实验报告

学院：数据科学与计算机学院　专业：计算机科学与技术　年级：2016 级
实验人姓名（学号）：王锡淮（16337236）
日期：　2018 年　3 月 24 日

实验 3 开发独立内核的操作系统

[实验目的]

1. 把原来在引导扇区中实现的监控程序(内核)分离成一个独立的执行体，存放在其它扇区中，为“后来“扩展内核提供发展空间。
2. 学习汇编与 c 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，增加实现实验要求 2 中的部分或全部功能。

[实验要求]

1. 编写引导程序和操作系统内核。
2. 该操作系统内核可以加载多个用户程序，并且和汇编模块相互调用。
3. 内核应具有以下功能：
   a. 在磁盘上建立一个表，记录用户程序的存储安排。
   b. 可以在控制台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等。
   c. 设计一种命令，并能在控制台发出命令，执行用户程序。
4. 规定时间内单独完成实验。
5. 实验三必须在实验二基础上进行，保留或扩展原有功能，实现部分新增功能。
6. 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星，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。
7. 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。
8. 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。
9. 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。

[实验内容]

1. 编写了一个命令行系统雏形，其具有以下指令：
   a. Ls 指令：查询程序的信息，包括程序名、字节数大小、在磁盘中所处的起始位置、文件类型（文档、可执行程序、文件夹）。
   b. Date 指令：显示当前的系统日期。
   c. Clear 指令：刷新屏幕。
   d. Man 指令：查看用户帮助文件。
   e. Exit 指令：退出操作系统.

f. reboot 指令：重启操作系统。
g. type 指令：查看文件内容。
h. rm 指令：删除文件。
i. ./+程序：运行用户程序。

2. 处理了内存的分布问题。
3. 实现一个文件系统雏形，具备删除、读取、执行文件的功能。
4. 了解磁盘上扇区的分布情况。
5. 带有屏幕自动滚动功能。
6. 带有回显、退格功能的键盘输入。

[实验方案]

实验工具以及环境

工具链：sublime text 3 编写代码，nasm 汇编.asm 文件，gcc 将编译进行到汇编这一步，ld 完成最后的链接，以实现 c 和汇编语言混合编译，然后是 bochs 调试，以及最后的 vmware 进行展示。此外，安装了 dd for windows，以及 ld 连接器等 linux 下经过修改能在 windows 下运行的小工具。

实验环境：windows10 系统。

方案思想

该操作系统雏形的设计参考的是 windows 下的 powershell 和 linux 下的 bash，实现的是最简单的命令行指令，界面设计力求方便简洁。而进程设计则仍是采用的最简单的未运行态和运行态两态模型，而系统内也于实验 2 中保持一致，使用的是基于进程的系统内核，文件信息既可以采用放在磁盘中待程序读取并进行序列化，也可以硬编码在程序中，因为在编写 man 指令的时候就已经尝试过了前者序列化和反序列化的过程，于是在第一版中采用的是在程序中硬编码的方法，在第二版中改成了从磁盘中读取的方法。用户程序也是基本使用的实验 2 中的用户程序。

本次实验的重点在于 c 语言和汇编语言的混合编译以及独立系统内核的编写，以及了解基本的编译原理、程序运行时的内存分布和磁盘的扇区分布。

实验原理

如何将 C 语言和汇编语言混合编译

在本次实验中使用的 GCC 编译器和 NASM 汇编器以及 ld 链接器。

想要使用 gcc 和 nasm 混合编译，首先了解二者的输出文件，nasm 的默认输出是 16 位的程序，而 gcc 的默认输出是 32 位的程序，而我们知道编写 32 位的程序需要进入保护模式，但是本次实验要在实模式下完成，于是需要让最终的程序输出为 16 位程序。

于是需要了解让 gcc 编译产生 16 位代码的方法，查看 gcc 的用户手册[gcc]发现了一些指令，

[-march]指令，可以指定机器类型，使用-march=i386 可以指定使用的是 intel i386CPU 指令体系。

[-m32]指令，该指令指示 int，long 还有指针类型的长度为 32 位，并且生成的程序在 i386 机器上运行。

[-mpreferred-stack-boundary=num] 该指令使栈 4 字节对齐，这样方便 16 位和 32 位的数据共存于栈。

[-ffreestanding]指令，由于我们编写出来的程序是要在只有 bios 的裸机上运行的，所以是没有任何库能够使用的，于是用该指令显示地指明这一点。

[-c]指令，该指令使得 gcc 在编译过程“预处理-编译-汇编-链接”中执行到汇编这一步，输出.o 文件。

除了在使用 gcc 时的选项设置之外，还要在.c 文件开头加__asm__(".code16gcc\n");语句，该语句中的__asm__是 gcc 中的基本汇编使用方法，而.code16gcc 指示 gcc 在汇编这一步生成 16 位代码。（其实以上-m32 选项和__asm__(".code16gcc\n");语句的功能等于-m16 选项）。

那么 nasm 的选项如何设置呢？因为 gcc 的默认输出文件是 elf 格式的，所以 nasm 文件也应输出 elf 格式的 32 位文件。使用的是[-f elf32]选项。

除此之外，还要在每个汇编文件首部加上[BITS 16]指示 nasm 该汇编文件是 16 位的。在该模式下，使用 32 位数据指令会被加上 0x66 前缀，指向 32 位的地址也被加上 0x67 前缀。

而至于链接这一步，使用 ld 工具，查看用户手册可知:

[-m i386pe]指示 ld 目标机器是 i386pe

[-N] 是让.data 和.text 段不对齐

[-Ttext]和[-Tdata]的功能相当于 org 指令

以上内容参考自一个 github 项目[5]。

在看这段描述的时候可能产生疑惑，就是为什么要在命令行里指示生成 32 位，然后又在代码文本里面设置生成 16 位代码呢？其原因是我们在编写汇编代码的时候习惯于使用 16 位的写法，而在编写 c 语言时许多变量又是 32 位的，为了兼容，以上操作使得代码变为在 nasm 中使用 16 位变量而在 gcc 中使用 32 位变量的 16 位程序。

详细用法可以看我的 run.bat 批处理文件，展示如下：

```
.\nasm.exe -f bin .\loader.asm -o loader.bin > log.txt
.\nasm.exe -f bin .\ball_A.asm -o ball_A.bin > log.txt
.\nasm.exe -f bin .\ball_B.asm -o ball_B.bin > log.txt
.\nasm.exe -f bin .\ball_C.asm -o ball_C.bin > log.txt
.\nasm.exe -f bin .\ball_D.asm -o ball_D.bin > log.txt
.\nasm.exe -f bin .\showBall.asm -o showBall.bin > log.txt
.\nasm.exe -f bin .\printnames.asm -o printnames.bin > log.txt
.\nasm.exe -f bin .\cleanPrint.asm -o cleanPrint.bin > log.txt
.\nasm.exe -f bin .\checkinput.asm -o checkinput.bin > log.txt
.\nasm.exe -f bin .\printBigName.asm -o printBigName.bin > log.txt
.\nasm.exe -f elf32 .\utils.asm -o utils.o > log.txt
gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -c .\os.c -o os.o > log.txt
gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -c .\utilsC.c -o utilsC.o > log.txt
rem ld -m i386pe -N .\os.o .\utils.o .\utilsC.o  -Ttext 0x7f00 -Tdata 0x9900 -o boot.tmp > log.txt
ld -m i386pe -N .\os.o .\utils.o .\utilsC.o  -T linkscript -o boot.tmp > log.txt
objcopy -O binary boot.tmp boot.bin > log.txt
```

如何编写汇编函数以便 C 语言调用（X86 函数调用约定）

CC 遵循 cdecl（C declaration，即 C 声明）是 C 语言的一种调用约定，在 X86 架构上，其内容包括：

1. 函数实参在线程栈上按照从右至左的顺序依次压栈。
2. 函数结果保存在寄存器 EAX/AX/AL 中
3. 浮点型结果存放在寄存器 ST0 中
4. 编译后的函数名前缀以一个下划线字符
5. 调用者负责从线程栈中弹出实参（即清栈）
6. 8 比特或者 16 比特长的整形实参提升为 32 比特长。
7. 受到函数调用影响的寄存器（volatile registers）：EAX, ECX, EDX, ST0 - ST7, ES, GS
8. 不受函数调用影响的寄存器： EBX, EBP, ESP, EDI, ESI, CS, DS

9. RET 指令从函数被调用者返回到调用者（实质上是读取寄存器 EBP 所指的线程栈之处保存的函数返回 地址并加载到 IP 寄存器）

除了以上内容，通过观察反汇编结果可以发现，gcc 编译出来的汇编代码在调用 C 语言编写的程序时的确遵循了上述约定。会在函数被调用者其流程为：

1. 将 ebp 压入栈中。
2. 将 esp 的值赋给 ebp。
3. 减小 esp，申请栈空间。
4. 将参数从右往左压入栈中。
5. 执行函数流程。
6. 将 ebp 的值弹出。
7. 将 ebp 的值赋给 esp。
8. 返回调用者。

对于 ebp 和 esp 的值的操作的目的在于分开在函数内部使用的局部参数和调用者申请的参数。

但是在调用汇编语言编写的程序时，gcc 在调用者部分所做工作为：

1. 然后将函数从右向左压入栈中。
2. 随后用 call 指令调用函数。

并无保存 ebp 寄存器的值和恢复 ebp 寄存器的值的过程。所以在被调用的函数中自己注意维护栈，自己注意寻找返回地址的位置。

于是，如果要使用汇编编写 C 语言可以调用的函数，需要注意

1. 函数名添加下划线。
2. 返回值保存在 eax 中。
3. 注意栈中返回地址在哪个位置。
4. 要使用传递的参数，需要自行在栈中寻找，同时需要注意参数对应大小。

如何在汇编代码中调用 C 语言的函数

如上一部分所言，函数调用者要完成的操作是在栈中申请足够大的空间，然后将参数从右往左压入栈中，并且在函数返回后清空栈中的参数。所以要在汇编代码中调用 C 语言的函数需要做的是：

1. 申请栈空间
2. 参数从右往左压入栈中
3. 调用函数
4. 清空栈中的函数

如何调用 BIOS 使得计算机关闭[2][7]

要使得计算机通过中断关闭，可以按顺序调用 int 15h 来关闭本系统服务、驱动器、其余所有的服务，相应的中断如下：

```
mov ax, 5301h ;function 5301h
xor bx, bx ;device id: 0000h (=system bios)
int 15h ;call interrupt: 15h

mov ax, 530eh ;function 530eh
mov cx, 0102h ;driver version
int 15h ;call interrupt: 15h

mov ax, 5307h ;function 5307h
```

```
mov bl, 01h ;device id: 0001h (=all devices)
mov cx, 0003h ;power state: 0003h (=off)
int 15h ;call interrupt: 15h
```

如何获取并输出系统时间

获得系统时间是一件较容易的是，既可以从 CMOS 的 RAM 里面获得[1]，也可以调用其中断 int 1ah/04h 获得，但是在将获得的 BCD 码转变成字符串输出的时候就要注意在 X86 中采用的是小端方式存放数据，低位字节在高位地址。

在磁盘中扇区的排列

在本次实验中要使用较多的扇区，必须知道扇区的排列方式[8]。

首先了解一下软盘的物理以及逻辑结构

以一块软盘为例，一块软盘有两个面，正面和反面（在中断调用中是两个磁头），每个面被 80 个圆环分成 80 个磁道（也叫柱面）（1-80），每个磁道由 18 个扇区（0-17）组成（注意在中断调用中标号从 1 开始），每个扇区的大小为 512 字节。所以一块软盘的大小为 2*80*18*512 = 1474560 字节 = 1440KB = 1.44M。

以下是举例：

物理地址	逻辑扇区地址
0 面 0 磁道 1 扇区	0
0 面 0 磁道 2 扇区	1
:	:
0 面 0 磁道 18 扇区	17
1 面 0 磁道 1 扇区	18
1 面 0 磁道 2 扇区	19
:	:
1 面 0 磁道 18 扇区	35
:	:

我们把逻辑扇区地址可以看成大小为 1.44M 的数组的下标，每个数组元素的大小为 512 字节。

所以我们希望能根据 面 磁道 扇区 找到和 逻辑扇区的对应的关系，这样就大大方便了我们的调用中断。

经过观察与验证，逻辑扇区（相对扇区）号与磁面、磁道、扇区的关系是：

设相对扇区号为 x（从 0 开始）

$$\frac{x}{\text{每磁道扇区数}} \rightarrow \begin{cases} \text{商 y} \rightarrow \begin{cases} \text{磁道号} = y >> 1 \\ \text{磁面号} = y \,\&\, 1 \end{cases} \\ \text{余数 z} \rightarrow \text{起始扇区号} = z+1 \end{cases}$$

操作系统运行时内存的分配

在编写本次实验时遇到了内存的分配问题，所以应当知晓内存中各部分的用处，如下图：

于是我们可以使用就只有自由内存区，同时还要注意地址不能超出 16 位。在本次实验中使用第一个自由内存区，即从 0x0500 开始存放系统内核和系统例程，而将用户程序放在 0xc200 开始的地方。

内存和磁盘的分配方案为：

name	size	lma	type
os	8704	512	exec
loader	512	0	exec
system routine	2048	9216	exec
ball_A	512	18432	exec
ball_B	512	18944	exec
ball_C	512	19456	exec
ball_D	512	19968	exec
printBigname	3072	20480	exec
manual.txt	1024	23352	document

如何将用户程序的信息放在磁盘内并读取：

这是一个序列化和反序列化的过程，再本实验中我的采取的序列化方案是

1. 再同一条文件信息中，以“|”将不同类型的信息隔开。
2. 再不同的文件信息中，以'\n'（注意选择 unix 换行结尾）隔开。
3. 最后以两个连续的'\n'标志文件结束。

详细样例如下：

```
|ball_A|512|18432|2|
|ball_B|512|18944|2|
|ball_C|512|19456|2|
|ball_D|512|19968|2|
|printBigname|3072|20480|2|
|manual.txt|3072|24576|1|
```

知道了序列化方案之后，反序列化就较为简单了。

文件系统雏形：

本实验中的文件操作只有读取（type 指令）和执行（./指令），所以需要的文件系统也较为简单，于是设计文件系统如下：

1. 以文件名为关键字（key），文件信息为记录（record），建立散列表作为文件目录，文件的组织方式为顺序排放。
2. 散列函数选择为 hash(str) = Σweight*str[i]，探测方法选择为平方探测法。
3. 每一条文件记录含有文件名、文件大小、文件在磁盘中的位置、文件类型。

这样就有了文件目录和文件组织方式，可以实现文件的读取和执行基本功能。

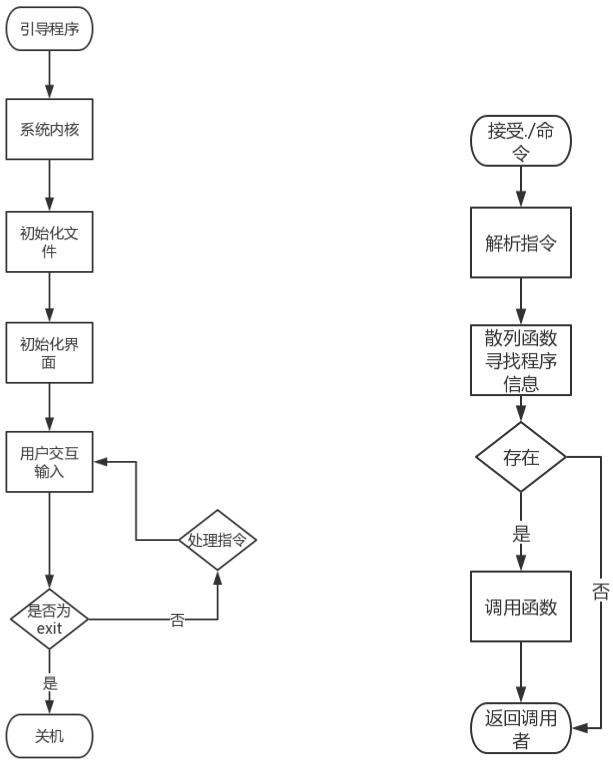
如何删除文件：

在磁盘清空文件比较麻烦和复杂，所以在本次实验中采用懒惰删除的办法，在每一条消息记录里边添加一个 deleted 标记，指示该文件是否被删除。

这样删除文件就变成了一个改变一个标记变量的过程，当要对文件进行其他处理，比如读取，执行，删除等等时，注意判断这一标记就比较方便了。

程序流程和算法思想

程序流程图如下：



在各个指令中，调用用户程序指令执行流程较为复杂，其流程图上方右侧图。

程序的流程设计来源于对于 windows 下的 powershell 和 linux 下的 bash 的使用经验，指令设计也是取材于最常用的指令。

数据结构与程序模块功能

Os.asm 中的系统内核主要部分：

```
int _main() {
    initialFile();
    initialScreen(1);
    char * in;
    struct info tmp = no;
    do {
        in = input();
        if(in[0] == '.' && in[1] == '/')
        {
            tmp = find(in+2);
            if(tmp.type != null){
                dispatch(tmp.lmaddress, tmp.size);
                clear();
                tmp = no;
            }
        }
        else {
            if(strcmp(in, rebootSen) == 0)
                reboot();
            if(strcmp(in, clearSen) == 0)
                clear();
            if(strcmp(in, dateSen) == 0)
                date();
            if(strcmp(in, manSen) == 0)
                man();
            if(strcmp(in, lsSen) == 0)
                ls();
        }
        newline();
    }while(strcmp(in, exitSen) != 0);
    shutdown();
    return 0;
}
```

本次实验实现的内核功能较为简单，但是齐全，用简单的条件判断以及循环逻辑就可以实现。

汇编语言编写的函数库，汇编语言编写的函数目录如下，其中主要是 printSentence、ClearScreen、getInput、dispatch、getDate、roll 函数需要加以说明。

```
void printSentence(char * message, int x, int y, int len);
void ClearScreen();
char * getInput();
void shutdown();
void dispatch(int address, int size);
void reboot();
void clear();
char * getManual();
char * getDate();
void roll();
```

PrintSentence 函数：

```
_printSentence:
    ;;; _printSentence(message, dh, dl, len) ;;;;
    push ebp
    mov ax, cs
    mov ds, ax
    mov es, ax

    mov cx, word [esp+14h]
    mov dl, byte [esp+10h]; 列
    mov dh, byte [esp+0ch]; 行
    mov bp, word [esp+08h]
    mov al,1
    mov bh,0
    mov bl,0fh          ;白底黑字
    push cs
    pop es
    mov ah,13h
    int 10h

    pop ebp
    pop ecx
    jmp cx
```

该函数的功能是在 dhdl 指定的行列位置打印出 message 参数指定的字符串，打印长度指定为 len 参数，而其实现则主要是利用 int 10h/13h 中断。

ClearScrenn 函数：

```
_ClearScreen:
    ;;; _ClearScreen() ;;;;
    mov ah, 06h
    mov al, 0 ; 0 是清空屏幕
    mov bh, 0fh; 白底黑字
    mov ch, 0
    mov cl, 0
    mov dh, 24
    mov dl, 79
    int 10h

    pop ecx

    jmp cx
```

这清屏函数的实现方式和我在之前的两个实验中采用的不一样，之前采用的是暴力清屏的办法，往显存区域写内容，这个方法的缺点是光标不会跟着移动，所以采用了 in10h /06h 或 int10h / 07h 中断来实现。

getInput 函数，该函数参考自[1]，实现较长，分开说明：

```
_getInput:
    ;;; char * _getInput() ;;;
    ;;; 功能是从键盘读取输入，支持删除，回显，回车表示输入完毕 ;;;
    ; 读取输入
    ;       -是否为删除
    ;                   是-退栈，显示字符，注意向屏幕输出空格覆盖
    ;       -是否为回车
    ;                   是-字符串末尾补'0'，返回字符串地址到eax中

    ;       -是否为普通字符
    ;                   是-入栈，显示字符
    ; 最大长度为100
```

这是 getInput 函数的主要逻辑，其主逻辑部分如下图左侧图，其中使用了 101 个字节长的缓冲区和存储了子程序地址的子程序表，以及栈顶指针。该功能实现主要依赖将缓冲区当作栈的方法，对于普通的字符输入，将其压入栈顶，并将其在当前光标位置打印出来，随后将光标后移一位；对于特殊字符的输入，如果是回车键，则在字符

串的最后一位补上 0 作为字符结尾，并且将缓冲区的地址返回给调用者；而如果是退格键，除非栈空，否则将栈顶回退，并将光标回退一格（如果到了边界则换行），并采用直接写显存的方式将要删除的字符覆盖，下图右侧是详细的处理过程

```
mov word [top], 0
getStr:
mov ah, 0
int 16h
cmp al, 20h
jl nochar
mov ah, 0
call charstart
jmp getStr

nochar:
cmp ah, 0eh ;退格键的扫描码
je backspace
cmp ah, 1ch ;enter键的扫描码
je enter
jmp getStr

backspace:
mov ah, 1
call charstart
jmp getStr

enter:
mov al, 0
mov ah, 0
call charstart

sub eax, eax
mov ax, input

pop ecx

jmp cx
```

```
Charmini:
;子程序，ah为功能号，
; 0为字符入栈，此时al为字符
; 1为出栈，al为返回的字符
; 二者都改变回显
charstart:
    push bx
    push dx
    push di
    push es

    cmp ah, 1
    jg Charminiret
    mov bl, ah
    mov bh, 0
    add bx, bx
    jmp word [getInputTable+bx]
; 注意top所在字节不是需要的内容。
charpush:
    cmp byte [top], 100
    jge Charminiret
    mov bx, [top]
    inc byte [top]
    mov byte [input+bx], al
    call charshow
    jmp Charminiret

charpop:
    cmp byte [top], 0
    jle Charminiret
    dec byte [top]
    mov bx, [top]
    mov al, [input+bx]
    call coverLast
    jmp Charminiret
```

其中显示的处理函数如下：

```
; 中断int 10h/13h显示字符串
charshow:
    push ebp
    mov ah, 3h
    int 10h
    ;此时dh, dl为行列位置
    mov al, 1h ; 光标位置改变
    mov ah, 13h
    mov bh, 0
    mov bl, 0fh ;黑底白字
    mov cx, 1
    push cs
    pop es
    mov bp, input
    add bp, [top]
    dec bp
    int 10h
    pop ebp
    ret
```

```
coverLast: ;处理退格键，覆盖最后一个字符
    mov ax, 0b800h
    mov es, ax
    mov ah, 3h
    int 10h
    ;此时dh, dl为行列位置
    ; 退一格
    cmp dl, 0
    jg noEndofLine
    mov dl, 80
    dec dh
noEndofLine:
    dec dl
    mov ah, 2h
    mov bh, 0
    int 10h

    mov ax, 80
    mul dh
    mov dh, 0
    add ax, dx
    mov dx, 2
    mul dx
    mov bx, ax

    mov word [es:bx], 0f20h
    ret

Charminiret:
pop es
pop di
pop dx
pop bx

ret
```

Dispatch 函数：

```
_dispatch:
    ;;;  void dispatch(int address, int size)  ;;;;;;
    offsetOfUserPrg equ 0e000h
    push ebp
    mov ax, cs
    mov ds, ax
    mov es, ax                ;设置段地址（不能直接mov es,段地址）

    ; 计算扇区
    mov ax, word [esp+0x8]
    mov dx, 0
    mov cx, 512
    div cx ;现在ax里面是相对扇区号
    mov cx, 18
    div cx ; 余数在dx里面
    mov cl, dl
    inc cl ; 起始扇区号确定

    mov dx, ax
    shr ax, 1
    mov ch, al ;磁道号
    mov dh, dl
    and dh, 00000001b
    mov dl, 0   ; 磁面号

    ; 计算大小
    push cx
    mov ax, word [esp+0xe]
    mov cx, 512
    mov dx, 0
    div cx ;al已设定
    pop cx

    mov bx, offsetOfUserPrg;偏移地址
    mov ah, 2 ;功能号
    int 13h

    call bx

    pop ebp
    pop ecx
    jmp cx
```

这是一个进程分派器，接受程序在磁盘中的起始地址和程序大小，计算出所在的磁头号、磁道号以及扇区号，然后计算出扇区数目，将其加载至 0xe000 处，并且运行该函数，在函数退出后返回调用者。

计算磁头、磁道、扇区号的方法如原理部分所言。

Getdate 函数的功能是将系统的年月日时间转变成字符串来返回，一下以“年”为例子说明：

```
_getDate:
    ;;;; char * _getDate() ;;;;;;;;;;
    mov ah, 04h
    int 1ah

    ;年
    mov bl, cl
    mov al, bl
    and al, 11110000b
    mov cl, 4
    shr al, cl

    and bl, 00001111b
    mov ah, bl
    add ax, 3030h
    mov word [date+6], '20'
    mov word [date+8], ax
```

首先使用中断 int 1ah/04h 来获取时间，其中年份信息以 BCD 码的形式存储在 cl 中，要做的工作主要是将小端方式存放的年份变成正常格式，在以 ASCII 码的格式存入字符串。

Roll 函数的功能是当光标超出底部边界的时候将屏幕向上滚动一行，有两种实现方式，第一种是调用 bios 自带的中断以实现这样的效果，第二种是从显存区域将当前界面的显示内容拷贝，再在上移一行之后的地方输出。此处展示较为简洁的第一种方式

```
_roll:
    ;;;;;;;; void roll();;;;;;;;
    ;复制上一行
    push ebp
    mov ah, 06h
    mov al, 1
    mov bh, 0fh
    mov cx, 0000h
    mov dx, 184fh
    int 10h

    pop ebp
    pop ecx
    jmp cx
```

这里的方式是将整个界面向上滚动一行。

C 语言编写的函数库：

```c
#define Len 10

void initialScreen(int welcome);
int strlen(char * sen);
char* input();
int strcmp(char * l, char * r);
void man();
int countLines(char * sen);
void newline();
void date();
void strcpy(char * sour, char * des);
void ls();

enum fileType
{
    null=0, docu, exec, folder
};

struct info
{
    int lmaddress;
    int  size;
    char name[20];
    enum fileType type;
};

void initialFile();

int hashfun(char * key);

void hash(char * key, struct info record);

struct info find(char * key);

void int2str(int org, char * str);
```

其中需要说明的是模仿 C 标准库实现的 strlen、strcpy

Int2str 和 strcmp 等字符串处理函数，以及采用散列技术的文件系统雏形。

实现的字符串处理函数：

这四个字符串处理函数逻辑是常见的逻辑，同时还是 C 语言实现的，难度不高，只是有不少细节需要注意，比如字符串最后要加上 0 作为结尾。

采用散列技术的文件系统雏形：

在该文件系统中，文件的组织采用的是顺序文件组织，文件目录使用的是散列技术来存放和查找（名字，记录）对。

哈希函数如下图左侧所示，存放函数如下图右侧所示：

```c
int hashfun(char * key){
    int weight = 29;
    int i = 0;
    int ret = 0;
    while(key[i]){
        ret = (ret + key[i]*weight) % Len;
        ++i;
    }
    return ret;
}
```

```c
void hash(char * key, struct info record)
{
    int inicode = hashfun(key);
    int code = inicode, i = 1;
    while(information[code].type!=null){
        code = (inicode + i * i) % Len;
        ++i;
    }
    information[code] = record;
}
```

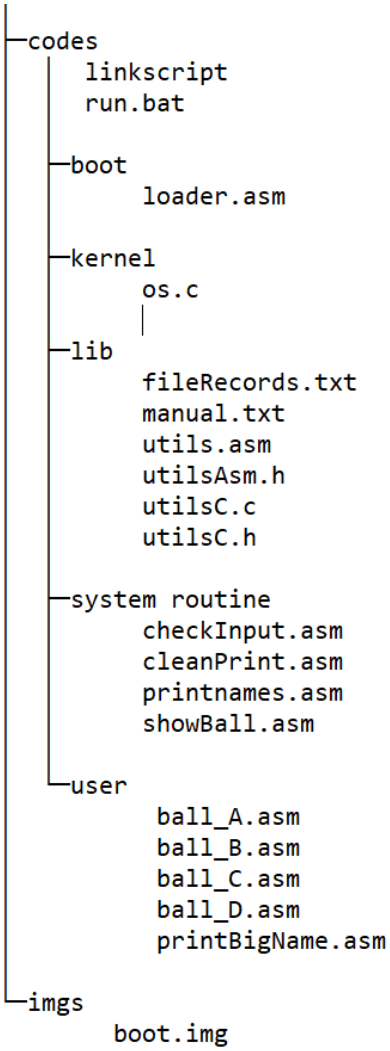
检索函数如下图所示：

```
struct info find(char * key)
{
    int inicode = hashfun(key);
    int code = inicode, i = 1;
    while(information[code].type == null || strcmp(key, information[code].name) != 0){
        code = (inicode + i * i) % Len;
        ++i;
        if(i > Len)
            return no;
    }
    return information[code];
}
```

至于序列化和反序列化，只需要在序列化的时候注意添加特殊字符和标记来表明数据类型和范围即可。

代码文档组成说明

提交的项目文件夹结构如下：



其中，在 codes 文件夹中，ball_A.asm，ball_B.asm，ball_C.asm，ball_D.asm，以及 printBigName.asm 是 5 个独立的用户程序，稍微修改自实验 2，其余的 checkInput.asm，cleanPrint.asm，printnames.asm，showBall.asm 是稍微修改的实验 2 中系统例程，而剩下的 loader.asm，os.c 分别是引导程序和系统内核程序，utils.asm，utilsAsm.h 是汇编语言实现的库函数，utilsC.c 和 utilsC.h 是 C 语言实现的库函数。最

后，manual.txt 是用户帮助文件，run.bat 是批处理文件，linkscript 是链接器 ld 的脚本配置文件，fileRecords.txt 是文件记录。

在 imgs 文件夹下，boot.img 是最后的系统镜像。

[实验过程]

主要流程

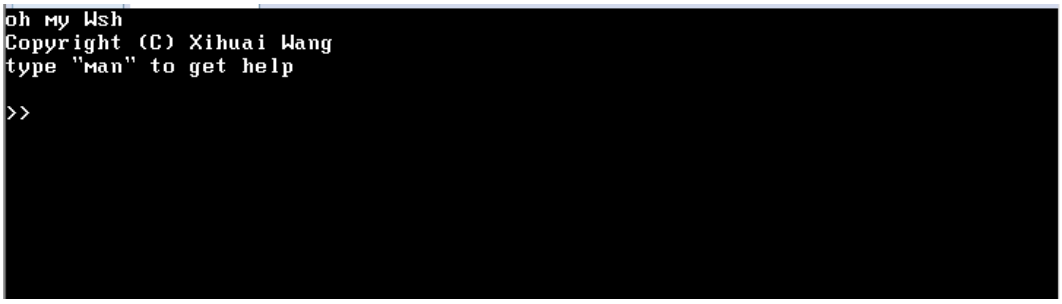
创建.img 虚拟软盘，编写汇编以及 C 语言的代码，其中 loader 和用户程序的代码使用 nasm 直接汇编出二进制文件，其余的和系统内核相关的代码使用[实验原理]部分所讲的混合编译技术得到二进制文件，再使用 bochs 调试，以及最后使用 vmware 展示结果。

输入输出说明

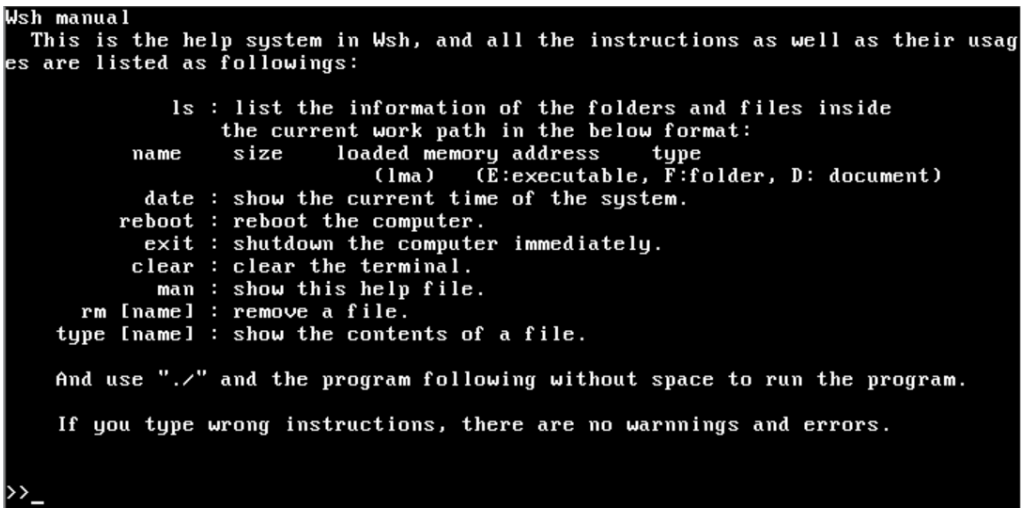
本次实验编写出来的操作系统是命令行形式的，其输入来自于用户的键盘输入，输出则为操作系统对于对应的用户输入的响应，包括了信息展示、过程状态的切换，以及程序调度。

结果展示

以下分别展示该操作系统的各个功能，首先是欢迎界面：



还有 man 指令显示的用户帮助文件：



然后是 ls 指令显示磁盘中的程序信息：

```
oh my Wsh
Copyright (C) Xihuai Wang
type "man" to get help

>>ls
         Name           size           Lma            Type
         printBigname   3072           20480          E
         ball_D         512            19968          E
         ball_C         512            19456          E
         ball_B         512            18944          E
         ball_A         512            18432          E
>>_
```

显示系统日期的 date 指令：

```
oh my Wsh
Copyright (C) Xihuai Wang
type "man" to get help

>>date
                              24/03/2018
>>_
```

清空屏幕的 clear 指令：

```
>>_
```

Type 指令展示：

```
>>type manual.txt
Wsh manual
  This is the help system in Wsh, and all the instructions as well as their usag
es are listed as followings:

            ls : list the information of the folders and files inside
                 the current work path in the below format:
         name    size    loaded memory address    type
                              (lma)   (E:executable, F:folder, D: document)
          date : show the current time of the system.
        reboot : reboot the computer.
          exit : shutdown the computer immediately.
         clear : clear the terminal.
           man : show this help file.
     rm [name] : remove a file.
   type [name] : show the contents of a file.

   And use "./" and the program following without space to run the program.

   If you type wrong instructions, there are no warnnings and errors.
```

Rm 指令展示：

删除 printBigname 前：

```
>>ls
         Name           Size           Lma            Type
         ball_C         512            19456          E
         manual.txt     3072           24576          D
         ball_D         512            19968          E
         ball_A         512            18432          E
         ball_B         512            18944          E
         printBigname   3072           20480          E
>>_
```

并且能够调用该子程序。

删除后：

```
>>rm printBigname
>>ls
        Name            Size            Lma             Type
        ball_C          512             19456           E
        manual.txt      3072            24576           D
        ball_D          512             19968           E
        ball_A          512             18432           E
        ball_B          512             18944           E
>>_
```

并且无法调用该子程序。

滚屏功能展示：

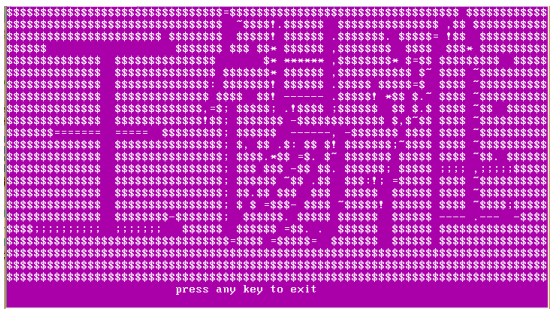
```
reboot : reboot the computer.
exit : shutdown the computer immediately.
clear : clear the terminal.
man : show this help file.

And use "./" and the program following without space to run the program.

If you type wrong instructions, there are no warnnings and errors.
>>
>>
>>
>>
>>
>>
>>
>>
>>
>>
>>
>>
>>
>>
>>
>>_
```

还有的关机和重启功能就不予展示了。

程序调用，分别是程序 ball_A 和 printBigname：



遇到的问题以及解决情况

1. 在本次实验中遇到的最大的问题自然是怎么样用 NASM 和 GCC 混合编译 C 语言和汇编语言。这个问题最终从一个 GitHub 项目里面找到了解决方案。其实回想起来，主要的问题就是要解决 GCC 和 NASM 汇编出来的文件的变量位数和地址位数的不匹配问题，以及注意 C 语言程序往栈中压入的变量位数和汇编语言

往栈中压入的变量位数不相同。而解决的方法最后都能从 GCC 和 NASM 以及最后的连接器的用户手册找到，但是问题在于用户手册涵盖的信息过于庞大，要自己一点一点地找到所需要的全部参数很困难。这一次的实验让我习惯于产看工具的用户手册，也知道了寻找解决问题的基本方案能够节省不少时间。

2. 在学习 X86 函数调用约定时，起初不能理解 enter 和 leave 的功能，在理解了 EBP 和 ESP 在调用函数时起到的作用之后，这些疑惑便引刃而解了。所以理解原理时解决问题的前提条件。
3. 在 X86 函数调用约定中，一般的返回值放在 eax 寄存器中，但是要注意的时如果返回的时汇编语言中的变量地址，要注意把高 16 位置 0，否者会出现访问越界的情况。
4. 在加载磁盘中的扇区时，由于没有注意逻辑（相对）扇区和磁面、磁道、扇区的对应关系，花费了不少时间和精力来调试，最后发现根本没有数据被加载，从中断的返回标志中找到了问题所在。
5. 理解错了滚动屏幕的中断的含义，然后自己实现了一遍滚屏功能。
6. 在函数运行时，esp 的值初始化为 0xffff，而我的内核放在了附近的位置，导致存储位置冲突，程序崩溃。将内核放到其它位置问题解决。
7. GCC 生成的二进制文件过于庞大，而加上-O 等优化选项后又出现了 bug。最后发现只在 os.c 编译时使用优化选项没有问题。
8. 在编写操作系统使用的 C 语言里面不能使用 malloc 函数，只能自己编写 malloc 或者使用静态数组，在实现文件系统雏形时没能使用链式结构，只是使用了哈希技术。
9. 输出的内容过多时无法完整显示，还没找到解决办法。
10. 不了解内存的分配，一开始将自己局限在很小的位置，后来才海阔天空。

[实验总结]

这次实验对我最大的提高就是知道了在汇编和 C 语言里面怎么样相互调用程序和变量。同时完成了一个具有齐全基本功能的命令行窗口也使我有了些成就感。但是同时也是十分挫败的，延长一周时间提交实验，这一周里我尝试了实现进程模式和完成具有修改和保存文件功能的更完整的文件系统，以及完成其余命令行功能，但是都没能完成，这些功能实现起来还需要懂得更多的原理，而这些原理正是我现在所欠缺的。

再总结一下我遇到的问题，除了艰难解决的混编兼容问题，尤为突出的是 gcc 汇编产生的二进制文件实在太大，让我的程序能够使用的内存减少不少，优化选项又会让程序出现问题，最后发现只能优化其中的 os.c 的汇编生成文件。还有就是相对扇区和绝对扇区的转化问题也困扰了我挺长时间。

我对于这个命令行还有不少想要实现的目标，比如接入鼠标，读取鼠标的数据进行交互，上下左右方向键分别有对应的功能：光标左右移动，历史命令切换，甚至是命令行补全功能。还有就是我的命令行会出现输出内容过多时显示不了的问题，比如写一个 more 指令。

而对于操作系统方面，我的设想是完成一个具有修改，读取，保存文件功能的文件系统，同时希望该文件系统中的文件目录能够组织成一棵树的形式，所以就要先学会怎么样写一个 malloc 函数，或者学会采用静态树。同时还涉及到跨段跳转的问题，这样才能够在较大的自由内存区放得下我的文件。并且能够进一

步重构我的代码，让代码提高复用性。还有就是我对利用时钟中断进行多任务多用户的操作系统产生了兴趣，并期待这一部分的学习。

在进程控制方面，我尝试了实现进程控制块和进程表，但是在保存和恢复上下文的这一部分被卡住了，不知道怎么做到将栈段改到内核栈或者用户栈或者进程表的同时保存或恢复变量。至少要学会实现挂起态和就绪态。

进一步学习的方向是进程控制和文件系统的理论和实现方法。我在不少书看到过这些内容，但是都是基于保护模式的，实在是看不懂，还要再找方法。

[参考文献]

[1]汇编语言（第三版），王爽.
[2] https://blog.csdn.net/weixin_37656939/article/details/79684611, BIOS 中断大全.
[3] https://zhuanlan.zhihu.com/p/28659560，程序段的划分
[4] https://github.com/Urinx/SomeCodes/tree/master/Asm/Boot，内存区域的划分
[5] https://github.com/richardtsai/homework，参考的项目
[6] https://zh.wikipedia.org/wiki/X86%E8%B0%83%E7%94%A8%E7%BA%A6%E5%AE%9A#cdecl, X86 调用约定
[7] https://blog.csdn.net/wbcuc/article/details/7873314，调用中断实现关闭计算机
[8] https://blog.csdn.net/beyondxj/article/details/77104988，模拟软盘的数据结构，加载指定软盘扇区中的数据