

操作系统原理实验报告

**实验名称:** 实验三从实模式到保护模式

**授课教师：** 张青

**学生姓名:** 曾慧蕾

**学生学号:** 21307358

1. **实验要求**

**Assignment 1**

**1.1**

复现Example 1，说说你是怎么做的并提供结果截图，也可以参考Ucore、Xv6等系统源码，实现自己的LBA方式的磁盘访问。

**1.2**

在Example1中，我们使用了LBA28的方式来读取硬盘。此时，我们只要给出逻辑扇区号即可，但需要手动去读取I/O端口。然而，BIOS提供了实模式下读取硬盘的中断，其不需要关心具体的I/O端口，只需要给出逻辑扇区号对应的磁头（Heads）、扇区（Sectors）和柱面（Cylinder）即可，又被称为CHS模式。现在，同学们需要将LBA28读取硬盘的方式换成CHS读取，同时给出逻辑扇区号向CHS的转换公式。最后说说你是怎么做的并提供结果截图。



参考资料见附录

**Assignment 2**

复现Example 2，使用gdb或其他debug工具在进入保护模式的4个重要步骤上设置断点，并结合代码、寄存器的内容等来分析这4个步骤，最后附上结果截图。gdb的使用可以参考appendix的“debug with gdb and qemu”部份。

**Assignment 3**

**assignment 3的寄存器请使用32位的寄存器。**

改造“Lab2-Assignment 4”为32位代码，即保护模式后执行自定义的汇编程序

1. **实验过程**

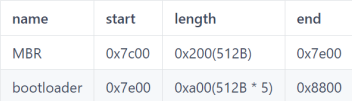
**1.Bootloader的加载：**

**1.1.1原理：**

1. 为了突破512字节的限制，我们通过一段程序来完成MBR无法完成的工作，例如从实模式跳转到保护模式、加载操作系统内核等。由于这段程序是为了加载操作系统内核而服务的，因此被称为bootloader.
2. 硬盘是外围设备的一种，处理器和外围设备的交换是通过I/O端口进行的。实际上，I/O端口是一些寄存器，位于I/O接口电路中。每一个端口在I/O电路中都会被统一编址。例如，主硬盘分配的端口地址是0x1f0~0x1f7，从硬盘分配的端口地址是 0x170~0x177.
3. 我们通过LBA来读取硬盘。首先我们设置LBA28作为起始的逻辑扇区号，然后将要读取的扇区数量写入0x1F2端口。由于这是一个8位端口，因此每次最多只能读写255个扇区。随后向0x1F7端口写入0x20，请求硬盘读。继而我们要等待其他读写操作完成。若在上一步中检测到其他操作已经完成，那么我们就可以正式从硬盘中读取数据。我们从0x1F0中连续读入一个扇区的数据。

**1.1.2**

在第一个例子中，我们将lab2中输出Hello World部份的代码放入到bootloader中，然后在MBR中加载bootloader到内存，并跳 转到bootloader的起始地址执行。目前，我们的内存地址安排如下，bootloader安排在MBR之后，预留5个扇区的空间。



我们先新建一个文件bootloader.asm，里面写我们要输出的内容。代码如下

*org 0x7e00*

*[bits 16]*

*mov ax, 0xb800*

*mov gs, ax*

*mov ah, 0x03 ;青色*

*mov ecx, bootloader\_tag\_end - bootloader\_tag*

*xor ebx, ebx*

*mov esi, bootloader\_tag*

*output\_bootloader\_tag:*

*mov al, [esi]*

*mov word[gs:bx], ax*

*inc esi*

*add ebx,2*

*loop output\_bootloader\_tag*

*jmp $ ; 死循环*

*bootloader\_tag db 'run bootloader HELLO WORLD 21307358'*

*bootloader\_tag\_end:*

然后我们在mbr.asm处放入使用LBA模式读取硬盘的代码，然后在MBR中加载bootloader到地址0x7e00。代码如下

*org 0x7c00*

*[bits 16]*

*xor ax, ax ; eax = 0*

*; 初始化段寄存器, 段地址全部设为0*

*mov ds, ax*

*mov ss, ax*

*mov es, ax*

*mov fs, ax*

*mov gs, ax*

*; 初始化栈指针*

*mov sp, 0x7c00*

*mov ax, 1 ; 逻辑扇区号第0~15位*

*mov cx, 0 ; 逻辑扇区号第16~31位*

*mov bx, 0x7e00 ; bootloader的加载地址*

*load\_bootloader:*

*call asm\_read\_hard\_disk ; 读取硬盘*

*inc ax*

*cmp ax, 5*

*jle load\_bootloader*

*jmp 0x0000:0x7e00 ; 跳转到bootloader*

*jmp $ ; 死循环*

*asm\_read\_hard\_disk:*

*; 从硬盘读取一个逻辑扇区*

*; 参数列表*

*; ax=逻辑扇区号0~15位*

*; cx=逻辑扇区号16~28位*

*; ds:bx=读取出的数据放入地址*

*; 返回值*

*; bx=bx+512*

*mov dx, 0x1f3*

*out dx, al ; LBA地址7~0*

*inc dx ; 0x1f4*

*mov al, ah*

*out dx, al ; LBA地址15~8*

*mov ax, cx*

*inc dx ; 0x1f5*

*out dx, al ; LBA地址23~16*

*inc dx ; 0x1f6*

*mov al, ah*

*and al, 0x0f*

*or al, 0xe0 ; LBA地址27~24*

*out dx, al*

*mov dx, 0x1f2*

*mov al, 1*

*out dx, al ; 读取1个扇区*

*mov dx, 0x1f7 ; 0x1f7*

*mov al, 0x20 ;读命令*

*out dx,al*

*; 等待处理其他操作*

*.waits:*

*in al, dx ; dx = 0x1f7*

*and al,0x88*

*cmp al,0x08*

*jnz .waits*

*; 读取512字节到地址ds:bx*

*mov cx, 256 ; 每次读取一个字，2个字节，因此读*

*取256次即可*

*mov dx, 0x1f0*

*.readw:*

*in ax, dx*

*mov [bx], ax*

*add bx, 2*

*loop .readw*

*ret*

*times 510 - ($ - $$) db 0*

*db 0x55, 0xaa*

然后我们编译bootloader.asm，写入硬盘起始编号为1的扇区，共有5个扇区。

nasm -f bin bootloader.asm -o bootloader.bin

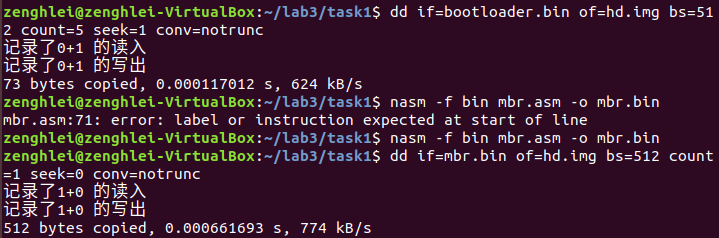
dd if=bootloader.bin of=hd.img bs=512 count=5 seek=1 conv=notrunc

mbr.asm也要重新编译和写入硬盘起始编号为0的扇区。

nasm -f bin mbr.asm -o mbr.bin

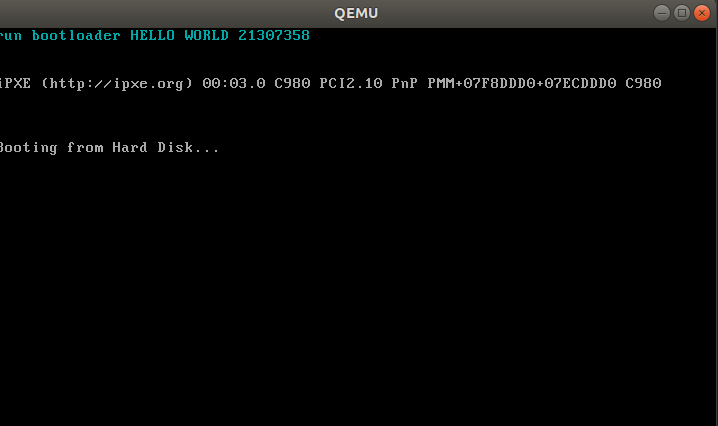
dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc

编译和写入：



接下来使用qemu运行即可：qemu-system-i386 -hda hd.img -serial null -parallel stdio

结果：



成功输出

**1.2.1 原理**

BIOS提供了实模式下读取硬盘的中断，其不需要关心具体的I/O端口，只需要给出逻辑扇区号对应的磁头（Heads）、扇区（Sectors）和柱面（Cylinder）即可。在这个任务中，采用INT 13h中断 02H来直接读取扇区



**转换公式：**



1.2.2

根据以上内容，改变mbr.asm中的load\_bootloader函数和asm\_read\_hard\_disk函数如下所示：

*load\_bootloader:*

*call asm\_read\_hard\_disk ; 读取硬盘*

*jmp 0x0000:0x7e00 ; 跳转到bootloader*

这里直接跳转就可

*asm\_read\_hard\_disk:*

*mov ah, 02h ;功能参数*

*mov al, 05h ;扇区数*

*mov ch, 00h ;柱面*

*mov cl, 2 ;扇区*

*mov dh, 00h ;磁头*

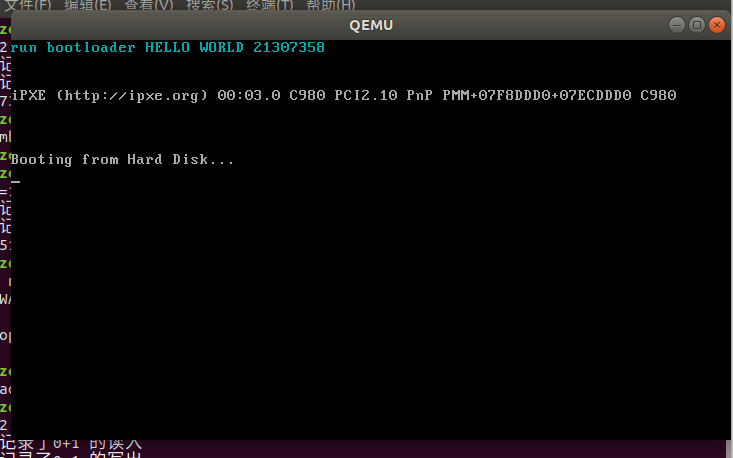
*mov dl, 80h ;hd*

*int 0x13*

这里写入各种参数后，写好中断号即可

在终端内重新编译、写入的操作如1.1.2，此处不赘述。

最终得到的输出：



成功输出

**2.进入保护模式：**

2.1 原理

1. 进入保护模式的步骤。

**(1) 准备GDT，用lgdt指令加载GDTR信息。**

**(2) 打开第21根地址线。**

**(3) 开启cr0的保护模式标志位。**

**(4) 远跳转，进入保护模式。**

1. 在前面，我们已经实现了bootloader的加载，我们需要在bootloader中跳转到保护模式。首先，我们需要定义段描述符，我们有代码段描述符、数据段描述符、栈段描述符和视频段描述符。

由于保护模式下的寄存器是32位寄存器，保护模式的地址线也是32位。因此，我们单纯使用偏移地址也可以访问保护模式的4GB空间。我们并不需要将程序分为一个个的段。为了简化地址的访问，我们让所有的程序运行在同一个段中，这个段的地址空间大小是4GB，也就是全部的地址空间。此时，我们让代码段描述符、数据段描述符和栈段描述符中的段线性基地址为0，那么偏移地址和线性地址就完全相同，这大大简化了编程的逻辑。

视频段描述符是显存所在的内存区域的段描述符。注意，GDT的第0个描述符必须是全0的描述符。接着，我们在GDT中依次放入0描述符，数据段描述符、堆栈段描述符、显存段描述符和代码段描述符，代码见补充。

为了让CPU知道GDT的位置，我们会事先设置好GDTR寄存器。我们在内存中使用一个48位的变量来表示GDTR的内容，然后把GDT的信息写入变量pgdt，把pgdt的内容加载进GDTR中。然后根据段描述符的内容设置好段选择子（boot.inc）

1. 代码段描述符详解：

对于代码段描述符，描述符高32位为0x00cf9800，低32位为0x0000ffff，因此各个部分含义如下。

• 段线性基地址。段线性基地址由三部分组成，但都是0，因此基地址为0。

• G=1，表示段界限以4KB为单位。

• D/B=1，表示操作数大小为32位。

• L=0，表示32位代码。

• AVL，保留位，不关心，置0即可。

• 段界限由两个部分组成，值为0xfffff，共20位。结合粒度和段界限，整个代码段的长度计算如下。

**长度 = (段界限+1)∗粒度 = (0xfffff+1)∗4KB = 220∗212B = 232B = 4GB**

• 因此整个代码段表示的范围是0x00000000~0xffffffff。由于基地址为0，偏移地址直接表示线性地址，因此也被称为平坦

模式。

• P=1，表示段存在。

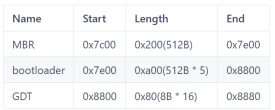
• DPL=0，表示最高优先级。

• S=1，表示代码段。

• TYPE=0x8，表示只执行，非一致代码段。

2.2

在进入保护模式之前，我们先对我们的内存地址进行规划。



MBR被自动加载到0x7c00，长度512字节，因此结束于0x7e00。我们的bootloader放置在MBR之后，长度限制在5个扇区，因 此结束于0x8800。虽然GDT中可放入8192个描述符，但我们并不打算定义如此多的段，实际上我们使用的段不会超过16个。我 们不妨将上述常量定义在一个独立的文件boot.inc中，如下所示。

*; 常量定义区*

*; \_\_\_\_\_\_\_\_\_\_\_\_\_Loader\_\_\_\_\_\_\_\_\_\_\_\_\_*

*; 加载器扇区数*

*LOADER\_SECTOR\_COUNT equ 5*

*; 加载器起始扇区*

*LOADER\_START\_SECTOR equ 1*

*; 加载器被加载地址*

*LOADER\_START\_ADDRESS equ 0x7e00*

*; \_\_\_\_\_\_\_\_\_\_\_\_\_GDT\_\_\_\_\_\_\_\_\_\_\_\_\_*

*; GDT起始位置*

*GDT\_START\_ADDRESS equ 0x8800*

接着，我们在bootloader.asm里面写如下内容，放入0描述符，数据段描述符、堆栈段描述符、显存段描述符和代码段描述符。

*%include "boot.inc"*

*org 0x7e00*

*[bits 16]*

*... ; 输出bootloader\_tag代码，此处省略*

*;空描述符*

*mov dword [GDT\_START\_ADDRESS+0x00],0x00*

*mov dword [GDT\_START\_ADDRESS+0x04],0x00*

*;创建描述符，这是一个数据段，对应0~4GB的线性地址空间*

*mov dword [GDT\_START\_ADDRESS+0x08],0x0000ffff ; 基地址为0，段界限为0xFFFFF*

*mov dword [GDT\_START\_ADDRESS+0x0c],0x00cf9200 ; 粒度为4KB，存储器段描述符*

*;建立保护模式下的堆栈段描述符*

*mov dword [GDT\_START\_ADDRESS+0x10],0x00000000 ; 基地址为0x00000000，界限0x0*

*mov dword [GDT\_START\_ADDRESS+0x14],0x00409600 ; 粒度为1个字节*

*;建立保护模式下的显存描述符*

*mov dword [GDT\_START\_ADDRESS+0x18],0x80007fff ; 基地址为0x000B8000，界限0x07FFF*

*mov dword [GDT\_START\_ADDRESS+0x1c],0x0040920b ; 粒度为字节*

*;创建保护模式下平坦模式代码段描述符*

*mov dword [GDT\_START\_ADDRESS+0x20],0x0000ffff ; 基地址为0，段界限为0xFFFFF*

*mov dword [GDT\_START\_ADDRESS+0x24],0x00cf9800 ; 粒度为4kb，代码段描述符*

*接着继续补充好gdtr的内容：*

*pgdt dw 0*

*dd GDT\_START\_ADDRESS*

然后把GDT的信息写入变量pgdt，把pgdt的内容加载进GDTR。

*;初始化描述符表寄存器GDTR*

*mov word [pgdt], 39 ;描述符表的界限*

*lgdt [pgdt]*

然后根据段描述符的内容设置段选择子。（boot.inc）

*; \_\_\_\_\_\_\_\_\_\_\_\_\_Selector\_\_\_\_\_\_\_\_\_\_\_\_\_*

*;平坦模式数据段选择子*

*DATA\_SELECTOR equ 0x8*

*;平坦模式栈段选择子*

*STACK\_SELECTOR equ 0x10*

*;平坦模式视频段选择子*

*VIDEO\_SELECTOR equ 0x18*

*VIDEO\_NUM equ 0x18*

*;平坦模式代码段选择子*

*CODE\_SELECTOR equ 0x20*

*接下来打开第21根地址线*

*in al,0x92 ;南桥芯片内的端口*

*or al,0000\_0010B*

*out 0x92,al ;打开A20*

*设置PE位：*

*cli ;中断机制尚未工作*

*mov eax,cr0*

*or eax,1*

*mov cr0,eax ;设置PE位*

最后一步，远跳转进入保护模式。

*jmp dword CODE\_SELECTOR:protect\_mode\_begin*

此时，jmp指令将CODE\_SELECTOR送入cs，将protect\_mode\_begin + LOADER\_START\_ADDRESS送入eip，进入保护模式。然后我们将选择子放入对应的段寄存器。

*;16位的描述符选择子：32位偏移*

*;清流水线并串行化处理器*

*[bits 32]*

*protect\_mode\_begin:*

*mov eax, DATA\_SELECTOR ;加载数据段(0..4GB)选择子*

*mov ds, eax*

*mov es, eax*

*mov eax, STACK\_SELECTOR*

*mov ss, eax*

*mov eax, VIDEO\_SELECTOR*

*mov gs, eax*

最后，我们输出“enter protect mode”。

*mov ecx, protect\_mode\_tag\_end - protect\_mode\_tag*

*mov ebx, 80 \* 2*

*mov esi, protect\_mode\_tag*

*mov ah, 0x3*

*output\_protect\_mode\_tag:*

*mov al, [esi]*

*mov word[gs:ebx], ax*

*add ebx, 2*

*inc esi*

*loop output\_protect\_mode\_tag*

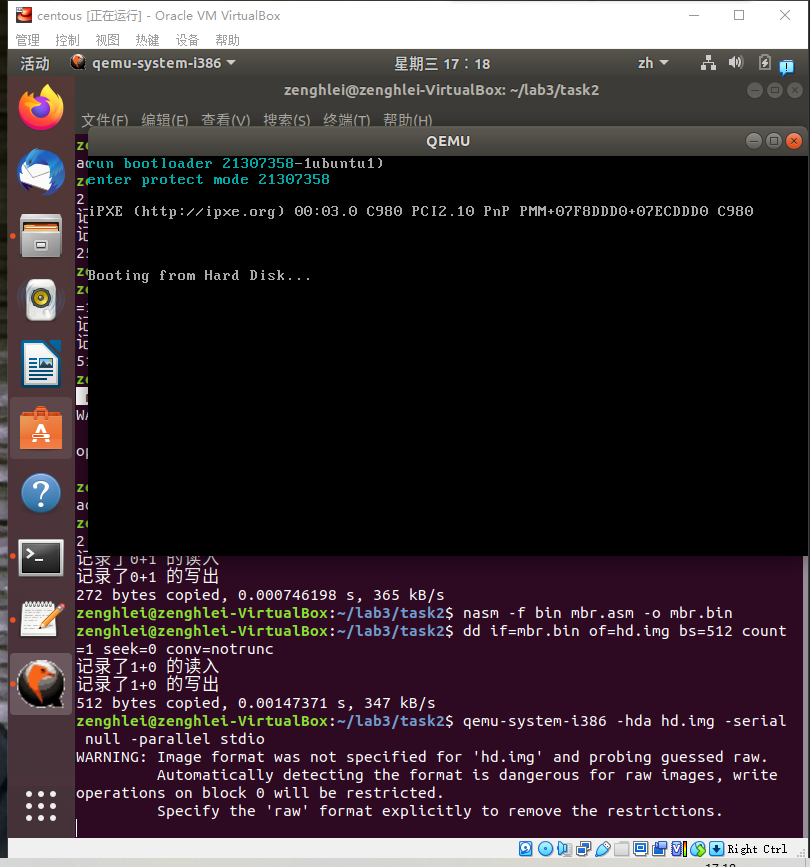
*... ; 省略*

*protect\_mode\_tag db 'enter protect mode'*

*protect\_mode\_tag\_end:*

最后一步，我们改造mbr，这一部分的改造代码同样可以参考任务1，这里不再赘述

汇编、写入文件后，qemu运行结果如下所示：



* 1. 调试部分

在调试之前，我们需要先对bootloader.asm和mbr.asm进行编译操作生成两个文件的符号表，这样才能在gdb中对其中的内容设置断点。

我们编译mbr.asm，生成可重定位文件mbr.o。其中，-g参数是为了加上debug信息。

nasm -o mbr.o -g -f elf32 mbr.asm

然后我们为可重定位文件mbr.o指定起始地址0x7c00，分别链接生成可执行文件mbr.symbol和mbr.bin（注意，这一步会在lab4中讲解，如果还没学习到lab4的内容，可以暂时跳过）。

ld -o mbr.symbol -melf\_i386 -N mbr.o -Ttext 0x7c00

ld -o mbr.bin -melf\_i386 -N mbr.o -Ttext 0x7c00 --oformat binary

对于bootloader.asm，我们执行上述类似的操作。

nasm -o bootloader.o -g -f elf32 bootloader.asm

ld -o bootloader.symbol -melf\_i386 -N bootloader.o -Ttext 0x7e00

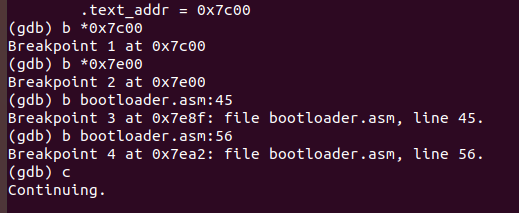
ld -o bootloader.bin -melf\_i386 -N bootloader.o -Ttext 0x7e00 --oformat binary

然后将mbr.bin和bootloader.bin分别写入hd.img，写入的位置是lab2-Example 2中指定的位置。

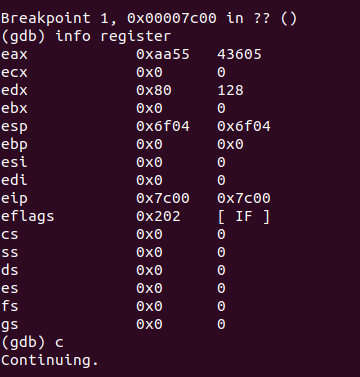
dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc

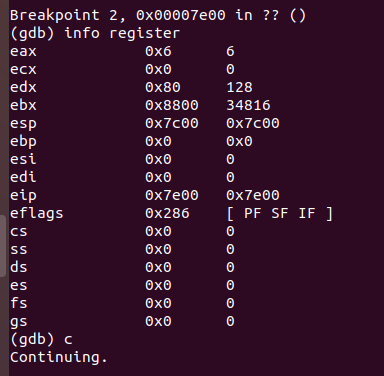
dd if=bootloader.bin of=hd.img bs=512 count=5 seek=1 conv=notrunc

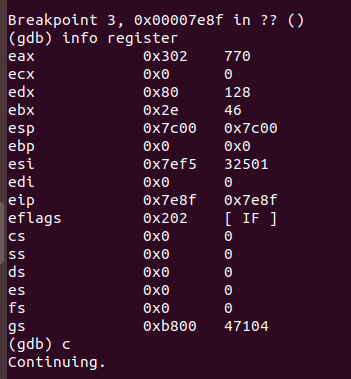
接着，在进入保护模式的四个重要步骤上设置断点

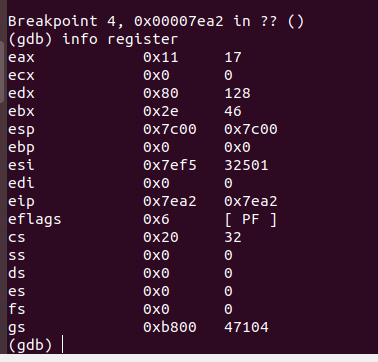


在运行过程中，四个断点的寄存器状态如图所示（按顺序）









在gdb下，我们使用info registers查看寄存器，可以看到段寄存器的内容变成了段选择子。

**3.小程序移植**

本任务主要考察的是一个程序的移植。在保留lab2 assignment4源代码的基础上，补充好进入保护模式的四个步骤以及对源代码中16位寄存器的修改就可以完成本任务。由于本任务的关键代码非常长，所以这里不赘述，关键代码文件在附件/src/task3中，有bootloader.asm和mbr.asm。

运行结果如下所示：



1. **总结**

实验教程里面写的都很详细。本次实验我初步理解了进入保护模式的原理以及bootloader和mbr的基本关系，也了解了移植大于512字节的程序所需要的基本步骤。

我在做任务2的时候在符号表那里卡了很久，而且上网也几乎查不到相关的资料，后来问了宿友才知道是gdbinit那里的第三第四步要在编译文件前加上绝对路径…可以的话希望以后的实验资料能备注的稍微详细一点orz大感谢了