

# 中山大学数据科学与计算机学院本科生实验报告

## (2020 学年春季学期)

课程名称：操作系统实验

任课教师：凌应标

助教：

年级&班级	2018 级 1 班	专业(方向)	计算机科学与技术(大数据方向)
学号	18308045	姓名	谷正阳
电话	13355426001	Email	<a href="mailto:Guzy0324@163.com">Guzy0324@163.com</a>
开始日期	2020.5.14	完成日期	2020.5.14

### 一、实验题目

操作系统 实验 3

### 二、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

### 三、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

## 四、实验内容：

### 1. 实验步骤

(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，在实验报告中描述这些工作。

(2) 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) 重写实验二程序，实验二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 1.44MB 软盘映像中，保存到特定的几个扇区。利用汇编和 c 程序混合编程监控程序命令保留原有程序功能，如可以按操作选择，执行一个或几个用户程序、加载用户程序和返回监控程序；执行完一个用户程序后，可以执行下一个。

(4) 利用汇编和 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。

(5) 重写引导程序，加载 COM 格式程序的独立内核。

(6) 拓展自己的软件项目管理目录，管理实验项目相关文档

## 2. 实验原理

1. c 语言转 nasm: <https://github.com/gitGNU/objconv>

2. 关于指令, 伪代码和地址的长度

<https://www.cnblogs.com/youxia/p/linux008.html>

3. 物理扇区和逻辑扇区的转化, int13 的深度使用:

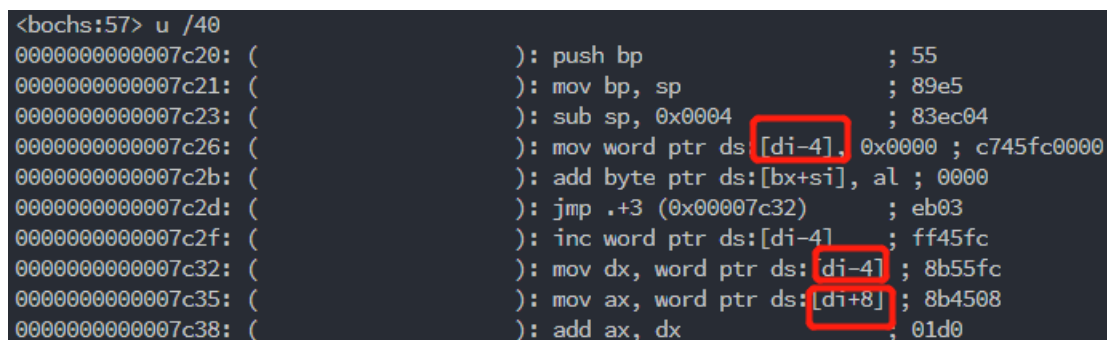
<https://blog.csdn.net/jltxgcy/article/details/8687881>

## 五、实验结果

(1) 我选择了 nasm 和 gcc 组合。

1) 按照老师给的样例代码, 我进行了实验, 发现以下两点: 1. gcc 编译出的函数名前需要加\_, 2. ld 连接时的参数应换为 elf\_32, 因为 elf 格式和 pe 格式是完全不同的, elf 格式用于 UNIX, pe 格式用于 Windows。而前两个的编译参数都指明是 elf 格式。

2) 成功运行, 并且显示字符串, 但是结果却是小写并未改成大写。而后我又编写了用来计算字符串长度的 count.c 文件, 发现了同样的问题, c 语言的函数不好用。通过 bochs 查看代码, 发现在 c 的函数中, 它使用了从未初始化且毫无道理的 di 寄存器和 si 寄存器等等, 如图:



```
<bochs:57> u /40
000000000007c20: (           ): push bp                ; 55
000000000007c21: (           ): mov bp, sp              ; 89e5
000000000007c23: (           ): sub sp, 0x0004          ; 83ec04
000000000007c26: (           ): mov word ptr ds:[di-4], 0x0000 ; c745fc0000
000000000007c2b: (           ): add byte ptr ds:[bx+si], al ; 0000
000000000007c2d: (           ): jmp .+3 (0x0007c32)      ; eb03
000000000007c2f: (           ): inc word ptr ds:[di-4]   ; ff45fc
000000000007c32: (           ): mov dx, word ptr ds:[di-4] ; 8b55fc
000000000007c35: (           ): mov ax, word ptr ds:[di+8] ; 8b4508
000000000007c38: (           ): add ax, dx               ; 01d0
```

经过老师的点播和同学的指引, 我终于知道了这个是 bochs 把机器码解释作是 16 位, 而实际上是 32 位的 nasm 的原因。因而需要将参数改为 -m16。而且需要在 gcc 开头加上

通过\_\_ASM()\_\_嵌入汇编的方式嵌入伪代码” .code16” 才能生成 16 位代码

3) 我发现我的 MinGW 版本的 gcc 不支持 -m16 参数, 因而转向 Linux。之前在家里的老旧电脑上装过 Linux, 使用强度并不大, 因而工具链不是很完备。所以我找到方案: 在 Linux 上编译, 在 Windows 上运行。由于两台电脑使用 U 盘或者邮件传文件效率很低, 于是我在 Linux 上搭建了一个内网的 SSH 服务器, 这样不仅可以通过 SFTP 方便的传送文件, 而且可以使用 SSH 客户端控制 Linux 的命令行来做到远程控制。

编译参数如下:

```
guzhengyang@guzhengyang-Vostro-14-5480:~/Documents/codes/实验3$ gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -c upper.c -o upper.o -fno-pie
guzhengyang@guzhengyang-Vostro-14-5480:~/Documents/codes/实验3$ nasm -f elf32 showstrn.asm -o showstrn.o
guzhengyang@guzhengyang-Vostro-14-5480:~/Documents/codes/实验3$ ld -m elf_i386 -N showstrn.o upper.o -Ttext 0x7c00 --oformat binary -o boot.bin
```

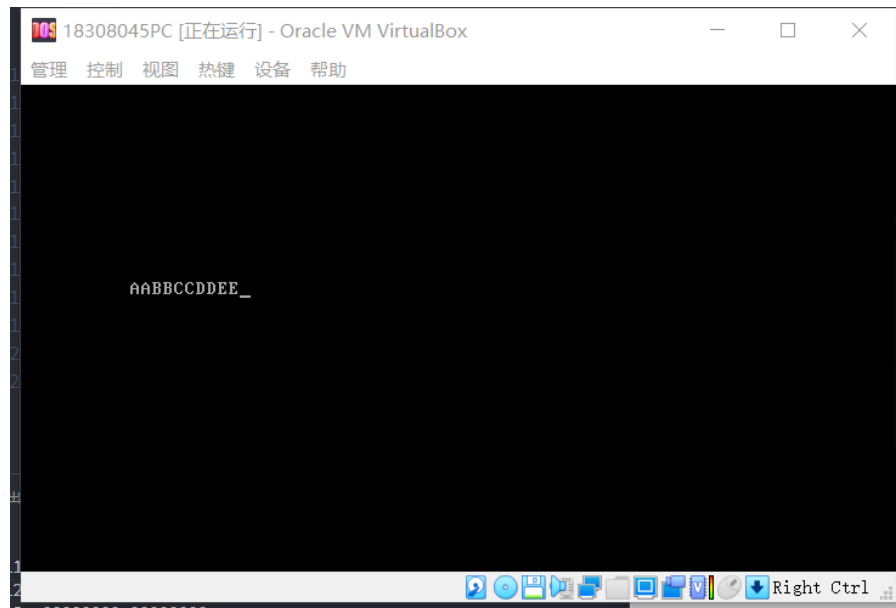
其中 -fno-pie 是经过群友的点播, 为了防止报错:

undefined reference to '\_\_GLOBAL\_OFFSET\_TABLE'

4) 发现在 bochs 中仍然不能运行, 在 c 函数 ret 后跳到了很诡异的地方, 然而在 VirtualBox 中却能够正确运行。从群中群友的只言片语中, 我获得了一个信息: 可能要使用 call dword 而不是 call, 结果真的在 bochs 中运行成功了。观察两者区别如下:

A. 使用 call:

VirtualBox:



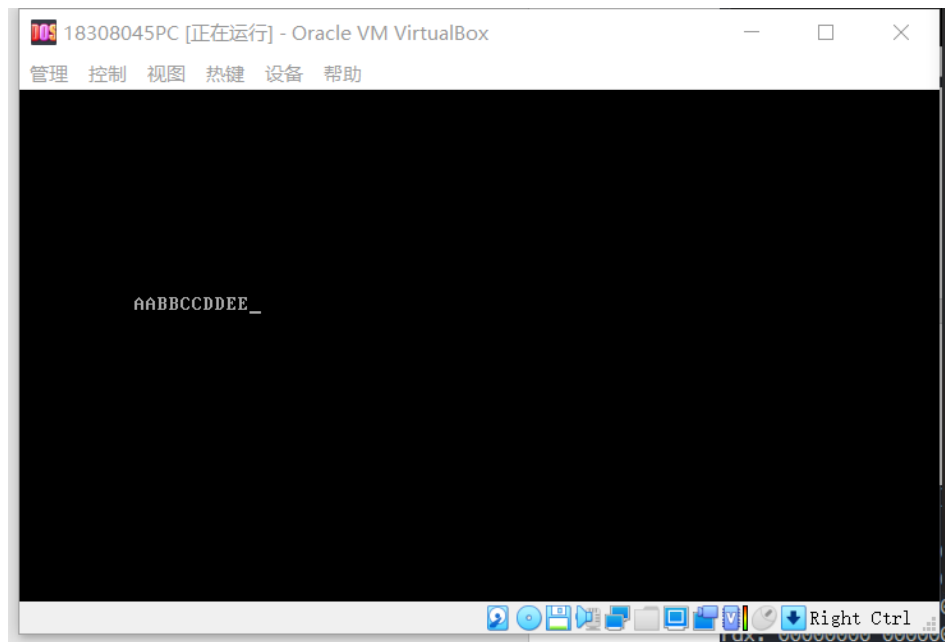
Bochs 中 call 前寄存器状态: 注意观察 sp 寄存器

```
(0) [0x000000007c08] 0000:7c08 (unk. ctxt): call .+24 (0x00007c23) ; e81800
<bochs:7> r
rax: 00000000_60000000
rbx: 00000000_00000000
rcx: 00000000_00090000
rdx: 00000000_00000000
rsp: 00000000_0000ffd6
rbp: 00000000_00000000
rsi: 00000000_000e0000
rdi: 00000000_0000070c
r8 : 00000000_00000000
r9 : 00000000_00000000
r10: 00000000_00000000
r11: 00000000_00000000
```

call 后: sp -= 2

```
(0) [0x000000007c23] 0000:7c23 (unk. ctxt): push ebp ; 6655
<bochs:9> r
rax: 00000000_60000000
rbx: 00000000_00000000
rcx: 00000000_00090000
rdx: 00000000_00000000
rsp: 00000000_0000ffd4
rbp: 00000000_00000000
rsi: 00000000_000e0000
rdi: 00000000_0000070c
r8 : 00000000_00000000
r9 : 00000000_00000000
r10: 00000000_00000000
r11: 00000000_00000000
```

B. 使用 call dword:



call dword 后: sp -= 4

```
(0) [0x000000007c26] 0000:7c26 (unk. ctxt): push ebp ; 6655
<bochs:9> r
rax: 00000000_60000000
rbx: 00000000_00000000
rcx: 00000000_00090000
rdx: 00000000_00000000
rsp: 00000000_0000ffd2
rbp: 00000000_00000000
rsi: 00000000_000e0000
rdi: 00000000_0000070c
r8 : 00000000_00000000
r9 : 00000000_00000000
r10: 00000000_00000000
r11: 00000000_00000000
```

ret 后: sp += 4

```
(0) [0x000000007c0e] 0000:7c0e (unk. ctxt): mov bp, 0x7cd4 ; bdd47c
<bochs:8> r
rax: 00000000_00007c00
rbx: 00000000_00000000
rcx: 00000000_00090000
rdx: 00000000_00000045
rsp: 00000000_0000ffd6
rbp: 00000000_00000000
rsi: 00000000_000e0000
rdi: 00000000_0000070c
r8 : 00000000_00000000
r9 : 00000000_00000000
r10: 00000000_00000000
r11: 00000000_00000000
```

C. 另一方面，两者的段寄存器在 call 前后均无变化：

call 前：

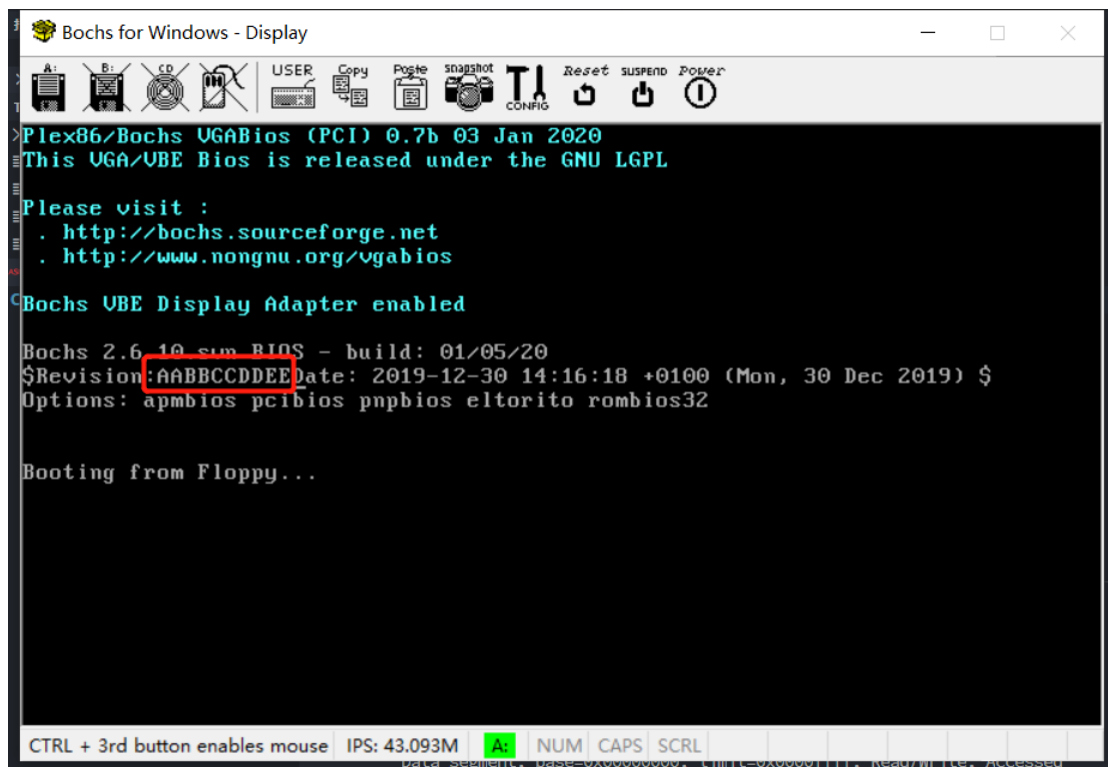
```
(0) [0x000000007c08] 0000:7c08 (unk. ctxt): call .+24 (0x00007c26) ; 66e818000000
<bochs:8> sreg
es:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
cs:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ss:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ds:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
    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
```

call 后：

```
(0) [0x000000007c26] 0000:7c26 (unk. ctxt): push ebp ; 6655
<bochs:11> 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=1
    Data segment, base=0x00000000, limit=0x0000ffff, Read/Write, Accessed
ds:0x0000, dh=0x00009300, dl=0x0000ffff, valid=1
    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
```

D. 猜想：bochs 的 ret 解释为 4bytes，而 vbox 为 2bytes，相对应 bochs 的 call 也要 4bytes，或者 vbox 的 call 解释为 4bytes，而 bochs 为 2bytes，相应 vbox 不需要修改 call 为 call dword。

结果:



- 5) 生成符号列表: 首先我想使用-S 参数生成 upper.s 进而通过 nasm -l 参数产生 lst 文件, 结果发现: 其中, 有部分语法是不熟悉的, 如图



经上网查询, 发现该语法属于 GNU ASM 语法:

<https://stackoverflow.com/questions/8406188/does-gcc-really-know-how-to-output-nasm-assembly>



属于 Intel 的语法，而 TASM 是基于 Intel 的语法的。

为了能够读懂代码，我找到一个工具 objconv-x64 可以将.o 文件转化成 NASM 语法的.asm 文件。参数如下：

```
objconv-x64 -fnasm upper.o upper.asm

Input file: upper.o, output file: upper.asm
Converting from COFF32 to Disassembly32

0 Debug sections removed
0 Exception sections removed
```

它最低支持 32 位 nasm，因而我需要使用 gcc 生成 32 位的目标文件，参数如下：

```
guzhengyang@guzhengyang-Vostro-14-5480:~/Documents/codes/实验3$ gcc -march=i386 -m32 -mpreferred-stack-boundary=2 -ffreestanding -c upper.c -o upper32.o -fno-pie
```

再使用该工具将其转换成 nasm。然后我得到了 nasm 语法的 upper32.asm 文件。

但是在生成 lst 文件过程中，我发现需要加-elf32 参数才能正确生成，另一方面，function 等等奇怪的语法需要注释掉。最终，正确生成 upper32.lst。

语法分析如下：

全局变量在此处声明，使用 db, dw, dd 的语法来创建：

```
1      8      global upper: ;function
2      9      global Message
```

局部变量放在栈内存中，如 dword [ebp-4H], 0:

```
1      14      upper: ; Function begin
2      15 00000000 55      push     ebp
; 0000 _ 55
3      16 00000001 89E5      mov      ebp, esp
; 0001 _ 89. E5
4      17 00000003 83EC04      sub      esp, 4
; 0003 _ 83. EC, 04
5      18 00000006 C745FC00000000      mov      dword [ebp-4H], 0
; 0006 _ C7. 45, FC, 00000000
```

为了研究函数调用的情况，我又写了 count32.c 并按照如上操作获得了 count32.asm，并观察函数调用情况：

参数传递：call 前 push 入栈中，call 后 sp 恢复到 push 前状态

```
1  push    dword [ebp+8H]
   ; 0032 _ FF. 75, 08
2  call    ccount
   ; 0035 _ E8, FFFFFFFC(rel)
3  add     esp, 4
   ; 003A _ 83. C4, 0
```

ebp、esp、eax：调用前后保持 ebp、esp 不变，其中 leave 就是用来恢复的语句，参数在栈 esp+地址长度+4\*n，返回值在 eax

```
1  push    ebp                                ; 002C _ 55
2  mov     ebp, esp
   ; 002D _ 89. E5
3  sub     esp, 16
   ; 002F _ 83. EC, 10
4  push    dword [ebp+8H]
   ; 0032 _ FF. 75, 08
5  call    ccount
   ; 0035 _ E8, FFFFFFFC(rel)
6  add     esp, 4
   ; 003A _ 83. C4, 04
7  cwde                                ; 003D _ 98
8  mov     dword [ebp-8H], eax
   ; 003E _ 89. 45, F8
9  mov     eax, dword [ebp-8H]
   ; 0041 _ 8B. 45, F8
10 inc     eax                                ; 0044 _ 40
11 mov     dword [ebp-4H], eax
   ; 0045 _ 89. 45, FC
12 mov     eax, dword [ebp-4H]
   ; 0048 _ 8B. 45, FC
13 leave                                ; 004B _ C9
14 ret                                     ; 004C _ C3
```

(2) 按照如上经验编写代码 count.c 和 str.asm，其中 str.asm 传递字符串给 count.c，count.c 返回字符串长度。过程中，发现 str.asm 若想打印多位的十进制数字，需要将数字转成字符串，

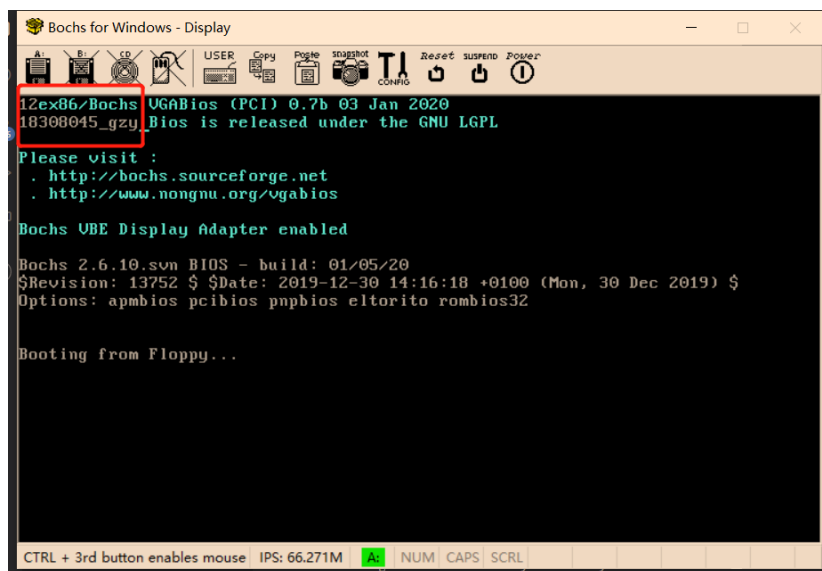
因为未发现 BIOS 中断中有可以打印数字的功能。另一方面，打印字符串的时候需要字符串长度，所以是 str.asm 先调用 `_str_len = count(_str)` 计算 `_str` 的长度 `_str_len`。再调用 `uint2str(_str_len, _str_len_str)`，将 `_str_len` 转成字符串 `_str_len_str`。再次调用 `eax = count(_str_len_str)` 计算 `_str_len_str` 长度，用于打印 `_str_len_str`。

结果 gcc 编译时出现了 `undefined reference to `__stack_chk_fail'` 的错误，经过上网查询：

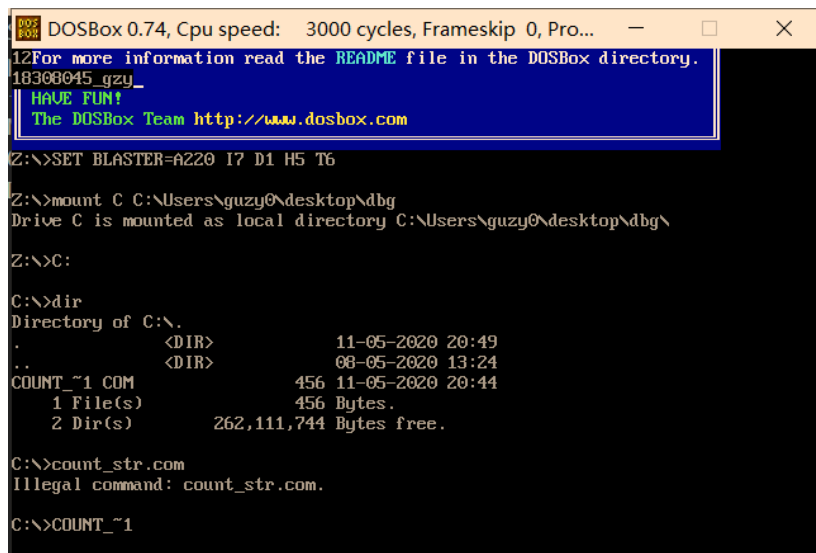
<https://blog.csdn.net/xiaominthere/article/details/18084865>

它说需要加参数 `-fno-stack-protector`。

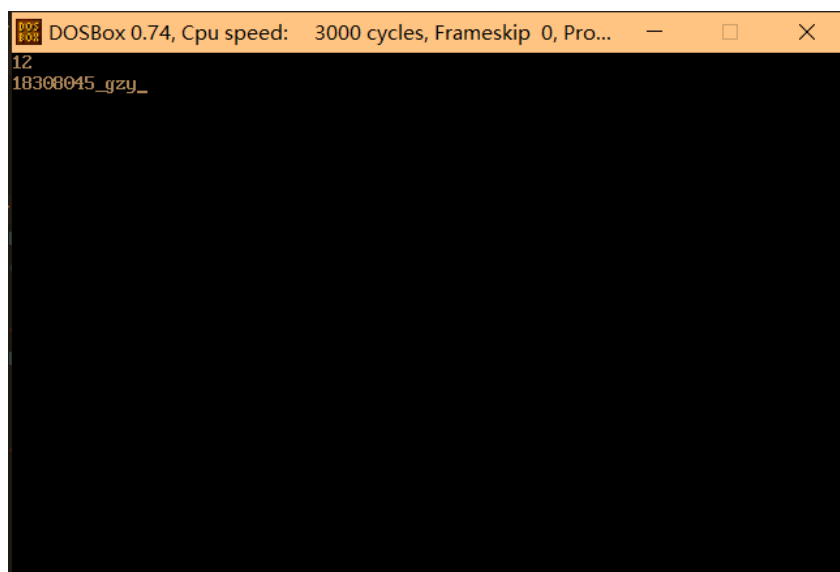
写入首扇区最终效果：



改成.com 程序，在 DOSBOX 中运行的效果：



清屏+显示+任意键(清屏+ret)效果:



另外, 在这里, 我调查了资料, 并观察了 leave 和 ret 的实际操作:

leave 前:

```
(0) [0x000000007caf] 0000:7caf (unk. ctxt): leave ; 66c9
<bochs:23> r
rax: 00000000_0000000c
rbx: 00000000_00000000
rcx: 00000000_00090000
rdx: 00000000_0000000c
rsp: 00000000_0000ffb8
rbp: 00000000_0000ffc8
rsi: 00000000_000e0000
rdi: 00000000_0000070c
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_00007caf
eflags 0x00000046: id vip vif ac vm rf nt IOPL=0 of df if tf sf ZF af PF cf
```

leave 后：恢复 esp，对应开始的 mov ebp, esp:

leave = mov esp, ebp

+ pop dword ebp

```
Next at t=2094437
(0) [0x000000007cb1] 0000:7cb1 (unk. ctxt): ret ; 66c3
<bochs:25> r
rax: 00000000_0000000c
rbx: 00000000_00000000
rcx: 00000000_00090000
rdx: 00000000_0000000c
rsp: 00000000_0000ffce
rbp: 00000000_00000000
rsi: 00000000_000e0000
rdi: 00000000_0000070c
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_00007cb1
eflags 0x00000046: id vip vif ac vm rf nt IOPL=0 of df if tf sf ZF af PF cf
<bochs:26> x /1hx sp
[bochs]:
0x000000000000ffce <bogus+ 0>: 0x7c14
```

另外如上，此时栈顶的内容为 call 后下一条指令的地址，ret 将其弹出，且弹了 32 位地址，

即 4bytes:

```
(0) [0x000000007c14] 0000:7c14 (unk. ctxt): add sp, 0x0004 ; 83c404
<bochs:28> r
rax: 00000000_0000000c
rbx: 00000000_00000000
rcx: 00000000_00090000
rdx: 00000000_0000000c
rsp: 00000000_0000ffd2
rbp: 00000000_00000000
rsi: 00000000_000e0000
rdi: 00000000_0000070c
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_00007c14
eflags 0x00000046: id vip vif ac vm rf nt IOPL=0 of df if tf sf ZF af PF cf
```

(3) 根据以上的试错，我总结出如下几点：

- A. 函数调用前后，esp，ebp 不变
- B. 进入函数栈顶首元素是 call 后下一条指令地址，栈顶下面的元素是参数
- C. 返回值在 eax
- D. 函数名 Windows 下加\_，Linux 下不加\_

根据这四条原则，我将 BIOS 中断的一些功能封装成函数：

光标位置打印字符：                    void \_put(char ch);

移动光标：                              void \_move(int x, int y);

有阻塞的读取字符和状态码：          unsigned short \_get();

清屏：                                  void \_cls();

加载指定扇区的用户程序并调用：      void \_callf(int cl, int ch, int dh, int len);

然后利用这些函数，在 c 中重写监控程序。并且根据上次实验的猜想：“由于频繁插入删除，使用链式存储 buffer 比连续存储 buffer 的效率高”，将 buffer 改成链式存储。

调用用户程序，发现问题：

1). c 语言内函数调用出错

观察调用函数后使用参数的情况：

```
(0) [0x000000007c90] 0000:7c90 (unk. ctxt): mov eax, dword ptr ss:[ebp+8] ; 67668b4508
<bochs:44> x /1hx ebp+8
[bochs]:
0x000000000000ffc4 <bogus+      0>:    0x0000
<bochs:45> x /1hx ebp+6
[bochs]:
0x000000000000ffc2 <bogus+      0>:    0x7d1c
<bochs:46> n
Next at t=2152740
(0) [0x000000007c95] 0000:7c95 (unk. ctxt): add eax, edx ; 6601d0
<bochs:47> x /1hx ebp+6
[bochs]:
0x000000000000ffc2 <bogus+      0>:    0x7d1c
<bochs:48> x /1hx ebp+8
[bochs]:
0x000000000000ffc4 <bogus+      0>:    0x0000
<bochs:49> x /1hx ebp+6
[bochs]:
0x000000000000ffc2 <bogus+      0>:    0x7d1c
```

发现参数的实际位置在 esp+2 (ebp+6 因为一开始 push dword ebp 且 mov ebp esp 了) 而函数却认为参数在 esp+4 (对应 ebp+8 理由同上), 根据总结 B, 就是 call 时使用了 16 位地址 2bytes, 而函数却认为它是 32 位地址 4bytes。精准地总结出问题, 再经过同学的点播, 我找到了解决方法: .16codegcc, 可以让 c 函数调用时传入 32 位地址, 和使用 16 位的指令 (符合 gcc)。

<https://www.cnblogs.com/youxia/p/linux008.html>

## 2). 加载用户程序时出错。

因为我的代码体量比较庞大, 所以我把用户程序存在了 32, 33, 34, 35 扇区。然而加载失败, 查看状态码 ah=0x20, 发现是控制器错误。

```
(0) [0x000000007c1a] 0000:7c1a (unk. ctxt): int 0x13 ; cd13
<bochs:15> n
Next at t=2095342
(0) [0x000000007c1c] 0000:7c1c (unk. ctxt): callf 0x0800:00000100 ; 669a000100000008
<bochs:16> r
rax: 00000000_60002000
rbx: 00000000_00000100
rcx: 00000000_00090021
rdx: 00000000_00000000
rsp: 00000000_0000ffd6
rbp: 00000000_00000000
rsi: 00000000_000e0000
rdi: 00000000_0000070c
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_00007c1c
eflags 0x00000083: id vip vif ac vm rf nt IOPL=0 of df if tf SF zf af pf CF
```

苦苦思索的过程中, 无意间翻到一篇帖子, 一个人说是加载第 19 扇区时出现和我同样的问题。终于意识到物理扇区和逻辑扇区的对应关系不是简单物理扇区=逻辑扇区+1 那么简单, 还有磁头号和磁道号。而这正是老师 FAT 实验课第一节课讲的内容!!!

编写函数转换逻辑扇区为物理扇区, 终于解决了问题。

3). 运行成功，但是运行序列中第一个用户程序后无法运行下一个。

发现是第一个用户程序 ret 时出现问题，最后发现把 call dword 段:偏移改成 call 段:偏移就行了，在这里 dword 反而多此一举。结合知识实模式下段地址 16 位，偏移地址 4 位，可能编译的时候将它自动处理成了 32 位压进栈，而 dword 就可能变成了压了 64 位进栈，对此我没有深究，但是通过观察，确实 call 段:偏移后 esp-=4。

(4) 因为是将 IO 等本来需要库才能实现的操作，用封装 BIOS 中断来实现，所以一方面在 C 中写代码可读性更强，思路也更加清晰；另一方面可以将旧的 c 语言代码迁移到本次实验中，因而扩展十分容易。

我扩展了如下功能：

1. 用户程序的名字不再是单个字符，而可以是一个字符串
2. 将展示用户程序信息的功能单独剥离出来，成为一条命令 list
3. 将调用用户程序的命令单独剥离出来，称为 call
4. 将过去写过的浮点数计算器迁移成用户程序，计算器支持“+\*/().”这七个非符号

(5) 重写引导程序，加载 COM 格式程序的独立内核。

(6) 将代码上传到 git，把报告保存到网盘。



## 六、实验感想

再写一遍本次实验试错得到的一些经验，方便以后查看：

### 1. 函数调用：

A. 函数调用前后，esp，ebp 不变

- 1) 使用 call dword 或者 call 段:偏移
- 2) 使用 ret
- 3) 保证函数内部前后 esp，ebp 不变
- 4) 若函数外参数压栈，调用后需要恢复 esp

B. 进入函数栈顶首元素是 call 后下一条指令地址，栈顶下面的元素是参数

- 1) 32 位地址来说，esp 是地址，esp+4\*n 是第 n 个参数

C. 返回值在 eax

- 1) 注意小端存储，short 接收的是 al，ah，因而转成结构体也需要按照 al，ah 来

D. 函数名 Windows 下加\_，Linux 下不加\_

### 2. 联合编译（使用 Linux）：

```
gcc -march=i386 -m16 -mpreferred-stack-boundary=2 -ffreestanding -c .c -o .o -fno-pie -fno-stack-protector
```

```
nasm -f elf32.asm -o.o
```

```
ld -m elf_i386 -N str.o .o -Ttext 0x100 --oformat binary -o .com
```

```
ld -m elf_i386 -N str.o .o -Ttext 0x7C00 --oformat binary -o .bin
```

## 附录（流程图，注释过的代码）：

代码见附件