

# 实验四：具有中断处理的内核

---

计算机科学与技术 18340146 宋渝杰

## 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。
3. 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!"。
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

## 实验过程：

### 操作环境：

- 操作系统：win10 + Linux
- 虚拟机：VirtualBox
- C 编译器：gcc
- x86 编译器：nasm
- 链接器：ld

1. **步骤一：**编写的对时钟中断的相应处理程序关键代码如下：

定义 8 号中断，放入中断向量表：

```

mov ax, 0000h          ; 内存前 64k 放置的中断向量表, 将段寄存器指向该处
mov es, ax
mov ax, 8h             ; 定义 8 号中断: 时钟中断
mov bx, 4
mul bx                 ; *4, 获取偏移地址
mov si, ax
mov ax, int8h
mov [es:si], ax        ; 设置时钟中断向量的偏移地址
add si, 2              ; +2, 获取中断函数入口地址
mov ax, cs
mov [es:si], ax        ; 设置时钟中断向量的段地址 = CS

```

对应的一些技术细节如下:

- 中断向量表的起始位置为 0000h, 设置前需要把段寄存器指向该处
- 时钟中断号为 8h, 偏移地址为  $8*4 = 20h$ , 段地址 (函数入口地址) 为  $8*4+2 = 22h$

8 号中断函数设计:

```

int8h:
cli                    ; 屏蔽外部中断
pusha                  ; 通用寄存器压栈
call draw              ; 调用风火轮旋转函数
mov al, 20h
out 20h, al            ; 将字节 20h 从 20h 号端口输出
out 0a0h, al           ; 中断结束命令
popa                   ; 通用寄存器出栈
sti                    ; 恢复外部中断
iret                   ; 中断结束返回

```

对应的一些技术细节如下:

- 在中断开始时, 需要 cli 指令屏蔽外部中断, 保证中断代码正常执行
- pusha 指令可以将所有通用寄存器按顺序压栈, 用于中断数据保护, popa 指令则是以对应的顺序将通用寄存器出栈, 用于中断后数据的返回
- 中断通过调用风火轮旋转函数 draw 来实现风火轮效果, 实现之后通过给 8259A 端口地址 20h 和 A0h 发送数据 20h, 标示中断结束

风火轮实现函数设计:

(其中 2-6 行参考了老师的代码, 第六行**有一处错**, 原来是 `mov [gs:((80*12+39)*2)], ax`, 显示的字符是在 12 行 39 列, 需要把 12 改成 24, 39 改成 79)

```

draw:
mov ax, 0B800h          ; 文本窗口显存起始地址
mov gs, ax              ; GS = B800h
mov ah, 0Fh             ; 0000: 黑底、1111: 亮白字 (默认值为07h)
mov al, byte[bar]        ; AL = 显示字符值 (默认值为20h=空格符)
mov [gs:((80*24+79)*2)], ax ; 写显存, 实现在 24 行 79 列显示字符
cmp byte[bar], '|'
je lslash
cmp byte[bar], '/'
je hslash
cmp byte[bar], '-'
je rslash
cmp byte[bar], '\'
je sslash
lslash:

```

```

        mov byte[bar], '/'
        ret
hslash:
        mov byte[bar], '-'
        ret
rslash:
        mov byte[bar], '\'
        ret
sslash:
        mov byte[bar], '|'
        ret

datadef:
        bar db '|'

```

该函数以 '|' -> '/' -> '-' -> '\' 为循环，不断在 24 行 79 列输出字符，实现风火轮旋转效果。

Linux 环境下使用 `nasm 4-1.asm -o 4-1.bin` 生成 bin 文件，再通过 `dd if=4-1.bin of=4-1.img conv=notrunc` 得到 4-1.img 文件后，测试结果如下图：



右下角风火轮以一秒约 4 圈的速度旋转，两次截图时分别旋转到 '-' 和 '/'

2. **步骤二：**以我的实验三程序 3-4 系列为基础，进行本次实验的步骤二

由于我的实验三已经支持了键盘指令输入，本质即为键盘中断响应，因此在该步骤二中无需增加键盘中断响应，而改为**着重增加时钟中断响应**

已有的键盘中断响应关键代码如下（c 模块）：

```
char gchar() {
    char ch;
    asm volatile("int 0x16\n" // asm 汇编内嵌编程，从键盘获取一个字符
                  : "=a"(ch) // 获取的字符放入 ch 中
                  : "a"(0x1000));
    return ch;
}

void gstr(char *s) {
    for (;;) {
        pchar(*s = gchar()); // 循环 gchar 并通过 pchar 显示输入的字符
        if (*s == '\r' || *s == '\n')
            break;
    }
    *s = '\0';
}
```

而对于时钟中断响应，可以直接将步骤一的风火轮代码植入实验三的内核即可，然而我打算加强时钟中断响应的效果和意义，因此我重新设计了另一种时钟中断响应的效果：

**数字钟：内核右下角添加一个时钟，用于记录并显示内核运行的时间（分：秒）**

具体操作步骤如下：

1. 独立版：将步骤一的“风火轮”改成“数字钟”，将 `draw` 部分代码修改替换即可

关键代码：

```
draw:
    inc byte[count]
    cmp byte[count],18           ; 延时，起到每秒显示一次的效果
    jne end
    mov byte[count],0
    inc byte[num1]                ; “秒”个位数++
    cmp byte[num1],10            ; 个位数超过 10 则十位数++
    jne show
change1:
    mov byte[num1],0             ; 重置“秒”个位数
    inc byte[num2]                ; “秒”十位数++
    cmp byte[num2],6             ; 十位数超过 6 则“分”个位数++
    jne show
change2:
    mov byte[num2],0             ; 重置“秒”十位数
    inc byte[num3]                ; “分”个位数++
    cmp byte[num3],10            ; 个位数超过 10 则“分”十位数++
    jne show
change3:
    mov byte[num3],0             ; 重置“分”个位数
    inc byte[num4]                ; “分”十位数++
    cmp byte[num4],6             ; 十位数超过 6 则重置
    jne show
change4:
    mov byte[num4],0             ; 重置“分”十位数
show:
```

```

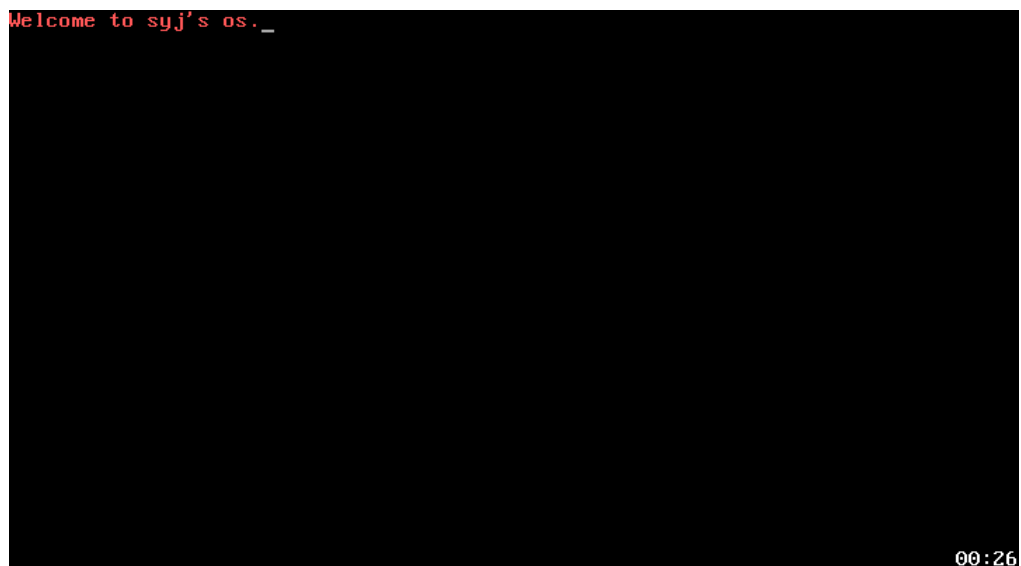
mov ax,0B800h          ; 文本窗口显存起始地址
mov gs,ax              ; GS = B800h
mov ah,0Fh             ; 0000: 黑底、1111: 亮白字（默认值为
07h）
mov al,byte[num4]
add al,48
mov [gs:((80*24+75)*2)],ax    ; 输出“分”十位数
mov al,byte[num3]
add al,48
mov [gs:((80*24+76)*2)],ax    ; 输出“分”个位数
mov al,':'
mov [gs:((80*24+77)*2)],ax    ; 输出“:”
mov al,byte[num2]
add al,48
mov [gs:((80*24+78)*2)],ax    ; 输出“秒”十位数
mov al,byte[num1]
add al,48
mov [gs:((80*24+79)*2)],ax    ; 输出“秒”个位数
end:
ret

datadef:
count db 0             ; 延时
num1 db 0              ; “秒”个位数
num2 db 0              ; “秒”十位数
num3 db 0              ; “分”个位数
num4 db 0              ; “分”十位数

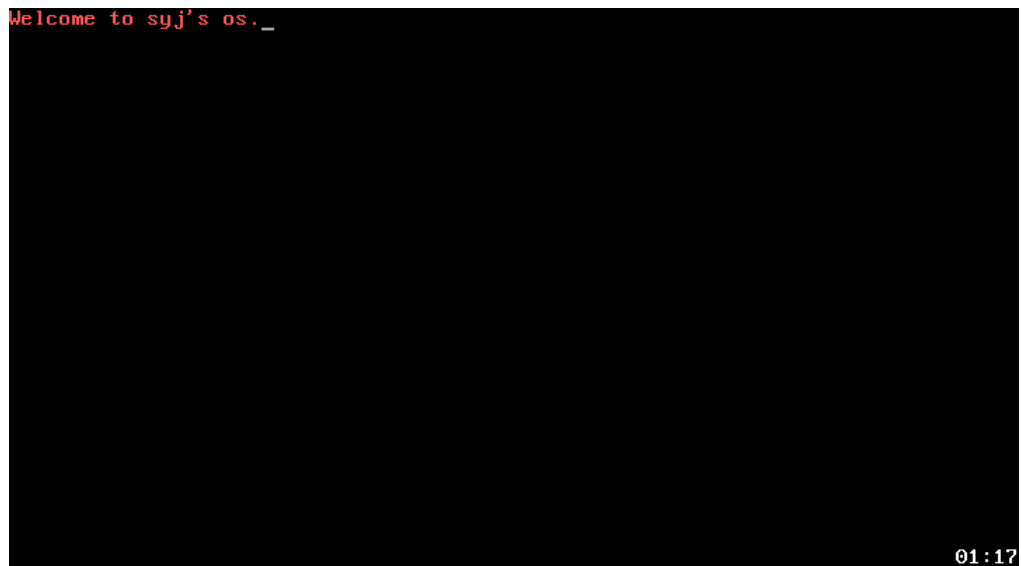
```

测试效果如下：（4-2-1.img）

当内核运行了 26 秒时，右下角数字钟显示 00:26



当内核运行了 1 分 17 秒时，右下角数字钟显示 01:17



2. 植入内核：在多次测试后，发现直接把关键代码复制进汇编模块部分是不可行的，不仅汇编模块的数字钟显示会部分乱码，而且 c 模块的提示字符串也会部分乱码（部分乱码：有的正常有的不正常），因此尝试把数字钟部分放入 c 模块，测试可行，具体步骤如下：

汇编模块：

1. 声明外部函数 `extern draw`，数字钟由 c 模块声明

2. 将 8 号时钟中断放入中断向量表：

```
mov ax, 0000h          ; 内存前 64k 放置的中断向量表，将段寄存器指向该处
mov es, ax
mov ax, 8h             ; 定义 8 号中断：时钟中断
mov bx, 4
mul bx                 ; *4，获取偏移地址
mov si, ax
mov ax, int8h
mov [es:si], ax        ; 设置时钟中断向量的偏移地址
add si, 2              ; +2，获取中断函数入口地址
mov ax, cs
mov [es:si], ax        ; 设置时钟中断向量的段地址 = CS
```

3. 定义 8 号中断函数：

```
int8h:
cli                   ; 屏蔽外部中断
push cs              ; 用于 c 模块函数返回
call draw            ; 调用数字钟显示函数
mov al, 20h
out 20h, al          ; 将字节 20h 从 20h 号端口输出
out 0a0h, al         ; 中断结束命令
sti                  ; 恢复外部中断
iret                 ; 中断结束返回
```

c 模块：

1. 声明计数变量，用于辅助时钟显示：

```
// 分别对应数字钟汇编代码里的延时显示、秒（个位、十位）、分（个位、十位）
int count=0,num1=0,num2=0,num3=0,num4=0;

// 判断变量，进入用户程序时不显示时钟
int judge=0;
```

2. 封装写显存函数，用于在屏幕指定位置显示一个字符：

```
void pchar2(char c,int x,int y) { // c: 显示字符, x: 行数, y: 列数
    asm volatile(                // asm 汇编内嵌编程
        "push es\n"
        "mov es, ax\n"
        "mov es:[bx],cx\n"
        "pop es\n"
        : // a: 显存起始地址 b: 行列位置 c: 颜色和字符
        : "a"(0xB800), "b"((x*80+y)*2), "c"((0x07<<8)+c)
        :);
    return;
}
```

3. 将汇编代码转换成 c 代码，并写入 c 模块形成函数：

```
void draw() {
    if (judge == 0) { // 用户程序时 judge = 1, 此时不显示时钟
        count++;
        if (count >= 18) { // 延时变量，控制每秒显示一次
            count = 0;
            num1++;
            if (num1 >= 10) { // “秒”个位数大于 10
                num1 = 0;
                num2++;
                if (num2 >= 6) { // “秒”十位数大于 6
                    num2 = 0;
                    num3++;
                    if (num3 >= 10) { // “分”个位数大于 10
                        num3 = 0;
                        num4++;
                        if (num4 >= 6) num4 = 0; // “分”十位数大于 6
                    }
                }
            }
        }
    }
    // 屏幕右下角显示数字钟
    pchar2((char)(num4+48),24,75);
    pchar2((char)(num3+48),24,76);
    pchar2(':',24,77);
    pchar2((char)(num2+48),24,78);
    pchar2((char)(num1+48),24,79);
}
}
```

4. 进入用户程序时要赋值判断变量 `judge = 1`，退出用户程序时 `judge = 0`

引导扇区模块和用户 asm 程序不变

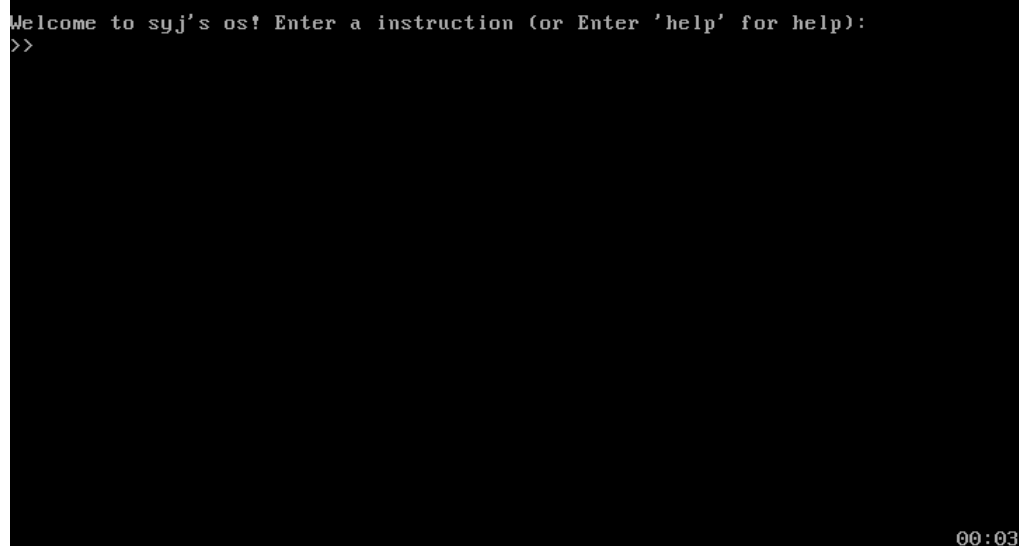
Linux 环境下使用以下编译命令行，生成 4-2.img 文件：

```
gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -  
fno-PIE -masm=intel -c 4-2c.c -o 4-2c.o  
nasm -felf 4-2asm.asm -o 4-2asm.o  
ld -m elf_i386 -N --oformat binary -Ttext 0x7e00 4-2asm.o 4-2c.o -o  
4-2.bin  
nasm 4-2os.asm -o 4-2os.bin  
  
/sbin/mkfs.msdos -C 4-2.img 1440  
dd if=1.com of=4-2.img seek=18 conv=notrunc  
dd if=2.com of=4-2.img seek=19 conv=notrunc  
dd if=3.com of=4-2.img seek=20 conv=notrunc  
dd if=4.com of=4-2.img seek=21 conv=notrunc  
dd if=4-2.bin of=4-2.img seek=1 conv=notrunc  
dd if=4-2os.bin of=4-2.img conv=notrunc
```

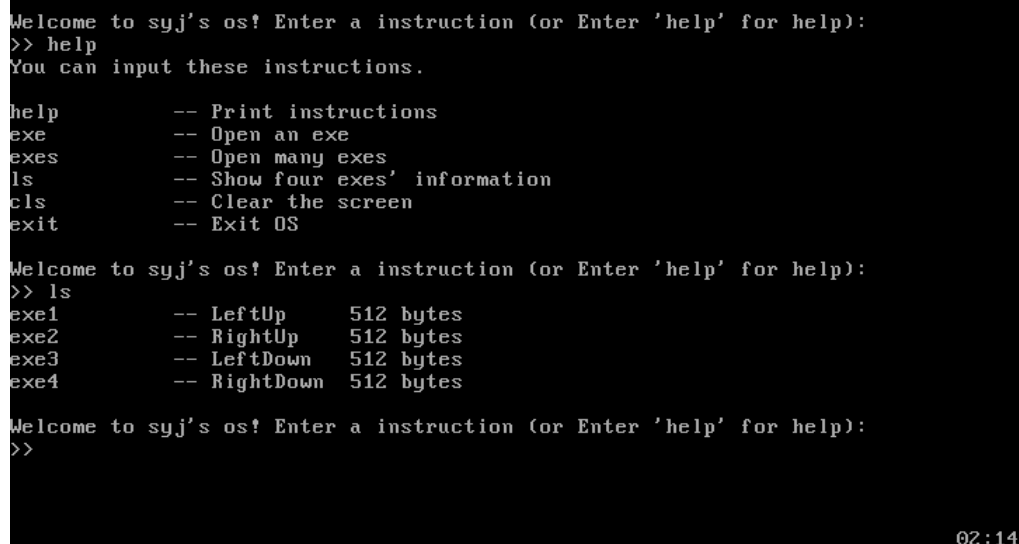
编译命令行和实验三基本一样，实验三报告中已经对参数进行技术讲解，这里不再赘述

### 3. 测试：生成 4-2.img 文件后，测试效果如下：

进入内核后，右下角数字钟开始计时：




测试 help 和 ls 指令：使用 int 16h 键盘中断响应从键盘分别读入 help 和 ls，输出提示信息后，数字钟计时为 02:14





输入 cls 指令清屏后，数字钟随着清屏指令短暂消失，1 秒后重新显示，继续计时：

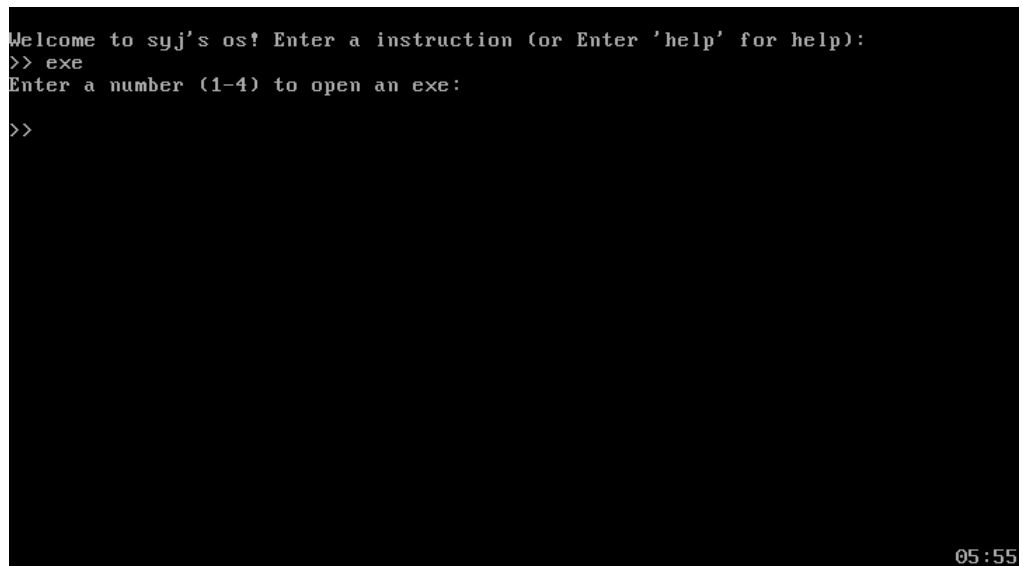
```
Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>>
```



05:22

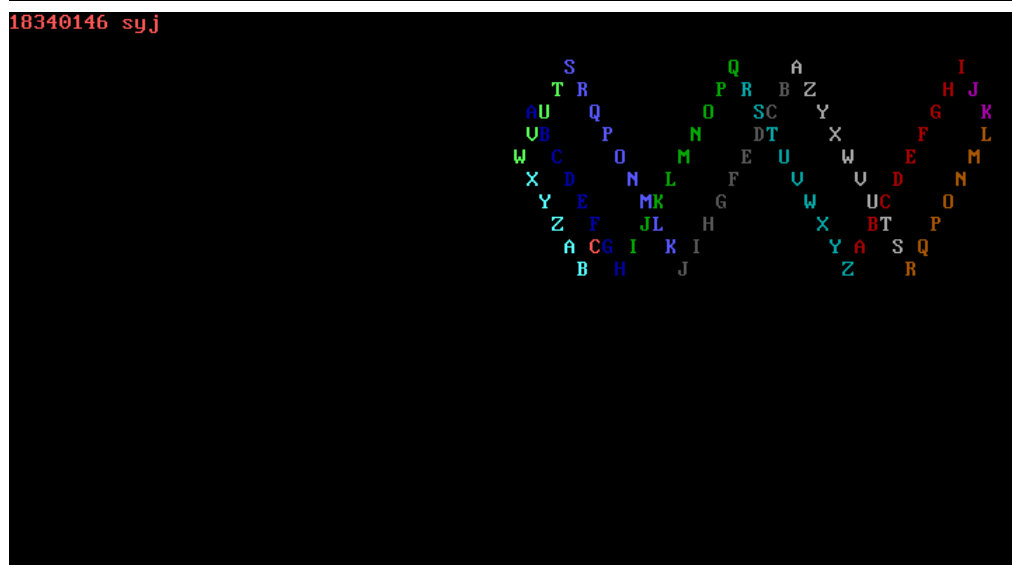
输入 exe 指令，再输入 2，调用第二个用户程序，此时数字钟不显示并停止计时，一段时间后用户程序自动结束并跳回到内核，数字钟重新显示并继续计时：

```
Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>> exe
Enter a number (1-4) to open an exe:
>>
```



05:55

```
18340146 syj
```



```
18340146 syj

S      K Q  A      I
T R    J L P R B Z  H J
AU Q   I OM SC Y   G K

Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>>
X D    F L    F P  U  U D  N
Y E    E MK   G   Q W   UC  O
Z F    JL H   R XU BT  P
A CG I K I   S UY A S Q
B H    J     T Z   R

08:03
```

输入 exes 指令，再输入自定义的用户程序执行顺序，在用户程序顺序执行期间，数字钟不显示并停止计时，一段时间后用户程序均自动结束并跳回到内核时，数字钟重新显示，并继续计时：

下图为 exes 指令输入用户程序执行顺序阶段，数字钟显示并计时：

```
18340146 syj

S      K Q  A      I
T R    J L P R B Z  H J
AU Q   I OM SC Y   G K

Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>> exes
Enter 4 number (1-4) to open 4 exes:
>> 132

A CG I K I   S UY A S Q
B H    J     T Z   R

10:26
```

下图为用户程序顺序执行阶段，数字钟不显示，停止计时：

```
18340146 syj

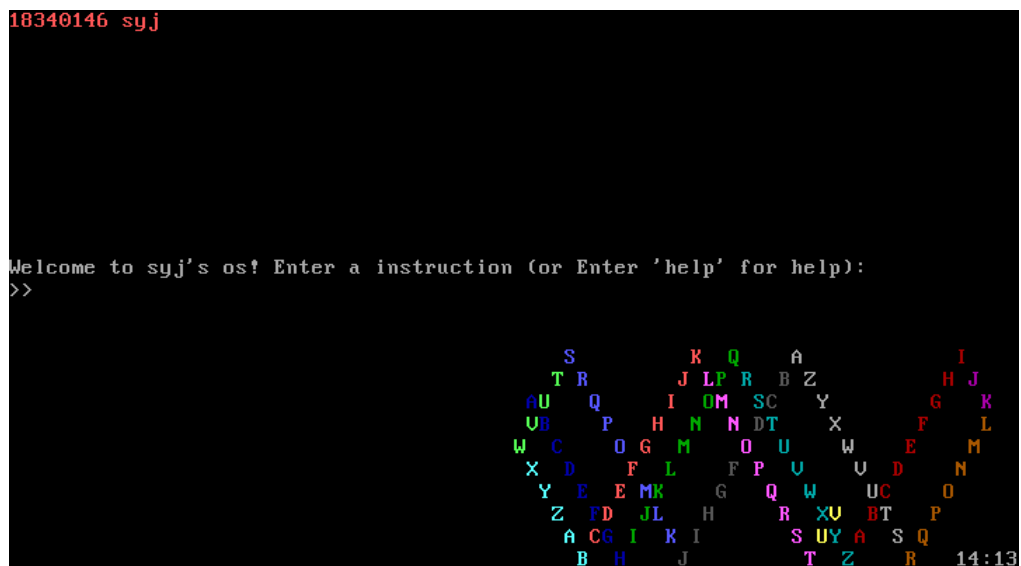
S      K Q  A      I
T R    J L P R B Z  H J
AU Q   I OM SC Y   G K

Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>> exes
Enter 4 number (1-4) to open 4 exes:
>> 132

A CG I K I   S UY A S Q
B H    J     T Z   R

10:26
```

下图为用户程序结束返回内核时，数字钟显示并继续计时：



3. **步骤三**：以步骤二为基础，添加“在用户程序执行时，可以通过键盘中断响应，在屏幕某个位置显示提示字符串OUCH!OUCH!”

由于题目要求**不能修改用户程序**，因此**不能采取在用户程序中添加键盘中断相应**这一简单的方式来实现。一番思考后决定**在时钟中断响应中添加键盘中断响应**，实现感应到键盘输入时，输出提示字符串

由于在步骤二中添加了 judge 这一判断变量，实现了进入用户程序时为这一判断变量赋值为 1，因此只需在时钟中断响应中特判进入用户程序时，不断扫描键盘缓冲区是否有输入，有的话取出并在屏幕某个位置显示提示字符串即可

具体步骤如下：

汇编模块：声明全局函数 schar，用于扫描键盘区是否有输入；声明外部判断变量 in，schar 扫描键盘有输入时赋值 1，无输入时赋值 0，用于辅助提示字符串的显示

```
global schar
extern in
schar:
    push ax
    mov ah, 01h                ; 功能号，扫描但不等待输入
    int 16h
    jz noschar
    mov word[in],1             ; 有输入，赋值为 1
    mov ah, 00h
    int 16h                    ; 从键盘输入缓冲区中取出输入
    pop ax
    ret
noschar:
    mov word[in],0             ; 无输入，赋值为 0
    pop ax
    ret
```

c 模块：声明外部函数 schar 和判断变量 in，意义如上，并扩展 draw 函数：

```
extern void schar();
int in=0;
void draw() {
    if (judge == 0) {
        ...                    // 和上文一样
    }
```

```

else {
    schar(); // 扫描键盘缓冲区
    if (in == 1) { // 有输入
        pchar2('O',xc,yc); // 输出提示字符串
        pchar2('U',xc,yc+1);
        pchar2('C',xc,yc+2);
        pchar2('H',xc,yc+3);
        pchar2('!',xc,yc+4);
        pchar2('O',xc,yc+5);
        pchar2('U',xc,yc+6);
        pchar2('C',xc,yc+7);
        pchar2('H',xc,yc+8);
        pchar2('!',xc,yc+9);
        xc = (xc+2)%24; // 更新下一次的输出位置
        yc = (yc+18)%65;
        in = 0;
    }
}
}

```

引导扇区模块和用户 asm 程序不变

Linux 环境下使用上文给出的编译命令行（注意修改文件名），生成 4-3.img 文件：

测试效果如下图：进入内核，输入指令 exe，再输入 3，打开用户程序 3。在用户程序执行过程中，键盘随意输入两次字符，屏幕内出现两次提示字符串“OUCH! OUCH!”



在用户程序自动结束并返回内核时，“OUCH! OUCH!”提示字符串不消失，内核输出新的提示输入的字符串，数字钟显示并继续计时

```
18340146 syj
OUCH!OUCH!
OUCH!OUCH!
Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>>

S      K Q      A      I
T R    J L P R B Z    H J
AU  Q    I OM SC  Y    G K
UB  P    H N N DT  X    F L
W C    O G M  O U  W    E M
X D    F L    F P U  U D  N
Y E  E MK  G  Q W  UC  O
Z  FD JL  H  R XU BT  P
A CG I K I  S UY A S Q
B H  J      T Z  R

00:09
```

再回到内核时，键盘随意输入，不再输出“OUCH! OUCH!”提示字符串，输入的字符只会显示在“输入指令”的部分












```
18340146 syj
OUCH!OUCH!
OUCH!OUCH!
Welcome to syj's os! Enter a instruction (or Enter 'help' for help):
>> abc

S      K Q      A      I
T R    J L P R B Z    H J
AU  Q    I OM SC  Y    G K
UB  P    H N N DT  X    F L
W C    O G M  O U  W    E M
X D    F L    F P U  U D  N
Y E  E MK  G  Q W  UC  O
Z  FD JL  H  R XU BT  P
A CG I K I  S UY A S Q
B H  J      T Z  R

00:17
```

4. 自己的软件项目管理目录如下图：

电脑 > 桌面 > S & W > 操作系统 > 18340146_宋渝杰_实验四_v0 >			
名称	修改日期	类型	大小
src	24/5/2020 上午 1:07	文件夹	
report.pdf	15/5/2020 下午 3:46	PDF 文件	713 KB
实验四.md	24/5/2020 上午 12:53	Markdown File	18 KB

名称	修改日期	类型	大小
 1.asm	24/5/2020 上午 12:41	Assembler Source	4 KB
 2.asm	24/5/2020 上午 12:37	Assembler Source	4 KB
 3.asm	24/5/2020 上午 12:41	Assembler Source	4 KB
 3-4.img	14/5/2020 下午 3:02	光盘映像文件	1,440 KB
 3-4asm.asm	14/5/2020 下午 2:55	Assembler Source	2 KB
 3-4c.c	14/5/2020 下午 3:01	C Source File	4 KB
 3-4os.asm	13/5/2020 下午 4:58	Assembler Source	1 KB
 4.asm	24/5/2020 上午 12:41	Assembler Source	4 KB
 4-1.asm	22/5/2020 下午 10:30	Assembler Source	2 KB
 4-1.img	22/5/2020 下午 9:52	光盘映像文件	1,440 KB
 4-2.img	24/5/2020 上午 12:43	光盘映像文件	1,440 KB
 4-2-1.asm	23/5/2020 下午 5:56	Assembler Source	3 KB
 4-2-1.img	23/5/2020 上午 12:40	光盘映像文件	1,440 KB
 4-2asm.asm	23/5/2020 下午 10:51	Assembler Source	2 KB
 4-2c.c	23/5/2020 下午 10:35	C Source File	5 KB
 4-2os.asm	23/5/2020 下午 7:33	Assembler Source	1 KB
 4-3.img	24/5/2020 上午 12:42	光盘映像文件	1,440 KB
 4-3asm.asm	24/5/2020 上午 12:26	Assembler Source	3 KB
 4-3c.c	24/5/2020 上午 12:18	C Source File	5 KB
 4-3os.asm	23/5/2020 下午 7:33	Assembler Source	1 KB
 README.md	24/5/2020 上午 1:07	Markdown File	1 KB

### 实验心得：

这次实验比之前的实验要稍微轻松一点，因为在前面的实验中已经接触过了键盘中断响应，因此主要的新内容就只有一个时钟中断响应，而且和键盘中断响应也有不少类似之处。

而对于其它技术层面的心得，我也在此一并讲述：

- **代码纠错：**这次程序在步骤一时参考了老师的写显存代码，原代码是 `mov [gs:((80*12+39)*2)],ax`，显示的字符是在 12 行 39 列，需要把 12 改成 24，39 改成 79。其他代码均未参考老师的代码。
- **实验过程中出现的出错问题及解决方式：**一个问题出现在时钟中断的频率，我上网搜索了很久，都找不到相关的频率描述，个人手动计时器测试后，发现 1 秒约中断 18 次，同时借用了同学的电脑，在不同的电脑上测试，频率也相同。因此数字钟采用每 18 次调用改变一次显示的方式。

```
void draw() {  
    if (judge == 0) { // 用户程序时 judge = 1, 此时不显示时钟  
        count++;  
        if (count >= 18) { // 延时变量，控制每秒显示一次  
            ...  
        }  
    }  
}
```

还有一个问题是：在步骤二中，一开始将数字钟的汇编代码直接复制植入内核的汇编模块时，时钟中断调用 draw 函数显示时钟时，会出现时钟数据乱码情况，而且还会影响到 c 模块的数据显示，但是如果不是通过时钟中断，而是通过别的方式调用 draw 函数（甚至是在用户程序运行时，时钟中断调用 draw 函数），均不会出现乱码情况。因此我也感觉非常的奇怪，采用各种数据段定位方式做了代码的修改，也没能成功，于是考虑把 draw 函数转换成 c 代码形式植入内核的 c 模块部分，测试后发现能正常显示了。于是就植入 c 模块，通过汇编模块时钟中断调用 c 函数来进行“数字钟”的显示。

同时，考虑到用户程序运行时，时钟的计时数据也许会存在不同步的情况（亲自测试后发现确实不同步），因此在运行用户程序时，时钟不显示并停止计时，到了用户程序结束之后，再显示时钟并继续计时。

- **步骤三的思考过程：**一开始看到步骤三时，一个很简单的思路就是在用户程序运行，显示反弹字符之后，调用一次键盘中断响应，检测键盘缓冲区是否有输入，有的话就显示提示字符串。后来发现要求不改变用户程序，就打消了这个想法。于是我就在思考在内核跳转到用户程序后，以什么形式才能调用键盘中断 int 16h 呢。在步骤二实验过程中，发现时钟中断在跳转到用户程序时仍然能继续中断，因此把键盘中断写入到时钟中断中，特判到用户程序进行时再运行检测键盘缓冲区代码即可。

而整个实验下来，也是存在着一些技术上的问题并没有得到完美地解决：

- 上文提到的将数字钟的汇编代码直接复制植入内核的汇编模块，通过时钟中断调用 draw 函数显示时钟时，出现时钟数据乱码情况的技术性原因，还是没有搞懂
- “数字钟”的频率，也许没有达到完美的 1 秒钟显示 1 次，可能会有些许误差