

---

## 《操作系统实验》

### 实验三：开发独立内核的操作系统

---

17341190 叶盛源

2019 年 3 月 29 日

## Contents

1 实验目的:	2
2 实验要求:	2
3 实验方案:	2
3.1 实验环境: . . . . .	2
3.2 实验工具: . . . . .	3
3.3 实验思想: . . . . .	3
3.4 实验过程: . . . . .	5
4 实验心得和总结:	15
5 参考文献:	16

## 1 实验目的:

---

- 1) 将实验二的原型操作系统分离为引导程序和MYOS内核，由引导程序加载内核
- 2) 搭建TCC+Tasm的交叉编译环境，为之后的操作系统实验做准备，熟悉掌握汇编语言和C语言的混编

## 2 实验要求:

---

- 1) 实验三要建立在实验二的基础上，将实验二的原型操作系统分离为引导程序和MYOS内核，由引导程序加载内核，**用C和汇编实现操作系统内核。**
- 2) 使用汇编和C语言混合编程技术，使得操作系统具有执行命令的能力。拓展内核汇编代码，增加有用的输入输出函数，供C模块中调用
- 3) 提供用户程序返回内核的一种解决方案
- 4) 在内核的C模块中实现增加批处理能力
  - i 在磁盘上建立表，记录用户的储存安排，实现一个简单文件系统。
  - ii 可以在控制台命令查到用户程序的信息，如程序名、字节数、在磁盘映像文件中的位置
  - iii 设计一种命令，命令中可以加载多个用户程序，依次执行，并能在控制台发出命令
  - iv 在引导系统前，在磁盘上储存一个批处理的脚本，然后系统可以执行这个脚本

## 3 实验方案:

---

### 3.1 实验环境:

- 1) 实验运行环境: Windows10
- 2) 虚拟机软件: VMware Function和DOSBox
- 3) TCC+Tasm+TLink 混合编译链接
- 4) NASM 编译器

### 3.2 实验工具：

- 1) 汇编语言：NASM、TASM
- 2) 文本编辑器：VScode、notepad++
- 3) 软盘操作工具：WinHex

### 3.3 实验思想：

- 1) 汇编和C语言的关系：
  - i 汇编是实现底层的指令，它负责和I/O进行交互，在我们的C语言中经常用到的printf之类的语句其实都是要通过汇编语言的帮助来实现。
  - ii 但操作系统因为功能繁多，而汇编语言函数和指令都太少，无法应对繁重的编写操作系统工作。所以我们可以借助C语言的帮助，通过用汇编实现一些简单的和I/O的输入输出操作函数，再利用C语言调用这些函数来实现更多更复杂的功能或算法。
- 2) 汇编和C语言的链接编译
  - i 使用TCC来编译我们写的C语言程序，指令为 `tcc 文件名.c` 如果编译成功，会生成一个.obj文件，如下图：

```
C:\YSY_ASM\OS3>tcc kernel.c
Turbo C Version 2.01 Copyright (c) 1987, 1988 Borland International
kernel.c:
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International
c0s.obj : unable to open file

available memory 447862
```

Figure 1: TCC编译C语言代码

- ii 使用TASM来编译汇编语言程序，指令为 `tasm 文件名.asm` 如果编译成功，会生成一个.obj文件,否则会出现error和出现的位置，如下图：

```
C:\YSY_ASM\OS3>tasm monitor.obj
Turbo Assembler Version 4.1 Copyright (c) 1988, 1996 Borland International

Assembling file:  monitor.ASM to  obj.OBJ
Error messages:  None
Warning messages:  None
Passes: 1
Remaining memory: 465k
```

Figure 2: TASM编译汇编语言代码

- iii 使用TLink来链接C和汇编各自生成的obj文件，如果成功会生成一个.com后缀的可执行文件，这个可执行文件在DOS操作系统下可以直接打开，指令为： `tlink 文件名.obj 文件名.obj,run.com` ,如下图：

```
C:\YSY_ASM\OS3>tlink /t /3 monitor.obj kernel.obj,run.com
Turbo Link Version 2.0 Copyright (c) 1987, 1988 Borland International
C:\YSY_ASM\OS3>
```

Figure 3: TLink链接两个.obj文件

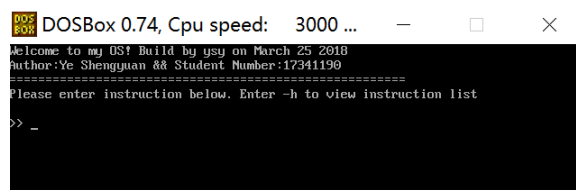


Figure 4: 使用DOSBox运行生成的.com文件

### 3) 混合编程过程、变量引用和参数传递：

- i **C变量名汇编后前面加了下划线** 如 `_a`和`_b`；函数名也如此，如`_f`，因此在汇编中的变量一般都有下划线，而C中的则没有。在C中引用汇编变量和过程也要去掉下划线。
- ii **C模块中调用汇编模块的函数** 如果C中调用了汇编模块的代码，需要在C文件头部用`extern`语句声明汇编函数（注意要去除下划线），如果要传入参数的话，C语言会自动将参数压栈，但要注意，**因为是near的调用，函数还会把此时的ip地址压入栈中，所以参数的位置在[bp+2]开始的位置。**
- iii **汇编模块中调用C模块中的函数** 调用前要用`extrn`声明C模块的函数。参数传递时，用`push word ptr _DATA :_b`就是实参b压栈，而`push word ptr _DATA :_a`就是实参a压栈，然后调用函数`f(a,b)`，之后两个POP指令是调用后清除栈中的参数。说明调用C函数时，参数按后面参数先进栈的顺序压栈。
- iv **模块中调用汇编模块中的过程** 如果汇编模块的过程要让C模块中调用，则应在TASM汇编中申明为`public`，汇编模块的过程从栈中取得参数，不应出栈，顺序与C进栈对应

### 4) 模块结构

- i) **loader模块**: 内容类似实验二中的引导程序, 用来引导内核加载进入内存并运行
- ii) **monitor模块**: 类似实验二中的监控程序, 但核心的代码改为使用C语言完成, 这里只用来通过调用C语言实现的函数。这里还可以加载文件属性的表单和在启动系统前完成一些批处理指令。
- iii) **kernal模块**: 用C语言和汇编语言实现的函数, 通过在C语言中调用汇编语言中写的I/O操作的基本库函数来实现内核主要功能。
- iv) **user模块**: 用户程序和一些批处理指令
- list模块**: 用户程序的文件表单, 用来实现简单的文件系统

### 3.4 实验过程:

#### 1) 设计loader模块:

因为内核的文件大小远大于一个扇区, 所以不能直接放在首扇区启动裸机, 我们需要一个引导程序通过从磁盘中把内核加载到内存单元07c00的位置, 再使用jmp语句跳转执行内核的程序代码。具体设计程序如下图:

```
org 7c00h ; BIOS将把引导扇区加载到0:7C00h处,
OS_offset equ 0A100h
;此程序是用来加载操作系统内核进入内存并跳转运行!

ReadOs:
    mov ax,cs ;段地址 ; 存放数据的段地址
    mov es,ax ;设置段地址 (不能直接设置)
    mov bx, OS_offset ;存放内核的内存偏移地址OS_offset
    mov ah,2 ;功能号
    mov al,4 ;扇区数, 内核占用扇区
    mov dl,0 ;驱动器号 ; 软盘为0, 硬盘为1
    mov dh,0 ;磁头号 ; 起始编号为0
    mov ch,0 ;柱面号 ; 起始编号为0
    mov cl,3 ;存放内核的起始扇区号
    int 13H ;调用读磁盘BIOS的int 13H
    jmp 0a00h:100h ;控制权移交给内核

times 510 - ($ - $$) db 0 ;将前510字节不是0就填0
db 0x55
db 0xaa
```

Figure 5: loader模块代码

#### 2) 设计内核:

内核主要分为三个部分:

- **第一部分是主程序(monitor.asm)**, 它是由汇编语言编写, 我们将使用它来调用另外两个包含了自己定义编写的库函数文件来运行操作系统。

- 第二部分就是用Turbo C语言实现的库函数(**kernal.c**)文件，它是内核中最核心的部分,主要是使用它来实现操作系统主画面内容的输出，同时它还可以处理单条指令或批处理指令完成相应的操作。
- 第三部分是使用汇编语言实现的最底层的函数库(**syscall.asm**)，它完成最基础的和显示器，内存，磁盘的交互，所有C库中的代码都要依靠调用它的函数来实现。

### 接下来就介绍内核中的代码段和实现的操作系统功能

#### i) 调用主函数(**monitor.asm**):

因为不像我们平时打的C语言，有main函数作为主函数，所以这次我将main函数写成了库函数里的一部分，通过在汇编语言的代码前引用外部函数 **extern \_cmain:near**，再使用语句 **call near ptr \_cmain** 来调用cmain函数来进入操作系统主界面。

同时这个monitor.asm文件也可以在调用cmain函数进入操作系统前执行一些其他的程序段。如仿照真正的计算机执行一段硬件的自检测程序，或者是一段批处理程序。代码示例如图：

```
start:
    mov ax,cs
    mov ds,ax; DS = CS
    mov es,ax; ES = CS
    mov ss,ax; SS = cs
    mov sp, 0FFF0h
    mov ah,2
    mov bh,0
    mov dx,0
    int 10h

    call near ptr _cmain ;运行系统
    jmp $
include syscall.asm
```

Figure 6: monitor.asm

#### ii) 底层硬件交互库函数(**syscall.asm**)

syscall中实现了一些可以让C函数调用的底层硬件交互函数，所有的C函数都是建立在这个库的基础上运行的，下面列举了一些函数并选取重要函数说明。这个库包括了：

- 清空显示屏函数 **\_cls**
- 从键盘读入单个字符函数 **\_ReadChar**

- 在显示器上显示单个字符函数 `_printChar`
- 从软盘中加载用户程序并运行 `_RunProm`
- 从软盘中加载文件表单 `_loadListFromDisk`
- 从软盘中加载批处理文件 `_loadbatchFromDisk`
- 从文件表单中读取数据 `_readList`
- 从批处理文件中读取字符串 `_readbatch`

**读字符和输出字符的函数：**输入使用了BIOS提供的int16号中断指令，在这里我们使用一个外部变量 `_in` 来装我们的输入字符，在C库中我们就可以直接调用这个变量 `in` 来使用刚刚输入的字符。输出字符使用int 10的0eh号功能，它可以输出字符并自动移动光标到下一格，这样就不用我们自己去维护光标的坐标，`[bp+4]` 是因为 `sp` 不能寻址只能把值先给 `bp`，而传入的参数因为有 `IP` 和刚刚压入保护的 `bp`，所以所在的位置为 `[bp+4]`。代码如下图：

```
public _Readchar
_Readchar proc
;输入单个到字符到a1
    mov ah,0
    int 16h
    mov byte ptr [_in],al
    ret
_Readchar endp

;输出字符
public _printChar
_printChar proc
    push bp
    mov bp,sp
    mov al,[bp+4];char\ip\bp bp才能寻址
    mov bl,0
    mov ah,0eh
    int 10h
    mov sp,bp
    pop bp
    ret
_printChar endp
```

Figure 7: 输入和输出字符

**从内存中读取数据：**我们先通过很熟悉的从软盘加载进内存的INT 13H中断功能来将文件表单和批处理文件从指定扇区加载进入内存中，接着我们求出我们要的数据所在的内存地址偏移量，然后将这个偏移量给到一个外部变量——字符指针类型的 `_str`，然后在C中我们就可以直接对应这个地址去该内存单元中的字符串或者数值了。代码如下图：

iii) 操作系统主界面和帮助菜单(kernal.c)

操作系统主界面展示了欢迎语句，作者的姓名学号和提示信息。当你输

```
public _readbatch
_readbatch proc
    push ax
    mov ax,08c00h
    mov word ptr [_str],ax
    pop ax
    ret
_readbatch endp

public _readList
_readList proc
    push bp
    mov bp,sp
    mov al,[bp+4]
    add ax,9c00h
    mov word ptr [_str],ax
    pop bp
_readList endp
```

Figure 8: 读文件表单和批处理文件

入-h命令的时候，你就会看到此操作系统所支持的所有命令格式。

- h: 输出帮助菜单,如图9。
- cls: 清空屏幕。
- p1: 运行软盘中的第一个用户程序，-p2即为运行第二个依次类推
- ls: 列出存储在第二个扇区中的表达，包含所有用户程序的名字、文件大小和所在扇区号的信息，后面在简单文件系统 板块还会提到，如图10。
- b: 进入批处理模式。当你进入批处理模式的时候，你可以一次性输入多条指令并按顺序依次执行，但要注意每个命令用空格隔开，如下图11。
- q: 退出程序。退出cmain函数中的主循环，并显示退出的语句，如下图12。

```
=====
:      Welcome to my OS!      :
:      Build on March 25 2018  :
:      Author:Ye Shengyuan & Student Number:17341198  :
=====
Please enter instruction below. Enter -h to view instruction list

>>-h
-h:view instruction list
-cls:clear the display
-p1:run the user program 1
-p2:run the user program 2
-p3:run the user program 3
-p4:run the user program 4
-p5:run the user program 5
-ls:show the file list
-b:begin processing batch
-q:exit the operating system

>>-
```

Figure 9: 主界面和帮助菜单

```
=====
:      Welcome to my OS!      :
:      Build on March 25 2018  :
:      Author:Ye Shengyuan & Student Number:17341198  :
=====
Please enter instruction below. Enter -h to view instruction list

>>-ls
Total Files:5
FileName      FileSize
1.PerInfor    1708
2.stone       4838
3.square     3278
4.sandclock   2808
5.loading     4588

>>-
```

Figure 10: 文件列表



```
please enter your instructions queue.  
Please separate instructions by Spaces. Enter -qb to leave batch processing.  
-h -p1 -p2 -cls -qb
```

Figure 11: 批处理模式

```
Thank you for using!  
Bye-Bye!  
-
```

Figure 12: 退出界面

## iv) C实现的库函数(kernel.c)

kernel.c中实现了一些基于syscall库函数的一些拓展函数：

- 输出字符串printf(char\*)
- 判断字符串是否相等if\_equal(char\*,char\*)
- 从键盘读命令字符串ReadCommand()
- 运行命令函数RunCommand(char\*)
- 批处理函数batch(char\*)

**输出字符串和读入字符串：**我们在syscall函数中已经实现了最底层的输出一个字符和读入一个字符的函数，因此我们只需要循环调用这两个函数，就可以从读写字符变成读写字符串了，但要注意：**在字符串的最后别忘了添加一个'\0'作为字符串的终止符号**。在Readcommand函数中还进行了**退格(backspace)操作**的设计，因为裸机操作系统中的退格只会让光标后移，所以我用ASCII码是否为8判断是否为退格，再将前一个的填充白来实现。代码如下图：

```
void printf(char* str)
{
    int i=0;
    while(str[i])
    {
        printChar(str[i]);
        i++;
    }
}
```

Figure 13: 输出字符串

```
void Readcommand(char* command)
{
    int i=0;
    Readchar();
    while(in!=13)
    {
        if(in==8)
        {
            printChar(in);
            printChar(32);
            i--;
            printChar(in);
            Readchar();
            continue;
        }
        printChar(in);
        command[i]=in;
        i++;
        Readchar();
    }
    command[i]='\0';
}
```

Figure 14: 输入指令字符串

**运行指令和批处理：**当我们读到指令的字符串的时候，我们就可以将指令放入Runcommand函数中，函数里面会判断指令是在执行那一项操作，利用ifelse语句进入分支跳转。而批处理函数batch则是读入一个长的指令串，指令用空格隔开，遇到空格就立刻将已经读到缓冲区的字符串放入Runcommand中执行。结合素后再清空缓冲区继续查找下面的子指令(subcommand)知道所有指令执行完成。代码示例如下：

```
void runcommand(char * com)
{
    if(if_equal(com, "-h"))
    {
        help();
    }
    else if(if_equal(com, "-cls"))
    {
        cls();
    }
    else if(if_equal(com, "-p1"))
    {
        cls();
        readlist(13);
        RunProm(*str);
        cls();
    }
    else if(if_equal(com, "-p2"))
    {
        cls();
        readlist(25);
        RunProm(*str);
        cls();
    }
    else if(if_equal(com, "-p3"))
    {
        cls();
        readlist(38);
        RunProm(*str);
        cls();
    }
}
```

Figure 15: 运行指令

```
batch(char * com){
    char subcommand[10];
    flag=1;
    i=0;
    j=0;
    leave_batch=1;
    while(com[i]!='\0' && flag && !leave_batch)
    {
        if(com[i]!=' ')
        {
            subcommand[j]=com[i];
            i++;
            j++;
        }
        else
        {
            subcommand[j]='\0';
            runcommand(subcommand);
            i++;
            j=0;
        }
    }
    if(com[i]=='\0')
    {
        subcommand[j]='\0';
        runcommand(subcommand);
    }
}
```

Figure 16: 批处理

**主程序cmain：**主程序是整个操作系统的主线，在main函数中，我先将文件表单和批处理文件加载进了内存指定的扇区中，然后读取批处理文件，在进入系统前先执行一段批处理指令序列（这个序列可以由用户自己修改batch文件中的内容）执行完后清屏并输出主菜单。同时利用变量flag来判断是否退出程序。当用户输入了退出指令，flag会被修改为0，于是循环停止，输出退出界面的信息，并结束操作系统。如下图：

```
cmain(){
    loadlistFromDisk();
    loadbatchFromDisk();
    readbatch();
    cls();
    batch(str);
    Menu();
    flag=1;
    while(flag)
    {
        printf("\n\nr>>");
        Readcommand(command);
        runcommand(command);
    }
    cls();
    printf("\n\nThank you for using!\n\nBye-Bye!\n\nr");
    Readchar();
}
```

Figure 17: 主程序cmain

## 3) 设计用户程序

用户程序沿用了实验二中所编写的四个用户程序，但因为我们在内核中调用用户程序之后需要在执行完后返回内核，所以我们使用了 `call Offset` 取代了原来的 `jmp Offset`，我们的用户程序返回方式也要随之更改：由原来的 `jmp 07c00` 更改为 `ret`，

```
127     mov ah,1
128     int 16h
129     mov bl,20h
130     cmp al,bl
131     jz Quit
132
133     jmp loop1
134
135 Quit:
136     ret
```

Figure 18: 使用ret返回

恰巧因为看到其他同学的用户程序受到启发，联想到这次实验要求的在加载系统前运行一段批处理程序，发现其实可以设计一个开机界面的用户程序，在启动系统前使用批处理执行，这样就达到了一种仿佛真的在打开一台古老的计算机的效果，于是设计了一个新的用户程序。当loading进度条满的时候，用户程序自动退出并自动进入操作系统。下图展示开机过程：



Figure 19: Loading 1



Figure 20: Loading 2



Figure 21: Loading 3

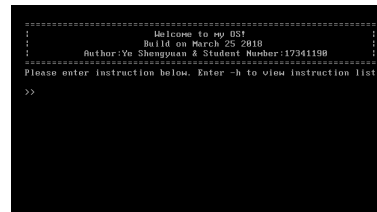
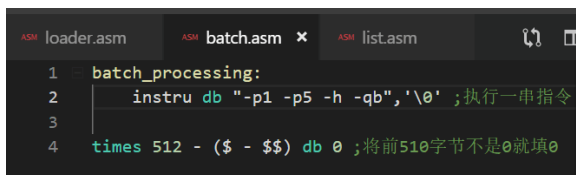


Figure 22: Loading 4

## 4) 简单批处理系统

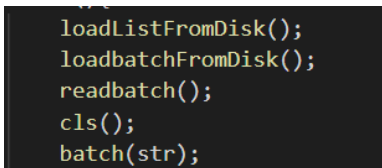
建立一个batch.asm文件用来装用户自己设计的批处理指令序列，要求每个子指令用空格隔开，并在最后结束指令序列后加入一个'\0',如下图：



```
1 batch_processing:
2   instru db "-p1 -p5 -h -qb", '\0' ;执行一串指令
3
4   times 512 - ($ - $$) db 0 ;将前510字节不是0就填0
```

Figure 23: batch.asm

使用nasm编译这个文件，并将生成的bin文件内容添加到软盘的其中一个扇区(实验中我将批处理文件放在了第十三个扇区),在cmain函数刚开始的位置，我们调用loadbatchFromDisk函数将这个扇区中的代码调入内存的其中一个区域，并使用readbatch函数获取这个区域的起始地址放入全局变量str中，这样就可以在C中把str当作一个字符串直接传入函数batch中运行了。代码如下图：



```
loadListFromDisk();
loadbatchFromDisk();
readbatch();
cls();
batch(str);
```

Figure 24: 从软盘中读批处理指令

## 5) 简单文件系统

创建一个list.asm表单，用来存放所有用户程序的信息。首先第一个字节是用户程序的总数，然后一个用户程序在一个程序段内，分别定义文件名，文件大小和文件存放的扇区号，中间定义一个字节的0隔开，用来方便C函数中的字符串操作，如下图25：

要使用这个表单来读取用户程序所在的扇区号和要加载的扇区数目，通过对齐，可以很好确定扇区号所在地址的偏移量，于是我们调用readList函数，传入偏移量，就可以获得扇区号，再通过调用RunProm函数传入扇区号作为参数，就可以运行该扇区的用户程序。代码如下图26：

```

nums db 5 ;文件数量
program1:
    name1 db "PerInfor" ;文件名字
    db 0
    size1 db "170" ;文件大小单位字节
    db 0
    sector1 db 8 ;扇区号
    db 0
program2:
    name2 db "stone" ;文件名字
    db 0
    size2 db "403" ;文件大小单位字节
    db 0
    sector2 db 9 ;扇区号
    db 0
program3:
    name3 db "square" ;文件名字
    db 0
    size3 db "327" ;文件大小单位字节
    db 0
    sector3 db 10 ;扇区号
    db 0

```

Figure 25: list.asm

```

else if(if_equal(com,"-p1"))
{
    cls();
    readList(14);
    RunProm((*str));
    cls();
}
else if(if_equal(com,"-p2"))
{
    cls();
    readList(26);
    RunProm((*str));
    cls();
}
else if(if_equal(com,"-p3"))
{
    cls();
    readList(39);
    RunProm((*str));
    cls();
}

```

Figure 26: 用List调用用户程序

当调用-ls命令的时候，需要要将List.asm文件中的文件数量，文件名字，文件大小等属性输出到显示器上。用前文的方法，我们先找到list.asm所在内存的首地址，先读取它的文件总数，然后以文件总数作为循环次数，每次都读取文件的文件名字和文件大小并输出出来，如下图展示：

```

printf("\n\rTotal Files:");
printf(count*0');
printf("\n\r FileName      FileSize\n\r");
while(count-->0)
{
    j=0;
    for(x=0;x<9;x++)
    {
        name[x]=32;
        size[x]=32;
    }
    while(str[i]!=0)
    {
        name[j]=str[i];
        j++;
        i++;
    }
    i++,j=0;
    while(str[i]!=0)
    {
        size[j]=str[i];
        j++;
        i++;
    }
    size[j]='0';
    i=i+3;
    printf(6-count-1*0');
    printf(". ");
    printf(name);
    printf(" ");
    printf(size);
    printf("\n\r");
}

```

Figure 27: 读取文件列表信息

## 6) 用软盘启动裸机:

将内核用tcc+tasm+tlink编译链接成.com文件, 再把5个用户程序、list、batch和loader用nasm编译形成bin文件, 加入到1.44MB软盘的合适扇区中并运行程序, 虚拟机运行如下图:



Figure 28: 开机界面1

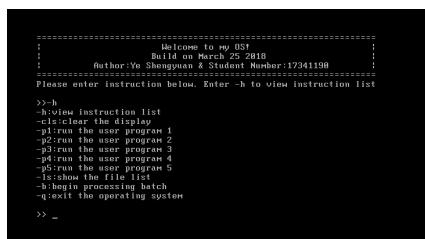


Figure 29: 操作系统界面

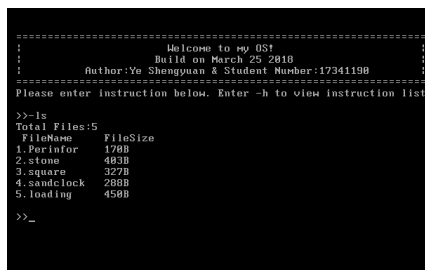


Figure 30: 文件列表界面

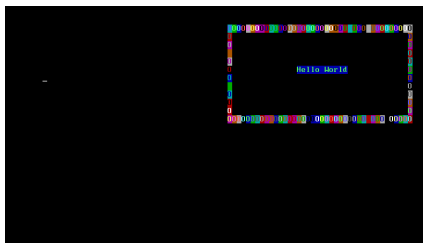


Figure 31: 用户程序界面



Figure 32: 退出界面

## 4 实验心得和总结：

这次实验是真的异常的艰难，因为时间有限而且难度比较大，花费了十分长的时间。由于从来没有接触过C和汇编混编的操作，而且是十分古老的C语言和汇编语言，还要专门提前去学习了解语法，在编写的过程中也是频繁的遇到很多的问题，还好老师给了Tcc和Tasm等编译环境，省去了一些要在网上查找的时间。经过不懈努力，和同学们的一起讨论和互相帮助下终究是完成了实验。

实验的第一步就要升级实验二中设计的操作系统原型——监控程序，将他发展成为一个操作系统内核，因为汇编语言是十分难以编写，而且也很难读懂的，所以我们需要借助高级语言Turbo C的帮助来实现我们的内核系统。上次我们只需要按键1、2、3、4来选择要运行哪一个用户程序，这次我们是通过输入指令，例如-help -quit这样的字符串来执行操作，同时还要有批处理，即能连续运行多个指令的能力，这是汇编语言非常难以完成的，所以我们只有结合汇编语言和C语言，才能顺利的实现众多复杂的内核功能。但汇编和C语言混用又十分麻烦，就拿传递参数来说，就需要用到栈来传递，虽然复杂，但也让我理解到了以往被包装起来看不到的函数参数传递的真谛。但其实内核中要实现的C函数相比于我们算法设计课上做的习题来说，还是比较简单的，所以实现起来还是比较轻松，难就难在汇编语言的晦涩难懂上了。

本来做好了指令控制和批处理以为就完成了，后来助教又增大了难度，让我们从磁盘中读取批处理指令序列，还要建立一个文件列表，从文件列表里读扇区号来调用户程序执行，这有考验了我如何从软盘中把数据读到C语言中的

一个变量里。后来经过不断的尝试才发现我们可以把软盘的数据先读入内存，再通过内存中的偏移地址传入变量指针中，就可以用字符串函数来处理了，这看似简单，但其实遇到了很多的bug，而我们又没有去下载汇编语言的调试环境，所以只能一直的用int10中断输出调试，因此花费了非常多的时间。

这次实验因为能力和时间有限，其实还有很多地方可以更进一步，比如批处理文件batch其实也可以当做用户程序文件加入文件列表，用扇区号载入运行，这点并没有完善。同时操作系统中有很多键盘输入的地方没有进行封装，如果输入错误了很容易造成死机或者程序无法正常退出等问题，这都是需要进一步改进的。虽然难度大，花费时间长，但终究还是有收获，看着自己做出来的简单操作系统还是很有成功感，期望下一个实验能继续学到更多新的知识。

## 5 参考文献:

---

- 1) nasm手册
- 2) <https://blog.csdn.net/longintchar/article/details/79511747>
- 3) <https://blog.csdn.net/cielozhang/article/details/6171783/>
- 4) [https://blog.csdn.net/lulipeng\\_cpp/article/details/8161982](https://blog.csdn.net/lulipeng_cpp/article/details/8161982)
- 5) <https://www.cnblogs.com/alwaysking/p/7789282.html>
- 6) <https://blog.csdn.net/cielozhang/article/details/6171783>
- 7) [https://blog.csdn.net/lulipeng\\_cpp/article/details/8161982](https://blog.csdn.net/lulipeng_cpp/article/details/8161982)