

操作系统实验报告

18 级计科超算 18340208 张洪宾

1 实验题目

具有中断处理的内核

2 实验目的

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

3 实验要求

- 知道 PC 系统的中断硬件系统的原理
- 掌握 x86 汇编语言对时钟中断的响应处理编程方法
- 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。
- 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

4 实验内容

- 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。

- 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。
- 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!”。
- 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
- 在老师要求的基础上，我将上次做的显示时间的模块稍微修改，添加到时间中断相应中，使得风火轮旁边可以实时显示当前时间。

5 实验方案

5.1 实验环境

- 操作系统：macOS Catalina 10.15.4
- 编译环境：Ubuntu 18.04 虚拟机
- 虚拟机：VirtualBox
- 汇编语言编译工具：NASM version 2.13.02
- C 语言编译工具：gcc version 7.5.0 (Ubuntu 7.5.0-3ubuntu1 18.04)
- 链接器：GNU ld (GNU Binutils for Ubuntu) 2.30
- 符号表工具：names(nm 命令)

5.2 编写风火轮程序

这个风火轮程序并不是特别复杂，就是在显示内存段的一个位置交替显示’|’、’/’和’\’。为了使得衔接更合理，我将风火轮改成了’-’、’\’、’|’、’/’的组合。

实现的关键是在显示内存段打印特定的字符，这个我们在实验一就已经完成了，所以并不是很难。这部分程序的关键代码如下 (完整代码见 wheel.asm):

```

1      mov ax,0B800h
2      mov gs,ax
3      mov ax,[count]
4      cmp ax,0
5      je  init
6      dec ax

```

```

7      mov [count],ax
8      mov si,wheel
9      add si,ax
10     mov byte al,[si]
11     mov ah,03h
12     mov [gs:(80*24+79)*2],ax;将字符放到显示内存段
13     jmp loop

```

为该程序制作了引导程序并放在裸机运行，可以看到风火轮在屏幕右下角快速旋转。

5.3 实现中断

本次实验最重要的事情是让我们学会实现中断。所以首先我们要了解中断的原理。

5.3.1 可编程中断控制器

在这里参考了[详解 8259A](#)。

可编程中断控制器 (PIC - Programmable Interrupt Controller) 是微机系统中管理设备中断请求的管理者。当 PIC 向处理器的 INT 引脚发出一个中断信号时, 处理器会立刻停下当时所做的事情并询问 PIC 需要执行哪个中断服务请求。PIC 则通过向数据总线发出与中断请求对应的中断号来告知处理器要执行哪个中断服务过程。处理器则根据读取的中断号通过查询中断向量表 (在 32 位保护模式下是中断描述符表) 取得相关设备的中断向量 (即中断服务程序的地址) 并开始执行中断服务程序。当中断服务程序执行结束, 处理器就继续执行被中断信号打断的程序。

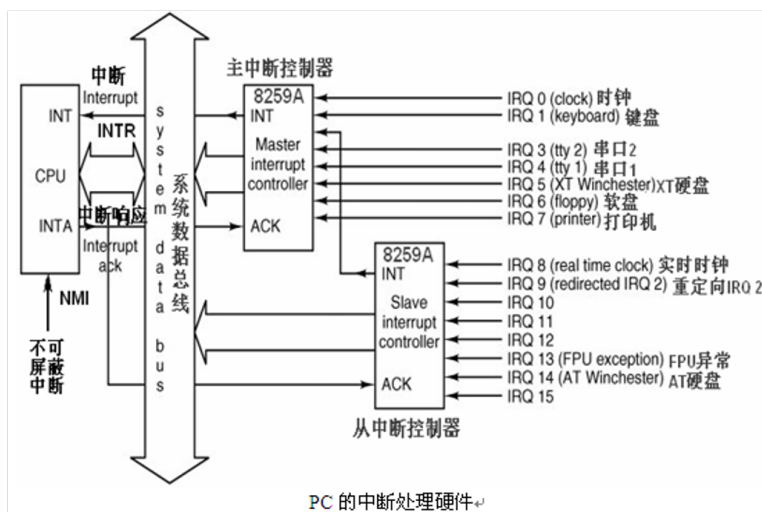


图 1: x86 使用的 8259A 中断控制器

它是联系中断和 CPU 的一个部件, 让 CPU 能够恰当地得知发生了什么中断并且有顺序地去执行中断。

5.3.2 8259A 的 I/O 端口

- 每一个 8259A 芯片都有两个 I/O ports，程序员可以通过它们对 8259A 进行编程。
- Master 8259A 的端口地址是 0x20，0x21；Slave 8259A 的端口地址是 0xA0，0xA1。

常用操作是：

```
mov al,20h ; AL = EOI  
out 20h,al ; 发送 EOI 到主 8259A  
out 0A0h,al  
iret
```

用于结束中断。

5.3.3 内存的分配

0x00000~0x003FF:	中断向量表
0x00400~0x004FF:	BIOS 数据区
0x00500~0x07BFF:	自由内存区
0x07C00~0x07DFF:	引导程序加载区
0x07E00~0x9FFFF:	自由内存区
0xA0000~0xBFFFF:	显示内存区
0xC0000~0xFFFFF:	BIOS 中断处理程序区

图 2: 内存分配表

通过内存分配表我们可以看出，除了我们熟悉的引导程序加载区和显示内存区，还有中断向量表、BIOS 数据区、BIOS 中断处理程序区。我们这次实验需要用到的是其中的中断向量表。

5.3.4 中断向量表

中断向量表存储了中断服务程序的地址。中断向量表的段基址为 0，中断服务程序的地址共有 32 位，其中低位为偏移地址，高位为段地址。

所以一个中断向量的地址占四个字节，也就是说，第 n 号中断的中断服务程序的地址应该是 $0:4n$ 。

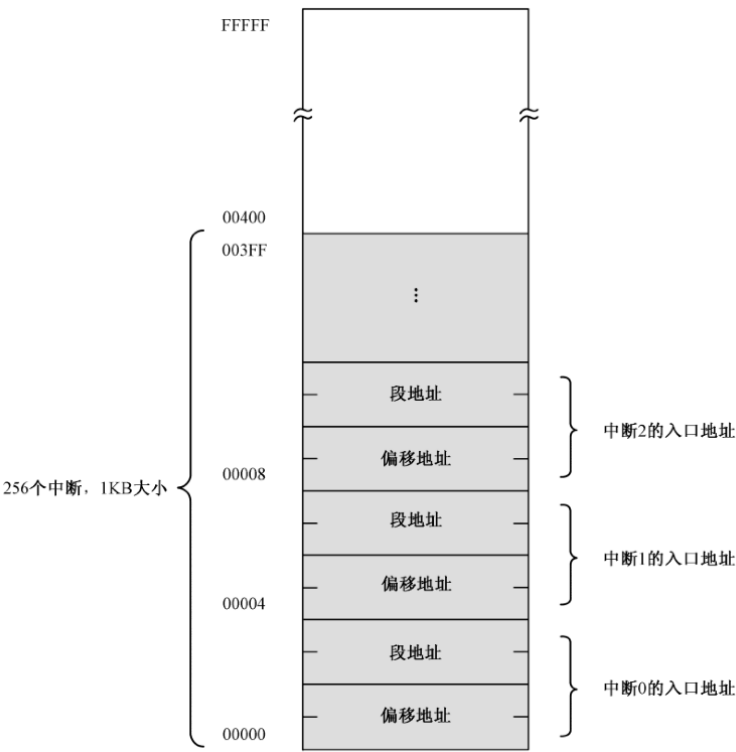


图 3: 中断向量表示意图

了解了这个知识我们才可能修改中断向量表，将中断服务程序换成我们自己的。

5.3.5 int 指令和 iret 指令

以前的实验中，我只是简单用类似于 `int 10h` 这样的指令来调用中断的功能，但是并没有理解这个指令的原理。

当调用中断当功能的时候，系统会对现场进行保护，首先让标志寄存器入栈，然后让 CS 入栈，最后让 IP 入栈。而 `iret` 则按对应的顺序让这些寄存器出栈。

5.3.6 实现对中断向量表的修改

中断向量表的修改其实就是将目标中断向量的地址改成我们自己实现的中断服务程序的入口地址，根据前面的讲述，如果要修改第 n 号中断的中断向量，首先我们找到 $0:4n$ 这个地址，然后将 $[0:4n]$ 开始的两个字节修改为我们实现的中断服务程序的 IP，将 $[0:4n + 2]$ 开始的两个字节修改为中断服务程序的 CS。

基于这个方法，我实现了两个模块，一个是将中断服务程序写入中断向量表指定的位置，另一个是将中断向量表的某一个中断向量转移到中断向量表的另外一个地方。在这里我本来在内核中使用了函数实现，最后发现需要在用户程序中调用这些函数，但是因为内核和用户程序不是作为一个整体进行链接的，所以相互调用很麻烦，于是我将上述两个模块修改为宏的形式，这样子就便于代码的复用了。

这两个模块如下：

```
1 %macro Modify_Vector 2           ; 修改中断向量表
2     push ax
3     push es
4     mov ax, 0
5     mov es, ax                 ; 基址为0
6     mov word[es:%1*4], %2      ; 设置中断服务程序偏移地址 ip
7     mov ax, cs
8     mov word[es:%1*4+2], ax    ; 设置中断服务程序的段地址CS
9     pop es
10    pop ax
11 %endmacro
12
13 %macro Replace_Vector 2          ; 将参数1的中断向量转移至参数2处
14     push ax
15     push es
16     push si
17     mov ax, 0
18     mov es, ax
19     mov si, [es:%1*4]
20     mov [es:%2*4], si
21     mov si, [es:%1*4+2]
22     mov [es:%2*4+2], si
23     pop si
24     pop es
25     pop ax
26 %endmacro
```

5.4 在原型操作系统中加入时钟中断

拥有了上面的函数，只要将前面的风火轮程序做适当修改，移植到内核中即可。然后在进入内核的时候用 `Modify_Vector` 将它写入中断向量表 08h 处。而为了保存原来的中断向量，则应该把原来的 08h 中断写到其他位置。查中断向量表，得：34h-40h 中断号未使用，于是我们就可以考虑用其中的中断号来保存被替代的中断向量。

在这里为了使风火轮旋转的速度不会太快也不会太慢，我用了应该 `delay` 变量，当 `delay` 不为 0 的时候将 `delay` 做自减并中断返回，当 `delay` 为 0 的时候在 (24,79) 处打印下一个字符，并将 `delay` 复原。经过检测，当 `delay` 初始值为 4 的时候效果较好。

而在这部分中比较麻烦的地方应该是我自己添加的实时显示时间了。在上次实验中，我已经写了获取当前时间的函数。这次为了实时显示，就是在中断服务程序中打印当前时间的字符串。而上网了解到，中断处理器一秒钟大约调用 18.2 次 08h 号中断，即使加上我们的 `delay` 变量，一秒钟的有效输出次数也大约为 4 次左右，足以实现秒精度的时间更新了。

于是我在 `lib.c` 中实现了在屏幕右下角输出当前时间的函数：

```
1 void print_time(){
2     unsigned char year = getTime(9);
3     unsigned char month = getTime(8);
4     unsigned char day = getTime(7);
5     unsigned char hour = getTime(4);
6     unsigned char min = getTime(2);
7     unsigned char sec = getTime(0);
8     char time[32];
9     for(int i = 0; i < 32; i++)time[i] = 0;
10    add_char(time, '2');
11    add_char(time, '0');
12    Splicing_string(time, num_to_string(bcd_to_num(year), 10));
13    add_char(time, '-');
14    Splicing_string(time, num_to_string(bcd_to_num(month), 10));
15    add_char(time, '-');
16    Splicing_string(time, num_to_string(bcd_to_num(day), 10));
17    add_char(time, ' ');
18    add_char(time, ' ');
19    Splicing_string(time, num_to_string(bcd_to_num(hour), 10));
20    add_char(time, ':');
21    Splicing_string(time, num_to_string(bcd_to_num(min), 10));
22    add_char(time, ':');
23    Splicing_string(time, num_to_string(bcd_to_num(sec), 10));
24    for(int i = 0; i < 21; i++){
25        print_pos(time[i], 24, i + 50);
26    }
27    return;
```

并在中断调用的程序中调用它，在输出风火轮字符的同时输出时间，便实现了时间的实时更新。

在这个基础上，我写了一个函数来打开、关闭时间中断响应程序，具体来说，检测到中断向量表的 08h 处为我的中断服务程序的时候恢复为原来的中断，否则将中断服务程序写入中断向量表的 08h 处来打开时间中断响应。

5.5 在用户程序中实现键盘中断响应

当用户按下键盘的某个按键之后，键盘会通过 8259A 中断控制器向 CPU 申请中断，这是一个硬中断。中断相应后，中断控制器送出 09h 中断号，然后进入中断服务程序进行下一步的操作。

在这里为了实现目标，我考虑将在 09h 中断服务程序中添加上打印”OUCH!OUCH!” 的功能。而由于中断可以嵌套，所以可以采用先将原来的 09h 号中断服务程序的地址转移到中断向量表其他位置，然后再由新的中断服务程序在打印”OUCH!OUCH!” 后调用原来的中断服务程序。

所以具体的过程变成了：当执行用户程序的时候，当敲击键盘时，键盘通过中断控制器向 CPU 申请中断，中断响应后，中断控制器送出 09h 号中断向量号，然后调用中断服务程序，中断服务程序在打印完毕中”OUCH!OUCH!” 调用原来的中断服务程序，然后发送 EOI 并中断返回。

由于要只在用户程序中实现，我只在用户程序中实现了这个中断响应。

进入用户程序时，用：

```
1 Replace_Vector 09h,42h
2 Modify_Vector 09h,intouch
```

先将原来中断向量表 09h 向量换到空闲位置 42h，然后将中断向量表 09h 处换成我们自己定义的中断服务程序的地址，就完成了对键盘中断的响应。

5.6 为 33 号、34 号、35 号和 36 号中断编写中断服务程序

这一部分跟之前的改写中断没有特别大的区别，我在这里写了四个类似的程序，分别是在屏幕的不同的四分之一处打印我的个人信息。

以 33 号中断为例：

```
1 int_33:
2     push ax
3     push si
4     push ds
5     push gs
6     print_with_color name,name_len,4,16,06h
7     print_with_color id,id_len,5,15,06h
8     print_with_color msg1,msg1_len,6,13,06h
9 check0:
```



```

10     mov ah, 01h
11     int 16h
12     jz  check0
13     mov ah, 00h
14     int 16h
15     cmp ax, 2e03h        ; 检测 Ctrl + C
16     jne  check0
17     pop  gs
18     pop  ds
19     pop  si
20     pop  ax
21     iret

```

在屏幕中的左上角会打印我的个人信息，可以用 Ctrl + C 来退出中断。

然后我在内核设计了一个接口用于调用这几个中断，在终端中输入 int 33(34、35、36) 来调用 33(34、35、36) 号中断。

5.7 对软盘结构的调整

在我将上述程序设计好并可以成功在裸机上运行的时候，打印出来的信息会出现截断的现象。我百思不得其解，于是查看了生成的 img 的二进制形式。

```

000023c0: 41 42 43 44 45 46 00 7A 68 61 6E 67 68 62 20 23  ABCDEF.zhanghb.#
000023d0: 20 00 00 00 20 20 20 5A 48 42 4F 53 20 76 65 72  .....ZHB0S.ver
000023e0: 73 69 6F 6E 20 31 2E 30 20 28 78 38 36 5F 31 36  sion.1.0.(x86_16
000023f0: 2D 70 63 29 0D 0A 20 20 20 54 68 65 73 65 20 73  -pc)....These.s
00002400: 60 50 06 56 B8 00 00 8E C0 26 8B 36 24 00 26 89  `P.V8...@&.6$.&.
00002410: 36 08 01 26 8B 36 26 00 26 89 36 0A 01 5E 07 58  6..&.6&.&.6..^X
00002420: 50 06 B8 00 00 8E C0 26 C7 06 24 00 D4 A4 8C C8  P.8...@&G$.T$.H
00002430: 26 A3 26 00 07 58 B8 00 B8 8E E8 66 FF 0E 0C A5  &#&..X8.8.hf...%
00002440: 75 F9 66 C7 06 0C A5 FF E0 F5 05 80 3E 14 A5 01  uyfG...%`u...>.%
00002450: 74 19 80 3E 14 A5 02 74 50 80 3E 14 A5 03 0F 84  t...>.%tP.>.%...
00002460: 85 00 80 3E 14 A5 04 0F 84 B7 00 FF 06 10 A5 FF  ...>.%...7....%
00002470: 06 12 A5 B8 0B 00 2B 06 10 A5 74 0C B8 1F 00 2B  ..%8...+.%t.8..+
00002480: 06 12 A5 74 14 E9 00 01 B8 1F 00 2B 06 12 A5 74  ..%t.i..8...+.%t

```

图 4: 查看的二进制文件

图中截取的部分是内核区域与第一个用户程序的交界处，因为我的第一个用户程序放在了整张软盘的第 19 个扇区，所以起始地址为 $18 * 512 = 9216 = 02400h$ ，即图中左方的方框的地址。而第一个用户程序扇区前的 15 个扇区用于存放内核，但是我们可以看到，内核的信息被截断了。于是我查看了一下编译内核的生成文件的大小，为 9664 字节，已经远远大于我分配的 15 个扇区 ($15 * 512 = 7680$ 字节)。

由此我发现我上次的分配并不是很合理，并没有考虑到随着内核功能的扩展，我给内核分配的磁盘空间可能会不够用。于是我重新对整个软盘的结构进行了组织。

程序	柱面号	磁头号	起始扇区号	扇区数	功能	载入内存的位置
引导程序	0	0	1	1	打印提示信息， 加载用户信息表和内核	07C00h
用户程序表	0	0	2	2	存储用户程序存储的位置 加载到内存中的位置	7E00h
内核	0	0	3	34	常驻内存，实现 OS 最基本的功能	8000h
用户程序 1	1	0	1	2	在屏幕左上角使个人信息跳动	0B300h
用户程序 1	1	0	3	2	在屏幕右上角使个人信息跳动	0B500h
用户程序 3	1	0	5	2	在屏幕左下角使个人信息跳动	0B700h
用户程序 4	1	0	7	2	在屏幕右下角使个人信息跳动	0B900h

这次我给内核和用户程序留下了充足的空间，便于它们的扩展。在完成修改后，操作系统又可以重新正常运行了。

5.8 修改 Makefile

对之前使用的 Makefile 进行修改，将 wheel.asm 加入编译的范畴。

```

IMG = zhbos.img
INFO = loader.bin user_info.bin
KERNEL = kernel.bin
PRO = a.com b.com c.com d.com

all:    pro img clean

img:$(INFO) $(KERNEL) $(PRO)
ifeq ($(IMG), $(wildcard $(IMG)))
    rm $(IMG)
endif
/sbin/mkfs.msdos -C $(IMG) 1440
dd if=loader.bin of=$(IMG) conv=notrunc
dd if=user_info.bin of=$(IMG) seek=1 conv=notrunc
dd if=kernel.bin of=$(IMG) seek=2 conv=notrunc
dd if=a.com of=$(IMG) seek=36 conv=notrunc

```

```

dd if=b.com of=$(IMG) seek=38 conv=notrunc
dd if=c.com of=$(IMG) seek=40 conv=notrunc
dd if=d.com of=$(IMG) seek=42 conv=notrunc

pro:$(INFO) $(KERNEL) $(PRO)
loader.bin: loader.asm
    nasm $< -o $@
user_info.bin: user_info.asm
    nasm $< -o $@
kernel.bin: kernel.o lib.o libc.o wheel.o
    ld -m elf_i386 -N -Ttext 0x8000 --oformat binary $^ -o $@

%.o : %.asm
    nasm -f elf32 $< -o $@

%.o : %.c
    gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding
    -fno-PIE -masm=intel -c $< -o $@
%.com : %.asm
    nasm $< -o $@

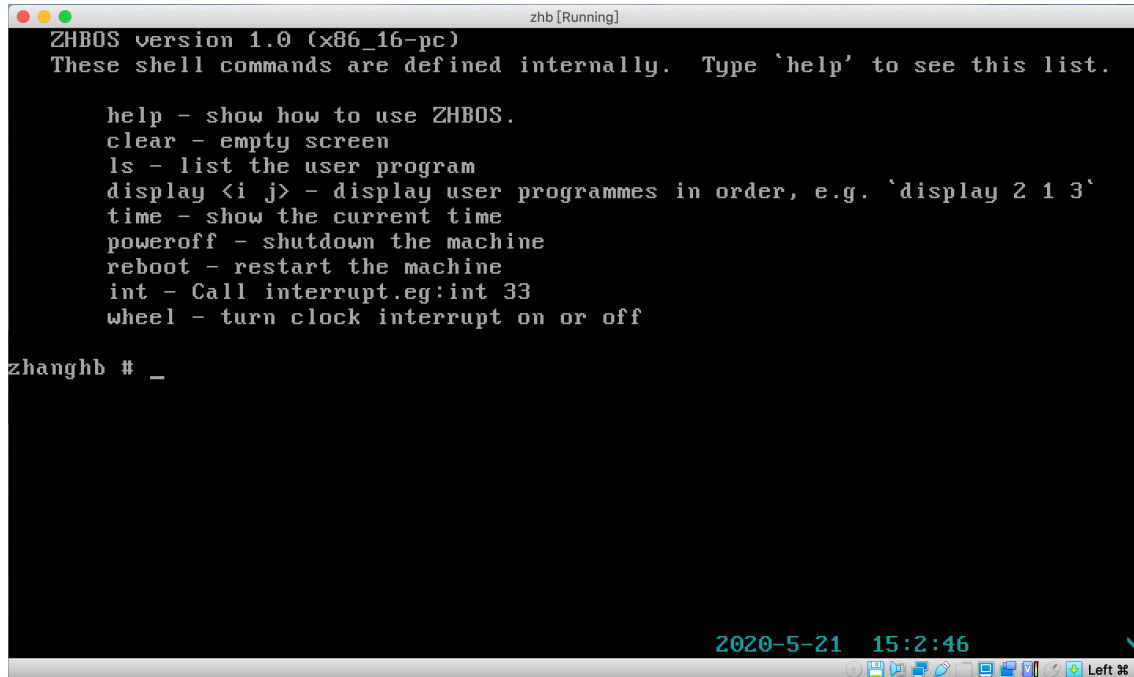
.PHONY: clean
clean:
    rm -rf *o *bin *com

```

6 实验过程

上次的实验已经讲述了 ls、help、display、time、clear、poweroff、reboot 等指令，本次就不再赘述，在这里详细介绍一下本次新添加的功能。

6.1 风火轮运转及实时显示时间

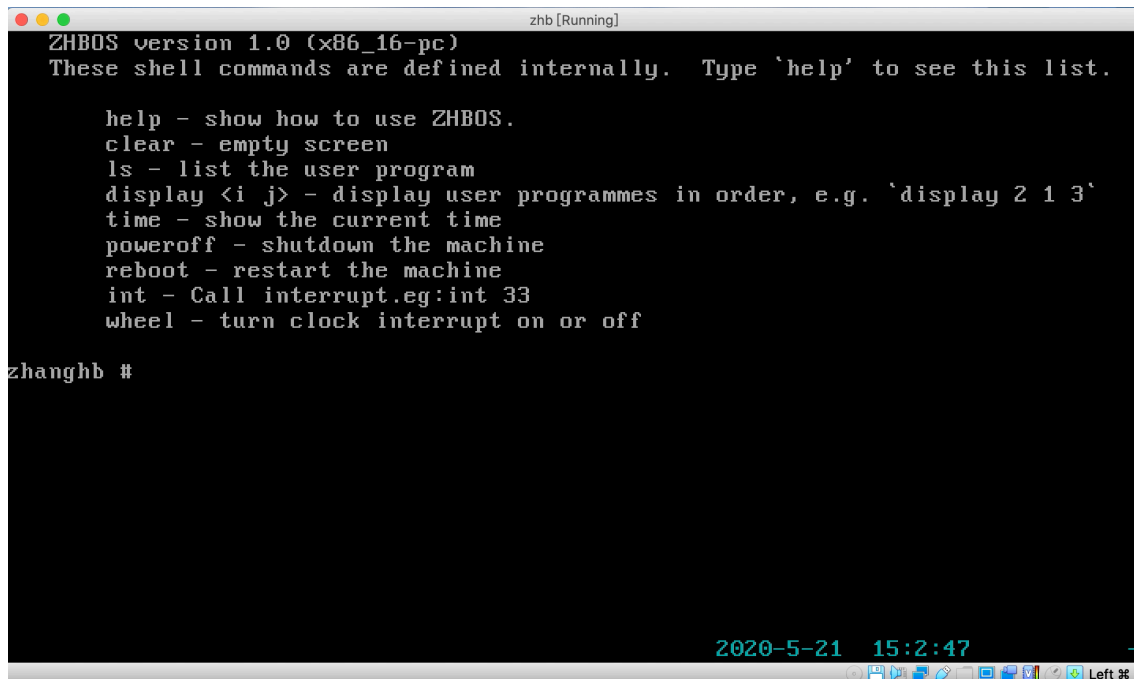


```
zhib [Running]
ZHBOS version 1.0 (x86_16-pc)
These shell commands are defined internally.  Type 'help' to see this list.

help - show how to use ZHBOS.
clear - empty screen
ls - list the user program
display <i j> - display user programmes in order, e.g. 'display 2 1 3'
time - show the current time
poweroff - shutdown the machine
reboot - restart the machine
int - Call interrupt.eg:int 33
wheel - turn clock interrupt on or off

zhanghb # _

2020-5-21 15:2:46
```



```
zhib [Running]
ZHBOS version 1.0 (x86_16-pc)
These shell commands are defined internally.  Type 'help' to see this list.

help - show how to use ZHBOS.
clear - empty screen
ls - list the user program
display <i j> - display user programmes in order, e.g. 'display 2 1 3'
time - show the current time
poweroff - shutdown the machine
reboot - restart the machine
int - Call interrupt.eg:int 33
wheel - turn clock interrupt on or off

zhanghb #

2020-5-21 15:2:47
```

把软盘加载到虚拟机中并启动，可以看到上面的界面：右下角的时钟实时显示，而位置为 (24,79) 的风火轮不断旋转。

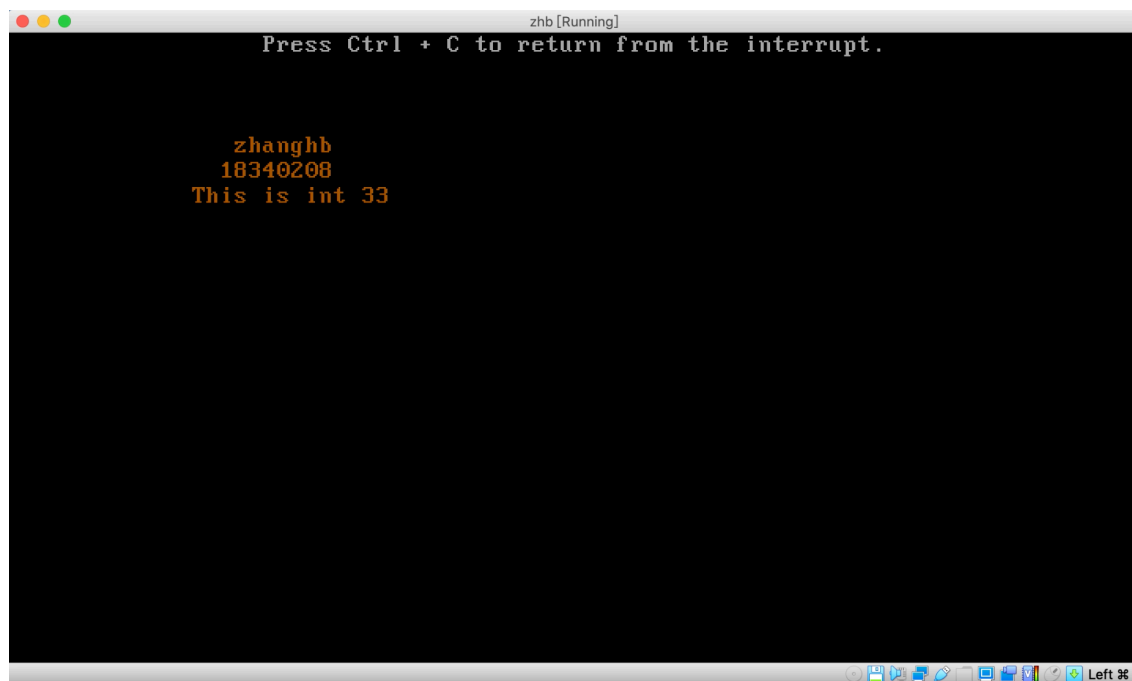
6.2 在用户程序中使用键盘中断响应

运行用户程序 1，在敲击键盘，屏幕右侧会出现”OUCH!OUCH!” 字符串。而在其他用户程序中也会在其他位置出现”OUCH!OUCH!” 字符串。



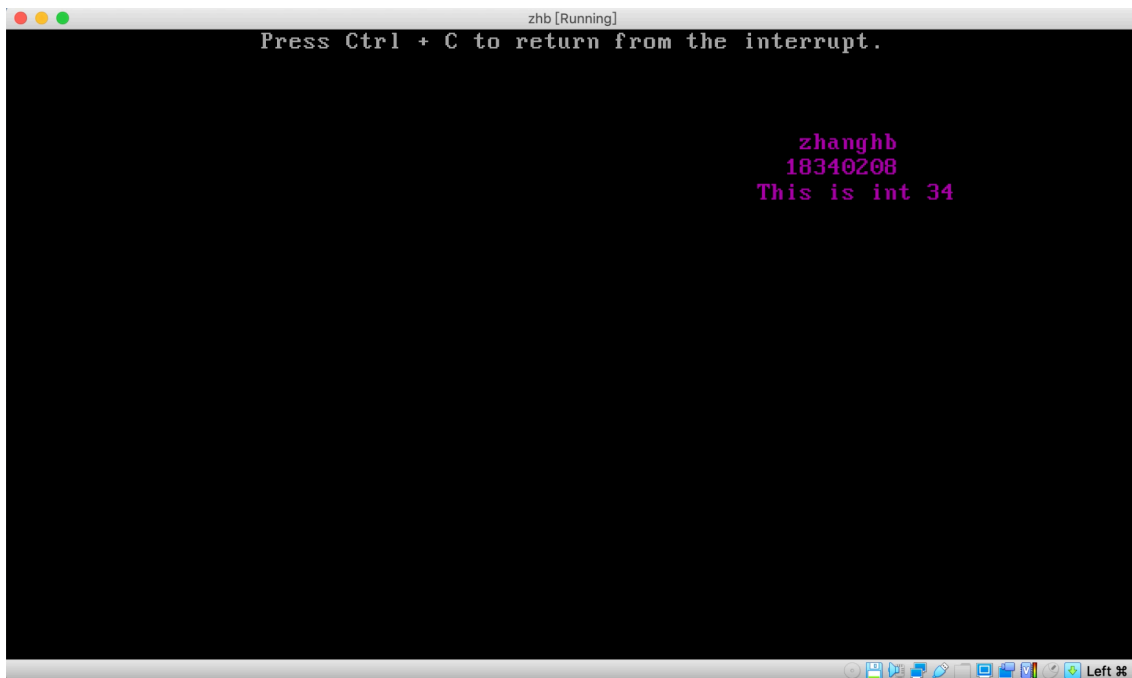
6.3 调用 33、34、35、36 号中断

在终端中输入 int 33, 可以看到



在屏幕的左上角出现了用橙色字符打印的我的个人信息。

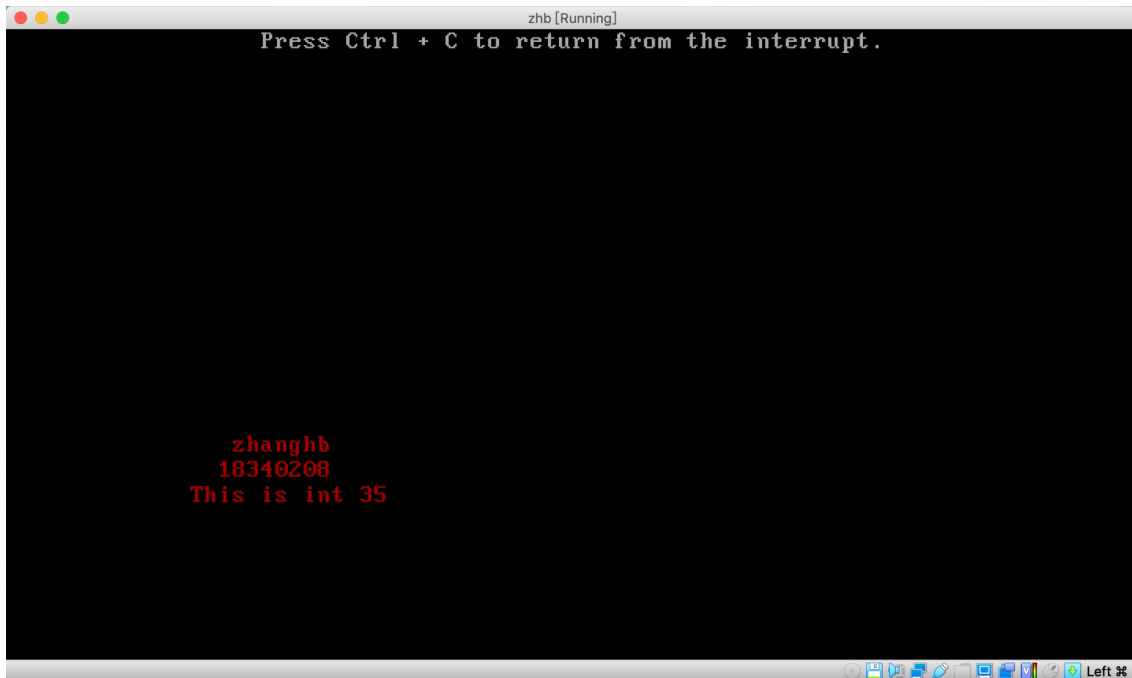
按下 Ctrl + C 退出，在终端中输入 int 34, 可以看到



A terminal window titled "zhh [Running]" with a black background. At the top, it says "Press Ctrl + C to return from the interrupt." In the upper right corner, the following text is printed in purple: "zhanghb", "18340208", and "This is int 34". The window has standard macOS window controls (red, yellow, green buttons) at the top left and a dock with various icons at the bottom.

在屏幕的右上角出现了用紫色字符打印的我的个人信息。

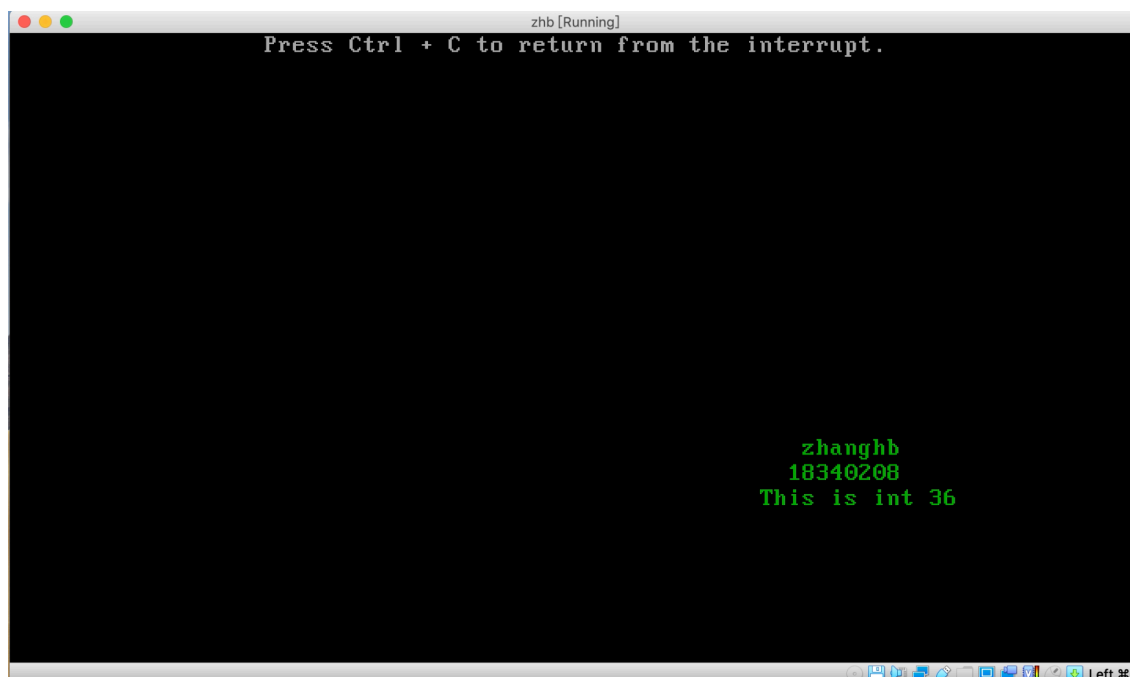
按下 Ctrl + C 退出，在终端中输入 int 35, 可以看到



A terminal window titled "zhh [Running]" with a black background. At the top, it says "Press Ctrl + C to return from the interrupt." In the lower left corner, the following text is printed in red: "zhanghb", "18340208", and "This is int 35". The window has standard macOS window controls (red, yellow, green buttons) at the top left and a dock with various icons at the bottom.

在屏幕的左下角出现了用红色字符打印的我的个人信息。

按下 Ctrl + C 退出，在终端中输入 int 36, 可以看到



在屏幕的右下角出现了用绿色字符打印的我的个人信息。

7 实验总结

这次实验并没有像实验三构建系统那么复杂，但是也十分麻烦。

7.1 对代码的重构

之前我在用户程序中使用了类似于 `org 0A500h` 这样的代码，然后在相应的用户程序表中将这个地址存储了起来。这样子上次实验中非常顺利地完成了相应的任务，但是在这次实验出了问题：

因为我发现分配给内核的软盘大小和内存大小都太小了，所以我在修改用户程序在软盘中的位置的同时也需要将用户程序在内存中的位置做适当地修改。然后我就需要对用户程序表和用户程序中的内存地址做修改，总共需要修改 5 个文件中的内容，非常地麻烦。于是我就想到了使用 `%include "Macro.asm"` 的办法，将我们需要使用的全局变量放到 `Macro.asm` 中，如果某个文件中需要用到其中的变量，只需要 `%include "Macro.asm"`。通过这种方式，下次如果还需要修改用户程序在内存中的地址，我只需要对 `Macro.asm` 中的值进行修改。

另外本来我是用函数的方式实现了写中断向量和将中断向量做转移的方法，但是我发现这样子让用户程序调用就会非常麻烦。所以我采用了宏定义的方式，并将这些函数放进 `Macro.asm`，只要用户程序 `include` 它就可以了。

7.2 段寄存器值得重点关注

在实验过程中，我最开始使用 pusha 之后就没有注意对 cs、ds、es、ss 等段寄存器进行保护，造成了写完中断向量表后无法继续运行程序。

7.3 软盘与扇区的关系

在《自己动手写操作系统》这本书的源代码的某个注释给出了这样的公式：

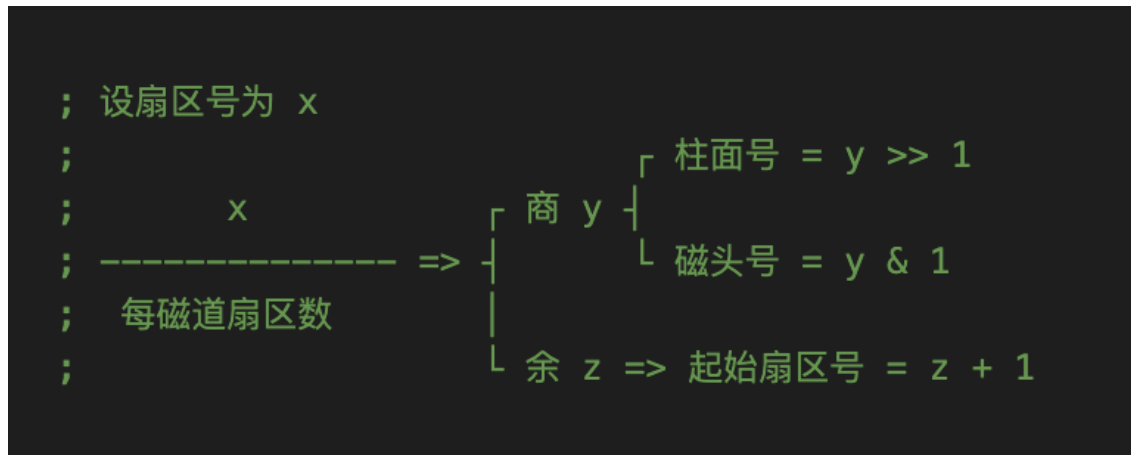


图 5: 计算软盘中的某个扇区的柱面号、磁头号 and 扇区号

通过这个公式我计算出了重新组织后的软盘中各个扇区应该拥有的柱面号、磁头号 and 扇区号。

7.4 心得体会

一转眼操作系统实验已经做了四个了，虽然每次做之前都一脸懵逼，但是通过认真学习真的可以比较深入理解操作系统的运行机制，就像这次我才真正理解了中断的运行机制。

不过目前我的代码还是存在一定的问题的，我并没有对代码做很好地封装，全都堆在 lib.c 和 lib.asm 中，下次我会尽量把它封装成比较好的 C 函数库，这样子使得代码更加整洁。