

# 操作系统实验报告

18 级计科超算 18340208 张洪宾

## 1 实验题目

实现系统调用

## 2 实验目的

- 学习掌握 PC 系统的软中断指令
- 掌握操作系统内核对用户提供服务的系统调用程序设计方法
- 掌握 C 语言的库设计方法
- 掌握用户程序请求系统服务的方法

## 3 实验要求

- 了解 PC 系统的软中断指令的原理
- 掌握 x86 汇编语言软中断的响应处理编程方法
- 扩展实验四的内核程序，增加输入输出服务的系统调用。
- C 语言的库设计，实现 putch()、getch()、printf()、scanf() 等基本输入输出库过程。
- 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

- 修改实验 4 的内核代码，先编写 save() 和 restart() 两个汇编过程，分别用于中断处理的现场保护和现场恢复，内核定义一个保护现场的数据结构，以后，处理程序的开头都调用 save() 保存中断现场，处理完后都用 restart() 恢复中断现场。

- 内核增加 int 20h、int 21h 和 int 22h 软中断的处理程序，其中，int 20h 用于用户程序结束是返回内核准备接受命令的状态；int 21h 用于系统调用，并实现 3-5 个简单系统调用功能；int 22h 功能未定，先实现为屏幕某处显示 INT22H。
- 保留无敌风火轮显示，取消触碰键盘显示 OUCH! 的功能。
- 进行 C 语言的库设计，实现 putch()、getch()、gets()、puts()、printf()、scanf() 等基本输入输出库过程，汇编产生 libs.obj。
- 利用自己设计的 C 库 libs.obj，编写一个使用这些库函数的 C 语言用户程序，再编译，在与 libs.obj 一起链接，产生 COM 程序。增加内核命令执行这个程序。

## 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)
- 调试工具：bochs

### 5.2 配置 bochs 环境

由于这次实验中的实现 save 和 restart 对堆栈操作的精度要求非常高，稍有不慎就会出现错误。我一度因为 restart 无法正确返回而停滞不前。

为了解决这个问题，我决定在实验工具链中加入 bochs 这一调试工具。我在我的 Ubuntu 系统下安装了 bochs，然后根据它给的 sample 写出了配置文件。

```

romimage: file=$BXSHARE/BIOS-bochs-latest
vgaromimage: file=$BXSHARE/VGABIOS-lgpl-latest
# 1.44=磁盘镜像位置
floppya: 1_44=zhbos.img, status=inserted
# 从软盘启动
boot: floppy
log: bochsout.txt
# 置鼠标不可用
mouse: enabled=0
keyboard: keymap = # $BXSHARE/keymaps/x11-pc-us.map
megs: 32

```

然后就可以进行调试了：

先在内核 0x8000 处用 b 命令设断点，然后用 c 命令执行到内核处，停下，如下图：

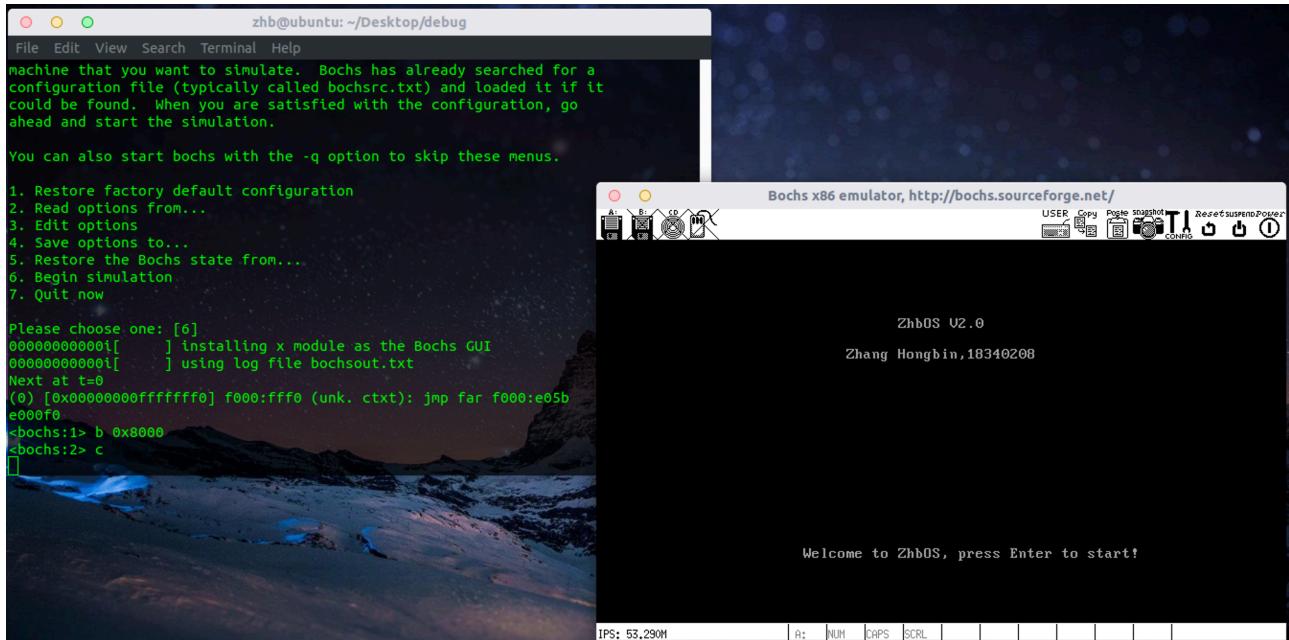


图 1: 在内核前停下

对于本次实验比较常用的栈和中断向量表，我们可以分别用 info ivt 和 print-stack 命令来查看具体的情况，如图：

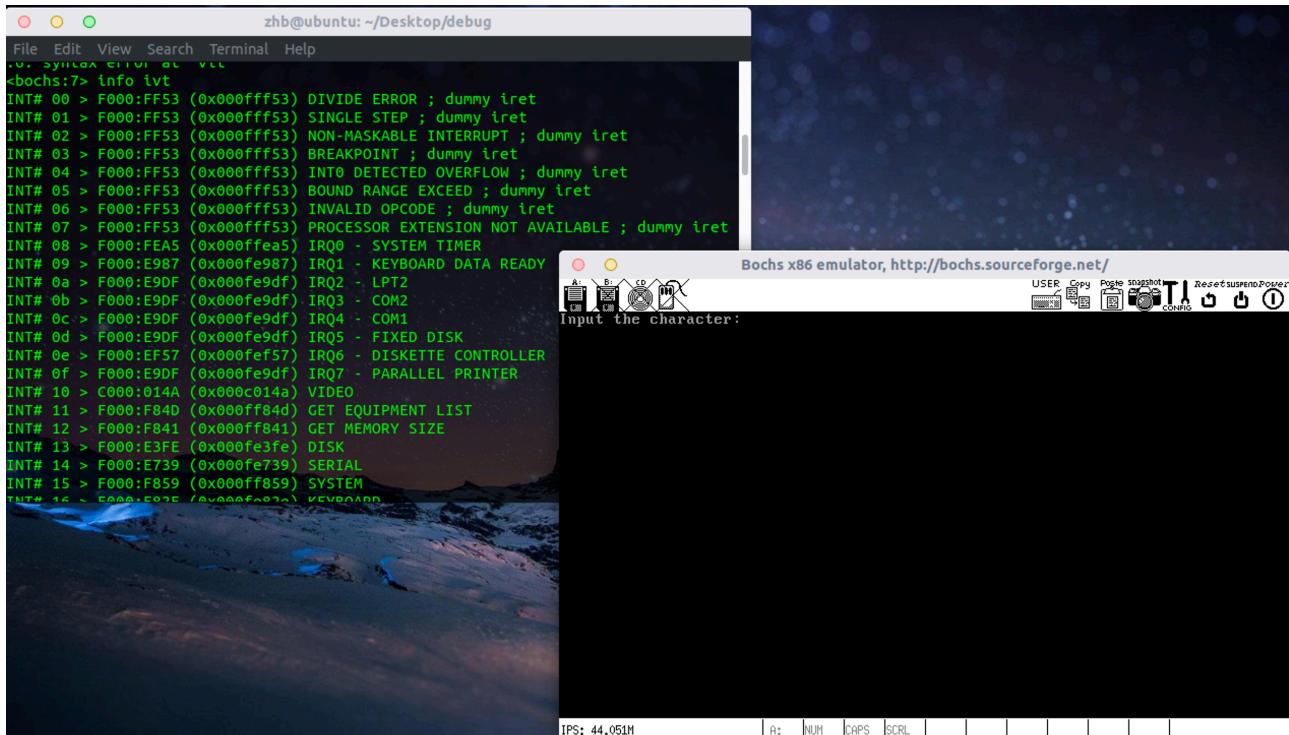


图 2: 用 info ivt 查看中断向量表的情况

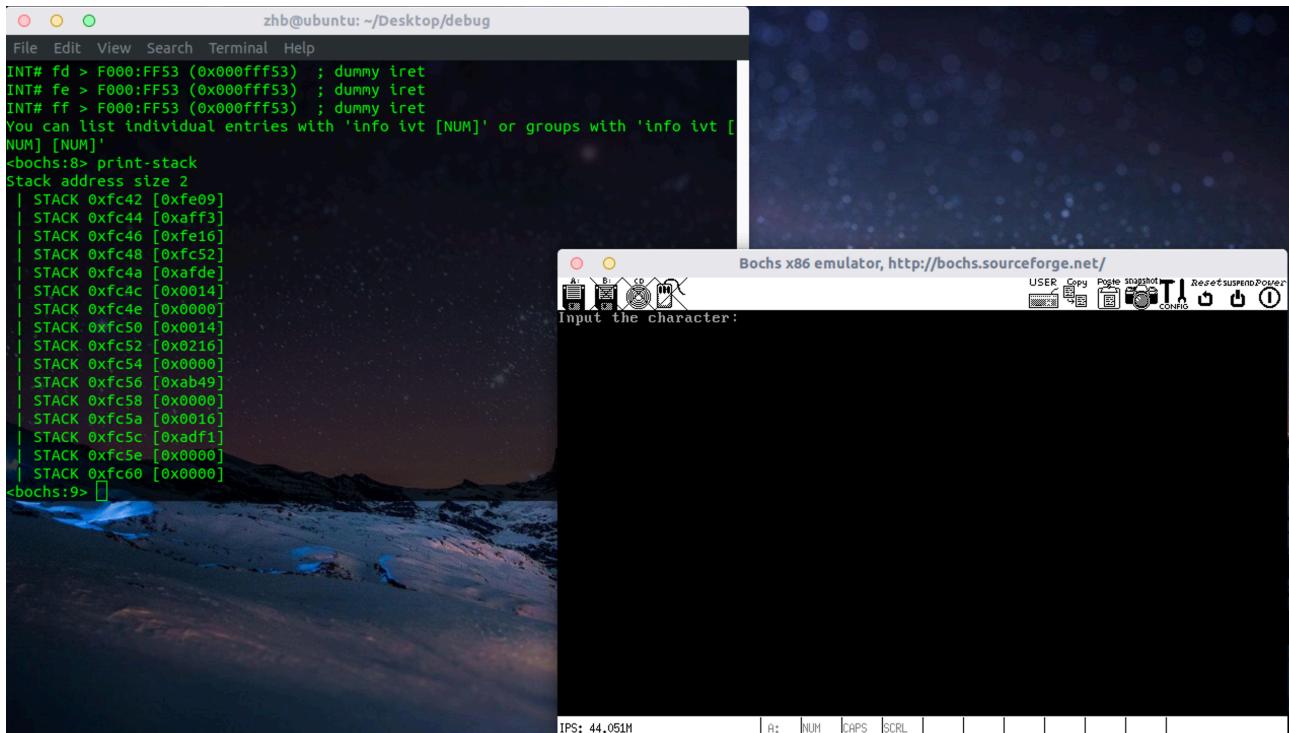


图 3: 用 print-stack 查看栈的情况

通过调试的方法就可以比较方便的找到了栈的错误，从而对代码做出一定的调整。

### 5.3 编写 save 和 restart 模块

考虑到 save 和 restart 对栈的精确程度非常之高，所以我觉得这里不太适合使用函数的方式，因为 call 会对 ip 进行压栈，会对栈的状态进行修改，不利于我们实现精细的操作。考虑了很久我决定再次使用宏的方式来实现这部分，通过宏的方式就相当于复制粘贴，就不会出现对栈的修改，同时可以以模块的方式对 save 和 restart 进行封装。

受此启发，我决定将寄存器状态表也以汇编中宏的方式进行定义，这样子就省去了从 C 语言 extern 的工作，减小了工作的复杂程度。

#### 5.3.1 用宏定义的方式定义整个寄存器表

这个模块我参考了老师和网上的代码，进行适当的修改，然后得出了寄存器表的宏：

```
1 %macro register 0          ; 参数：段值
2   dw 0                     ; ax, 偏移量=+0
3   dw 0                     ; cx, 偏移量=+2
4   dw 0                     ; dx, 偏移量=+4
5   dw 0                     ; bx, 偏移量=+6
6   dw 0FE00h               ; sp, 偏移量=+8
7   dw 0                     ; bp, 偏移量=+10
8   dw 0                     ; si, 偏移量=+12
9   dw 0                     ; di, 偏移量=+14
10  dw 0                     ; ds, 偏移量=+16
11  dw 0                     ; es, 偏移量=+18
12  dw 0                     ; fs, 偏移量=+20
13  dw 0B800h               ; gs, 偏移量=+22
14  dw 0                     ; ss, 偏移量=+24
15  dw 0                     ; ip, 偏移量=+26
16  dw 0                     ; cs, 偏移量=+28
17  dw 512                  ; flags, 偏移量=+30
18 %endmacro
```

#### 5.3.2 封装 Save 模块

当进入中断后，我直接将寄存器表保存了起来。然后遇到了一个非常严重的问题，当我寄存器保存在寄存器表后，我运行程序，发现输入被阻塞了，而时间中断正常运行，显示着当前的时间。我用 bochs 调试，发现可能是因为保存的过程中因为奇奇怪怪的原因对寄存器的状态造成了破坏，为了解决这个问题，我决定对所有寄存器进行压栈，然后通过 sp 找到对应的寄存器保存到寄存器表中。

此处代码过长，只截取一部分：

```

1 %macro Save 1      ;参数：寄存器状态表
2     push ss
3     push gs
4     push fs
5     push es
6     push ds
7     push di
8     push si
9     push bp
10    push sp
11    push bx
12    push dx
13    push cx
14    push ax
15    mov bp, sp
16    mov di, %1
17    mov ax, [bp]
18    mov [cs:di], ax
19    mov ax, [bp+2]
20    ... (省略)
21 %endmacro

```

注意到，在压入 sp 之前我们压入了 8 个寄存器，再加上中断发生的时候压入的 flag、cs 和 ip 寄存器，在寄存器表中 sp 的值比进入中断之前 sp 的值小了  $11*2$ 。这是我们实现 restart 的一个最关键的要点。如果 sp 的值没有正确恢复的话，那么 iret 时就会跳转到错误的地方，并将错误的 flag 弹出，这样子的话 OS 就会崩溃。

接下来就是对 restart 的实现：

### 5.3.3 对 restart 模块的封装

正如上面分析的，从寄存器表中拿出恢复寄存器后，我们需要将 sp 寄存器恢复到正确的值。

这里还有一个细节，作为中间变量的 si 寄存器，我们需要在最后再将其恢复。

```

1 %macro Restart 1      ;参数：寄存器状态表
2     mov si, %1
3
4     mov ax, [cs:si+0]
5     mov cx, [cs:si+2]
6     mov dx, [cs:si+4]
7     mov bx, [cs:si+6]
8     mov sp, [cs:si+8]
9     mov bp, [cs:si+10]
10    mov di, [cs:si+14]
11    mov ds, [cs:si+16]

```

```

12  mov es, [cs:si+18]
13  mov fs, [cs:si+20]
14  mov gs, [cs:si+22]
15  mov ss, [cs:si+24]
16  add sp, 11*2
17
18  push word[cs:si+30]
19  push word[cs:si+28]
20  push word[cs:si+26]
21
22  push word[cs:si+12]
23  pop si
24 %endmacro

```

就此， save 和 restart 模块封装完毕。

## 5.4 实现软中断和系统调用

### 5.4.1 int 20h

根据我的设计，在原来的情况下，我在用户程序使用 retf 指令来返回内核的，这个技术细节在 lab3 的实验报告有详细的介绍。而此次实验，为了实现 int 20h，再配合上前面实现的 save 和 restart 模块，整个过程可以分解为：

用户程序执行完毕之后，用 int 20h 进入中断服务程序，首先用 save 保存寄存器的状态，然后想办法将栈最顶上的三个寄存器变成操作系统的 flag、cs 和 ip。

这个任务与普通中断的不同之处在于，普通中断最后还是要跳回原来的地址，所以我们只需要在 iret 前恢复栈就行了。但对于 int 20h 中断，我们需要跳转回内核，所以 iret 跳回的不应该是用户程序的地址。在这种情况下，结合我在 lab3 中的设计，应该先把栈恢复了，然后把栈顶的三个元素变成 flag、Run 模块的 cs 和 Run 模块的 ip，然后再用 iret 跳回。

具体的栈的示意图如下：

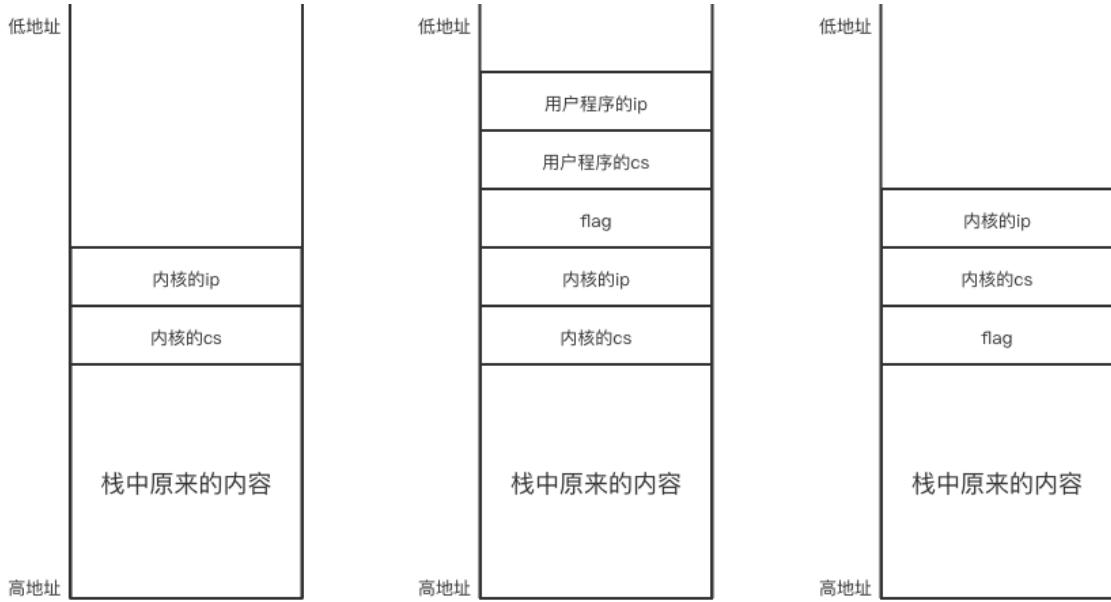


图 4: 左图是进入用户程序后返回前栈的状态，中间的图片是在用户程序调用 int 20h 后栈的状态，右图是 iret 前应该达到的状态

可以看出，我们需要在中断服务程序在 save 后恢复栈，然后将用户程序的 ip 和 cs 弹出，再想办法将 flag 放到 ip 和 cs 的下方。

在这里我采用的方法是先做两次 pop，丢弃用户程序 ip 和 cs，然后再做三次 pop 将 flag、内核 ip 和内核 cs 保存在三个通用寄存器，然后再按最终顺序让这三个通用寄存器入栈，最后使用 iret 回到内核。

```

1 int20h:
2     Save ret_table
3     Restart ret_table
4     pop ax
5     pop bx
6     pop cx
7     pop ax
8     pop bx
9     push cx
10    push bx
11    push ax
12    iret
13    Restart ret_table

```

然后用上次封装好的 Modify\_Vector 模块和 Replace\_Vector 模块，在进入内核的时候将该软中断写入中断向量表就行了。

### 5.4.2 int 21h

其实这次实现的 int 21h 和上次实现的 int 33, int 34, int 35, int 36 没有什么本质上的区别。但是这次要使用我们封装的 save 模块和 restart 模块。

这次我设计的系统调用是显示我的个人信息，不过与上次不同的是，这次的个人信息显示一小段时间就会自动从中断返回，然后回到内核。

为了实现这个功能，我设计了一个延时模块，就是在打印个人信息之后延长一段时间再做中断返回。在这里由于一个寄存器只有 16 位，所以最大可以存储 65535，如果只做这么多次访存是明显无法让个人信息显示足够长的时间的。为此我用了一个循环嵌套，用 C 表达具体的思路应该如下：

```
1 for(int i = 0; i < 65535; i++){
2     for(int j = 0; j < 10000; j++)
3 }
```

用汇编实现这个模块，如下：

```
1 delay_count equ 0
2 delay1 dw 0
3 delay2 dw 0
4 out:
5     mov ax,[delay1]
6     cmp ax,65535
7     je come_out
8
9 temp:
10    mov ax,[delay2]
11    cmp ax,10000
12    je next
13    inc word[delay2]
14    jmp temp
15
16 next:
17    mov ax,0
18    mov [delay2],ax
19    inc word[delay1]
20    jmp out
21
22 come_out:
23    mov ax,0
24    mov [delay1],ax
25    Restart ret_table
26    iret
```

而为了实现功能号的功能，我定义了应该变量 function，在调用中断之前，要求操作系统的使用者输入中断号，在这里我规定中断号从 1-4，如果键入的内容不是 1-4 则直接返回。如果输入为 1-4，

则将中断号保存到 function 中，然后在中断服务程序中将 function 取到 ah 中，然后根据 ah 的值来确定执行哪一个中断服务程序。

这一部分的关键代码如下：

```
1 int_21h:
2     Save ret_table
3     mov ah,byte[function]
4     cmp ah,01h
5     je pro1
6     cmp ah,02h
7     je pro2
8     cmp ah,03h
9     je pro3
10    cmp ah,04h
11    je pro4
12
13 call_int_21h:
14     pusha
15     call write_21h
16     print msg21h,msg21h_len,13,14
17 input:
18     mov ah, 0
19     int 16h
20     cmp al, '1'
21     jl err
22     cmp al, '4'
23     jg err
24     sub al, '0'
25     mov byte[function],al
26     int 21h
27 err:
28     call Recover_21h
29     popa
30     retf
```

通过这样的方式，我们就可以实现简单的系统调用。

#### 5.4.3 int 22h

在 int 21h 的基础上，实现 int 22h 就非常简单了。我在这里实现了在屏幕中间打印”int 22h”，然后隔一段时间返回。这里的代码与 int 21h 的代码大同小异，不再赘述。

## 5.5 取消 OUCH! 功能

在这里我把实验三未做 OUCH! 功能的用户程序代码稍作修改，匹配到最新版的代码，就完成了这部分内容。

## 5.6 对库的设计和封装

这一部分恰好是我上次实验总结的时候希望可以做到的。为了防止用户程序对内核进行操作，我封装了两个库，应该是只有内核才能调度的内核库，一个是用户程序可以使用的用户库。而这次因为只需要使用输入输出，所以我只给用户库封装了 stdio.h。

而到了这次实验之后，我们的程序变得非常复杂，所有代码文件零零散散加起来有二十多，非常凌乱，于是我在封装完库之后，对 src 中的文件做了一定的安排。

整理后的文件夹的结构如下：

```
[zhb@ubuntu:~/Desktop/src$ tree
.
├── kernel
│   ├── ckernel.c
│   └── kernel.asm
├── lib
│   ├── interrupt.h
│   ├── lib.asm
│   ├── math.h
│   ├── stdio.h
│   ├── string.h
│   ├── system.h
│   ├── time.h
│   └── wheel.asm
├── loader.asm
└── Macro.asm
Makefile
user
├── a.asm
├── b.asm
├── c.asm
└── d.asm
user_info.asm
user_lib
└── stdio.h
    └── user_lib.asm
zhbos.img

4 directories, 21 files
zhb@ubuntu:~/Desktop/src$ ]
```

图 5: 用 tree 命令查看文件夹结构

事实上我只是将之前写好的函数分门别类的放到不同的文件中，然后让 C 文件在使用对应的代码

的时候先使用 include，这也与我们平时编程是一致的。

## 5.7 用 C 编程用户程序

其实用 C 编写用户程序与编写内核的方法并没有什么不同，首先在 asm 中用 \_start 进入定位到用户程序并进入，然后在我写的 main 函数中调用我封装的用户库中的 stdio.h。依次做读取字符，打印字符，读取字符串，打印字符串的工作，结束后用 int 20h 返回内核。这与普通的用户程序并无二致，C 代码如下：

```
1 #include "../../user_lib/stdio.h"
2
3 void main(){
4     clean();
5     char hint0[80] = "Input the character:";
6     print(hint0);
7     char s = getch();
8     putchar(s);
9     putchar('\r');
10    putchar('\n');
11    char hint01[80] = "The character you input is:";
12    print(hint01);
13    putchar(s);
14    putchar('\r');
15    putchar('\n');
16    char string[MaxSize];
17    char hint1[80] = "Input the string:";
18    print(hint1);
19    get_buf(string);
20    char hint11[80] = "The string you input is:";
21    print(hint11);
22    print(string);
23    putchar('\r');
24    putchar('\n');
25    char hint12[80] = "Press any key to exit.";
26    print(hint12);
27    s = getch();
28    return;
29 }
```

而由于我在实验三中就考虑到用户程序的扩展性，所以我不需要修改内核代码，只需要修改用户程序表，给该程序在软盘和在内存中都安排合适的位置即可。

我给新的用户程序分配的内存位置是 0xAB00，然后为它在软盘上分配了四个扇区，存储在 a,b,c,d 四个用户程序之后，起名为 e。

而在编译的时候，我首先将用户程序库进行编译，然后再对用户程序进行编译，但是在链接的时

候，需要使用-Ttext 来将该用户程序重定向到给用户程序分配的内存处。

## 5.8 重写 Makefile

此次实验改变了整个项目的结构，因此原来的 Makefile 不能再使用了，需要对 Makefile 进行重构，如下：

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

KernelSrc= ./kernel/
UserSrc = ./user/
LibSrc = ./lib/
UserLibSrc = ./user_lib/
CUserSrc = ./user/e/

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) bs=512 count=1 conv=notrunc
dd if=user_info.bin of=$(IMG) bs=512 seek=1 count=1 conv=notrunc
dd if=kernel.bin of=$(IMG) bs=512 seek=2 count=34 conv=notrunc
dd if=a.com of=$(IMG) bs=512 seek=36 count=2 conv=notrunc
dd if=b.com of=$(IMG) bs=512 seek=38 count=2 conv=notrunc
dd if=c.com of=$(IMG) bs=512 seek=40 count=2 conv=notrunc
dd if=d.com of=$(IMG) bs=512 seek=42 count=2 conv=notrunc
dd if=e.com of=$(IMG) bs=512 seek=44 count=4 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 ckernel.o wheel.o
    ld -m elf_i386 -N -Ttext 0x8000 --oformat binary $^ -o $@
e.com: e.o user_lib.o ce.o
    ld -m elf_i386 -N -Ttext 0x0AB00 --oformat binary $^ -o $@

%.o : $(KernelSrc)%.asm
    nasm -f elf32 $< -o $@
%.o : $(UserLibSrc)%.asm
    nasm -f elf32 $< -o $@
%.o : $(CUserSrc)%.asm
    nasm -f elf32 $< -o $@
%.o : $(LibSrc)%.asm
    nasm -f elf32 $< -o $@
%.o : $(CUserSrc)%.c
    gcc -march=i386 -m16 -mpreferred-stack-boundary=2
    -ffreestanding -fno-PIE -masm=intel -c $< -o $@
%.o : $(KernelSrc)%.c
    gcc -march=i386 -m16 -mpreferred-stack-boundary=2
    -ffreestanding -fno-PIE -masm=intel -c $< -o $@
%.com : $(UserSrc)%.asm
    nasm $< -o $@

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

```

注意到我们引入了目录，所以得先指定文件所在的目录，然后在目标文件中寻找目标文件。

## 6 实验过程

### 6.1 确认 save 和 restart 正确运行

由于我在时间中断的中断服务程序（即上次实验实现的风火轮）处就使用了 save 和 restart 模块，所以我们只需要看到时间中断正常运行且 shell 可以正常输入即可。

```
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 # di_
2020-6-5 12:26:14
```

图 6: 进入内核后正确执行时间中断

如图，当进入系统后，风火轮和时间正确展示，且输入不会被阻塞，说明时间中断正常运行，程序并没有出现错误，可以得出：save 和 restart 模块正确工作。

## 6.2 确认 int 21h 和 int 22h 正确运行

输入 int 21h，按下回车，进入 21h 号系统调用。

```
This is int 21h, please input function number 1~4
```

图 7: 进入 21h 系统调用

输入 1，让功能号为 01h，执行 21h 号中断，如下：

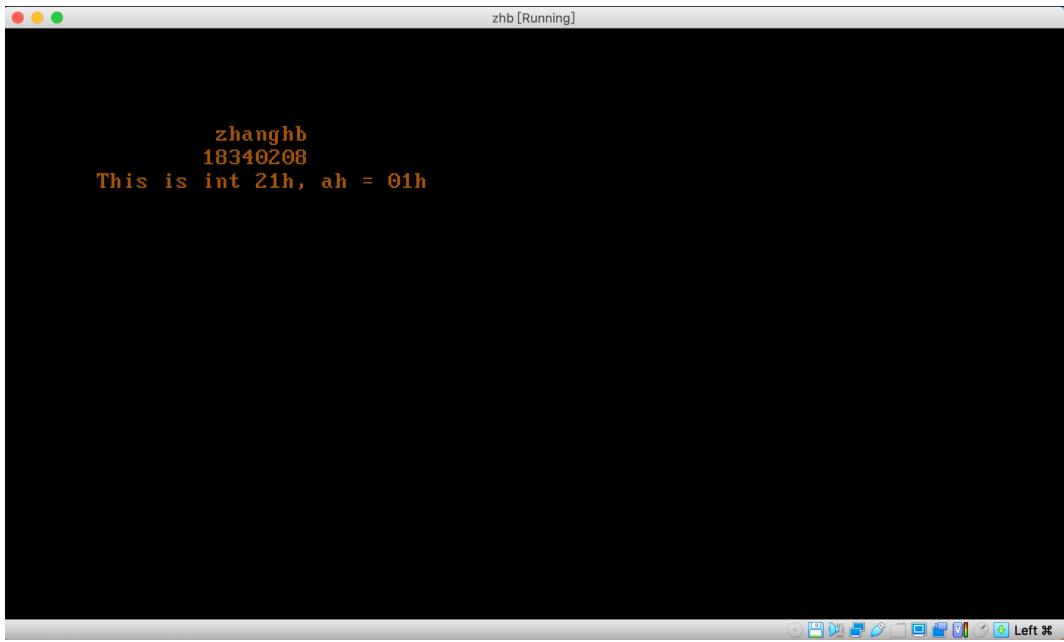


图 8：进入 21h 系统调用，ah = 01h

过了一小段时间后自动退出，再次进入 21h 中断，输入 2，让功能号为 02h，执行 21h 号中断，如下：

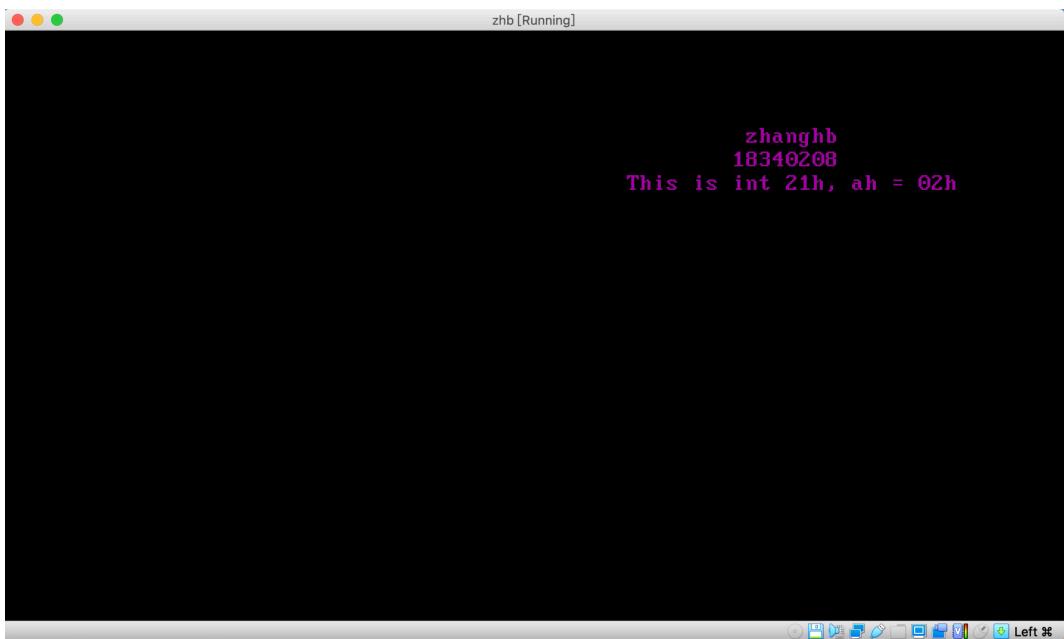


图 9：进入 21h 系统调用，ah = 02h

过了一小段时间后自动退出，再次进入 21h 中断，输入 3，让功能号为 03h，执行 21h 号中断，如

下：

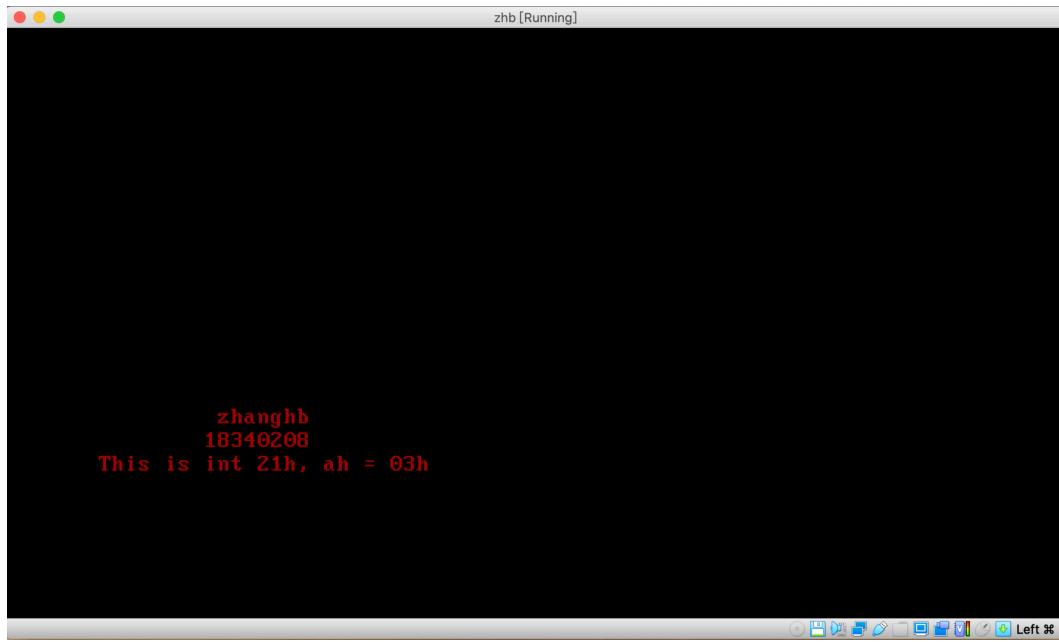


图 10: 进入 21h 系统调用，ah = 03h

过了一小段时间后自动退出，再次进入 21h 中断，输入 4，让功能号为 04h，执行 21h 号中断，如下：

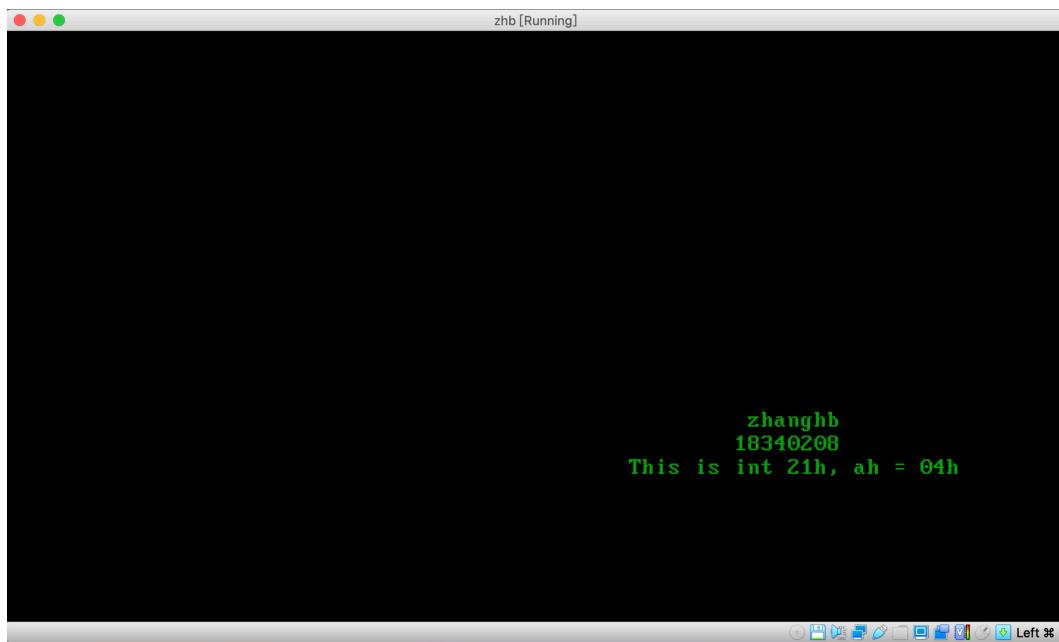


图 11: 进入 21h 系统调用，ah = 04h

然后输入 int 22h，打印 int 22h，如下：

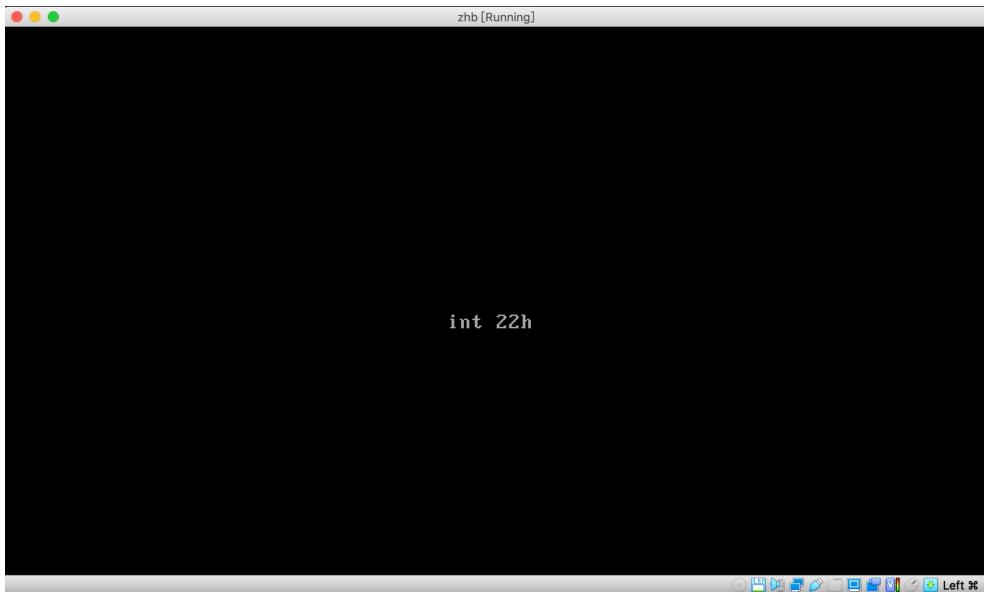


图 12: 打印 int 22h

### 6.3 对 int 20h 返回程序的测试

我将用户程序中的 retf 全都改成了 int 20h，然后执行用户程序，与之前的效果是相同的，具体情况详见附件中的视频。

### 6.4 测试用 C 编写的用户程序

在 shell 中输入 display 5，可以看到：

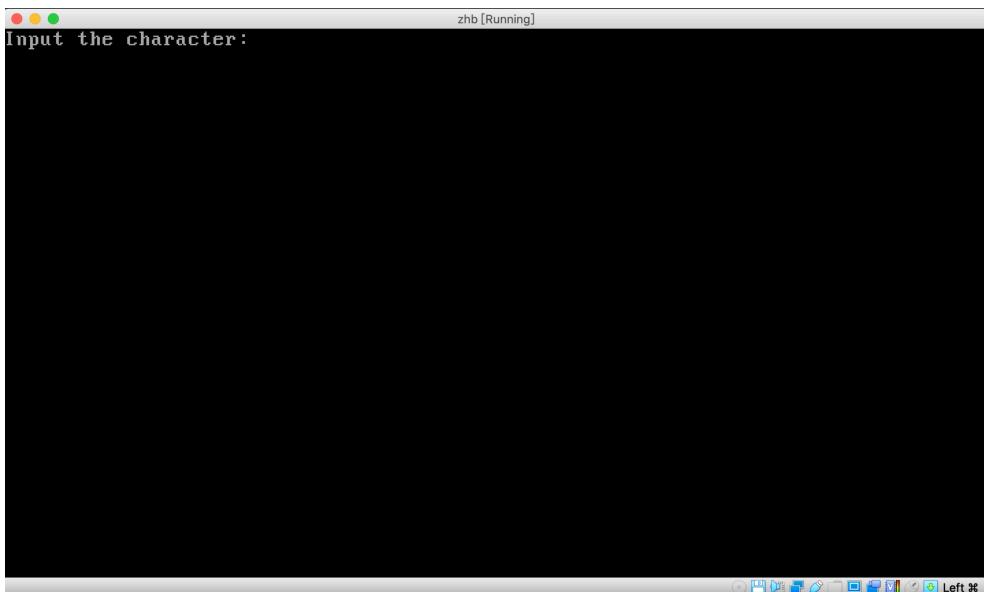


图 13: 执行 C 编写的用户程序

按照提示输入字符，接着用户程序显示处我输入的字符，并提示我输入字符串：

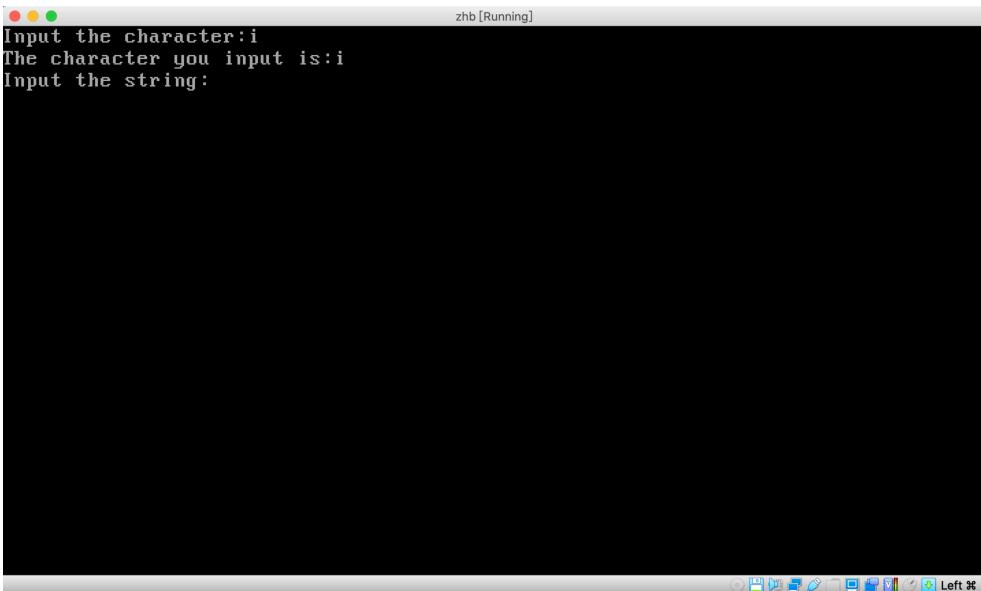


图 14: 执行 C 编写的用户程序

输入字符串后，按回车，如下：

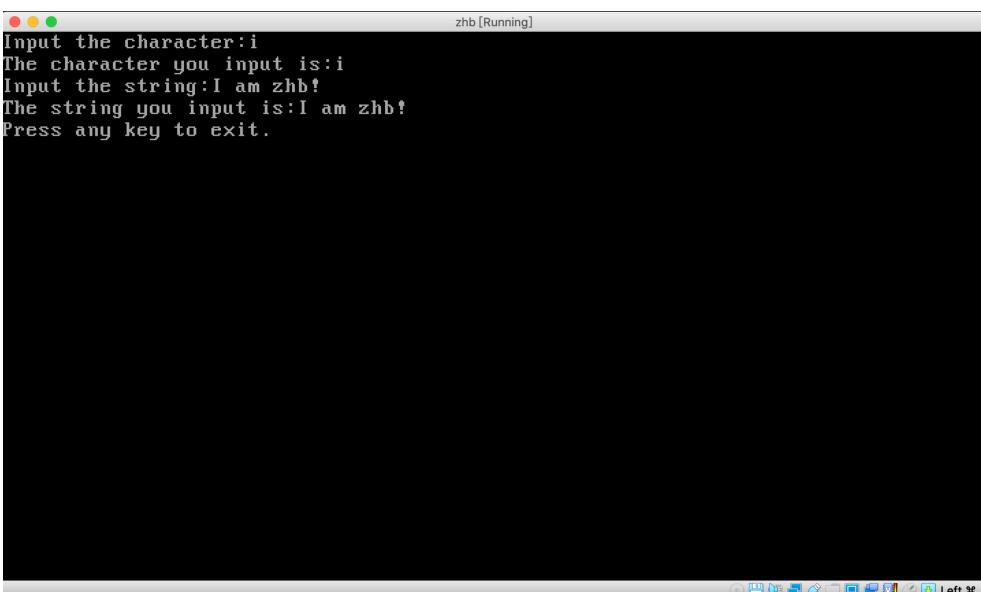


图 15: 执行 C 编写的用户程序

根据提示按下键盘，退出用户程序，回到 shell。

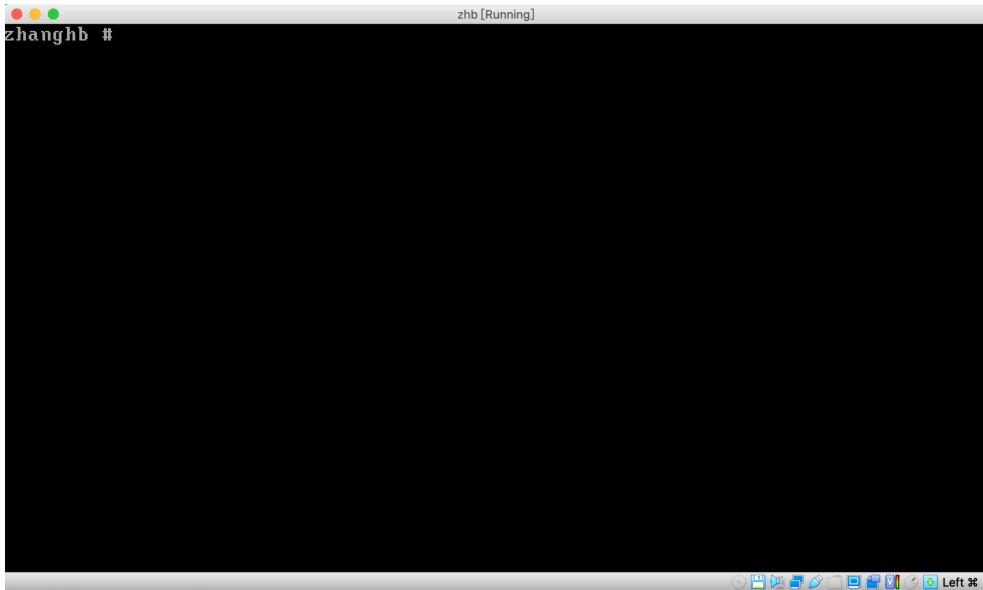


图 16: 返回 shell

最后输入 poweroff，关机，结束实验。

## 7 实验总结

### 7.1 遇到的主要障碍

这次实验虽然难度不大，但是非常麻烦，具体体现在以下几个方面：

- 栈的处理精度：由于对栈的精度要求非常高，稍有不慎就会跳转到奇奇怪怪的地方，所以我们需要非常小心地对栈进行操作。在这里我为了找到对栈的错误操作专门下载并使用了 bochs 模拟器。
- 链接时正确匹配重定向的位置和用户程序加载到内存中的位置，稍有不慎则会出现用户程序无法正确读取字符串，出现乱码。
- 封装库的时候因为内核使用的库之间存在相互依赖的关系，所以我花了很长时间将它们分离。

### 7.2 软盘结构的调整

在此次实验的过程中，我新增加了用户程序，因此对软盘结构进行了调整。

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
C 语言用户程序	1	0	9	4	输入和输出字符与字符串	0AB00h

### 7.3 心得体会

封装了用户库后，我理解了为什么要使用系统调用，如果不提供接口给用户的话，那用户使用内核中的函数就需要与内核一同编译，这样子的话每次增加一个用户程序就需要重新编译内核，这是非常不合理的事情。

而通过这次实验，我也大概理解了 PCB 表的基本原理，也为下次实验打下了基础。