实验三 ：C与汇编混合编程

目录

一.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。

二.实验要求

1，规定时间内单独完成实验。
2，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。
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.实验工具与环境（gcc+nasm+ld）

【环境】:

GCC 编译器
NASM 汇编器
LD 链接器

(可以直接将自己电脑中的编译器的.bin 文件添加到环境变量中的 path，这样就可以直接使用 gcc 和 ld 了。nasm 可以从网上下载，也是开源的。)

【编译命令】:

GCC 编译命令(file.c 编译成 cfile.o)
gcc -march=i386 -m32 -mpreferred-stack-boundary=2 -ffreestanding -c file.c -o cfile.o
NASM 汇编命令(file.asm 编译成 afile.o)
nasm -f elf32 file.asm -o afile.o
LD 链接命令(链接 afile.o 与 cfile.o 为临时文件 kernel.tmp，程序将被加载到内存 0x8100)
ld -m i386pe -N afile.o cfile.o -Ttext 0x8100 -o kernel.tmp
objcopy 命令(将.tmp 文件复制成.bin 文件)
objcopy -O binary kernel.tmp kernel.bin

【混合编程过程】

使用 gcc 编译 C 程序生成.o 文件，使用 nasm 汇编 x86 程序生成.o 文件，使用 ld 将.o 文件进行链接生成.tmp 文件，使用 objcopy 命令将.tmp 文件复制成.bin 文件。

2.方案的思想

1.程序内容:

从引导扇区进入引导程序，经过跳转进入到内核程序。

内核程序相当于一个控制台。在内核程序中，可以输入命令，了解程序的使用方法，了解当前的用户程序，也可以加载子程序到内存并运行。

四个子程序分别实现不同的功能。（在子程序中按下Esc键即可返回内核程序。）

第一个子程序是一个模拟打乒乓球游戏。屏幕左右方各有一个字符组成的’球拍’。用户通过‘w’，‘s’，‘up’，‘down’按键来分别使其上下移动。‘球’也由字符形成，会在屏幕中不断移动。‘球’撞到‘球拍’上则会反弹，撞到上下屏幕边缘也会反弹，撞到左右屏幕边缘则游戏结束。

第二个子程序是计算器。用户输入含有‘(’,’)’，‘+’，‘-’，‘/’，‘*’符号的正确表达式，程序就会输出准确的计算结果。

第三个子程序是文本编辑器。用户可以在这个程序写文本，程序会相应的显示，也可以通过退格键修改文本，回车键移动光标。

第四个子程序是显示个人信息。在屏幕特定位置显示本人的个人信息。

2.实现方法：

在汇编模块中实现输入输出，读取内存，读取扇区的基本函数。在C中以这几个函数为基础构建出更多符合自己的功能需求的函数。例如在C中实现读字符串，对比字符串，读数字这样的函数。以此来达到内核的控制功能。并且以这些函数实现复杂的用户程序。

3.相关原理：

【C 与汇编混合编程的过程】

使用 gcc 编译.c 文件生成.o 文件，使用 nasm 汇编.asm 文件生成.o 文件，使用 ld 将.o 文件进行链接生成.tmp 文件，使用 objcopy 命令将.tmp 文件复制成.bin 文件。

【文件头注意点】

使用 16 位的代码，汇编模块开头加[bits 16]。C 程序模块开头加__asm__(".code16gcc")。

【符号与变量引用规则】：

引用C程序中的变量和函数名。在编译后，前面都加了下划线，所以汇编程序中引用C程序中的变量和函数名时，要加下划线；

引用汇编程序中变量和函数名。汇编程序中变量名和函数名前面要加下划线，C程序中才能在引用时去掉下划线。

【函数调用参数传递与栈操作】：

汇编模块中调用C模块中的函数：

调用前要用extern声明C模块的函数。根据C中函数原型，用栈传递参数，顺序后参先进栈。C函数的返回值从eax寄存器中取得。调用C函数后，要将栈中参数弹出。进栈出栈以4字节为单位。

C模块中调用汇编模块中的过程和变量引用：

汇编模块的过程从栈中取得参数，不必出栈，直接引用栈中的值，顺序与C进栈对

应。返回值需要放置eax寄存器才能被C模块使用。如果C中想引用汇编模块中的变量和标识符，汇编模块中要用global声明这些符号。

【Call 指令与 ret 指令具体操作】：

该指令进行的具体操作分解如下：

SP	SP - 2
[SP]	IP
IP	IP + disp

RET

该指令完成的具体操作如下所示：

IP	[SP]
SP	SP + 2

【扇区在磁盘中的位置】：

一个磁道有18个扇区。由于BIOS无法跨磁道读扇区，对扇区的位置以磁头，柱面做了区分。对第q个扇区。它在磁盘中的位置计算方法为，

磁头 = （q/18）%2

柱面号 = （q/18）/2

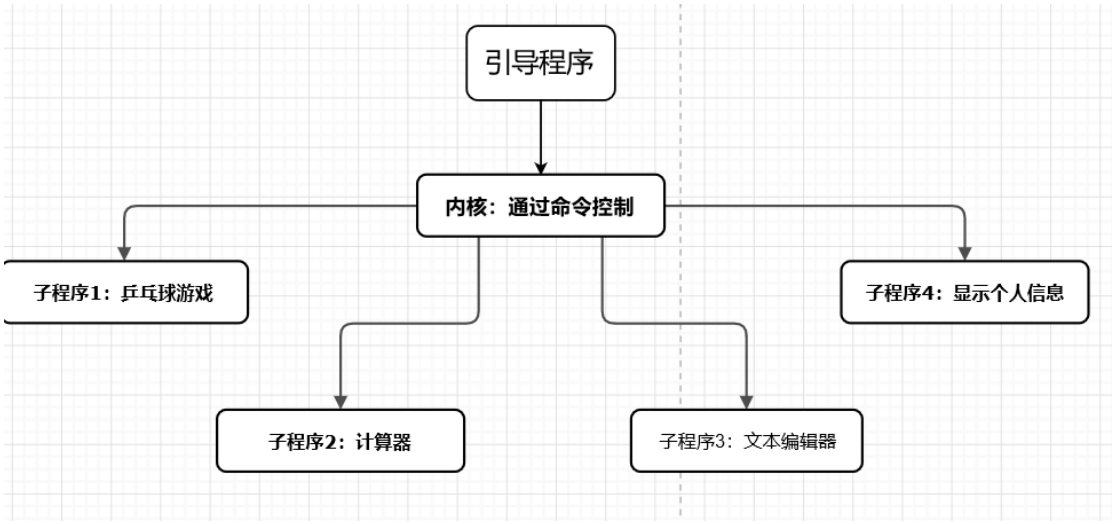
扇区起始号 = q % 18 + 1

【使用的输出中断】：

AH=1 置光标类型	CH 低 4 位=光标开始线 CL 低 4 位=光标结束线		文本模式下光标大小；当第 4 位为 1 时光标不显示
AH=2 置光标位置	BH=显示页号 DH=行号 DL=列号		左上角坐标是(0,0)
AH=3 读光标位置	BH=显示页号	CH=光标开始行 CL=光标结束行 DH=行号 DL=列号	CH 和 CL 含光标类型 左上角坐标是(0,0)
AH=5 选择当前显示页	AL=新页号		
AH=6 向上滚屏	AL=上滚行数 BH=填空白行的属性 CH=窗口左上角行号 CL=窗口左上角列号 DH=窗口右下角行号 DL=窗口右下角列号		①当滚动行数为 0 时表示清除整个窗口 ②仅影响当前显示页
AH=0EH TTY 方式显示	BH=显示页号 AL=字符代码		光标处显示字符并后移光标；解释回车、换行、退格和响铃等控制符
AH=0FH 取当前显示模式		(AL)=显示模式号 (AH)=最大列数 (BH)=当前页号	
AH=13H 写字符串(在 AT 及其以上系统上才有此功能)	AL=写模式 BH=显示页号 BL=属性 CX=串中的字符数 DH=写串的起始行号 DL=写串的起始列号 ES:BP=要写串首地址		①解释回车等控制符 ②写模式仅低 2 位： 位 0:0 表示移动光标，1 表示不动光标 位 1:0 表示串中字符代码和属性交替，1 表示串中只含字符代码

我的输出函数就是使用的0Eh号功能，也配合使用了2,3,6号功能

4.程序流程：



5.算法和数据结构：

计算器算法：栈实现

首先在字符串最后面加上‘=’符号，而后扫描整个字符串。

使用两个数组作为数字栈，和符号栈，分别存放数字和符号。

遇到数字就入数字栈。

对于符号，首先对他们的优先级进行排序，如‘(’符号的优先级为最大，‘=’的优先级最小。扫描符号时，当该符号的优先级大于栈顶符号的优先级时，入栈。当该符号的优先级小于栈顶符号优先级时，从数字栈中取出两个数字，从符号栈中取出一个符号，进行运算，计算结果入数字栈。当然还有几个特殊情况要额外考虑一下，例如当符号栈为空时，就必须入栈。

6.程序关键模块：

【输出字符函数】:

```
_Putchar:
    push bp
    mov bp,sp
    mov ax,[bp+6]
    pop bp
    mov bh,0          ;页号
    cmp al,1bh        ;判断Esc键,
    jz Esc_key
    cmp al,0dh        ;判断换行
    jz Enter_key
    cmp al,08h  ;
    jz backspace
    mov ah,0eh
    int 10h
    ret
Esc_key:
    jmp 0x8100
Enter_key:
    mov bh,0
    mov ah,3
    int 10h
    inc dh
    mov dl,00h
    mov ah,2
    int 10h
    ret
backspace:
```

从缓存区取出一个字符，判断其是否属于特殊键 esc，enter，backspace。对于 esc 键，直接返回内核；对于 enter 键，通过中断读光标位置和置光标位置，将光标移动到下一行首位；对于 backspace 键，退格后输出空格。其他字符在光标处显示并移动光标。

内核程序命令判断：

```
while(1)
{
    Getstr(apply);
    if(is_equal(apply,order[0]))
    {
        for(i = 0; i < 9; i++)
            Putstr(helpinf[i]);
    }
```

用户输入请求 apply，通过与内核中存储的命令集 order 逐一做对比判断是否输入正确命令，并判断所输入命令是什么。

子程序1控制球拍并显示球拍：

```
38      mov ah,1
39      int 16h
40      jz dic
41      cmp al,1bh
42      jz 0x8100                   ;输入Esc返回内核
43      xor eax,eax                 ;因为调用C程序返回时，栈会弹回32位，所以压栈也要压32位
44      mov ax,_Move                ;
45      call eax
46  ;确定方向
47  dic:
```

```
int left[2] = {2,5};
int right[2] = {2,5};
void Move()
{
    int key = Getkey();
    if(key == 0x1177 && left[0] > 0)//'w'
    {
        left[0]--;
        Setchar(left[0], 0, '8');
        Setchar(left[1], 0, ' ');
        left[1]--;
    }
```

当检测到有输入时，判断是否esc键，是就跳回内核，否则判断是否移动球拍。left数组

表示左边的球拍.[0]表示球拍上沿,[1]表示球拍下沿。Getkey()获取输入的按键。通过判断当前输入的按键，决定是否移动球拍并且如何移动。对右边球拍同理。

子程序2计算器算法：

```
//  if num
if(str[i] >= '0' && str[i] <= '9')
{
    j = 0; // j = varlen
    while(str[i+j] >= '0' && str[i+j] <= '9')
    {
        var[j] = str[i+j];
        j++;
    }
    i = i + j - 1;
    num[num_size] = Getint(var,j);
    num_size++;
    continue;
}
```

扫描到数字入数字栈。

```
// symbol
else
{
    if(symb_size == 0 || str[i] == '(' || Priority(str[i]) > Priority(symb[symb_size-1])) //入栈
    {
        symb[symb_size] = str[i];
        symb_size++;
    }
    else if(symb[symb_size-1] == '(' && str[i] != ')')
    {
        symb[symb_size] = str[i];
        symb_size++;
    }
    else //出栈
// 栈顶元素为左括号，扫描符号为右括号
if(symb[symb_size-1] == '(' && str[i] == ')')
{
    symb_size--;
    judge = 0;
    break;
}

else
{
    //数字栈出两位，符号栈出一位，计算结果入数字栈
    num_size--;
    num[num_size-1] = calcul(num[num_size-1], num[num_size], symb[symb_size-1]);
    symb_size--;
```

对于符号，对于栈为空；遇到左括号；以及栈顶符号为左括号，符号不为右括号；这三种特殊情况，以及符号优先级大于栈顶符号优先级时，符号入符号栈，扫描下一位。否则需要出栈：当栈顶元素为左括号，扫描括号为右括号时：符号栈出栈一位，丢弃当前符号，扫描下一位。否则数字栈出两位，符号栈出一位。计算结果入数字栈，再次扫描当前符号。

四.实验过程与结果

1.实验过程

【磁盘和内存安排方案】：

主引导程序放于磁盘第 1 个扇区，即引导扇区。将被自动加载到主存 07c00H ~ 07dffH。
内核程序存放于磁盘第 2 到第 17 个扇区，共 16 个扇区。由引导程序加载 08100H 开始的内存空间，即主存 08100H ~ 0a0ffH。
4 个用户子程序分别存放于磁盘 19~36,37~54,55~72,73~90 号扇区；每个子程序都占 16 个扇区。在执行时都由内核加载到 0a100H 开始的内存空间，即主存 0a100H ~ 0c0ffH。

【编译链接内核的命令】：

脚本kernel.cmd存放命令

kernel.cmd

```
gcc -march=i386 -m32 -mpreferred-stack-boundary=2 -ffreestanding -c kernel.c -o kernel.o
gcc -march=i386 -m32 -mpreferred-stack-boundary=2 -ffreestanding -c func.c -o cfunc.o
nasm -f elf32 func.asm -o afunc.o
ld -m i386pe -N kernel.o cfunc.o afunc.o -T linkerscript.txt -o kernel.tmp
objcopy -O binary kernel.tmp kernel.bin
```

linkerscript.txt存放分段信息

linkerscript.txt

```
SECTIONS
{
    . = 0x8100;
    .text : { *(.text) }
    . = 0x9100;
    .data : { *(.data) }
    .rdata : { *(.rdara) }
    .bss : { *(.bss) }
}
```

```
\原程序文件>.\kernel.cmd
\原程序文件>gcc -march=i386 -m32 -mpreferred-stack-boundary=2 -ffreestanding -c kernel.c -o kernel.o
\原程序文件>gcc -march=i386 -m32 -mpreferred-stack-boundary=2 -ffreestanding -c func.c -o cfunc.o
\原程序文件>nasm -f elf32 func.asm -o afunc.o
\原程序文件>ld -m i386pe -N kernel.o cfunc.o afunc.o -T linkerscript.txt -o kernel.tmp
\原程序文件>objcopy -O binary kernel.tmp kernel.bin
```

【编译引导程序的命令】

nasm boot.asm -o boot.bin

```
\MyOS\实验3：C与汇编混合编程\原程序文件及可执行文件\原程序文件>nasm boot.asm -o boot.bin
```

【编译链接用户子程序的命令】

用户程序 1

```
\原程序文件>gcc -march=i386 -m32 -mpreferred-stack-boundary=2 -ffreestanding -c tabletennis.c -o ctabletennis.o
\原程序文件>nasm -f elf32 tabletennis.asm -o atabletennis.o
\原程序文件>ld -m i386pe -N atabletennis.o ctabletennis.o -Ttext 0xA100 -o tabletennis.tmp
\原程序文件>objcopy -O binary tabletennis.tmp tabletennis.bin
```

用户程序2

```
\原程序文件>gcc -march=i386 -m32 -mpreferred-stack-boundary=2 -ffreestanding -c calculator.c -o calculator.o
\原程序文件>gcc -march=i386 -m32 -mpreferred-stack-boundary=2 -ffreestanding -c func.c -o cfunc.o
\原程序文件>nasm -f elf32 func.asm -o afunc.o
\原程序文件>ld -m i386pe -N calculator.o cfunc.o afunc.o -Ttext 0xA100 -o calculator.tmp
\原程序文件>objcopy -O binary calculator.tmp calculator.bin
```

用户程序3

```
\原程序文件>gcc -march=i386 -m32 -mpreferred-stack-boundary=2 -ffreestanding -c text_editor.c -o text_editor.o
\原程序文件>gcc -march=i386 -m32 -mpreferred-stack-boundary=2 -ffreestanding -c func.c -o cfunc.o
\原程序文件>nasm -f elf32 func.asm -o afunc.o
\原程序文件>ld -m i386pe -N text_editor.o cfunc.o afunc.o -Ttext 0xA100 -o text_editor.tmp
\原程序文件>objcopy -O binary text_editor.tmp text_editor.bin
```

用户程序4

```
\原程序文件>nasm information.asm -o information.bin
```

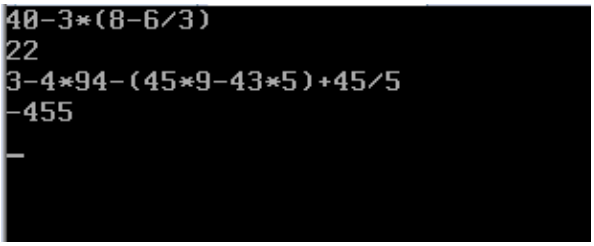
2.运行结果：

内核函数：“help”命令，“check”命令，“run”命令

```
what
invalid command

help
Input 'help' for help
Input 'run filename' to run a process
Input 'check filename' to see a process's information
Enter 'Esc' to come back to kernel at any time
There are four process here :
1.tabletennis
2.calculator
3.text editor
4.information

check tabletennis
name: tabletennis
memory:       8KB
SectorNo:      19
SectorNums:    16
condition:
game:
table tennis

run tabletennis_
```

第一个子程序：tabletennis

第二个子程序：calculator



输入

40-3*（8-6/3)

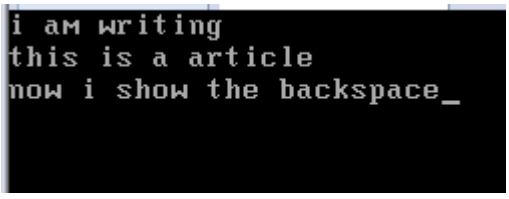
3-4*94-(45*9-43*5)+45/5

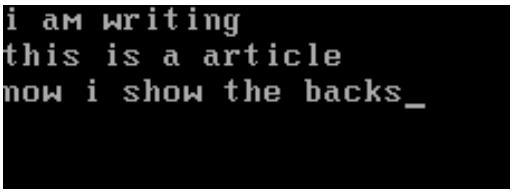
输出

22

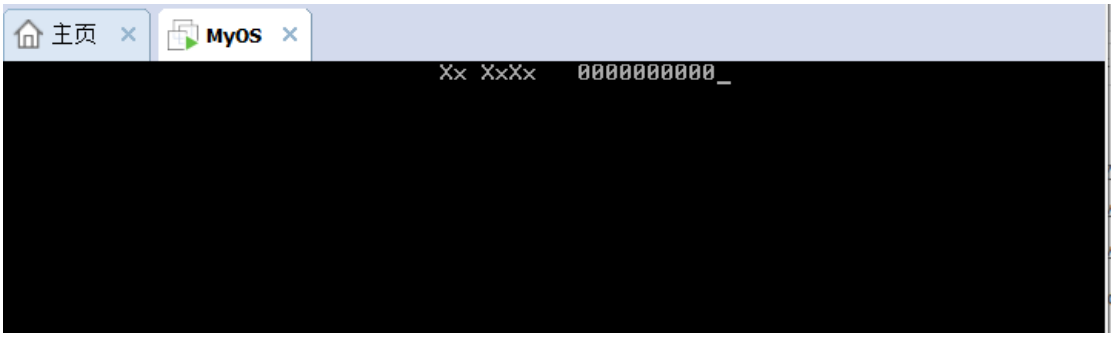
-455

第三个子程序：文本编辑器





第四个子程序：显示个人信息



3.遇到的问题及解决情况：

①【原链接命令出错】

```
E:\OS_exp\OStoolsDOS\3gcc\i686-7.1.0_old\bin>ld -m i386pe -N bar.o foo.o -Ttext 0x100  --oformat binary -o boot.bin
ld: cannot perform PE operations on non PE output file 'boot.bin'.
```

网上的信息比较少，我在某个角落里看到这么几句话。

“旧的MinGW版本存在“ld”根本无法创建非PE文件的问题。也许现在的版本有同样的问题。解决方法是使用“ld”创建PE文件，然后使用“objcopy”将PE文件转换为2进制文件”。

“这是连接器的一个已知问题，它不能链接PE文件和
同时转换为BINARY格式。您需要链接到PE
格式化文件，然后使用OBJCOPY将其转换为BINARY格式
之后。“

最后找到的中间文件格式是.tmp文件格式。另外这个问题似乎只在Windows中出现，使用Linus的朋友似乎没这个问题。实际使用的链接方法见上面实验方案3.1。

②【链接时，文件放置的顺序不同会导致最后生成的执行文件不同】

哪个文件在前面，链接出来的程序，从哪个文件开始执行。例如c程序放在前面，最后的执行文件中，程序由C程序处开始执行。另外，程序会从遇到的第一个函数开始执行，因此，要把主函数的定义放在最前面，后面再放其他函数的定义。又或者把主函数和辅助函数分开编译再链接。

③【从汇编模块调用 C 模块的函数时，不能用 call 函数名】

call function会将IP压入2位的栈，而C函数返回时，会弹出4位至eip中。因此会改变了栈顶里的值。破坏了数据结构。而从C调用汇编时，由于栈是以地址降低的方向增大的并且x86的存储方式是是高高低低，即低位放低地址。所以虽然C压了eip的四位地址入栈，但弹时把eip的低2位返回到ip中，使得ip的值没有改变，并且没有破坏栈里的数据，所以没问题。解决方法是在汇编中调用C函数时，使用如下代码。

xor eax,eax

mov ax,_funcname

call eax

④【链接时直接指定内存地址导致生成的文件过大】

```
\原程序文件>ld -m i386pe -N kernel.o cfunc.o afunc.o -Ttext 0x8100 -o kernel.tmp
\原程序文件>objcopy -O binary kernel.tmp kernel.bin
```

kernel.bin　　BIN 文件　　16 KB

通过自己安排代码段和数据段的位置，控制占用空间大小。使用的linkerscript.txt内容如下，

```
SECTIONS
{
    . = 0x8100;
    .text : { *(.text) }
    . = 0x9100;
    .data : { *(.data) }
    .rdata : { *(.rdara) }
    .bss : { *(.bss) }
}
```

这个文件的意思是，把代码段合并后放在内存0x8100处，把数据段合并后放在内存0x9100处。仅当代码段大小小于4KB。

```
\原程序文件>ld -m i386pe -N kernel.o cfunc.o afunc.o -T linkerscript.txt -o kernel.tmp
\原程序文件>objcopy -O binary kernel.tmp kernel.bin
```

kernel.bin　　BIN 文件　　8 KB

在winhex中打开可以发现，在命令中直接指定地址时，生成的2进制文件的尾部通常会携带有较长的无意义空白字节，这平白增大了该2进制文件的大小。另外也可以直接忽略其尾部，在处理用户子程序时，我就直接在命令中指定内存地址。

五.实验创新点

1.使用了 nasm+gcc 的工具链

2.设计了有用户交互的子程序:乒乓球游戏

3.实现了计算器功能

4.输入相对灵活，可以退格，回车换行，可以 Esc 键返回

六.实验总结：

实验一和实验二中，主引导程序同时作为内核。主引导程序有着 512 字节空间的巨大限制，以其为内核显然不合理。因此，在实验三中，引导程序与内核程序分离开来。实验三主要包括三个部分，引导程序，内核，四个用户程序。引导程序：引导程序的作用变得纯粹而贴近实际应用，仅用于加载执行内核。内核即是操作系统：丰富了内核的命令系统，内核用于控制执行用户程序。用户程序:用户程序只是用于配合反应内核的效果,因此不是重点。实际上本次实验的重点,是 C 与汇编混合的方法,C 语言的加入大大提高了程序表达能力，是继续后面的实验的根基。

这次的实验的坑比较多,C 与汇编的混合是个难点。从一开始的编译和链接环境就出了问题，最后我直接使用了 IDE 的.bin 文件。然后在链接的过程中又遇到了 ld 无法创建非 PE 文件的问题。我在网上很快就找到了解决这个问题的方法，就是先创建 PE 文件再用 objcopy 复制成非 PE 文件。具体操作大概就是,先生成.tmp 文件再用 objcopy 转化为.bin 文件。

另外在实验中我遇到了函数调用函数返回时地址压栈的位数的问题。这是个很容易掉的坑。

在一个扇区的柱面磁头的问题上我也遇到了困难,最后是在于渊的《一个操作系统的实现》上找到答案的。其中有详细的柱面号计算方法。

本次实验，我在几本书中得到了许多的帮助，书列于参考文献中。

参考文献：

1．《80x86 汇编语言程序设计教程》 （杨季文）
2．《nasm 中文手册》 （csdn）
3．《一个操作系统的实现》 于渊