

操作系统实验项目

C 与汇编开发独立批处理的内核

郑戈涵 17338233 931252924@qq.com

摘要

本次实验共完成两个任务: 使用汇编语言与 c 语言的混合编程和独立内核的设计与加载

目录

1 实验目的	3
2 实验要求	3
3 实验内容	3
3.1 分析 C 程序编译获得的符号列表文档	3
3.2 汇编和 c 程序混合编程	3
3.3 扩展内核程序	3
4 实验原理	4
4.1 实模式和保护模式	4
4.2 函数调用约定	4
4.3 保护模式的运行模式反转	5
4.4 混合注意事项	5
5 实验过程	6
5.1 混合编程	6
5.2 修改引导程序	8
5.3 设计库过程	9
5.4 监控程序修改	10
5.5 设计用户程序的加载方式	13
5.6 修改用户程序	14
5.7 软盘扇区安排	14
5.8 镜像文件生成	14
6 程序使用说明	14
6.1 实验环境	14
6.2 编译方法	15
6.3 运行与演示	15
7 总结与讨论	16
7.1 特色, 不足与改进	16
7.2 收获	16
7.3 感想	16

1 实验目的

1. 加深理解操作系统内核概念
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

2 实验要求

1. 知道独立内核设计的需求
2. 掌握一种 x86 汇编语言与一种 C 高级语言混合编程的规定和要求
3. 设计一个程序，以汇编程序为主入口模块，调用一个 C 语言编写的函数处理汇编模块定义的数据，然后再由汇编模块完成屏幕输出数据，将程序生成 COM 格式程序，在 DOS 或虚拟环境运行.
4. 汇编语言与高级语言混合编程的方法，重写和扩展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.
5. 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，展示原有功能或加强功能可以工作.
6. 编写实验报告，描述实验工作的过程和必要的细节，如截屏或录屏，以证实实验工作的真实性

3 实验内容

3.1 分析 C 程序编译获得的符号列表文档

1. 编译样板 C 程序，获得符号列表文档.
2. 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.

3.2 汇编和 c 程序混合编程

写一个汇编和 c 程序混合编程实例，展示你所用的这套组合环境的使用。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。执行程序可以在 DOS 中运行.

3.3 扩展内核程序

1. 把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 1.44MB 软盘映像中，保存到特定的几个扇区.
2. 扩展监控程序命令处理能力.
3. 重写引导程序，加载 COM 格式程序的独立内核.

4 实验原理

4.1 实模式和保护模式

实模式 [5] 实模式（英语：Real mode）是 Intel 80286 和之后的 x86 兼容 CPU 的操作模式。实模式的特性是一个 20 比特的区段存储器地址空间（意思为只有 1 MB 的存储器可以被定址），可以直接软件访问 BIOS 例程以及周边硬件，没有任何硬件等级的存储器保护观念或多任务。

实模式的缺陷 [7]

- 实模式下操作系统和用户程序属于同一特权级，这哥俩平起平坐，没有区别对待。
- 用户程序所引用的地址都是指向真实的物理地址，也就是说逻辑地址等于物理地址，实实在在地指哪打哪。
- 用户程序可以自由修改段基址，可以不亦乐乎地访问所有内存，没人拦得住。以上 3 个原因属于安全缺陷，没有安全可言的 CPU 注定是不可依赖的，这从基因上决定了用户程序乃至操作系统的数据都可以被随意地删改，一旦出事往往都是灾难性的，而且不容易排查。
- 访问超过 64KB 的内存区域时要切换段基址，转来转去容易晕乎。
- 一次只能运行一个程序，无法充分利用计算机资源。
- 共 20 条地址线，最大可用内存为 1MB，这即使在 20 年前也不够用。

保护模式 [4] 是一种 80286 系列和之后的 x86 兼容 CPU 的运行模式。保护模式有一些新的特性，如存储器保护，标签页系统以及硬件支持的虚拟内存，能够增强多任务处理和系统稳定度。现今大部分的 x86 操作系统都在保护模式下运行，包含 Linux、FreeBSD、以及微软 Windows 2.0 和之后版本。

虚拟 8086 模式 [3] 在 80386 微处理器及更高版本中，虚拟 8086 模式（也称为虚拟实模式，V86 模式或 VM86）允许执行实模式应用，这些实模式应用无法在处理器运行保护模式操作系统时直接在保护模式下运行。它是一种硬件虚拟化技术，它允许 386 芯片模拟多个 8086 处理器。它源于 80286 保护模式的痛苦经历，它本身并不适合很好地运行并发实模式应用程序。

4.2 函数调用约定

函数调用约定是什么？调用约定，calling conventions，从字面上理解，它是调用函数时的一套约定，是被调用代码的接口，它体现在：

- 参数的传递方式
- 参数的传递顺序
- 寄存器环境是调用者保存，还是被调用者保存，保存哪些寄存器

X86 调用约定 cdecl(C declaration，即 C 声明) 是源起 C 语言的一种调用约定，也是 C 语言的事实上的标准（本次实验采用该约定）。在 x86 架构上，其内容包括：

- 函数实参在线程栈上按照从右至左的顺序依次压栈。

- 函数结果保存在寄存器 EAX/AX/AL 中
- 浮点型结果存放在寄存器 ST0 中
- 编译后的函数名前缀以一个下划线字符
- 调用者负责从线程栈中弹出实参（即清栈）
- 8 比特或者 16 比特长的整形实参提升为 32 比特长。
- 受到函数调用影响的寄存器（volatile registers）：EAX, ECX, EDX, ST0 - ST7, ES, GS
- 不受函数调用影响的寄存器：EBX, EBP, ESP, EDI, ESI, CS, DS
- 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）

4.3 保护模式的运行模式反转

CPU 处于实模式下时，并不是变成了纯粹的 16 位 CPU，它相当于 8086 的加强版，依然可以使用 32 位下的资源。也就是说，资源是共通的，无论哪种模式都可以在指令中使用它们，同样一句汇编代码，编译器无法确定它到底是属于实模式，还是属于保护模式呢，需要人为告诉编译器一些信息：

bits 16 是告诉编译器，下面的代码帮我编译成 16 位的机器码。

bits 32 是告诉编译器，下面的代码帮我编译成 32 位的机器码。

4.4 混合注意事项

- 编写混合的代码时，跳转部分要尤其注意，包括跳转和返回，*nasm* 支持 *call dword* 和 *retf* 语句，调用 c 函数时使用 *call dword* 能够在跨段时保存 CS 和 IP 到栈中，调用汇编的过程中使用 *retf* 能够在返回时弹出 CS 和 IP，正确跨段返回。
- 由于函数语句执行时会破坏原有环境，在每次进入汇编时都需要使用 *pusha* 将所有常用寄存器入栈，离开汇编时使用 *popa* 将所有常用寄存器出栈。
- 不能在传入 *char** 类型变量时使用字面量字符串，比如 `print("Message")` 这样的，使用后可能出现下图的错误：read_virtual_checks():read beyond limit.

```

Bochs for Windows - Console
00001421847i[PCI ] i440FX PMC write to PAM register 59 (TLB Flush)
00001422570i[BIOS ] bios_table_cur_addr: 0x000f9ff4
00029625326i[BIOS ] Booting from 0000:7c00
(0) Breakpoint 1, 0x0000000000086a3 in ?? ()
Next at t=97165840
(0) [0x000000000086a3] 0000:86a3 (unk. ctxt): push 0x66678750 ; 666850876766
<bochs:3> n
Next at t=97165841
(0) [0x000000000086a9] 0000:86a9 (unk. ctxt): lea ax, ds:[di-34] ; 8d45de
<bochs:4>
Next at t=97165842
(0) [0x000000000086ac] 0000:86ac (unk. ctxt): push eax ; 6650
<bochs:5>
Next at t=97165843
(0) [0x000000000086ae] 0000:86ae (unk. ctxt): call .-472 (0x000084dc) ; 66e828feffff
<bochs:6>
00097165862e[CPU0 ] read_virtual_checks(): read beyond limit
00097165870e[CPU0 ] read_virtual_checks(): read beyond limit
00097165878e[CPU0 ] read_virtual_checks(): read beyond limit
00097165886e[CPU0 ] read_virtual_checks(): read beyond limit
00097165894e[CPU0 ] read_virtual_checks(): read beyond limit
00097165902e[CPU0 ] read_virtual_checks(): read beyond limit
00097165910e[CPU0 ] read_virtual_checks(): read beyond limit
00097165918e[CPU0 ] read_virtual_checks(): read beyond limit
00097165926e[CPU0 ] read_virtual_checks(): read beyond limit
00097165934e[CPU0 ] read_virtual_checks(): read beyond limit
00097165942e[CPU0 ] read_virtual_checks(): read beyond limit
00097165950e[CPU0 ] read_virtual_checks(): read beyond limit
00097165958e[CPU0 ] read_virtual_checks(): read beyond limit
00097165966e[CPU0 ] read_virtual_checks(): read beyond limit

```

图 1: 传入字面值字符串参数

5 实验过程

本次实验考虑到 *dosbox* 和 *tcc,tasm,tlink* 上手比较复杂, 因此使用了 *gcc,nasm,ld*。由于出现了内核程序, 这次实验开始将会在实模式和保护模式上有较大差别, 我将使用实模式完成实验。

1. 编写汇编程序和 c 程序, 生成镜像并测试
2. 修改引导程序, 只加载内核程序, 并跳转
3. 设计 c 库和汇编库, 封装基本的 io 操作
4. 修改监控程序, 扩展功能
5. 设计用户程序的加载方式
6. 修改用户程序
7. 生成镜像, 在虚拟机中加载

5.1 混合编程

我设计的样板 c 程序将同时用于符号列表文档分析和测试混合编程

5.1.1 分析 c 程序汇编结果

生成汇编文件可以使用下面的两种方法, 第一种方是通过 *objdump* 生成, 比较清晰, 简单, 第二种包含了 *gcc* 产生的附加信息, 比较复杂。

代码 1: 生成汇编文件

```

1 # 方法一
2 gcc -fno-pie -c -m16 -march=i386 -nostdlib -ffreestanding
3 -mpreferred-stack-boundary=2 -lgcc -shared str.c -o str.o
4 objdump -s -ld -C -S str.o > str.s
5 # 方法二
6 gcc -fno-pie -c -m16 -march=i386 -nostdlib -ffreestanding

```

```
7 -mpreferred-stack-boundary=2 -lgcc -shared str.c -S -o str.o
```

下面对 gcc 编译产生的文件进行分析，首先是比较简短的 c 语言代码：

Code 2: str.c

```
1  extern void  cls ();
2  extern void  putchar(char c);
3  extern void  printPos(char *str, int n, int row, int col);
4  char *msg = "wodemsg1";
5  int strlen(char *str)
6  {
7      int cnt = 0;
8      while (str[cnt++] != '\0')
9          ;
10     return cnt-1;
11 }
12 void printInfo()
13 {
14     cls();
15     char *Info = "wodeinfo";
16     putchar(strlen(Info)+ '\0');
17     printPos(Info, strlen(Info), 6, 23);
18     printPos(msg, strlen(msg), 9, 23);
19 }
20
21 void printmsg(){
22     int row = 9, col=23;
23     printPos(msg, strlen(msg), row, col);
24 }
```

程序完成了简单的两个功能，计算字符串的长度和打印两个字符串到屏幕上。为节省篇幅，我去掉了两个比较复杂的函数，代码请看15，gcc 生成的汇编文件中分了非常多的部分，在开头有标签（如.LC0:），后面引用时会用此名字引用。

全局变量 全局的变量前面都有 *global* 标签，比如第 4 行的 `msg`，在链接之后生成的文件里，汇编的代码也是能够调用这些变量的。

函数调用 函数声明也是有标签的，比如 29 行的 `printmsg`。在 32 行，函数将 `ebp` 压栈，35 行将 `esp` 传给 `ebp`，使用 `ebp` 作为基址进行寻址。37 行将 `esp` 向下移动 8 个字节，完成局部变量的空间分配。然后 38, 39 两句 `movl` 将栈中分配的局部变量进行初始化，分别赋值 9 和 23。40 行将 `msg` 的地址取到 `eax` 中并压栈，这样子就完成了 `strlen` 调用时的参数传递。42 行调用 `strlen` 过程。接下来的分析基本相同。

5.1.2 混合编程程序设计

混合编程中，c 代码若要使用汇编的函数，需要使用 `extern` 声明此函数，汇编代码中需要在开头将其声明为 `global`；汇编代码若要使用 c 的函数，需要用 `extern` 关键字为了体现混合编

程的优势，代码里面使用了汇编调用 c 函数的方法和 c 函数调用汇编的方法。最后 *test.asm* 里面直接调用这些过程。*printpos* 函数定义在 *str.c2* 中。注意 c 语言的函数调用约定，通过栈基址寄存器 *bp* 获得参数。

test.asm 中只调用了 2 中的 *printmsg* 函数。

接下来使用 Makefile 生成 *test.img*。

代码 3: 生成测试镜像文件

```
1 make testimg
```

用 bochs 运行，bochs 的配置可以参考附录。使用 bochs 的原因是 vmware 在这次实验中的显示出现不少问题，bochs 均能正常显示。

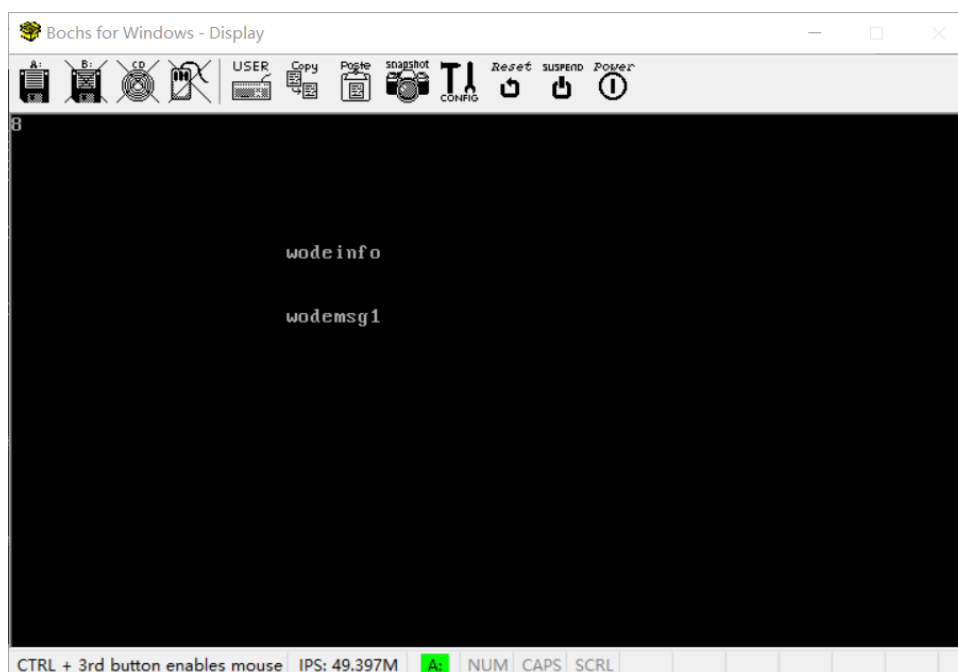


图 2: 运行过程截图

5.2 修改引导程序

在上一次实验中，我已将内核从加载程序中独立出来，但是引导程序中将四个用户程序预先加载到了内存中，按照我的设计，用户程序由用户自己载入并运行，因此引导程序中去掉那四个加载扇区的代码即可，这里不放出代码。由于内核需要丰富的功能，而汇编相对于 c 语言难以实现复杂功能，因此虽然我使用了汇编文件作为内核，引导程序将跳转至此，但是内核文件仅完成两件事

1. 打印欢迎语句
2. 在用户按下回车键后跳转到 c 程序中

获得输入用到了 BIOS 的 16 号 0 功能调用，代码来自上次实验。整个内核的代码如下：

Code 4: kernel.asm

```
1 BITS 16
2 extern Greet
```



```

3 extern shell
4
5 global _start
6 _start:
7     call dword Greet
8 WaitForKey:
9     mov ah, 0
10    int 16h
11    cmp al, 0dh
12    jne WaitForKey
13    call dword shell
14    jmp WaitForKey

```

5.3 设计库过程

由于此次项目的函数较多，为了更加清晰的说明函数的设计，附录中给出了库中用到的变量表2。所有 c 代码均放在 `mystring.h`，汇编代码均放在 `lib.asm` 中，交互模块所用的函数在 `interface.h` 中。

为了让内核有更加丰富的功能，首先要让 io 操作能够容易进行，也就是封装。io 操作包括输入和输出，这两项依赖于汇编，在建立了基本 io 操作的基础上，需要 c 语言来完成简单的字符串操作。

5.3.1 获得输入

获得输入是 bios 调用的基本操作，直接使用调用 16 号中断即可，代码请看12

5.3.2 拆分字符串

该函数用于解析用户的输入，通过遍历字符与空格比对，将对应字符分别赋值到传入的字符串地址中。代码请看11

5.3.3 比较字符串

该函数用于解析用户的输入，用通过遍历字符直到找到'\0' 或不同的字符，将此处的字符 ASCII 码之差返回。代码请看13

5.3.4 计算字符串长度

该函数在测试部分已实现，请参考2

5.3.5 输出字符串

输出字符串有两种，分别是在特定位置输出，用于打印欢迎页面等一次性功能，另一种是在光标处输出，这种用的最多。第一种和老师之前提供的代码没有太大差异，注意按照 `_cdecl` 的调用约定，所有变量在压栈时使用 4 个字节，`pusha` 压栈 8 个寄存器，共 16 个字节。代码同5。

5.3.6 输出字符

输出字符只需考虑光标处，使用 BIOS 的 10 号 0E 功能中断。代码如下：

Code 5: putchar

```
1 putchar:
2     pusha
3     mov bp, sp
4     add bp, 16+4
5     mov al, [bp]
6     mov bh, 0
7     mov ah, 0Eh; 功能码
8     int 10h
9     popa
10    retf
```

5.3.7 清屏

清屏是交互界面必须的功能，调用 BIOS 的 10 号 3 中断完成。代码如下：

Code 6: 字符串输出宏

```
1 cls:
2     pusha
3     mov ax, 0003h
4     int 10h
5     popa
6     ret
```

以上便是此次实验需要的所有字符相关的 IO 功能。

5.4 监控程序修改

现有的监控程序只能输入四个字符，为了扩展，我模仿了 zsh 的界面，能够获取用户输入的命令。进入界面时会输出欢迎用的字符串，等待输入时会打印“->”字符。我添加了三个函数，分别是读取输入的 readbuff 和 greet 和 prompt。后两个只是调用写好的 printpos 直接输出字符串，所以不作说明。

5.4.1 获得输入-readbuff 函数

由于 BIOS 并没有提供获取键盘输入同时显示对应字符并控制光标的功能，需要自己实现，读取时总共有三种情况，

1. 缓冲区中有字符（串）
2. 缓冲区已满
3. 缓冲区为空

首先检查传入的字符串是否为空，初始化缓冲区长度变量，然后用定义好的 getch 获得字符。接下来进入循环。

循环体内 首先检查是否输入了非法字符，按照三种情况依次判断。合法字符为键盘上可以显示为字符的键和回车，空格，退格和 **ctrl-c**。若为非法字符重新开始循环，离开循环。若为回车，将字符串长度置为零，代表之前的输入无效，然后离开循环。

缓冲区中有字符(串) 若输入的字符是退格，则按顺序输出退格空格退格，理由是退格会使光标退后而空格才会改变屏幕上的内容。不为退格则显示输入的字符，并将字符放入缓冲区，修改缓冲区长度加一。

缓冲区已满 此时只能退格，处理方式请参考第一段。

缓冲区为空 此时只能输入，处理方式请参考第一段。

离开循环体 此时需要换行，分别输出 `'\r'`，`'\n'`，已达到换行的效果。`'\n'` 并不会将光标放回屏幕左侧。代码如下：

Code 7: readbuff

```
1 void readbuff(char *buff, int maxLength)
2 {
3     if (!buff)
4         return;
5     int len = 0;
6     while(1) {
7         char c = getch();
8         if (!(c==0xD || c=='\b' || c==3 || c>=32 && c<=127)) { continue; }
9         if (c==0x0D){
10             break;
11         }
12         else if (c==3){
13             len = 0;
14             break;
15         }
16         if (len > 0 && len < maxLength-1) {
17             if (c == '\b') {
18                 putchar('\b');
19                 putchar('_');
20                 putchar('\b');
21                 --len;
22             }
23             else {
24                 putchar(c);
25                 buff[len] = c;
26                 ++len;
27             }
28         }
29         else if (len >= maxLength-1) {
30             if (c == '\b') {
31                 putchar('\b');
```

```

32         putchar(' ');
33         putchar('\b');
34         --len;
35     }
36 }
37 else {
38     if(c != '\b') {
39         putchar(c);
40         buff[len] = c;
41         ++len;
42     }
43 }
44 }
45 putchar('\r'); putchar('\n');
46 buff[len] = '\0';
47 }

```

5.4.2 其他功能

为了完善交互体验，本程序提供关机功能，本质上是通过写端口实现的，代码如下：

Code 8: 强制关机

```

1 shutdown:
2     mov ax, 2001H
3     mov dx, 1004H
4     out dx, ax

```

5.4.3 交互模块-shell 函数

shell 函数负责循环获取用户输入并处理。该监控程序提供了五个功能，分别是：

1. help: 提示用户可以输入的内容
2. cls: 清屏
3. shutdown: 关机
4. run: 运行程序
5. ls: 显示可运行的程序信息

这几个功能被定义为枚举类型，并且有对应的字符串数组用于存放名字以与用户输入比对。函数开头还定义了用户输入缓冲区，命令字符串和目标字符串。以及用于处理 ctrl-c 的空字符串，专门使用局部变量存放的理由是，整个项目中无法使用常量字符串，理由未知。使用后会访问不可访问的地址导致死机。

5.4.4 交互逻辑

进入循环前清屏，在循环开始时打印提示符 “->”，然后使用 *readbuff* 函数获得输入，依次与 *commands* 中的字符串使用 *strcmp* 比对，相等时调用对应的功能函数。而最后要判断是否输

入了 `run`, 是则检查输入的格式, 格式用正则表达式可以表示为 “`run\s+[1234]+`”, 如果格式符合, 则进行处理。处理过程请看下一节。

5.5 设计用户程序的加载方式

由于时间原因, 我设计的加载方式是用户能够通过命令按照自己输入的数字顺序自动运行程序。

为了使用户能够看到可以加载的程序信息, 我定义了结构体 `sector`, 和对应的数组 `Comlist` 用于装程序的信息。在检查 `run` 命令的格式正确后, `target` 字符串中放入了代表运行顺序的数字串, 用 `for` 循环依次将其转为数字并调用运行函数。

5.5.1 运行用户程序-loadUsrProgram

该函数定义在汇编代码中, 传入的参数分别是程序的柱面号, 磁头号, 起始扇区号, 扇区数, 起始扇区号等信息, 调用时这些信息都在栈里, 加载用户程序的代码参考老师第二次提供的引导代码, 调用的参数偏移量依次加 4, 读入扇区后, 需要跳转到程序中运行。这里有三个编程时非常需要注意的点:

堆栈的对齐 由于使用的是 c 语言的 `_cdecl` 的调用约定, 参数的栈空间释放工作由 c 语言编译出来的汇编完成, 但是 `call` 和 `ret` 这样隐式压栈和出栈需要汇编自己安排, 考虑到可能会出现跨段的问题, 我使用了 `call far, call dword, ret`, 那么每次都以四个单位入栈和出栈 `CS` 和 `IP` 是没有问题的, 由于程序有一定的代码量, 编写代码者必须保证每个过程对栈的空间维护都是统一的。

远调用 在进入该函数前, 用户程序目前并未放入内存, 而所有信息都作为参数被传入, 所以进入用户程序时需要自己手动计算段地址和偏移地址, 并使用 `call far` 指令跳转, 这是唯一能够直接同时修改 `CS`, `IP` 并且地址能够来自内存的指令。地址可以来自内存意味着是可变的, 只要在下方定义一个四字节的变量, 在函数进行时用参数为其赋值, 就能用来作为 `call far` 的参数, 这里还有一点需要注意, `call far` 会从参数中分离出 `CS`(后两个字节), `IP`(前两个字节), 而传入的参数是一个完整的四字节偏移地址 (例: `0x0A100`), 需要用寄存器将其分离并移位得到段寄存器, 计算公式为

$$ds = ([bp + 20] >> 4) - 10 \quad (1)$$

其中 `[bp+20]` 是传入的第一个参数, 因为 `pusha` 压栈了 16 个字节。得到后将两个参数分别放入内存中名为 `program` 的地址处, 注意前两个字节是 `IP`, 后两个字节是 `CS`。然后 `call far [program]`, 这样就通过间接寻址完成远调用。

段寄存器的保护 由于运行用户程序的时候有跨段, 而用户程序中是否修改了段寄存器是无法知道的, 因此在从用户程序返回时, 需要将 `ds, es` 复原, 否则之后的程序将无法从数据段取出正确数据。复原的方式可以有两种, 压栈然后出栈, 或者直接在用户程序返回后将 `cs` 借 `ax` 赋值给 `ds` 和 `es`, 我使用的是第二种方法。

注意好以上三点, 便可以写出正确的代码。代码请看14:

5.6 修改用户程序

用户程序和第二次的基本一致，唯一的区别在于程序开始时将常用寄存器压栈，离开时出栈，返回时使用 `retf`。

5.7 软盘扇区安排

因为现在的用户程序由内核来加载，只需要将程序放入镜像的正确位置就可以了。安排请看表1：

5.8 镜像文件生成

使用 `gcc` 生成 `c` 源文件对应的汇编，`nasm` 生成内核汇编源文件和库过程汇编源文件的 `elf` 格式的汇编，使用 `ld` 链接可以生成二进制文件，将引导程序，内核程序和用户程序放入镜像中的合适位置即可生成镜像。

代码相关信息请看 `readme.md`。

在 `wsl` 或 `linux` 的 `shell` 中执行 `make all/make img` 即可生成镜像 (`.img`) 文件，在 `VMware` 中加载该镜像，查看结果。到此，实验结束。

6 程序使用说明

6.1 实验环境

6.1.1 VMware Workstation 15

`VMware Workstation` 是 `VMware` 公司推出的一款桌面虚拟计算软件，具有 `Windows`、`Linux` 版本。此软件可以提供虚拟机功能，使计算机可以同时运行多个不同操作系统。此次实验中用于运行镜像文件。

6.1.2 nasm 2.13.02

`Netwide Assembler` 是一款基于英特尔 `x86` 架构的汇编与反汇编工具。在此次实验中用生成二进制文件 (`.bin`)

表 1: 软盘扇区安排

磁头号	扇区号	扇区数 (大小)	内容
0	1	1 (512 B)	引导程序
0	2~17	16 (8 KB)	操作系统内核
1	1~2	2 (1 KB)	用户程序 1(LU.com)
1	3~4	2 (1 KB)	用户程序 2(LD.com)
1	5~6	2 (1 KB)	用户程序 3(RU.com)
1	7~8	2 (1 KB)	用户程序 4(RD.com)

6.1.3 wsl

本次实验中用到了 shell 文件以批处理源代码生成镜像，整合于 windows 系统的 wsl 能够很方便的处理在 windows 系统上生成的文件。

6.1.4 VSCode 1.44.2

vscode 能够方便的使用 wsl 和 latex 的 pdf 生成功能。加载虚拟机以外的所有任务都可以通过 vscode 完成。

6.2 编译方法

6.2.1 系统要求

生成镜像文件时，可以使用 linux 操作系统或者带 wsl 的 windows 系统。

6.2.2 编译过程与参数

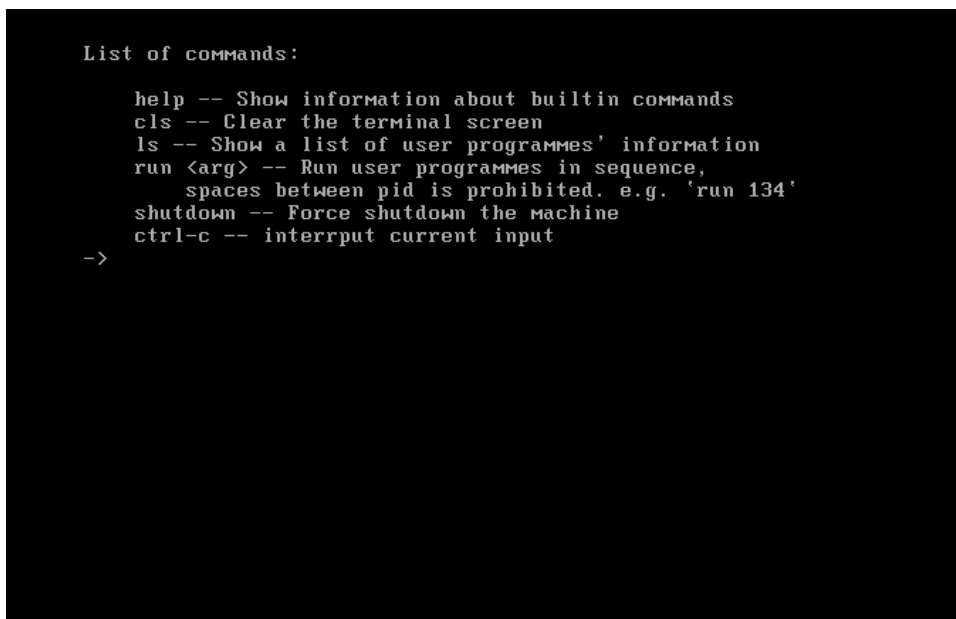
在源代码目录下使用 wsl，执行下列代码即可得到镜像文件 os17338233.img。

代码 9: 生成镜像

```
1 make img
```

6.3 运行与演示

由于篇幅原因，演示部分将主要放在已录制好的 OSdemo.mkv 中。进入内核的界面如图 3 所示。



```
List of commands:
  help -- Show information about builtin commands
  cls -- Clear the terminal screen
  ls -- Show a list of user programmes' information
  run <arg> -- Run user programmes in sequence,
             spaces between pid is prohibited. e.g. 'run 134'
  shutdown -- Force shutdown the machine
  ctrl-c -- interrupt current input
->
```

图 3: 运行过程截图

具体的演示过程可以查看已录制好的 OSdemo.mkv

7 总结与讨论

7.1 特色, 不足与改进

本程序提供了较为简洁可用的命令行, 支持用户自己载入并按照自己指定的顺序运行程序。为了避免用户恶意修改用户程序的地址, 我没有提供修改用户程序信息的功能, 命令行能够为用户提供相对清晰的信息。不足之处是用户程序信息是自己手动写入, 无法自动扫描。

7.2 收获

本次实验是非常考验汇编和 c 的编程能力的, 尤其是在 debug 方面, 由于汇编会修改上下文, 而 c 用到了很多底层 IO 操作, 汇编后又不容易看出代码结构, 因此单元测试各个模块尤其重要。这次实验我不仅掌握了混合编程的约定, 也加强了 c 的字符串操作的编程能力, 还加深了对 c 语言编译后的汇编文件的理解。此次实验我也学习了如何编写 Makefile 文件 [6], 大大提高了生成项目目标的效率。最大的收获是关于 bochs, 在同学的帮助下, 我学会了如何使用 bochs 对内核进行调试, 重要的问题都通过 bochs 解决。解决例子会在感想中说明。同时, 在本报告的编写过程中, 我也练习了利用 L^AT_EX 编写文档的能力。[1, 2]

7.3 感想

作为操作系统的第三个实验, 代码量和第二次相比又有较大的增加, 并且完成的功能也更加复杂, 要自学的东西翻了好几倍, 包括环境配置, 编译和链接的方法, 混合编程的约定, 以及内核的调试。

由于网上的资料并没有特别详细的介绍如何生成裸机使用的二进制文件, 编译链接的方法我是通过不断尝试并与同学交流才知道的。

实验的每个步骤我都遇到了不少问题, 除了与同学探讨外, 《操作系统真象还原》[7] 这本书也提供了非常大的帮助, 对我对汇编, 调用约定以及实模式, 保护模式的理解都有非常大的帮助。里面关于 bochs 的演示过程激励我学习 bochs 的使用方法, 在同学的帮助下, 我也掌握了 bochs 的调试方法。可以说, 这次实验除非把所有汇编和 c 的原理都完全吃透, 否则内核调试几乎是必须的, 因为代码量太大。仅仅看到虚拟机的显示结果是几乎不能看出问题的。

在写混合编程测试代码时, 我就遇到了一个严重的问题, 所有代码都没有问题, 但是 vmware 虚拟机显示不正确, 与同学探讨后才得知, vmware 运行某些镜像会有问题, 而 bochs 不会, 我尝试改用 bochs 运行, 果然就没有遇到。但是后面我完成整个项目后, 在 vmware 上也是能正常显示的, 目前并不知道原因。

通过混合编程理解了调用约定后, 完成整个项目的大体工作只花了数小时。其中时间主要花在读取用户输入上, 由于我之前对各种逃逸字符的理解还不到位, 运行镜像发现屏幕并没有按照我的想法来显示, 查找了相关资料才改正过来。

最大的问题毫无疑问是用户程序的加载，这部分是纯汇编代码，由 c 语言传入结构体的各个分量后调用，我花了大约一整天的时间来解决这个问题，由于代码的问题分散在三处，在我使用 bochs 之前，我尝试修改了也没法获得正确的结果，所以我在凌晨时才决定使用 bochs 进行调试，之所以之前没有使用，是因为 bochs 中出现了过多 c 编译产生的汇编代码，阅读起来及其费力，与 objdump 生成的文件不同，bochs 的反汇编完全不显示所在的模块，因此一旦离开我熟悉的纯汇编代码，cpu 执行的语句是属于哪个模块我都无法确定。因此在我决定使用 bochs。bochs 的调试方式和 gdb 相似，在 windows 系统上有单独的版本。为了运行镜像，首先需要设置配置文件，包括镜像路径，磁盘类型等，然后运行 bochsdgb。配置文件可以在 src 目录下找到。

调试过程 我首先在 0x8100(内核程序起点)处设置断点，并使程序执行到此，然后反汇编了约 1400 行代码（图4），搜索到纯汇编所在的位置（图5）（图6），在汇编代码开头设置断点（图7），继续，然后在窗口中输入命令执行。然后查看所有寄存器的值，包括段寄存器（图8）。然后设置断点到 retf 语句，再次检查，发现段寄存器在 retf 前已经被修改了，再次设置汇编程序开头为断点，继续，执行到第二个程序时，查看段寄存器，依旧是被修改过的状态，这样就知道了其中有一个问题在于 cs，而两次检查时发现了一个新问题，那就是栈顶 sp 发生了变化，差距为 2，于是我重新启动 bochs，在执行完第一个程序后，停留在返回的 ret 语句，并开始逐步运行代码后，然后将代码复制出来，与 gcc 生成的文件通过 objdump 反汇编得到的文件一一比对语句，终于找到了对应的模块。之所以这么做，是因为我当时并不能确定离开了汇编后是否执行了下一条语句，可能是因为 vmware 的问题，我在调用 loadUsrProgram 后面即使加上了输出字符串的语句，在虚拟机中也没有显示。比对后得知进入了 cls 模块。于是我发现在调用纯汇编代码 cls 的子过程返回时，栈顶并未正常恢复，是因为代码没有使用 retf。修改后终于正常运行了程序。而第三个问题是我的 ls，我发现执行了汇编的代码后，我的全局变量居然被修改了，而我的汇编代码中并没有访问到全局变量，使用的都是压栈的参数，c 中也只是传入参数后调用了函数而已。于是我将全局变量修改为 const，终于解决了所有问题。

实际的实验过程远比上述过程所说的复杂，因为如何找到 c 语言中的对应模块和寻找我所写的纯汇编代码的方法并不是一开始就知道，而是不断摸索才发现可以这么做的。相信这次实验的收获会对以后的实验有很大的帮助。

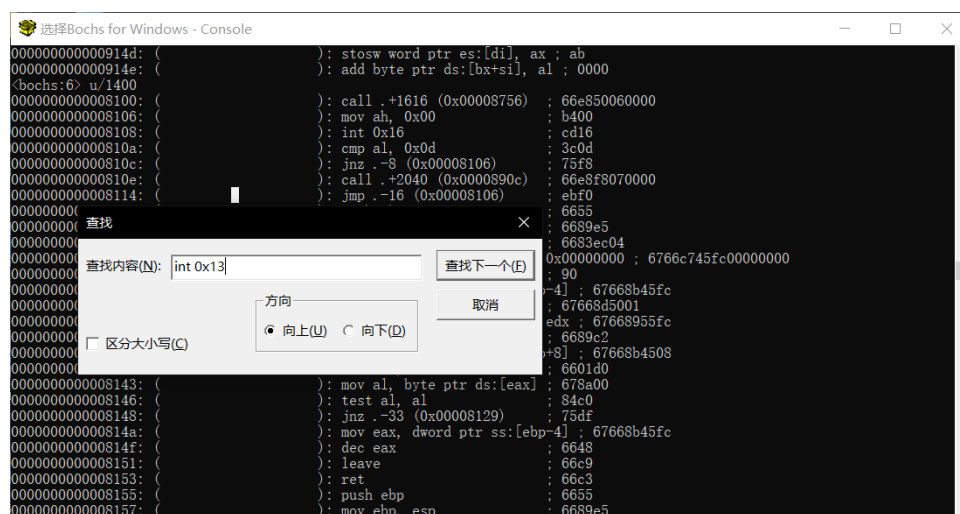


图 4: 用 bochs debug 过程 1

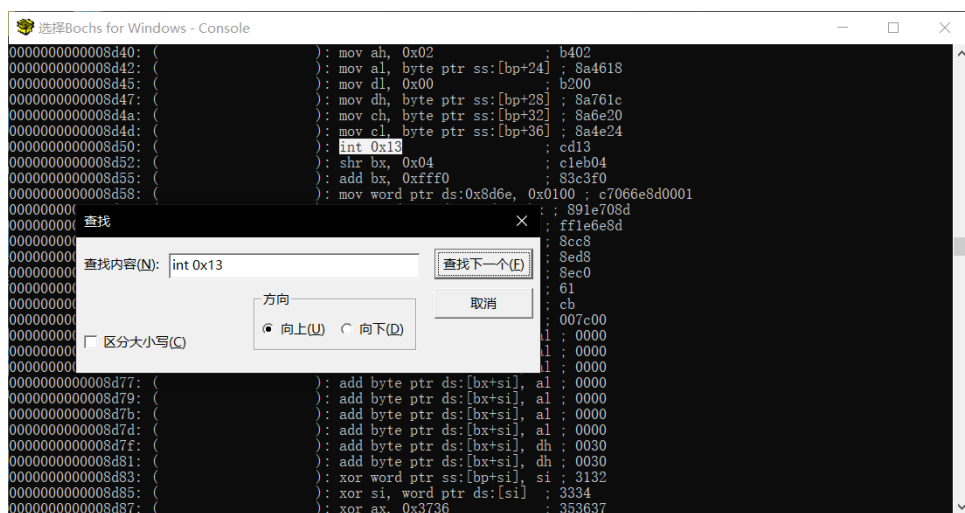


图 5: 用 bochs debug 过程 2

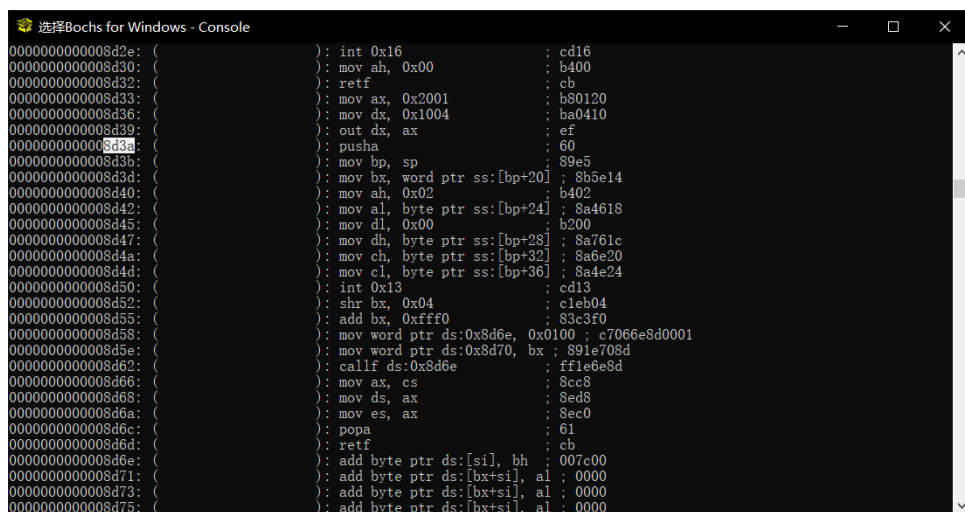


图 6: 用 bochs debug 过程 3

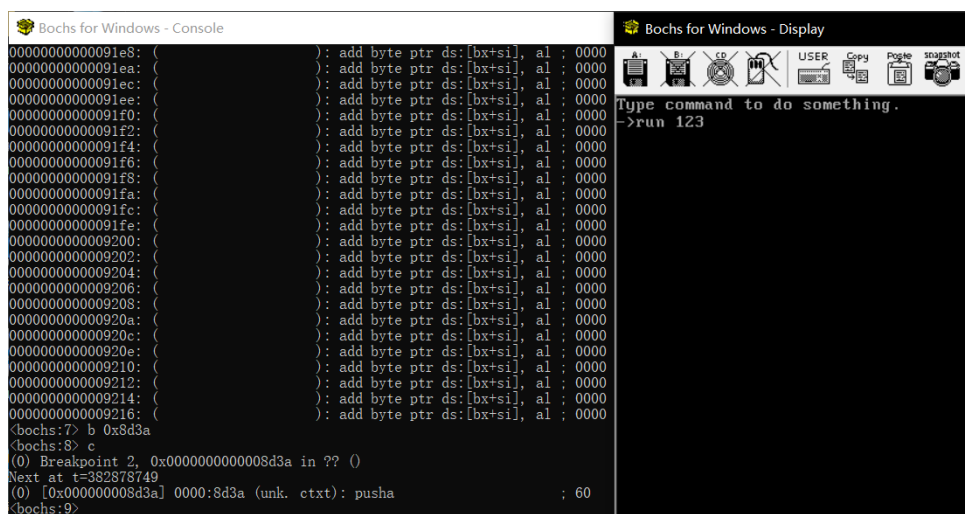


图 7: 用 bochs debug 过程 4

```

<bochs:9> sreg
es:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
cs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ss:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ds:0x0000, dh=0x00009300, dl=0x0000ffff, valid=7
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
fs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
gs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ldtr:0x0000, dh=0x00008200, dl=0x0000ffff, valid=1
tr:0x0000, dh=0x00008b00, dl=0x0000ffff, valid=1
gdtr:base=0x000000000000f9af7, limit=0x30
idtr:base=0x0000000000000000, limit=0x3ff
<bochs:10> r
rax: 00000000_0000a100
rbx: 00000000_00000001
rcx: 00000000_00000002
rdx: 00000000_00000000
rsp: 00000000_0000ff1a
rbp: 00000000_0000ffce
rsi: 00000000_00000000
rdi: 00000000_00000001
r8 : 00000000_00000000
r9 : 00000000_00000000
r10: 00000000_00000000
r11: 00000000_00000000
r12: 00000000_00000000
r13: 00000000_00000000
r14: 00000000_00000000
r15: 00000000_00000000
rip: 00000000_00008d3a
eflags 0x00000006: id vip vif ac vm rf nt IOPL=0 of df if tf sf zf af PF cf
<bochs:11>

```

图 8: 用 bochs debug 过程 5

References

1. Leslie Lamport, *L^AT_EX: a document preparation system*, Addison Wesley, Massachusetts, 2nd edition, 1994.
2. Contributors to Wikibooks, *LaTeX Bibliography Management*. Wikibooks, 2019., en.wikibooks.org/wiki/LaTeX/Bibliography_Management.
3. Virtual 8086 mode *Virtual 8086 mode*, *Wikipedia, The Free Encyclopedia* 2019., https://en.wikipedia.org/wiki/Virtual_8086_mode
4. Virtual 8086 mode 保护模式, *Wikipedia, The Free Encyclopedia* 2019., <https://zh.wikipedia.org/wiki/>
5. Virtual 8086 mode 实模式, *Wikipedia, The Free Encyclopedia* 2019., <https://zh.wikipedia.org/wiki/>
6. how-to-write-makefile 跟我一起写 *Makefile* 陈皓 2020., <https://seisman.github.io/how-to-write-makefile/overview.html>
7. Leslie Lamport, *操作系统真象还原*, 郑钢, 人民邮电出版社 2016.,

表 2: 项目变量表

变量声明	内容	类型	功能
checknum	检查是不是合法数字（是数字且在范围内）	函数	格式检查
find	寻找字符串中特定字符的位置	函数	字符串操作
isnum	判断是不是数字	函数	格式检查
printchar	在光标处重复打印同一字符	函数	IO
printPos	在某一位置打印字符串	函数	IO
putchar	在光标处打印单个字符	函数	IO
split	将字符串在第一个空格处分开成两个	函数	字符串操作
strcmp	比较两个字符串的大小	函数	字符串操作
strlen	计算字符串长度	函数	字符串操作
cls	清屏	汇编过程	IO
getch	获得一个键盘输入	汇编过程	IO
shutdown	强制关机	汇编过程	特殊功能
loadUsrProgram	加载并运行用户程序	汇编过程	IO
Greet	打印欢迎信息	函数	交互
middlecol	计算居中打印的位置	函数	
printDetail	打印软盘中用户程序信息	函数	IO
printInfo	打印欢迎提示信息	函数	IO
printMiddle	居中打印字符串	函数	IO
prompt	打印'->'	函数	交互
readbuff	获取用户回车前的完整输入	函数	IO
newline	新起一行	宏	IO
OffsetOfUserPrg 1	用户程序 1 地址偏移量	宏	保存信息
OffsetOfUserPrg2	用户程序 2 地址偏移量	宏	保存信息
OffsetOfUserPrg3	用户程序 3 地址偏移量	宏	保存信息
OffsetOfUserPrg4	用户程序 4 地址偏移量	宏	保存信息
sector	用户程序信息表结构体	结构体	保存信息

8 附录

Code 10: getch

```

1 getch:
2     mov ah,0
3     int 16h
4     mov ah,0
5     retf

```

Code 11: split

```

1 void split(char*buff, char*dest1, char*dest2){
2     dest2[0] = '\0';

```

```

3   int len = strlen(buff);
4   int n = find(buff, len, '_');
5   if (n==len-1)
6       n = len;
7   int i;
8   for (i = 0; i < n; i++)
9   {
10      dest1[i] = buff[i];
11  }
12  dest1[i] = '\0';
13  while (buff[i]&&buff[i]!='_')
14  {
15      i++;
16  }
17  int j = 0;
18  while(buff[i]){
19      dest2[j++] = buff[i++];
20  }
21  dest2[j] = '\0';
22  }

```

Code 12: getch

```

1  int find(char *buff, int len, char c)
2  {
3      if (buff[len] != '\0')
4          return 0;
5      int i = 0;
6      while (i < len && buff[i++] != c)
7          ;
8      return i - 1;
9  }

```

Code 13: strcmp

```

1  int find(char *buff, int len, char c)
2  int strcmp(char *lhs, char *rhs)
3  {
4      int i = 0;
5      while (1)
6      {
7          if (lhs[i] == '\0' || rhs[i] == '\0')
8              break;
9          if (lhs[i] != rhs[i])
10             break;
11         ++i;
12     }
13     return lhs[i] - rhs[i];
14 }

```

Code 14: 强制关机

```

1  loadUsrProgram:
2      pusha
3      mov bp, sp
4      mov bx, [bp+20]          ; 偏移地址; 存放数据的内存偏移地址
5      mov ah, 2                ; 功能号
6      mov al, [bp+24]          ; 扇区数
7      mov dl, 0                ; 驱动器号; 软盘为0, 硬盘和U盘为80H
8      mov dh, [bp+28]          ; 磁头号; 起始编号为0
9      mov ch, [bp+32]          ; 柱面号; 起始编号为0
10     mov cl, [bp+36]           ; 起始扇区号 ; 起始编号为1
11     int 13H; 调用读磁盘BIOS的13h功能
12     shr bx, 4
13     add bx, -10h
14     mov word[program], 0x100
15     mov word[program+2], bx
16     call far [program]
17     mov ax, cs
18     mov ds, ax
19     mov es, ax
20     popa
21     retf

```

Code 15: str.o

```

1  .file "str.c"
2  .code16gcc
3  .text
4  .globl msg
5  .section .rodata
6  .LC0:
7      .string "wodemsg1"
8      .data
9      .align 4
10     .type msg, @object
11     .size msg, 4
12 msg:
13     .long .LC0
14     .text
15     .globl strlen
16     .type strlen, @function
17 .LFE0:
18     .size strlen, .-strlen
19     .section .rodata
20 .LC1:
21     .string "wodeinfo"
22     .text
23     .globl printInfo
24     .type printInfo, @function

```

```

25 .LFE1:
26     .size    printInfo , .-printInfo
27     .globl   printmsg
28     .type    printmsg , @function
29 printmsg:
30 .LFB2:
31     .cfi_startproc
32     pushl    %ebp
33     .cfi_def_cfa_offset 8
34     .cfi_offset 5, -8
35     movl     %esp, %ebp
36     .cfi_def_cfa_register 5
37     subl     $8, %esp
38     movl     $9, -4(%ebp)
39     movl     $23, -8(%ebp)
40     movl     msg, %eax
41     pushl    %eax
42     call     strlen
43     addl     $4, %esp
44     movl     %eax, %edx
45     movl     msg, %eax
46     pushl    -8(%ebp)
47     pushl    -4(%ebp)
48     pushl    %edx
49     pushl    %eax
50     call     printPos
51     addl     $16, %esp
52     nop
53     leave
54     .cfi_restore 5
55     .cfi_def_cfa 4, 4
56     ret
57     .cfi_endproc
58 .LFE2:
59     .size    printmsg , .-printmsg
60     .ident    "GCC:(Ubuntu7.5.0-3ubuntu1~18.04)7.5.0"
61     .section    .note.GNU-stack , "" ,@progbits

```