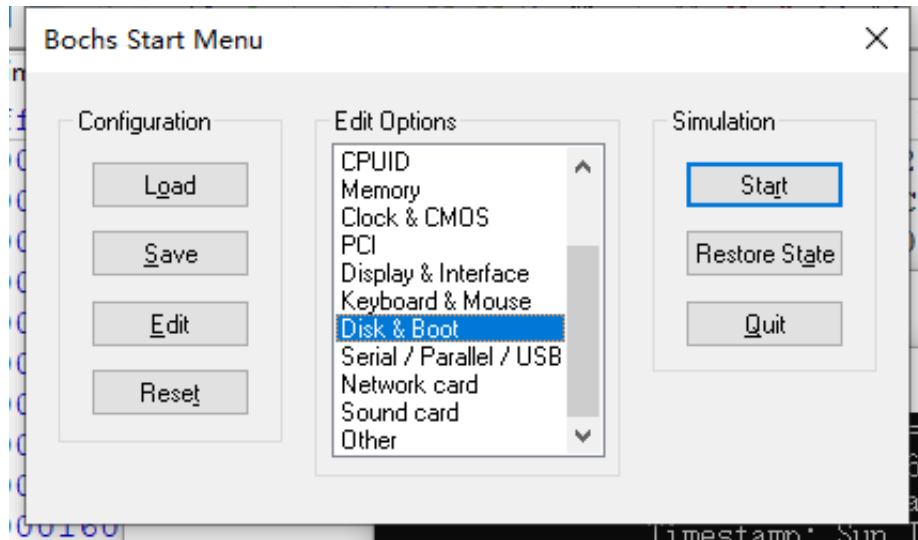
其次，老师例程给的跳转语句是jmp 8100h。但com格式的程序用的是org 100h，实际上，这条跳转语句应该写成800h:100h。

## (2) 汇编程序出现bug

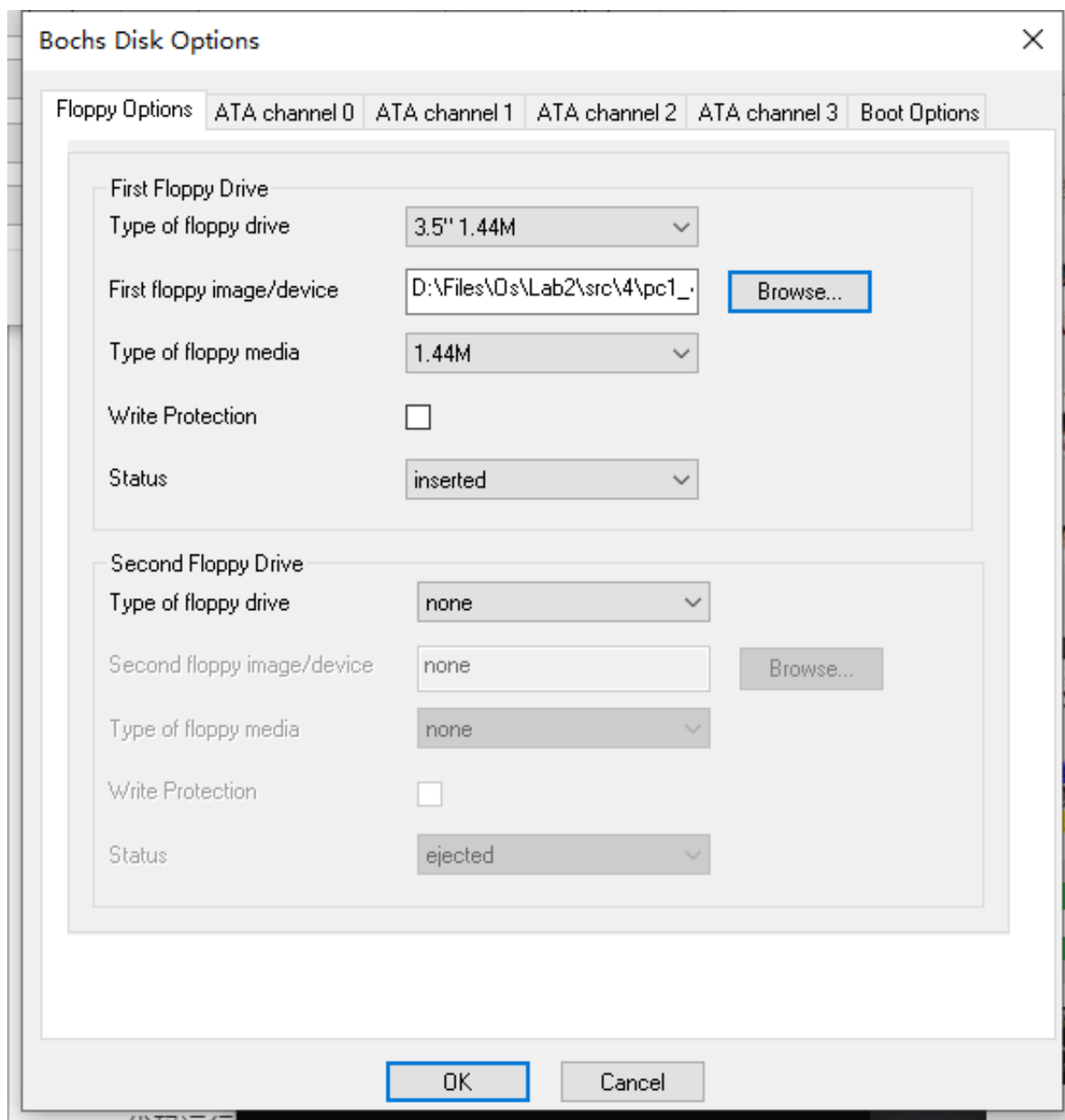
由于是第一次调试汇编程序，比较不熟练，遇到了很多坑点，下面是我的调试的经验

### 使用bochs调试的方法

打开bochsdbg.exe



配置软盘映像文件



配置后启动虚拟机

```
Bochs for Windows - Console
000000000000i [CPU0 ]      rdtscp
000000000000i [CPU0 ]      ffxsr
000000000000i [CPU0 ]      xapic
000000000000i [PLUGIN] reset of 'pci' plugin device by virtual method
000000000000i [PLUGIN] reset of 'pci2isa' plugin device by virtual method
000000000000i [PLUGIN] reset of 'cmos' plugin device by virtual method
000000000000i [PLUGIN] reset of 'dma' plugin device by virtual method
000000000000i [PLUGIN] reset of 'pic' plugin device by virtual method
000000000000i [PLUGIN] reset of 'pit' plugin device by virtual method
000000000000i [PLUGIN] reset of 'vga' plugin device by virtual method
000000000000i [PLUGIN] reset of 'floppy' plugin device by virtual method
000000000000i [PLUGIN] reset of 'acpi' plugin device by virtual method
000000000000i [PLUGIN] reset of 'hpet' plugin device by virtual method
000000000000i [PLUGIN] reset of 'ioapic' plugin device by virtual method
000000000000i [PLUGIN] reset of 'keyboard' plugin device by virtual method
000000000000i [PLUGIN] reset of 'harddrv' plugin device by virtual method
000000000000i [PLUGIN] reset of 'pci_ide' plugin device by virtual method
000000000000i [PLUGIN] reset of 'unmapped' plugin device by virtual method
000000000000i [PLUGIN] reset of 'biosdev' plugin device by virtual method
000000000000i [PLUGIN] reset of 'speaker' plugin device by virtual method
000000000000i [PLUGIN] reset of 'extfpurq' plugin device by virtual method
000000000000i [PLUGIN] reset of 'parallel' plugin device by virtual method
000000000000i [PLUGIN] reset of 'serial' plugin device by virtual method
000000000000i [PLUGIN] reset of 'gameport' plugin device by virtual method
000000000000i [PLUGIN] reset of 'iodebug' plugin device by virtual method
000000000000i [PLUGIN] reset of 'usb_uhci' plugin device by virtual method
000000000000i [      ] set SIGINT handler to bx_debug_ctrlc_handler
Next at t=0
(0) [0x0000fffff0] f000:fff0 (unk. ctxt): jmpf 0xf000:e05b      ; ea5be000f0
<bochs:1>
```

在0x7c00处设置断点

```
Bochs for Windows - Console
00001052065i [BIOS] region 4: 0x0000c000
00001054107i [BIOS] PCI: bus=0 devfn=0x0a: vendor_id=0x8086 device_id=0x7020 class=0x0c03
00001054320i [UHCI] BAR #4: i/o base address = 0xc020
00001054936i [BIOS] region 4: 0x0000c020
00001055070i [UHCI] new IRQ line = 9
00001056992i [BIOS] PCI: bus=0 devfn=0x0b: vendor_id=0x8086 device_id=0x7113 class=0x0680
00001057236i [ACPI] new IRQ line = 11
00001057250i [ACPI] new IRQ line = 9
00001057277i [ACPI] new PM base address: 0xb000
00001057291i [ACPI] new SM base address: 0xb100
00001057319i [PCI] setting SMRAM control register to 0x4a
00001221412i [CPU0] Enter to System Management Mode
00001221412i [CPU0] enter_system_management_mode: temporary disable VMX while in SMM mode
00001221422i [CPU0] RSM: Resuming from System Management Mode
00001385443i [PCI] setting SMRAM control register to 0x0a
00001412159i [BIOS] MP table addr=0x000f9e90 MPC table addr=0x000f9dc0 size=0xc8
00001414030i [BIOS] SMBIOS table addr=0x000f9ea0
00001416216i [BIOS] ACPI tables: RSDP addr=0x000f9fd0 ACPI DATA addr=0x01ff0000 size=0xff8
00001419463i [BIOS] Firmware waking vector 0x1ff00cc
00001421943i [PCI] i440FX PMC write to PAM register 59 (TLB Flush)
00001422666i [BIOS] bios_table_cur_addr: 0x000f9ff4
00001551537i [VBIOS] VGABios $Id: vgabios.c 226 2020-01-02 21:36:23Z vruppert $
00001551608i [BXVGA] VBE known Display Interface b0c0
00001551640i [BXVGA] VBE known Display Interface b0c5
00001554283i [VBIOS] VBE Bios $Id: vbe.c 228 2020-01-02 23:09:02Z vruppert $
00014040188i [BIOS] Booting from 0000:7c00
(0) Breakpoint 1, 0x0000000000007c00 in ?? ()
Next at t=14040243
(0) [0x0000000000007c00] 0000:7c00 (unk. ctxt): mov ax, cs ; 8cc8
<bochs:3>
```

等虚拟机加载了引导程序以后，反汇编引导程序代码以作调试用

```
Bochs for Windows - Console
Next at t=14040243
(0) [0x0000000000007c00] 0000:7c00 (unk. ctxt): mov ax, cs ; 8cc8
<bochs:3> u 0x7c00 0x7e00
000000000000007c00: ( ) : mov ax, cs ; 8cc8
000000000000007c02: ( ) : mov ds, ax ; 8ed8
000000000000007c04: ( ) : mov es, ax ; 8ec0
000000000000007c06: ( ) : mov ss, ax ; 8ed0
000000000000007c08: ( ) : mov ax, 0xb800 ; b800b8
000000000000007c0b: ( ) : mov gs, ax ; 8ee8
000000000000007c0d: ( ) : mov sp, 0x0000 ; bc0000
000000000000007c10: ( ) : jmp .+50 (0x00007c44) ; eb32
000000000000007c12: ( ) : mov bp, 0x7d12 ; bd127d
000000000000007c15: ( ) : mov ax, ds ; 8cd8
000000000000007c17: ( ) : mov es, ax ; 8ec0
000000000000007c19: ( ) : mov cx, 0x0064 ; b96400
000000000000007c1c: ( ) : mov ax, 0x1301 ; b80113
000000000000007c1f: ( ) : mov bx, 0x0031 ; bb3100
000000000000007c22: ( ) : mov dh, 0x00 ; b600
000000000000007c24: ( ) : mov dl, 0x00 ; b200
000000000000007c26: ( ) : int 0x10 ; cd10
000000000000007c28: ( ) : jmp .+44 (0x00007c56) ; eb2c
000000000000007c2a: ( ) : mov ax, cs ; 8cc8
000000000000007c2c: ( ) : mov es, ax ; 8ec0
000000000000007c2e: ( ) : mov bx, 0x8100 ; bb0081
000000000000007c31: ( ) : mov ah, 0x02 ; b402
000000000000007c33: ( ) : mov al, 0x01 ; b001
000000000000007c35: ( ) : mov dl, 0x00 ; b200
000000000000007c37: ( ) : mov dh, 0x00 ; b600
000000000000007c39: ( ) : mov ch, 0x00 ; b500
000000000000007c3b: ( ) : int 0x13 ; cd13
```

从上面可以看出，查看反汇编代码是比较辛苦的，所以我把这些结果copy到了vscode的一个文件，一边单步调试一边看代码。

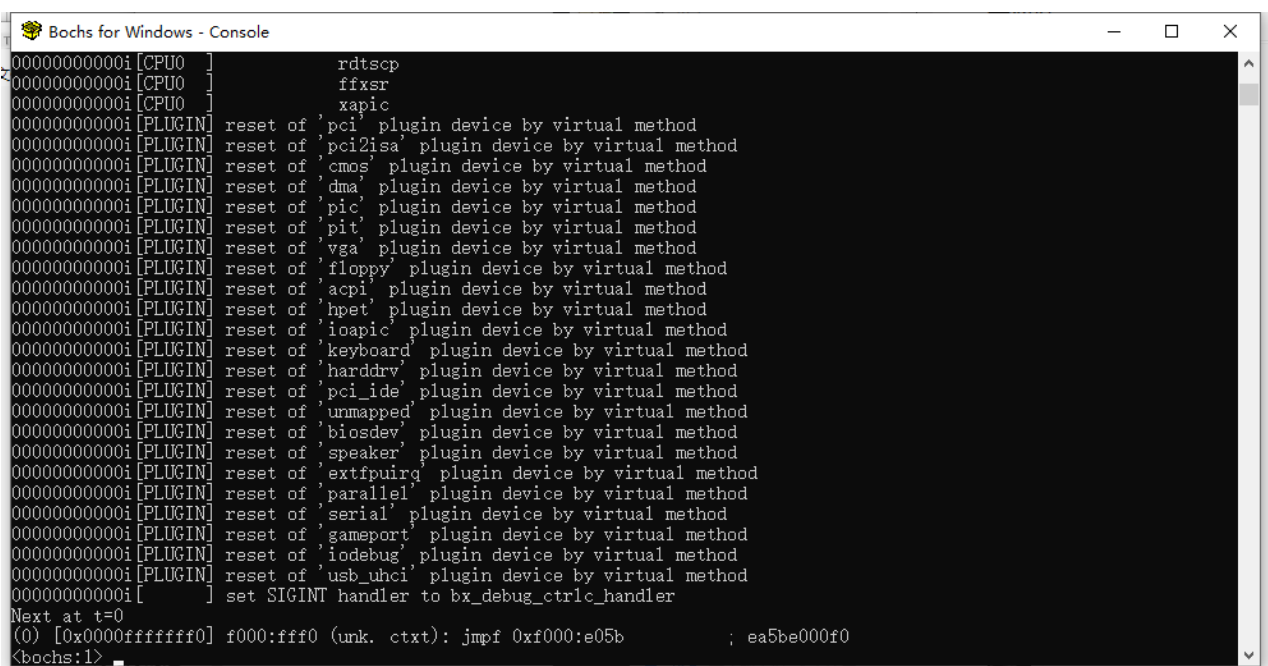
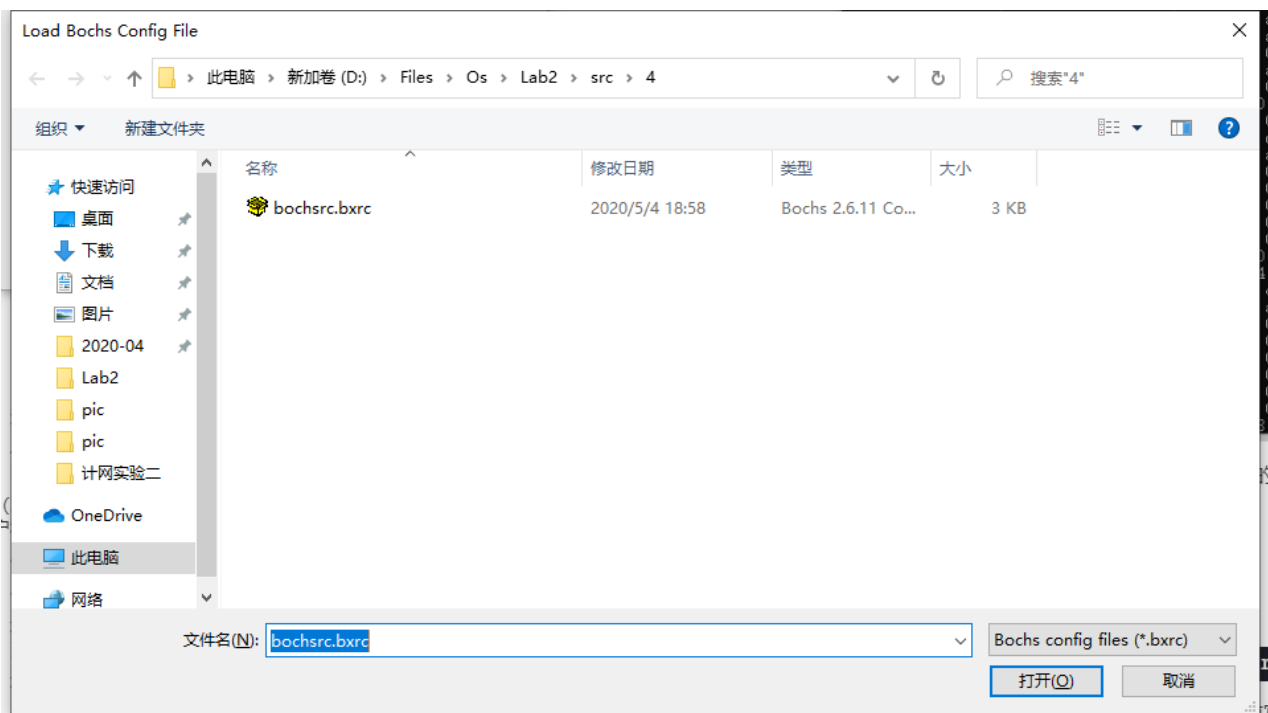
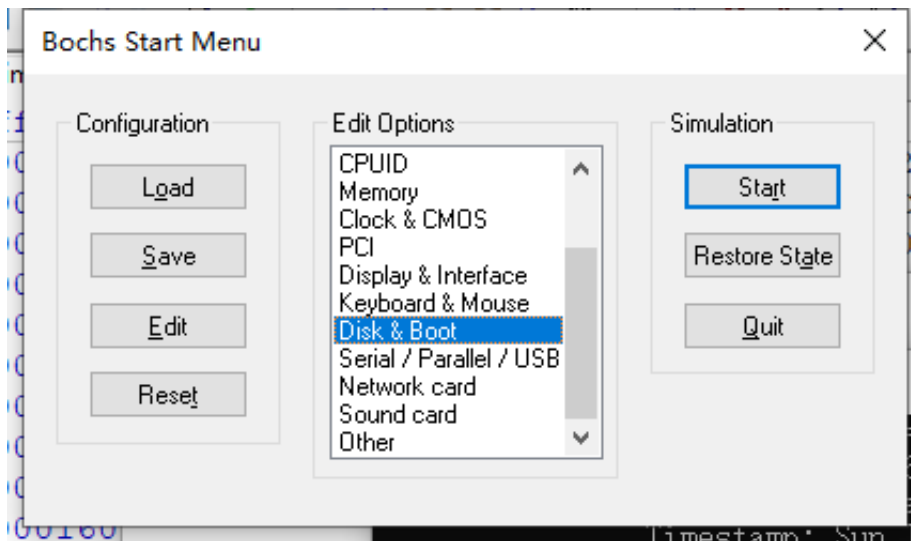
## 排错的过程

首先，我是先编译存在错误的代码

```
PS D:\Files\Os\Lab2\src\2> nasm -f bin os1.asm -o os1.img
```

接着，打开bochsdbg.exe，选择load加载之前写好的bochsrc配置文件





启动后，由于我其实并不知道代码具体在内存的哪个位置，我将第一个断点设在0x7c00，然后进行运行

```
Bochs for Windows - Console
00001052065i [BIOS] region 4: 0x0000c000
00001054107i [BIOS] PCI: bus=0 devfn=0x0a: vendor_id=0x8086 device_id=0x7020 class=0x0c03
00001054320i [UHCI] BAR #4: i/o base address = 0xc020
00001054936i [BIOS] region 4: 0x0000c020
00001055070i [UHCI] new IRQ line = 9
00001056992i [BIOS] PCI: bus=0 devfn=0x0b: vendor_id=0x8086 device_id=0x7113 class=0x0680
00001057236i [ACPI] new IRQ line = 11
00001057250i [ACPI] new IRQ line = 9
00001057277i [ACPI] new PM base address: 0xb000
00001057291i [ACPI] new SM base address: 0xb100
00001057319i [PCI] setting SMRAM control register to 0x4a
00001221412i [CPU0] Enter to System Management Mode
00001221412i [CPU0] enter_system_management_mode: temporary disable VMX while in SMM mode
00001221422i [CPU0] RSM: Resuming from System Management Mode
00001385443i [PCI] setting SMRAM control register to 0x0a
00001412159i [BIOS] MP table addr=0x000f9e90 MPC table addr=0x000f9dc0 size=0xc8
00001414030i [BIOS] SMBIOS table addr=0x000f9ea0
00001416216i [BIOS] ACPI tables: RSDP addr=0x000f9fd0 ACPI DATA addr=0x01ff0000 size=0xff8
00001419463i [BIOS] Firmware waking vector 0x1ff00cc
00001421943i [PCI] i440FX PMC write to PAM register 59 (TLB Flush)
00001422666i [BIOS] bios_table_cur_addr: 0x000f9ff4
00001551537i [VBIOS] VGBios $Id: vgabios.c 226 2020-01-02 21:36:23Z vruppert $
00001551608i [BXVGA] VBE known Display Interface b0c0
00001551640i [BXVGA] VBE known Display Interface b0c5
00001554283i [VBIOS] VBE Bios $Id: vbe.c 228 2020-01-02 23:09:02Z vruppert $
00014040188i [BIOS] Booting from 0000:7c00
(0) Breakpoint 1, 0x0000000000007c00 in ?? ()
Next at t=14040243
(0) [0x0000000000007c00] 0000:7c00 (unk. ctxt): mov ax, cs ; 8cc8
<bochs:3>
```

到达0x7c00后，其实我的监控程序已经加载到内存里了，只要反汇编他们就可以了。

```
Bochs for Windows - Console
Next at t=14040243
(0) [0x0000000000007c00] 0000:7c00 (unk. ctxt): mov ax, cs ; 8cc8
<bochs:3> u 0x7c00 0x7e00
0000000000007c00: (    ): mov ax, cs ; 8cc8
0000000000007c02: (    ): mov ds, ax ; 8ed3
0000000000007c04: (    ): mov es, ax ; 8ec0
0000000000007c06: (    ): mov ss, ax ; 8ed0
0000000000007c08: (    ): mov ax, 0xb300 ; b800b8
0000000000007c0b: (    ): mov gs, ax ; 8ee3
0000000000007c0d: (    ): mov sp, 0x0000 ; bc0000
0000000000007c10: (    ): jmp .+50 (0x00007c44) ; eb32
0000000000007c12: (    ): mov bp, 0x7d12 ; bd127d
0000000000007c15: (    ): mov ax, ds ; 8cd8
0000000000007c17: (    ): mov es, ax ; 8ec0
0000000000007c19: (    ): mov cx, 0x0064 ; b96400
0000000000007c1c: (    ): mov ax, 0x1301 ; b80113
0000000000007c1f: (    ): mov bx, 0x0031 ; bb3100
0000000000007c22: (    ): mov dh, 0x00 ; b600
0000000000007c24: (    ): mov dl, 0x00 ; b200
0000000000007c26: (    ): int 0x10 ; cd10
0000000000007c28: (    ): jmp .+44 (0x00007c56) ; eb2c
0000000000007c2a: (    ): mov ax, cs ; 8cc8
0000000000007c2c: (    ): mov es, ax ; 8ec0
0000000000007c2e: (    ): mov bx, 0x8100 ; bb0081
0000000000007c31: (    ): mov ah, 0x02 ; b402
0000000000007c33: (    ): mov al, 0x01 ; b001
0000000000007c35: (    ): mov dl, 0x00 ; b200
0000000000007c37: (    ): mov dh, 0x00 ; b600
0000000000007c39: (    ): mov ch, 0x00 ; b500
0000000000007c3b: (    ): int 0x13 ; cd13
```

接着就是一步步看寄存器的值和下一条语句。

```
Bochs for Windows - Console
0000000000007de2: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007de3: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007de4: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007de5: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007de6: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007de7: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007de8: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007de9: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007dea: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007deb: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007dec: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007ded: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007dee: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007def: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007df0: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007df1: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007df2: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007df3: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007df4: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007df5: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007df6: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007df7: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007df8: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007dfa: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007dfb: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007dfd: (    ): add byte ptr ds:[bx+1], al ; 0000
0000000000007dfe: (    ): push bp
0000000000007dff: (    ): stosb byte ptr es:[di], al ; aa
<bochs:5> b 0x7cfa
<bochs:0> c
```

```
Bochs for Windows - Display
CTRL + 3rd button enables mouse | IPS: 28.863M | NUM: CAPS: SCRL:
```

```
Bochs for Windows - Console
0000000000007df8: (                ): add byte ptr ds:[bx+si], al ; 0000
0000000000007dfa: (                ): add byte ptr ds:[bx+si], al ; 0000
0000000000007dfc: (                ): add byte ptr ds:[bx+si], al ; 0000
0000000000007dfe: (                ): push bp                      ; 55
0000000000007dff: (                ): stosb byte ptr es:[di], al ; aa
<bochs:5> b 0x7cfa
<bochs:6> c
(0) Breakpoint 1, 0x0000000000007c00 in ?? ()
Next at t=2452372617
(0) [0x0000000000007c00] 0000:7c00 (unk. ctxt): mov ax, cs          ; 8cc8
<bochs:7> r
rax: 00000000_0000384e
rbx: 00000000_000002ce
rcx: 00000000_0009000d
rdx: 00000000_00000000
rsp: 00000000_00000000
rbp: 00000000_00000005
rsi: 00000000_000e0000
rdi: 00000000_0000ffac
r8 : 00000000_00000000
r9 : 00000000_00000000
r10: 00000000_00000000
r11: 00000000_00000000
r12: 00000000_00000000
r13: 00000000_00000000
r14: 00000000_00000000
r15: 00000000_00000000
rip: 00000000_00007c00
eflags 0x00000046: id vip vif ac vm rf nt IOPL=0 of df if tf sf ZF af PF cf
<bochs:8>
000000007dfc: (                ): add byte ptr ds:[bx+si], al ; 0000
```

## 遇到的bug

### (1) call 800:100h

这条语句初看很正常，但是在调试中，我发现他实际跳转地址是0x320:0x100，而不是预期表的0x800:0x100。

思考后，我发现原来800使用了十进制数，而不是16进制数。

将它改成call 800h:100h就正确运行了。

### (2) 交互界面的光标不正常

这个异常没有截图，其情形类似于后面的输出盖住了前面的输出，导致读起来信息不连贯。

我没有想到什么好的解决办法，只能通过指定绝对位置（可以通过调整入口参数dx实现）来达到想要的效果。

## 实验总结

这是我在操作系统实验课的第二个实验，总体来说还是比较顺利，也让我对这个实验课越来越感兴趣了。比较让我吃惊一点是，我可以通过修改少量代码，再利用winimage软件，就把我实验一写的代码放到dos系统下运行，这是让我觉得比较神奇的一点。其次，本次实验里，我也开始初步接触到使用bochs软件调试img文件。虽然整个过程比较艰辛，但是也算一次不错的尝试，起码我已经有能力去调试我的原型操作系统了。

不过，回顾本次实验过程，我觉得有以下几点自己做的还是不好：

- 汇编代码写的太繁琐，代码不够清晰
- 代码模块化也不够好，导致很多重复的代码出现
- 整个代码编译运行调试过程有太多手工操作的地方了，实际上，这些重复的过程完全可以写一个小的脚本程序作为帮助。

对于上面说的第三点，我还要补充一下——我现在整个实验流程是：代码编写->nasm编译->winhex手动复制nasm生成bin文件到软盘映像文件->virtualbox虚拟机运行软盘映像文件。

实际上，我们可以利用一个批处理文件bat，一键完成这些操作，不过现在我还没有把这个批处理文件写出来，不过理论上应该是可行的。

同时，正如同我在实验一里所说，希望自己能够学会使用32位x86汇编语言，也就是在保护模式下编写原型操作系统。这样写出来的程序，就可以在我现在的操作系统（win10）下顺利地运行了吧。