**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 | [**Guzy0324@163.com**](mailto:Guzy0324@163.com) |
| 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

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

1. **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

1. **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)写一个汇编程和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。利用汇编程和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。

(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：

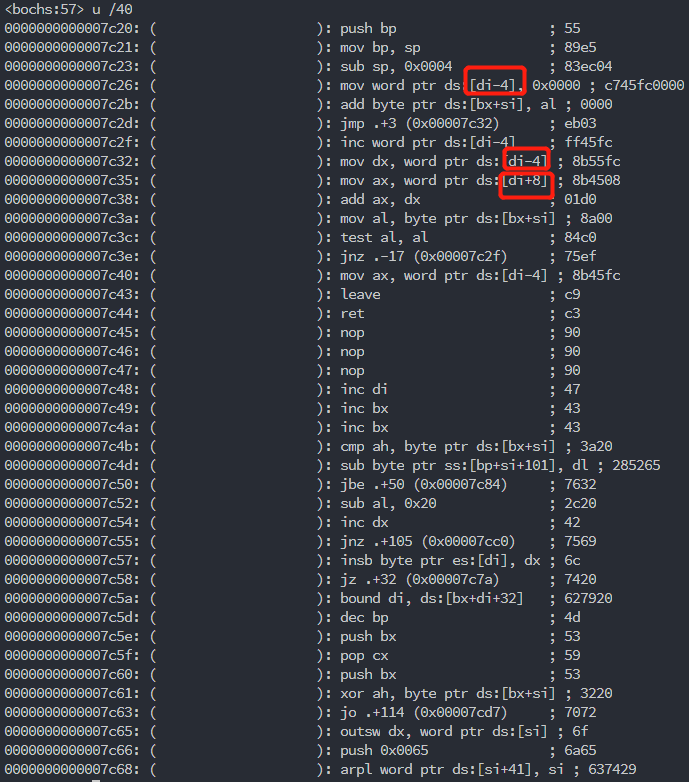
<https://blog.csdn.net/jltxgcy/article/details/8687881>

**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。

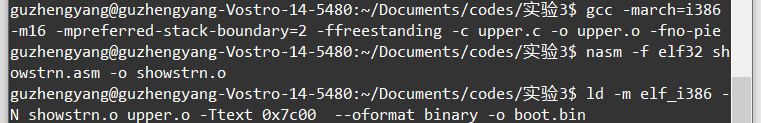
2）成功运行，并且显示字符串，但是结果却是小写并未改成大写。而后我又编写了用来计算字符串长度的count.c文件，发现了同样的问题，c语言的函数不好用。通过bochs查看代码，发现在c的函数中，它使用了从未初始化且毫无道理的di寄存器和si寄存器等等，如图：



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，然而实际上是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：



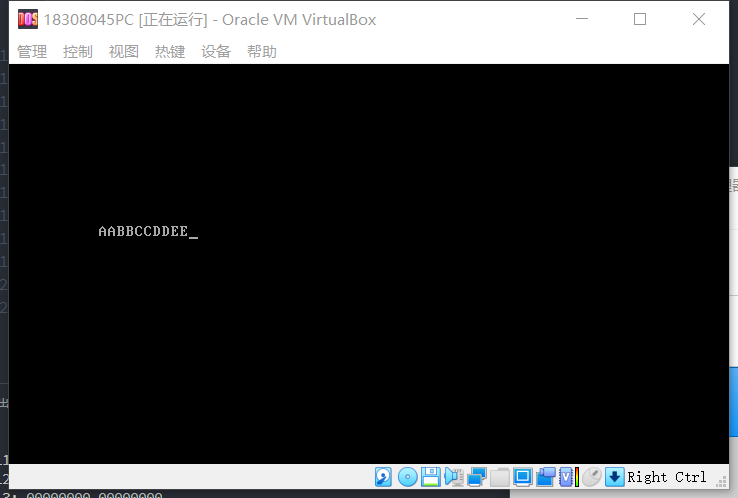
其中-fno-pie是经过群友的点播，为了防止报错：

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

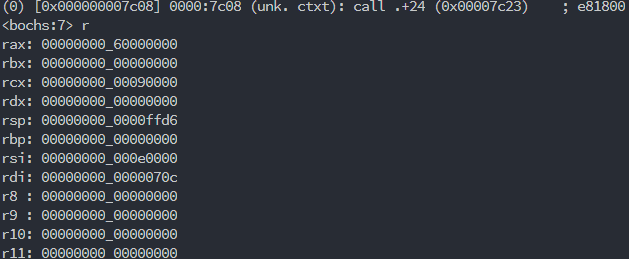
4） 发现在bochs中仍然不能运行，在c函数ret后跳到了很诡异的地方，然而在VirtualBox中却能够正确运行。从群中群友的只言片语中，我获得了一个信息：可能要使用call dword而不是call，结果真的在bochs中运行成功了。观察两者区别如下：

A. 使用call：

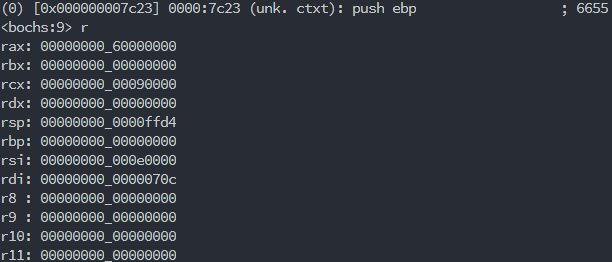
VirtualBox：



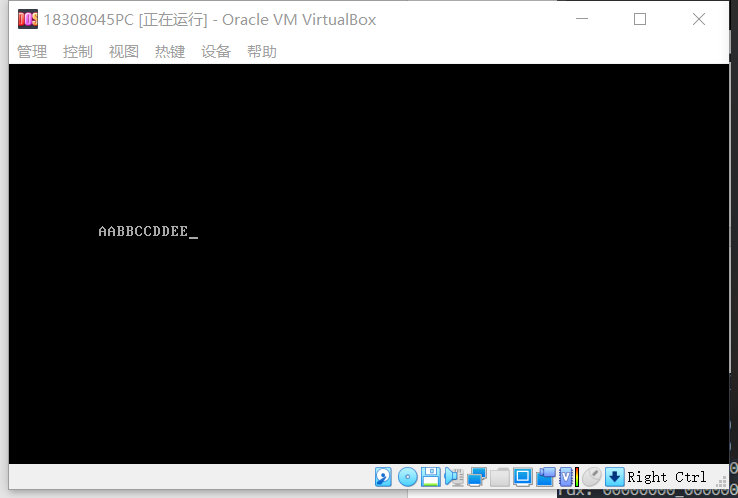
Bochs中call前寄存器状态：注意观察sp寄存器



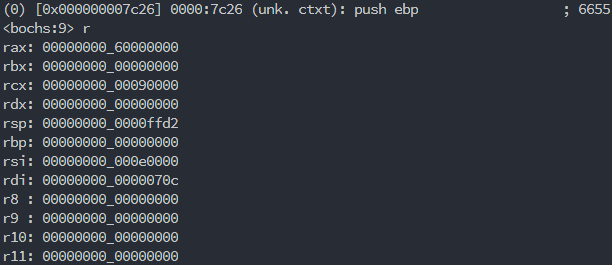
call后：sp -= 2



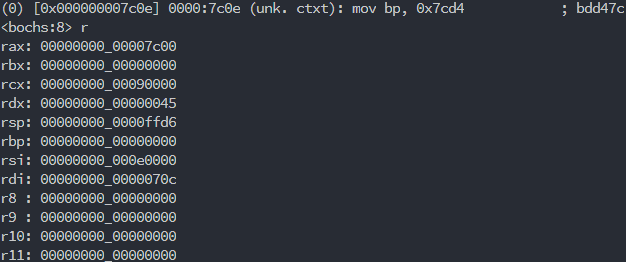
B. 使用call dword：



call dword后：sp -= 4

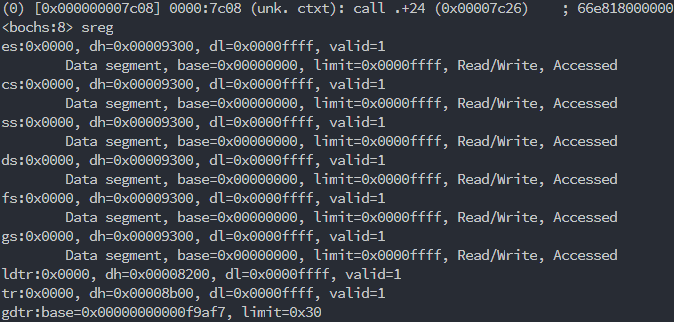


ret后：sp += 4

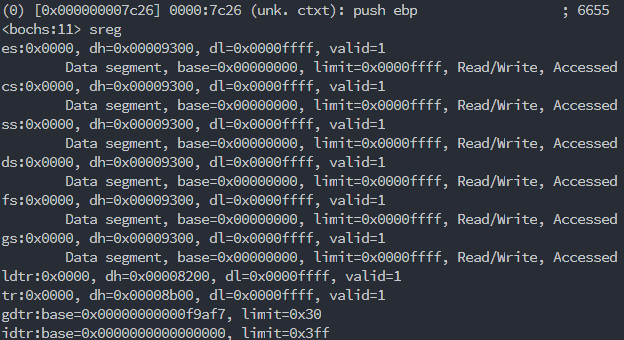


C. 另一方面，两者的段寄存器在call前后均无变化：

call前：

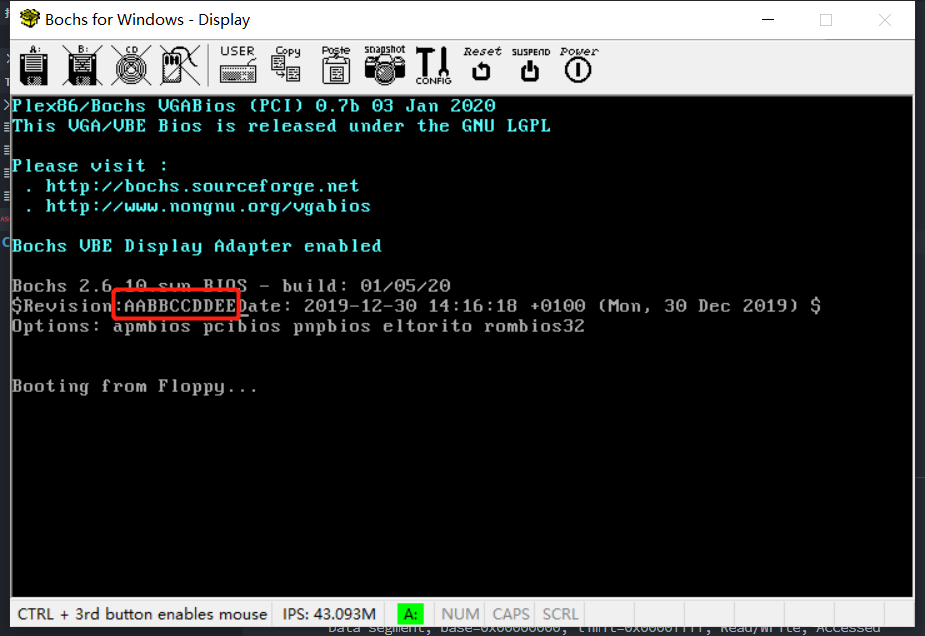


call后：

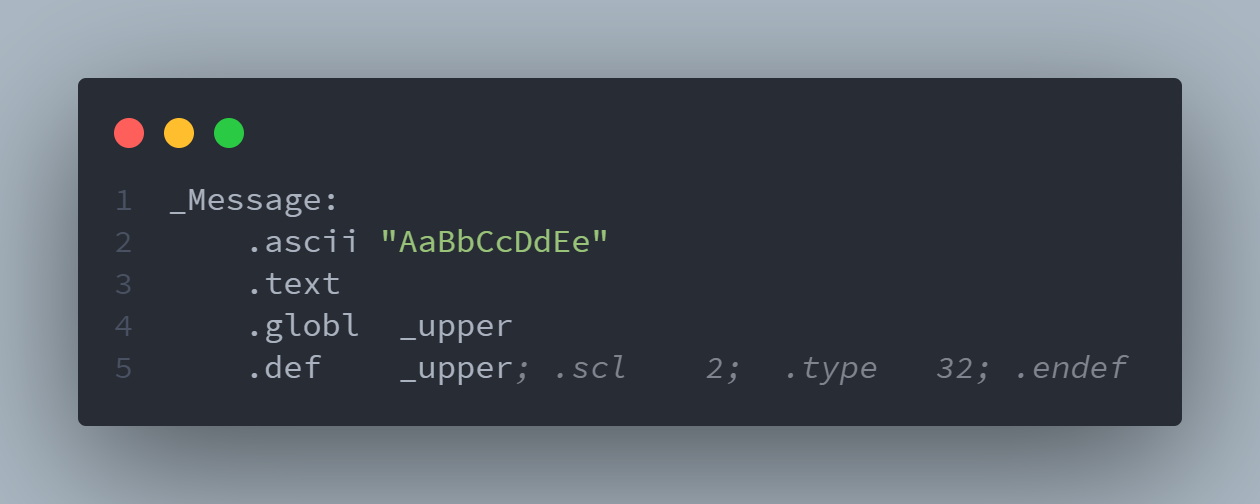


D. 猜想：bochs的ret解释为4bytes，而vbox为2bytes，相对应bochs的call也要4bytes，或者vbox的call解释为4bytes，而bochs为2bytes，相应vbox不需要修改call为call dword。

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

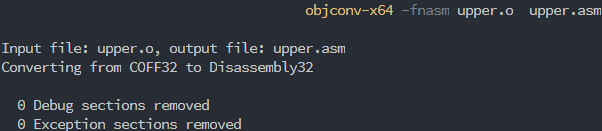


经上网查询，发现该语法属于GNU ASM语法：

<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：



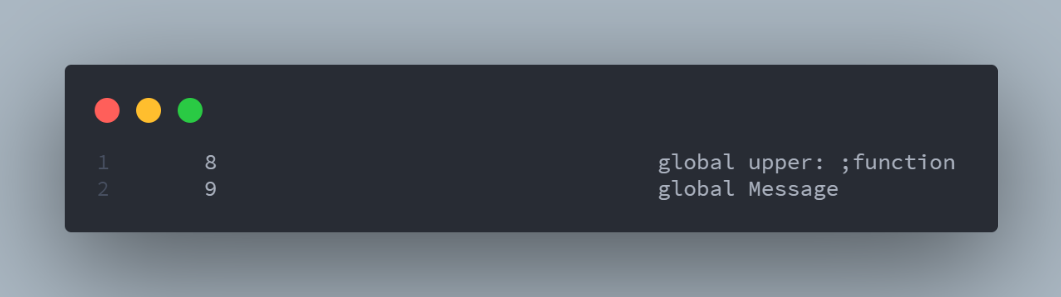
它最低支持32位nasm，因而我需要使用gcc生成32位的目标文件，参数如下：

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。

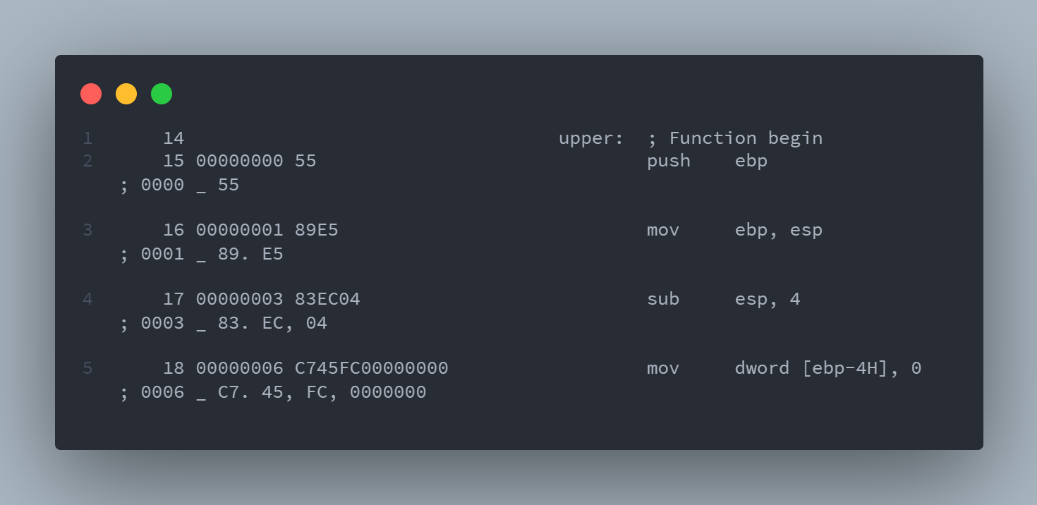
但是在生成lst文件过程中，我发现需要加-elf32参数才能正确生成，另一方面，function等等奇怪的语法需要注释掉。最终，正确生成upper32.lst。

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：

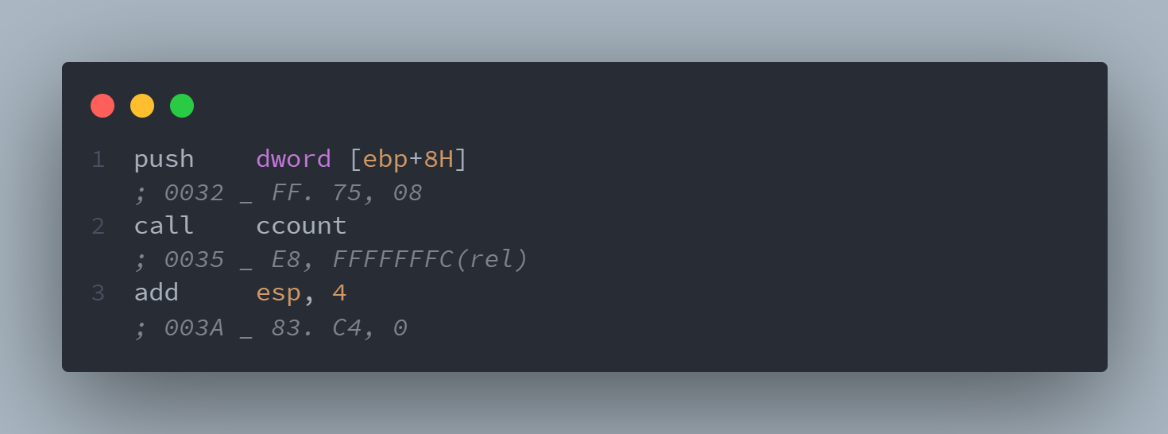


局部变量放在栈内存中，如dword [ebp-4H], 0：



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



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



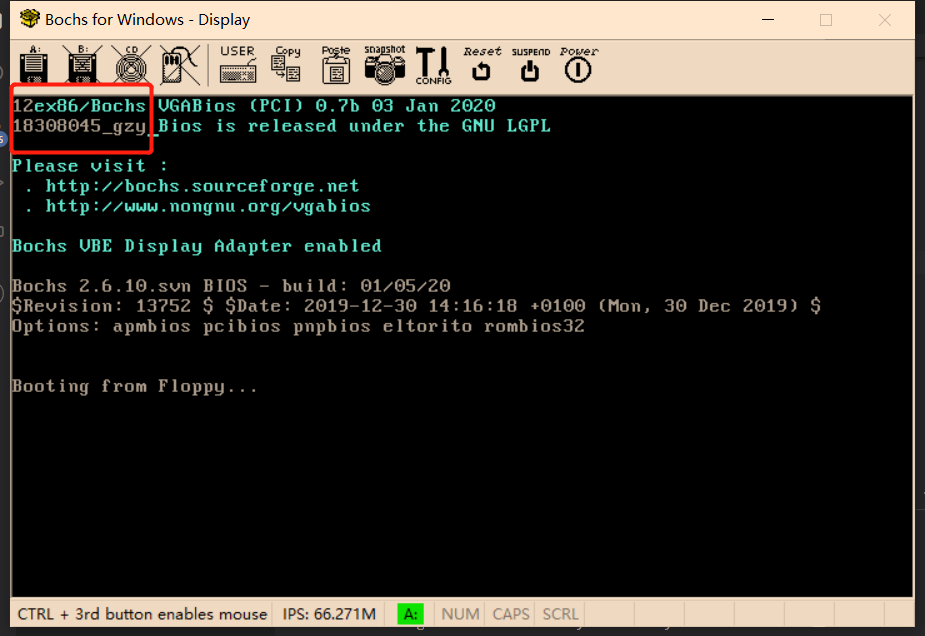
(2) 按照如上经验编写代码count.c和str.asm，其中str.asm传递字符串给count.c，count.c返回字符串长度。过程中，发现str.asm若想打印多位的十进制数字，需要将数字转成字符串，因为未发现BIOS中断中有可以打印数字的功能。另一方面，打印字符串的时候需要字符串长度，所以是str.asm先调用\_str\_len = count(\_str)计算\_str的长度\_str\_len。再调用uint2str(\_str\_len, \_str\_len\_str)，将\_str\_len转成字符串\_str\_len\_str。再次调用eax = count(\_str\_len\_str)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：

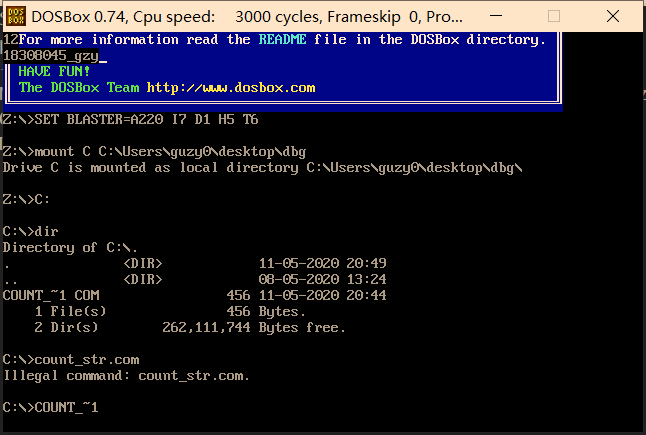
<https://blog.csdn.net/xiaominthere/article/details/18084865>

它说需要加参数-fno-stack-protector。

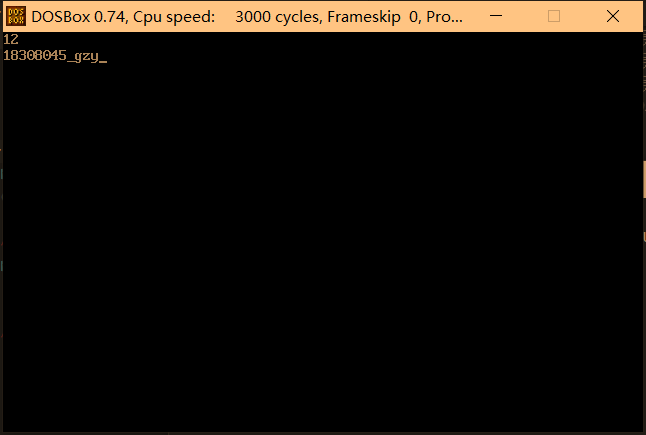
写入首扇区最终效果：



改成.com程序，在DOSBOX中运行的效果：

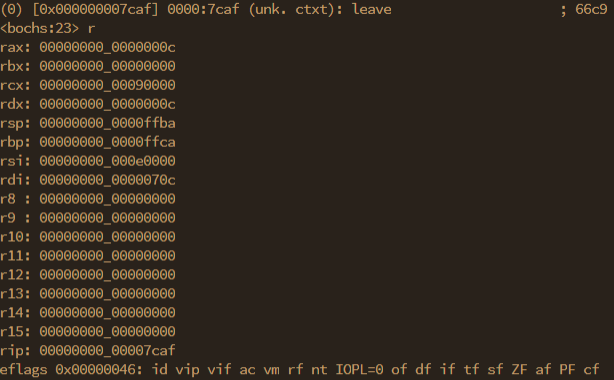


清屏+显示+任意键(清屏+ret)效果：



另外，在这里，我调查了资料，并观察了leave和ret的实际操作：

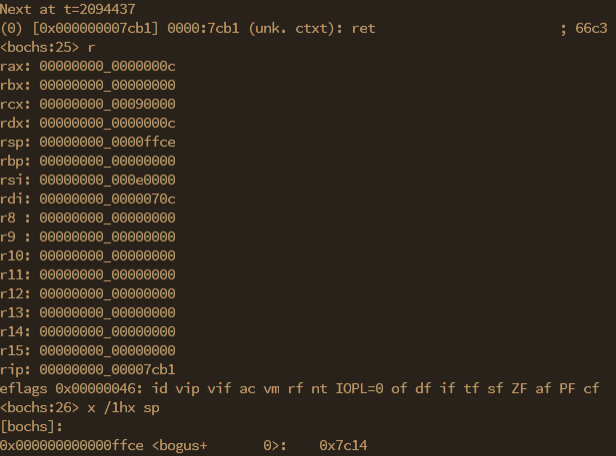
leave前：



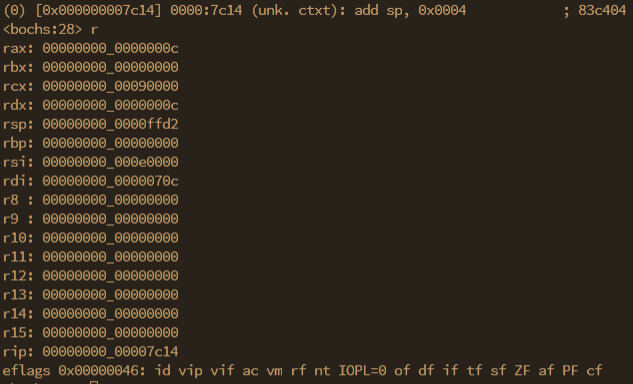
leave后：恢复esp，对应开始的mov ebp, esp：

leave = mov esp，ebp

+ pop dword ebp



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：



(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();

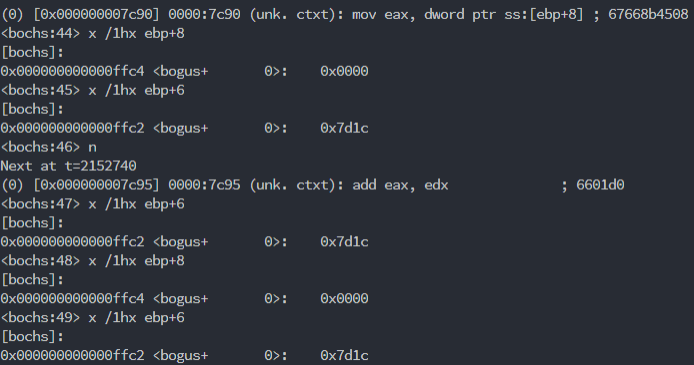
加载指定扇区的用户程序并调用： void \_callf(int cl, int ch, int dh, int len);

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：

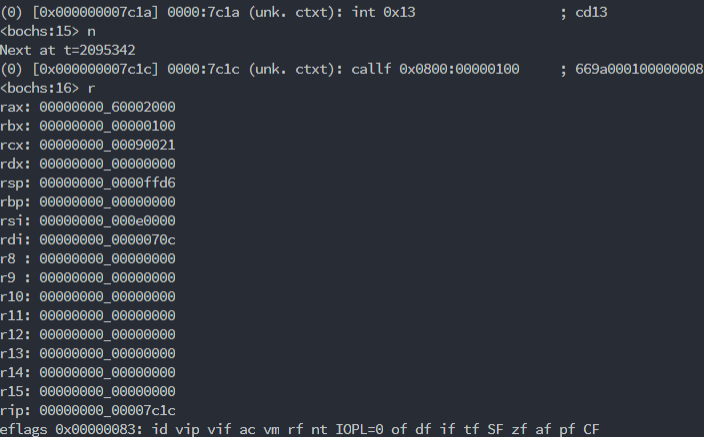


发现参数的实际位置在esp+2（ebp+6因为一开始push dword ebp且mov ebp esp了）而函数却认为参数在esp+4（对应ebp+8理由同上），根据总结B，就是call时使用了16位地址2bytes，而函数却认为它是32位地址4bytes。精准地总结出问题，再经过同学的点播，我找到了解决方法：.16codegcc，可以让c函数调用时传入32位地址，和使用16位的指令（符合gcc）。

<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。



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 a.o c.o -Ttext 0x100 --oformat binary -o .com

ld -m elf\_i386 -N a.o c.o -Ttext 0x7C00 --oformat binary -o .bin

**附录（流程图，注释过的代码）：**

代码见附件