



本科生实验报告

实验课程: 操作系统实验

实验名称: lab3 从实模式到保护模式

专业名称: 计算机科学与技术

学生姓名: 杜翊菲

学生学号: 23336062

实验地点: 实验大楼 B201

实验成绩:

报告时间: 2025 年 3 月 29 日

Lab3 从实模式到保护模式

一、实验要求

1. 学习到如何从 16 位的实模式跳转到 32 位的保护模式，然后在平坦模式下运行 32 位程序
2. 学习到如何使用 I/O 端口和硬件交互，为后面保护模式编程打下基础

二、实验过程（每个部分包括实验操作、关键代码和结果截图）

（一）Example1: bootloader 的加载

1. 复现 Example 1

在实验开始前，在家目录下创建lab3文件夹并进入其中：

```
mkdir ~/lab3 && cd lab3
```

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被安排在MBR之后，预留5个扇区的空间。

name	start	length	end
MBR	0x7c00	0x200(512B)	0x7e00
bootloader	0x7e00	0xa00(512B * 5)	0x8800

在lab3目录下，我们先新建一个文件 `bootloader.asm`，然后将lab2的 `mbr.asm` 中输出Hello World部份的代码，放入 `bootloader.asm`，`bootloader.asm` 如下所示。

```
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'
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。

```
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

```

由于在我们的实验中，我们假设bootloader不会超过5个扇区。但是，直接在MBR的代码中指定bootloader的大小是危险的，我们在第二个例子中再讨论这个问题。

将bootloader读取到起始位置为0x7e00的内存后，我们执行远跳转到0x7e00。注意，我们现在是在实模式下，`0x0000:0x7e00` 表示段地址为0x0000, 偏移地址为0x7e00。执行这条语句实际上等价于以下过程。其中，符号 `:=` 表示赋值。

```

cs := 0x0000
ip := 0x7e00

```

但是，执行下面这条语句是错误的，因为这是近跳转。

```

jmp 0x7e00

```

等价于以下过程。实际上是跳转到距离下一条指令地址为0x7e00处的地址，不是我们期望的。

```
ip := ip + 0x7e00
cs 不变
```

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。

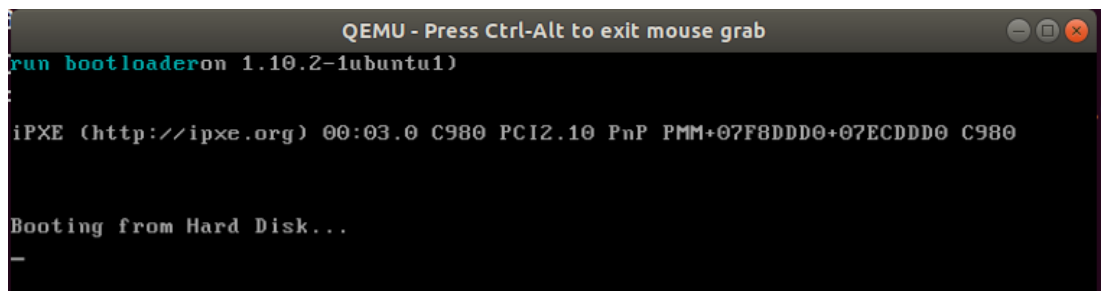
```
nasm -f bin mbr.asm -o mbr.bin
dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
```

使用qemu运行即可，

```
qemu-system-i386 -hda hd.img -serial null -parallel stdio
```

由于代码已经给出示例，我们只需按如上教程操作后，即可成功复现 Example1，截图如下：

```
yifeidu@one:~$ cd ~/lab3
yifeidu@one:~/lab3$ nasm -f bin bootloader.asm -o bootloader.bin
yifeidu@one:~/lab3$ dd if=bootloader.bin of=hd.img bs=512 count=5 seek=1 conv=notrunc
记录了0+1 的读入
记录了0+1 的写出
52 bytes copied, 0.000188171 s, 276 kB/s
yifeidu@one:~/lab3$ nasm -f bin mbr.asm -o mbr.bin
yifeidu@one:~/lab3$ dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
记录了1+0 的读入
记录了1+0 的写出
512 bytes copied, 0.000181763 s, 2.8 MB/s
yifeidu@one:~/lab3$ qemu-system-i386 -hda hd.img -serial null -parallel stdio
WARNING: Image format was not specified for 'hd.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write operations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```



```
QEMU - Press Ctrl-Alt to exit mouse grab
run bootloader on 1.10.2-1ubuntu1)
iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980
Booting from Hard Disk...
```

2. 修改 Example1，将 LBA28 读取硬盘的方式换成 CHS 读取

这里涉及到 C/H/S 与 LBA 的转换关系。

物理寻址方式：

又称 CHS (Cylinder 柱面/Head 磁头/Sector 扇区) 方式，用柱面号（即磁道号）、磁头号（即盘面号）和扇区号来表示一个特定扇区。柱面和扇区从 0 开始编号，扇区从 1 开始编号。

磁盘容量=磁头数×柱面数×扇区数×512 字节

逻辑寻址方式：

以扇区为单位的线性寻址，即 LBA，将所有的扇区统一编号。C/H/S 中的扇区编号是从“1”至“63”，而逻辑扇区 LBA 方式下扇区是从“0”开始编号，所有扇区编号按顺序进行。对于任何一个硬盘，都可以认为其扇区是从 0 号开始。

CHS 与 LBA 之间的相互转换：在 CHS 寻址方式中，读取某一扇区之间要读取的扇区数即为此扇区的 LBA 参数。

逻辑编号（即 LBA 地址）=（柱面编号×磁头数+磁头编号）×扇区数+扇区编号-1【磁头数：硬盘磁头的总数，扇区数：每磁道的扇区数】

C/H/S 地址			LBA 编号
柱面	磁头	扇区	
0	0	1	0
0	0	2	1
0	0	3-63	2-62
0	1	1	63
0	1	2-63	64-125
0	2	1-63	126-188
0	3	1-63	189-251
1	0	1	252
1	0	2-63	253-314
1	1	1	315
...

(1) 修改后的关键代码：

```

; 设置CHS参数（示例：读取柱面0，磁头0，扇区2）
mov cx, 0x0001 ; 柱面号（高8位）和磁头号（低8位）
mov dh, 0x00   ; 磁头号（高8位）
mov cl, 0x02   ; 扇区号（1-63）

```

原 LBA 参数 ax（逻辑扇区号）改为 CHS 参数 cx, dh, cl

(2) 运行截图如下：

```

yifeidu@one: ~/lab3
yifeidu@one:~$ cd ~/lab3
yifeidu@one:~/lab3$ nasm -f bin mbr.asm -o mbr.bin
yifeidu@one:~/lab3$ dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv=notrunc
记录了1+0 的读入
记录了1+0 的写出
512 bytes copied, 0.000386727 s, 1.3 MB/s
yifeidu@one:~/lab3$ qemu-system-i386 -hda hd.img -serial null -parallel stdio
WARNING: Image format was not specified for 'hd.img' and probing guessed raw.
        Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
        Specify the 'raw' format explicitly to remove the restrictions.

```

(二) Example2: 进入保护模式

在第二个例子中，我们根据前面“进入保护模式”的描述，在bootloader中进入保护模式，并在进入保护模式后在显示屏上输出‘protect mode’。

在进入保护模式之前，我们先对我们的内存地址进行规划。

Name	Start	Length	End
MBR	0x7c00	0x200(512B)	0x7e00
bootloader	0x7e00	0xa00(512B * 5)	0x8800
GDT	0x8800	0x80(8B * 16)	0x8880

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
```

其中，`equ` 是汇编伪指令。例如，编译器会在编译时将 `LOADER_SECTOR_COUNT` 出现的地方替换成 `5`，`LOADER_SECTOR_COUNT equ 5` 不会对应任何的二进制指令，即不会出现在最终的bin格式文件中。

在操作系统内核设计的过程中，内存规划是一件令人苦恼的事情。从上面的例子可以看到，`bootloader`紧跟在MBR后面，`GDT`紧跟在`bootloader`后面，看起来非常紧凑。但是，只要其中一个发生变化，那么可能我们要重新规划内存。也就是说，没有一种内存规划方案是完美的。

规划好内存后，我们就准备进入保护模式。回忆一下我们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。**

我们下面分步来看，完整代码放置在 `src/example-2/bootloader.asm` 下。

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。即便如此，前面已经讲过，段的存在是为了让CPU执行段保护，阻止程序越界访问。但是，我们在后面的实验中会实现二级分页机制，此时页保护也可以阻止程序越界访问。也就是说，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。这种内存访问模式被称为平坦模式。

使用平坦模式的另外一个原因是：C/C++在编译和链接后得到的二进制程序默认是运行在平坦模式下的，即C/C++程序从线性地址0开始，可以访问4GB的内存空间。C/C++的这种默认方式是为了方便程序重定位，我们通过地址变换部件和分页机制来将程序的线性地址变换成物理地址。当然，我们到后面再讨论这个问题。

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，部分 `bootloader.asm` 代码如下。

```
%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
```

为了让CPU知道GDT的位置，我们需要设置GDTR寄存器。回忆一下GDTR寄存器，其高32位表示GDT的起始地址，低16位表示GDT的界限。所谓界限，就是现在GDT的长度减去1。此时，我们已经放入5个段描述符，因此，GDT的界限如下所示。

$$\text{界限} = 8 * 5 - 1 = 39$$

我们在内存中使用一个48位的变量来表示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。

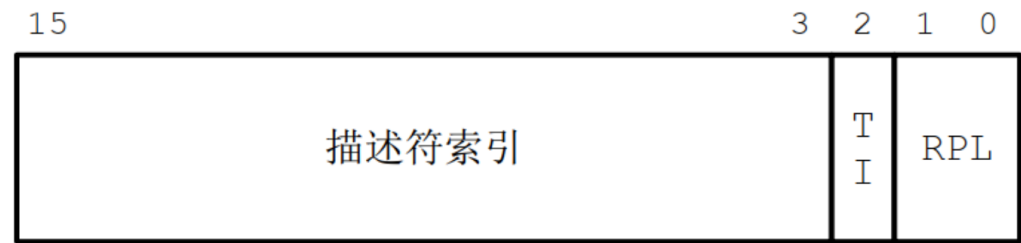
```
;初始化描述符表寄存器GDTR
mov word [pgdt], 39 ;描述符表的界限
lgdt [pgdt]
```

然后根据段描述符的内容设置段选择子。

```
; _____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
```

我们以代码段选择子为例来解释段选择子的含义，段选择子的结构如下。



- 代码段描述符是GDT中第4个描述符，因此高13位为4。
- TI=0表示GDT，第2位为0。
- 我们的特权级设为最高特权级，因此RPL为0。

因此代码段选择子为0x20。此时，GDT已经设置完毕，我们现在结合段描述符的结构研究下代码段的内存和具体含义，段描述符如下所示。



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，因此段线性基地址为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0xffff，共20位。结合粒度和段界限，整个代码段的长度计算如下。

$$\text{长度} = (\text{段界限} + 1) * \text{粒度} = (0xffff + 1) * 4KB = 2^{20} * 2^{12}B = 2^{32}B = 4GB$$

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。

其他段描述符的分析方法类似，同学们可以自行分析。注意，数据段和栈段的寻址空间都是0x00000000~0xffffffff，线性地址也都是由偏移地址直接给出，非常方便。视频段并不是平坦模式，而是仅限于显存的表示范围。

准备好GDT后，接下来的内容就非常轻松了。

我们先打开第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:

```

至此，我们差不多已经完成，最终得到的 `bootloader.asm` 如下所示：

```

%include "boot.inc"
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

;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

;初始化描述符表寄存器GDTR
mov word [pgdt], 39 ;描述符表的界限
lgdt [pgdt]

```

```

in al,0x92 ;南桥芯片内的端口
or al,0000_0010B
out 0x92,al ;打开A20

cli ;中断机制尚未工作
mov eax,cr0
or eax,1
mov cr0,eax ;设置PE位

```

```

;以下进入保护模式
jmp dword CODE_SELECTOR:protect_mode_begin

```

```

;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

```

```

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

jmp $ ; 死循环

```

```

pgdt dw 0
      dd GDT_START_ADDRESS

bootloader_tag db 'run bootloader'
bootloader_tag_end:

protect_mode_tag db 'enter protect mode'
protect_mode_tag_end:

```

但我们需要改造下 `mbr.asm` 。如下所示。

```

%include "boot.inc"

[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, LOADER_START_SECTOR
mov cx, LOADER_SECTOR_COUNT
mov bx, LOADER_START_ADDRESS

load_bootloader:
    push ax
    push bx
    call asm_read_hard_disk ; 读取硬盘
    add sp, 4
    inc ax
    add bx, 512
    loop load_bootloader

    jmp 0x0000:0x7e00 ; 跳转到bootloader

jmp $ ; 死循环

```

```

; asm_read_hard_disk(memory,block)
; 加载逻辑扇区号为block的扇区到内存地址memory

```

```

asm_read_hard_disk:
    push bp
    mov bp, sp

    push ax
    push bx
    push cx
    push dx

```

```

mov ax, [bp + 2 * 3] ; 逻辑扇区低16位

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

xor ax, ax
inc dx ; 0x1f5
out dx, al ; LBA地址23~16 = 0

inc dx ; 0x1f6
mov al, ah
and al, 0x0f
or al, 0xe0 ; LBA地址27~24 = 0
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 bx, [bp + 2 * 2]
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

pop dx
pop cx
pop bx
pop ax
pop bp

ret

```

```
times 510 - ($ - $$) db 0
db 0x55, 0xaa
```

这里涉及到C/C++参数传递的规则，对于一个带有参数和返回值的C函数。

```
int function(arg1, arg2, arg3);
```

将其翻译汇编代码后，在调用function前，汇编代码依次将arg3, arg2, arg1入栈，然后将eip的内容入栈作为返回地址。接着，汇编代码跳转到function的第一条指令中执行，通过ebp来引用函数在栈中的参数。当函数返回后，汇编代码将返回值放入到eax中，将之前保存的栈顶的返回地址送入eip，程序返回。当程序返回后，之前压栈的参数还保留在栈中，此时汇编代码会调整esp的内容来清除栈中的函数参数。

以上面的例子为例，我们读磁盘的函数可以如下表示。

```
void asm_read_hard_disk(int memory, int block)
```

调用前后的代码如下。

```
push ax
push bx
call asm_read_hard_disk ; 读取硬盘
add sp, 4
inc ax
add bx, 512
```

此时，ax保存的是逻辑扇区号，bx保存的是加载地址，因此ax先入栈，bx后入栈。调用完asm_read_hard_disk后，栈中还保留着之前的函数参数，而每个参数占2个字节(16位)，因此add sp, 4相当于连续pop掉两个参数。

使用和example 1相同的命令，我们编译汇编代码：

创建img：

```
qemu-img create hd.img 10m
```

首先我们编译 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
```

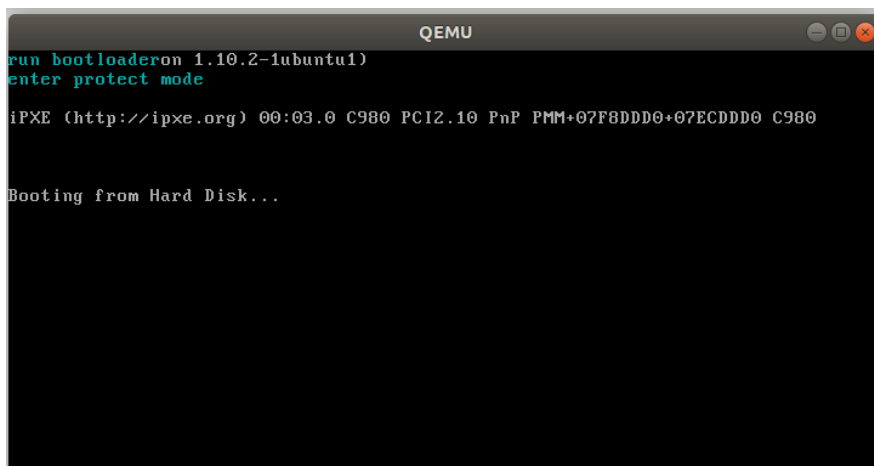

使用qemu运行即可,

```
qemu-system-i386 -hda hd.img -serial null -parallel stdio
```

按照以上步骤调试运行, 结果如下:

(1) 进入保护模式:

```
yifeidu@one:~$ cd ~/lab3/example-2
yifeidu@one:~/lab3/example-2$ qemu-img create hd.img 10m
Formatting 'hd.img', fmt=raw size=10485760
yifeidu@one:~/lab3/example-2$ nasm -f bin bootloader.asm -o bootloader.bin
yifeidu@one:~/lab3/example-2$ dd if=bootloader.bin of=hd.img bs=512 count=5 seek
=1 conv=notrunc
记录了0+1 的读入
记录了0+1 的写出
254 bytes copied, 0.000471305 s, 539 kB/s
yifeidu@one:~/lab3/example-2$ nasm -f bin mbr.asm -o mbr.bin
yifeidu@one:~/lab3/example-2$ dd if=mbr.bin of=hd.img bs=512 count=1 seek=0 conv
=notrunc
记录了1+0 的读入
记录了1+0 的写出
512 bytes copied, 0.000436107 s, 1.2 MB/s
yifeidu@one:~/lab3/example-2$ qemu-system-i386 -hda hd.img -serial null -paralle
l stdio
WARNING: Image format was not specified for 'hd.img' and probing guessed raw.
Automatically detecting the format is dangerous for raw images, write o
perations on block 0 will be restricted.
Specify the 'raw' format explicitly to remove the restrictions.
```

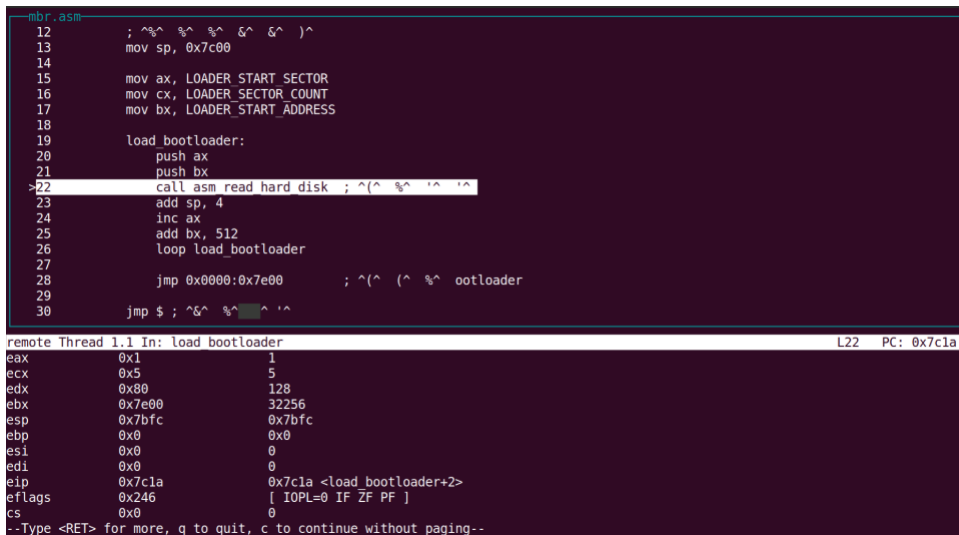


```
run bootloader on 1.10.2-1ubuntu1
enter protect mode

iPXE (http://ipxe.org) 00:03.0 C980 PCI2.10 PnP PMM+07F8DDDD+07ECDDDD C980

Booting from Hard Disk...
```

(2) Debug 过程



```
mbr.asm
12 ; ^ ^ ^ ^ ^ ^ ^ ^ ^ ^
13 mov sp, 0x7c00
14
15 mov ax, LOADER_START_SECTOR
16 mov cx, LOADER_SECTOR_COUNT
17 mov bx, LOADER_START_ADDRESS
18
19 load bootloader:
20 push ax
21 push bx
22 call asm read hard disk ; ^ ^ ^ ^ ^ ^
23 add sp, 4
24 inc ax
25 add bx, 512
26 loop load_bootloader
27
28 jmp 0x0000:0x7e00 ; ^ ^ ^ ^ ^ ^ ootloader
29
30 jmp $ ; ^ ^ ^ ^ ^ ^ ^ ^ ^ ^

remote Thread 1.1 In: load bootloader L22 PC: 0x7c1a
eax 0x1 1
ecx 0x5 5
edx 0x80 128
ebx 0x7e00 32256
esp 0x7bfc 0x7bfc
ebp 0x0 0
esi 0x0 0
edi 0x0 0
eip 0x7c1a 0x7c1a <load_bootloader+2>
eflags 0x246 [ IOPL=0 IF ZF PF ]
cs 0x0 0
--Type <RET> for more, q to quit, c to continue without paging--
```

