



《操作系统实验》

实验报告

(实验三)

学 院 名 称 : 数据科学与计算机学院

专业 (班级) : 16 计算机类 4 班

学 生 姓 名 : 陈泓仰

学 号 : 15303009

时 间 : 2018 年 3 月 29 日

成绩：

实验三：开发独立内核的操作系统

一. 实验目的

1. 把原来在引导扇区中的监控程序（内核）分离成一个独立的执行体，存放在其他扇区中，为“后来”扩展内核提供发展空间。
2. 学习汇编与C语言混合编程技术，改写实验二的监控程序，扩展其命令处理能力，增加实现实验2要求中的部分或者全部功能。
3. 搭设GCC与nasm的交叉编译环境，为之后的操作系统实验做好准备。同时了解C与汇编在32位寄存器下相互调用的过程，以及注意事项。

二. 实验要求

实验的具体内容与要求。

1. 规定时间内单独完成实验，
2. 实验三必须在实验二的基础上进行，保留或扩展原有功能，实现部分新增功能。
3. 监控程序以独立的可执行程序实现，并由引导程序加载进内存适当位置，内核获得控制权后开始显示必要的操作提示信息，实现若干命令，方便使用者（测试者）操作。
4. 制作包含引导程序，监控程序和若干可加载并执行的用户程序组成的1.44M软盘映像。

5. 在指定时间内，提交所有相关源程序文件和软盘映像文件，操作使用说明和实验报告。
6. 实验报告格式不变，实验方案、实验过程或心得体会中主要描述个人工作，必须有展示技术性的过程细节截图和说明。

三. 实验方案

1. 实验工具和环境：

正如以往一样，由于我实在mac OS系统下做的实验，所以老师给的Turbo C和Tasm工具链没有一个我能用得上，然后查阅了很多资料，包括上GitHub上面看各位大神写16位操作系统所使用的工具环境，发现大部分人用的其实是GCC+NASM那套，在知道GCC+NASM编写16位操作系统是可行的之后，我开始搭建GCC+NASM的交叉编译环境。

好在用的是MacOS这种unix产物，安装GCC来编译GCC是一个非常容易的操作，不用搭设其他什么环境。

现在给出交叉编译的方法：

- 1) 首先需要先确保自己在系统中安装了GCC，然后，下载一个旧版本的GCC安装包（以4.8为例）和binutils，放在同一个文件夹中。
- 2) 然后进入该文件夹中，并修改环境变量。

```
cd $PROJECT_HOME  
mkdir toolchain
```

```
export PREFIX=$PROJECT_HOME/toolchain
export TARGET=i386-elf
export CC=/usr/local/bin/gcc-4.8
export CXX=/usr/local/bin/g++-4.8
export LD=/usr/local/bin/gcc-4.8
export CFLAGS=-Wno-error=deprecated-declarations
```

- 3) 解压binutils，并进入该文件夹中，记录配置文件。并开始make编译并安装。这里的binutils主要是提供将.o文件链接为二进制可执行文件bin的ld工具。

```
# make binutils (for i386-elf)

./configure --prefix=$PREFIX --target=$TARGET --disable-nls
make
make install
```

- 4) 同样的操作，解压下载的GCC包，进入该文件夹，记录配置文件，开始make编译并安装。

```
# make gcc (for i386-elf)

./configure --prefix=$PREFIX --target=$TARGET --disable-nls --enable-languages=c --without-headers
make all-gcc
make install-gcc
```

- 5) 返回根目录，现在你就可以看到自己搭设的工具链。
- 6) 搭设好环境之后，我们的GCC，还有ld工具都必须使用我们所编译的到的GCC和binutils中的。

- 7) 在每一个模块中使用make来实现代码的编译，链接，和删除。Make文件有点像批处理文件，通过预先写好makefile，处理好各个模块之间的依赖关系和链接关系，省去了编译环节大量的重复操作，一劳永逸。下列贴出makefile文件例子：

```
LD=/Users/chen/toolchain/bin/i386-elf-ld
LDFLAGS=-melf_i386 -N
CC=/Users/chen/toolchain/bin/i386-elf-gcc
CCFLAGS=-march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding
AS=nasm
ASFLAGS=

ROOT=..
SYSCALL=../syscall
USER=../shell
UP=../user

USER_HEADER=$(USER)/command.h
USER_OBJ=$(USER)/command.o
USER_PRO=$(UP)/user1.o $(UP)/user2.o $(UP)/user3.o $(UP)/user4.o
all: kernel.bin

kernel.bin: kernel.o $(SYSCALL)/use.o $(SYSCALL)/sysc.o $(USER_OBJ) $(USER_PRO)
    $(LD) $(LDFLAGS) -Ttext 0xA100 --oformat binary -o $@ $^
kernel.o: kernel.c kernel.h $(USER_HEADER)
    $(CC) $(CCFLAGS) -c $^
$(SYSCALL)/sysc.o:
    cd $(SYSCALL)/ && make
    cd ../kernel
$(SYSCALL)/use.o:
    cd $(SYSCALL)/ && make
    cd ../kernel
$(USER)/command.o:
    cd $(USER)/ && make
    cd ../kernel
$(UP)/user1.o:
    cd $(UP)/ && make
    cd ../kernel

clean:
    rm *.bin -f
    rm *.o -f
    rm *.gch -f
rebuild:
    make clean
    make
```

2. 相关基础原理

- 1) 在进行编写内核的实验之前，我们必须清楚，汇编和C这两者到底起着什么作用。
 - a) 汇编主要负责的是底层的实现，负责与I/O进行交互，设置I/O端口实现I/O操作，

同样的我们可以使用汇编来初始化中断向量表 and 实现中断处理，以及初始化各种寄存器。

- b) 但是一个操作系统中又不能纯粹的只使用汇编写，高级语言只要几行代码就能实现的功能，如果用汇编语言写出来的话，通常就需要十几行甚至几十行才能实现。因此使用C语言，我们可以更简单的实现更复杂的功能或算法。

2) C与汇编的互相调用

- a) 要是想让nasm可以被C所调用,这里必须在nasm汇编代码的该例程前声明global 函数名，同时在c代码中声明extern。
- b) 要是想让C可以被nasm调用，只需要在 nasm 汇编代码前声明 extern 函数名 然后使用call函数名即可。
- c) 但是由于我们使用的是GCC和nasm交叉编译，虽然我们可以通过设置gcc指令的参数来使用16位代码，但是寄存器还是32位的，这就导致了程序在相互调用的过程中，存在压栈的问题。为了抵消影响，在被调用的汇编函数返回调用的C函数时，我们必须多弹出一个字长的数据，才能最终返回到合适的地址。
- d) 当我们在汇编中要调用含有参数和返回值的C函数时，C函数的参数其实也是压在栈中，并通过bp来寻找参数的。而返回值会存在ax寄存器中。

3) C与汇编互相调用的具体过程：

- a) 栈帧结构：

函数调用经常是嵌套的，在同一时刻，堆栈中会有多个函数的信息。每个未完成运行的函数占用一个独立的连续区域，称作栈帧(Stack Frame)。栈帧是堆栈的逻辑片段，当调用

函数时逻辑栈帧被压入堆栈，当函数返回时逻辑栈帧被从堆栈中弹出。栈帧存放着函数参数，局部变量及恢复前一栈帧所需要的数据等。

编译器利用栈帧，使得函数参数和函数中局部变量的分配与释放对程序员透明。编译器将控制权移交函数本身之前，插入特定代码将函数参数压入栈帧中，并分配足够的内存空间用于存放函数中的局部变量。使用栈帧的一个好处是使得递归变为可能，因为对函数的每次递归调用，都会分配给该函数一个新的栈帧，这样就巧妙地隔离当前调用与上次调用。

b) C函数被调用时的入栈顺序是这样的：

实参 N~1→主调函数返回地址→主调函数帧基指针 EBP→被调函数局部变量 1~N

主调函数会将参数按照调用约定压入栈中（从右往左），然后将指令指针EIP（32位）入栈保存主调函数的返回地址，同时进入被调函数时，被调函数将主调函数的帧基指针BP入栈，并将主调函数的栈顶指针ESP值赋给被调函数的EBP(作为被调函数的栈底)，接着改变ESP值来为函数局部变量预留空间。

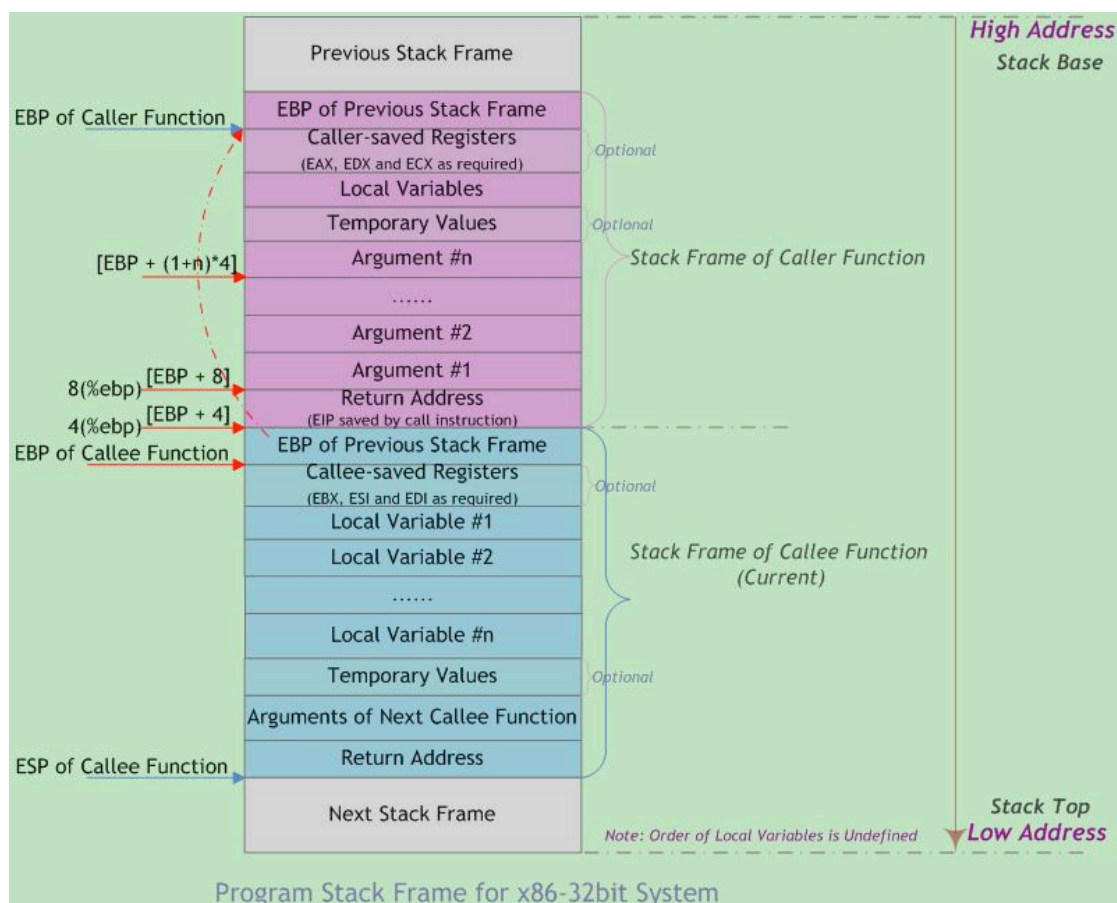
此时被调函数帧基指针指向被调函数的栈底。以该地址为基准，向上(栈底方向)可获取主调函数的返回地址、参数值，向下(栈顶方向)能获取被调函数的局部变量值，而该地址处又存放着上一层主调函数的帧基指针值。因此当被调函数想要获得主调函数的参数时，即可将[bp+6]开始的第一个参数一次穿进被调函数中即可。本级调用结束后，将EBP指针值赋给ESP，使ESP再次指向被调函数栈底以释放局部变量；再将已压栈的主调函数帧基指针弹出到EBP，并弹出返回地址到EIP。

c) 汇编函数被调用时的入栈时：

由于，C函数在调用汇编函数时也是按照调用约定一样来取得参数的，所以主调函数在获得参数的时候总会少了一个字，因此被调函数应该在将所有参数全部压入栈中之后，再压

入一个字的数据，这样才能达到主调函数与被调函数两者之间的平衡。

d) 下面附调用过程说明图：



注：这里的被调函数获得参数的ebp的位置与16位的操作系统有区别，第一个参数位置应为 $[ebp+6]$ ，第二个是 $[ebp+10]$ ，以此类推。

4) 加载用户程序

我将内核程序加载到0x7e00处，然后将用户程序加载到了0xc000处，在调入用户之前需要先将各种段寄存器压栈。

3. 模块结构

由于缺少开发经验，因此在这次开发操作系统的过程中，不知道应该如何分配操作系统的模块结构，但是想着一定不能只是单纯的将汇编和C文件分开，而是应该有层次有条理的将各

种功能妥善管理。于是分为以下模块：

- 1) syscall: 用汇编语言实现最底层的与I/O端口交互的功能，包括显示字符，输入字符，从软盘中读写程序等。同时用C语言将这些功能进行封装，供其他模块调用。
- 2) shell: 主要是用C语言实现命令行运行。
- 3) user: 存放用户程序的模块。
- 4) loader: 引导程序，将内核程序引导到内存，并开始执行内核。
- 5) kernel: 用C语言来调用其他模块实现的函数，起着逻辑层上的调节作用。这是一个操作系统中最为核心的地方。

4. 程序实现的功能描述：

- 1) 实现了最基本的底层输入输出函数，也实现了对字符串的比对和提取处理的函数。
- 2) 使用字符串输出函数，让整个操作系统有良好的引导界面。
- 3) 增加了实验二所没有的多道批处理系统，可以对用户输入的多用户程序队列进行执行。
- 4) 通过实现中断，用户程序调用中断来返回内核程序。
- 5) 通过读取扇区，将用户程序加载到内存中。

5. 程序流程：

- 1) 引导程序：将内核加载到内存中，并跳到内核程序中开始执行。
- 2) 内核程序：首先先执行清屏操作，然后调用终端程序，在终端程序中会不断的显示命令行，提示用户如何输入可识别的命令。

- 3) 批处理程序：这个内核中实现了一个批处理系统，用户可以输入多个可运行的程序队列，在按下enter键之后，会带着整个队列跳转到执行用户程序的函数中执行。
- 4) 执行用户程序的函数：这里将用户程序放在不同软盘中，当用户程序被调用时，就加载在内存的0xC000处。

四. 实验过程与结果

说明：根据需要书写相关内容，如：

主要工具安装使用过程及截图结果、程序过程中的操作步骤、测试数据、输入及输出说明、遇到的问题及解决情况、关键功能或操作的截图结果。

1. 算法说明、模块介绍：

1) 加载程序：

使用中断，将存在软盘扇区的内核程序加载到内存的0XA100处，并跳转到内核程序中开始执行。相关中断介绍：

读扇区↵	13H↵	02H↵	AL: 扇区数(1~255)↵ DL: 驱动器号(0和1表示软盘, 80H和81H等表示硬盘或U盘)↵ DH: 磁头号(0~15)↵ CH: 柱面号的低8位↵ CL: 0~5位为起始扇区号(1~63), 6~7位为硬盘柱面号的高2位(总共10位柱面号, 取值0~1023)↵ ES:BX: 读入数据在内存中的存储地址↵	返回值: ↵ ■ 操作完成后ES:BX指向数据区域的起始地址↵ ■ 出错时置进位标志CF=1, 错误代码存放在寄存器AH中↵ ■ 成功时CF=0、AL=0↵
------	------	------	--	---

相关代码：

```

Read0S:
;OS OFFSET
mov ax, OS_SEGMENT
mov es, ax
mov bx, OS_OFFSET
mov ah, 2 ; kind of function
mov al, 0x20 ; read num of shanqu,小心! 不止加载一个扇区!
mov dl, 0 ; floppy
mov dh, 0 ; citou
mov ch, 0 ; zhumian
mov cl, 2 ; start_shanqu
int 13h
JUMP_TO_OS:
;Clear Screen
mov ax, 3
int 10h
;Print
mov ax, cs
mov es, ax
mov ax, 1301h
mov bx, 0006h
mov dh, 10
mov dl, 35
mov bp, Load_Info
mov cx, Load_Info_Len
int 10h
;Check Key
mov ah, 00h
int 16h
;Enter Kernel
jmp 0:OS_OFFSET

```

在这里必须注意，我们要将扇区数传到al寄存器中，扇区数必须足够大，否则无法将完整的内核程序加载到内存，会出现读不到指令的情况。

2) 底层syscall模块：

在这个模块中，用汇编实现最基本的显示字符和读取用户输入字符的函数。

分别为：

a) 按位置显示字符[_printchar(char,pos,color)]:

```
ret
_printchar:
    enter 0,0
    push di
    mov ax,0xB800
    mov es,ax
    mov ecx,[bp+6];char//ip = 2 bytes,esp = 4 bytes
    mov edi,[bp+10];pos
    mov edx,[bp+14];color
    mov ch,dl
    mov [es:di],cx
    pop di
    leave
    newret
```

这里要注意参数压栈的顺序问题。

b) 按照扇区位置，读取（存入）扇区

```
_loadP:
    enter 0,0
    push es
    push ds
    push cs
    push ebx
    push dx
    SetInt 20h,_SetINT20h
    mov ax,word [Pg_Segment]
    mov es,ax
    mov dl,0
    mov ax,[bp+10];起始扇区号
    mov bl,18
    div bl
    ;inc ah
    mov cl,ah
    xor ah,ah
    mov bl,2
    div bl
    mov dh,ah
    mov ch,al
    mov al,byte [bp+6];读多少个扇区
    mov bx,word [bp+14]
    mov ah,2
    int 13h
    jmp bx;跳到用户程序。
```

在调用子程序的过程中，发现了读取扇区时的一个需要特别注意的细节。扇区数是0-17，然后每过18个扇区，磁头就会变换位置（从0到1，或者从1到0），因此根据这个规律，用除余法，写了一个可以根据实际的扇区数读存扇区的函数。

c) 调用中断显示字符函数[_showchar(char)]

```

_showchar:
    enter 0,0
    push bx
    mov ax,[bp+6] ; ASCII码
    mov ah,0eh ; 功能号
    ;mov bh,0
    mov bl,0 ; Bl设为0
    int 10H ; 调用中断
    pop bx
    leave
    newret

```

d) 读取用户输入的函数[_readinput()]

```

_readinput:;not yet try;wait for input
    enter 4,0;use for output
    mov ah,00h
    int 16h
    mov ah,0
    mov [esp],eax
    mov eax,[esp];save the output by this way
    leave
    newret

```

注意：

enter 4, 0 和 leave 是配套的。

他们共同的作用是：

Enter 4, 0	leave
Push ebp	Add ebp,4
Mov ebp,esp	Mov esp,ebp
Sub ebp,4	Pop ebp

目的是为汇编函数开辟栈，保存参数。

最后将输出结果放在eax寄存器中。

- e) 设置了一个改写中断的宏，通过这个宏可以免去重复的代码量。

```
%macro SetInt 2;载入中断向量表的宏
    push es
    push di
    push bx
    mov ax,0
    mov es,ax
    mov ax,%1
    mov bx,4
    mul bx
    mov di,ax
    mov ax,%2
    mov [es:di],ax
    mov ax,cs
    mov [es:di+2],ax
    pop bx
    pop di
    pop es
%endmacro
```

但在改写时，需要注意的是，应该对相关的寄存器进行压栈保护，否则破坏现场，会对内核造成影响。

- f) 要注意的是由于C调用汇编函数时，会因为压了一个32位的地址的原因，所以我们不能单纯的使用ret指令返回，而是需要多pop一个字节的数据出来，再返回主调函数。

```
%macro newret 0;inorder to get back,have to match with enter-leave
    pop dx
    jmp dx
%endmacro
```

- g) 我们在编写可以被调用的汇编函数之前，如需用到一些寄存器的话，则

需要对某些寄存器进行压栈保护。表中显示了这样的一些寄存器：

寄存器	状态
eax	用于保存输出值，但是可能在函数返回之前被修改
ebx	用于指向全局偏移表，必须保留
ecx	在函数中可用
edx	在函数中可用
ebp	C程序使用它作为堆栈基址指针；必须保留
esp	C程序使用它作为堆栈基址指针；必须保留
edi	C程序使用它作为堆栈基址指针；必须保留
esi	C程序使用它作为局部寄存器；必须保留
ST (0)	保存浮点输出值
ST(1)–ST (7)	在函数中可用

3) syscall模块中的接口函数：

主要是对汇编函数调用的一些封装，留出调用接口：


```
#ifndef USE_H
#define USE_H
extern void _clearscreen();
extern void _printchar(char pchar, int pos, int color);
extern char _readinput();
extern void _showchar(char phar);
extern void _setPoint();
extern void _row_the_screen();
extern void _getch();
extern void _write(int count, int start, void* addr);
void print(char const* Messeage, int row, int colume);
void prints(char const *Messeage);
void print_next_line(char const* Messeage);
int strlen(char const *Messeage);
void read_and_print_input();
char waitforinput();
void clearscreen();
int strcmp(char const *m1, char const *m2);
void bulidmap();
void showtable();
struct info{
    char name[4][16];
    int size[4];
    int sector[4];
    int num;
};
#endif
```

4) shell模块，实现了命令行处理的终端函数，以及加载用户程序的命令：

- a) 首先不断的读取键盘输入：
- b) 然后分情况对键盘输入进行判断处理。

当按空格时，会将输入的字符转为数字，存进用户程序执行队列中。

```

if(save == 32){//空格
    if(num_of_queue == Maxsize){//已达最大队列数，对用户进行提示。
        prints("\n\r");
        print_next_line(controlMsg1);
        prints(CMDHead);
        continue;
    }
    _showchar(save); //显示输入
    queue[num_of_queue] = number; //将数据存入执行队列中
    num_of_queue ++;
    number = 0;
    continue;
}

```

当按数字时，会对数字进行ascii码到数值的转换。

```

else if(save >= '0' && save <= '9'){
    _showchar(save);
    number = number * 10 + save - '0'; //防止有多位数程序需要执行
    continue;
}

```

当按英文字母时，会将英文字幕存入缓冲字符串中，等待字符串匹配功能调用。

```

else{
    _showchar(save);
    // if(save == '-' || save == 'q' || save == 'h' || save == 'l' || save == 's'){
    if(index_of_str < 10){ //对字符串进行赋值操作，必须按照这种发式向字符串传值，否则会出现bug。
        recived[index_of_str] = save; //字符串相应位置赋值
        index_of_str ++; //字符串指针指向下一位
        recived[index_of_str] = 0; //字符串末尾赋值为0
        //_showchar('0' + strlen(recived));
        //continue;
    }
    else{
        continue;
    }
}

```

当按回车时，会根据执行队列和缓冲字符串的情况，来执行不一样的功能。

```
if(strcmp(recived,help) == 1){//缓冲字符串==help字符串
    prints(HelpMsg1);
}
else if(strcmp(recived,quit) == 1){//缓冲字符串==quit字符串
    prints(ByeByeMsg);
    //prints(filelist);
}
else if(strcmp(recived,lst) == 1){//缓冲字符串==ls字符串
    prints(filelist);
}
while(index_of_str != 0){//clear the string
    recived[index_of_str] = 0;
    index_of_str --;
}
prints("\n\r");
prints(CMDHead);
continue;
}

if(number != 0){//
    queue[num_of_queue] = number;
    num_of_queue++;
}
run(queue,num_of_queue);//执行队列中的程序开始执行。
num_of_queue = 0;//清空执行队列。
number = 0;
prints(CMDHead);
while(index_of_str != 0){//clear the string
    recived[index_of_str] = 0;
    index_of_str --;
}
continue;
```

当队列为空时，会弹出队列为空的提示。

5) 内核模块:

首先需要先使用内联编译，将初始化段寄存器:

```
#ifndef __KERNEL_H_
#define __KERNEL_H_
__asm__(".globl _start\n");
__asm__("_start:\n");
__asm__("mov $0, %eax\n");
__asm__("mov %ax, %ds\n");
__asm__("mov %ax, %es\n");
__asm__("jmp $0, $main\n");
#endif
```

最后内核文件中，调用函数实现即可。

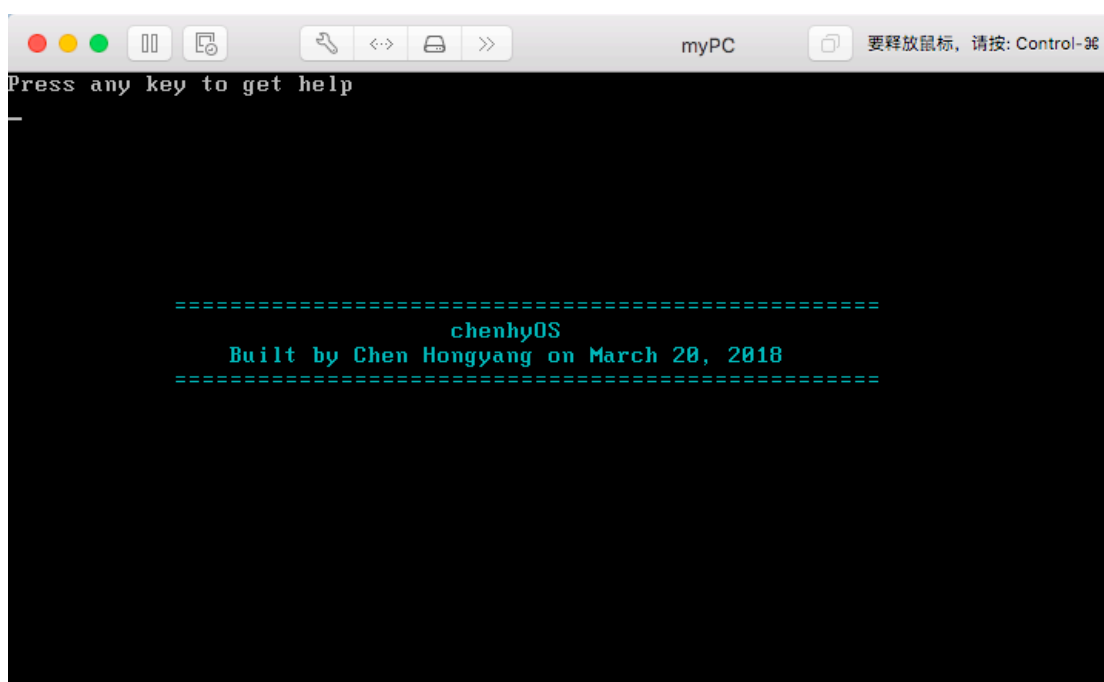
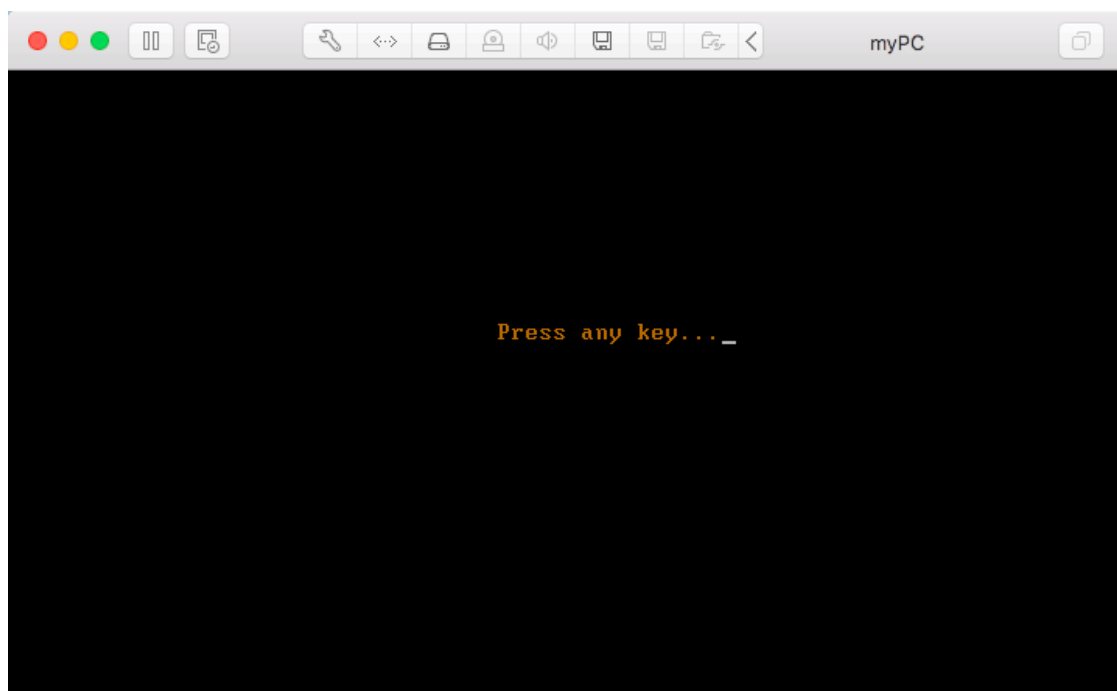
2. 算法实现：

这次实验并没有涉及到文件管理系统，内存的分配，也没有专门的给文件结构表开设一个特别的数据结构。比较亮眼的就只有用C实现了多道批处理。具体算法如下：

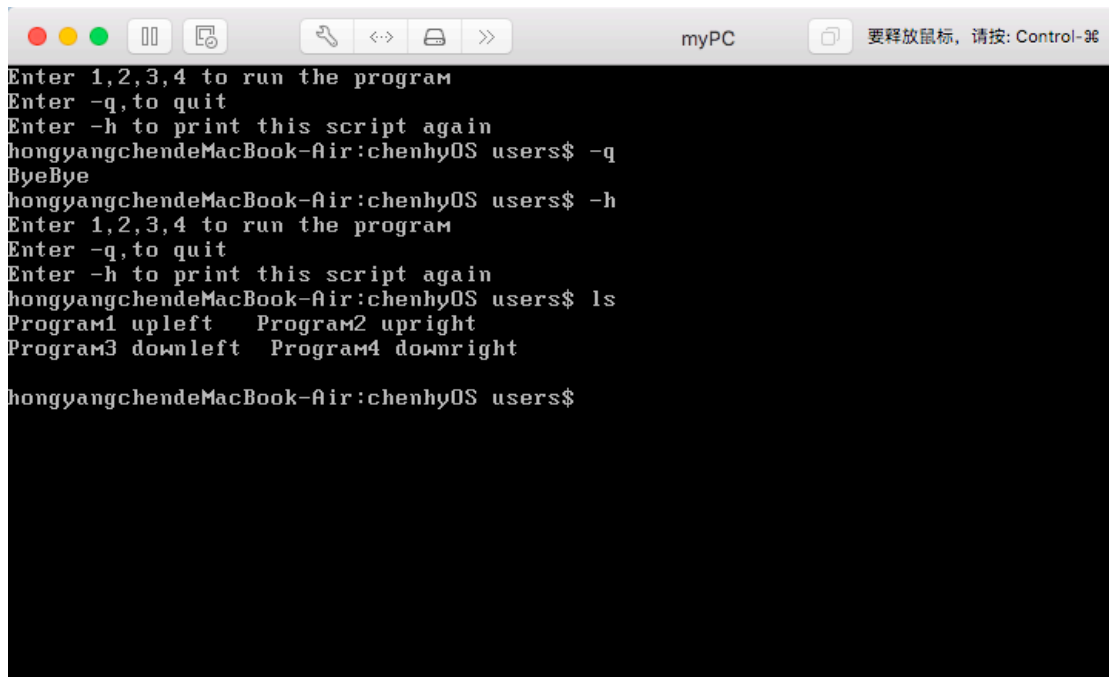
- 1) 先将用户的输入数字进行处理，转化为数字，在每次按了空格键时将数字存在执行队列中。
- 2) 在按了回车键之后，调用runProgram函数，运行执行队列中的用户程序。
- 3) 由于没有实现分时系统，因此，**写了一个读取用户键盘状态的中断**。用户程序通过使用这一中断，读取用户键盘，但用户按任意键之后继续执行执行队列中的下一程序。

3. 最终效果展示：

开机界面：

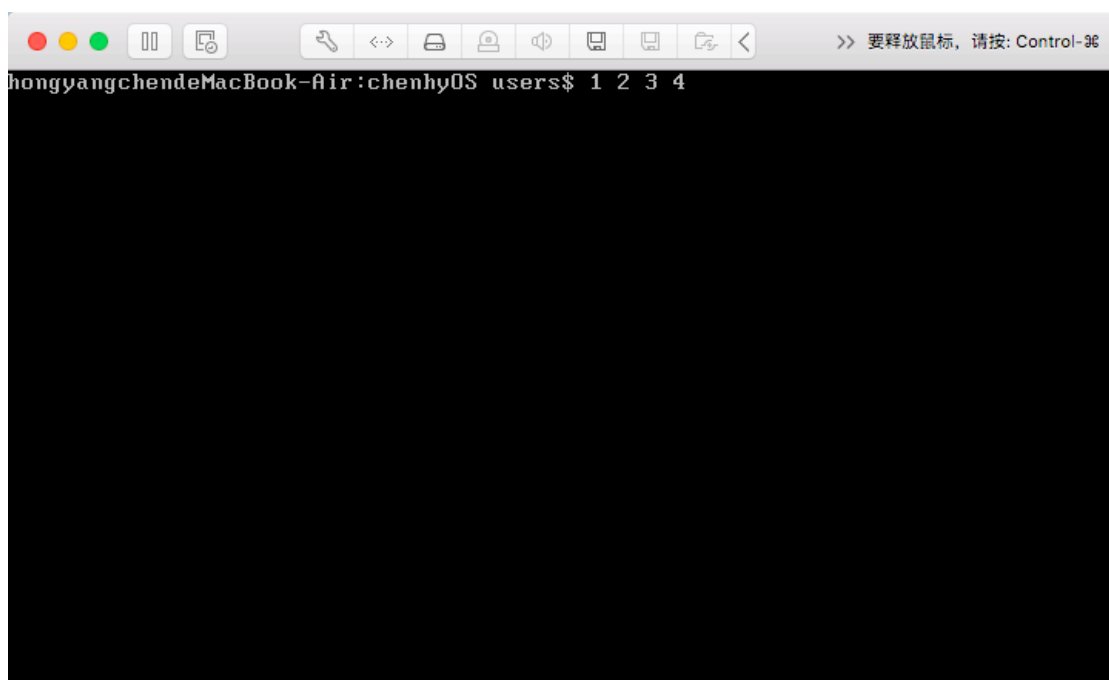


命令行界面：

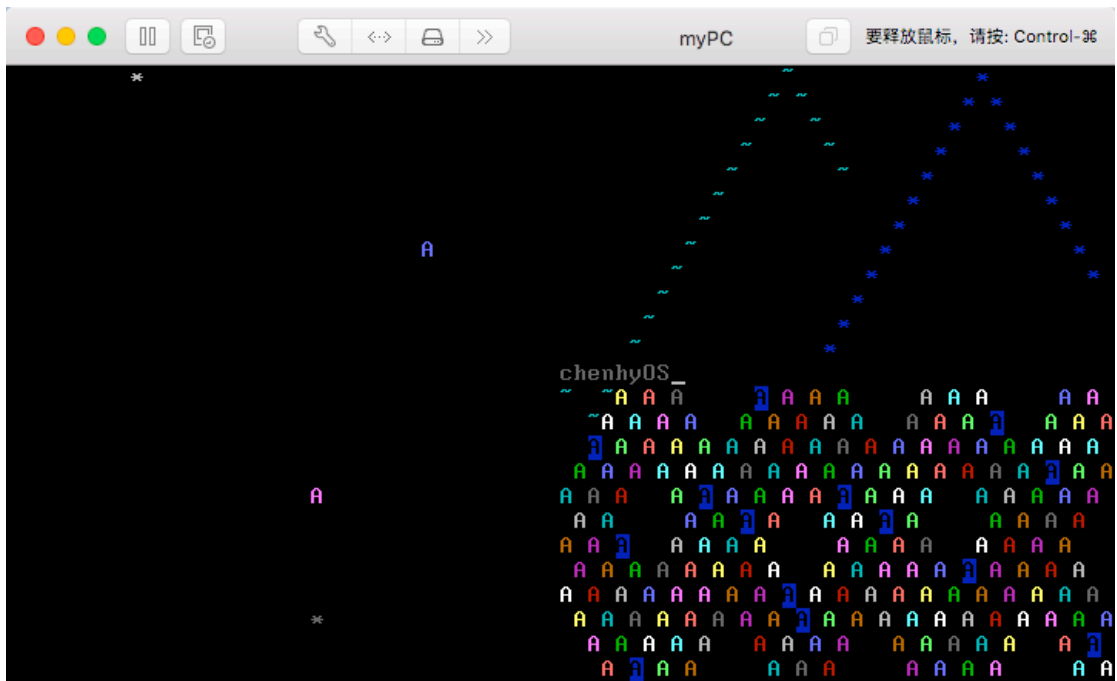
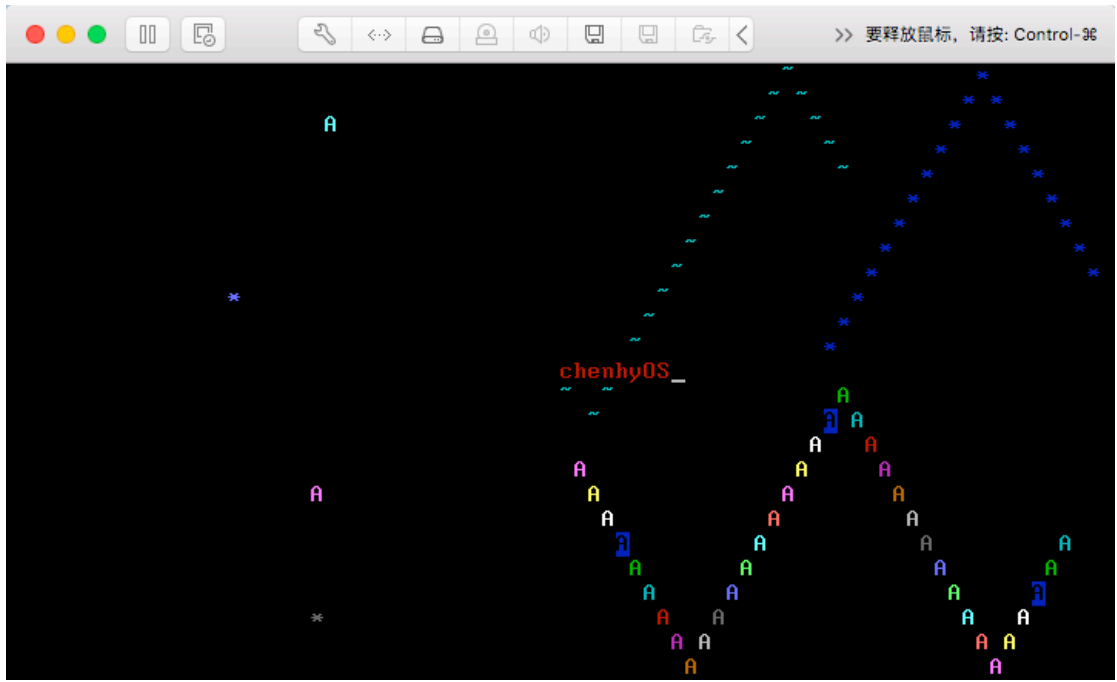


```
myPC 要释放鼠标, 请按: Control-⌘
Enter 1,2,3,4 to run the program
Enter -q,to quit
Enter -h to print this script again
hongyangchendeMacBook-Air:chenhyOS users$ -q
ByeBye
hongyangchendeMacBook-Air:chenhyOS users$ -h
Enter 1,2,3,4 to run the program
Enter -q,to quit
Enter -h to print this script again
hongyangchendeMacBook-Air:chenhyOS users$ ls
Program1 upleft  Program2 upright
Program3 downleft Program4 downright
hongyangchendeMacBook-Air:chenhyOS users$
```

多道批处理程序:



```
>> 要释放鼠标, 请按: Control-⌘
hongyangchendeMacBook-Air:chenhyOS users$ 1 2 3 4
```



五. 实验心得

体会和建议。

在这次实验,我的前期准备工作做的非常的辛苦。由于从来没有接触到C和汇编的混合编译,而且老师给的工具在mac OS系统中还用不了,因此必须另辟蹊径。好在上网查了好一段时间之后,发现网上有使用gcc和nasm写16位操作系统的例子,因此我就开始搭建交叉编译环境,并开始学习写make文件来自动化编译。整个过程虽然非常辛苦,但是学到了许多之前并没有接触过的东西,比如交叉编译以及写make文件等知识。

然而搭建好环境只是其中的一步,要想写出一个可以用来加载用户程序的内核其实还需要付出大量精力,特别是需要才更多的坑。虽然gcc可以编译出16位代码,但是其中的汇编实现用的还是32位的寄存器,所以C与汇编的相互调用是一个很棘手的问题。我们需要考虑到4种情况,包括汇编带参数的C函数,汇编调用有返回值的C函数,C调用带参数的汇编函数,C调用有返回值的汇编函数。按照我的理解能力,网上大部分的资料在没有经过我的实践之前,都是难以理解的。于是乎,我开始研究C与汇编互调其中的过程,在弄清楚了每一句C语言背后用汇编实现的语句以及其过程之后,我也开始摸索出一套可以专门用来使用的规则。在这个摸索的过程中,我也学习并且了解了bochs这个调试工具的使用方法。而这时回顾之前找的那些资料,我可以惊喜的发现,原来这里面的东西说的都很有道理,这一个一个的过程其实就是我的这一套规则的理论基础。

在解决了相互调用这一棘手的问题之后,我就开始思考如何去分配我的操作系统代码的结构,使之变得比较有条理。在网上查了各种资料之后,我决定将代码分成几个模块,每个模块负责不同的功能。

虽然,这次的操作系统算是基本完成了任务,但是,我还是有很多并没有做好的地方。首先是虽然在磁盘中建立一个表来存储程序,但是并没有能够真正的利用这个表来显示,用

户程序的相关内容还是需要自己手动输出。其次就是关于用户程序的加载问题了。一开始，我的用户程序和内核程序是链在一起的，两者并没有什么特别明显的分界。也就是说我的用户程序的实现其实就算是内核的一部分，这在逻辑上是行不通的。后来，经过和舍友的讨论之后，决定用类似于加载内核程序的方法，在内核中加载用户程序。但是在实现这一功能的时候发现这里面有一点比较奇怪的地方，那就是在程序中扇区数只能为0到17的问题。与之同时也就是如何解决加载用户程序的时候不会覆盖到内核程序的问题，因此必须妥善处理用户程序以及内核程序在内存中的位置。虽然我将内核程序加载到0x7e00,将用户程序加载到0xC000,之后可以解决这个问题，但是随着内核的不断增大，内核在内存所占的大小也会越来越大，因此随后的实验必须开拓出新的方法来解决这一个棘手的问题。

六. 参考资料

1. 《orange's:一个操作系统的实现》
2. “The C Calling Convention and the 8086: Using the Stack Frame”
<http://ece425web.groups.et.byu.net/stable/labs/StackFrame.html#QuickReference>
3. nasm手册
4. “在汇编函数中使用C函数”
<https://blog.csdn.net/longintchar/article/details/79511747>
5. “C语言函数调用栈”
<http://www.cnblogs.com/clover-toeic/p/3755401.html>
6. “linux平台学x86汇编（十九）：C语言中调用汇编函数”
<https://blog.csdn.net/shallnet/article/details/45651601>