

操作系统实验报告

18 级计科超算 18340208 张洪宾

1 实验题目

C 与汇编开发独立批处理的内核

2 实验目的

- 加深理解操作系统内核概念
- 了解操作系统开发方法
- 掌握汇编语言与高级语言混合编程的方法
- 掌握独立内核的设计与加载方法
- 加强磁盘空间管理工作

3 实验要求

- 知道独立内核设计的需求
- 掌握一种 x86 汇编语言与一种 C 高级语言混合编程的规定和要求
- 设计一个程序，以汇编程序为主入口模块，调用一个 C 语言编写的函数处理汇编模块定义的数据，然后再由汇编模块完成屏幕输出数据，将程序生成 COM 格式程序，在 DOS 或虚拟环境运行。
- 汇编语言与高级语言混合编程的方法，重写和扩展实验二的的监控程序，从引导程序分离独立，生成一个 COM 格式程序的独立内核。
- 再设计新的引导程序，实现独立内核的加载引导，确保内核功能不比实验二的监控程序弱，展示原有功能或加强功能可以工作。
- 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

- 寻找或认识一套匹配的汇编与 C 编译器组合。利用 C 编译器，将一个样板 C 程序进行编译，获得符号表文件，分析全局变量、局部变量、变量初始化、函数调用、参数传递情况，确定一种匹配的汇编语言工具，在实验报告中描述这些工作。
- 写一个汇编和 C 程序混合编程实例，展示你所用的这套组合环境的使用。汇编模块中定义一个字符串，调用 C 语言的函数，统计其中某个字符出现的次数（函数返回），汇编模块显示统计结果。执行程序可以在 Dos 中运行。
- 重写实验二程序，实验二的的监控程序从引导程序分离独立，生成一个 COM 格式程序的独立内核，在 1.44MB 软盘映像中，保存到特定的几个扇区。利用汇编和 c 程序混合编程监控程序命令保留原有程序功能，如可以按操作选择，执行一个或几个用户程序、加载用户程序和返回监控程序；执行完一个用户程序后，可以执行下一个。
- 利用汇编和 c 程序混合编程的优势，多用 c 语言扩展监控程序命令处理能力。
- 重写引导程序，加载 COM 格式程序的独立内核。
- 拓展自己的软件项目管理目录，管理实验项目相关文档。

- 在老师要求的基础上，我在我的内核中添加了获取当前时间的功能，关机功能和重启功能。

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 2.11.08
- 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 编译工具链的选择

我本来是想要继续使用 Mac 作为实验环境，然而写了一个简单的汇编程序 hello world，发现无法链接。

出现了这样的报错：

```
building for macOS-x86_64 but attempting to link with file built for unknown-unsupported file format
```

上网查了资料，发现 macOS 在 2018 年就停止了对 x86 的 32 位应用的支持，而我计划做的操作系统的是 32 位的实模式的内核，所以我只能切换开发机器的操作系统。非常自然的，我切换到了与 macOS 同样基于 Unix 的 Linux 系统。

汇编语言编译工具我继续使用 NASM；C 语言编译工具我选择了 gcc，这是一个成熟的 GNU 编译器，也是我平时使用较多的编译工具。而链接工具我使用了 Linux 自带的 ld 工具。

而查看符号表我选择了 Linux 自带的 nm 命令，这一部分会在实验过程中体现。

5.3 混合编程实例

由于 Linux 上不支持对 x86 的 32 位实模式程序的运行，所以我决定结合我实验一和实验二写的引导扇区，将这段混合编程实例的程序放在虚拟机中运行。

在这段程序中，为了解 C 和汇编的相互调用，我既在 C 中调用汇编的函数，也在汇编中调用 C 的函数。

首先，我用汇编实现了输出一个字符的 putchar 函数和清屏的 clean 函数。在这里有两个要点：

- 首先是 putchar 的参数传递，C 语言和汇编混合编译的时候使用栈传递参数的，而为了保存当前的寄存器状态，一般要先压栈，因此需要通过栈顶指针再做偏移才能找到 C 语言起始参数的地址。关于获取传递的参数方法，在后面的实现中也是要时常用到，我会在总结中详细讲解。
- 从 C 语言调用汇编函数，需要用 ret 返回。

putchar 和 clean 的实现借助了中断调用，代码如下：

```
1 BITS 16
2 [global clean]
3 [global putchar]
4 ; 输出一个字符
5 putchar:                                ; 函数：在光标处打印一个字符
6     push bp
7     push ax
8     push bx
9     mov bp, sp
```

```

10      add bp, 6+4          ; 参数地址
11      mov al, [bp]        ; al=要打印的字符
12      mov ah, 0Eh        ; 功能号： 打印一个字符
13      mov bh, 0          ; bh=页码
14      int 10h            ; 打印字符
15      pop bx
16      pop ax
17      pop bp
18      ret
19
20 ; 清屏函数
21 clean:
22      push ax
23      mov ax, 0003h
24      int 10h
25      pop ax
26      ret

```

在文件中加入 global 来使得 C 语言可以对这些函数进行调用。

然后我在对应的.c 文件中加入两行代码：

```

1      extern void clean();
2      extern void putchar(unsigned char c);

```

这样子就可以在 C 语言中调用刚刚写的两个函数了。

而在 C 文件中我实现了计算汇编代码传来的字符串中字符 ‘a’ 的数量的统计，该函数为 fun，并输出相应的提示信息，返回 ‘a’ 的数量。代码如下：

```

1      int count(char*target){
2          int count = 0;
3          int total = 0;
4          while(target[count] != 0){
5              count++;
6              if(target[count] == 'a')total++;
7          }
8          return total;
9      }
10     int strlen(char*target){
11         int count = 0;
12         while(target[count] != 0){
13             count++;
14         }
15         return count;
16     }
17     void print(char* str) {
18         for(int i = 0; i < strlen(str); i++) {

```

```

19         putchar(str[i]);
20     }
21 }
22
23 int fun(char*s){
24     clean();
25     int len = count(s);
26     char*hint1 = "The string \n";
27     char*hint2 = "\n has ";
28     print(hint1);
29     print(s);
30     print(hint2);
31     return len;
32 }

```

然后最后实现 main.asm。用 [extern fun] 来在汇编中使用 fun 函数。在这里也有两个问题：

- 参数的传递用压栈的形式，对参数列表从右往左压栈。
- 返回值使用 ax 寄存器传递回汇编代码。

在这里为给定了一个字符串 “I am zhanghb”，在汇编代码中调用 fun 来查看结果，并结合在 fun 函数中输出的提示信息，补充其他输出内容。main.asm 代码如下：

```

1  BITS 16
2  [extern fun]
3  global _start
4  _start:
5      push msg
6      call dword fun ; 进入命令行界面
7      add ax, '0'
8      mov ah, 0Eh ; 功能号：打印一个字符
9      mov bh, 0 ; bh=页码
10     int 10h ; 打印字符
11     mov al, ' '
12     mov ah, 0Eh ; 功能号：打印一个字符
13     mov bh, 0 ; bh=页码
14     int 10h ; 打印字符
15     mov al, 'a'
16     mov ah, 0Eh ; 功能号：打印一个字符
17     mov bh, 0 ; bh=页码
18     int 10h ; 打印字符
19     mov al, '!'
20     mov ah, 0Eh ; 功能号：打印一个字符
21     mov bh, 0 ; bh=页码
22     int 10h ; 打印字符
23     jmp $

```

24

25 msg db 'I am zhanghb',0

因为文件比较少，命令比较简单，我简单写了一个 sh 批处理脚本 complier.sh 来对程序进行编译。不过编译参数也是一个比较大的问题，经过很长时间的摸索，阅读官方文档，上网查资料，我终于确定了如下的编译参数。并且结合我的引导程序，我将初始地址重定向为 0x8000。

```
#!/bin/bash
rm *.img
nasm loader.asm -o loader.bin
nasm -f elf32 main.asm -o main.o
nasm -f elf32 count.asm -o count.o
gcc -c -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -fno-PIE -masm=intel -c count.c
-o c_count.o
ld -m elf_i386 -N -Ttext 0x8000 -oformat binary main.o count.o c_count.o -o program.bin
/sbin/mkfs.msfdos -C example.img 1440
dd if=loader.bin of=example.img bs=512 conv=notrunc
dd if=program.bin of=example.img bs=512 seek=1 conv=notrunc
rm *.o
rm *.bin
echo "[+] Done."
```

运行 sh complier.sh 即可获得 example.img。

5.4 设计出软盘的结构

软盘的结构对引导程序读扇区有着比较大的影响，所以设计出合理的软盘结构尤为关键。

在实际设计的过程中发现，在 3.5 英寸软盘中，一个磁头下面只有 18 个扇区，而扇区号从 1 开始，所以第 19 个扇区的磁头号为 1，扇区号为 1。

为了保证内核的可扩展性，我给内核分配了足够大的空间。结合上次的经验，我构造出了软盘的结构，如下：

| 程序 | 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 | 1 | 1 | 打印提示信息 并加载用户信息表和内核 | 07C00h |
| 用户程序表 | 0 | 2 | 2 | 存储用户程序存储的位置与 程序加载到内存中的位置 | 7E00h |
| 内核 | 0 | 3 | 15 | 常驻内存，实现 OS 最基本的功能 | 8000h |
| 用户程序 1 | 1 | 1 | 1 | 在屏幕左上角使个人信息跳动 | 0A300h |
| 用户程序 1 | 1 | 2 | 1 | 在屏幕右上角使个人信息跳动 | 0A500h |
| 用户程序 3 | 1 | 3 | 1 | 在屏幕左下角使个人信息跳动 | 0A700h |
| 用户程序 4 | 1 | 4 | 1 | 在屏幕右下角使个人信息跳动 | 0A900h |

用户程序我选择了之前的程序，于是需要改写引导程序，实现用户程序表，以及实现内核程序。

5.5 改写引导程序

上次实验中我将引导程序和监控程序结合在一起，在总结中我也提到这是非常不科学的做法，于是在这次实验中我将引导程序单独分出来。

在引导程序中，我打印了提示信息，然后将用户程序表加载到了内存的指定位置，然后再将内核加载到内存中的指定位置，并跳转到内核去。

这一部分代码与前面的基本一致，关键代码如下：

```

1 BITS 16
2 offset_user_info equ 7E00h; 用户程序表在内存中位置
3 offset_kernel equ 8000h; 内核在内存中的位置
4 Load_user_pro_info:
5     mov cl, 2
6     mov ax, cs
7     mov es, ax
8     mov ah, 2
9     mov al, 1
10    mov dl, 0
11    mov dh, 0
12    mov ch, 0
13    mov bx, offset_user_info
14    int 13h
15 Load_kernel:
16    mov cl, 4
17    mov ax, cs
18    mov es, ax
19    mov ah, 2
20    mov al, 16
21    mov dl, 0

```

```

22     mov dh, 0
23     mov ch, 0
24     mov bx, offset__kernel
25     int 13h
26     jmp offset__kernel
27     jmp $

```

5.6 设计用户程序表

本来我想在这里使用 FAT12 文件结构的，但是考虑到我对汇编还不够熟悉，我决定先用最简单的文件结构。

由于现在我只有四个用户程序，而对于一个程序来说，有几个指标比较重要：

- 程序在软盘中的位置，包括柱面号、磁头号、扇区号。
- 程序的大小。在这里所有的用户程序都是 512 字节
- 程序加载到内存中的位置。

再结合程序的名称，我定义如下结构体：

```

1  struct user_info
2      .name:      resb    16
3      .size:      resw    1
4      .cylinder:  resb    1
5      .head:      resb    1
6      .sector:    resb    1
7      .addr:      resw    1
8  endstruct

```

该结构体的信息依次是程序的名称、程序的大小、程序的柱面号、程序的磁头号、程序的扇区号、程序在内存中的地址。

虽然这次只有四个程序，但是为了便于内核的扩展，我用一个变量来存储用户程序的数量

```
count_of_program db 4
```

然后结合我设计的软盘的结构，我就可以写出完整的用户程序表了。

```

1  user1:
2  istruc user_info
3      at user_info.name,      db          'a'
4      at user_info.size,      dw          512
5      at user_info.cylinder,  db          0
6      at user_info.head,      db          1

```



```

7      at  user_info.sector ,      db      1
8      at  user_info.addr ,      dw      offset_user1
9 iend
10
11 user2:
12 istruc user_info
13     at  user_info.name ,      db      'b'
14     at  user_info.size ,      dw      512
15     at  user_info.cylinder ,  db      0
16     at  user_info.head ,     db      1
17     at  user_info.sector ,    db      2
18     at  user_info.addr ,     dw      offset_user2
19 iend
20
21 user3:
22 istruc user_info
23     at  user_info.name ,      db      'c'
24     at  user_info.size ,      dw      512
25     at  user_info.cylinder ,  db      0
26     at  user_info.head ,     db      1
27     at  user_info.sector ,    db      3
28     at  user_info.addr ,     dw      offset_user3
29 iend
30
31 user4:
32 istruc user_info
33     at  user_info.name ,      db      'd'
34     at  user_info.size ,      dw      512
35     at  user_info.cylinder ,  db      0
36     at  user_info.head ,     db      1
37     at  user_info.sector ,    db      4
38     at  user_info.addr ,     dw      offset_user4
39 iend

```

于是我们就可以通过访问用户程序表来获取每一个用户程序的地址。

5.7 内核的设计

这是本次实验最复杂也是最重要的部分，我的目标是做一个简单的 shell 来输入命令，并执行程序。

在混合编程实例中的 putchar 和 clean 可以进行复用，剩下的部分设计如下：

5.7.1 输入输出的实现

我的想法是通过利用中断调用实现最简单的读和写字符的函数，然后对这些函数进行封装，得到更加完善的输入输出函数。

putchar 的实现在上面已经提到了，不再赘述。

关于输入，我实现了类似于 Windows 的 conio.h 库的 getch() 无回显输入，这样子可以让后面读取键盘字符是可以对删除和回车等输入做出反应，代码如下：

```
1 getch:
2     mov ah, 0
3     int 16h
4     mov ah, 0
5     ret
```

受上次使用的宏的启发，我还实现了一个在指定位置打印指定字符串的函数：

```
1 print_at_point:
2     pusha
3     mov si, sp
4     add si, 16+4 ; 找到首个参数
5     mov ax, cs
6     mov ds, ax
7     mov bp, [si]
8     mov ax, ds
9     mov es, ax
10    mov cx, [si+4]
11    mov ah, 13h
12    mov al, 01h
13    mov bh, 00h
14    mov bl, 07h
15    mov dh, [si+8]
16    mov dl, [si+12]
17    int 10h
18    popa
19    ret
```

这就是最基本的输入输出的底层实现了，然后在用 C 对这些函数进行封装：

首先封装 putchar 得到一个打印字符串变量的函数：

```
1 void print(char* str) {
2     for(int i = 0; i < strlen(str); i++) {
3         putchar(str[i]);
4     }
5 }
```

通过该函数即可将字符串输出到当前光标位置处。

然后为了实现在命令行中输入命令，并可以用 Backspace 进行删除，用 Enter 进行结束命令输入并执行命令，我封装了一个函数来实现对命令流的获取。首先封装 putchar 得到一个打印字符串变量的函数：

```

1 void get_buf(char*res){
2     int i = 0;
3     while(1){
4         if(i < 0)i = 0;
5         char c = getch();
6         if(!(c == 0xD || c == '\b' || c >= 32 && c <= 127 || c == '\n'))continue;
7         if(i < MaxSize - 1){
8             if(c == 0x0D)break;
9             else if(c == '\b'){
10                 putchar('\b');
11                 putchar(' ');
12                 putchar('\b');
13                 i--;
14             }
15             else{
16                 putchar(c);
17                 res[i++] = c;
18             }
19         }
20         // 达到了最大值了
21         else{
22             if(c == 0x0D)break;
23             else if(c == '\b'){
24                 putchar('\b');
25                 putchar(' ');
26                 putchar('\b');
27                 i--;
28             }
29         }
30     }
31     res[i] = 0;
32     putchar('\r');
33     putchar('\n');
34 }

```

这里要注意的细节是：到达最大长度后就不能再输入字符了，在这里的 `MaxSize = 20`，足够满足一般的命令行了。

至此基本的输入输出函数已经完成。

5.7.2 获取用户程序的信息

要加载用户程序的信息，则需要先将获取用户程序的总数量，然后比较目标用户程序编号和用户程序的总数，如果目标用户程序编号超过了用户程序的总数显然是无效输入。

如果要在软盘中读取用户程序，我们需要获取柱面号、磁头号和扇区号来定位程序所在位置。

而根据我定义的结构体，一个用户程序表项的大小为 $16 + 2 + 1 + 1 + 1 + 2 = 23$ 字节，而在用户程序表项内部，用户程序名所在位置的偏移量是 0，用户程序大小所在位置的偏移量是 16，用户程序的柱面号所在位置的偏移量是 18，用户程序的磁头号所在位置的偏移量是 19，用户程序的扇区号所在位置的偏移量是 20，用户程序在内存中的地址所在位置的偏移量是 21。

而事实上，获取用户程序信息 6 项信息的方法基本相同，只有获取用户程序名的时候返回的是字符串地址，获取用户程序结构体中其他信息的时候返回的是数值。在这里为了不赘述，只取获取用户程序的总数、获取用户程序名称和获取用户程序大小三个函数进行示例：

获取用户程序总数的 C 接口为：

```
char get_number_of_program();
```

实现方式为：

```
1 get_number_of_program:
2     mov al, [offset_user_info]
3     ret
```

可以看出，在内存中加载用户程序信息的位置处的第一个信息应该是用户程序总数，所以将其放进 al 寄存器并返回。

而获取指定用户程序的名称的 C 语言接口为：

```
unsigned char* get_name(unsigned short id);
```

其汇编实现方式为：

```
1 get_name:
2     push bp
3     push bx
4     mov bp, sp
5     add bp, 4+4
6     mov al, [bp]
7     add al, -1
8     mov bl, 23
9     mul bl
10    add ax, 1
11    add ax, offset_user_info
12    pop bx
13    pop bp
14    ret
```

可以看到，这里是首先将程序的 id 减去 1，乘以 23，再加上 1（用户程序总数占据了一个字节），得到在用户程序表中目标用户程序的基地址，由于用户程序名的偏移量为 0，所以加上用户程序信息表的地址后直接返回该地址。

与此略有不同的是对用户程序其他信息的获取，因为获取其他信息的返回值应该是数值而不是地址，所以应该用 [] 进行数值的获取：

以获取用户程序的大小的函数为例，其 C 语言接口如下：

```
short get_size(short order);
```

```
1  get_size:
2      push bx
3      push bp
4      mov bp, sp
5      add bp, 4+4
6      mov al, [bp]
7      add al, -1
8      mov bl, 23
9      mul bl
10     add ax, 16
11     add ax, 1
12     mov bx, ax
13     add bx, offset_user_info
14     mov ax, [bx]
15     pop bp
16     pop bx
17     ret
```

可以看到，获取指定用户程序大小的函数的方法跟获取用户程序姓名的方法类似，首先得到用户程序表项的基地址，然后加上该信息在表项中的偏移量（在这里是 16），就得到了存储用户程序大小的地址，然后用 [] 获取该地址中的值，存储到 ax，返回。

而剩下的函数与 get_size 实现的方式类似，不再赘述。

5.7.3 实现运行指定用户程序

当我们获取了用户程序信息后，我们就可以把它加载到内存中来运行了。但是这其实也是一个非常麻烦的事情，我们需要将之前获取的用户程序在内存中的地址、柱面号、磁头号、扇区号、用户程序的大小传递给该函数，然后用 BIOS 的 13h 号调用将程序加载到内存中进行运行。

然后还有一个问题，如果要想实现批处理执行，我们就不能简单的在用户程序用 jmp 指令，因为根本就无法定位 C 程序执行到哪里了。如果直接 jmp 回操作系统，则无法使用批处理功能。在这里我想了很久，参考了很多资料，最终找到了解决方法。关于这一部分详细的解决方法我会在实验总结提到。

运行指定用户程序的函数 C 语言接口为：

```
void Run(short addr,char cylinder,char head,char sector,short len);
```

```
1 Run:
2     pusha
3     mov bp,sp
4     add bp,4+16
5     mov ax,cs
6     mov es,ax
7     mov bx,[bp]; 加载到内存中的地址
8     mov ah,2
9     mov dl,0
10    mov ch,[bp + 4]; 柱面号
11    mov dh,[bp + 8]; 磁头号
12    mov cl,[bp + 12]; 扇区号
13    mov al,[bp + 16]; 扇区数(程序大小)
14    int 13H
15    call dword pushCsIp
16 pushCsIp:
17     mov si, sp           ; si 指向栈顶
18     mov word[si], return
19     jmp [bp]
20 return:
21     popa
22     ret
```

然后就可以用该函数运行指定的用户程序了。

5.7.4 获取系统时间

考虑到现在的操作系统一般都会加上显示时间功能，我决定给我的操作系统也加上查询时间的功能。

在这里就要涉及对芯片的访问了。在 PC 机中，有一个 CMOS RAM 芯片。此芯片中存放着当前的时间：年、月、日、时、分、秒。六个信息的长度都为一个字节，存放单元为：

- 秒：0
- 分：2
- 时：4
- 日：7

- 月：8
- 年：9

这些数据以 BCD 码存放。要读取 CMOS 信息，首先向 70h 端口写入要访问的单元地址，然后从 71h 端口读入数据。我在这里参考王爽的《汇编语言》，写了一个函数，提取指定位数的时间信息的 BCD 码。

该函数的 C 语言接口如下：

```
unsigned char getTime(unsigned char bit)
```

其汇编实现如下：

```
1 getTime:
2     push bp
3     push sp
4     mov bp, sp
5     add bp, 4+4
6     mov al, [bp]
7     out 70h, al
8     in al, 71h
9     mov ah, 0
10    pop sp
11    pop bp
12    ret
```

用该函数可以获得 CMOS RAM 中第 bit 位的 BCD 码，然后通过转化就可以获当前时间了。

将 BCD 码转化为数值的函数如下：

```
1 unsigned char bcd_to_num(unsigned char bcd)
2 {
3     return ((bcd & 0xF0) >> 4) * 10 + (bcd & 0x0F);
4 }
```

然后就可以获取系统的时间了：

```
1 unsigned char year = getTime(9);
2 unsigned char month = getTime(8);
3 unsigned char day = getTime(7);
4 unsigned char hour = getTime(4);
5 unsigned char min = getTime(2);
6 unsigned char sec = getTime(0);
```

就这样我们得到了精确到秒的系统时间。

5.7.5 实现关机功能

在INT 15H 5307H: Set Power State中我发现了 15H 号 BIOS 调用可以实现关机功能，于是根据上面的代码实现了关机函数，其 C 语言接口为：

```
void Poweroff();
```

汇编实现：

```
1 Poweroff:
2     mov ax,5307H ;高级电源管理功能,设置电源状态
3     mov bx,0001H ;设备ID,1:所有设备
4     mov cx,0003H ;状态,3:表示关机
5     int 15H
6     jmp $ ;loop at current postion
```

5.7.6 实现重启功能

可以通过向 64h 端口写入 0FEh 实现重启功能。我实现了该函数，其 C 语言接口为：

```
void Reboot();
```

该函数的汇编实现为：

```
1 Reboot:
2     mov al, 0FEh
3     out 64h, al
4     ret
```

5.7.7 对 shell 的封装

在封装之前，我还做了一些其他工作，比如输出提示信息，将数字转化为指定进制形式的字符串，将数字字符串转化为数字的函数等，在这里就不做赘述。

完成了上述工作，我们就可以来做 shell 了，其实 shell 的实现是对上述函数的封装，所以我用了—个 while(1)，然后用 get_buf 函数获取命令和其参数，然后再对整个字符流进行分析。

我先从字符流中获取命令，然后根据命令对类型决定接下来的操作。然后我实现了 help、ls、clear、display、time、poweroff、reboot 几条命令，它们的意义分别是打印提示信息、展示所有用户程序的信息、清空屏幕、运行一个或多个程序、获取当前时间、关机、重启。

以上命令都是使用了之前写好的函数，所以并没有什么特别值得讲的，不过有一点要注意，如果 display 后面的用户程序编号参数超过了用户程序总数，则需要做特殊处理，否则 run 函数会定位到无效区域，这样子使得 OS 崩溃。

5.8 总结

为了更好地说明整个 OS 如何运行，我做了一个流程图，展示 OS 的操作：

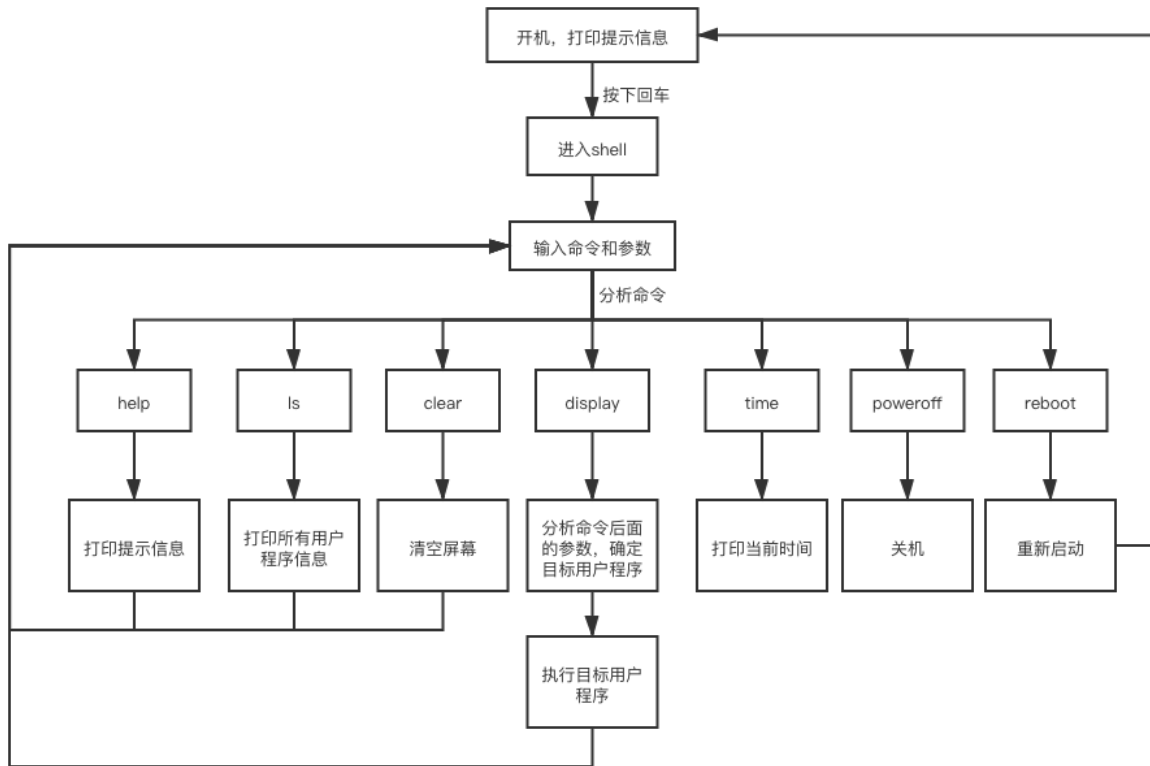


图 1: OS 的操作

6 实验过程

6.1 运行混合编程实例并分析符号表

6.1.1 编译运行

进入混合编程实例所在文件夹，输入 sh compiler.sh 命令，完成编译。

6.1.2 用 nm 获得符号表

在 Linux 终端中输入 nm *.o, 获得符号表结果如下：

```
zhh — zhh@ubuntu: ~/Desktop/example — ssh 10.37.129.3 — 80x24
26 bytes copied, 0.000117979 s, 220 kB/s
1+1 records in
1+1 records out
600 bytes copied, 9.9588e-05 s, 6.0 MB/s
[+] Done.
zhh@ubuntu:~/Desktop/example$ nm *.o

c_count.o:
          U clean
00000000 T count
000000e2 T fun
0000008e T print
          U putchar
00000058 T strlen

count.o:
00000015 T clean
00000000 T putchar

main.o:
          U fun
0000002c t msg
00000000 T _start
zhh@ubuntu:~/Desktop/example$
```

图 2: 获取符号表

在 `c_count` 的源文件 `count.c` 中使用了外部的 `clean` 函数和 `putchar` 函数 (图中的 U 类型), 而定义了 `count` 函数, `print` 函数, `strlen` 函数和 `fun` 函数 (图中 T 类型)。

而在 `count.o` 的源文件 `count.asm` 中则定义了 `clean` 函数和 `putchar` 函数 (图中 T 类型)。

在 `main.o` 的源文件 `main.asm` 中则定义了 `msg` 整个全局变量 (图中 t 类型), 使用了外部函数 `fun` (图中 U 类型), 定义了函数 `_start` (图中 T 类型)。

由此可见, 我们可以用 `nm` 工具读取 elf 的 symbol 信息, 获取符号表。

6.1.3 运行混合编程实例

将生成的 `example.img` 装载到虚拟机, 运行, 得到如下结果:

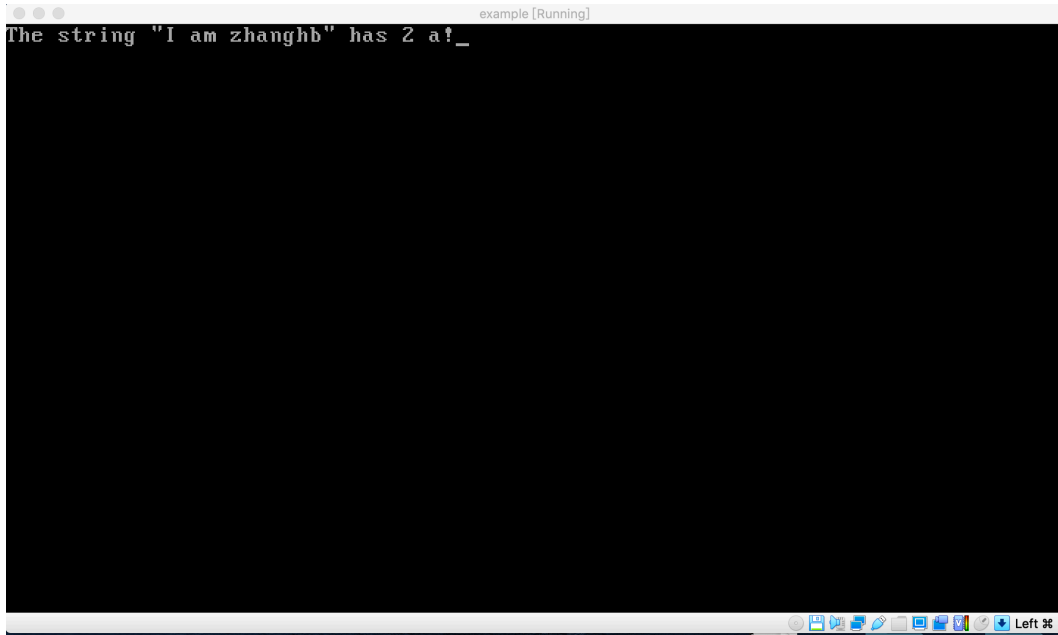


图 3: 混合编程实例的运行结果

可以看到，屏幕中打印的信息显示，我指定的字符串：“I am zhanghb”中有两个字母’a’。这与事实相符，说明这个程序正确运行。

6.2 运行带有独立批处理内核的操作系统

6.2.1 编译

这次我重新写了一个 makefile:

```
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=3 conv=notrunc
dd if=a.com of=$(IMG) seek=18 conv=notrunc
dd if=b.com of=$(IMG) seek=19 conv=notrunc
dd if=c.com of=$(IMG) seek=20 conv=notrunc
dd if=d.com of=$(IMG) seek=21 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
    ld -m elf_i386 -N -Ttext 0x8000 --oformat binary $^ -o $@

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

%.o : %.c
    gcc -march=i386 -ml6 -mpreferred-stack-boundary=2 -ffreestanding
    -fno-PIE -masm=intel -c $< -o $@
%.com : %.asm
    nasm $< -o $@

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

```

在 Linux 终端下输入 make, 得到了 zhbos.img。

6.2.2 运行 zhbos.img

将 zhbos.img 装载到虚拟机, 运行:

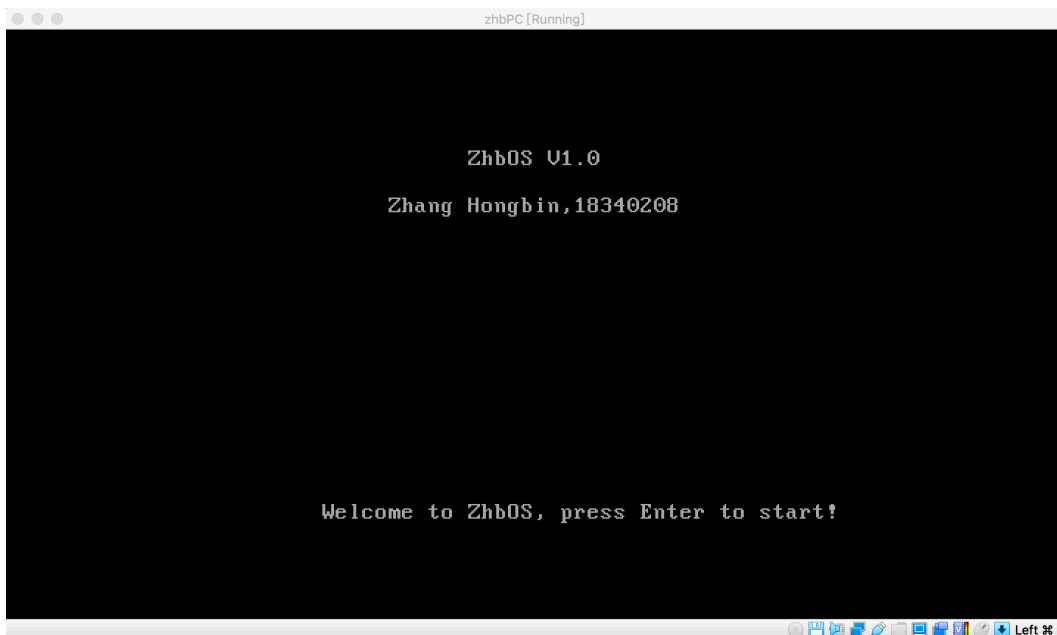


图 4: 初始化界面

按下了 Enter 键后, 进入了 shell:

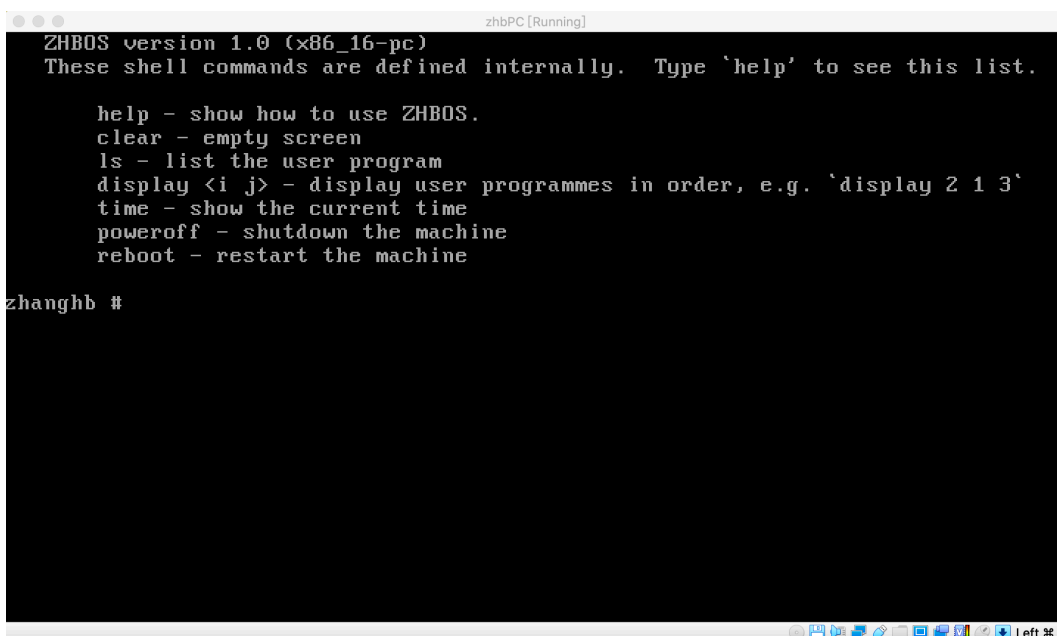


图 5: 进入了 shell

可以看到打印的提示信息。根据提示信息, 输入 ls, 显示用户程序表:

```
zhhbPC [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

zhanghb # ls
There are 4 user programs.
  Name      Size      Address      Cylinder      Head      Sector
  a         512      0xA300H         0           1           1
  b         512      0xA500H         0           1           2
  c         512      0xA700H         0           1           3
  d         512      0xA900H         0           1           4
zhanghb # _
```

图 6: 用 ls 查看用户程序表

可以看到所有用户程序的程序名，程序大小，加载到内存的地址，在软盘中的柱面号、磁头号和扇区号。

然后输入 display 1 运行第一个用户程序：

```
zhhbPC [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

zhanghb # ls
There are 4 user programs.
  Name      Size      Address      Cylinder      Head      Sector
  a         512      0xA300H         0           1           1
  b         512      0xA500H         0           1           2
  c         512      0xA700H         0           1           3
  d         512      0xA900H         0           1           4
zhanghb # display 1
```

图 7: 输入 display 1

然后按下回车键运行程序 1:

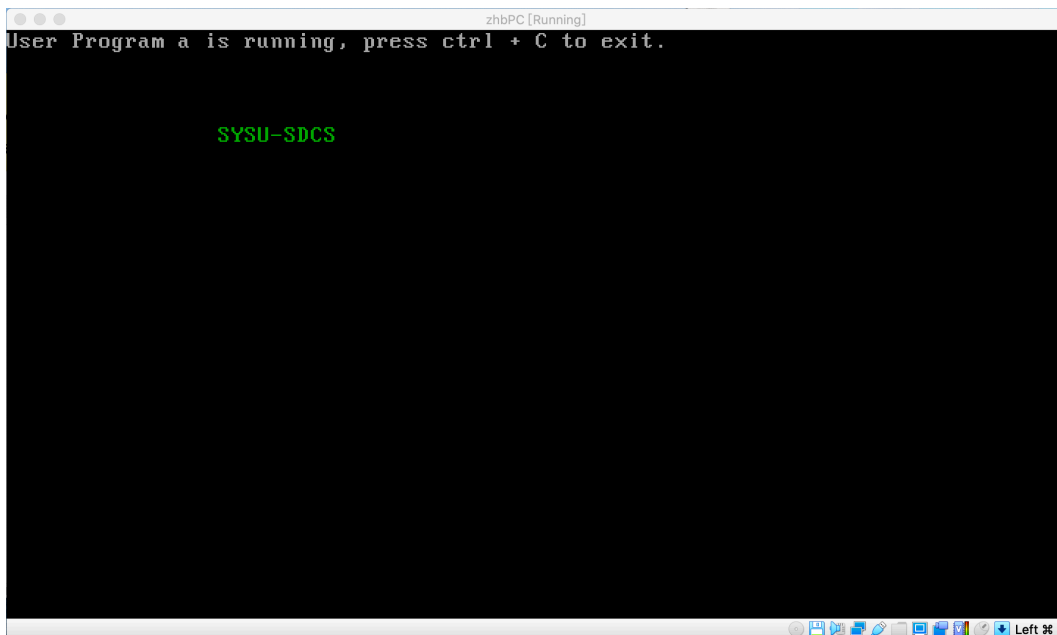


图 8: 用 display 运行第一个程序

然后按下 Ctrl + C 回到 shell:

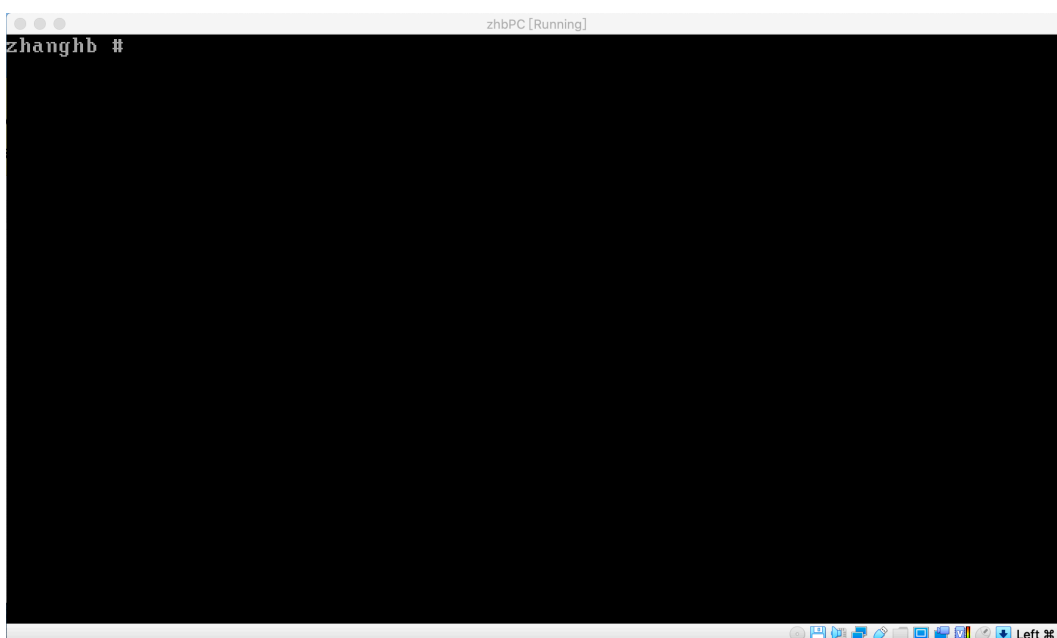
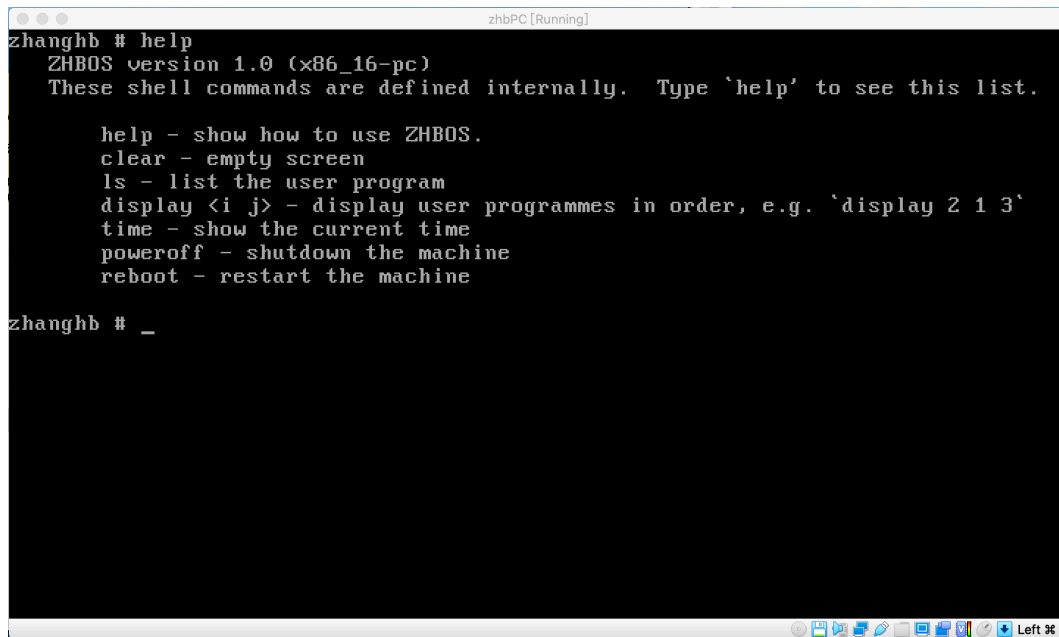


图 9: 返回 shell

然后输入 `display 1 3 2 4`, 可以看到先执行第一个用户程序, 按下 Ctrl + C 后执行第三个用户程序, 再按下 Ctrl + C 后执行第二个用户程序, 再按下 Ctrl + C 第四个用户程序, 最后按下 Ctrl + C 回到了 shell, 批处理结果符合预期。在这里省略这一部分截图, 在附件的视频中可以看到运行情况。

返回 shell 后，输入 help 显示提示信息：

A terminal window titled 'zhhPC [Running]' shows the output of the 'help' command. The prompt is 'zhanghb #'. The output lists several internal shell commands: '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), and 'reboot' (restart the machine). The prompt changes to 'zhanghb # _' at the end.

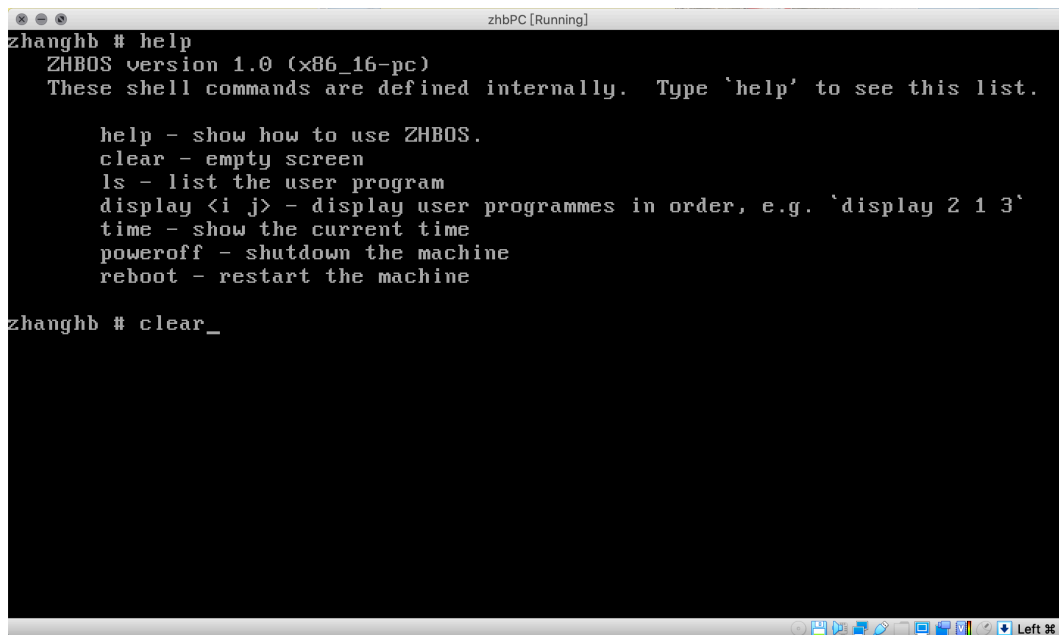
```
zhanghb # help
ZHBOS version 1.0 (x86_16-pc)
These shell commands are defined internally.  Type 'help' to see this list.

    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

zhanghb # _
```

图 10: 输入 help 显示提示信息

输入 clear:

A terminal window titled 'zhhPC [Running]' shows the output of the 'clear' command. The prompt is 'zhanghb #'. The output is identical to Figure 10, listing the internal shell commands. However, the prompt at the bottom is 'zhanghb # clear_', indicating the command has been entered but not yet executed.

```
zhanghb # help
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

zhanghb # clear_
```

图 11: 输入 clear

按下回车实现清屏:

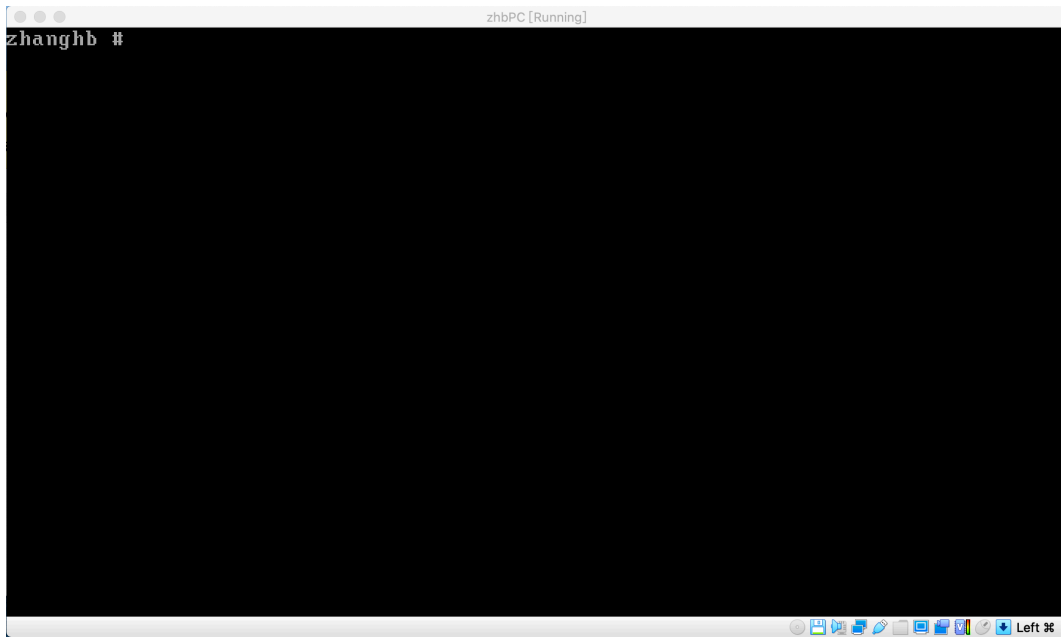


图 12: 实现清屏

然后输入 time 命令打印当前系统时间：

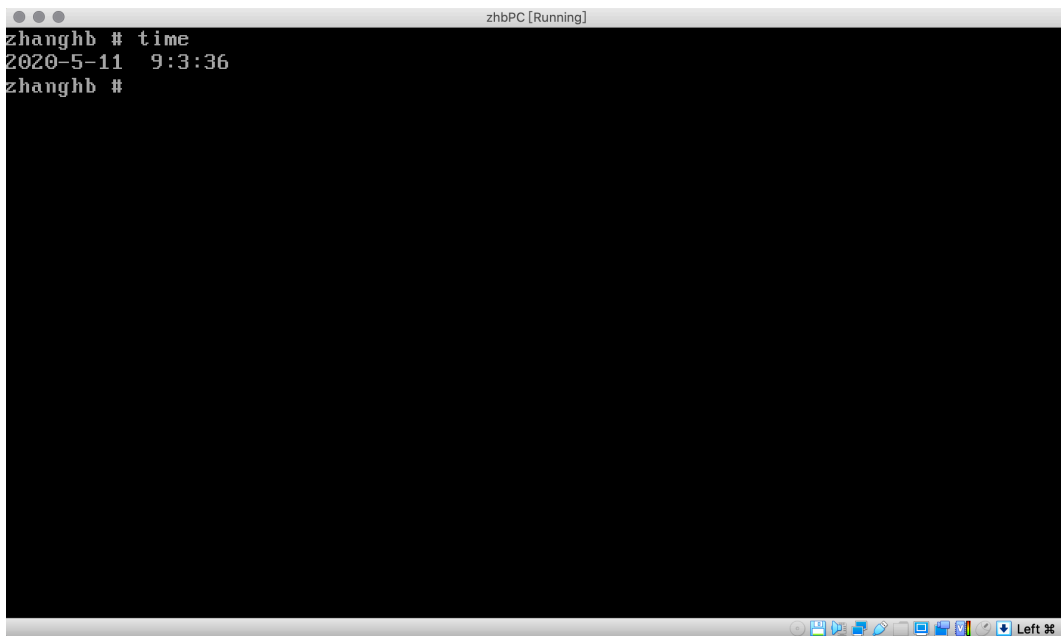


图 13: 获取当前时间

可以看到精确到秒到系统时间。

输入 reboot，按下回车，重启 OS：

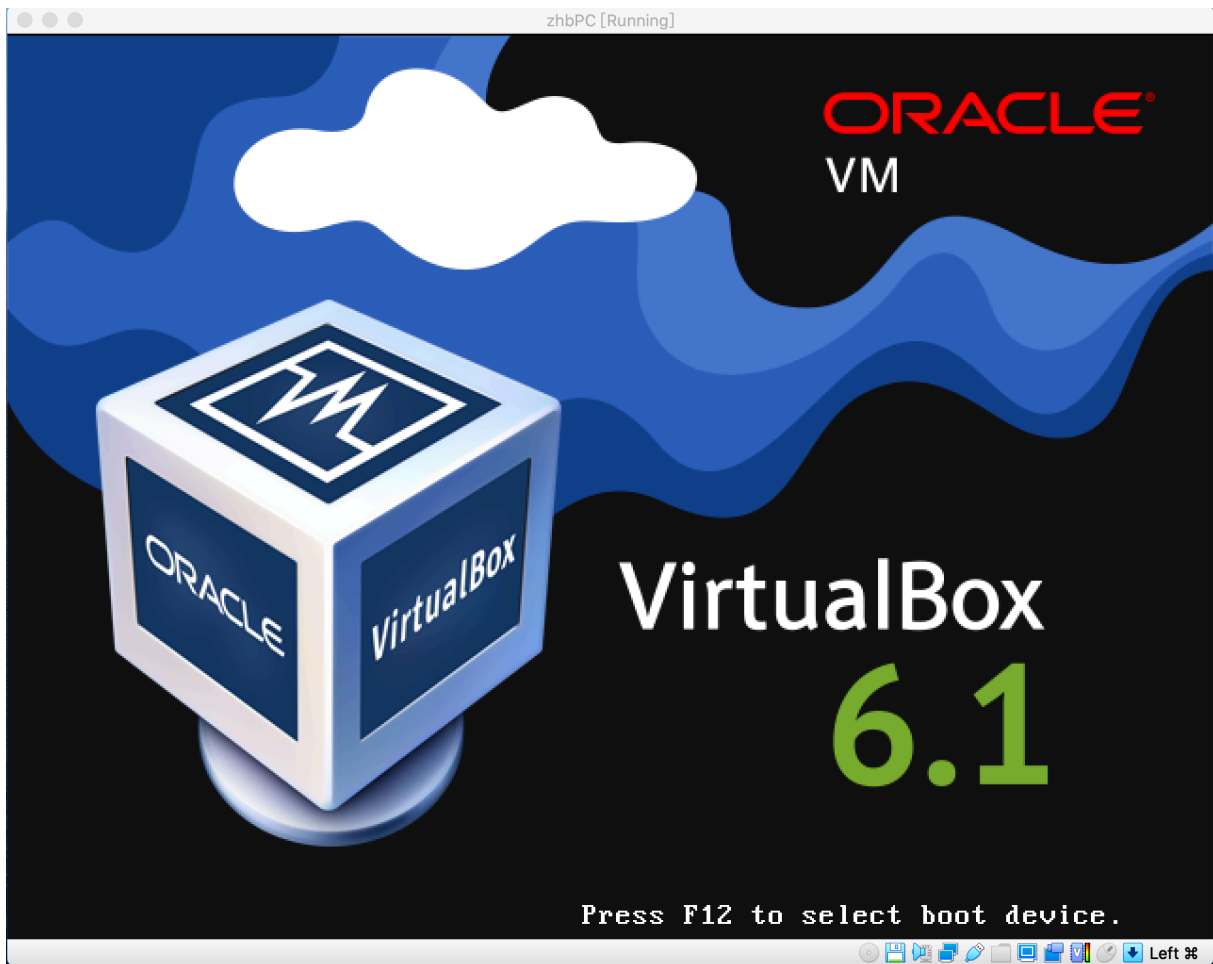


图 14: 重启 OS

可以看到操作系统进行了重启，然后进入了起始界面。

重新进入 shell，输入 `poweroff`，可以看到虚拟机被关闭，回到了虚拟机管理界面。（详情见附件展示视频）

7 实验总结

7.1 遇到的问题解决方法

此次实验难度较大，要点很多，值得思考的地方也有很多。

- 首先是 C 和汇编混合编译参数传递的问题，汇编代码中调用 C 函数只需要根据参数列表从右到左依次压入栈中。但是比较麻烦的是 C 代码中调用汇编函数，这时候也是通过栈传递参数的。但是很多时候，我们需要对函数中用到的寄存器进行压栈用于保存，以防函数调用的过程破坏了原来的运行情况。在这里以 `void putchar(unsigned char c)` 这个函数为例子，来分析用栈传递参数的

情况：

```
1 ; 输出一个字符
2 putchar: ; 函数：在光标处打印一个字符
3     push bp
4     push ax
5     push bx
6     mov bp, sp
7     add bp, 10 ; 参数地址
8     mov al, [bp] ; al=要打印的字符
9     mov ah, 0Eh ; 功能号：打印一个字符
10    mov bh, 0 ; bh=页码
11    int 10h ; 打印字符
12    pop bx
13    pop ax
14    pop bp
15    ret
```

首先，x86 的栈是从高地址到低地址增长的，即后进栈的变量在低地址处。

从 C 语言调用汇编函数相当于使用了 call，首先会将传递的参数压栈，然后会将当前 IP(call 的下一条指令压栈)。然后进入汇编的函数模块中，函数将 bp,ax,bx 压入栈，然后使用了 mov bp,sp 将旧的 bp 压入栈，这样子可以得出，参数地址应该是栈顶指针 sp 加上一段偏移量，而因为每个寄存器都是两个字节长度，所以这个偏移量应该是 $2 * 5 = 10$ 。栈的示意图如下：

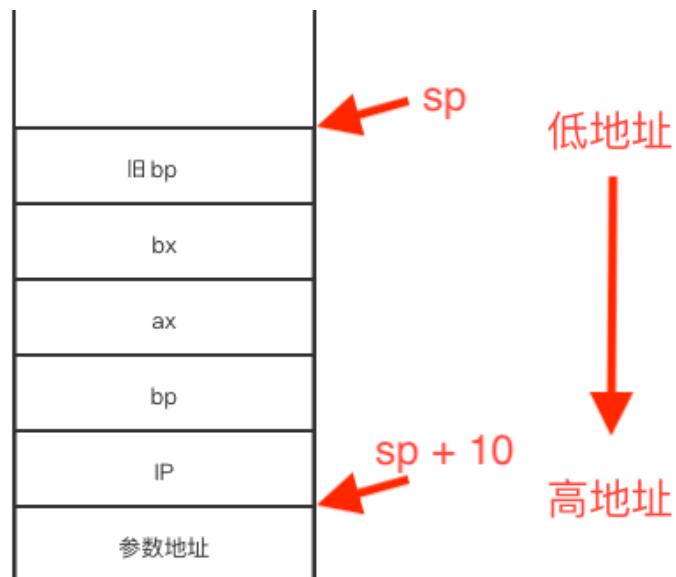


图 15: 栈的示意图

其他的汇编函数参数地址的计算方法与之类似。

- 在 x86 实模式中，不管是汇编代码中调用 C 函数还是 C 代码中调用汇编函数，都是通过 ax 存放返回值。
- 在实现运行指定用户程序的时候，为了实现批处理操作，我们需要执行完一个用户程序后回到批处理指令处，而不是回到 shell 开始的位置。而我想过使用类似于 call user_program 的方式，但是事实上这是行不通的，因为用户程序是单独编译的，所以这样子无法通过链接。

我在这里想了很久，最开始考虑修改 IP 寄存器，然后发现这个寄存器我们是无法修改的。我在网上看了很多资料，想了一个比较合适的方法：首先用 call dword 将 CS 和 IP 压栈，然后通过栈顶指针找到 IP 的值位置，然后用栈顶指针对 IP 地址的值进行修改，修改为返回操作系统的函数的地址：

```

1      call dword pushCsIp
2      pushCsIp:
3          mov si, sp          ; si 指向栈顶
4          mov word[si], return
5          jmp [bp]
6      return:
7          popa
8          ret

```

当用户程序执行结束后，使用 retf 返回，然后执行 return 指令，用 ret 返回 C 语言中对 Run 函数的调用处，就可以实现批处理操作了。

- 最开始访问用户程序表的时候，没有注意到要加上存储用户程序数量的偏移，因此出现了读取错误。

7.2 心得体会

这次实验相比上次之前的实验难度大了很多，汇编和 C 混合编译非常复杂，要研究 gcc、nasm、ld 的许多编译参数。为了实现混合编译，还得理解函数调用时的栈变化，参数的传递方式，返回值的传递方式，光是学明白这几样就花了我很长的时间。

而为了做好这次实验还得熟悉汇编语言，有了前两个实验的基础相对来说汇编语言的实现会变得简单一点，然而之前的编程极少涉及到栈，所以搞明白如何对栈操作又花了我很多时间。

还有很多很细节的内容，比如实现获取命令和参数的字符流的函数 get_buf，这个是我用之前做 FAT12 文件系统时用到的构建编辑器的函数修改而来的，但是结合自己实现的 IO 就显得非常的麻烦。还有很多细节的问题，也是经过深思熟虑才最终完成。

总体来说，这个实验收获非常大，通过这次实验，我更好地理解了内核的概念，也对操作系统这门课充满了更多的热情，对接下来的实验充满了期待。