

实验四：异步事件的编程技术

18340225 钟婕

实验目的

- 1、PC系统的中断机制和原理
- 2、理解操作系统内核对异步事件的处理方法
- 3、掌握中断处理编程的方法
- 4、掌握内核中断处理代码组织的设计方法
- 5、了解查询式I/O控制方式的编程方法

实验要求

- 1、知道PC系统的中断硬件系统的原理
- 2、掌握x86汇编语言对时钟中断的响应处理编程方法
- 3、重写和扩展实验三的的内核程序，增加时钟中断的响应处理和键盘中断响应。
- 4、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

实验内容

(1)编写x86汇编语言对时钟中断的响应处理程序：设计一个汇编程序，在一段时间内系统时钟中断发生时，屏幕变化显示信息。在屏幕24行79列位置轮流显示'|'、'/'和'\'(无敌风火轮)，适当控制显示速度，以方便观察效果，也可以屏幕上画框、反弹字符等，方便观察时钟中断多次发生。将程序生成COM格式程序，在DOS或虚拟环境运行。

(2)重写和扩展实验三的的内核程序，增加时钟中断的响应处理和键盘中断响应。，在屏幕右下角显示一个转动的无敌风火轮，确保内核功能不比实验三的程序弱，展示原有功能或加强功能可以工作。

(4)扩展实验三的的内核程序，但不修改原有的用户程序，实现在用户程序执行期间，若触碰键盘，屏幕某个位置会显示"OUCH!OUCH!"。

(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

实验环境与工具

1、实验环境：

- 1) 实验运行环境：Windows10
- 2) 虚拟机软件：DosBox、VirtualBox

2、实验工具

- 1) 汇编编译器：NASM、TASM
- 2) C语言编译器：TCC
- 3) 文本编辑器：Visual Studio 2017
- 4) 软盘操作工具：WinHex
- 5) 调试工具：Bochs

实验方案与过程

• 实验思路与原理

总体思路

本次实验主要是基于时间中断和键盘中断的实现。基于上课学习的内容，可以将本次实验的工作概括为：设计一个自己的时钟中断和键盘中断，并让程序响应执行。当然，具体的实验过程需要考虑许多细节问题，包括：时钟中断触发太快如何解决？键盘中断何时修改与恢复？键盘中断显示字符过快如何解决？等系列实验细节。

实验原理

根据实验内容和要求可知，本次实验是在实验三的基础上，增加MyOS对于中断操作的处理，理解中断机制。根据课堂所学以及资料查询可知，将本人对于中断的理解以及此次实验设计的有关中断的知识、原理阐述如下：

1. 中断技术

中断(interrupt)是指对处理器正常处理过程的打断。中断与异常一样，都是在程序执行过程中的强制性转移，转移到相应的处理程序。

根据中断的特点，可以将中断分成两大类：硬中断与软中断

■ 硬中断

硬中断（外部中断）——由外部（主要是外设[即I/O设备]）的请求引起的中断。

具体有以下几种中断：

- 1) 时钟中断（计时器产生，等间隔执行特定功能）
- 2) I/O中断（I/O控制器产生，通知操作完成或错误条件）
- 3) 硬件故障中断（故障产生，如掉电或内存奇偶校验错误）

■ 软中断

软中断（内部中断）——由指令的执行引起的中断。

中断指令（软中断int n、溢出中断into、中断返回iret、单步中断TF=1）

■ 异常/程序中断

（指令执行结果产生，如溢出、除0、非法指令、越界）

2. 中断处理程序

PC中断的处理过程包含：硬件实现的保护断点现场、执行中断处理程序以及返回到断点接着执行。

■ 硬件实现的保护断点现场

要将标志寄存器FLAGS压栈，然后清除它的IF位和TF位

再将当前的代码段寄存器CS和指令指针寄存器IP压栈

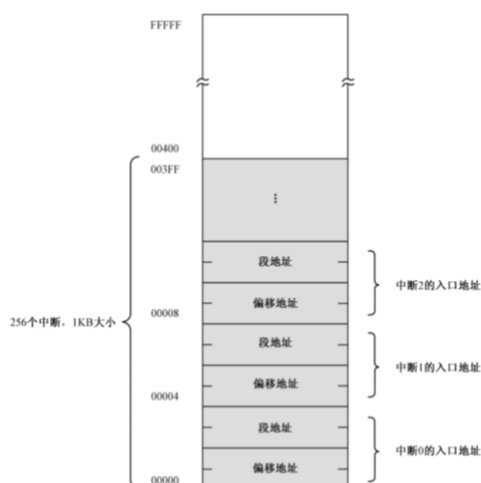
这是由硬件实现的过程，因此在编程中不涉及这方面的工作。

■ 执行中断处理程序

处理器根据已经拿到的中断号，它将该号码乘以4（毕竟每个中断在中断向量表中占4字节），就得到了该中断处理程序入口点在中断向量表中的偏移地址。

中断向量表：

X86计算机在启动的时候，会在内存的低位区（地址为0~1023，大小为1KB）创建含有256个中断向量的中断向量表，**中断向量其实就是中断处理程序的入口地址**，每个中断向量占4个字节，4个字节的具体分配为CS：IP（16位段地址：16位偏移地址），根据X86的小端存储方式，IP偏移地址存储在低字节，CS段地址存储在高字节。具体如下图：



处理器根据中断号在中断向量表中找到该中断号对应的中断处理程序入口点在中断向量表的偏移地址后，从表中依次取出中断程序的偏移地址和段地址，并分别传送到IP和CS。因此，处理器就开始执行CS：IP对应的入口点的中断处理程序了。

■ 返回到断点接着执行

所有中断处理程序的最后一条指令必须是中断返回指令iret。这将导致处理器依次从堆栈中弹出（恢复）IP、CS和FLAGS的原始内容，于是转到主程序接着执行。

至此就完成对于一次中断的响应，并返回到主程序继续执行。

3. 开中断与关中断

■ IF标志位

CPU是否响应在INTR(可屏蔽中断)线上出现的中断请求，取决于标志寄存器FLAGS中的IF标志位的状态值是否为1。可用机器指令STI/CLI置IF标志位为1/0来开/关中断。

■ 8259A芯片

8259A芯片是一种可编程的中断控制器。

在一个8259A芯片中，有如下三个内部寄存器：

- 1) IMR (Interrupt Mask Register, 中断屏蔽寄存器) ——用作过滤被屏蔽的中断
- 2) IRR (Interrupt Request Register, 中断请求寄存器) ——用作暂时放置未被进一步处理的中断

3) ISR (In-Service Register, 在使用中断) ——当一个中断正在被CPU处理时, 此中断被放置在ISR中。

8259A的I/O端口:

每个可编程中断控制器8259A都有两个I/O端口

主8259A所对应的端口地址为20h和21h。

从8259A所对应的端口地址为A0h和A1h。

我们可以通过in/out指令读写这些端口, 来操作这两个中断控制器。

■ STI/CLI指令

当我们进入中断处理程序执行的时候, 此时仍然有可能有其他的中断请求, 为了防止其他中断进入干扰程序的执行, 我们可以选择屏蔽中断。使用指令CLI就可以修改处理器状态字中的中断相关的对应标志寄存器的值, 实现中断屏蔽。如果我们结束了当前中断处理程序的执行时, 需要开启中断, 就可以使用STI指令。

另外, 当我们编写的硬件中断处理程序结束的时候, 我们需要给8259A芯片发送EOI信号来将存储当前正在处理的中断的ISR寄存器的对应位置清零, 来结束中断服务程序。

4. 时间中断

系统时钟的中断号为08H, 连接在主8259A的0号引脚上。计算机每隔一段很短的时间就会触发一次时钟中断, 因此在设计时钟中断的处理程序时, 为了便于视觉观察, 需要进行延时计数响应。

5. 键盘中断

键盘中断的中断号为09h, 键盘中断是由于键盘按键输入触发的。当有按键被按下时, KBC检测到按键按下, 并发出中断请求信号。原来的键盘中断包括通过60号端口读取操作扫描码, 并转换成ASCII码, 将其存入键盘缓冲区的过程。在本次实验中, 就是需要将原来的键盘中断在执行用户程序时, 修改为我们自己编写的键盘中断程序, 在屏幕某个位置显示“OUCHOUCH!”信息。

6. 如何修改中断

根据上述描述, 本人可将本次实验中断机制的实现归纳为:

- 1) 编写中断处理程序
- 2) 修改中断向量表

当然, 对于键盘中断的修改仅限于执行用户程序之时, 因此需要在读取扇区加载用户程序之前才修改中断向量表, 在执行完用户程序返回时, 需要还原中断向量表, 否则会影响正常的输入命令操作!

• 编写一个时钟中断响应的汇编程序

根据实验要求, 需要设计一个汇编程序, 在一段时间内系统时钟中断发生时, 屏幕变化显示信息。在屏幕24行79列位置轮流显示'|'、'/'和'\'(无敌风火轮), 适当控制显示速度, 以方便观察效果, 也可以屏幕上画框、反弹字符等, 方便观察时钟中断多次发生。

根据上述内容与要求, 本人计划在实验读字符反弹用户程序的基础上, 增加对于时钟中断的响应程序, 并将程序编译成二进制COM格式文件即可。如实验原理部分所述, 编写时间中断处理程序, 需要考虑两个方面: 修改中断向量表和编写中断响应程序。

◦ 修改中断向量表

由于时钟中断是08h号中断，因此为了让系统触发时钟中断时能够成功进入我自己编写的响应程序Timer，需要将Timer程序的入口地址赋值给中断向量表的相应位置中。根据实验原理部分的阐述可知，中断向量表位于内存地址0~1023位置处，共1KB大小，每个中断的中断向量占4个字节，高地址存放的是响应程序的段地址，低地址存放的是响应程序的偏移地址。而每个中断的中断响应程序的地址对应在中断向量表的位置位于中断号×4的地址处开始的4个字节。

因此，时钟中断的响应程序的偏移地址存放在0：32处，段地址存放在0：34处。

修改中断向量表的代码如下：

```
xor ax,ax;中断向量表位于0~1023处，段地址为0
mov es,ax
mov word[es:32],Timer;将响应程序的偏移地址赋值给低地址的2个字节
mov word[es:34],cs;将段地址赋值给高地址的2个字节
```

特别注意地是，一定要在程序的开头就更改中断向量表，这样才能使一运行程序触发时钟中断时就成功响应。

○ 编写中断响应程序

首先，对于编写中断响应程序Timer，由于需要在屏幕第24行79列轮流显示四个字符，因此考虑将四个字符分别对应为‘1’、‘2’、‘3’、‘4’四个状态，用一个state变量保存当前状态（即当前需要显示的字符），每次显示一个字符之后，state状态变量自增，形成一个1~4状态的循环过程。接着考虑字符的显示问题，在之前的实验中，我们对于显示字符有两种基本办法：利用0B800h显存段地址显示字符和利用BIOS的10h中断号13h功能号的显示字符串功能调用。在本次实验刚刚开始设计时，我尝试使用BIOS的功能调用来显示字符，但在实验测试运行过程中发现，字符虽然能够正确显示，但却出现了屏幕上滑，类似浏览网页鼠标滑动时的屏幕效果。我当时百思不得其解，反复核查代码也没有发现明显错误，因此十分不解。

之后，我将此问题与同学分享，同学猜测建议可能是BIOS功能调用的问题，因此我上网查阅资料，了解到原来BIOS的10h号功能调用，在显示完字符串之后，会自动将光标后移，而由于我是在屏幕24行79列也即最右下角的位置显示字符的，因此光标后移之后，就会使得屏幕出现滑动的效果，无法达到预期效果。了解到这个问题之后，我果断弃用了这个方法，选择使用显存段地址的方式显示。

因此类似于实验一的过程，将GS段地址赋予显存段地址0B800H,ax寄存器高位ah保存字符的显示属性，低位al保存需要显示的字符的ASCII值，再将ax寄存器的值赋值给对应显存地址，其中偏移的计算公式为： $(80 \times \text{行坐标} + \text{列坐标}) \times 2$ 、在这里就是： $((80 \times 24 + 79) \times 2)$ 。

除此之外，还需要特别注意的是，由于时间中断的触发比较快，肉眼难以观察到字符轮转形成的风火轮效果，因此需要多次触发响应一次，因此利用一个tcount变量用以计数，当时钟中断触发4次之时，才会进入中断响应程序的显示字符部分，否则，程序则发送EOI信号给主从8259A控制器，再次触发中断。中断的返回利用iret指令。

具体代码如下：

```
Timer:
;保护寄存器
;下列寄存器在此过程会被改变，需要保护
push ax
push bx
push cx
push dx
push bp
push es
push ds
```

```
dec byte[tcount];判断计数是否满4，即tcount值是否为0
jz tJudge
jmp INTR;若未满足，则发送EOI信号给主从8259A
```

tJudge段用于比较判断存放当前状态的state值，根据state值判断需要显示哪一个字符。

```
tJudge:
mov byte[tcount],tdelay ;达到四个时间中断输出一行，此时计数器要重置
cmp byte[state],1;依次比较state的值，类似于switch语句
jz C1
cmp byte[state],2
jz C2
cmp byte[state],3
jz C3
cmp byte[state],4
jz C4
```

C1~C4段执行将不同状态各自对应的字符赋值给al寄存器，便于之后的显示，同时更改state状态的值，由于这是一个循环过程，状态为1~4，因此当当前状态为4时，需要重置状态值为1。下面摘选C4段的代码，其他三段类似。

```
C4:
mov al,byte[ch4];将对应的字符赋值给al寄存器
mov byte[state],1 ;状态在1~4变化，当当前状态为4时置为0
jmp tshow
```

tshow段用以显示字符，利用显存段地址0B800H显示字符，ax寄存器保存字符的显示属性和字符ASCII值。具体如下：

```
tshow:
mov ah,0Fh ; 0000: 黑底、1111: 亮白字（默认值为07h）
mov word[gs:((80*24+79)*2)],ax;偏移地址公式（（80×行坐标+列坐标）×2）
mov byte[tcount],tdelay;重置tcount计数器的值
```

最后，特别注意的是，中断返回需要利用iret指令，且当程序计数值未满足时，需要给主从8259A控制器发送EOI信号，才能返回。如下所示：

```
INTR:
mov al,20h
out 20h,al ;发送EOI到主8259A
out 0A0h,al ;发送给从8259A
pop ds
pop es
pop bp
pop dx
pop cx
pop bx
pop ax
iret;中断返回指令
```

基于上述，就可以完成时钟中断响应程序的编写。

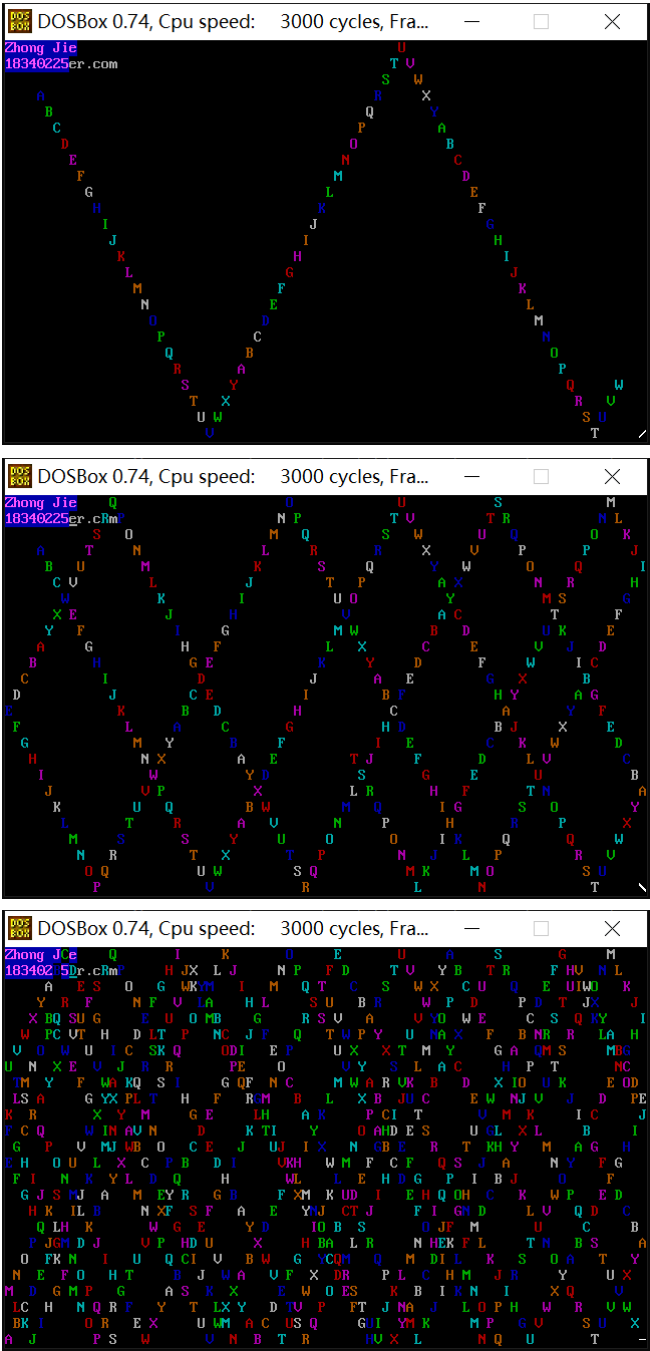
○ 画框字符反弹运动

根据实验要求，此时画框其他区域需要有字符的一些运动，因此类似于实验一的程序，完成字符45度的反弹轨迹，程序类似于实验一的程序，再此不再赘述。

当然，需要特别说明的是，在程序设计初期，我对于画框字符反弹运动程序段以及对应时钟中断程序段这两部分放在一个程序中，还有些许担忧，我担心两者会相互干扰，后来，在自己的冷静思考之后，发现程序的指令是顺序执行的，字符的运动与实验一类似是正常执行的，只不过系统会定时触发时钟中断，此时会中断主程序(也即字符反弹运动程序)的执行，转去执行自己编写的响应程序，执行完成之后再返回主程序，在非中断响应阶段，字符反弹运动的程序段是不会执行到Timer程序段的，因此我的担心是多余的。

实验结果

利用NASM将Timer.asm生成COM格式程序，并在DosBox下运行COM格式程序结果如下：



- 重写和扩展实验三的的内核程序，增加时钟中断的响应处理和键盘中断响应

基于实验三的内核程序设计，我的内核程序有C语言和汇编语言混合编程而成，由三个模块组成：入口程序模块、C语言程序结构组织模块和汇编底层功能实现组成。考虑到需要增加时钟中断和键盘中断的响应，而这两部分均涉及硬件的实现，因此选择用汇编语言编写响应程序。

在实验原理部分，我总结了修改中断的两大步骤：修改中断向量表和编写中断响应程序。由于时钟中断是需要在整个系统运行中持续响应的，因此对于时钟中断的中断向量的修改需要在系统最开始处，而对于键盘中断，由于在操作主界面还需要输入命令进行选择的操作，因此只能在执行用户程序时才能执行我自己编写的键盘中断响应程序，而在其他时候，均需要正常的键盘中断响应程序。由此，对于键盘中断的中断向量的修改需要在准备执行用户程序之时才能进行，且在执行完用户程序之后需要立即恢复正常的键盘中断，确保正常输入操作命令。

下面具体阐述有关时钟中断和键盘中断的设计：

• 时钟中断

在实验内容的第一部分——编写一个时钟中断响应的汇编程序，已经具体描述了时钟中断的设计过程。在此就主要就在TASM汇编编程下的时钟中断编程程序与上述NASM汇编的不同之处。

◦ 修改中断向量表

对于时钟中断的中断向量的修改需要在整个系统开始时，根据实验三的系统整个设计，在引导扇区程序加载内核程序至内存相应位置，并将控制权交给内核程序之时，内核程序即从入口模块开始执行，入口模块主要是指程序入口，即从C模块的cmain函数开始执行。因此为了达到风火轮效果贯穿系统运行始终，在入口模块程序调用C模块cmain函数之前就先修改了时钟中断的中断向量。

修改方法与上述类似，只不过在TASM汇编下，段地址在[]之外，且需要利用：数据类型+ptr的形式取得对应内存地址的值并指明类型。

具体如下：

```
;定义向量中断表，内存地址为[0:1023]
;低位为IP值，高字节为cs值
;每个中断向量占32位，cs、ip各占两个字节！
xor ax,ax
mov es,ax;清零段地址，中断向量在内存：0~1023处
mov word ptr es:[32],offset Timer;时钟中断响应程序的偏移地址
mov word ptr es:[34],cs
```

◦ 编写时钟中断的响应程序

整个时钟中断响应程序的设计思路与上述的时钟中断响应汇编程序一致，只不过在此是利用TASM汇编语言实现的，而且由于在实验三中设计的内核程序分成了三个模块，其中一个模块是汇编语言实现的硬件相关的底层功能程序，内含多个汇编过程供C模块调用。而本次实验时钟中断的响应程序也是硬件相关，因此同样选择用汇编实现，并且将响应程序段放在硬件相关的底层功能程序中。

对于TASM语言下的编写与NASM不同的是，需要利用offset取得变量的偏移地址，而且对于段地址：偏移地址形式下取值时，需要将段地址写在[]外，同时需要利用ptr取值、指明数据类型等。

由于在实验三的内核程序中，很少使用汇编模块定义的变量，因此对于利用汇编语言来使用汇编模块中定义的变量不够熟练，特别是对于指明段地址时使用那个寄存器感到迷惑与不确定。在不使用段前缀形式指明段地址时，访问自定义变量时，默认使用DS寄存器。因此在编程初期，我不适用段前缀显性指明段地址，即使用默认段地址，但在编译时，会出现关于段前缀的报错问题，因此我不得不采用段前缀形式指明段地址。但由于对于DS、ES等寄存器的陌生感，我抱着试试看的心态，考虑到在入口模块已经将DS、ES、SS寄存器赋值成与CS寄存

器一致，而在整个程序过程中除了修改中断向量表时借用了es寄存器，并也已经立即恢复外，并未更改这三个段寄存器的值，而且在TASM语言显示字符时，必须使用es寄存器作为段地址，因此，我在访问自定义变量时也指明段地址为es寄存器的值。

在这样编写之后，测试运行程序，发现程序运行无误，达到预期。为了搞清楚程序运行过程中DS、ES、SS、CS等段寄存器的值的变化，且为了确认访问变量时的ES寄存器的值是否与默认的DS寄存器值一致，我利用Bochs工具进行调试，设置断点进入程序段，并利用了sreg命令查看CPU寄存器的状态，具体如下：

```
(bochs:18) sreg
es:0x0800, dh=0x00009300, dl=0x8000ffff, valid=1
Data segment, base=0x00008000, limit=0x0000ffff, Read/Write, Accessed
cs:0x0800, dh=0x00009300, dl=0x8000ffff, valid=1
Data segment, base=0x00008000, limit=0x0000ffff, Read/Write, Accessed
ss:0x0800, dh=0x00009300, dl=0x8000ffff, valid=7
Data segment, base=0x00008000, limit=0x0000ffff, Read/Write, Accessed
ds:0x0800, dh=0x00009300, dl=0x8000ffff, valid=1
Data segment, base=0x00008000, limit=0x0000ffff, Read/Write, Accessed
fs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
gs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:0x0000, dh=0x00008b00, dl=0x0000ffff, valid=1
gdtr:base=0x000000000000fab7, limit=0x30
ldtr:base=0x0000000000000000, limit=0x3ff
```

根据Bochs调试结果发现，在访问自定义变量是ES寄存器的值是与DS寄存器的值一致的，因此可以访问到特定的变量值，实现设定的功能。而出现这个现象的原因也多亏于在更改ES寄存器值之后，我及时将ES寄存器恢复了原值，因此可以在程序运行过程中，保持一致。

之后，尝试了将ES寄存器改成DS寄存器也是可以成功运行的，当然在显示字符时，是需要利用es寄存器的，且须将es寄存器的值设为显存起始地址。

而对于时钟中断响应程序的其他设计，比如：计数器的设定、四个字符对应的四个状态的设定以及发送EOI信号给主从8259A控制器的设定均与“编写一个时钟中断响应程序的汇编程序”部分所述一致，再此就不再赘述了。

● 键盘中断

根据实验内容与要求，需要扩展实验三的的内核程序，但不修改原有的用户程序，实现在用户程序执行过程中，若触碰键盘，屏幕某个位置会显示“OUCH!OUCH!”。基于上述要求，其实就是要求我们重写一个键盘中断的中断响应程序，在用户程序执行过程中，触发键盘中断时需要执行我们自己编写的中断响应程序。

根据实验原理部分，本人简要概括的修改中断的过程可知，修改中断需要涉及两个方面：修改对应中断的中断向量以及编写中断响应程序。但对于键盘中断的考虑过程不似时间中断般简单，由于在实验三设计的内核程序中，系统执行的操作是由用户输入的命令决定的，因此如果像时间中断一样，在内核程序入口处就修改键盘中断的中断向量，那么会导致用户无法成功输入命令，系统无法执行相应的操作。因此，对于所修改的键盘中断响应程序必须仅在用户程序执行期间有效。

那么，对于键盘中断的修改必须在准备加载用户程序并执行之前才能进行，而且为了在结束用户程序执行之后，回到主界面仍能进行输入命令的操作，在刚刚结束用户程序执行返回内核程序之后，必须马上恢复原有的键盘中断（即将键盘中断的中断向量恢复成原来的键盘中断）。

而对于键盘中断的中断响应程序的编写，则需要利用BIOS的10h中断的显示字符串的功能调用。

下面关于修改中断向量以及编写响应程序分别具体阐述：

○ 修改键盘中断的中断向量

根据上述可知，键盘中断的修改中断向量涉及到：中断向量的修改与恢复以及如何选择修改与恢复的时间问题。由于自己编写的键盘中断仅需要在执行用户程序的过程中有效，因此很自然地想到：在准备加载用户程序并执行之前修改中断向量，而在执行完用户程序返回之时恢复原来的键盘中断的中断向量。

而由于在实验三的内核程序设计中，将内核程序分成了三个模块，其中实现读取扇区加载相应的用户程序并执行的过程是用汇编语言编写的，并保存在实现底层功能的汇编模块中，因此为了实现键盘中断的中断向量修改的尽快修改、尽快恢复的目标，决定在实验三编写的加载用户程序的过程中，增加修改与恢复键盘中断的中断向量的操作。

由于键盘中断的中断号是09h，因此键盘中断的中断向量在中断向量表中的位置是内存地址36开始后的四个字节的位置，根据“高高低低”的原则，键盘中断的中断响应程序的偏移地址存放在地址36开始的两个字节中，而段地址CS存放在地址38开始的两个字节中。当然，由于涉及到在执行完用户程序返回时，需要恢复原来的键盘中断，因此在修改中断向量之前，需要用变量保存原来的中断向量，在此定义了一个字节数组normal保存原来的中断向量，具体代码如下：

```
; 保护正常的09h键盘输入
xor ax,ax
mov es,ax; ax寄存器清零，并赋值给es寄存器
mov bp,offset normal; 保存原来的键盘中断的变量normal
; 测试ptr作用！和es:[]用途
mov ax,word ptr es:[36] ; 保存IP值
mov word ptr [bp],ax
mov ax,word ptr es:[38] ; 保存cs值
mov word ptr [bp+2],ax

; 修改键盘中断为自定义中断int09h
mov word ptr es:[36],offset int09h; 自己编写的中断响应程序int09h
mov word ptr es:[38],cs
```

normal是一个含有两个字大小的元素的数组，分别取用这两个元素保存原来键盘中断的偏移地址和段地址，先保存原来的中断向量，再修改成自己编写的中断响应程序的偏移地址和段地址。

在修改完键盘中断之后，便可以读取相应用户程序所在的扇区至内存的相应位置并跳转执行并返回了，这部分在实验三的报告已经阐述，在此不再赘述。而在返回之后，涉及到对于原来的中断向量的恢复，此时需要利用暂存原来的中断向量的变量normal进行中断向量的恢复，由于在保存之时，normal第一个元素保存的是原来中断向量的偏移地址，第二个元素保存的是原来的中断向量的段地址，因此在恢复的时候将对应元素赋值给对应内存地址即可，具体如下：

```
; 恢复正常的键盘中断
xor ax,ax
mov es,ax
mov bp,offset normal; 将暂存变量的偏移地址赋值给bp寄存器，便于访问
mov ax,word ptr [bp] ; 类型+ptr指明数据类型！ptr可以取内存地址的值也可以作为指明数据类型！
mov word ptr es:[36],ax; 将第一个元素赋值给低地址的两个字节
mov ax,word ptr [bp+2]; 第二个元素赋值给高地址的两个字节
mov word ptr es:[38],ax
```

至此，就可以完成对于键盘中断的修改。

为了测试基于上述的修改、保存、恢复操作是否成功，我利用Bochs进行调试验证：

首先设置断点，s命令进入子程序_UP:

```
Bochs for Windows - Console
0000366d: ( ): mov ax, 0x0ac9 ; b8c90a
<bochs:15> s
Next at t=156564539
(0) [0x00000000081a5] 0800:01a5 (unk. ctxt): push ax ; 50
<bochs:16> u/20
000081a5: ( ): push ax ; 50
000081a6: ( ): push bx ; 53
000081a7: ( ): push cx ; 51
000081a8: ( ): push dx ; 52
000081a9: ( ): push ds ; 1e
000081aa: ( ): push es ; 06
000081ab: ( ): push bp ; 55
000081ac: ( ): xor ax, ax ; 33c0
000081ae: ( ): mov es, ax ; 8ec0
000081b0: ( ): mov bp, 0x034e ; bd4e03
000081b3: ( ): mov ax, word ptr es:0x24 ; 26a12400
000081b7: ( ): mov word ptr ss:[bp], ax ; 894600
000081ba: ( ): mov ax, word ptr es:0x26 ; 26a12600
000081be: ( ): mov word ptr ss:[bp+2], ax ; 894602
000081c1: ( ): mov word ptr es:0x24, 0x02d0 ; 26c7062400d002
000081c8: ( ): mov word ptr es:0x26, cs ; 268c0e2600
000081cd: ( ): mov bp, sp ; 8bec
000081cf: ( ): mov ax, cs ; 8cc8
000081d1: ( ): mov es, ax ; 8ec0
000081d3: ( ): mov bx, 0xa100 ; bb00a1
<bochs:17> r
rax: 0x00000000_0000003d rcx: 0x00000000_00090002
rdx: 0x00000000_00000000 rbx: 0x00000000_00000000
rsp: 0x00000000_0000ffea rbp: 0x00000000_00000000
rsi: 0x00000000_000e0000 rdi: 0x00000000_0000ffac
```

对比源代码可知，normal变量的偏移地址、自己编写的int09h中断响应程序的偏移地址。

先通过info ivt命令查看所有中断的中断向量：

```
选择Bochs for Windows - Console
<bochs:62> info ivt
INT# 00 > F000:FF53 (0x000fff53) DIVIDE ERROR ; dummy iret
INT# 01 > F000:FF53 (0x000fff53) SINGLE STEP ; dummy iret
INT# 02 > F000:FF53 (0x000fff53) NON-MASKABLE INTERRUPT ; dummy iret
INT# 03 > F000:FF53 (0x000fff53) BREAKPOINT ; dummy iret
INT# 04 > F000:FF53 (0x000fff53) INTO DETECTED OVERFLOW ; dummy iret
INT# 05 > F000:FF53 (0x000fff53) BOUND RANGE EXCEED ; dummy iret
INT# 06 > F000:FF53 (0x000fff53) INVALID OPCODE ; dummy iret
INT# 07 > F000:FE53 (0x000ffe53) PROCESSOR EXTENSION NOT AVAILABLE ; dummy iret
INT# 08 > 0800:0241 (0x00008241) IRQ0 - SYSTEM TIMER
INT# 09 > F000:E987 (0x000fe987) IRQ1 - KEYBOARD DATA READY
```

可知原来键盘中断的中断向量为：F000:E987

在执行完保存原中断向量和修改新中断向量之后，通过x命令查看normal向量内存地址的内容：

```
bochs:25> x /4bx 0x0800:0x034e
bochs:
0x0000000000000834e <bogus+ 0>: 0x87 0xe9 0x00 0xf0
bochs:26>
```

与原来的键盘中断一致，证明保存成功，再利用x命令查看内存此时键盘的中断向量：

```
bochs:28> s
Next at t=156564554
(0) [0x00000000081c8] 0800:01c8 (unk. ctxt): mov word ptr es:0x26, cs ; 268c0e2600
bochs:29> s
Next at t=156564555
(0) [0x00000000081cd] 0800:01cd (unk. ctxt): mov bp, sp ; 8bec
bochs:30> x /4bx 0x0000:0x024
bochs:
0x00000000000000024 <bogus+ 0>: 0xd0 0x02 0x00 0x08
bochs:31>
```

此时键盘中断的中断向量已经修改成功。

接着继续执行读取扇区以及加载用户程序的操作，执行完成用户程序返回后，进行中断向量的恢复，查看恢复以后的键盘中断的中断向量：

```
bochs:49> xp /4bx 0x0000:0x0024
bochs:
0x00000000000000024 <bogus+ 0>: 0x87 0xe9 0x00 0xf0
bochs:50>
```

已经成功恢复成原来的中断向量。

说明对于键盘中断的中断向量的修改是成功的。

○ 编写键盘中断的中断响应程序

根据实验要求，需要在用户程序执行期间，每次按下一个键盘按键，屏幕某个位置就会显示“OUCH! OUCH!”信息。基于此要求很自然地联想到利用BIOS的10h号中断显示字符串的功能调用进行显示，此中断调用在前几次实验中已经熟练地多次使用了。由于要求规定，每按下一个按键就需要显示字符串，但由于键盘中断的响应程序执行时间很短，因此为了方便观察字符串的显示效果，通过查阅BIOS的中断大全，了解到BIOS的15h号中断号的86h功能号的中断调用可以进行延时操作，在显示字符串后，调用此中断进行延时2s的操作，便于显示观察。

且为了区别每一次按下按键的响应过程，在显示字符串时，先进行了延时2s的操作，2s之后调用清除屏幕内容的功能调用进行该位置字符串的清除操作，同时对于每一次的中断响应，在显示字符串时进行了变色操作，设置一个color变量存储字符的显示属性，初值为1，每次响应之后color自增1，当color值为8时，重新置1，实现字符串显示的颜色变化循环。

首先，对于显示字符串，调用BIOS的10h显示字符串的功能调用：

```
mov ax,cs
mov es,ax;段地址
mov ah,13h ;功能号
mov al,0
mov bl,byte ptr es:[color];显示属性
mov bh,0
mov dh,12;显示字符串的行坐标
mov dl,35;显示字符串的列坐标
mov bp,offset Info ;"OUCH!OUCH!"字符串的偏移地址
mov cx,11;要显示的字符串长度
int 10h
inc byte ptr es:[color];颜色变量自增
mov al,8
cmp al,byte ptr es:[color];判断变量是否为8
jz Reset;若为8重置为1
jmp Dump;不为，延时
```

显示完字符串之后，进行延时操作，延时2s：

```
mov ah,86h ;BIOS的15h中断号86h功能号的延时功能
mov cx,1Eh ;CX: DX= 延时时间（单位是微秒），CX是高字，DX是低字
mov dx,8480h ;2s=2000000us=0x1E8480
int 15h
```

延时显示之后，需要调用BIOS的10h号中断清除该位置的字符串，便于下一次的显示：

```
mov ah,6
mov al,0
mov ch,12 ;清屏的左上角坐标
mov cl,35
mov dh,12 ;清屏的右下角坐标
mov dl,45
mov bh,7 ;默认属性，黑底
int 10H
```

最后，特别注意的是，在结束中断响应程序的操作，准备返回主程序时，需要利用in指令读写指定8259A的端口，并发送EOI信号给主从8259A控制器，再利用iret指令返回。

```

in al,60h
mov al,20h                ; AL = EOI
out 20h,al                ; 发送EOI到主8529A
out 0A0h,al               ; 发送EOI到从8529A
pop bp
pop es
pop ds
pop dx
pop cx
pop bx
pop ax
iret;

```

至此，键盘中断的中断响应程序就设计完成了，修改中断向量以及中断响应程序的返回是关键！

- **修改21h、22h、23h、24h这四个中断与编写一个用户程序调用四个中断**

编写实现用户自定义中断即是：在内核中，对33号、34号、35号和36号中断编写各自的中断服务程序，分别在屏幕1/4区域内显示一些个性化信息。再编写一个汇编语言的程序，作为用户程序，利用int 33、int 34、int 35和int 36产生中断调用你这4个服务程序。简单说来，就是将33号~36号的四个中断的响应程序分别对应于实验二中编写的四个用户程序，当发生中断调用时，需要执行对应的用户程序。而由于33号~36号不像时钟中断和键盘中断可以由系统或者用户按键触发，因此还需要一个用户程序通过int指令显性地调用这四个中断，验证这四个中断的服务程序。

- **实现四个用户自定义中断**

根据上述描述以及实验原理部分的总结，其实实现这四个中断主要就是需要：修改中断向量表以及编写中断服务程序。由于这四个中断不与我的程序运行过程的操作相冲突，因此为了方便，与时间中断类似，在内核程序的入口模块调用cmain函数进入内核主程序之前就先修改这四个中断的中断向量，分别将描述这四个中断的子程序段的偏移地址赋值给：中断号×4开始的低地址的两个字节，将CS段地址赋值给：中断号×4+2开始的高地址的两个字节即可。具体操作如下：

```

;定义向量中断表，内存地址为[0:1023]
;低位为IP值，高字节为CS值
;每个中断向量占32位，cs、ip各占两个字节！
xor ax,ax
mov es,ax;清零段地址，中断向量在内存：0~1023处
mov word ptr es:[33*4],offset int21h
mov word ptr es:[33*4+2],cs

mov word ptr es:[34*4],offset int22h
mov word ptr es:[34*4+2],cs

mov word ptr es:[35*4],offset int23h
mov word ptr es:[35*4+2],cs

mov word ptr es:[36*4],offset int24h
mov word ptr es:[36*4+2],cs

```


int21h~int24h为我编写的这四个中断的对应的中断服务程序，由于入口模块只负责确定内核程序的入口点，而还有一个模块负责底层功能的实现，这个模块包含若干子过程、子程序，因此将这四个中断对应的中断服务程序也编写在底层功能的实现模块，而只需要在入口模块include底层功能实现模块即可。

下面阐述这四个中断对应的中断服务程序的编写：

由于四个中断正好对应四个用户程序，因此按照顺序对应的方式：21h中断对应第一个用户程序~24h中断对应第四个用户程序，根据要求所示，这四个中断的中断服务程序其实就是执行对应的用户程序，而在实验三的设计中，我设计了一个过程UP负责读取对应用户程序所在的扇区并加载执行用户程序的功能，UP过程需要一个参数，用以指明读取的扇区。

因此，结合上次实验实现的过程_UP,对于这四个中断服务程序，我们只需要传入一个这四个中断程序对应的用户程序所在的扇区号作为参数，调用UP过程执行用户程序即可，再利用iret指令结束中断服务程序的执行，返回即可。如下选取int21h的中断服务程序示例，其他三个类似：

```
;编写21h~24h中断服务程序，分别执行四个用户程序
int21h:
    mov ax,56;对应的第一个用户程序在第8个扇区，‘8’=56
    push ax;传入所在扇区号的参数
    call _UP;调用加载用户程序过程
    pop ax;汇编模块调用传入参数时，需要在调用返回时出栈参数
    iret;中断服务程序返回
```

由于UP过程的实现已经在实验三详细叙述了，在此就简要说明，不再详述。加载执行用户程序，只需将用户程序存放在软盘的特定扇区中，内核程序加载某个用户程序时，只需要读取对应用户程序所在的扇区，将其加载在内存自己设定的位置并跳转执行即可。我将扇区号作为此“加载用户程序”的参数，由C模块根据用户的选择将对应用户程序的扇区号以字符形式作为参数传给此_UP过程。注意，在此过程调用时同样涉及从栈中取参数的过程，需要根据入栈的内容判断从栈中哪个位置取得参数！

基于上述，可以实现这四个用户自定义中断。

○ 编写一个用户程序调用上述中断

由于21h~24h这四个中断不会由系统自动触发，也不能类似于键盘中断由用户输入触发，因此需要编写一个用户程序，显性调用这四个中断。显性调用这四个中断的指令很简单，只需要利用int指令+中断号的形式，就会触发中断，CPU会执行中断号对应的中断服务程序并返回。

■ 设计用户程序

为了避免此用户程序与调用之前设计的四个用户程序执行时发生冲突，将本用户程序加载在内存偏移地址为0B100H处，因此程序开头用org指令声明即可。接着只需要使用int指令显性调用这四个中断即可，最后加上ret指令，方便调用本用户程序时的返回操作。

下面展示代码：

```
org 0B100H;程序加载到内存初始偏移量为B100h处,不能与用户程序加载到相同地址;因为中断调用需要加载到内存相应地址,若为同一地址会覆盖
```

```
start:
    int 21h;中断调用21h号,执行第一个用户程序
    int 22h;中断调用22h号,执行第二个用户程序
    int 23h;中断调用23h号,执行第三个用户程序
    int 24h;中断调用24h号,执行第四个用户程序
    jmp Quit
Quit:
    ret;ret指令返回程序
times 512 - ($ - $$) db 0 ;将前512字节不是0就填0
```

■ 加载本用户程序

但值得特别注意的是,在前几次实验中,我们均将用户程序加载到内存同一个地址中执行,但由于此用户程序是显性调用中断,且这四个中断的中断服务程序就是需要加载执行之前设计的四个用户程序的。因此如果,我们将此用户程序也加载到与之前设计的用户程序在同一内存地址中,那么当系统执行此用户程序时触发中断,进入中断服务程序,在中断服务程序中会调用_UP过程,加载之前的用户程序至同一内存地址执行,那么本用户程序就会被新加载的用户程序所覆盖,出现中断无法返回的现象。

因此本用户程序的加载方法与之前加载用户程序的方法类似使用BIOS的13h读扇区功能,但加载的内存地址却不能与之前的用户程序相同!因此本次实验,选择将本用户程序加载到内存偏移地址为0B100H处,避免与之前的四个用户程序冲突。

而且,本用户程序在执行时,也需要像之前的四个用户程序一样,由用户输入命令,内核程序执行,因此也涉及到C模块调用汇编模块的操作,因此需要在底层功能实现模块将此过程命名加上下划线,且用public声明便于C模块的调用。在软盘组织时,将本用户程序放在第14个扇区,因此读取扇区的扇区号为14,不用传入参数指定扇区号。

下面展示底层功能实现模块中,加载此用户程序的代码:

```
OffsetofINT equ 0B100H
;实验要求一个用户程序调用21h、22h、23h、24h中断
;中断即调用程序
public _RunInt
_RunInt proc
    push ds
    push es;保护现场,保护ds、es寄存器的值
    push ax
    push bx
    push cx
    push dx
    mov ax,cs
    mov es,ax ;置es与cs相等,为段地址,入口时CS为0A00H
    mov bx,OffsetofINT ;数据常量,代表偏移地址,段地址:偏移地址为内存访问地址
    mov ah,02h ;功能号02h读扇区
    mov al,1 ;读入扇区数
    mov dl,0 ;驱动器号,软盘为0, U盘和硬盘为80h
    mov dh,0 ;磁头号为0
    mov ch,0 ;柱面号为0
    mov cl,14 ;起始扇区号
    int 13h ;中断号
;将对应用户程序加载到内存为0800H:0B100H处
    mov bx,OffsetofINT ;将偏移地址赋值给bx
    call bx ;程序跳转到对应内存位置执行用户程序
```

```

    pop dx
    pop cx
    pop bx
    pop ax
    pop es
    pop ds ;恢复现场，逆序出栈
    ret ;函数模块返回
_RunInt endp

```

整体的读取扇区以及call调用执行思路与实验三设计相同，唯有加载到内存的偏移地址不同！

• 用户程序的修改

由于在本次实验中要求：在用户程序执行过程中，触发键盘中断时，要响应自己编写的键盘中断服务程序，因此在用户程序过程中原有的键盘中断服务程序无效，而原有的键盘中断服务程序是将用户按下的按键对应的字符的ASCII码值，存入键盘缓冲区。而由于原来的键盘那中断服务程序失效了，因此键盘缓冲区中不再是存放按键字符了，所以我们也不能通过int 16h BIOS功能调用从键盘缓冲区中读取字符，来判断结束用户程序的运行了。

因此，在本次实验中，我将用户程序的退出方式更改为延时计数退出，即定义一个计数变量，用户程序的字符运动每运动一次，计数变量加一，在每次运动结束都判断此计数变量的值是否达到上限，若达到上限，则用户程序以ret指令返回主程序，结束执行；反之，继续执行。

下面展示相关部分代码：

```

mov bx,150;设定上限150次
    mov ax,word[count];计数变量count
cmp ax,bx
    jz Quit
    jmp Loop
Quit:
    ret;ret指令用栈中的数据，修改IP的内容，从而实现近转移

```

• 内核程序的新增功能

本次实验要求确保：本次实验的内核功能不比实验三弱。因此在实验三的基础上，我扩展了一些命令操作并且增加关机界面。

◦ 内核程序所包含的命令操作如下：

下述命令的具体展示会在实验结果部分详述。

1. help-display the command list; //显示菜单
2. cls-clear the window; //清屏
3. UP1-run the first program of user; //运行第一个用户程序
4. UP2-run the second program of user;
5. UP3-run the third program of user;
6. UP4-run the fourth program of user;
7. INT-run the interrupt program; //执行调用21h~24h中断的用户程序
8. dir-display the table of UPInfo; //显示用户程序文件信息
9. ls-dispaly the tree of files; //列出系统所包含的所有文件（树型）
10. batch-run a series of commands; //批处理，指定多个命令
11. qb-exit the batch running; //退出批处理执行
12. quit-exit the system; //退出系统

INT命令的实现即前面所述的编写一个用户程序实现四个用户自定义中断的过程，在此不赘述。

ls命令的实现也主要是调用C模块自己一用汇编底层模块的输出一个字符的功能封装的printf函数实现的，关于如何封装printf函数，也已经在实验三的操作系统内核程序设计——输出字符串函数中详细叙述了。

dir命令即显示用户程序相关的信息，以类表格形式显示，我单独创建了一个用户程序用以显示相关信息，这点在之前已经的实验中已经详述，在此不赘述。

而关于**batch——批处理命令**，这点我是受之前实验要求的设计一种命令可以按指定顺序执行多个用户程序的启发，我将这种命令不再局限于执行多个用户程序，只要是菜单中显示的命令均可以进行批处理操作，均可以按照用户输入的次序依次操作，完成批处理功能！

实现过程主要是，依赖于实验三中由C模块调用汇编模块的输入一个字符、输出一个字符过程而分别封装成的InStr——**输入字符串函数**、printf——**输出字符串函数**。在内核程序的C模块中修改实验三中的多命令函数，用户通过InStr输入字符串，多命令分析函数依次以空格为分隔符，提取出每个命令执行，执行后返回提取下一个命令直到字符串提取完成，结束批处理。

下面展示相关代码：

```
void ANLcmd(char * cmd)
{
    int index = 0; //多命令字符串下标
    char subcmd[10]; //用以存储子命令的数组
    int j = 0;
    while (cmd[index] != '\0' && end && flag) //当多名字符串到达结尾或执行了终止系统命令或执行了终止多命令处理命令时退出执行
    {
        if (cmd[index] == 32) //空格为分隔符
        {
            subcmd[j] = '\0'; //得到一个子命令
            Run(subcmd); //执行子命令
            j = 0; //子命令数组下标重新归零
        }
        else
        {
            subcmd[j] = cmd[index]; //未遇到空格，则读取子命令字符
            j++;
        }
        index++;
    }
    if (cmd[index] == '\0') //最后一个命令
```

```

{
    subcmd[j] = '\0';
    Run(subcmd); //执行最后一个命令
}
}

```

◦ 关机界面

在实验三我实现了开机界面的显示，在本次实验我增加了在用户输入“quit”命令退出系统时的关机界面，界面不涉及字符的运动，主要以图形显示的方式展现，界面显示了一个笑脸图案以及显示了本人的相关信息，作为结束。

首先利用int10h BIOS的功能调用将屏幕清屏成蓝色，再利用int 10h的显示字符串的功能调用显示相关信息。下面展示部分代码：

```

clear:
    mov ah,06h; 向上滚屏
    mov al,0h ;滚动行数（0~清窗口）
    mov ch,0 ;左上角X坐标
    mov cl,0 ;左上角Y坐标
    mov dh,24 ;右下角X坐标
    mov dl,79 ;右下角Y坐标
    mov bh,0x10;空白区域的显示属性：蓝底黑字，无闪烁无加亮
    int 10h ;BIOS功能调用

showInfo:
    ;调用int10h显示操作系统关机的相关提示信息
    inc word[count]
    mov ax,cs
    mov es,ax
    mov bp,Message;偏移地址，采用段地址：偏移地址形式表示串地址
    mov ax,1301h ;ah=13h为功能号，al=01h指光标位于串尾
    mov bx,0014h ;bh=00h为页码，bl=14h为显示属性：蓝底红字
    mov dx,0407h ;dh为行号，dl为列号,第4行，7列
    mov cx,MessageLength
    int 10h
    mov ax,cs
    mov es,ax
    mov bp,str0
    mov ax,1301h
    mov bx,001eh
    mov dx,0c13h
    mov cx,42
    int 10h

```

关于上述的新增的内核功能会在实验结果部分具体展示。

• MyOS的模块结构

本次实验，本人的整个系统所含模块的组织如下：

1. boot模块：含boot.asm。

作用：引导扇区程序，用以加载操作系统内核，并将系统控制器交给内核

2. kernal模块：含rukou.asm kernal.c basic.asm

作用如下：

rukou.asm: 用以修改时钟中断、21h、22h、23h、24h中断的中断向量，并调用内核程序的入口——cmain函数。

kernal.c: 用以组织整个内核程序，主导系统的命令输入、程序执行操作，根据底层汇编模块封装相关函数，如：输入输出字符串函数、字符串比较函数等等。

basic.asm: 汇编语言编写的底层功能实现模块，包含清屏、输入一个字符、输出一个字符、读取扇区加载程序、时钟中断服务程序、键盘中断服务程序、21h~24h中断的服务程序。

3. UP模块：含UP1~UP4、Interrupt.asm

作用：含四个用户程序，用以执行字符运动，新增一个显性调用21H~24H中断的用户程序。

4. 表格模块：含table.asm

作用：显示四个用户程序相关的信息，包括所在扇区、地址以及文件大小

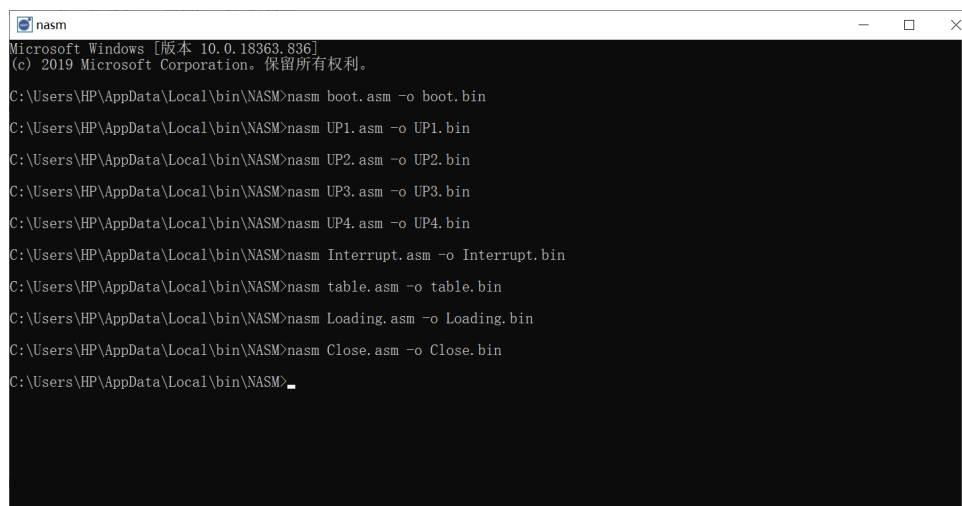
5. 开关机界面模块：含Loading.asm Close.asm

作用：用以系统启动和退出时的界面显示。

• 实验操作过程

1. NASM编译

打开NASM，在框内输入指令，对于引导扇区程序boot.asm、四个用户程序UP系列、新增Interrupt中断调用用户程序，表格程序table.asm以及开关机界面显示Loading、Close程序，我将他们都编译成BIN格式。因此输入nasm+输入文件名+-o+输出文件名即可。生成同名的二进制文件。

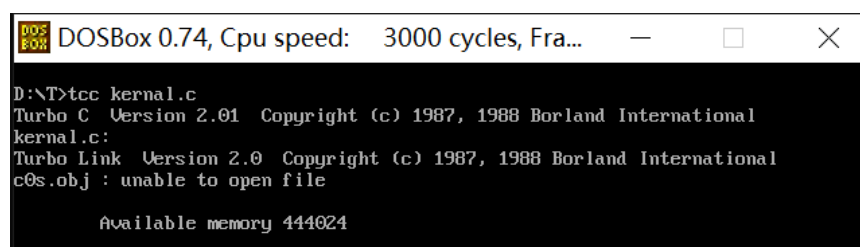


2. TCC+TASM+TLINK链接编译

在DosBox中链接编译内核程序：rukou.asm、kernal.c、baxis.asm

■ TCC编译

用tcc+.c 形式生成同名.obj文件：



■ TASM编译

用tasm+.asm形式先生成同名.obj文件，由于rukou.asm包含了basic.asm文件，因此还需要用tasm+rukou,obj形式再编译一次：

```
D:\T>tasm rukou.asm
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland International

Assembling file:   rukou.asm
Error messages:   None
Warning messages: None
Passes:           1
Remaining memory: 462k
```

```
D:\T>tasm rukou.obj
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland International

Assembling file:   rukou.ASM to obj.OBJ
Error messages:   None
Warning messages: None
Passes:           1
Remaining memory: 462k
```

■ TLINK链接

用tlink /t /3 .obj .obj, .com形式生成内核COM格式文件:

```
D:\T>tlink /t /3 rukou.obj kernal.obj, test.com
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International

D:\T>a_
```

3. winHex工具编辑软盘

不同于实验一，在Linux环境使用dd命令进行1.44MB虚拟软盘映像的生成，在实验二、三中，我利用WinHex编辑软盘，两者相比较，我认为WinHex工具更为简便操作。因此，本次实验依然使用WinHex编辑软盘。

首先生成一个1.44MB的空白软盘myos4.img.利用winHex工具打开上述生成的各个二进制文件，先将“boot.bin”二进制文件内容拷贝到空白软盘首扇区，然后将内核程序test.com存放在扇区号为2开始的六个扇区空间，起始地址为200h；接着再从第8个扇区开始，依次存放四个用户程序(UP1~4)、表格程序、Loading程序、新增Interrupt调用21h~24h中断程序以及新增的关节界面程序Close。第一个用户程序UP1存放在第二个扇区（E00H~1000H），第二个用户程序UP2存放在第三个扇区（1000H~1200H）的规则，以此类推将UP1~UP4的二进制文件以及表格程序、Loading程序、Interrupt程序和Close程序拷贝到对应扇区中。

拷贝操作的过程如下所示：

boot.bin	loading.bin	UP1.bin	UP2.bin	UP3.bin	UP4.bin	table.bin	TEST.COM	Interrupt.bin	Close.bin								
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	
00000010	00	B1	02	CD	13	EA	00	01	00	08	00	00	00	00	00	00	± í ē
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	
00000170	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	
00000190	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	
000001B0	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	
000001D0	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	
000001F0	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA		

U*

上图为“boot”引导扇区程序二进制文件内容，利用操作：选中需要复制内容，右键——编辑——复制选块——正常，完成复制操作。

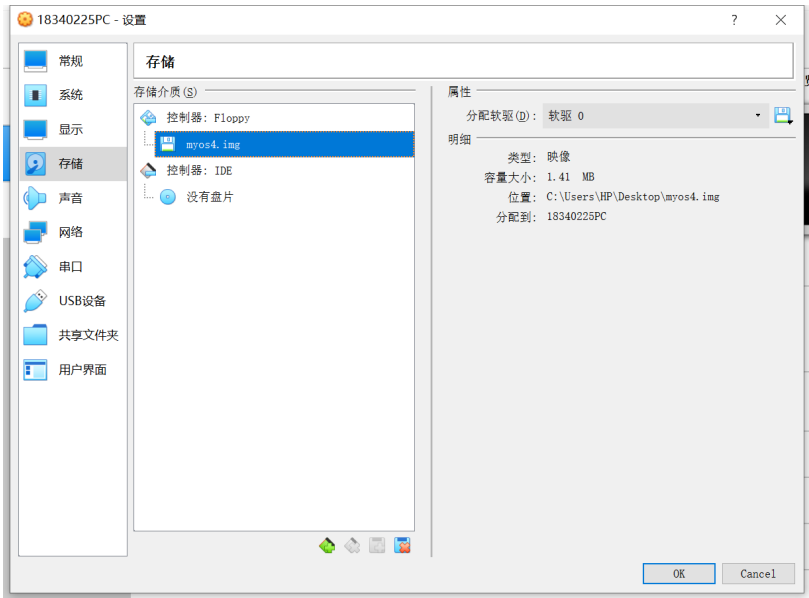
boot.bin	loading.bin	UP1.bin	UP2.bin	UP3.bin	UP4.bin	table.bin	TEST.COM	Interrupt.bin	Close.bin	myos4.img						
Offset	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
00000010	00	B1	02	CD	13	EA	00	01	00	08	00	00	00	00	00	00
00000020	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000030	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000040	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000050	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000060	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000070	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000080	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000090	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000A0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000B0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000C0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000D0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000E0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
000000F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000100	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000110	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000120	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000130	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000140	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000150	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000160	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00
00000170	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
00000190	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
000001B0	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
000001D0	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
000001F0	00	00	00	00	00	00	00	00	00	00	00	00	00	00	55	AA

上图为1.44MB软盘，将上述”boot“的内容复制到此软盘的首扇区，操作为：右键——剪贴板数据——粘贴。

其余程序的拷贝操作类似。

4. 软盘启动裸机

将上述生成的引导扇区程序二进制文件、内核COM格式文件、用户程序以及表格程序、Loading程序、新建的显性调用用户自定义中断Interrupt程序以及关节界面程序Close,依次按照扇区分配复制到1.44MB软盘的对应位置并保存后，将1.44MB软盘作为引导盘启动虚拟机。

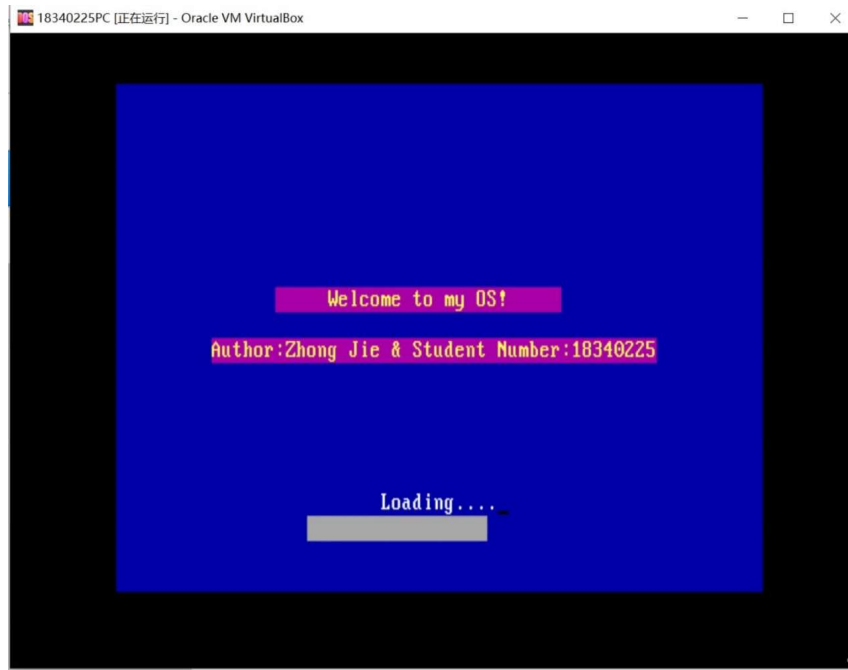


• 实验运行结果

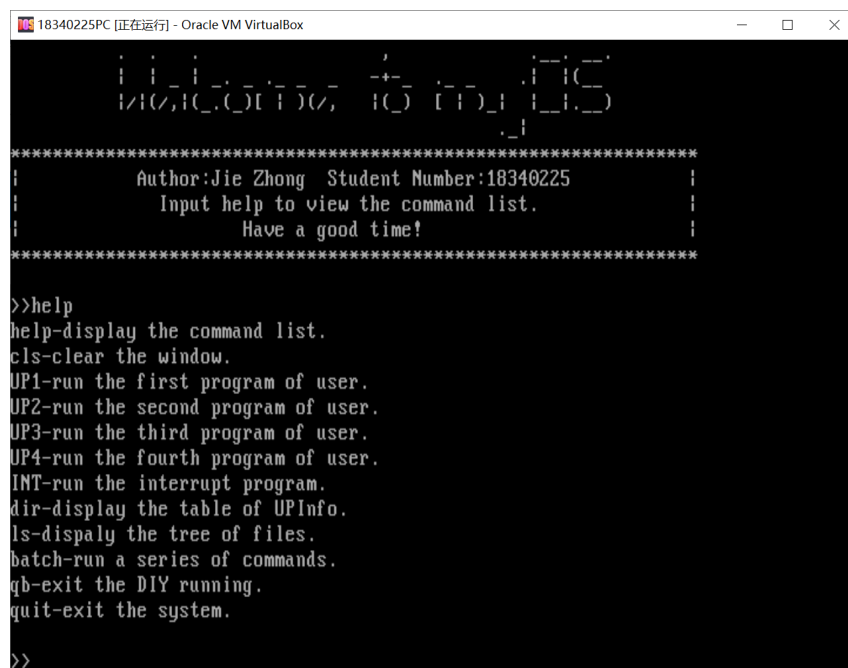
成功启动虚拟机之后，虚拟机就会将程序加载到虚拟机的内存空间中运行，可以看到Loading程序的启动界面，也可以观察到内核程序的界面，接着输入help命令，依次按照菜单，输入选择运行不同的命令如：UP1~UP4、dir、ls、batch、INT、quit，执行不同的用户程序，显性调用用户自定义中断21h~24h、查看用户程序信息、列出系统文件组织等功能，用户子程序的运行均以延时退出的方式返回到内核程序。可以看到时钟中断响应程序的风火轮效果，而且可以测试在用户程序运行期间触发键盘中断的显示效果。

程序运行结果如下：

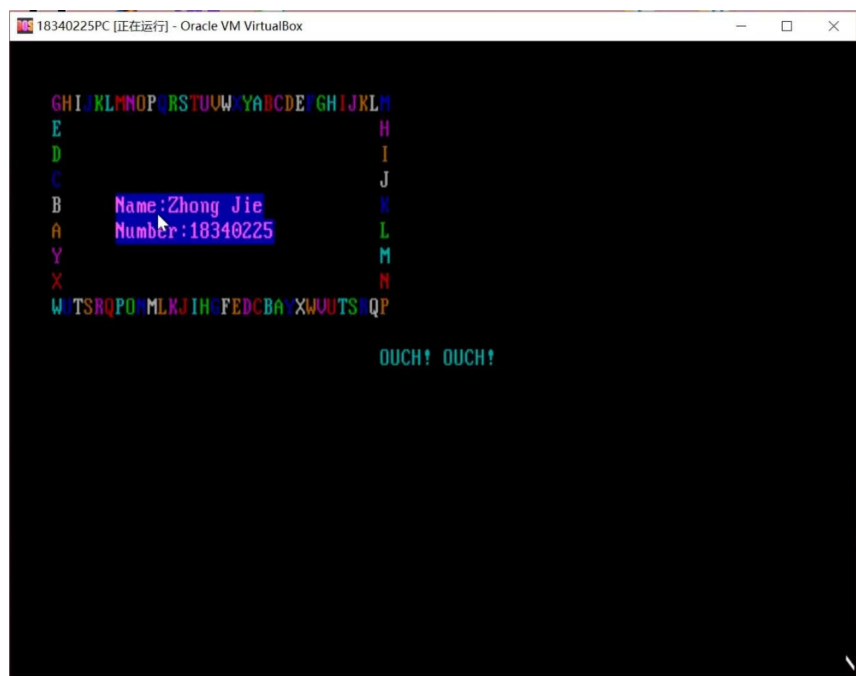
Loading程序加载界面：



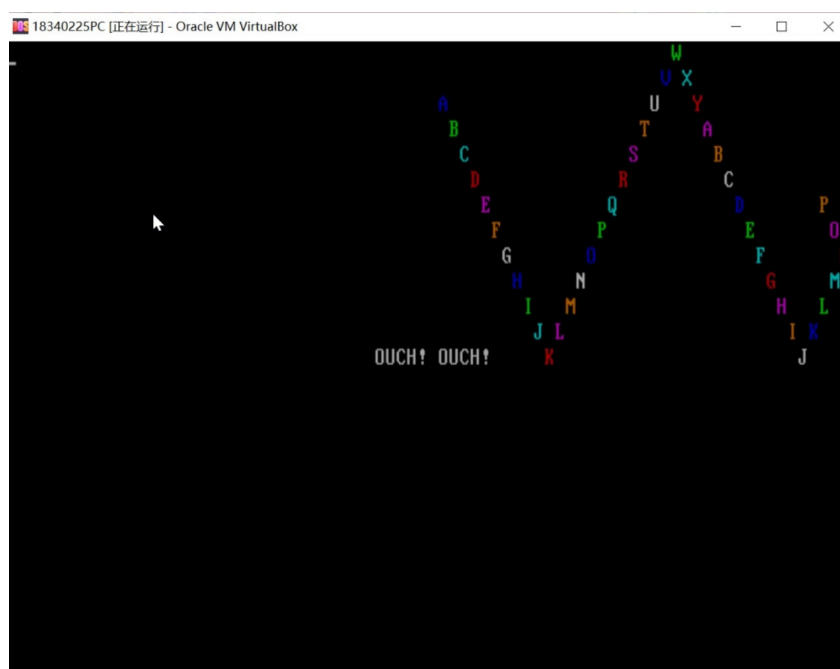
内核界面，输入“help”，显示命令清单：



输入“cls”,系统执行清屏操作，接着输入“UP1”，显示第一个用户程序，并触发键盘中断：



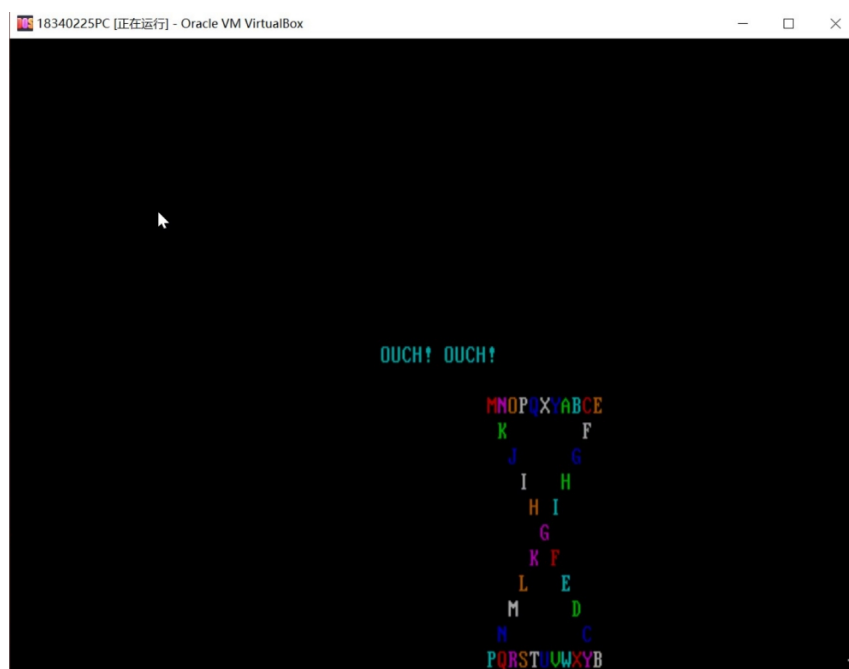
延时回到内核程序，接着输入“UP2”，显示第二个用户程序，并触发键盘中断：



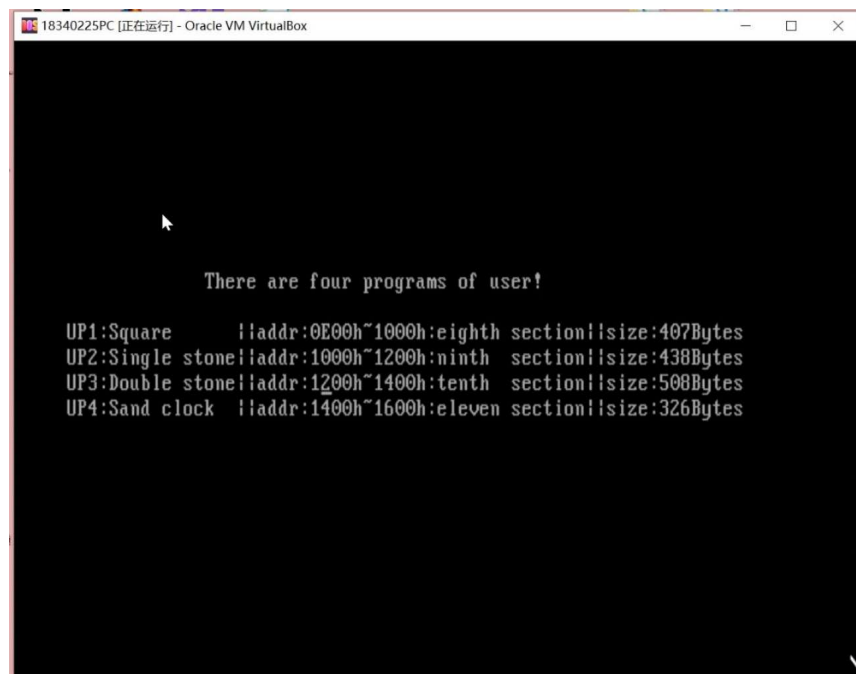
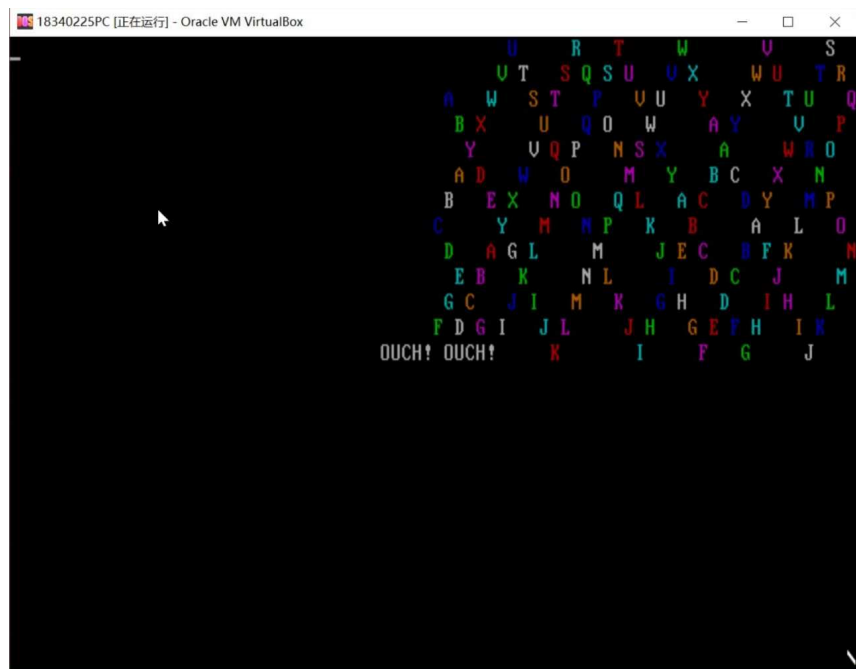
延时回到内核程序，接着输入“UP3”，显示第三个用户程序，并触发键盘中断：



延时回到内核程序，接着输入“UP4”，显示第第个用户程序，并触发键盘中断：



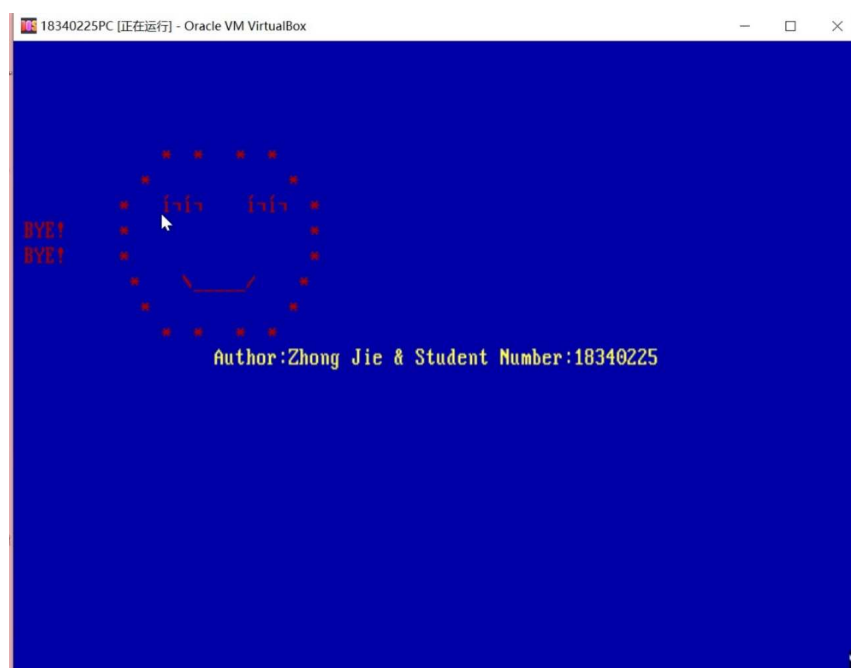
延时回到内核程序，接着输入“ls”，查看此简易系统的文件组织：



接着输入“INT”命令，测试执行，新增的显性调用21h~24h四个用户自定义中断的程序：会依次执行UP1~UP4四个用户程序：



输入“quit”退出系统，显示关机界面与提示语句。



至此，完成所有功能测试，实验运行结果与预期相符合。

实验心得

相比于前三次实验，本次实验的工作量相对较小，是在实验三完成的内核程序基础上，加入对于时钟中断以及键盘中断的响应，当然我也自己编写21h~24h的中断响应程序，并编写了一个用户程序显性调用这些中断。本次实验的核心是中断，因此对于中断的深刻理解是完成本次实验的关键。在之前的课程中，我接触过中断的概念，对于计算机如何响应中断，如何找到中断服务程序的入口地址也有些许理解，但理论与实践过程仍然是有差别的，在实验过程中，我虽然把握了整个中断设计的大方向即：修改中断向量以及编写中断服务程序，但在一些细节问题上仍遇到了不少麻烦。

比如：关于中断服务程序的返回问题，在实验初期，我很单纯地认为只要在中断服务程序的结束位置加上iret指令就行了，后来发现在键盘中断测试时，用户程序返回以后，原来的键盘中断无法恢复了。我将自己的代码与老师给的样例代码比较，发现我在程序末尾缺少了发送EOI信号，给主从8259A中断控制器的指令，加上以后程序就正常运行了，能够达到预期效果。但为什么一定要加上这两条指令呢，我满是疑惑，于是我查阅资料，发现：EOI是外部中断的中断结束命令，CPU执行：`mov al,20h, out 20h,al`这两条指令就是给8259A芯片发送EOI命令，通知8259A芯片一个中断完成，8259A将负责把ISR中的位清除(即开中断)，以便以后可以继续接受中断。如果不加的话，8259A永远收不到中断结束命令，那么就认为某一个中断一直在执行，所以如果遇到比这个阻塞的中断级别低或者相等的中断发生时就不会再响应了。因此就会出现我之前遇到的键盘中断无法恢复的问题，这其实是我的键盘中断程序没有发送EOI信号告知8259A控制器中断结束，因此就不会成功执行接下来的指令了。理解清楚了这个问题，让我对于中断服务程序返回时的操作更加清楚了。

还有就是关于TASM汇编时的段地址问题，由于本次实验的第一个内容是编写一个时钟中断的响应程序，程序为COM格式，因此我选择用最熟悉的NASM语言编译并成功之后，根据第二项内容，我需要在实验三的内核程序中加上时钟中断的服务程序，因此我根据TASM的语法规则对于第一项内容的时钟中断的响应程序进行了修改，但在访问自定义变量时，我遇到了问题，通过查阅资料了解到在访问自定义变量（标志）时，标志本身代表着偏移地址，而段地址则默认使用DS寄存器，在编程初期我就运用了这个规则，结果在TASM编译时，编译器返回需要段前缀声明的错误，因此我不得不显性指明段前缀，而在实验三中了解到TASM语言显示字符是需要利用ES寄存器进行访问的，因此我开始纠结到底是使用DS寄存器还是ES寄存器作为我访问自定义变量时的段地址呢？与其花时间纠结，不如尝试一下，于是我先使用ES寄存器作为段地址访问我的自定义变量，并用Bochs工具进行调试，利用x、xp命令去查看以ES寄存器内容作为段地址时所访问的内容是否为我所需要的自定义变量的内容，同时还利用sreg命令在程序运行期间多次查看ES、DS寄存器的值。发现其实用ES寄存器访问是没有问题的，在程序运行过程中，虽然在更改中断向量时对ES寄存器的值进行了修改，但是在更改完之后及时进行了恢复，因此在程序运行的绝大部分时候，由于：`mov ax,cs mov es,ax mov ds,ax`指令的执行，ES寄存器与DS寄存器的值都是相等的且会等于CS寄存器的值，因此使用ES寄存器、DS寄存器作为段地址均是可以成功访问自定义变量的。

最后就是关于对于寄存器的保护问题，在键盘中断响应程序测试初期，显示字符串老是不能显示出来，后来用Bochs调试，发现对于键盘中断向量的修改并没有问题，而是在键盘中断响应程序执行过程中，ax~dx寄存器的值会发生莫名其妙的变化，这点折磨了我很长时间，后来我怀疑是我没有对这几个寄存器进行保护的原因，于是，我在进入子程序前加上push ax~dx以及一些在子程序段会被改变的寄存器的指令，用以保护寄存器，并在子程序段结束之前用pop指令逆序恢复寄存器的值。果然成功了，整个程序正常显示运行了。因此，在之后我都特别注意对于寄存器的保护问题，以防再出现这种情况。我认为，注意保护现场与恢复现场是一个很好的习惯，特别对于程序的嵌套调用时，更显得至关重要。

实验过程就是在不断解决一个又一个问题中成长起来的，这次实现虽然完成了实验要求与内容，但还有很多的问题需要改善，比如对于内核程序的功能的进一步扩展、键盘中断的防抖问题以及将FAT12文件系统的结合问题。希望在之后的实验中可以进一步完善我的简易操作系统，进一步提高自己的汇编编程水平，再接再厉！

参考文献

1. 《BIOS中断大全》 :https://blog.csdn.net/weixin_37656939/article/details/79684611
2. 《突破512字节的限制》 : <https://blog.csdn.net/cielozhang/article/details/6171783/>
3. 《X86汇编如何设置延时》 : <https://blog.csdn.net/longintchar/article/details/70149027>
4. 《Bochs调试指令》 : https://blog.csdn.net/ddna/article/details/4997695?utm_medium=distri_bute_pc_relevant.none-task-blog-baidujs-1