# 实验三：C与汇编开发独立批处理的内核

## 1 实验题目

分离引导程序与系统内核，并使用 C 与汇编混合编程的方式重写内核，功能同上次监控程序。
具体来说，共有四项：

1. 寻找或认识一套匹配的汇编与 C 编译器组合。利用 c 编译器，将一个样板 C 程序进行编译，获得符号列表文档，分析全局变量、局部变量、变量初始化、函数调用、参数传递情况，确定一种匹配的汇编语言工具。
2. 写一个汇编程和 C 程序混合编程实例，汇编模块中定义一个字符串，调用 C 语言的函数，统计其中某个字符出现的次数（函数返回），汇编模块显示统计结果。
3. 重写实验二程序，实验二的的监控程序从引导程序分离独立，生成一个 COM 格式程序的独立内核。在监控程序保留原有程序功能的基础上，利用汇编程和 C 程序混合编程的优势，多用 C 语言扩展监控程序命令处理能力。
4. 重写引导程序，加载COM格式程序的独立内核。

## 2 实验目的

1. 加深理解操作系统内核概念，知道独立内核设计的需求；
2. 了解操作系统开发方法；
3. 掌握汇编语言与高级语言混合编程的方法；
4. 掌握独立内核的设计与加载方法；
5. 加强磁盘空间管理工作。

## 3 实验要求

再设计新的引导程序，实现独立内核的加载引导，确保内核功能不比实验二的监控程序弱，例如以下项目：

1. 在磁盘上建立一个表，记录用户程序的存储安排；
2. 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；
3. 设计一种命令，并能在控制台发出命令，执行用户程序。

# 4 实验方案

工具与环境：Windows（x86_64），gcc 或 clang，NASM，恰当版本的链接器，bochs 虚拟机（1MB 内存、1.44MB 软驱）

混合编程有多种方式，如 C 内嵌汇编、中间文件链接等。本次实验采用中间文件链接的方式实现混合编程，难点在于寻找恰当的工具搭配、正确的链接行为、两种语言的函数、过程、变量的相互引用。实现了混合编程以后，底层的操作在汇编实现，其余操作在 C 语言实现，封装好基本操作，即可方便地编写内核。

结合布置的实验内容，我将按照如下步骤进行：

**1、实现汇编调用 C 的函数与变量** C 编写一个 a+b 功能的函数，汇编传参并从 C 的变量中得到返回值。以此步骤学习混合编程，同时解决三个难点，以及实验内容 1。

**2、实现 C 调用汇编的过程与变量** 编写实验内容 2。

**3、混合编写内核** 分为基本操作的封装、监控程序功能的实现、内核与引导程序分离三步。

实验过程部分将以流畅的方式叙述实验细节，更多试错过程写在实验总结部分。

# 5 实验过程

## 5.1 工具、环境与链接

### 5.1.1 链接原理

首先需要了解中间文件链接的原理。汇编程序和 C 程序分别用汇编器、C 编译器生成各自的中间文件，然后通过链接器把两份中间文件链接起来，得到目标可执行文件。

中间文件相当于源程序的直白的机器语言翻译。链接器是组合两份语言程序的关键工具，其作用包括解析相互调用的符号、合并相同的段（代码段、数据段、只读数据段等）、重定位地址等。

更深入的知识可以阅读《深入理解计算机系统》第七章[1]学习。

### 5.1.2 组合工具选择

我的工作环境是 64 位 Windows，有多套工具可以选择。

- **tcc+tasm+tlink**

这套工具整体偏古老，例如老师提供的 tcc 据说是不支持 c99 标准的。且 tasm 语法与之前常用的 nasm 不同，这会有一定的不便性。

但其优点也在于古老。我们编写的操作系统是 16 位实模式的，这套工具能编译出 16 位的代码，契合设计理念。这其实是难能可贵的。

- **gcc+NASM**

这套工具的特点就是常见易得，平时许多人的工作环境就是如此，也是 Linux 下的标配。

缺点也很多。首先，现行 gcc 不支持编译成 16 位程序，目标构架最低为 i386，栈对齐不能小于 4 字节，所有试图使其成为 16 位的编译参数都是镜花水月。这会导致与 16 位汇编的组合变得比较尴尬。不过好在它能通过一些语句设置成“模拟 16 位”，即虽然用着 32 位寄存器和指令、32 位栈对齐，却使用 16 位实模式地址。

其次是 gcc 本身的设计缺陷，一种版本只能产生一种可执行格式，例如安装 MinGW-x86_64-gcc 只能生成基于 x86_64 的可执行格式，即便手动设置目标构架（即编译参数 `-march=i386`），生成的也是 i386:x86_64。这在与汇编链接的时候可能会产生目标构架不匹配的问题，解决方法是尝试各种不同版本的链接器，直至可行为止，例如我 `src` 目录下的 2.25.3 版本的 `ld3.exe`。

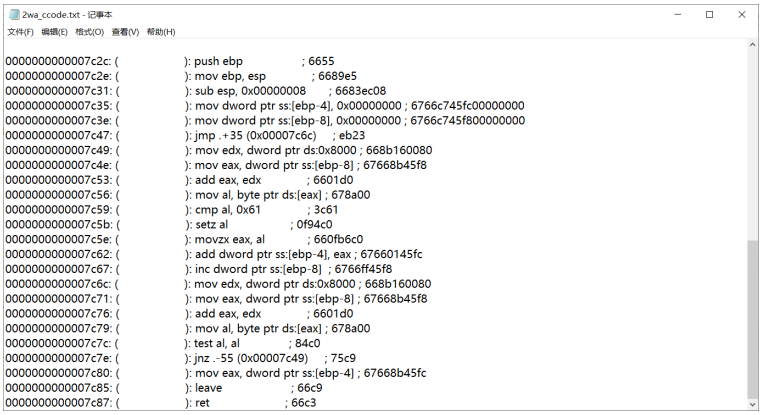



图 1: gcc只能生成32位代码

- **clang+NASM**

clang 的优点仅在于它能够方便地生成各种可执行格式，能“正规地”“有理有据地”解决目标构架不匹配的问题（避免了 gcc 上述解决方案由于强行链接而可能潜在的隐患）。

它同样不能编译出 16 位程序，且编译结果比 gcc 劣些（汇编代码精简度、编译出来的文件大小等方面）。

对比 gcc，它在链接时对于汇编有语法要求差异，例如 clang+NASM 时汇编引用 C 的函数、变量等不需要加下划线。（因为 gcc 会自动为全局符号加下划线，而 clang 不会。）

我的选择是 gcc+NASM，原则上使用 16 位实模式，辅以 32 位寄存器，以及 C 程序的“用 32 位模拟 16 位”。

### 5.1.3 链接方式

在 Windows 下中间文件、可执行文件是 PE 格式的，而我们的目标是简单的二进制 COM、BIN 格式。大多数 Windows 下的链接器是不能这样一步到位地完成链接任务的。

```
D:\大学资料\courses\操作系统原理\lab\lab3>ld -m i386pe -N myOSASM.o myOSC.o -Ttext 0x7c00 --
oformat binary -o myOS.bin
ld: cannot perform PE operations on non PE output file 'myOS.bin'
```

解决方法是，先用链接器生成 PE 格式的临时可执行文件，再用 objcopy 命令，从复杂的格式中提取我们需要的二进制代码，生成 COM 文件。编译命令如下：（.o 为中间文件）

```
ld3 --entry=_start -T ld.lds -o <filename>.tmp <ASM_filename>.o <C_filename>.o
objcopy -O binary <filename>.tmp <filename>.com
```

其中，--entry=_start 为指定程序入口（是一个叫做 _start 的段），-T 是指定我们自己编写的链接脚本 ld.lds。如果没有链接脚本，那么默认情况下代码段、数据段等之间会隔得很开，一个简单的 a+b 程序都会有二十几个扇区，造成空间浪费。链接脚本可以规定链接格式，我们指定代码段、数据段、只读数据段都挨在一起，变得紧凑。

同时，混合编程中 org 语句不能用了，因此 org 也需要在链接脚本中指定。

可在 src 目录下查看这个脚本。

## 5.2 汇编调用C

搭好上面的基础，就可以开始写代码了。

先从最简单的开始，C 中实现一个 a+b 函数，即传入两个参数，计算两个参数的和，保存在一个变量中。用汇编来调用它，以及从 C 的变量中取得结果。

### 5.2.1 C部分

C 要做的事很简单：

```
__asm__(".code16gcc\r\n");
int c;
void sum(int a,int b) {
    c=a+b;
}
```

不同的只是头上加了一句话。这就是 5.1.2 节所述的，实现“用 32 位模拟 16 位”的语句。

生成中间文件的编译命令为

```
gcc -c -m32 -march=i386 -mpreferred-stack-boundary=2 -static <C_filename>.c -o <C_filename>.o
```

其中，-c 表示只编译不链接，-m32 表示生成 32 位程序（有个 -m16 参数，但实际上并没有效果），-march=i386 表示目标构架为 i386，-mpreferred-stack-boundary=2 表示栈按 4 字节对齐，-static 表示使用静态链接（为了后面使用库函数）。

### 5.2.2 汇编部分

汇编是 16 位的，但要生成 32 位格式的中间文件去跟 C 程序链接。主要部分代码如下：

```

BITS 16

extern _sum
extern _c

```

```
6  section .text
7
8  global _start
9  _start:
10 push dword 0x2000000            ; push args, a=3, b=2
11 push dword 0x3000000
12 push 0
13 call _sum                       ; call void sum() in C
14 pop ecx                         ; pop args
15 pop ecx                         ; pop args
16
17 mov ax, 0xB800                  ; output result
18 mov es, ax
19 mov dword ecx, [_c]             ; result in ds:[_c]
20 shr ecx, 24                     ; efficient bits are at most significant two bits
21 xor cx, 48
22 mov [es:160*2], cl
23 mov byte [es:160*2+1], 0x7
24
25 call delay
26
27 ret
```

要点如下：

1、开头写上 `BITS 16` 以声明这是 16 位程序。以往的程序直接编译成 BIN 等格式因此编译器知道这是 16 位的，但现在要生成 32 位格式的中间文件，因此要显式声明；

2、C 程序被引用的函数、内存变量，都要在这里 `extern` 声明，使用时加下划线；

3、汇编程序的过程、内存变量要被扩大作用域的（比如被 C 程序调用，比如作为整个文件的入口），要用 `global` 声明，加下划线；

4、注意 C 程序的 `c` 这个变量，是它的指针到达了这里，因此是地址引用；

5、传参时，参数以 C 中声明的倒序压入栈中，并且注意大小端的问题。注意过程完毕后要把参数退栈；

6、注意栈是 32 位对齐的，例如 `call` 之前要 `push 0` 以凑够位数，从汇编返回 C 程序的话要 `o32 ret`。

编译命令如下：

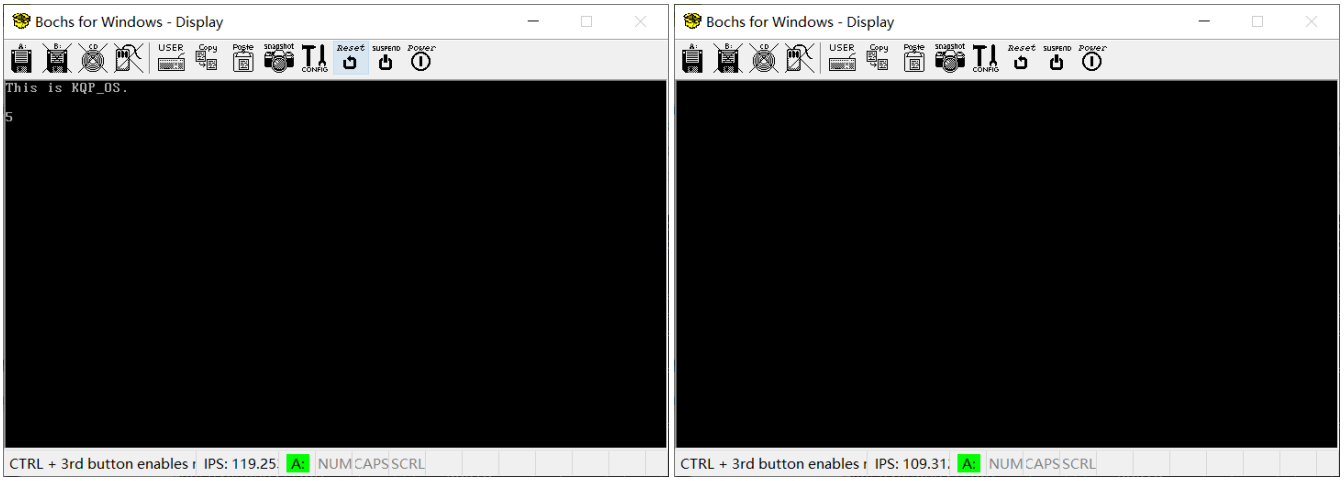
```
1 nasm -f elf32 <ASM_filename>.asm -o <ASM_filename>.o
```

表示生成 elf32 格式的中间文件。

### 5.2.3 运行结果

为了更方便操作，可以把上述共 4 条命令（两条编译命令、两条链接命令）合成一个 `makefile.bat`。

执行这 4 条命令即可得到 COM 程序。将其放入上次实验做的 `init_monitor` 里（执行用户程序，然后清屏死机），得到效果如下：




我的参数是 $a=3, b=2$，结果输出 5，说明该步骤成功。

## 5.3 C调用汇编：统计字符串中 a 的个数

有了上面的基础，实验内容 2 就是一个混合编程加深练习了。

汇编程序数据段定义一个字符串 `st`，调用 C 的过程统计字母 a 的个数。此处使用到 C 程序调用汇编内存变量。C 程序调用汇编过程涉及的细节都包含在这些实验中了，不额外进行实验。

C 程序如下：

```
1  __asm__(".code16gcc\r\n");
2
3  extern char* st;
4
5  int count()
6  {
7      int re=0;
8      for(int i=0; st[i]!=0; i++) re+=(st[i]=='a');
9      return re;
10 }
```

注意使用 `extern` 声明 `st` 是别的地方定义的。

汇编程序主要部分如下：

```
1  BITS 16
2
3  extern _count
4  global _st
5
6  section .text
7
8  global _start
9  _start:
10 push 0
11 call _count                          ; call void () in C
```

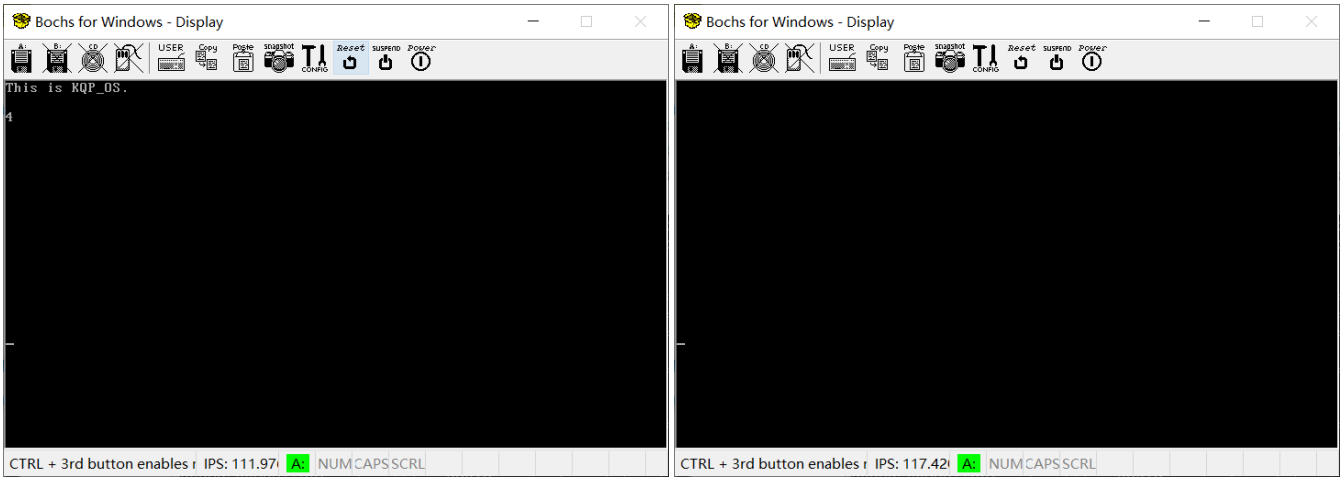
```
12
13  ; here output result,  result is in eax
14
15  call delay
16
17  ret
18
19  section .data
20  st_content db 'asdjaklfafdsa\0'
21  _st dd st_content
```

要点如下：

1、函数返回值存在 eax 中；

2、注意数据段的定义。基于“凡引用对方变量，都是拿对方变量的指针过来”，C 程序会把汇编拿过来的变量先当成指针访问其内存，再使用。因此这里为了保持字符串的指针形式，传参要传指针的指针。

编译链接成 COM，放到 init_monitor 中运行，结果如下：




可以看到，st_content 里的 a 的个数确实为 4，该步骤成功。

## 5.4 混合编写内核

现在要用混合编程重写内核。内核的功能目前如同监控程序：1、可以加载执行用户程序；2、可以使用命令控制。

当然，虽然有了 C 语言，但很多基础操作都是不可用的，比如 printf、scanf，也有“加载执行用户程序”这样的功能必须使用底层来实现。因此需要先封装一番，以方便后续操作。

### 5.4.1 基本操作封装

此处封装三个基本操作：标准输出、标准输入、加载执行用户程序。

对于标准输出，虽然 printf 不能用了，但是大一时写过一个大作业 myprintf，用 putchar 实现了 printf。此处将 myprintf.c 拿来复用，将 putchar 改为调用汇编的 int 10h 输出单个字符就能用了。int

10h 已经实现了光标自动前移、自动折行、到达屏幕底后自动上滚，非常方便了。为了避免程序过长，将 myprintf 功能阉割，暂时只保留字符、字符串、整数、相关格式的输出。

汇编输出单个字符的过程如下：

```
global _Put                                 ; print a single char
_Put:
    mov word ax, [esp+4]                    ; In C, we use Put(ch), here is ch
    mov bl, 0x07
    mov ah, 0xe
    int 10h
    o32 ret
```

对于标准输入，BIOS 中断服务只有“读入单个字符”这种操作。要实现像命令行那种输入方式，需要自行封装输入缓冲区。最好的方式是采取“洛谷P2201 数列编辑器”[2]那样的方法，用两个栈，分别代表光标前和光标后的内容，光标移动则视为两个栈之间的操作。但这种方式对于退格、中间插入字符比较麻烦，需要把整个缓冲区重新输出，并调整光标位置。

出于时间原因，这种方法留到后面的实验再实现。此处仅封装一个普通的缓冲区，不接受方向键、退格，只能一路往前写。那么所要实现的，只是读入一个字符就将其加入缓冲区并立即输出，遇到回车就结束。

C 代码如下：（汇编读入的字符 askii 码存于 char_askii，键盘码存于 char_kb）

```
extern char char_askii;
extern char char_kb;
extern void Getchar();

char buf[maxlen];
short buflen;
void ReadString()
{
    for(buflen=0; ; )
    {
        Getchar();
        if (char_askii>=32)       // that can be print to screen
        {
            Put(char_askii);
            buf[buflen++]=char_askii;
        } else if (char_askii==13)        // Enter, finish!
        {
            Put(char_askii);
            Put('\n');
            buf[buflen]=0;
            break;
        }
    }
}
```

汇编代码读入单个字符如下：

```
global _Getchar                 ; read a single char
_Getchar:
    mov ah, 0                   ; function: input from keyboard
    int 16H                     ; interrupt
```

```
5       mov [_char_askii], al
6       mov [_char_kb], ah
7       o32 ret
```

对于加载用户程序，就跟上次实验的代码一样了，新增一个参数表示用户程序所在扇区即可。

### 5.4.2 建立命令

此处实现两种简单命令：输入 1 至 4 表示运行 4 个用户程序（上次的四个射字母），输入“table”输出一个程序信息表格。由于有了高级语言，一切都很方便，将来命令处理这部分还要做成一个语法自动机。

代码如下：

```
1  while (1)
2  {
3      Put('\r'), Put('\n'), Put('>');
4      ReadString();
5      if (buf[0]=='t' && buf[1]=='a' && buf[2]=='b' && buf[3]=='l' && buf[4]=='e' && buflen==5)
6      {
7          myprintf("  ProgID  Sector   Large\r\n");          // table
8          for(int i=1; i<=4; i++) myprintf("\%8d\%8d\%6dKB\r\n",i,i+8,1);
9      } else if (buf[0]>='1' && buf[0]<='4')             // load client program
10     {
11         load_client(buf[0]-'0'+8);
12     } else                          // invalid command
13     {
14         myprintf("undefined command!\r\n");
15     }
16 }
```

### 5.4.3 引导程序、用户程序的修改

引导程序目前没有什么工作了，它只需把内核加载到内存即可。目前内核编译出来大约有 5 个扇区以上（有一大段空段，是数据内存初始为空而已，要留够位置，不要忽略）。

此处修正以前的一个 bug，call 用户程序的时候，要 call 段基地址:偏移地址，否则 cs:ip 和 es:di 等尽管指向同一个地方，但处于不同的段基，存在隐患。

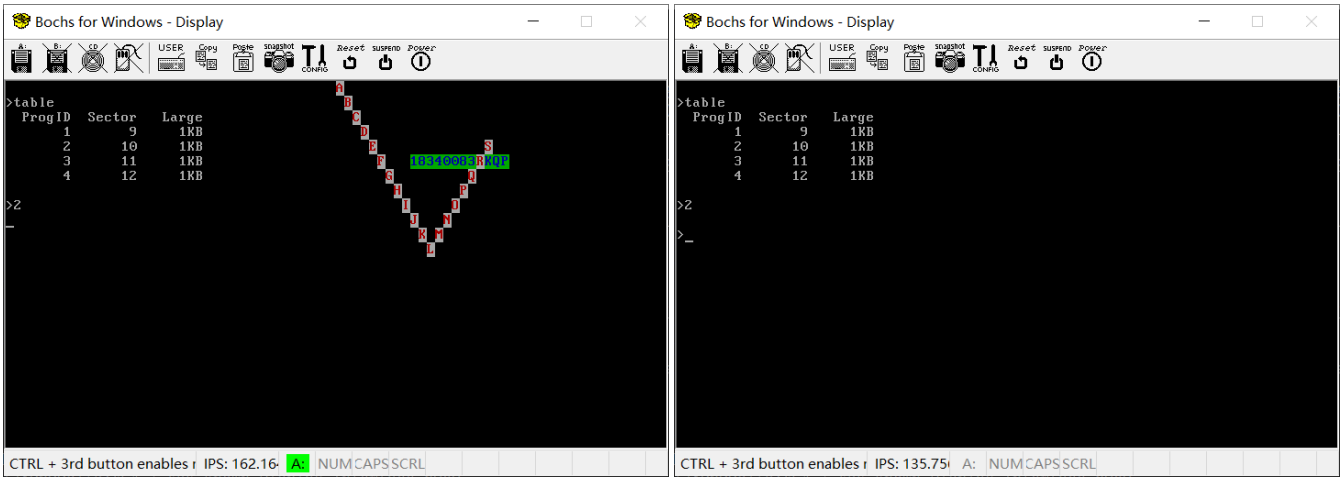
至于用户程序，我只是加上了执行完后清屏。

### 5.4.4 测试

将内核编译，从 2 号扇区开始存放。将之前的四个射字母程序分别放于 9、10、11、12 号扇区。

引导程序仅加载内核至 0x8100 处，内核加载用户程序至 0x18100 处。

运行结果如下：




可以看到，输入 2 命令，执行第二个弹射程序，执行完后回到内核，可以继续输入。该步骤成功。

# 6 实验总结

这次实验足足做了一周。最大的困难依然是参考资料的稀缺、零碎，甚至错误。

## 6.1 主要问题在于环境与工具

如实验方案所述，我是通过 a+b 程序来起步学习混合编程的，这个过程花了三天之久。

最艰辛的问题莫过于每个人的软件、环境、配置版本不同，每个同学遇到的问题都不太一样，而每个同学的解决方案也不同，这些解决方案却又不通用。有的同学使用高版本的链接器解决问题，有的同学使用低版本的链接器解决问题，有的同学的链接器直接一步到位生成 COM 文件也没有问题，而我一开始试遍了所有方法、所有编译命令、所有链接器，都做不出来。

解决方案是咨询一位底层专家舍友。他阅读过《深入理解计算机系统》《程序员的自我修养：链接、装载与库》，因此懂得链接的知识，他为我解答关于链接过程的种种错误。

后来遇到了生成二进制文件过大的问题，我无法解决，求助于他，他使用 clang 与 llvm 的链接工具得到了正常的 COM 文件。我一时迷信，认为 gcc 及其链接工具很有问题以至于不能在我的机器上完成任务。但后来再与别的同学沟通与尝试，我们认识到其实真正起作用的不是 clang，而是舍友使用的链接脚本。

## 6.2 其他错误

gcc 的位数问题也是意想不到的问题。原以为有了 `-m16`、`.code16gcc` 等命令，它就能编译出和我 NASM 一样的语句。后来摸索中发现，它在骗我，就一副“狗改不了吃屎”的样子。我也只得妥协，采用 16+32 这样强行凑合的方式编程。

## 6.3 待完成项目

1、内存管理，比如栈空间，现在我的栈空间是，把一个程序加载到某个 64K 段上，那么段末就是栈，这显然太粗糙了；

2、程序大小，记录在程序头部，这样实现加载程序时，能自动判断加载多少个扇区；

3、输入缓冲区的任务。

# 参考文献

[1] Randal E.Bryant等,龚奕利等译,深入理解计算机系统（原书第三版）,机械工业出版社,463页

[2] 洛谷P2201数列编辑器,https://www.luogu.com.cn/problem/P2201

[3] 李忠等,x86汇编语言：从实模式到保护模式,北京,电子工业出版社,第3部分 32位保护模式

[4] 《程序员的自我修养：链接、装载与库》

[5] 总结——gcc+nasm交叉编译在16位实模式互相引用的接口,https://blog.csdn.net/laomd/article/details/80148121