# 实验四：具有中断处理的内核

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:

- Windows 10-64bit
- Vmware WorkStation 15 pro 15.5.1 build-15018445：虚拟机软件
- NASM version 2.13.02：汇编程序的编译器，在 linux 下通过 sudo apt-get install nasm 下载
- Ubuntu-18.04.4:安装在 Vmware 的虚拟机上
- 代码编辑器：Visual Studio Code 1.44.2
- GNU ld 2.30：链接器

**实验思路：**

实验 4 是在实验 3 的基础上添加一些额外的功能，所以在对于和实验 3 相同部分的程序介绍时的叙述会稍微简略。

首先给出我们在本次实验中需要实现的程序如下:

**（1） 引导程序：**

**bootloader.asm：**

主要完成三项工作：1、显示提示信息；2、加载用户程序的信息表到内存中；3、加载并且跳转到操作系统的内核。

本程序与前面的实验所使用的引导程序无异，故不作介绍了。

**（2） 用户程序信息表：**

**userproginfo.asm：**

在实验 3 中，我们定义了 4 个用户程序在用户程序信息表中，现在我们需要添加多一个调用 int 33-36 的用户程序，同时定义的用户程序数量也从 4→5 。具体修改后的代码如下：

```
UserProgNumber:
    db 5                    ; 用户程序数量

UserProgInfo:
    UserProgInfoBlock 1, 'b', 1024, 0, 1, 1, offset_usrprog1
    UserProgInfoBlock 2, 'a', 1024, 0, 1, 3, offset_usrprog2
    UserProgInfoBlock 3, 'c', 1024, 0, 1, 5, offset_usrprog3
    UserProgInfoBlock 4, 'd', 1024, 0, 1, 7, offset_usrprog4
    UserProgInfoBlock 5, 'interrupt_caller', 512, 0, 1, 9, offset_intca
ller
```

其功能也与实验 3 的功能一致，故不作介绍了。

**（3） 操作系统内核：**

本次实验的内核与实验 3 的有所不同，具体如下：

操作系统内核包含了如下程序：

| 文件名 | 功能 |
| --- | --- |
| osstarter.asm | 监控程序，接收用户命令，执行相应的用户程序 |
| myos.asm | 包含 n 个汇编编写的函数 |
| myos_c.c | 包含 n 个 C 编写的函数 |
| stringio.h | myos_c.c 的头文件，实现了输入输出等功能 |

**osstarter.asm：**

和实验 3 相比，多了两行重要的代码如下：

```
    MOVE_INT_VECTOR 08h, 38h
    WRITE_INT_VECTOR 08h, Timer ; 装填时钟中断向量表
```

其中，Timer 为风火轮中断处理程序，这也是本次实验新增的一个功能。

**myos.asm:**

与上一个版本的 myos.asm 相比，多了一个函数用来处理风火轮的打开与关闭，具体代码如下：

```
switchHotwheel:                 ; 函数：打开或关闭风火轮
    push es
    mov ax, 0
    mov es, ax
    mov ax, [es:08h*4]          ; ax=08h 号中断处理程序的偏移地址
    cmp ax, Timer               ; 检查 08h 号中断处理程序是否是风火轮
    je turnoff                  ; 如果是，则关闭
    WRITE_INT_VECTOR 08h, Timer ; 如果不是，则打开
    mov ax, 1                   ; 返回 1 表示风火轮已打开
    jmp switchDone
    turnoff:
    MOVE_INT_VECTOR 38h, 08h
    mov ax, 0B800h              ; 文本窗口显存起始地址
    mov gs, ax                  ; GS = B800h
    mov ah, 0Fh                 ; 黑色背景
    mov al, ' '                 ; 显示空格
    mov [gs:((80*24+79)*2)], ax ; 更新显存
    mov ax, 0                   ; 返回 0 表示风火轮已关闭
    switchDone:
    pop es
    retf
```

该函数由 myos_c.c 进行调用，myos_c.c 通过调用 switchHotwheel()这个函数，对风火轮进行打开或者关闭操作。

**myos_c.c：**

更新之后最主要的部分就是 switchHotwheel()函数的内容，由于

其具体实现是在汇编里面实现的，所以在 C 里面只需要对其进行简单调用即可，具体如下：

```
if(strcmp(cmd_firstword, commands[hotwheel]) == 0) {
            const char* turned_on = "Hotwheel has been turned on.\r\n";
            const char* turned_off = "Hotwheel has been turned off.\r\n
";
            if(switchHotwheel()==0) {
                print(turned_off);
            }
            else {
                print(turned_on);
            }
        }
```

要执行这个命令，只需要输入 hotwheel 即可实现对风火轮的关闭或者打开操作了。

**stringio.h：**

对字符串的处理的各个函数，在此不做过多叙述，详见源代码。

**（4） 四个实验 3 的用户程序+一个调用 int 33-36 的用户程序：**

**a.asm、b.asm、c.asm、d.asm：**

这四个用户程序和实验 3 的用户程序相比，多了两行重要的代码：

```
    MOVE_INT_VECTOR 09h, 39h
    WRITE_INT_VECTOR 09h, IntOuch
```

这两行代码的作用就是转移中断向量，并写入中断向量表中。为后面编写中断处理程序 intouch.asm 做准备。其中，MOVE_INT_VECTOR 和 WRITE_INT_VECTOR 为头文件 macro.asm 定义的宏，定义如下：

```
%macro WRITE_INT_VECTOR 2       ; 写中断向量表；参数：（中断号，中断处理程
序地址）
    push ax
    push es
    mov ax, 0
```

```
    mov es, ax                  ; ES = 0
    mov word[es:%1*4], %2       ; 设置中断向量的偏移地址
    mov ax,cs
    mov word[es:%1*4+2], ax     ; 设置中断向量的段地址=CS
    pop es
    pop ax
%endmacro
%macro MOVE_INT_VECTOR 2        ; 将参数 1 的中断向量转移至参数 2 处
    push ax
    push es
    push si
    mov ax, 0
    mov es, ax
    mov si, [es:%1*4]
    mov [es:%2*4], si
    mov si, [es:%1*4+2]
    mov [es:%2*4+2], si
    pop si
    pop es
    pop ax
%endmacro
```

具体的思想是：中断响应后，先到内存找到中断向量表，然后跳转至已经保存了寄存器的中断程序。中断程序执行完之后，还原寄存器，最后调用中断返回指令。

**e.asm：**

这个程序主要起到调用作用：当用户从键盘按下 3/4/5/6 时，系统相应地执行 int 33/34/35/36 进入中断处理程序。而本次实验的中断处理程序是将显示屏幕从上到下简单均分成 4 个部分，每一种调用会相应地将对应的区域进行颜色填充处理，具体调用代码如下：

```
Keyboard:
    mov ah, 0
    int 16h
    cmp al, '3'         ; 按下 3
    je callInt33        ; 执行 int 33
    cmp al, '4'         ; 按下 4
    je callInt34        ; 执行 int 34
```

```
    cmp al, '5'          ; 按下 5
    je callInt35         ; 执行 int 35
    cmp al, '6'          ; 按下 6
    je callInt36         ; 执行 int 36
    cmp al, 27           ; 按下 ESC
    je QuitUsrProg       ; 直接退出
    jmp Keyboard         ; 无效按键，重新等待用户按键
callInt33:
    mov word[start_row], 0
    mov word[end_row], 5
    int 33
    jmp QuitUsrProg
callInt34:
    mov word[start_row], 6
    mov word[end_row], 11
    int 34
    jmp QuitUsrProg
callInt35:
    mov word[start_row], 12
    mov word[end_row], 17
    int 35
    jmp QuitUsrProg
callInt36:
    mov word[start_row], 18
    mov word[end_row], 24
    int 36
    jmp QuitUsrProg
```

**int33~36.asm：**

实现对应区域的颜色填充处理的代码，由 e.asm 提供的 start_row 和 end_row 两个参数决定区域的所在范围。部分关键代码如下：

```
Int33~36:
    push ax
    push si
    push ds
    push gs

    mov ax,cs
    mov ds,ax                ; DS = CS
    mov ax,0B800h            ; 文本窗口显存起始地址
    mov gs,ax                ; GS = B800h
    mov ax, [start_row]      ; ax=start_row
```

```
    mov ah, 2*80                ; ah=2*80
    mul ah                      ; ax=start_row * 2 * 80
    mov si, ax                  ; si 初始化为起始位置指针
disploop:
    mov al, [temp_char]         ; 要显示的字符
    mov [gs:si], al             ; 显示字符
    inc si                      ; 递增指针
    mov ah, [temp_color]        ; 字符颜色属性
    mov [gs:si], ah             ; 显示颜色属性
    inc si                      ; 递增指针
    add byte[temp_color], 11h ; 改变颜色
    call Delay                  ; 延时

    mov ah, 01h                 ; 功能号：查询键盘缓冲区但不等待
    int 16h
    jz NoEsc                    ; 无键盘按下，继续
    mov ah, 0                   ; 功能号：查询键盘输入
    int 16h
    cmp al, 27                  ; 是否按下 ESC
    je QuitInt                  ; 若按下 ESC，退出用户程序
NoEsc:
    mov ax, [end_row]           ; al=end_row，ah 实际上无用
    mov ah, 2*80
    mul ah                      ; ax=end_row * 2 * 80
    add ax, [start_row]         ; ax=start_row + end_row*2*80
    cmp si, ax
    jne disploop                ; 范围全部显示完，中断返回
```

**hotwheel.asm：**

该程序要在第 24 行、第 79 列的位置实现风火轮的效果显示，采用的方法是：通过一个寄存器 si 实现无限循环的操作，si 从 0 递增至 4 之后重新置零，当 si 处于 0-3 中的任何一个数时，显示风火轮的四分之一的状态（即-/|\其中一个状态）。部分关键代码如下：

初始状态：

```
DataArea:
    delay equ 3                 ; 计时器延迟计数
    count db delay              ; 计时器计数变量，初值=delay
    hotwheel db '-\|/'          ; 风火轮字符
    wheel_offset dw 0           ; 风火轮字符偏移量，初值=0
```

循环显示：

```
    dec byte [count]              ; 递减计数变量
    jnz EndInt                    ; >0：跳转
    mov byte[count],delay         ; 重置计数变量=初值 delay
    mov si, hotwheel              ; 风火轮首字符地址
    add si, [wheel_offset]        ; 风火轮字符偏移量
    mov al, [si]                  ; al=要显示的字符
    mov ah, 0Ch                   ; ah=黑底，淡红色
    mov [gs:((80*24+79)*2)], ax ; 更新显存
    inc byte[wheel_offset]        ; 递增偏移量
    cmp byte[wheel_offset], 4     ; 检查偏移量是否超过 3
    jne EndInt                    ; 没有超过，中断返回
    mov byte[wheel_offset], 0     ; 超过 3 了，重置为 0
```

**intouch.asm：**

该程序是用以实现“OUCH! OUCH!”在屏幕的某个位置显示的功能。根据要求，当用户在执行用户程序中按下任意一个按键时，该字符串在合适的位置显示。当用户按下键盘的按键时，系统引发 09h 号中断，在 BIOS 提供的中断处理程序中，首先从 60h 端口中读出按键，然后将其存入缓冲区，并由 int 16h 读出并清除。关键的处理就是将用户按下的字符进行清除，避免字符在端口处堵塞或者残留在缓冲区中。部分关键代码如下：

```
    mov ax, cs              ; 置其他段寄存器值与 CS 相同
    mov ds, ax              ; 数据段
    mov bp, ouch_msg        ; 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, ouch_msg_len    ; CX = 串长
    mov ax, 1300h           ; AH = 13h（功能号）、AL = 01h（光标不动）
    mov bx, 0007h           ; 页号为 0(BH = 0) 黑底白字(BL = 07h)
    mov dh, 20              ; 行号=0
    mov dl, 40              ; 列号=0
    int 10h                 ; BIOS 的 10h 功能：显示一行字符

    call Delay
```

```
    mov ax, cs           ; 置其他段寄存器值与 CS 相同
    mov ds, ax           ; 数据段
    mov bp, ouch_clear   ; 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, ouch_msg_len ; CX = 串长
    mov ax, 1300h        ; AH = 13h（功能号）、AL = 01h（光标不动）
    mov bx, 0007h        ; 页号为 0(BH = 0) 黑底白字(BL = 07h)
    mov dh, 20           ; 行号=0
    mov dl, 40           ; 列号=0
    int 10h              ; BIOS 的 10h 功能：显示一行字符
Delay:                   ; 延迟一段时间
    push ax
    push cx
    mov ax, 580
delay_outer:
    mov cx, 50000
delay_inner:
    loop delay_inner
    dec ax
    cmp ax, 0
    jne delay_outer
    pop cx
    pop ax
    ret


    ouch_msg db 'OUCH! OUCH!'
    ouch_msg_len equ $-ouch_msg
    ouch_clear db '           '
```

先显示"OUCH! OUCH!"，然后调用 Delay 延迟一会，接着 clear 清除掉。

**混合编译链接：**

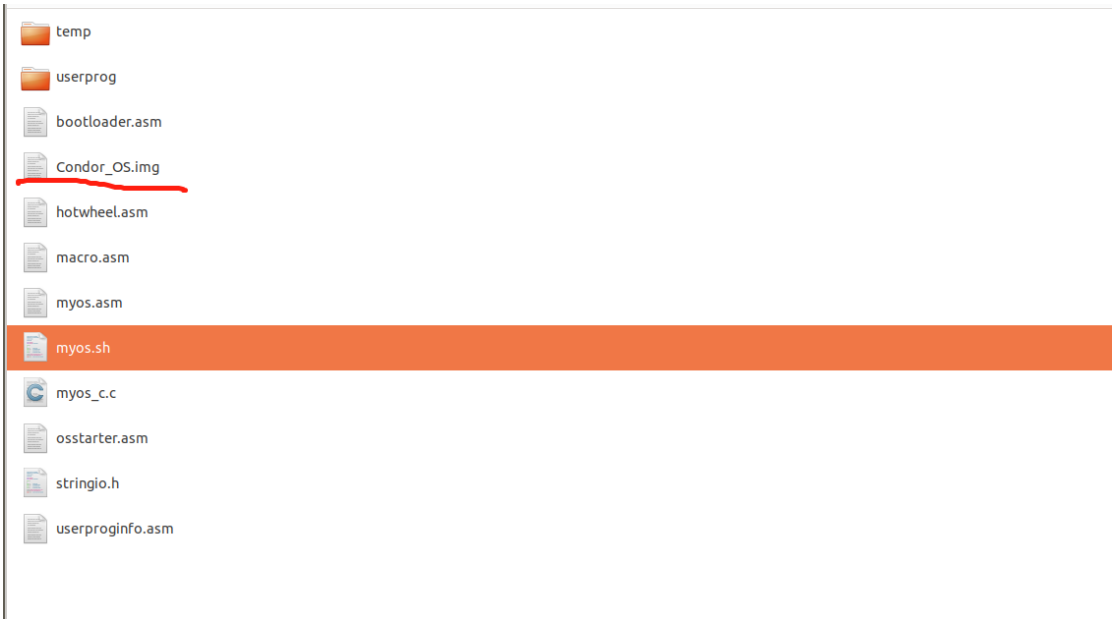
同实验 3 一样，将上述所有的文件联合编译整合，保存为 myos.sh 如下：

```
#!/bin/bash
rm -rf temp
mkdir temp
rm *.img
```

```
nasm bootloader.asm -o ./temp/bootloader.bin
nasm userproginfo.asm -o ./temp/userproginfo.bin

cd userprog
nasm b.asm -o ../temp/b.bin
nasm a.asm -o ../temp/a.bin
nasm c.asm -o ../temp/c.bin
nasm d.asm -o ../temp/d.bin
nasm e.asm -o ../temp/e.bin
cd ..

nasm -f elf32 hotwheel.asm -o ./temp/hotwheel.o

nasm -f elf32 osstarter.asm -o ./temp/osstarter.o
nasm -f elf32 myos.asm -o ./temp/myos.o
gcc -fno-pie -c -m16 -march=i386 -masm=intel -nostdlib -ffreestanding -
mpreferred-stack-boundary=2 -lgcc -shared myos_c.c -o ./temp/myos_c.o
ld -m elf_i386 -N -Ttext 0x8000 --
oformat binary ./temp/osstarter.o ./temp/myos.o ./temp/myos_c.o ./temp/
hotwheel.o -o ./temp/myos_c.bin
rm ./temp/*.o

dd if=./temp/bootloader.bin of=Condor_OS.img bs=512 count=1 2> /dev/nul
l
dd if=./temp/userproginfo.bin of=Condor_OS.img bs=512 seek=1 count=1 2>
 /dev/null
dd if=./temp/myos_c.bin of=Condor_OS.img bs=512 seek=2 count=16 2> /dev
/null
dd if=./temp/b.bin of=Condor_OS.img bs=512 seek=18 count=2 2> /dev/null
dd if=./temp/a.bin of=Condor_OS.img bs=512 seek=20 count=2 2> /dev/null
dd if=./temp/c.bin of=Condor_OS.img bs=512 seek=22 count=2 2> /dev/null
dd if=./temp/d.bin of=Condor_OS.img bs=512 seek=24 count=2 2> /dev/null
dd if=./temp/e.bin of=Condor_OS.img bs=512 seek=26 count=2 2> /dev/null


echo "Done."
```

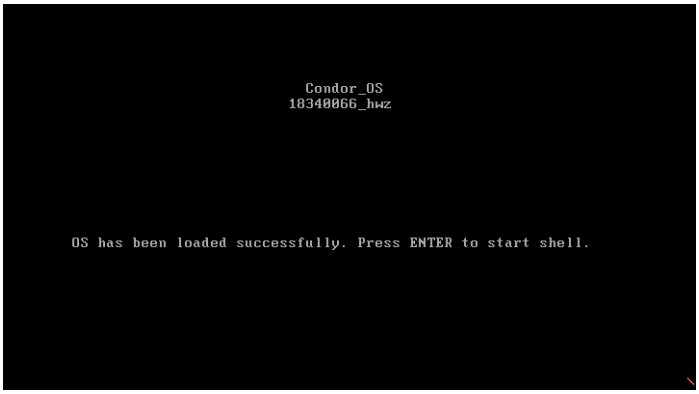
运行之，得到 Condor_OS.img：

```
condor@condor-virtual-machine:~/oslab/实验4/myos$ ./myos.sh
Finished.
condor@condor-virtual-machine:~/oslab/实验4/myos$
```

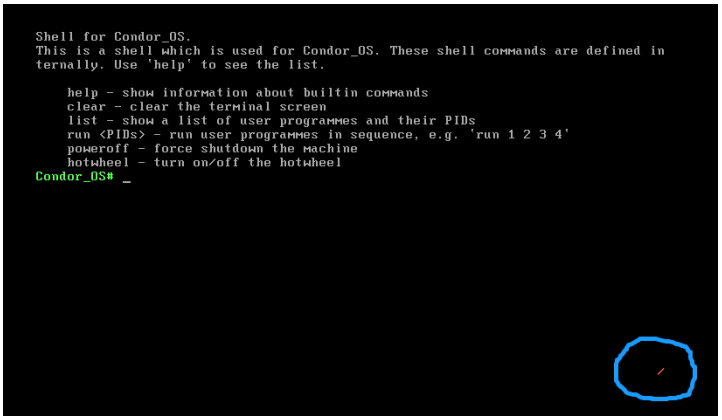



**实验结果：**

开机：



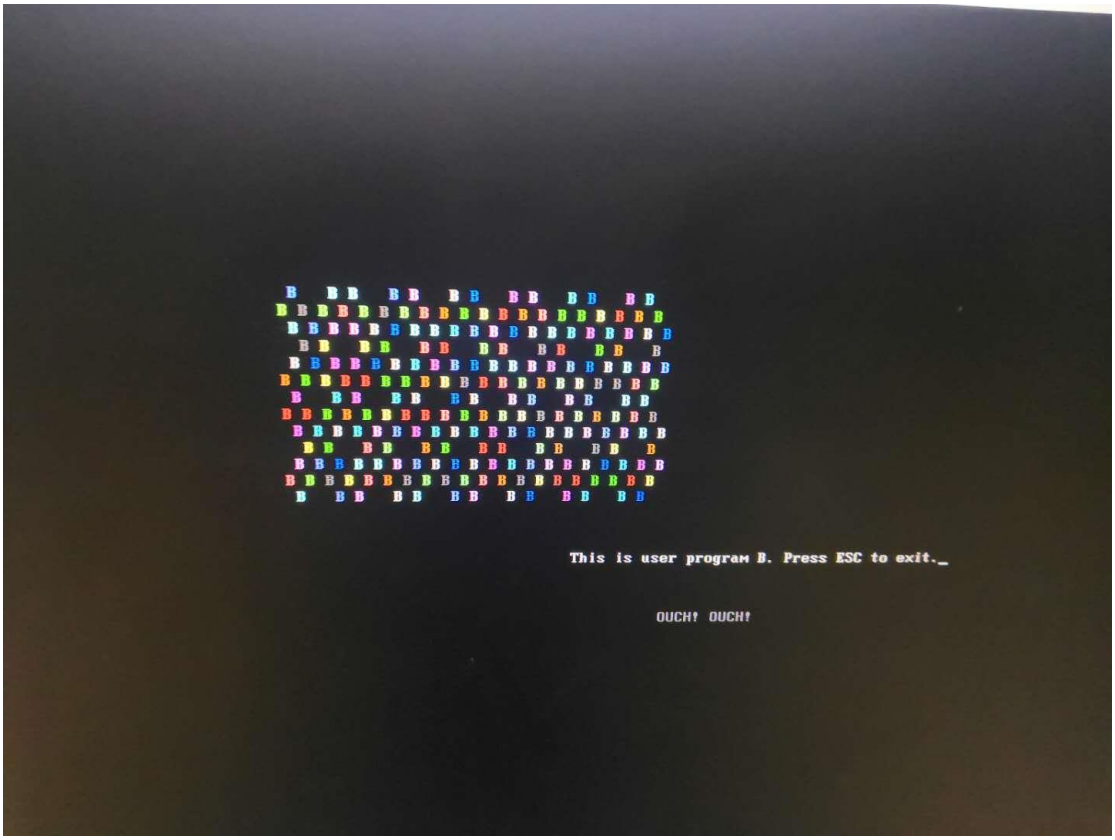

右下角风火轮（上图也有）：



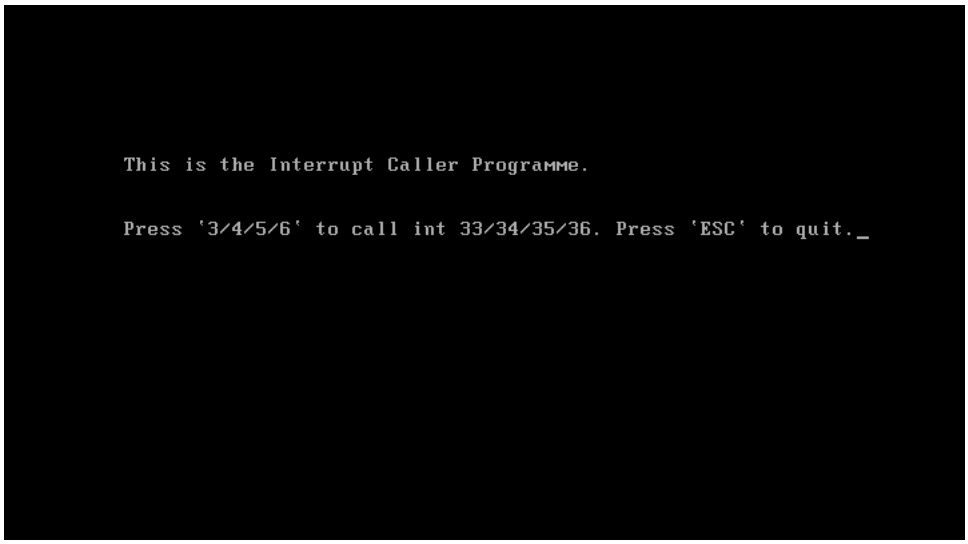

将风火轮关掉：

```
Shell for Condor_OS.
This is a shell which is used for Condor_OS. These shell commands are defined in
ternally. Use 'help' to see the list.

    help - show information about builtin commands
    clear - clear the terminal screen
    list - show a list of user programmes and their PIDs
    run <PIDs> - run user programmes in sequence, e.g. 'run 1 2 3 4'
    poweroff - force shutdown the machine
    hotwheel - turn on/off the hotwheel
Condor_OS# hotwheel
Hotwheel has been turned off.
Condor_OS# _
```

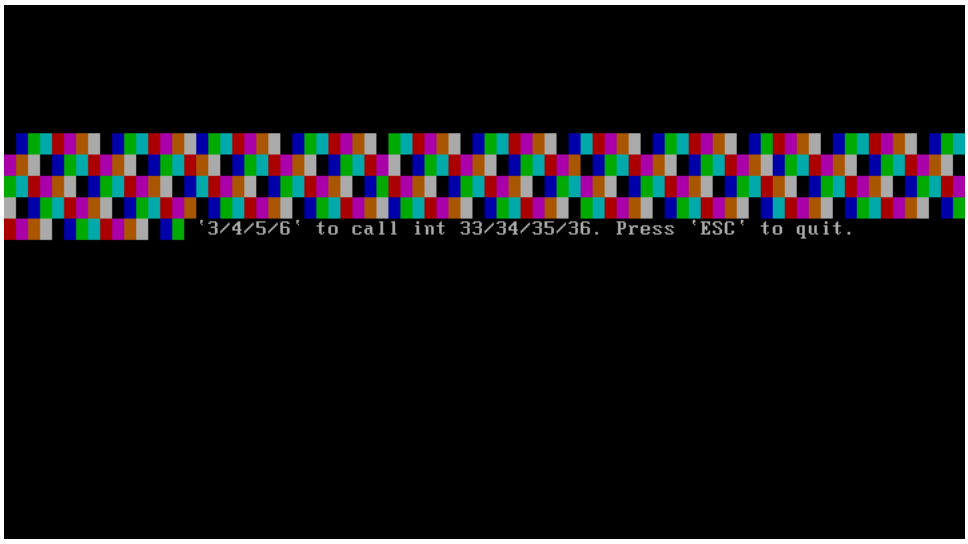
按下任意按键显示“OUCH! OUCH!”:




run 5:




Press 4：




**实验心得：**

这次实验让我更加清晰地理解了中断对于操作系统而言到底是怎么样的一回事，对其的理解也算是有所加深。

但是在本次实验中，在按下键盘来调用中断程序时，如果是长按某个按键，那么操作系统就会在显示“OUCH! OUCH!”的画面卡死，随后按下任何按键都会产生蜂鸣声。个人猜想是缓冲区堵塞造成的原因，这个问题的具体结论我还不得而知。