

操作系统实验总结

作者：zxjou

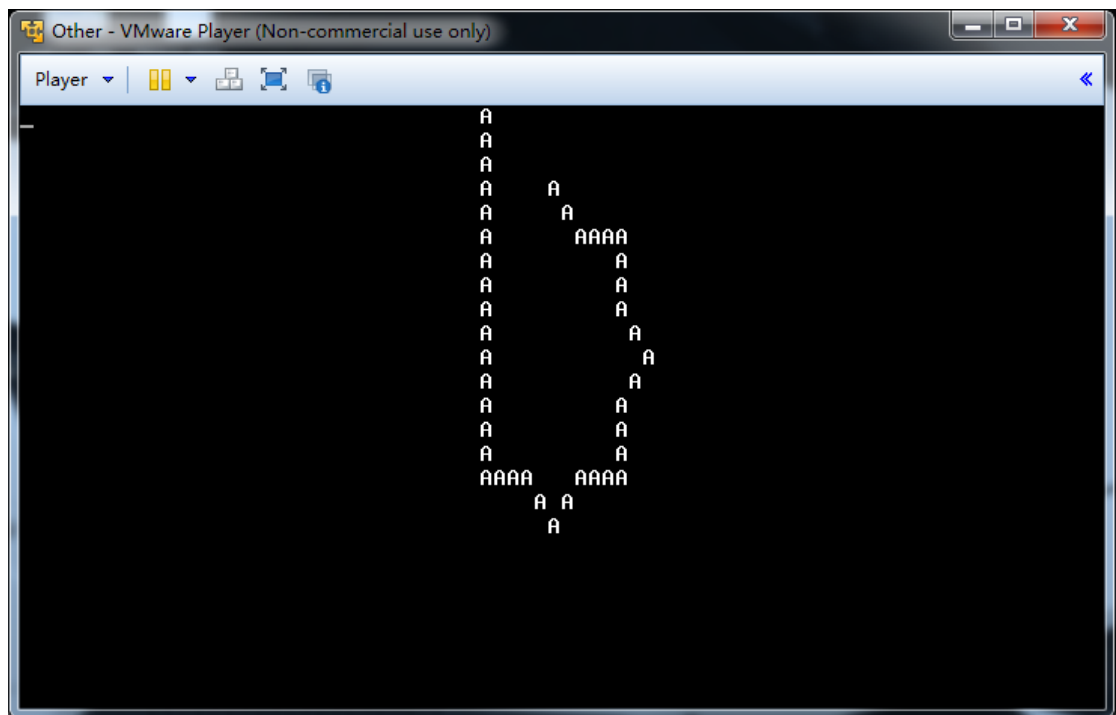
对于操作系统实验，基本上占据了我大约一半的学习时间，每次都在寻找 bug 以及 debug 当中度过，每一次的实验都是倾尽心血，一点点的改代码知道自己的满意，当得出结果时的喜悦是没有经历过的人感受不到的，下面就说说我在这几个实验当中经过的努力取得的成果。

实验一：

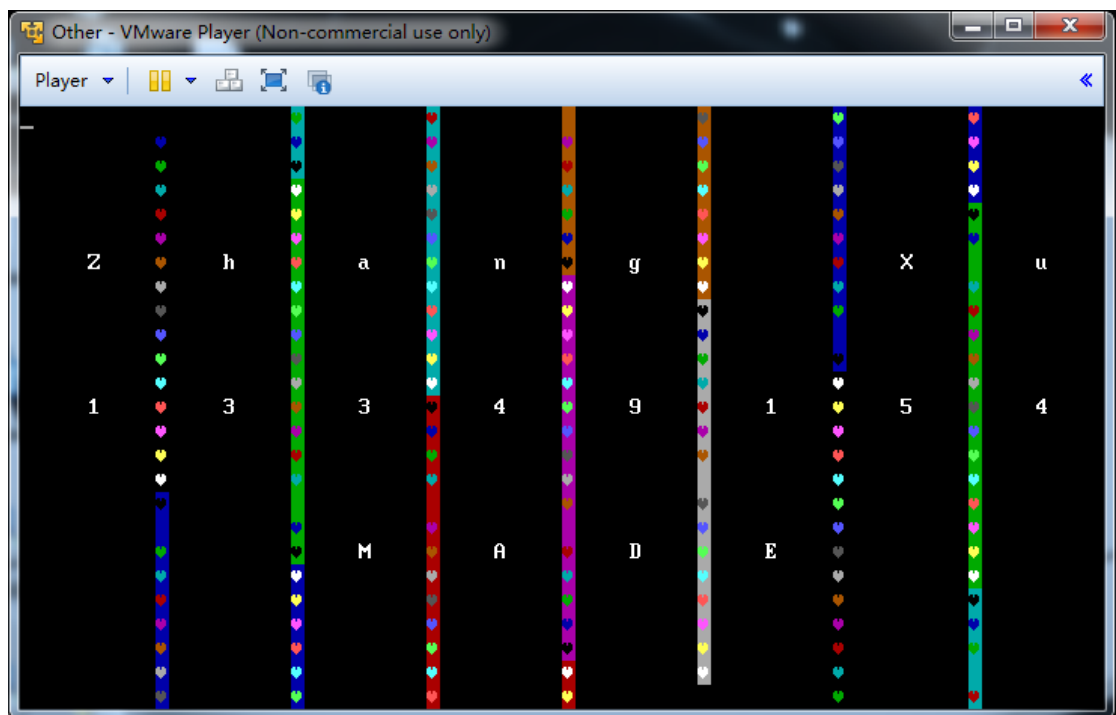
这个实验，老师只是给出一个 stone 的范例，要求我们自己写出一个类似的可以跑出的画面效果，当做启动的引导扇区。对于刚刚接触操作系统的我来说要查的和学习的知识实在是太多了，首先老师给的源代码中 org 100h 是在 DOS 或者 window 下运行需要加载的位置，在 BIOS 下这个是不对的，全班所有都不能出来结果，最后我看书和查资料搞到半夜两点，搞懂了引导扇区加载程序是在 7c00h 段地址处，我试了试终于可以做到，然后我把这个结果告诉了其他人分享了一下。

然后我自己写了一个比较复杂的代码，准备在画面上显示一个六芒星形状的东西，在代码写好并 debug 后在 VM 上不能运行毫无结果，这让我很苦恼，搞了很久还是不行，然后第二天和同学交流了一下然后找老师确认软盘引导扇区不能超过 512KB 字节，否则无效。老师说引导扇区顾名思义，它的职责只能是引导进入操作系统，并不需要太复杂，只能是 512 字节。我的代码较为复杂早已超过，没办法只能再想其他的界面效果

因为引导扇区只会执行到 512 字节处，所以我先前设计的比较复杂的图形就不可以显示正常了，下面是失败的截图：



第二天我自己又去做了另外一个界面也就是下面的那个结果图，这一次比较好做，自己控制住了代码的长度，并且把示例代码 `stone.asm` 做了大幅修改，自己用了几个子程序以免代码的重复，并且优化了代码布局，显得思路条理更加的清晰了，这里遇到的问题主要是 NASM 编译的语法和我之前 MASM 的语法有出入，很多语法都需要自己去查然后测试，这里又花费了一些时间，最终将实验做好了也是松了一口气。



其中为了这个图形，我想了很多，比如先是什么样的字符比较好，我选择的是 ASCII 码值为 3 的心形，这个显示比较自然也比较好看，为了不枯燥，我写了一个循环当显示字符的时候，每一个字符的颜色和背景都是不断变化的，并且这个图形是不断由上往下然后再由下往上不断循环显示的。

具体代码见附件代码，这里不详细说了，主要是几个循环以及判断条件，并且将临界条件控制的很好，没有 bug。

实验二：

这个实验我对实验一的内容作了大幅度的修改，我重新制作了一个界面，显示的是由 03h 心形组合成的更大的心形，每一层有不同的颜色组成，因为我很喜欢苹果，很崇拜乔布斯，所以这 6 种颜色是完全仿照苹果最原始的 logo 颜色分布，下面是苹果 logo 和我制作的效果图对比：





说实话这次实验我自己做了整整一星期，关键是我把上一个实验做的东西给摒弃了，重新做了一个界面，这个界面图形的设计以及输出耗费了我好多的时间，显示的代码不能太长否则超过 510 字节的话就不能起到引导的作用了，所以我一直在精简代码，写了一个二重循环显示，按理说不是很难。。但是我在这里犯了一个小错误，导致我 debug 了好长的时间，那就是读取数据是 dw 的，所以地址每次是加 2，而我一直是加 1，显示一直不正常，还有乱码的存在，我搞到凌晨 4 点才发现。。

然后就是操作界面代码，在这个阶段主要是 BIOS 的输入输出，每次输入我都将输入的数据存在一个预先的位置，当用的时候在读取就可以了，这里的关键也是难点就是我们需要按照用户的要求顺序执行程序，那就需要执行完一个程序再跳回操作程序处，再接着下一个程序执行直至全部执行完毕。那么问题来了，如何去跳转呢，这就需要先预先编译操作界面代码获得跳转处的实际内存地址，然后子程序执行完之后再跳转回去。这一点也是和另一位同学讨论得到的方法，然后去询问老师，老师说确实也只能这么做，最多自己去写一个中断，但是内部机理还是这个。在这里我还犯了一个错误导致我又搞了几天。。。那就是我在循环执行代码的时候，用 `cmp ax, [order+bx]`，其实我定义 `order db 0,0,0`，那就意味着这两个类型不符，我当时不知道怎么想的认为从 order 处读出的数据会在

高字节自动补 0。。。其实会在下一个字节读数据。。。然后程序运行的就不正常了。。。其实这点小问题真的是不该犯啊，一定要加强细节之处的处理，不要想当然，做到滴水不漏。下面是我认为比较有特点的引导扇区实现的代码：

```
org 7c00h          ; BIOS 将把引导扇区加载到 0:7C00h 处，并开始执行
Offset0fUserPrg1 equ 8100h
```

Start:

```
    mov     ax, 0600h
    mov     bx, 0700h
    mov     dx, 184fh
    int     10h

    mov ax, cs
    mov ds, ax          ; DS = CS
    mov es, ax          ; ES = CS
    mov ax, 0B800h      ; 文本窗口显存起始地址
    mov gs, ax          ; GS = B800h
    mov bx, 0
```

loop1:

```
    push bx              ; 入栈保护
    mov ax, word[row+bx] ; 计算显示行数
    mov word[x], ax
    mov ax, word[col+bx] ; 计算显示列数
    mov word[y], ax
    mov cx, 0
```

show:

```
    push bx              ; 入栈保护
    call delays           ; 延时程序
    call cal              ; 计算显示位置
```

```

inc word[y]                ; 右移显示字符串
mov ah, [color+bx]         ; 0000: 黑底、变换前景颜色
mov di, [image+bx]         ; di=当前串的偏移地址
mov bx, cx
mov al, [di+bx]            ; AL = 显示字符值 (默认值为 20h=空格符)
mov word[gs:bp], ax        ; 显示字符的 ASCII 码值
inc cx
pop bx                     ; 出栈还原 bx
mov ax, word[number+bx]
cmp ax, cx
jz  loop2
jmp show

loop2:
pop bx                     ; 出栈还原 bx
inc bx
inc bx
mov ax, 16                 ; 循环显示 8 次字符串结束
cmp ax, bx
jz  LoadnEx               ; 显示结束
jmp loop1                 ; 返回上一层

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
mov bx, Offset0fUserPrg1   ; 偏移地址    ; 存放数据的内存偏移地
址
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                ; 起始扇区号    ; 起始编号为 1
```

```
    int 13H                  ; 调用读磁盘 BIOS 的 13h 功能
```

```
    ; 用户程序 a.com 已加载到指定内存区域中
```

```
    call delayss
```

```
    jmp OffSetOfUserPrg1
```

```
    jmp $                    ;无限循环
```

```
cal:
```

```
    push bx                  ; 计算显存地址
```

```
    xor ax, ax
```

```
    mov ax, word[x]
```

```
    mov bx, 80
```

```
    mul bx
```

```
    add ax, word[y]
```

```
    mov bx, 2
```

```
    mul bx
```

```
    mov bp, ax
```

```
    pop bx
```

```
    ret
```

```
delays:                        ; 延时程序
```

```
again:
```

```
    dec word[count]          ; 递减计数变量
```

```
    jnz again                ; >0, 跳转;
```

```
    mov word[count], delay
```

```

    dec word[dcount]                ; 递减计数变量
    jnz again                      ; >0, 跳转;
    mov word[count], delay
    mov word[dcount], ddelay
    ret

delayss:                          ; 长时间延时程序
    mov cx, 20
continue:
    call delays
    loop continue
    ret

delay equ 40000                   ; 计时器延迟计数, 用于控制画框的速度
ddelay equ 500                   ; 计时器延迟计数, 用于控制画框的速度
count dw delay
dcount dw ddelay
x dw 0
y dw 0
img1 db 3, ' ', 3                ; 显示的图形
img2 db 3, 3, 3, ' ', 3, 3, 3
img3 db 3, 3, 3, 3, 3, 3, 3, 3, 3
img4 db 3, 3, 3, 3, 3, 3, 3
img5 db 3, 3, 3
img6 db 3
img7 db 'Welcome to Original OS that made by ZX.' ; 显示的提示语
img8 db 'Please wait several secs, the OS is loading...'
number dw 5, 7, 9, 7, 3, 1, 39, 45
image dw img1, img2, img3, img4, img5, img6, img7, img8

```

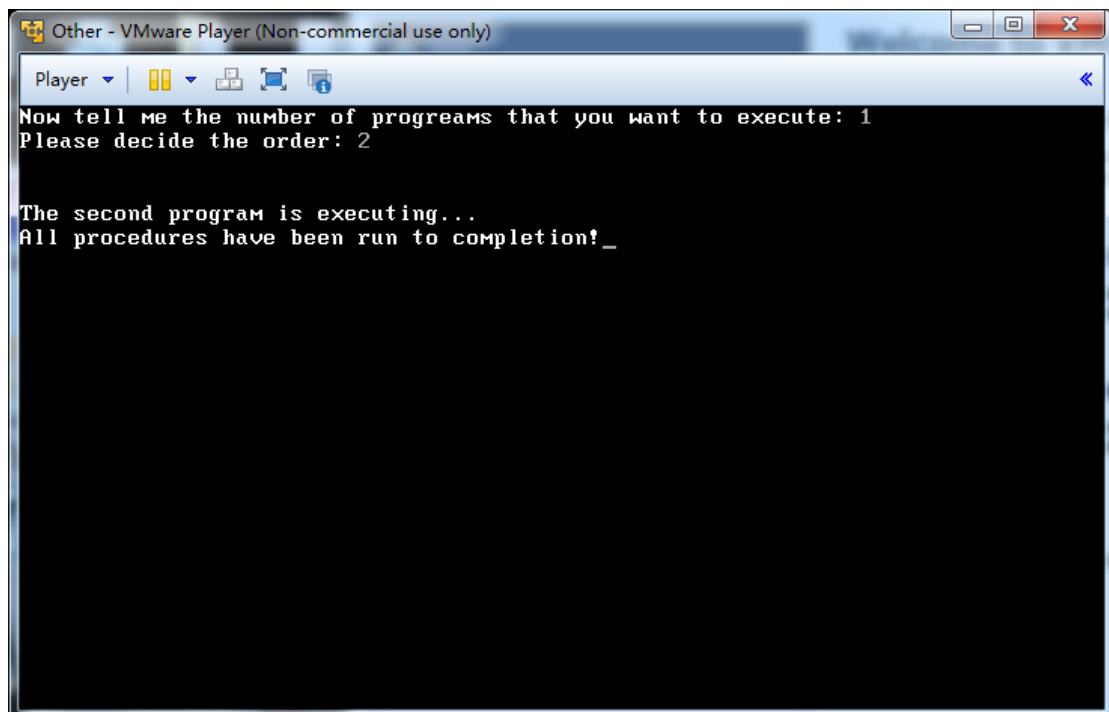


```
row dw 8, 9, 10, 11, 12, 13, 17, 19
col dw 37, 36, 35, 36, 38, 39, 10, 10
color dw 0ah, 0eh, 06h, 0ch, 0dh, 0bh, 0fh, 0fh

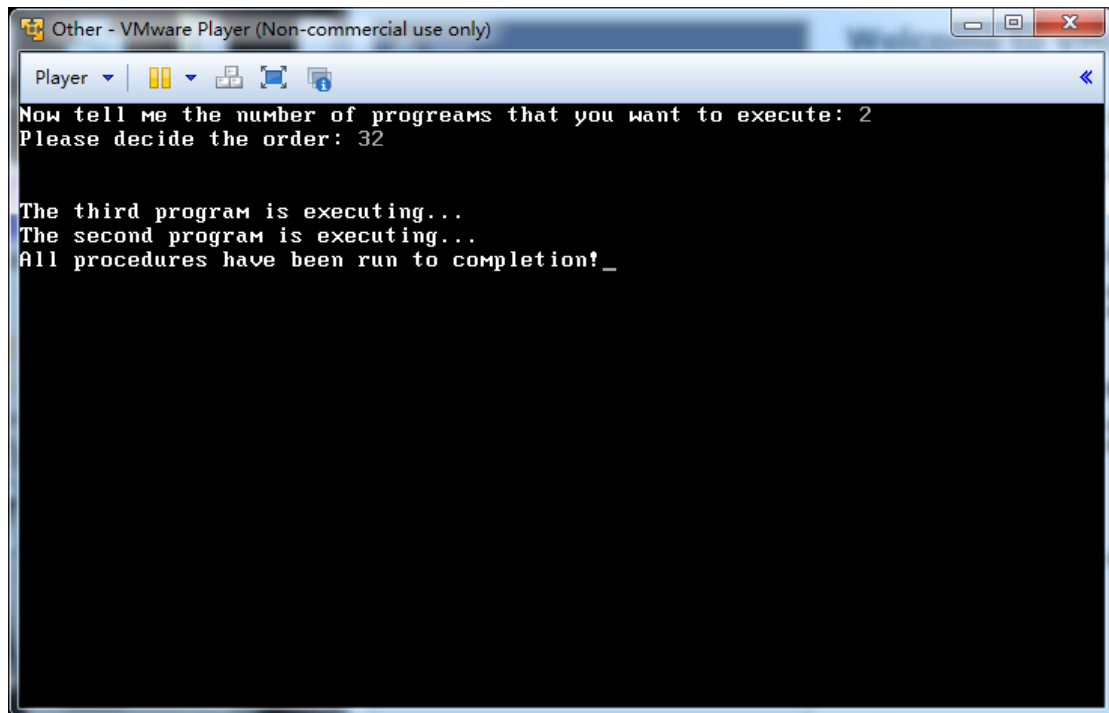
times 510-($-$$) db 0          ; 补全引导扇区
db 0x55, 0xaa
```

个人认为我自己写的这个引导扇区界面实现的循环十分的简洁，没有多余无用的代码。关键在于我红色标记的那几段代码，这几段代码是把要输出的语句、位置以及颜色都做了综合存储，节省了大量的重复代码，使代码简洁易看以及更加简洁。还有那三个程序的代码也会在代码附件当中，老师可以去自行查看。完全是按照老师的要求随便输入执行程序顺序，然后按照顺序执行每个程序，个人感觉没什么很大的特点，就不多说了。下面给出程序执行部分测试结果的截图：

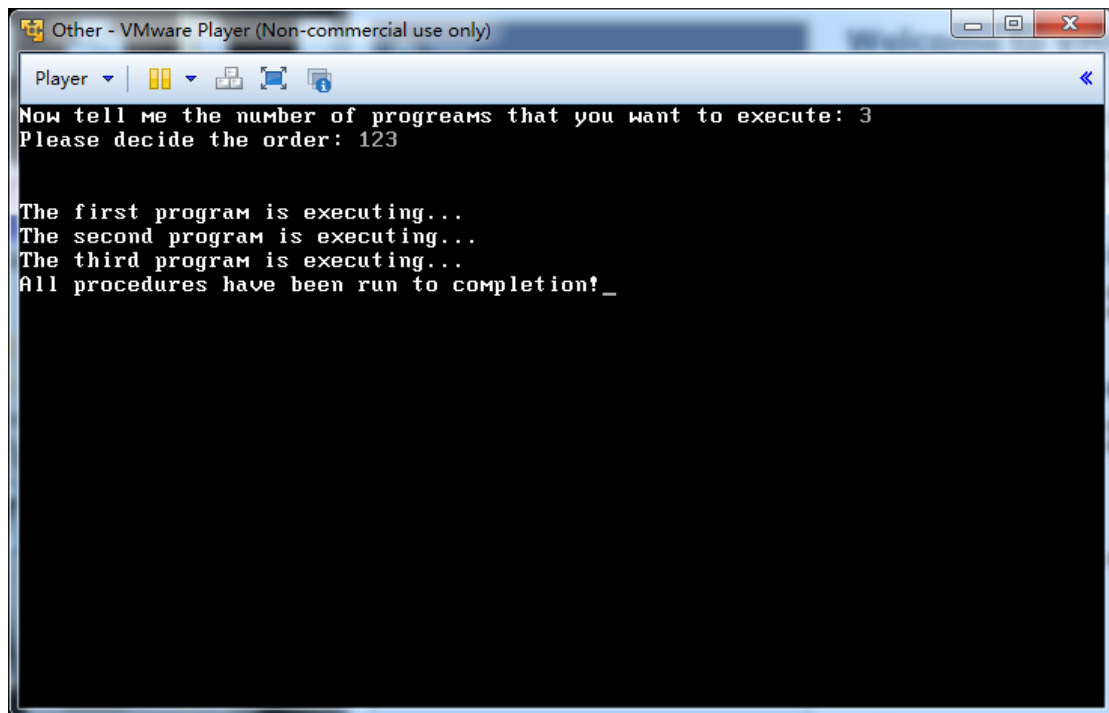
运行一个程序：

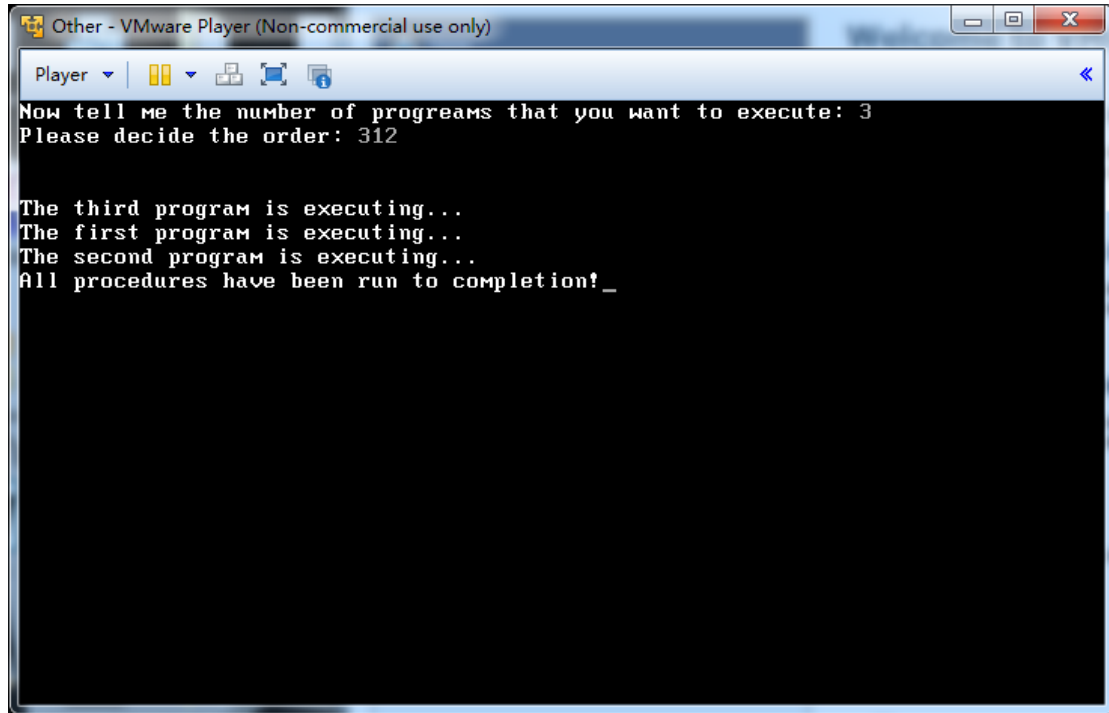


运行两个程序：



运行三个程序：





实验三：

这次实验引导程序开机界面没有变化，我主要对实验二的内核部分完全进行了改写，增加了一些功能、更加友好的界面以及指令，符合实验要求将.c 文件和.asm 文件进行联合编译生成.com 文件然后嵌入软盘中制作内核。这次内核是关键部分，用 TASM 和 TCC 结合生成.com，其他的用户程序因为用不到.c 文件，不需要连接，所以暂且还用比较简便的 NASM 来进行编译。

我的内核部分主要是模仿了 DOS 系统，有比较好的用户界面，还有一定的错误提示和越界刷新的功能，因为我感觉作为一个操作系统，良好的界面是必须的，所以我就完全自己一个人做了一个仿照 DOS 的界面系统用来输入指令。输入 help 指令可以查看详细的指令列表以及功能，如下图所示：

```
These are the available commands:

_help          show you all the commands
-restart       restart the OS
-shutdown      shutdown the OS
-time         show you the real time
-run proc1     run the prgream1
-run proc2     run the prgream2
-run proc3     run the prgream3

You can press the Enter key to return.
```

还有这次老师要求内核与引导程序，还要求写一个显示时间和日期的程序，然后自己还写了三个简单的小程序用来展示内核对系统的调用，对于那个显示时间的程序一开始是不会的，但是后来查找资料发现其实可以调用 BIOS 里面的 1Ah 中断来返回当前的时间和日期，这就比较简单了，但是如何动态显示时间呢，在这点上我自己想到的是动态刷新时间，我自己写了一个循环不断显示时间，但是这存在一个问题，那就是不断地刷新的话电脑会受不了，而且我需要不运行的时候要返回我自己的内核界面，这怎么办呢，我自己又想了好久，想到了一种方法，那就是要在每次循环当中检测是否有按键按下，如果有的话就退出返回内核界面，否则就继续显示时间。下面是我自己写的 time 显示代码：

```
org 8b00h                                ; BIOS 将把引导扇区加载到 0:8b00h 处，并开始执行
```

```
start:

    mov ax, 0600h
    mov bx, 0700h
    mov cx, 0h
    mov dx, 184fh
    int 10h
```

mov ax, cs	; 置其他段寄存器值与 CS 相同
mov ds, ax	; 数据段
mov es, ax	; 置 ES=DS
mov bp, message1	; BP=当前串的偏移地址
mov cx, length1	; CX = 串长
mov ax, 1300h	; AH = 13h (功能号)、AL = 00h (无光标)
mov bx, 0fh	; 页号为 0 (BH = 0) 黑底亮白字 (BL = 0fh)
mov dh, 14	; 行号=0
mov dl, 24	; 列号=0
int 10h	; BIOS 的 10h 功能: 显示一行字符

lp:

mov ax, cs	
mov ds, ax	; DS = CS
mov es, ax	; ES = CS
mov ah, 04h	
int 1Ah	
mov al, cl	
call bcd_to_ascii	
mov [message], ah	
mov [message+1], al	
mov al, dh	
call bcd_to_ascii	
mov [message+3], ah	
mov [message+4], al	
mov al, dl	
call bcd_to_ascii	
mov [message+6], ah	
mov [message+7], al	

```

mov ah, 02h
int 1Ah
mov al, ch
call bcd_to_ascii
mov [message+9], ah
mov [message+10], al
mov al, cl
call bcd_to_ascii
mov [message+12], ah
mov [message+13], al
mov al, dh
call bcd_to_ascii
mov [message+15], ah
mov [message+16], al

```

```

mov ax, ds                ; ES:BP = 串地址
mov es, ax                ; 置 ES=DS
mov bp, message           ; BP=当前串的偏移地址
mov cx, length            ; CX = 串长 (=10)
mov ax, 1300h             ; AH = 13h (功能号)、AL = 00h (无光标)
mov bx, 0fh               ; 页号为 0 (BH = 0) 黑底亮白字 (BL = 0fh)
mov dh, 12                ; 行号=12
mov dl, 31                ; 列号=31
int 10h                   ; BIOS 的 10h 功能: 显示一行字符
call delays
mov ah, 01h
int 16h
jz lp
mov ah, 00h

```

```
int 16h
jmp 8100h
```

```
bcd_to_ascii:                ;BCD 码转 ASCII
                               ;输入: AL=bcd 码
                               ;输出: AX=ascii

    mov ah, al                ;分拆成两个数字
    and al, 0x0f              ;仅保留低 4 位
    add al, 0x30              ;转换成 ASCII

    shr ah, 4                 ;逻辑右移 4 位
    and ah, 0x0f
    add ah, 0x30
    ret
```

```
delays:                       ; 延时程序
again:
    dec word[count]           ; 递减计数变量
    jnz again                 ; >0, 跳转;
    mov word[count], delay
    dec word[dcount]          ; 递减计数变量
    jnz again                 ; >0, 跳转;
    mov word[count], delay
    mov word[dcount], ddelay
    ret
```

```
delay equ 4000                ; 计时器延迟计数
ddelay equ 500                ; 计时器延迟计数
count dw delay
```

```

dcount dw ddelay
message db 0,0,'/',0,0,'/',0,0,' ',0,0,':',0,0,':',0,0
length equ ($-message)
message1 db 'You can press any key to return.' ; 显示的提示语
length1 equ ($-message1)

```

在这段代码里面，我认为比较重要的代码就是我红色标记出来的 `mov ah, 01h` 和 `int 16h`，这两句虽然代码量不多，但是起到了检测是否有按键按下的效果，其中的 `16h` 的 `01h` 功能就是检测缓冲区是否内输入内容，有的话就返回 1，否则就返回 0，那么如果你不想再显示时间的话就可以按下任意键用来返回内核操作界面，十分的简单。还有一处我认为比较重要的就是另外一处我红色标记的代码，这段代码主要是用来将 BCD 码转换成 ASCII 码，这是显示时间至关重要的一步，主要思想就是分开提取，主要步骤都在注释当中了。还有其他内核界面的代码我就不展示了，老师可以去我提供的代码附件当中查看，其实这才是我花费时间最长的，当中有很多循环调用，限于篇幅只说一些最重要的吧。

这次试验我自己又做了整整一星期，关键是这次内核部分代码不可用 NASM，要换用新的 TASM 和 TCC 的组合来编译 c 文件和 asm 文件，语法又和 NASM 有冲突，所以还要熟悉琢磨 TASM 的语法以及熟悉和 c 文件之间函数调用的方法。因为这次实验比较复杂而且要用到很多 32 位的程序，所以我干脆在虚拟机上装了一个 XP 系统，然后复制一些必要的实验软件我就可以进行方便的实验了。

这次内核的重写，我把用到的程序以及输出全部封装成函数，只要直接调用就可以了。主函数在 c 文件里面，当运行时，汇编 asm 文件会调用 c 文件里面的 Main 函数。这里在调用时首先遇到的问题老师 PPT 上面有讲，在 asm 文件中要事先声明要调用的 c 文件的函数或者是 c 文件中要调用 asm 中的函数，这些函数前面必须加下划线，因为在编译时 c 会自动在函数前面加上下划线。另外一个老师没说的是当 c 文件调用 asm 文件时，在 asm 里面的函数前必须加上 `public` + 函数名，这点老师没说，自己编译出错琢磨查找资料搞定。还有比如 `jmp` 跳转不可以直接是立即数之类的语法问题真的是遇到很多很多，每次调试又要都全部编译运行，真的是很耗时。

实验四：

这次试验和实验三基本上是一样的框架，但是在实验三的基础上修改了 int 08h、int 09h 以及自己需要写 int 33h、34h、35h、36h 中断，自己把原来的代码修改了一下，help 中多了一条指令-int，也就是对上面 4 个中断的调用。实验要求改写 int 08h 中断输出\、|、/的循环显示，但是我自己不想做这么简单，我直接让时钟在右下角输出时间。然后对于 int 09h 中断的改写，我是在时间的上方输出” OUCH! OUCH! ”，而且每当有按键按下时会有 6 种不同颜色的变换。对于 int 33h、34h、35h、36h 中断的编写，我是直接按照要求在各四分之一处显示出各个中断的调用。

内核操作界面如下：



这次实验完全是在实验三的基础上面添加中断程序，说难的话也不是很难，但是也没有老师说的那么简单，有很多的知识我们不知道，这就导致了我们会遇到许多的“坑”，需要自己去摸索。我是从中午开始一直做到晚上 1 点才做完的，这期间遇到了很多的问题，也和同学们一起探讨了，自己也学到了很多。

在做中断时，我首先是把终端的定义放在我的内核里面，然后发现我的程序死在了那里，既不能按键也不能运行，我搞了好久都未果，那我我想到将中断的声明放在引导程序和内核程序的中间，因为中断本身就是开机定义的东西放在内

核里面也不是很合适，而且这样做可以把中断与内核完全分开，debug 也是很方便，所以我的中断是放在引导程序和内核程序的中间的。对于 int 08h 因为我要输出那个时间，之前在子程序中实现过时间的输出，但是一放进中断中就各种问题，自己调试了好久，显示自动卷屏输出，然后是光标莫名其妙的在移动，这里面都需要自己一次一次的 debug。Int 09h 的问题更多，首先是吧原来的 int 09h 覆盖，所以不能往缓冲区里面写东西，那么我想到可以把原来的 int 09h 在中断向量表中的地址先取出来，然后可以定义 int 37h 是原来的 int 09h，那么我在调用修改之后的 int 09h 中首先调用已经写好的 int 37h，然后再输出我想要的“OUCH! OUCH! ”，这样就完美无缺了。但是事事难行，我在输出字符串的时候直接调用 mov ax,cs、mov ds,ax、mov es,ax 就会出问题，因为这里的 cs 是原来运行程序处的 cs，与这里的 cs 是不符的，那么 ds 以及 es 段内读取的数据就是错误的，所以这里我在定义中断前先存了本段的 cs 在 mark 中，然后在调用 mark 给 cs 就可以了，这里我发现各个程序中 cs 是不同的，这与以前我认为都是 0 的思想完全不一样，这是一个新发现（不知道是不是正确的，但是对于我遇到的问题我发现的这个解释是正确的）。下面是有关中断的代码：

```
org 7e00h                                ; BIOS 将把引导扇区加载到 0:7e00h 处，并  
开始执行
```

```
start:
```

```
    xor ax,ax                            ; AX = 0  
    mov es,ax                            ; ES = 0  
    mov si,[es:24h]  
    mov word [es:0dch],si  
    mov di,[es:26h]  
    mov word [es:0deh],di  
    mov word [es:20h],Timer               ; 设置时钟中断向量的偏移地址  
    mov word [es:24h],Key                 ; 设置键盘输入中断向量的偏移地址  
    mov word [es:0cch],Int33              ; 设置 Int33 中断向量的偏移地址
```

```

mov word [es:0d0h], Int34      ; 设置 Int34 中断向量的偏移地址
mov word [es:0d4h], Int35      ; 设置 Int35 中断向量的偏移地址
mov word [es:0d8h], Int36      ; 设置 Int36 中断向量的偏移地址
mov ax, cs
mov word [es:22h], ax          ; 设置时钟中断向量的段地址=CS
mov word [es:26h], ax          ; 设置键盘输入中断向量的段地址=CS
mov word [es:0ceh], ax         ; 设置 Int33 中断向量的段地址=CS
mov word [es:0d2h], ax         ; 设置 Int34 中断向量的段地址=CS
mov word [es:0d6h], ax         ; 设置 Int35 中断向量的段地址=CS
mov word [es:0dah], ax         ; 设置 Int36 中断向量的段地址=CS
mov word [mark], ax            ; 将 cs 地址存储在 mark 当中
mov ds, ax
mov es, ax                     ; 设置段地址
mov bx, 8400h                  ; 偏移地址      ; 存放数据的内存偏移地址
mov ah, 2                      ; 功能号
mov al, 3                      ; 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, 5                      ; 起始扇区号  ; 起始编号为 1
int 13h                        ; 调用读磁盘 BIOS 的 13h 功能
jmp 8400h

color dw 0ah, 0eh, 06h, 0ch, 0dh, 0bh
count dw 0
mark dw 0

```

Timer:

```
pusha
push ds
push es
mov ah, 04h
int 1Ah
mov al, cl
call bcd_to_ascii
mov [message1], ah
mov [message1+1], al
mov byte[message1+2], '/'
mov al, dh
call bcd_to_ascii
mov [message1+3], ah
mov [message1+4], al
mov byte[message1+5], '/'
mov al, dl
call bcd_to_ascii
mov [message1+6], ah
mov [message1+7], al

mov ah, 02h
int 1Ah
mov al, ch
call bcd_to_ascii
mov [message1+9], ah
mov [message1+10], al
mov byte[message1+11], ':'
mov al, cl
call bcd_to_ascii
```

```

mov [message1+12],ah
mov [message1+13],al
mov byte[message1+14],':'
mov al,dh
call bcd_to_ascii
mov [message1+15],ah
mov [message1+16],al

mov ax, ds                ; 数据段
mov es, ax                ; 置 ES=DS
mov bp, message1          ; BP=当前串的偏移地址
mov cx, length1           ; CX = 串长
mov ax, 1300h              ; AH = 13h (功能号)、AL = 00h (无光标)
mov bx, 000fh              ; 页号为 0 (BH = 0) 黑底亮白字 (BL = 0fh)
mov dh, 24                 ; 行号
mov dl, 62                 ; 列号
int 10h                   ; BIOS 的 10h 功能: 显示一行字符

mov al,20h                 ; AL = EOI
out 20h,al                 ; 发送 EOI 到主 8529A
out 0A0h,al                ; 发送 EOI 到从 8529A
pop es
pop ds
popa
iret

```

Key:

pusha

push ds

push es

sti

int 37h

mov ax, word[mark]

mov ds, ax ; 数据段

mov es, ax ; 置 ES=DS

mov bp, message2 ; BP=当前串的偏移地址

mov cx, length2 ; CX = 串长

mov ax, 1300h ; AH = 13h (功能号)、AL = 00h (无光标)

mov si, [count]

mov bx, [color+si] ; 页号为 0 (BH = 0) 黑底亮白字 (BL = 0fh)

inc byte[count]

inc byte[count]

mov dl, byte[count]

cmp dl, 0ch

jnz hehe

mov word[count], 0

hehe:

mov dh, 23 ; 行号

mov dl, 62 ; 列号

int 10h

mov al, 20h ; AL = EOI

out 20h, al ; 发送 EOI 到主 8529A

out 0A0h, al ; 发送 EOI 到从 8529A

pop es

pop ds

popa

iret

Int33:

```
pusha
push ds
push es
mov ax, word[mark]
mov ds, ax
mov es, ax
mov bp, message3          ; BP=当前串的偏移地址
mov cx, length3           ; CX = 串长
mov ax, 1300h             ; AH = 13h (功能号)、AL = 00h (无光标)
mov bx, 000fh             ; 页号为 0 (BH = 0) 黑底亮白字 (BL = 0fh)
mov dh, 6                 ; 行号
mov dl, 19                ; 列号
int 10h                   ; BIOS 的 10h 功能: 显示一行字符
pop es
pop ds
popa
iret
```

Int34:

```
pusha
push ds
push es
mov ax, word[mark]
mov ds, ax
mov es, ax
mov bp, message4          ; BP=当前串的偏移地址
```

mov cx, length4	; CX = 串长
mov ax, 1300h	; AH = 13h (功能号)、AL = 00h (无光标)
mov bx, 000fh	; 页号为 0 (BH = 0) 黑底亮白字 (BL = 0fh)
mov dh, 6	; 行号
mov dl, 55	; 列号
int 10h	; BIOS 的 10h 功能: 显示一行字符
pop es	
pop ds	
popa	
iret	

Int35:

pusha	
push ds	
push es	
mov ax, word[mark]	
mov ds, ax	
mov es, ax	
mov bp, message5	; BP=当前串的偏移地址
mov cx, length5	; CX = 串长
mov ax, 1300h	; AH = 13h (功能号)、AL = 00h (无光标)
mov bx, 000fh	; 页号为 0 (BH = 0) 黑底亮白字 (BL = 0fh)
mov dh, 18	; 行号
mov dl, 19	; 列号
int 10h	; BIOS 的 10h 功能: 显示一行字符
pop es	
pop ds	
popa	
iret	

Int36:

```
pusha
push ds
push es
mov ax, word[mark]
mov ds, ax
mov es, ax
mov bp, message6          ; BP=当前串的偏移地址
mov cx, length6           ; CX = 串长
mov ax, 1300h             ; AH = 13h (功能号)、AL = 00h (无光标)
mov bx, 000fh             ; 页号为 0 (BH = 0) 黑底亮白字 (BL = 0fh)
mov dh, 18                ; 行号
mov dl, 55                ; 列号
int 10h                   ; BIOS 的 10h 功能: 显示一行字符
pop es
pop ds
popa
iret
```

```
bcd_to_ascii:              ;BCD 码转 ASCII
                            ;输入: AL=bcd 码
                            ;输出: AX=ascii
mov ah, al                 ;分拆成两个数字
and al, 0x0f               ;仅保留低 4 位
add al, 0x30               ;转换成 ASCII

shr ah, 4                  ;逻辑右移 4 位
and ah, 0x0f
```

```

    add ah, 0x30
    ret

message1 db 0,0,'/',0,0,'/',0,0,' ',0,0,':',0,0,':',0,0
length1 equ ($-message1)
message2 db 'OUCH! OUCH!'
length2 equ ($-message2)
message3 db 1,' Int 33',1
length3 equ ($-message3)
message4 db 1,' Int 34',1
length4 equ ($-message4)
message5 db 1,' Int 35',1
length5 equ ($-message5)
message6 db 1,' Int 36',1
length6 equ ($-message6)

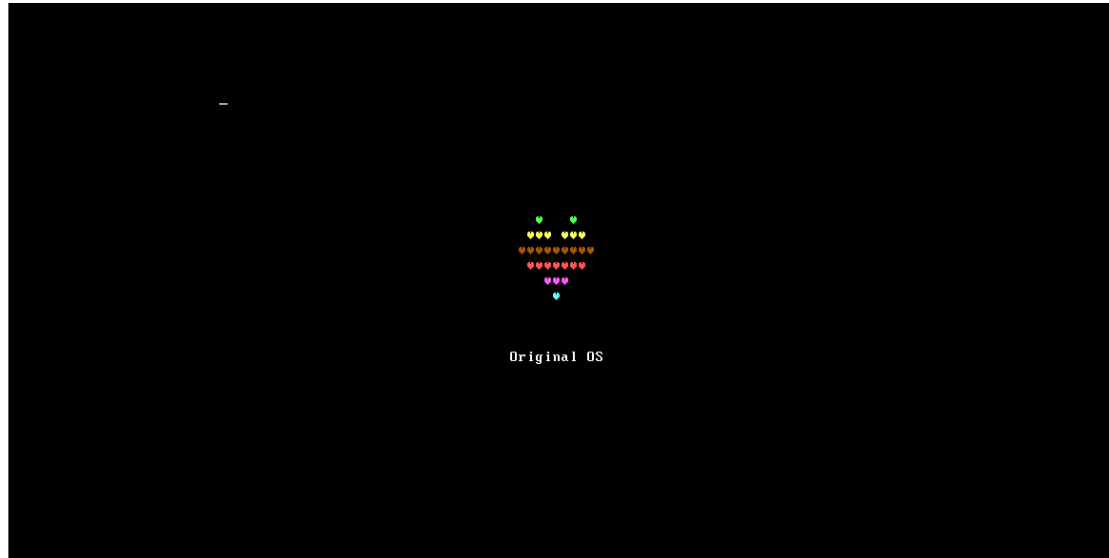
```

在一开始标记的红色的代码，是中断的关键部分，是将中断的入口地址赋值，将中断的 08h, 09h 以及 33h、34h、35h、36h 的入口地址放入中断向量表。首先，当调用 int 08h 时钟中断的时候会在内核地址的右下角不断显示时间，因为我感觉老师要求的循环显示\、|和/太简单了，没有什么新意，因为作为一个操作系统必须要有一个系统时间供用户来查看，我就自己多写了一些代码，在界面的右下角显示时间，因为时钟中断是一秒钟刷新 18 次，那么时间可以动态显示而不需要自己去处理，一举两得。然后是修改键盘中断，也就是 09h，因为原先的键盘中断是不可以丢失的，那么我就把原先的 int 09h 中断的入口地址存放在了 37h 的地方，这样就不会丢失原先的键盘中断，那么我把原先的 09h 中断的入口地址放入了自己修改的代码的地方。当要调用键盘中断的时候，先进入 int 09h 存放的地址空间，也就是我自己写的代码空间，但是怎么检测是否按下键盘呢，这时候刚才存储起来的原先的键盘中断就显示出他的强大功能，先开一下中断允许，也就是 sti，之后调用 int 37h，这时候就和原先的键盘中

断完全一样了，下面的就是我自己附加的循环改变颜色显示 OUCH!OUCH! 的功能。

下面是我实验的截图：

开机界面：



内核界面：



Help 指令：

These are the available commands:

```
_help      show you all the commands
-restart   restart the OS
-shutdown  shutdown the OS
-time      show you the real time
-run proc1 run the prgram1
-run proc2 run the prgram2
-run proc3 run the prgram3
-int       run the interrupt program
```

You can press the Enter key to return.

OUCH! OUCH!
15/04/07 00:50:48

Time 指令:

15/04/07 00:50:26

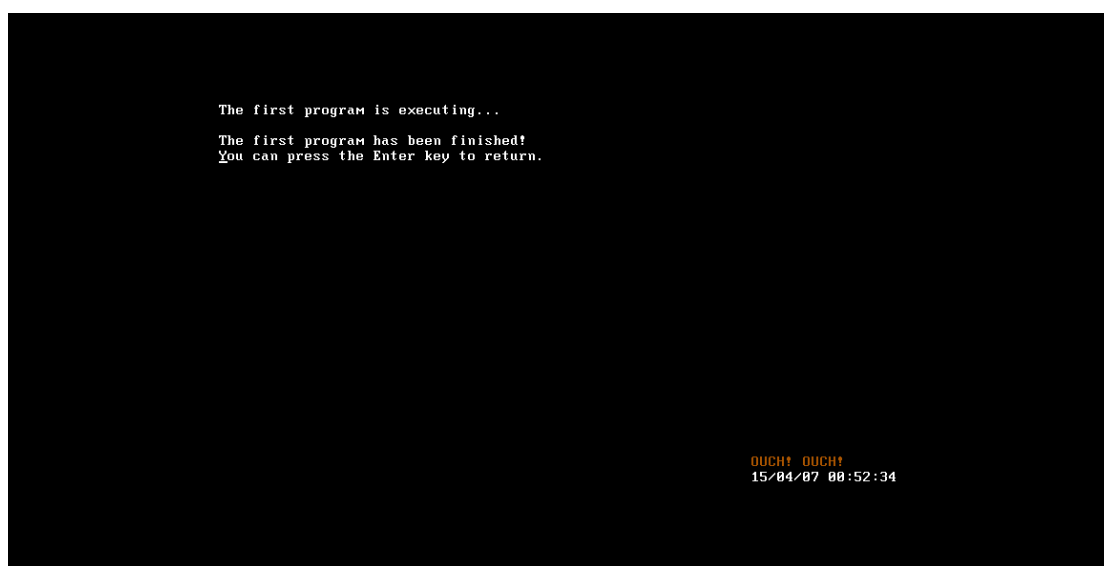
You can press any key to return.

OUCH! OUCH!
15/04/07 00:50:26

Shutdown 指令:



Run proc1 指令:



Run proc2 指令:

```
The second program is executing...
The second program has been finished!
You can press the Enter key to return.

OUCH! OUCH!
15/04/07 00:53:01
```

Run proc3 指令:

```
The third program is executing...
The third program has been finished!
You can press the Enter key to return.

OUCH! OUCH!
15/04/07 00:53:36
```

实验五:

这次试验主要是自己写一些 c 库函数以及对 int 21h 进行多功能的扩充, 增加一些功能号, 实现一些中断 21h 的功能。其中 c 库函数按照老师的要去也必须经过 int 21h 来调用。Int 21h 中主要实现了 c 中的 getchar (char *ch)、putchar (char *ch)、gets (char *str)、puts (char*ch) 以及还增加了在屏幕中央输出 “ouch”、将字符串输出到任意位置、将字符串小写转大写、将字符串大写转小写等功能。

这次实验其实并没有什么可以说明的地方, 无非就是多写几个库函数用来调

用,这次实验的难点不在于 int 21h 的写法,而是在意 scanf 和 printf 的写法,对于这两个函数都是需要不定参数的传递,其中需要 stdarg.h 这个库里面的宏定义,但是这个库里面又包含了其他库里面的东西,所以如果利用原本 c 库里面的东西的话会使得最终所有苦衷的东西全部包含进来,所以最终要想写一个不定参数的函数基本是不可能的,最终我也只是对这两个函数简单的实现了一下,不可能完全实现。自己查了好多的资料,也了解了一下不定参数函数的写法,了解了其中的原理,可能自己在有时间的时候会想着写一下吧。这次试验总体不难,和关键在于 scanf 和 printf 的写法。虽然没有完美实现但是也知道了其中的难度,了解了其中的机理。

按照老师的要求,这几个函数需要自己写在 stdio.h 文件当中,c 文件可以自由调用,而在 stdio.h 库当中函数的实现必须和汇编来结合,具体的实现代码要在 int 21h 中调用,那么也就是要实现三层调用。实现代码展示如下:

Stdio.h 库:

```
extern getch();
extern putch();
extern getstr();
extern putstr();

void getchar(char *ch)
{
    getch(ch);
}

void putchar(char *ch)
{
    putch(ch);
}
```

```
void gets(char *str)
{
    getstr(str);
}
```

```
void puts(char *str)
{
    putstr(str);
}
```

汇编中的 int 21h 关键性代码:

Int21:

```
push ax
push ds
push es
mov bp, sp
mov si, [bp+14]
cmp ah, 0
jz getch
cmp ah, 1
jz putch
cmp ah, 2
jz getstr
cmp ah, 3
jz putstr
cmp ah, 4
jz ouch_
cmp ah, 5
jz lowtoupp_
```



```
cmp ah, 6
jz upptolow_
cmp ah, 7
jz showanyway_
```

```
ouch_: jmp ouch
lowtoupp_: jmp lowtoupp
upptolow_: jmp upptolow
showanyway_: jmp showanyway
```

getch:

```
;mov si, [sp]
mov ah, 0
int 16h
mov [si], al
mov ah, 0eh
int 10h
```

```
mov al, 20h          ; AL = EOI
out 20h, al          ; 发送 EOI 到主 8529A
out 0A0h, al         ; 发送 EOI 到从 8529A
pop es
pop ds
pop ax
iret
```

putch:

```
;mov si, offset _ch
mov ah, 0eh
```

```
mov al,[si]
```

```
int 10h
```

```
mov al,20h ; AL = EOI
```

```
out 20h,al ; 发送 EOI 到主 8529A
```

```
out 0A0h,al ; 发送 EOI 到从 8529A
```

```
pop es
```

```
pop ds
```

```
pop ax
```

```
iret
```

```
getstr:
```

```
;mov si,offset _string
```

```
mov bx,0
```

```
loop1:
```

```
mov ah,0
```

```
int 16h
```

```
mov [si+bx],al
```

```
mov ah,0eh
```

```
int 10h
```

```
add bx,1
```

```
mov ah,0dh
```

```
cmp ah,al
```

```
jnz loop1
```

```
mov al,20h ; AL = EOI
```

```
out 20h,al ; 发送 EOI 到主 8529A
```

```
out 0A0h,al ; 发送 EOI 到从 8529A
```

```
pop es
```

```

    pop ds
    pop ax
    iret

putstr:
    ;mov si,offset _string
    mov bx,0
loop2:
    mov ah,0eh
    mov al,[si+bx]
    int 10h
    add bx,1
    mov ah,0dh
    cmp ah,al
    jnz loop2

    mov al,20h                ; AL = EOI
    out 20h,al                ; 发送 EOI 到主 8529A
    out 0A0h,al                ; 发送 EOI 到从 8529A
    pop es
    pop ds
    pop ax
    iret

ouch:
    mov ax, cs                ; 置其他段寄存器值与 CS 相同
    mov ds, ax                ; 数据段
    mov es, ax                ; 置 ES=DS
    mov bp, offset _Message14 ; BP=当前串的偏移地址

```

mov cx, 4	; CX = 串长
mov ax, 1300h	; AH = 13h (功能号)、AL = 00h (无光标)
mov bx, 0fh	; 页号为 0 (BH = 0) 黑底亮白字 (BL = 0fh)
mov dh, 12	; 行号
mov dl, 38	; 列号
int 10h	; BIOS 的 10h 功能: 显示一行字符

mov al, 20h	; AL = EOI
out 20h, al	; 发送 EOI 到主 8529A
out 0A0h, al	; 发送 EOI 到从 8529A
pop es	
pop ds	
pop ax	
iret	

lowtoupp:

mov bx, 0

loop3:

mov ah, 0eh

mov al, [si+bx]

cmp al, 97

jb lpp1

sub al, 32

lpp1:

mov [si+bx], al

int 10h

add bx, 1

mov ah, 0dh

cmp ah, al

jnz loop3

```
mov al, 20h          ; AL = EOI
out 20h, al          ; 发送 EOI 到主 8529A
out 0A0h, al         ; 发送 EOI 到从 8529A
pop es
pop ds
pop ax
iret
```

upptolow:

```
    mov bx, 0
loop4:
    mov ah, 0eh
    mov al, [si+bx]
    cmp al, 91
    jae lpp2
    add al, 32
lpp2:
    cmp al, 2dh
    jnz lpp3
    sub al, 32
lpp3:
    mov [si+bx], al
    int 10h
    add bx, 1
    mov ah, 0dh
    cmp ah, al
    jnz loop4
```

```

mov al, 20h                ; AL = EOI
out 20h, al                ; 发送 EOI 到主 8529A
out 0A0h, al               ; 发送 EOI 到从 8529A
pop es
pop ds
pop ax
iret

```

showanyway:

```

mov ah, 02h
mov bh, 0
int 10h                    ; BIOS 的 10h 功能: 显示一行字符

```

```

mov bx, 0

```

loop5:

```

mov ah, 0eh
mov al, [si+bx]
int 10h
add bx, 1
mov ah, 0dh
cmp ah, al
jnz loop5

```

```

mov al, 20h                ; AL = EOI
out 20h, al                ; 发送 EOI 到主 8529A
out 0A0h, al               ; 发送 EOI 到从 8529A
pop es
pop ds

```

```
pop ax
```

```
iret
```

```
public _getch
```

```
_getch proc
```

```
mov ah, 0
```

```
int 21h
```

```
ret
```

```
_getch endp
```

```
public _putch
```

```
_putch proc
```

```
mov ah, 1
```

```
int 21h
```

```
ret
```

```
_putch endp
```

```
public _getstr
```

```
_getstr proc
```

```
mov ah, 2
```

```
int 21h
```

```
ret
```

```
_getstr endp
```

```
public _putstr
```

```
_putstr proc
```

```
mov ah, 3
```

```
int 21h
```

```
ret
```

```
_putstr endp
```

```
_ouch proc
```

```
    mov ah, 4
```

```
    int 21h
```

```
    ret
```

```
_ouch endp
```

```
_lowtoupp proc
```

```
    mov ah, 5
```

```
    int 21h
```

```
    ret
```

```
_lowtoupp endp
```

```
_upptolow proc
```

```
    mov ah, 6
```

```
    int 21h
```

```
    ret
```

```
_upptolow endp
```

```
_showanyway proc
```

```
    mov ah, 7
```

```
    mov dh, 24
```

```
    mov dl, 0
```

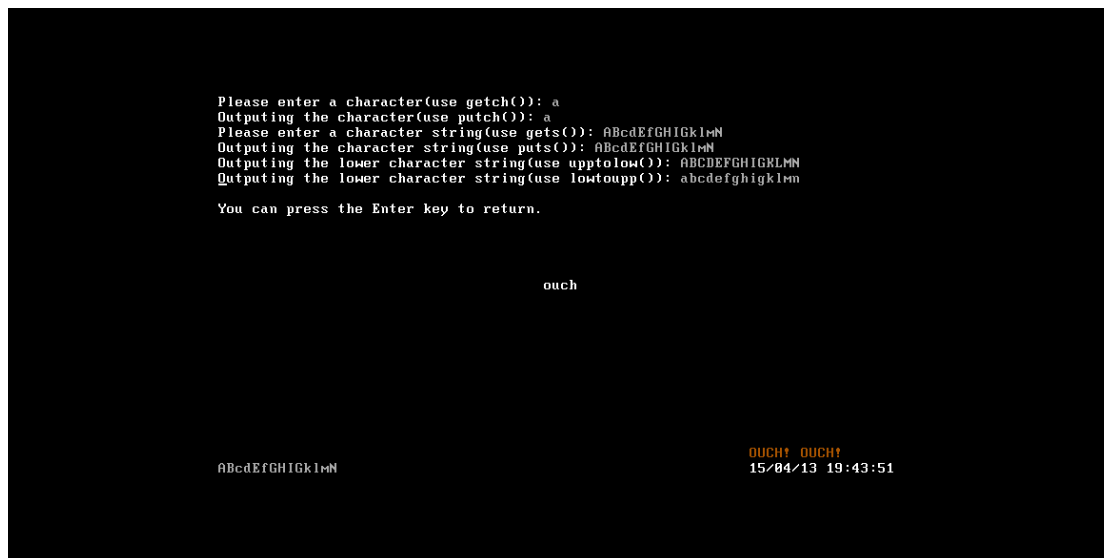
```
    int 21h
```

```
    ret
```

```
_showanyway endp
```

这段代码一开始的对 21h 功能的定义十分的重要,当对 ah 进行赋值的时候,

可以通过对 ah 值的判断去执行不同的功能，例如 ah 是 0 的话，就执行 getchar 功能，实现了汇编中断和 c 语言的 link。具体其他没有什么改动的代码可以查看附件的代码文件。下面是对关键 stdio.h 函数测试输出的截图：



```
Please enter a character(use getch()): a
Outputting the character(use putchar()): a
Please enter a character string(use gets()): ABcdEfGHIGklmN
Outputting the character string(use puts()): ABcdEfGHIGklmN
Outputting the lower character string(use upptolow()): ABCDEFGHIGKLMN
Outputting the lower character string(use lowtoup()): abcdefghigklmn
You can press the Enter key to return.

ouch

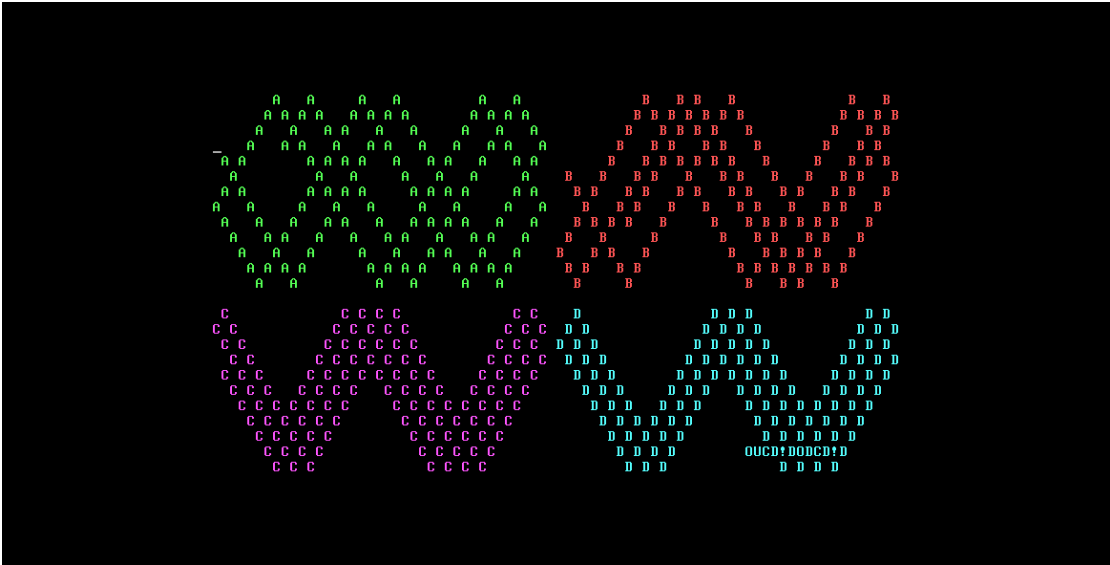
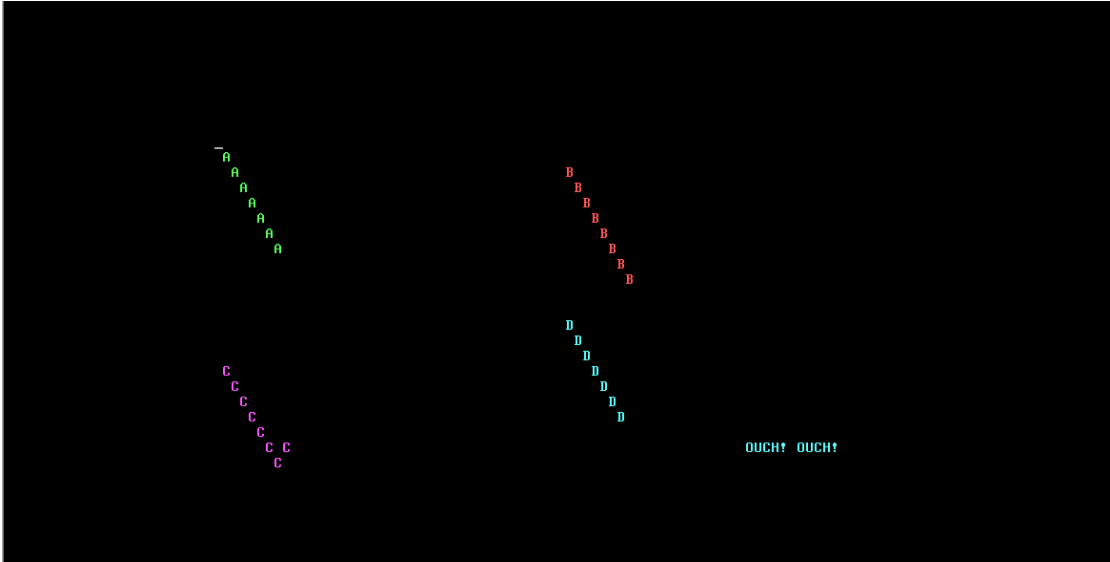
ABcdEfGHIGklmN                                OUCH! OUCH!
                                                15/04/13 19:43:51
```

实验六：

这次实验主要是自己写出多线程的执行，在屏幕的四分之一处分别显示四个不同的程序，而且需要时钟中断来控制线程的执行，每当时钟中断一定次数之后就切换进程，这样不断循环。实验中我选取的四个程序是第一次给出的 stone 程序的改版，这样比较明显的可以看出每个程序的运行。我完全符合老师的要求把四个程序分别加载在第 2、3、4、5 个 64k 的内存地址，也就是段地址为 1000h、2000h、3000h、4000h。这四个段完全独立的运用自己的栈地址，很好的解决了老师说到的共用栈的问题。因为时钟是 1s 要中断 18.3 次，所以我先计数 18 次之后就切换进程来实现多进程。而且我的多进程是完全独立于原先的内核，也就是说这个多进程的展示我用一个函数程序封装了起来，对原先的操作系统不产生任何的影响，我可以执行一个指令来执行多线程的展示，然后可以按任意键来返回原来的操作系统，对原来的操作系统不产生任何影响。这里的时钟中断就需要分支，一个是原先我的实时显示时间，另一个就是多进程需要的中断响应切换进程，这里就需要一个 flag 来判断到底要执行哪一套时钟中断效果。执行多线程

的时候可以开 flag，当返回操作系统的时候需要关 flag。只要执行 thread（线程）这一个指令就可以进入多线程程序界面。

Thread 指令不断执行过程的动态截图显示(注：因为有键盘中断，多以 OUCH！OUCH！虽然我在执行程序的时候清屏了两次，但是在跳转的时候还是会有显示)：





这次实验的关键代码主要是对进程数据结构的定义，还有对 Timer 的改写，我是在参考了老师给出的示例代码，自己大幅度修改写出适合我自己 OS 的代码，那么下面给出这些代码。

内核.c 中关于进程数据结构关键性代码：

```
typedef struct RegisterImage{  
    int SS;  
  
    int GS;  
  
    int FS;  
  
    int ES;  
  
    int DS;  
  
    int DI;  
  
    int SI;  
  
    int BP;  
  
    int SP;  
  
    int BX;  
  
    int DX;  
  
    int CX;
```

```

    int AX;

    int IP;

    int CS;

    int FLAGS;
}RegisterImage;

typedef struct PCB{

    RegisterImage regImg;

    int Process_Status;
}PCB;

PCB pcb_list[4];

int pcbStartFlag=0;

int saveNumber=0;

int runNumber=0;

int NEW=0;

int RUNNING=1;

void Save_Process(int gs, int fs, int es, int ds, int di, int si, int bp, int
sp,

                int dx, int cx, int bx, int ax, int ss, int ip, int cs, int flags)
{

    pcb_list[saveNumber].regImg.AX = ax;

    pcb_list[saveNumber].regImg.BX = bx;

    pcb_list[saveNumber].regImg.CX = cx;

    pcb_list[saveNumber].regImg.DX = dx;

    pcb_list[saveNumber].regImg.DS = ds;

    pcb_list[saveNumber].regImg.ES = es;

```

```

pcb_list[saveNumber].regImg.FS = fs;
pcb_list[saveNumber].regImg.GS = gs;
pcb_list[saveNumber].regImg.SS = ss;

pcb_list[saveNumber].regImg.IP = ip;
pcb_list[saveNumber].regImg.CS = cs;
pcb_list[saveNumber].regImg.FLAGS = flags;

pcb_list[saveNumber].regImg.DI = di;
pcb_list[saveNumber].regImg.SI = si;
pcb_list[saveNumber].regImg.SP = sp;
pcb_list[saveNumber].regImg.BP = bp;
saveNumber++;
if(saveNumber==4)
    saveNumber=0;
}

```

```

void Run_Number()
{
    runNumber++;
    if(runNumber==4)
        runNumber=0;
}

```

```

void init(PCB* pcb, int segment, int offset)
{
    pcb->regImg.GS = 0xb800;
    pcb->regImg.SS = segment;
    pcb->regImg.ES = segment;
}

```

```

    pcb->regImg.DS = segment;
    pcb->regImg.CS = segment;
    pcb->regImg.FS = segment;
    pcb->regImg.IP = offset;
    pcb->regImg.SP = offset - 4;
    pcb->regImg.AX = 0;
    pcb->regImg.BX = 0;
    pcb->regImg.CX = 0;
    pcb->regImg.DX = 0;
    pcb->regImg.DI = 0;
    pcb->regImg.SI = 0;
    pcb->regImg.BP = 0;
    pcb->regImg.FLAGS = 512;
    pcb->Process_Status = NEW;
}

```

```

void init_PCB()
{
    init(&pcb_list[0], 0x1000, 0x100);
    init(&pcb_list[1], 0x2000, 0x100);
    init(&pcb_list[2], 0x3000, 0x100);
    init(&pcb_list[3], 0x4000, 0x100);
}

```

```

PCB* Current_Process()
{
    return &pcb_list[runNumber];
}

```

```

void special()
{
    if(pcb_list[runNumber].Process_Status==NEW)
        pcb_list[runNumber].Process_Status=RUNNING;
}

```

时钟中断 Timer 的关键性代码：

Timer:

```

    cmp word ptr[_pcbStartFlag], 0
    jz  regular_timer
    jmp another_timer

```

regular_timer:

```

    push ax
    push bx
    push cx
    push dx
    push sp
    push bp
    push si
    push di
    push ds
    push es
    mov ah, 04h
    int 1Ah
    mov al, cl
    call bcd_to_ascii
    mov si, offset message15
    mov [si], ah

```

```
mov [si+1],al
mov al,'/'
mov [si+2],al
mov al,dh
call bcd_to_ascii
mov [si+3],ah
mov [si+4],al
mov al,'/'
mov [si+5],al
mov al,dl
call bcd_to_ascii
mov [si+6],ah
mov [si+7],al

mov al,' '
mov [si+8],al

mov ah,02h
int 1Ah
mov al,ch
call bcd_to_ascii
mov [si+9],ah
mov [si+10],al
mov al,':'
mov [si+11],al
mov al,cl
call bcd_to_ascii
mov [si+12],ah
mov [si+13],al
```



```
mov al, ':'  
mov [si+14], al  
mov al, dh  
call bcd_to_ascii  
mov [si+15], ah  
mov [si+16], al
```

```
mov ax, ds                ; 数据段  
mov es, ax                ; 置 ES=DS  
mov bp, offset message15 ; BP=当前串的偏移地址  
mov cx, length15          ; CX = 串长  
mov ax, 1300h             ; AH = 13h (功能号)、AL = 00h (无光标)  
mov bx, 000fh             ; 页号为 0 (BH = 0) 黑底亮白字 (BL = 0fh)  
mov dh, 24                ; 行号  
mov dl, 62                ; 列号  
int 10h                   ; BIOS 的 10h 功能: 显示一行字符  
mov al, 20h               ; AL = EOI  
out 20h, al               ; 发送 EOI 到主 8529A  
out 0A0h, al              ; 发送 EOI 到从 8529A  
pop es  
pop ds  
pop di  
pop si  
pop bp  
pop sp  
pop dx  
pop cx  
pop bx  
pop ax
```

```

    iret

timercount db 0

another_timer:
    mov bx, offset timercount
    mov al, [bx]
    add al, 1
    cmp al, 18
    jnz next_timer
    mov al, 0
    mov [bx], al
    jmp Pro_Timer

next_timer:
    mov [bx], al
    mov al, 20h                ; AL = EOI
    out 20h, al                ; 发送 EOI 到主 8529A
    out 0A0h, al               ; 发送 EOI 到从 8529A
    iret

Pro_Timer:
;*****
;*                Save                *
;*****

    push ss
    push ax
    push bx
    push cx
    push dx
    push sp

```

push bp

push si

push di

push ds

push es

. 386

push fs

push gs

. 8086

mov ax, cs

mov ds, ax

mov es, ax

call near ptr _Save_Process

. 386

pop gs

pop fs

pop es

pop ds

pop di

pop si

pop bp

pop sp

pop dx

pop cx

pop bx

pop ax

pop ss

Pre:

```
mov ax, cs
```

```
mov ds, ax
```

```
mov es, ax
```

```
call near ptr _Run_Number
```

```
call near ptr _Current_Process
```

```
mov bp, ax
```

```
;*****
```

```
;*                Restart                *
```

```
;*****
```

Restart_:

```
push word ptr ds:[bp+30]
```

```
push word ptr ds:[bp+28]
```

```
push word ptr ds:[bp+26]
```

```
push word ptr ds:[bp+2]
```

```
push word ptr ds:[bp+4]
```

```
push word ptr ds:[bp+6]
```

```
push word ptr ds:[bp+8]
```

```
push word ptr ds:[bp+10]
```

```
push word ptr ds:[bp+12]
```

```
push word ptr ds:[bp+14]
```

```
push word ptr ds:[bp+18]
```

```
push word ptr ds:[bp+20]
```

```
push word ptr ds:[bp+22]
```

```
push word ptr ds:[bp+24]
```

```
pop ax
```

```
pop cx
```

```
pop dx
```

```
pop bx
```

```
pop bp
```

```
pop si
```

```
pop di
```

```
pop ds
```

```
pop es
```

```
. 386
```

```
pop fs
```

```
pop gs
```

```
. 8086
```

```
push ax
```

```
mov al, 20h
```

```
out 20h, al
```

```
out 0A0h, al
```

```
pop ax
```

```
iret
```

这次实验关于多线程一开始真的是一头雾水，然后自己想到多线程也就是模拟 cpu 的过程，那么就可以把所有寄存器保存好就可以完成和原 cpu 一样的功能了，所以何老师给出的示例代码以及思路有异曲同工之妙，我只不过是参考了老师给出的代码，但是我并没有有完全照抄老师代码，我做了大幅度的修改，增加了一些其他的功能，完全适合我的操作系统，精简了好多我认为没有用的定义。

实验中也是问题很多，关键是 TASM 实在是资料少的可怜，出了错误并不能迅速找到解决的办法，只能自己一遍一遍的去尝试，而且有一些错误 bug 现在还不清楚是怎么回事，因为实在是太奇怪也无法测试。我上面说过的判断时钟中断的 flag 变量如果在 c 中定义的话会出现明显的错误，它的值我赋值为 1 之后没

有真正的赋值为 1，而且在 cmp 它和 0 时候会出现错误。我在 asm 文件中定义为 DW 之后是可以正常使用的，这个问题现在不知道是因为什么，可以查看我的源代码我将他给注释掉。还有一个遇到的就是爆栈的，老师给出的代码里面要 save 原来的程序，那么调用 save 函数，push 了一些寄存器值，但是调用完函数之后没有 pop 掉，最后会导致爆栈，我出现的现象是多线程子程序会执行一段时间之后停下来死掉，无法回到原来的操作系统。发现这个问题之后加以改正，多线程程序正常运行。

实验七：

这次试验主要实现的就是父子进程之间的协调工作，父进程交给子进程实现某些功能，在这里确实遇到了很多的问题。一开始并不清楚父子程序怎么调用内核的数据机构，然后想到了目前只能中断处理机制可以把他们联系起来。

另一点就是老师 ppt 上面讲到的 return 返回机制，就是 c 中的函数返回的值会存储在汇编的 ax 寄存器当中，那么这就要相当的小心了，因为弄不好的话，ax 的值就会很容易在不知不觉中改变，确保返回值再返回调用的过程当中不会被改变，这对 fork() 的返回值尤其重要，因为要对 pid 产生影响。比如这里：

```
void memcpy( PCB* F_PCB, PCB* C_PCB )
{
    transform();
    C_PCB -> regImg.SP = F_PCB -> regImg.SP;
    C_PCB -> regImg.GS = F_PCB -> regImg.GS;
    C_PCB -> regImg.ES = F_PCB -> regImg.ES;
    C_PCB -> regImg.DS = F_PCB -> regImg.DS;
    C_PCB -> regImg.CS = F_PCB -> regImg.CS;
    C_PCB -> regImg.FS = F_PCB -> regImg.FS;
    C_PCB -> regImg.IP = F_PCB -> regImg.IP;
    C_PCB -> regImg.AX = 0;
    C_PCB -> regImg.BX = F_PCB -> regImg.BX;
    C_PCB -> regImg.CX = F_PCB -> regImg.CX;
    C_PCB -> regImg.DX = F_PCB -> regImg.DX;
    C_PCB -> regImg.DI = F_PCB -> regImg.DI;
    C_PCB -> regImg.SI = F_PCB -> regImg.SI;
    C_PCB -> regImg.BP = F_PCB -> regImg.BP;
    C_PCB -> regImg.FLAGS = F_PCB -> regImg.FLAGS;
}
```

就需要先对子程序的 ax 先赋值为 0，然后返回的时候 pid 就可以是 0 了。并且时刻关注栈段的 push 与 pop 防止出现值的变动。

这里想说的就是关键在于耐心去找到一点点的错误，可能错误微乎其微但是发生确实致命的，比如下面这个：

```
void CountLetter()
{
    int i;
    LetterNumber=0;
    for(i=0;i<80;i++)
    {
        if(str[i]=='\0')
            break;
        else
            LetterNumber++;
    }
}
```

本来 LetterNumber 这个变量在声明的时候就已经赋值为 0 了，但是总是出错，显示出的统计字母数不是真实的数字，最后在这个计数函数里面又声明了一次就可以了，这一点我现在还不是很清楚为什么会这样，可能内部有一些机理我们不知道吧。下面是我的父子进程共享的一段代码：

```
char str[80]="2013_CS_class3_zhangxu_13349154";
char strNumber[5];
int LetterNumber=0;
int pid=0;

void CountLetter();
void printsum();

void fsprocess()
{
    pid=fork();
    if (pid==-1)
```

```

    {
        printf("error in fork!");
        exit(-1);
    }
    else if(pid)
    {
        wait();
        printsum();
        exit(0) ;
    }
    else
    {
        CountLetter();
        exit(0);
    }
}

```

```

void CountLetter()
{
    int i;
    LetterNumber=0;
    for(i=0;i<80;i++)
    {
        if(str[i]=='\0')
            break;
        else
            LetterNumber++;
    }
}

```



```

void printsum()
{
    int temp=LetterNumber;
    int a=temp/10;
    int b=temp-a*10;
    strNumber[0]=a+48;
    strNumber[1]=b+48;
    strNumber[2]='\0' ;
    printf(&strNumber);
}

```

①这里的 fork()、wait() 还有 exit() 这三个程序都和内核是分离的，无法调用内核声明的 PCB 数据结构，也和内核没有什么联系，那么要怎么办才可以做到与内核联系呢？在这里，我想到了利用贯穿始终的中断，将这三个还是封装成 int 21h 中断，然后就可以在任何地方调用了，见下面的代码：

```

public _fork
_fork proc
    mov ah,8
    int 21h
    ret
_fork endp

```

```

public _wait
_wait proc
    mov ah,9
    int 21h
    ret
_wait endp

```

```

public _exit
_exit proc
    mov ah, 10
    int 21h
    ret
_exit endp

```

在内核中，当 int 21h 调用这三个功能的时候，就需要在 c 中写出来，那么最终这三个程序由 do_fork()、do_wait、do_exit() 实现，这三个函数如下：

```

int do_fork()
{
    int i;
    for(i=1;i<8;i++)
    {
        if(pcb_list[i].Process_Status == READY)
        {
            pcb_list[i].Id = i;
            memcpy(&pcb_list[0], &pcb_list[i]);
            return i;
        }
    }
    return -1;
}

```

```

void do_wait()
{
    pcb_list[0].Process_Status = BLOCKED;
}

```

```

}

void do_exit()
{
    if(runNumber==0)
        pcb_list[0].Process_Status = BLOCKED;
    else
    {
        pcb_list[0].Process_Status = READY;
        pcb_list[1].Process_Status = BLOCKED;
    }
}

```

②在父进程当中当调用生成子进程实现轮转的时候需要将父进程的各个存储器的内容复制给子进程的 PCB，那么下面就是 copy 函数：

```

void memcpy( PCB* F_PCB, PCB* C_PCB )
{
    transform();
    C_PCB -> regImg.SP = F_PCB -> regImg.SP;
    C_PCB -> regImg.GS = F_PCB -> regImg.GS;
    C_PCB -> regImg.ES = F_PCB -> regImg.ES;
    C_PCB -> regImg.DS = F_PCB -> regImg.DS;
    C_PCB -> regImg.CS = F_PCB -> regImg.CS;
    C_PCB -> regImg.FS = F_PCB -> regImg.FS;
    C_PCB -> regImg.IP = F_PCB -> regImg.IP;
    C_PCB -> regImg.AX = 0;
    C_PCB -> regImg.BX = F_PCB -> regImg.BX;
    C_PCB -> regImg.CX = F_PCB -> regImg.CX;
    C_PCB -> regImg.DX = F_PCB -> regImg.DX;
}

```

```

C_PCB -> regImg.DI = F_PCB -> regImg.DI;
C_PCB -> regImg.SI = F_PCB -> regImg.SI;
C_PCB -> regImg.BP = F_PCB -> regImg.BP;
C_PCB -> regImg.FLAGS = F_PCB -> regImg.FLAGS;
}

```

这里面需要注意的是父子进程的栈段是不同的，不能将父进程的栈段和子进程共享，那就就需要在不同的栈段将父进程的内容复制过来，这里的 transform() 函数就是实现的这个功能，下面的它的关键代码：

```

public _transform
_transform proc
    push es
    push ds
    push di
    push si
    mov ax, 1000h
    mov ds, ax
    mov ax, 2000h
    mov es, ax
    mov si, 0h
    mov di, 0h
    mov cx, 100h
    lodsb
    stosb
    pop si
    pop di
    pop ds
    pop es

```

`_transform endp`

里面的 `lodsb` 和 `stosb` 配合使用可以将内存中的一块内容取出来然后在写到另一块内容，十分简洁实用。

③为了防止在时钟中断的时候，没有初始轮转的 PCB，那么在执行 `fork()` 时就必须手动保存父进程的状态，防止子进程在复制父进程的状态是并不是执行 `fork()` 之后的状态，下面是保存的代码：

`fork:`

```
    push ss
    push ax
    push bx
    push cx
    push dx
    push sp
    push bp
    push si
    push di
    push ds
    push es
    . 386
    push fs
    push gs
    . 8086

    mov ax, cs
    mov ds, ax
    mov es, ax
    call near ptr _Save_Process
```

. 386

pop gs

pop fs

pop es

pop ds

pop di

pop si

pop bp

pop sp

pop dx

pop cx

pop bx

pop ax

pop ss

call _do_fork

mov al, 20h ; AL = EOI

out 20h, al ; 发送 EOI 到主 8529A

out 0A0h, al ; 发送 EOI 到从 8529A

pop es

pop ds

iret

④对于时钟轮转有多定义了几个状态：

int READY = 1;

int RUNNING = 2;

int BLOCKED = 3;

int END = 4;

这些是主要的修改细节以及关键性代码，其余的相对不是很重要的代码就不展示了

操作系统实验感想：

这一学期，自己学习的操作系统，虽然自己做的并不是很好，但是全部都是我一个人一点一点 code 来的，按照老师的要求以及指导，自己私下不断琢磨，克服了很多困难，最终走到了这一步，我十分的开心，因为我自己做出了属于自己的原型操作系统！

自己学会了操作系统的基本原理，我相信以后就算碰到再大的困难都不会退缩了，因为这个实验我学会了很多：不会就多思考，不懂就多查资料，遇到错误就耐心的排查 debug 以及和其它人多多的交流等等。说实话为了操作系统我把其他的课程都分出时间来搞，每天都在熬夜去搞，我相信我的心血并没有白费，再次感谢老师的教授，也感谢自己的努力。