实验三报告

院系：数据科学与计算机学院　　专业：计算机科学（大数据与人工智能方向）　　年级：大二下学期

指导教师：凌应标

【实验题目】用 C 和汇编实现操作系统内核

【实验目的】

增加批处理能力

提供用户返回内核的一种解决方案

在内核的 C 模块中实现

在磁盘上建立一个表，记录用户程序的存储安排

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

设计一种命令，命令中可加载多个用户程序，依次执行，并能在控制台发出命令

在引导系统前，将一组命令存放在磁盘映像中，系统可以解释执行【实验要求】

【实验方案】

软件工具：

为了方便编译 elf，编译环境从 Windows 环境换成了 Linux（真香）（不过代码和虚拟机还是 window 环境下的）

VSCode 换成了 Nodepad++

编译采用 NASM 和 gcc 进行 16 位实模式

相关知识和原理：

32 位下编译 16 位系统需要的参数和设置：

在 nasm 中

需要加入-f elf32 参数，并且在 asm 文件内使用[bits 16]标记

在 gcc 中

需要在开头加入 __asm__(".code16gcc\n");

在编译参数中加入 -fno-PIC -march=i386 -m16 -c -mpreferred-stack-boundary=2 -ffreestanding

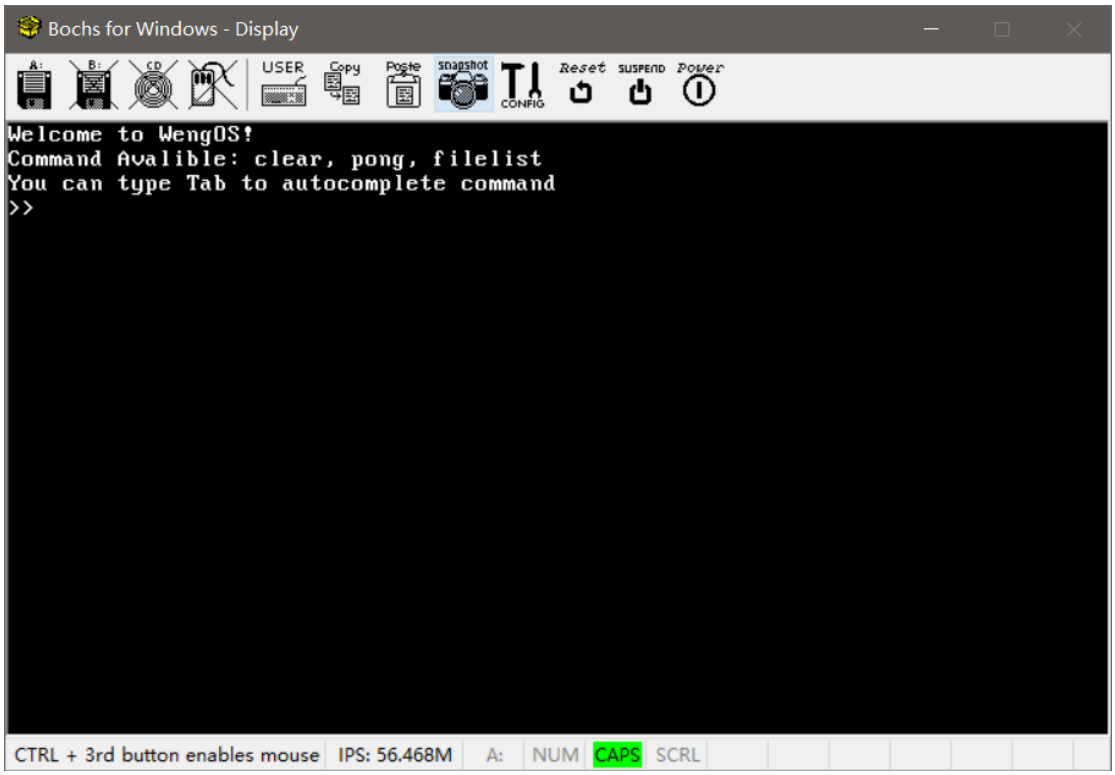
32 位编译下的 16bit 实模式下，C 语言和汇编程序相互调用规则（见参考资料）

汇编中栈的初始化与调用，函数调用上下文的保存

按键捕捉中的相关按键的 ascii 码（回车为\r，退格为\b，Tab 为\t）

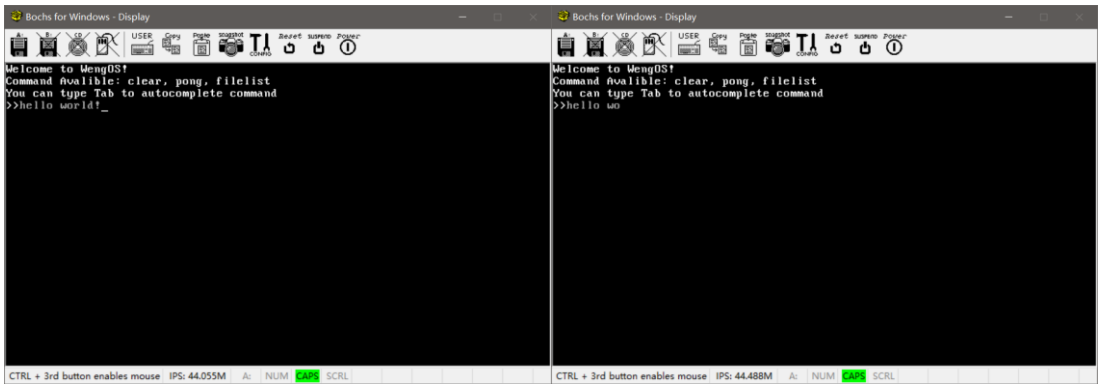
程序流程：

虚拟机启动时，由引导程序将内核程序加载如内存调用，显示欢迎界面和命令行



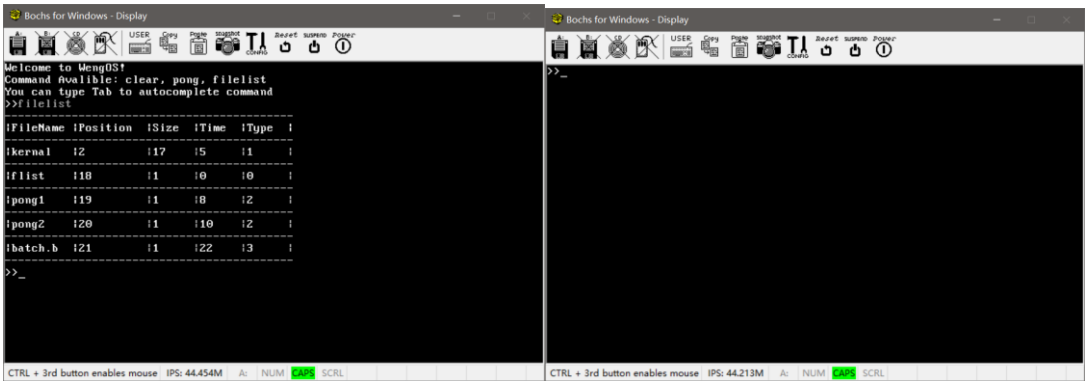
通过键盘可以输入字符并在命令行中显示

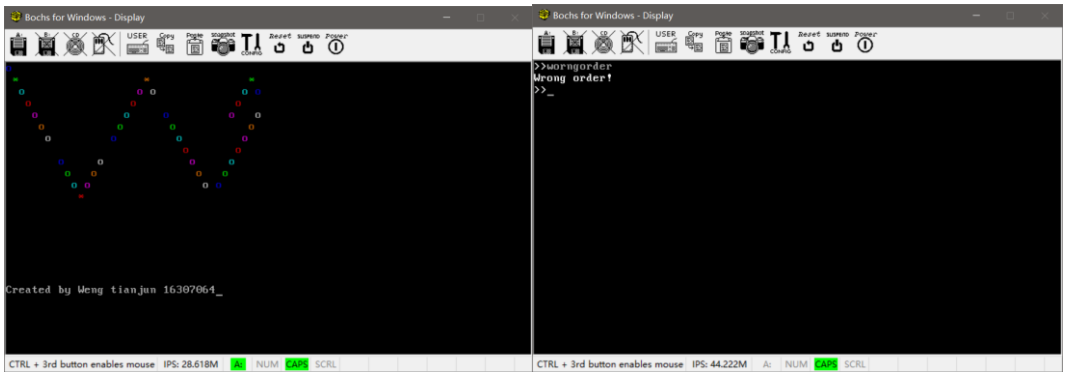
【额外】按下退格键可以撤销已经输入的字符（右图为撤销一部分指令后的结果）



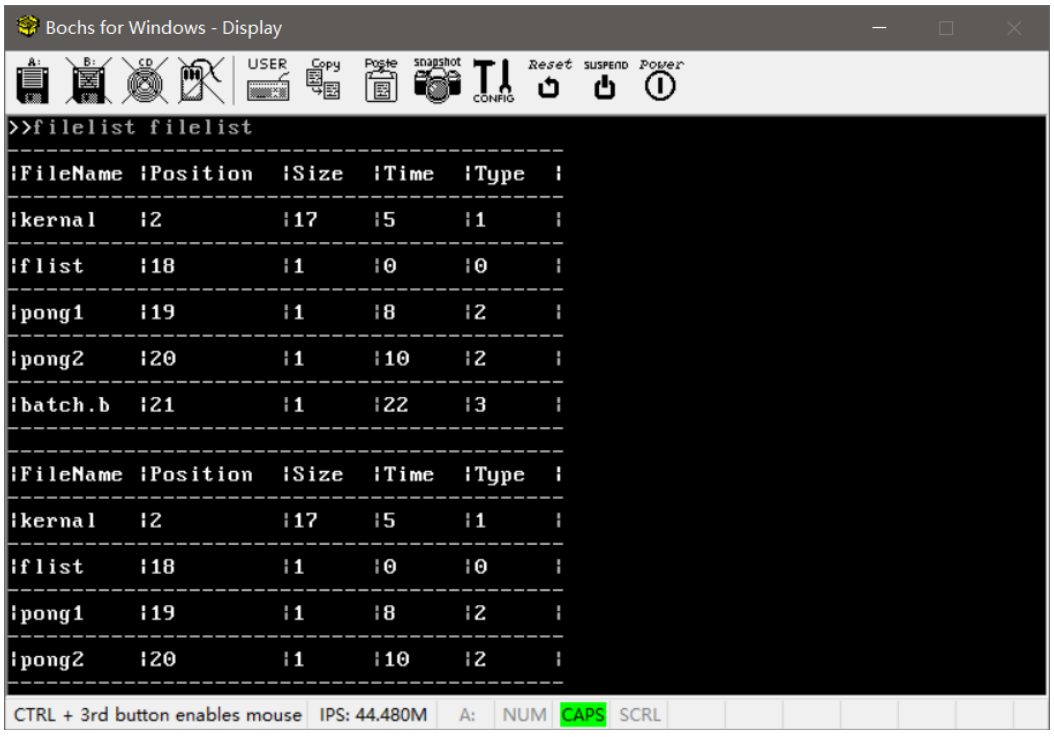
【额外】游标显示，游标会指向下一个文字被输入的地点（时间关系，没有做方向键移动游标）

指令正确时，可以执行对应的指令效果，指令错误时则提示指令错误，下图为三条预置指令效果（filelist，clear，pong）以及指令输入错误时的效果

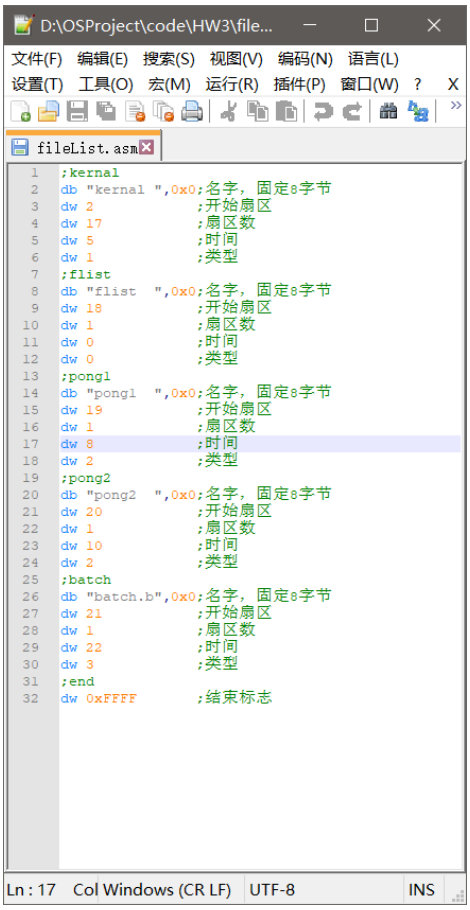




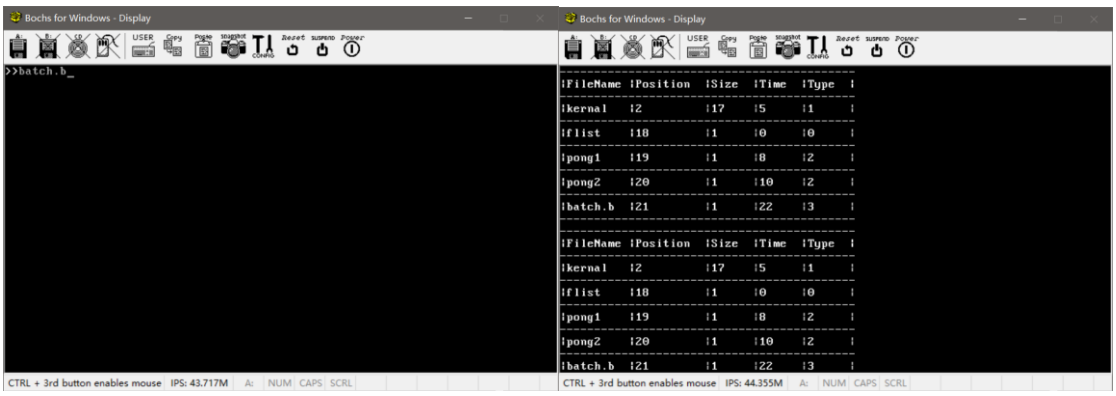
批处理功能，可以在同一行输入多条指令顺序执行，中间用空格隔开

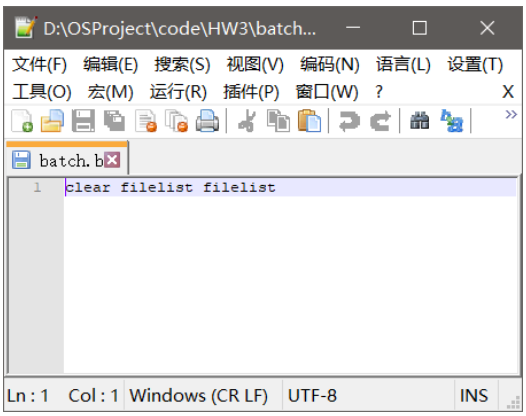


简易的文件系统，filelist 的输出是根据放在扇区 2 的 flist 文件来进行输出的，flist 里存放着每个文件的名字、位置、大小、时间和类型，会一直读到结束标志为止



硬盘调用批处理，当文件类型为 3 时，将会被内核认为是批处理文件，它会将里面的内容（ASCII 码限定）读取并按顺序执行





【额外】自动补全，对于预先设置的指令和文件系统中所有类型 3 的批处理文件名字，当用户只输入一部分的时候，可以按下 Tab 键自动让内核进行文字补全

【实验过程】

内核加载

方式同实验二，将内核从硬盘中加载到内存里，再通过 jmp 指令跳跃到目的地。

与实验二相比改进的地方是学会了使用 jmp cs:ip 的方式来进行跳跃，而非在实验二中自己使用的 jmp ip 的方式，从而将 cs 数据传入到了内核之中供后续偏移计算使用。

```
KERNAL_ADDRESS equ 0xA000
KERNAL_OFFSET  equ 0x0100
org 0x7c00
[BITS 16]
start:
    ;设置数据段
    mov ax, cs
    mov ds, ax
    mov ss, ax
    mov es, ax
    mov ss, ax
    mov sp, 0x7e00

    ;读取内核
    mov ah, 02h                                          ;功能号
    mov cl, 2                                            ;起始扇区号
    mov ch, 0                                            ;柱面号
    mov al, 17                                           ;扇区数
    mov dl, 0                                            ;驱动器号
    mov dh, 0                                            ;磁头号
    mov bx, KERNAL_ADDRESS + KERNAL_OFFSET
    int 0x13

    jmp KERNAL_ADDRESS>>4:KERNAL_OFFSET

times 510-($-$$) db 0
dw 0xAA55
```

欢迎界面

为了显示欢迎界面，我用汇编实现了 Pringf 函数，用来在 C 语言中显示欢迎的标语，显示方式与实验一相同，是循环直接修改显存达到的

重点在于兼容 C 语言的调用方式，让该函数可以在 C 中调用，因此这段代码段会在最开始时利用 push 来保存使用到的寄存器，mov bp,sp 来为后续调取参数做准备，执行完函数后利用 pop 将保存的寄存器恢复，再调用 o32 ret 回到原函数中（o32 是为了兼容 C 语言生成的 32bit 的 call）

```
Printf:
;void Printf(char* msg);
    push bp
    push es
    push ax
```

通过注释提醒自己这个函数在 C 语言中调用的方式，利用 push 保护寄存器内容

```
mov ax, 0b800h
mov es, ax
mov bp, sp

mov si, word[bp+(BIAS_CALL+BIAS_PUSH*3)]
mov di, word[screenCusor]
mov ah, 0Fh
.1:
mov al,byte[si]
inc si
test al, al
jz  .2
cmp al, 0Ah ;回车
jnz .3
push ax
    mov ax, di
    mov bl, 160
    div bl
    and ax, 0FFh
    inc ax
    mov bl, 160
    mul bl
    mov di, ax
pop ax
jmp .1
.3:
mov [es:di], ax
add di, 2
jmp .1

.2:
mov [screenCusor], di
```

循环显示字符，遇上回车则换行处理，这里用上了 C 语言变量 scerrnCusor，是通过 extern 导入的

```
;退出
pop ax
pop es
pop bp
o32 ret
```

恢复现场，并用 o32 ret 回到调用处

其余与老师提供的参考代码中功能类似的函数，只是额外做了 32 位的兼容，故对这类函数下文不再重复叙述。

拥有这些从汇编得到的基础函数后，就可以开始写 C 内核了，首先创建一个用于被汇编调用的 C 语言入口函数，用来显示欢迎页面并监听用户输入。

```
void CommandOn(){
    //载入文件信息
    Open(1,1,1,fileInfoList);
    //载入可执行batch信息
    LoadBatch();

    cmdCurrent = 0;
    Clear();
    Printf("Welcome to WengOS!\n");
    Printf("Command Avalible: clear, pong, filelist\n");
    Printf("You can type Tab to autocomplete command\n");
    Printf(">>");

    while(1){
        cmdBuffer[cmdCurrent] = GetChar();
        switch(cmdBuffer[cmdCurrent]){
            case '\r':
                cmdBuffer[cmdCurrent] = '\0';
                Printf("\n");
                CheckCommand(cmdBuffer,cmdCurrent);
                cmdCurrent = 0;
                Printf(">>");
                break;
            case '\t':
                AutoCompelete(cmdBuffer);
                break;
            case '\b':
                if(cmdCurrent>0){
                    BackChar();
                    cmdCurrent--;
                }
                break;
            default:
                if(cmdCurrent<CMD_BUFFER_LEN){
                    PutChar(cmdBuffer[cmdCurrent]);
                    cmdCurrent++;
                }
                break;
        }
    }

}
```

Open 是由汇编封装的磁盘读取函数，Clear 是汇编封装的清屏函数，GetChar，PutChar，BackChar 是汇编封装的键盘按键获取，字符输入，字符删除函数，其中【额外】的游标就是在这里利用中断修改的，而 CheckCommand，AutoCompelete 将会在后文介绍

这个函数在被装载的内核中调用，32 位中为了兼容 16bit 的 call 调用，需要先 push 0 来对齐栈，可以看到内核中与命令行有关的就只有调用这个 C 函数一句，剩下则是包含了通用 C 函数的 Utils.asm 的引用和与文件系统相关部分的汇编函数，因此内核逻辑基本是由 C 来完成的。

```
Commander:
    push 0 ;C函数调用兼容
    call CommandOn
    jmp $
```

之后的逻辑便是利用封装好的键盘中断逐字读取用户输入，并在检测到用户的特殊输入”\b” ，”\t”，”\r”时进行特殊处理

先对”\b”进行说明，它只是简单的调用了 BackChar 汇编函数，这个函数的作用是将当前 screenCusor 这个 C 变量指向位置的字符置空，利用中断移动游标，然后减少 cmdCurrent 值以达到删除缓冲区字符串的目的

再对”\t”进行说明，这是【额外】中的自动填充函数，它的原理是利用字串匹配来搜索当前最后一个空格之后的字符串和所有已经被注册在 orderList 变量中的字符串进行匹配，取出第一个完全匹配的字串，将后续部分循环使用 PutChat 函数填充上去，再对 cmdCurrent 等命令行变量进行一定的处理即可。OrderList 变量内存储着预先设置的三个指令，以及通过文件系统扫描出的所有批处理文件名字。

最后对”\r”进行说明，这是最核心的部分，它会循环检测被填入 cmdBuffer 缓冲区变量的字符串，并与预先设置好的字符串以及从文件系统中读取（后文介绍）的文件名进行匹配(CommandMatch 函数，因为只是简单的字符串匹配所以不详细叙述)，完全匹配成功则返回整数 cmdId，供执行函数用于判断需要执行的功能，在此处 0 为清屏，1 为显示文件列表，2 为执行 pong 程序，更大的数字代表执行对应批处理文件，-1 代表指令错误。

为了能判断一行中的多条指令，检测程序会以空格为分界，在匹配成功后返回当前匹配到的长度，供下一次判断使用，直到长度大于或等于指令的长度为止

```
int CheckCommand(char *cmd,int end){
    int start = 0,cmdId=-1;
    while(start<end){
        cmdId = CommandMatch(cmd,&start,end);
        switch(cmdId){
            case -1:
                Printf( "Wrong order!\n");
                return -1;
            case 0://clear
                Clear();
                break;
            case 1://filelist
                FileList();
                break;
            case 2://pong
                Pong();
                break;
            default:
                Open(batchPos[cmdId]%18+1,batchPos[cmdId]/18,batchSize[cmdId],batchBuffer);
                ReadBatch(batchBuffer);
                break;
        }
    }

    return cmdId;
}
```

Clear 只是简单调用汇编提供的 clear 功能

Pong 则是以类似实验二的方式读取文件再跳转过去，重点在于如何返回原来的函数中，由于需要修改 cs 和 ip 二者，所以跳过去时采用的是 push 跳转回来时需要的 cs 和 ip，再 push 要跳转的 cs 和 ip，再后调用 retf 跳转到目的地点。

```
;设置返回地址
push cs
push .return
;利用retf跳转
sub bx, 0x100
shr bx,4
push bx
push 0x100
retf
```

由于程序也有属于自己的栈，因此在程序初始化栈前要再用已 push 的寄存器保存 ss 和 sp 栈寄存器到当前栈中，再处初始化栈，而恢复时也需要先读取出存入的栈指针（不能直接 pop 到 ss 和 sp！不然栈会在 pop 之后立刻变动，需要 pop 到别的寄存器后再利用别的寄存器进行赋值），恢复栈到原来的地方后再 pop 恢复寄存器。最后使用 retf，跳回最开始设置好的返回地址处

```
;恢复栈指针
pop ax
pop bx
mov sp, ax
mov ss, bx
pop es
pop ds
pop cx
pop bx
pop ax
retf
```

FileList 涉及到文件系统，因此先从文件系统的初始化谈起。

在 CommandOn 函数中利用汇编封装的 Open 将装有文件信息的磁盘内容读取到 7e00（fileInfoList 的值，之所以读取到这里是考虑到 boot 部分启动后都不会再用了，不如废物利用一下），而 LoadBatch 则是对这些信息进行解析并将类型为 3 的文件名字注册到 orderList 数组中去，位置和大小注册到 batchSize，batchPos 数组中去，以供批处理指令执行和命令自动填充判断使用。

```
void LoadBatch(){
    char name[8] = "";
    int time,type;
    int index = 0;
    while(1){
        if(GetFileInfo(index,orderList[orderLen],&batchPos[orderLen],&batchSize[orderLen],&time,&type)){
            index++;
            if(type==3){
                orderLen++;
            }
        }else{
            break;
        }
    }
}
```

在 filelist 和 LoadBatch 中，最重要的函数就是 GetFileInfo 这个函数，它是汇编封装的函数，使我们能够从已经放入内存的文件数据中读取到特定文件的信息，它定义在 kenral.asm 中，是除了调用 c 函数以外仅存的一个函数。

```
GetFileInfo:
;int GetFileInfo(int fileIndex,char* name,int *sector,int *size,int *time,int *type);
    push bx
    push cx
    push dx
    push es
    push bp

    ;es和bp初始化
    mov bp, sp
    mov ax, FILE_LIST
    shr ax, 4
    mov es, ax

    ;计算文件偏移
    mov ax, word[bp + (BIAS_CALL+BIAS_PUSH*5+BIAS_ARG*0)]
    mov bx, FILE_BIAS
    mul bx
    xor esi, esi
    mov si, ax

    ;判断是否到底
    mov bx, word[es:si]
    cmp bx,0xFFFF
    jnz .read
    mov eax, 0
    jmp .return
```

C 函数调用必要的保存寄存器，利用传入函数的参数计算文件在 7e00 内存中的偏移，再利用偏移计算是否到达底部供 C 部分确认是否停止循环

```
;读取文件信息并赋值到指针
.read:
push dword[bp + (BIAS_CALL+BIAS_PUSH*5+BIAS_ARG*1)]
push esi
xor eax, eax
mov ax, FILE_LIST
push eax
push 0; 调用兼容
call Readline
add esp,BIAS_ARG*3
mov esi,eax
```

字符串的赋值需要单独处理，Readline 汇编函数大体作用从传入地址处（这里是 7e00）获得字符串并一个个赋值到传入的字符串指针里

```
;扇区
xor eax, eax
mov bx, word[bp + (BIAS_CALL+BIAS_PUSH*5+BIAS_ARG*2)]
mov ax, word[es:si]
mov dword[bx], eax
add si,2;大小
mov bx, word[bp + (BIAS_CALL+BIAS_PUSH*5+BIAS_ARG*3)]
mov ax, word[es:si]
mov dword[bx], eax
add si,2;时间
mov bx, word[bp + (BIAS_CALL+BIAS_PUSH*5+BIAS_ARG*4)]
mov ax, word[es:si]
mov dword[bx], eax
add si,2;类型
mov bx, word[bp + (BIAS_CALL+BIAS_PUSH*5+BIAS_ARG*5)]
mov ax, word[es:si]
mov dword[bx], eax
;返回读取正确
mov eax, 1

;函数返回
.return:
pop bp
pop es
pop dx
pop cx
pop bx
o32 ret
```

将读取到的数据赋值给“指针”（因为传入的是指针，所以要先获取指针地址对应的地址，再 mov 到对应地址处的变量处），最后是 C 调用的恢复寄存器

获取到这些信息后，我们就可以利用 Printf 来显示信息了，对于整数，我利用了 C 语言构造了一个简单的整数转字符串的函数 Int2Str（并且输出转换字符串的长度）来用于表头的显示，这些显示被简单的封装在了 GetFileInfo 函数中。

```
void FileList(){
    Printf("-------------------------------------------\n");
    Printf("|FileName |Position  |Size  |Time  |Type  |\n");
    Printf("-------------------------------------------\n");
    char name[CMD_BATCH_NAMELENGTH] = "";
    int sector,size,time,type;
    int index = 0;
    while(1){
        if(GetFileInfo(index,name,&sector,&size,&time,&type)){
            FileInfo(name,sector,size,time,type);
            index++;
        }else{
            break;
        }
    }
}
```

最后说说批处理执行的实现，用户输入已经在上方的用户输入检测中提到了，这里主要阐述当用户输入批处理时，如何载入并执行它。

通过文件系统我们可以得知批处理文件所在的位置和大小，再利用 Open 函数载入到一个临时的地点（我使用了 0xF000）,使用 Readline 函数读取每一行批处理指令到临时的缓冲字符数组，再将数组送入 CheckCommand 字符串处理运行，可以看到这些都是前面介绍过的内容，只不过在这里重新组装了起来。

唯一要注意的是磁头号，由于内核大小较大，batch 被我分配到了 19 扇区之后，而读取的时候不能直接读取 18 扇区，而是 1 磁头的 1 扇区处，因此从文件系统中获取到的扇区参数需要进行一定的处理。

```
default:
    Open(batchPos[cmdId]%18+1,batchPos[cmdId]/18,batchSize[cmdId],batchBuffer);
    ReadBatch(batchBuffer);
    break;


void ReadBatch(int batchBuffer){
    char tempBuffer[CMD_BUFFER_LEN];
    int current=0;
    int prev = 0;
    while(1){
        prev = current;
        current = Readline(batchBuffer,current,tempBuffer);
        if(tempBuffer[0] == '\0'){
            break;
        }else{
            char *start = tempBuffer;
            while(*start == ' ') start++;
            if(!CheckCommand(start,current-prev)){
                return;
            }
        }
    }
}
```

以上就是内核实现的全部内容

【实验总结】

尽管实验过程里说得十分简单，自己实际做起来的时候却遇上了很多的问题。

【保护模式之殇】

首先是 32 位的系统，也许是自己过于自信，在最开始的一周里试图使用 32 位保护模式来进行后续的操作，在观看了《Orange 操作系统的实现》之后感觉不算太难而信心满满，只是现实给了自己一耳光：明明是和书上一模一样的代码，最终只能导致虚拟机重启死机。而网上的关于保护模式的事例代码直接赋值粘贴编译后仍然是死机，而不知道是什么情况。经过了一周的折腾（期间包括切换虚拟机，切换调试工具，切换编译器，切换操作系统，调整编译参数，调整代码参数等一系列操作）后依然无果，最终只好无奈转向实模式。

【gcc 编译】

为了能够让 nasm 和 gcc 成功联合编译，尝试了网上许多的做法，而可能是 gcc 版本更新后增加了什么新的东西，网上的参数原样照抄并不能成功，最后增加了-fno-PIC 参数后才成功。这之后又在 gcc 的 16 位兼容的 32 位代码和 nasm 的 16bit 代码的相互调用问题上折腾了许久（网上很多资料都是片面的，只针对一篇文章完全无法达到想要的效果，经过多篇文章的综合最后总算是成功了。）

【栈指针调试】

尽管知道了相互调用的格式，但是在逻辑和细节方面也遇上了很多问题，其中绝大部分的问题是栈指针错误，汇编的调用没有计算正确栈的偏移，或者 push 和 pop 位数不等之类的情况。以至于后期一旦出现问题第一个反应就是看栈指针。通过近三天的逐指令调试，最深的感觉就是：C 真是一门方便的语言，不需要管栈的对齐，逐语句调试而非逐指令调试等等。在汇编上的调试不直观给我带来了许多困扰。不过当 BUG 全部被解决的时候，所带来的成就感是无与伦比的。

【指针，磁头】

在后半期制作中遇上的花费较长时间的地方在于指针和磁头，指针让我感觉有些绕的地方是需要从参数中引用之后，对引用后的地址进行赋值这一点，而磁头则是对于 18 扇区之后的读取需要换磁头才能读取。

【自我反思】

在完成实验三的代码后，我总结了自己在这两周中主要花费时间的地方，最终发现很多都源自于对编译器和汇编的基本概念不清晰有关，本次实验实际上的工作量其实并不大：如果是现在的自己重新制作，或许需要的时间可以降低到原来的 1/4。只是为了弥补这些基础，这两周花费了不少的时间。但这也意味着自己掌握了更多的知识，感觉到了进步，得到了许久未体验过的成就感。在这种意义上来说，自己的对操作系统和编译器方面的进步和面对难题沉下心来应对的体验正是这次实验给我带来的最大的意义。

【参考文献】

Windows 下 bochs 的 gdb 调试和 gui 调试编译
https://sourceforge.net/p/bochs/discussion/39592/thread/fd3c3be9/
https://sourceforge.net/p/bochs/discussion/39592/thread/59621f0690/

gcc 编译参数
http://cppblog.com/SEMAN/archive/2005/11/30/1440.html

关于报错 undefined reference to _GLOBAL_OFFSET_TABLE_
https://stackoverflow.com/questions/45422374/undefined-reference-to-global-offset-table-only-when-generating-binaries

C 和 ASM 相互调用
https://blog.csdn.net/laomd/article/details/80148121
https://blog.csdn.net/zdy0_2004/article/details/43880695
https://stackoverflow.com/questions/37950640/what-operand-size-should-ret-default-to/37963640#37963640
http://www.cnblogs.com/jackyzzy/archive/2013/03/25/2981543.html

指针
http://www.cppblog.com/xingkongyun/archive/2009/04/07/79219.html