# 从Bootloader开始

Bootloader是计算机启动过程中的必要执行环节，它负责将操作系统内核从启动设备(通常是软盘或硬盘)加载到内存中执行的具体工作。这部分内容主要讨论一个简单有效的Bootloader应该完成哪些必要的工作，并且介绍如何来编码实现一个Bootloader。

Bootloader因其与系统启动引导过程紧密相关，通常只能采用汇编语言进行编写，于是也就成为了系统中唯一一处使用较多汇编开发的模块。为了减少使用汇编进行开发的难度，我们使Bootloader所负责的工作最小化，完成将位于磁盘上的内核映像加载到内存中的特定位置，以及为完成该任务的稍许准备工作，即为Bootloader的全部职责。

然而，我们并不准备一蹴而就直接实现一个功能完备的的最终版本Bootloader，而是将采取循序渐进的方式，从最基本的让Bootloader显示字符开始，逐步加入进入保护模式，取磁盘，加载内核到内存的能力，直至完整功能的Bootloader。

**[Version 0 from Tachikoma]**

## Version 0:“Hello world!”Bootloader

这个版本Bootloader的目标是：在屏幕上打印”Hello world!”字样！下面的代码1.1实现这样的功能，它调用BIOS 0x10# 中断，在屏幕上显示字符串。

**代码1.1** “Hello World!” Bootloader from boot.s

org 0x7c00

; 对齐到 0x7c00，BIOS从这个入口启动OS

[BITS 16] ; 实模式

start:

mov ax,cs

mov ds,ax ; 置数据段地址

mov es,ax

mov bp,msg ; es:bp = msg

mov cx,13 ;串长 = 13

mov dx,0x1800 ; DH,DL = 起始行、列 = 24,0

mov bx,0x000c ; BH = 页号 = 0; BL = 样式 = 0xc = 黑底红字

mov ax,0x1301 ; AH = 0x13 , 调用0x10中断功能为显示字符串; AL = 0x1 , 光标跟随移动，BL表示样式

int 0x10 ;调用BIOS中断#10 打印文字

loop:

jmp loop ;死循环

msg db 'Hello world !'

times 510-($-$$) DB 0

DW 0xAA55 ; 最后两行用于补齐512字节大小

; 关于为什么不写成

; ORG 510

; DW 0xAA55

; 参看 NASM 手册

对于代码基本的理解已经详细的写在注释里，当然，注释不足以让我们完全理解这段代码，缺失的部分我们将在看到"Hello World!"后再作补偿。当前的首要任务是，编译并让Bootloader运行出正确的效果。

**编译：**我们使用 nasm来对boot.s源代码进行编译，生成boot.bin，格式为纯二进制格式，大小恰好为512字节。编译命令如下：

**nasm -f bin boot.s -o boot.bin**

**部署：**我们知道，Bootloader应当被写入到磁盘的第一扇区(0柱面0磁头1扇区)中。Bochs使用普通文件来模拟物理磁盘设备，对于宿主操作系统来说仅仅是一个普通文件，而对于虚拟机中操作系统而言，它是一块真实的磁盘，这类文件也被称为虚拟磁盘文件。在我们的例子中使用的是一个名为C.IMG的虚拟磁盘文件。物理磁盘设备第一扇区(0柱面0磁头1扇区)，对虚拟磁盘文件来说便是该文件头部512字节区域。

由此可知，部署的任务便是将boot.bin写入到C.IMG的头部512字节区域中。部署过程的命令如下：

**partcopy boot.bin 0 200 c.img 0**

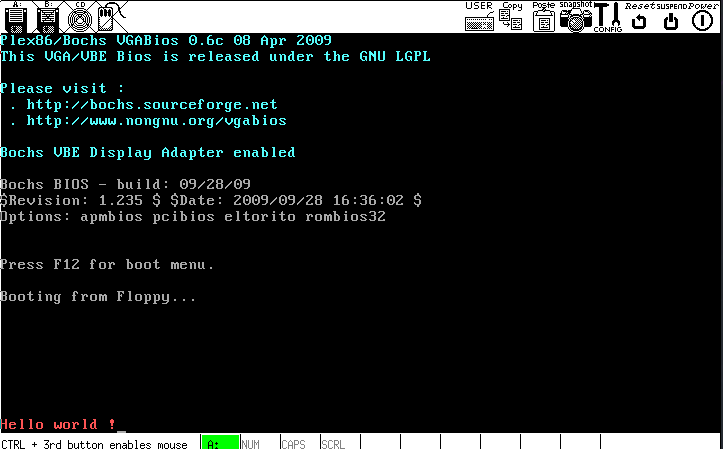
这里我们需要借助一个叫做PARTCOPY的小工具来完成这样的特殊“拷贝”。它与通常的copy命令功能不同在于，它允许我们拷贝源文件中的部分数据去覆盖目标文件中的某个偏移量开始处的数据。

PARTCOPY命令格式为：PARTCOPY SRC source\_offset length DST dest\_offset，参数默认使用16进制。

**运行：**启动Bochs虚拟机，以bochsrc.bxrc为配置文件。 (-q选项去除了不必要的bochs提示)。

**bochs -q -f bochsrc.bxrc**

可以看到效果如图1.1所示，左下角出现了Hello world ! 的字样，我们仍然需要面对遗留的疑问。



**图1.1** “Hello World!” Bootloader运行效果

目前为止的示例中体现了从系统启动，并成功运行我们编写的Bootloader中的代码，打印出”Hello World!”字样，但是代码中仍然有些疑问，譬如：

* 为什么是org 0x7C00而不是其它任意值？
* 为什么最后两个字节为0xAA55？

**[Part 2: from Jeou]**

## 系统初启

### 系统引导过程

系统初启时，CPU处于实模式下(实模式我们要等到保护模式再详细说明)，其CS:IP寄存器被设置为FFFF:0000，实际访问的是物理地址0xFFFF0，通常该单元中仅仅存放一条jmp指令，跳转到BIOS自检程序入口处开始执行。BIOS自检程序是由硬件生产厂商预置在系统ROM中的一段引导程序，BIOS自检程序会完成一系列对外设的检测和初始化操作。

到目前为止，我们的OS并没有参与，一切都是由系统自动完成的，随后BIOS自检程序依据用户指定的顺序查找启动设备。通过检查存储设备上的第一个扇区0柱面0磁头1扇区中的末尾2个字节是否为0x55AA，来判断它是否为一个引导扇区(Boot Sector)。如果是0x55AA，则BIOS自检程序会将该引导扇区读入到内存0x7C00处，并跳转到CS:IP = 0000:7C00处开始执行Bootloader的第一条指令，正式将控制权交给引导扇区中的Bootloader代码。至此，BIOS程序完成了所有工作，Bootloader开始执行，我们的OS开始接手工作。

### 需要使用org 0x7C00的原因

这样就可以解释为什么是org 0x7C00而不是其它任意值？org 指令的作用是指示程序将要被加载到内存的起始地址，并对程序段中每一次对内存的访问都加上偏移量0x7C00(没有org指令时的偏移量为0)。换句话说，取定org 0x7C00而非其它任意值是为了与PC启动过程中引导扇区加载到物理内存内存0x7C00处的规范相互吻合。

举例来说，如果取值org 0x8c00，而引导扇区却被加载到0x7C00，这意味着每次对内存的引用加上0x8c00偏移量，显然访问不到正确的引导扇区中的数据。

### 引导扇区标志字节0x55AA

代码中我们将引导扇区512字节中的最后2个字节填充为0x55AA，这是判断一个扇区是否为引导扇区的标志字节，BIOS自检程序通过判断这2个标志字节来决定是否加载引导扇区到内存中，也就是说，无论我们编写的Bootloader逻辑如何正确，如果缺少了这2个标志字节，那么它连被加载到内存中执行的机会都没有。

语句**times 510-($-$$) DB 0**是为了填充一系列的0直至塞满前510个字节，然后补上末尾的0x55AA两个标志字节。$$表示当前节(section)起始处的字节偏移值，而$表示的是当前指令的字节偏移值，$和$$的具体值要到编译时才可以确定。具体在我们代码中，当前节起始处的字节偏移值$$应该为0，而$的值则为当前其所属的times语句编译后的字节偏移，那么两者相减($-$$)的值便是整段程序编译后占据的字节长度，总共510字节减去Bootloader中程序段占据的长度即为需要填充的0的个数。这便是510-($-$$)取值的由来。

[Version 1 from Tachikoma]

## Version 1：走进保护模式的时代

先前Version 0中的Bootloader仅仅演示了系统启动时成功执行我们编写的Bootloader的效果，在屏幕上打印出一串字符，但是还毫不具备我们需要的功能。这样的Bootloader有意义吗？

显然是有的，通过它我们至少知道如何编写一个能够被正确执行到的引导扇区，知道任何一个Bootloader所需具备的要素有哪些。在此基础上，将显示字符串的代码替换成有实际意义的功能。这个Bootloader版本的目标是：

* 从实模式进入保护模式，突破1MB空间的限制
* 由于启用保护模式后，BIOS调用不再可用，我们需要新的方法在屏幕上显示反馈，方便我们验证

一般来说，从实模式进入保护模式需要以下四个关键步骤：

1. 建立GDT表与GDTR数据结构
2. 加载GDT(即设置GDTR寄存器指向GDT)
3. 设置CR0寄存器中的PE(Protection Enable)位来启用保护模式
4. 一条JMP指令，由实模式跳入保护模式

首先，我们需要建立了一个GDT表，其中包含4个GDT表项，即4个段描述符(Segment Descriptor)，每个描述符所描述段的属性在表1.1中给出，和代码1.2中的GDT表完全等价。

**代码1.2** 建立GDT表 from boot.s

gdt: ; gdt表头的占位描述符

dw 0x0000

dw 0x0000

dw 0x0000

dw 0x0000 ;GDT表第0项必须为全零

label\_kernel\_code: ; 核心态代码段描述符

dw 0xFFFF ; limit(L15~0)=0xffff

dw 0x0000 ; base(B15~0)=0x0000

dw 0x9A00 ; P=1 , 段存在内存

; DPL=0 , 特权级

; TYPE=1010 , 执行/可读

; B23~16 = 0x00

dw 0x00CF ; B31~24 = 0x00

; G=1 , 段限单位为 4KB

; D/B=1 , 32bit

; L19~16=0xf

label\_kernel\_data: ; 核心态数据段描述符

dw 0xFFFF ; limit(L15~0)=0xffff

dw 0x0000 ; base(B15~0)=0x0000

dw 0x9200 ; P=1 , 段存在内存

; DPL=0 , 特权级

; TYPE=0010 , 可读/写

; B23~16 = 0x00

dw 0x00CF ; B31~24 = 0x00

; G=1 , 段限单位为 4KB

; D/B=1 , 32bit

; L19~16=0xf

label\_video: ; 以显存地址0xB8000开始的段的描述符（为了达到测试GDT的目的）

dw 0xFFFF ; limit(L15~0)=0xffff

dw 0x8000 ; base(B15~0)=0x8000

dw 0x920B ; P=1 , 段存在内存

; DPL=0 , 特权级

; TYPE=0010 , 可读/写

; B23~16 = 0x0B

dw 0x00CF ; B31~24 = 0x00

; G=1 , 段限单位为 4KB

; D/B=1 , 32bit

; L19~16=0xf

;GDT End

**表1.1** GDT表中段描述符属性

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 段描述符 | 起始地址 | 限长 | G | S | Type | DPL | D/B | P |
| 空描述符 | 0x00000000 | 0x0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 核心态代码段 | 0x00000000 | 0xfffff | 1 | 1 | 执行/可读 | 0 | 1 | 1 |
| 核心态数据段 | 0x00000000 | 0xfffff | 1 | 1 | 可读/写 | 0 | 1 | 1 |
| 显存段 | 0xB8000 | 0xfffff | 1 | 1 | 可读/写 | 0 | 1 | 1 |

从表1.1中可以看到，GDT表中第零项全部设置为0，以此构造一个空描述符作为占位符。GDT中有效描述符从第一项开始，我们总共定义了核心态代码段，核心态数据段，以及显存段三个有效段描述符。

核心态代码段、数据段的属性大致相同，起始地址都是0x00000000，段长为0xFFFFF，特权级DPL都为0级；代码段和数据段的区别在于：代码段是执行/可读，数据段是可读/写。

显存段起始地址为0xB8000，段长为0xFFFFF，其作用适用于保护模式下向显存写入信息，这些信息会反馈在屏幕上，以此验证程序的正确性。

另外为了方便，代码1.3中我们定义了一些段选择子常量。

**代码1.3** 段选择子常量 from boot.s

kernel\_code\_selector equ label\_kernel\_code - gdt

kernel\_data\_selector equ label\_kernel\_data - gdt

video\_selector equ label\_video – gdt

随后，我们建立一个gdtr，它包含两部分信息，GDT表的起始地址和限长，简略的说，它解决了系统应该到哪里找到GDT的问题。gdtr数据结构长度为6个字节，其中2字节的限长(Limit)在前，4字节的起始地址(Base Address)在后。gdtr代码片段(代码1.4所示)紧随gdt代码片段，这样可以通过$-gdt计算GDT表总长度。

**代码1.4** gdtr数据结构 from boot.s

gdtr: ;gdtr数据结构

dw $-gdt ; GDT Limit

dd gdt ; GDT Base Address

在准备完GDT表和gdtr这两个保护模式所必需的数据结构之后， 意味着进入保护模式前的一切准备工作就绪了。接下来就可以执行进保护模式的操作了，具体方法如代码1.5所示。

**代码1.5** 进入保护模式 from boot.s

[BITS 16] ;实模式

lgdt [gdtr] ;根据gdtr所指示的内容加载GDTR寄存器

mov eax, cr0

or eax, 1

mov cr0,eax ; 启用保护模式

jmp dword kernel\_code\_selector:\_start ; 设置 CS 和 EIP 为保护模式的地址形式

[BITS 32] ; 保护模式 32位代码

\_start:

mov ax, kernel\_data\_selector

mov ds, ax

mov es, ax

mov ss, ax ; 设置 数据段寄存器 为保护模式的地址形式

...

这里必须要说明一下流程：

1. 加载gdtr数据结构到GDTR寄存器，系统通过gdtr了解到gdt到底在哪里。
2. 通过修改CR0控制寄存器的PE比特位（相当于保护模式开关）来启用保护模式。
3. 一条JMP指令，跳转入保护模式。此刻，所有的段寄存器(CS、DS、ES、SS)值并不是段选择子，而是从先前实模式下继承来的段基地址，我们需要修正它们：
   1. 通过 jmp CS : EIP 的形式，修改代码段寄存器CS=kernel\_code\_selector，EIP=\_start，该JMP指令使程序跳转至\_start标号处在保护模式下继续执行。
   2. \_start后面几句，修改数据段寄存器(DS、ES、SS)=kernel\_data\_selector。

最后，我们需要看到效果，就需要往屏幕上输出些信息。而这里需要新办法显示回馈信息。为什么不继续采用Version 0中的方式呢？这是因为原先实模式下BIOS中断在保护模式下不再可用，我们需要另外的方式—直接往显存中写字符，显存地址开始于0xB8000，直接向其中写值便可显示在屏幕上。

**代码 1.6** 显示回馈字符’P’ from boot.s

mov ax, video\_selector

mov gs, ax

mov ah,0xc ;黑底红字

mov al,'P'

mov [gs:80\*2\*24],ax

; 已经在GDT中添加了第三个段，段基址为0xB8000(显存起始地址)

; 主要为了测试GDT起了作用

; 上面语句在屏幕第25行输出红色的P

; (80为每行字符数，\*2因为偏移是字节量)

**编译、部署、运行：**与Version 0中的编译、部署、运行步骤使用的命令行没有区别，如下所示：

**nasm -f bin boot.s -o boot.bin**

**partcopy boot.bin 0 200 c.img 0**

**bochs -q -f bochsrc.bxrc**

运行可以看到效果如图1.1所示，左下角显示了红色的P，表明保护模式已经成功启用，GDT已经生效（我们使用了video\_selector），自此 我们彻底摆脱了1M地址空间的束缚。



**图1.2** 进入保护模式Bootloader运行效果

**我们的问题：**

我们需要详细了解下面的内容：

1. 保护模式到底是什么？它与实模式的区别在哪里？

2. GDT,GDTR

3. 段选择子

4. CR0寄存器

什么，你已经知道这些问题了？好吧，我们只是想为那些迷茫的人指一条明路，上述问题的答案都可以在第二部分保护模式章节中寻得（警告：保护模式并非三言两语可以道尽，阅读之前请先培养好足够的耐心）

## Version 2：加载内核映像

先前的Bootloader版本中，无论是显示一行欢迎字符或者是从实模式进入保护模式，都并非Bootloader的最终目的，而仅仅是一些准备工作，并且这些准备阶段工作因操作系统不同会有所出入，但基本上大同小异。不过Bootloader最终目标则相对唯一，无非是将内核从磁盘加载到内存中。对V6++的Bootloader具体而言，**就是将内核从磁盘加载到1M起始的物理内存中。**

此目标看似简单，其实包含着几个有待解决的问题。内核存放在磁盘上什么地方？内核本身是什么格式？大小是多少？以及如何进行磁盘的读取？

只有在这些问题都得到回答之后，Bootloader才可能正确无误地在磁盘上找到内核并加载进内存。为此我们需要进行一些明确的设定。

**内核位置、大小：**内核在磁盘中的位置如图1.3所示。磁盘扇区编号从0#开始，引导扇区占据0#扇区；内核紧随其后占据从1#扇区开始的连续个扇区，假定内核占据的扇区不越过第199#扇区(就V6++而言预留的空间已经足够)；从200#扇区开始为文件系统。

引导扇区

内核

文件系统

**图 1.3** 内核在磁盘中的位置

**内核的格式：**内核采用纯二进制格式，而不是PE或ELF等可行性格式，从而避免了让Bootloader从PE或ELF头中分析出程序入口地址和段信息的麻烦。

需要简单解释一下的是，内核由许多个源代码文件经过编译、链接最终获得一个可执行格式(PE或ELF)文件，然后将PE或ELF头、符号表、调试信息等从可执行文件中去除，仅提取并保留内核的代码段、数据段、bss段等用于执行的必要数据，这才得到我们需要的纯二进制格式的内核。关于如何提取出纯二进制格式的内核会在编译、链接内核的章节中介绍，不是本节的必要基础。

立足于之前版本已经进入保护模式的基础上，我们将通过以下的代码来完成从磁盘加载内核到1M物理内存处的目标。

**代码1.7 \_load\_kernel**的代码from boot.s

mov ecx, KERNEL\_SIZE ;cx = 扇区数KERNEL\_SIZE，作为loop的次数

mov eax, 1 ;LBA寻址模式下sector编号从0开始。 #0是引导扇区，#1扇区开始才是kernel的首扇区

mov ebx, 0x100000 ;目标存放地址从1M处开始，每次loop递增512 bytes

**\_load\_kernel:**

push eax

push ebx

add ebx, 512

call **\_load\_sector**

inc eax

loop \_load\_kernel

jmp 0x100000 ;0x7c3b

**代码1.8** \_load\_sector的代码 from boot.s

**\_load\_sector:**

push ebp

mov ebp,esp

push edx

push ecx

push edi

push eax

mov al,1 ;读1个扇区

mov dx,1f2h ;扇区数寄存器 0x1f2

out dx,al

mov eax,[ebp+12] ;[ebp+12]对应上面mov eax, 1 push eax指令入栈的值，eax为要读入的扇区号

;LBA28(Linear Block Addressing)模式输入扇区号的Bits 7~0， 共28 Bits扇区号

inc dx ;扇区号寄存器 0x1f3

out dx,al

shr eax,8 ;LBA28(Linear Block Addressing)模式输入扇区号的Bits 15~8 放入AL中， 共28 Bits扇区号

inc dx ;Port：DX = 0x1f3+1 = 0x1f4

out dx,al

shr eax,8 ;LBA28(Linear Block Addressing)模式输入扇区号的Bits 23~16放入AL中， 共28 Bits扇区号

inc dx ;Port：DX = 0x1f4+1 = 0x1f5

out dx,al

shr eax,8

and al,0x0f

or al,11100000b ;Bit(7和5)为1表示是IDE接口，Bit(6)为1表示开启LBA28模式，Bit(4)为1表示主盘。

;Bit(3~0)为LBA28中的Bit27~24位

inc dx ;Port：DX = 0x1f5+1 = 0x1f6

out dx,al

mov al,0x20 ;0x20表示读1个sector，0x30表示写1个sector

inc dx ;Port：DX = 0x1f6+1 = 0x1f7

out dx,al

.test:

in al,dx

test al,10000000b

jnz .test

test al,00001000b

jz .load\_error

mov ecx,512/4

mov dx,0x1f0

mov edi,[ebp+8] ;取得call前入栈参数[ebp+8] = 0x100000 = 1MB

rep insd

xor ax,ax

jmp .load\_exit

.load\_error:

mov dx,0x1f1

in al,dx

xor ah,ah

.load\_exit:

pop eax ;对应上面的PUSH EAX，这里也加一句 :-)

pop edi

pop ecx

pop edx

leave ;Destory stack frame 等价于2条指令: MOV EBP, ESP 和 POP EBP

retn 8 ;retn 恒等于 ret，我在后面加立即数8，是使ESP = ESP + 8，单位为byte

;销毁调用\_load\_sector前,2次PUSH eax压入栈的参数--读取扇区号& 目的物理地址

**编译：**对boot.s其进行编译得到boot.bin，大小恰好为512字节。编译命令如下：

**nasm -f bin boot.s -o boot.bin**

nasm编译器在生成目标二进制时，会自动根据boot.s中汇编指令占据的字节数量，向上取整512的倍数来确定目标文件的大小，这说明新加入了加载内核的逻辑之后，我们的Bootloader仍旧落在512字节范围内，没有超出引导扇区的512字节大小限制。

**伪内核：**为检验Bootloader加载内核是否正确，我们需要有一个充当“内核”的文件。为此我们提供了一个kernel.bin文件，至于kernel.bin是如何得到的，后面会进行专门的解释。目前只需要知道kernel.bin是一个12KB大小的“内核”文件，我们通过观察Bootloader是否正确地将kerne.bin的内容读入了1M物理内存处来检验其正确性。

**部署：**根据图1.1中的布局要求，将boot.bin和kernel.bin这两部分拷贝到一个用做虚拟磁盘的文件C.IMG中。boot.bin拷贝到C.IMG的0#扇区中，而kernel.bin拷贝到C.IMG中从1#扇区开始往后的空间中。部署过程的命令如下：

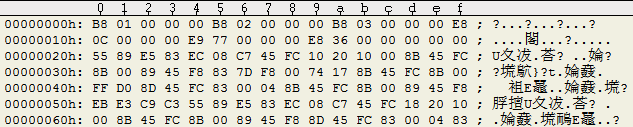
**partcopy boot.bin 0 200 c.img 0**

**partcopy kernel.bin 0 3000 c.img 200**

**运行：**启动Bochs的调试器，以bochsrc.bxrc为配置文件。在1M内存处设置断点。这样做是因为bootloader在执行完\_load\_kernel循环之后，紧接着就通过**jmp 0x100000**跳转到1M内存处继续执行下去，所以当设置在1M内存处的断点生效之时，意味着bootloader已完成了内核的加载。

**bochsdbg -q -f bochsrc.bxrc**

检验Bootloader加载内核是否正确，可以通过比对kernel.bin的数据与Bochs中1M内存处的数据是否一致。运行结果表明，1M内存处的读入数据与十六进制编辑器中kernel.bin数据完全一致，内核载入成功。至此，我们已经完成了一个具备内核加载能力的Bootloader(截图中仅仅比较内核映像的部分字节，但足以表明程序的正确性了)。



**图 1.4** 十六进制编辑器中的kernel.bin数据

(0) [0xfffffff0] f000:fff0 (unk. ctxt): jmp far f000:e05b ; ea5be000f0

**<bochs:1> lb 0x100000 ;1M内存处设置断点**

**<bochs:2> c ;继续运行**

(0) Breakpoint 1, 0x0000000000100000 in ?? ()

Next at t=17856996

(0) [0x00100000] 0008:0000000000100000 (unk. ctxt): mov eax, 0x00000001 ;

b801000000

**<bochs:3> xp /80bm 0x100000 ;检查1M内存处的数据**

[bochs]:

0x0000000000100000:**B8 01 00 00 00 B8 02 00 00 00 B8 03 00 00 00 E8**

0x0000000000100010:**0C 00 00 00 E9 77 00 00 00 E8 36 00 00 00 00 00**

0x0000000000100020:**55 89 E5 83 EC 08 C7 45 FC 10 20 10 00 8B 45 FC**

0x0000000000100030:**8B 00 89 45 F8 83 7D F8 00 74 17 8B 45 FC 8B 00**

0x0000000000100040:**FF D0 8D 45 FC 83 00 04 8B 45 FC 8B 00 89 45 F8**

**图1.5** 运行Bochs查看内存数据

### \_load\_kernel函数

由给出的代码不难发现，\_load\_kernel函数(代码1.7)完成了整个内核加载(暂且称之为函数，确切说应该只是一个标号)。

之前已经明确设定内核位置和大小，我们编写\_load\_kernel使之从1#扇区开始读取磁盘，直到第199#扇区结束。因此我们定义常量KERNEL\_SIZE = 199，它决定了循环执行\_load\_kernel的次数，每一次循环调用\_load\_sector函数读取一个扇区的数据。\_load\_sector函数进行真正的磁盘扇区的读取，而\_load\_kernel 并不涉及磁盘I/O操作，后面会单独对\_load\_sector函数进行分析。

在\_load\_kernel每一次循环调用\_load\_sector时，需要为\_load\_sector准备两个参数：当前读取扇区号和内核加载的目标内存地址。eax寄存器中存放当前要读取扇区号，从1#扇区开始，每次loop递增1；内核加载目标地址存于ebx寄存器中，初始从1M内存处开始，每次loop时加载目标地址递增512字节，使得下一磁盘扇区的数据读入到后继512字节内存中。\_load\_kernel函数循环执行KERNEL\_SIZE次，完成加载内核至1M内存的工作。

整个\_load\_kernel函数的过程如果转换为等价的伪代码，基本上会是这样(代码1.9)：

void \_load\_kernel()

{

int sector = 1;

int address = 0x100000; //加载到1M目标内存地址

for(int i = 0; i < KERNEL\_SIZE; i++)

{

\_load\_sector(address, sector); //调用\_load\_sector读取一个扇区数据

address += 512;

sector += 1;

}

}

定义常量KERNEL\_SIZE为内核可能占用的最大扇区数目是为了方便，该数值只要大于内核实际占用扇区数都可正确运行。刚开始的时候，内核规模可能很小，只有二、三十个扇区不到，KERNEL\_SIZE取值过大会读入多余的空扇区，不会导致任何副作用，但是这至少避免了我们每次内核大小增长时都要去修改读取磁盘扇区的数目。

### \_load\_sector函数

\_load\_sector函数完成从磁盘读取单个扇区到目标内存地址的功能。值得注意的是，整个加载内核过程是在保护模式下进行的，这就存在两个问题：

问题1：保护模式下，不能借助如实模式中INT13 BIOS中断，将如何读取磁盘数据？

问题2：当磁盘读取完数据时会以发送中断方式通知CPU，关中断条件下如何得知已经读取到数据？

在进入保护模式之前必须要屏蔽中断(注意跳入保护模式前有cli关中断指令)，这样做是因为保护模式下还没有准备好响应外设中断的处理函数，假如此时有中断进入系统将会崩溃，只有将各种中断对于的处理函数设置完成后才可以开中断，而在\_load\_kernel整个执行过程中始终是关中断的。**因此\_load\_sector函数要实现的是：不借助INT13 BIOS中断以及关中断条件下读取一个扇区的数据。**

那么\_load\_sector函数究竟是如何实现这一目标的呢？简单来说，就是通过对磁盘控制器发送控制命令来读取指定扇区中的数据，以此来替代只有在实模式下才有的INT13 BIOS中断。

首先，需要明确的是CPU，磁盘控制器和驱动器三者之间的关系(如图1.6)。CPU向控制器发送命令；磁盘控制器根据接收到的命令指挥驱动器执行相应操作，譬如读取驱动器某一扇区的数据；操作完成之后磁盘控制器会向CPU发送中断。



**图1.6** 磁盘控制器与CPU、驱动器关系

在这整个过程中，需要我们程序来做的就是向磁盘控制器发送控制命令，然后处理完成操作后控制器的中断请求­(如果有中断能进来的话)。至于磁盘控制器如何指挥驱动器寻找到并读写扇区数据则不归我们负责。☺

一般来说，对磁盘控制器编程来实现数据在磁盘与内存之间传送，主要有以下三种方式：

1. **Polling(轮询)：**CPU向磁盘控制器发送读/写命令，启动一次I/O操作。由于磁盘这类物理设备读写操作需要一定时间，而CPU并不能知道本次操作需要多少时间，于是在本次I/O操作完成之前唯一可以做的就是周期性地询问磁盘控制器“操作是否已经完成”；对于读操作，也就是是否已经完成读取扇区操作，数据已在缓冲区中。
2. **Interrupt-driven I/O(中断方式I/O)：**CPU向磁盘控制器发送读/写命令启动一次I/O操作，然后CPU可以继续执行后续指令，而并不需要轮询磁盘控制器检查操作是否完成。等到I/O操作完成之后，磁盘控制器会向CPU发出中断告知本次I/O操作完成，数据已经在缓冲区中。
3. **DMA(Direct Memory Access, 直接内存访问)：与中断方式I/O类似，由CPU向磁盘控制器和DMA控制器发送命令启动一次I/O操作，较中断方式I/O更先进的地方在于，整个数据传送不需要CPU介入，基本不占用CPU周期；而是由DMA控制器操纵直接在磁盘与内存之间进行**成块**数据传送。CPU在此过程中可以执行后续指令，直至本次I/O操作完成后会向CPU发送中断信号，此时磁盘数据已经在内存中。**

**由上面可以看出，中断方式I/O和DMA需要在中断打开的条件下使用，这两种方式都会用发送中断信号方式通知I/O操作结束；而正如上述第二个问题中提到的，关中断条件下如何能得知读取数据完成呢？答案只能是采取Polling方式进行读取工作。**

**另外需特别注意，除了DMA方式中CPU收到中断表示数据已经读入到内存中(对读操作而言)，Polling以及Interrupt-driven I/O中读取完成并不表示数据已经在内存中，仅仅意味数据已经从扇区中读入到磁盘控制器缓冲区中，我们还需要将它从缓冲区读入内存，这一环节在\_load\_sector函数中有体现。**

### ATA磁盘控制寄存器接口

到目前为止，我们仅仅用一句“**CPU向磁盘控制器发送命令启动一次I/O操作**”表述来概括对磁盘控制器的编程操作，但具体发送什么命令？向哪里发送命令才可以让磁盘控制器接收到呢？这些便是本节要解释的关键问题。

首先，磁盘控制器是什么？从硬件角度来说，当然是一块芯片；而从软件角度来说，磁盘控制器可以被认为是由一系列寄存器组成的，包括：状态寄存器、命令寄存器、数据寄存器、扇区号寄存器等，还加上用于临时存储数据的控制器缓冲区。我们通过对寄存器写入恰当的数值来发送I/O操作命令，告诉控制器要读写的扇区号是多少，操作方式为读或写，并通过读取状态寄存器的值了解操作是否成功完成等，所以说将磁盘控制器抽象为一组寄存器是比较恰当的。

其次，如何访问这些不同的寄存器？每个寄存器都有其特定的端口地址，我们通过IN/OUT指令访问指定的端口地址就可以访问到这些寄存器，对其写入或读取数据。至于每个寄存器具体的端口地址则是系统体系结构规范确定的。

下面表1.2中给出了ATA磁盘控制寄存器的端口地址和用途。表中有两个寄存器共用一个端口地址的情况：在读操作时，访问一个寄存器；而在写操作时，访问另一个寄存器。这样安排是为了节省I／O地址资源。

**表1.2 ATA磁盘控制寄存器端口及用途**

|  |  |  |
| --- | --- | --- |
| 端口 | 读操作 | 写操作 |
| 0x1f0 | 数据寄存器 | |
| 0x1f1 | 错误寄存器(错误状态) | 写前预补偿寄存器 |
| 0x1f2 | 扇区数寄存器 | |
| 0x1f3 | 扇区号寄存器 | |
| 0x1f4 | 柱面号寄存器(低位) | |
| 0x1f5 | 柱面号寄存器(高位) | |
| 0x1f6 | 驱动器/磁头寄存器——驱动器号/磁头号 | |
| 0x1f7 | 主状态寄存器 | 命令寄存器 |
| 0x3f6 | 辅助状态寄存器 | 磁盘控制寄存器 |
| 0x3f7 | 驱动器地址寄存器 | ---- |

这里对表1.2中给出的各个寄存器进行说明：

1. **数据寄存器(0x1f0，读/写)**

这是一对16位数据传输器，用于扇区读、写和磁道格式化操作。该寄存器用于在内存和控制器缓冲区之间进行32位数据交换，即每次4字节。在使用polling和中断方式I/O传输时要使用该寄存器。当要进行一个扇区512直接的数据读取时，应当使用rep insd指令将控制器缓冲区中的数据经由数据寄存器(0x1f0)读入内存。

1. **错误寄存器(0x1f1，读)/写前预补偿寄存器(0x1f1，写)**

这是一个8位的寄存器。读操作时，它反映控制器在诊断方式(在加 电、复位或执行驱动器诊断命令时所处的方式)或操作方式(执行除诊断命令以外的所有命令时所处的方式)下的错误原因。例如，在操作方式下，该寄存器的最高位为1表示访问的扇区是坏扇区，最低位为1表示地址标志没有找到。只有当主状态寄存器(0x1f7)的位0等于1时该寄存器的数据才有效指示错误状态。

在写操作时，作为写前预补偿寄存器，该寄存器的内容等于柱面号除以4，表示开始写预补偿的起始柱面号。现在的磁盘基本已不使用该寄存器。

#### 表1.3 错误寄存器状态含义

|  |  |  |
| --- | --- | --- |
| 值 | 诊断命令时 | 其它命令时 |
| 0x01 | 无错误 | 数据标志丢失 |
| 0x02 | 控制器出错 | 磁道0错 |
| 0x03 | 扇区缓冲区错 | ---- |
| 0x04 | ECC部件错 | 命令放弃 |
| 0x05 | 控制处理器错 |  |
| 0x10 | ---- | ID未找到 |
| 0x40 | ---- | ECC错误 |
| 0x80- | ---- | 坏扇区 |

1. **扇区数寄存器(0x1f2，读/写)**

该寄存器记录要访问的扇区的数目。用于多扇区操作时，每访问一个扇区，其值自动减1。如果初值为0，则表示扇区数是256。

LBA(Logical Block Addressing,逻辑块寻址)采用从0开始的数值对磁盘扇区进行编号，允许程序使用绝对的扇区号访问磁盘，而不再是原先依据磁盘物理结构的“磁柱-磁头-扇区(CHS)”寻址模式。在访问磁盘时，由 IDE 控制器把逻辑扇区号转换为实际磁盘对应的柱面、磁头、扇区等参数。

1. **扇区号寄存器(0x1f3，读/写)**

该寄存器记录要访问的起始扇区号。用于多扇区操作时，每完成一个扇区读写操作寄存器的值自动加1。如果使用LBA28寻址(逻辑块寻址)方式，它记录起始逻辑扇区号的第7～0位。

1. **柱面号寄存器(0x1f4，0x1f5，读/写)**

这是由两个8位寄存器组成的16位寄存器，0x1f4为低字节，0x1f5为高字节。它用来记录要操作的柱面的编号。如果使用LBA28寻址方式，这两个寄存器分别存放起始逻辑扇区号的第15～8位和第23～16位。

1. **驱动器／磁头寄存器(0x1f6，读/写)**

该寄存器记录要访问的驱动器号、磁头号以及寻址方式。对于IDE接口的磁盘而言，其中第7位和第5位必须置1，而第6位置1时开启LBA模式，否则就用“磁柱-磁头-扇区(CHS)”寻址方式。第4位为驱动器选择，指定此次访问的对象是主盘还是从盘，置0时访问的主盘，置1时访问的从盘。当采用LBA模式时第3位~第0位表示LBA28位中的第24~27位。

1. **主状态寄存器(0x1f7，读)/命令寄存器(0x1f7，写)**

在读时，该端口地址对应主状态寄存器，反映磁盘控制器在执行命令后的状态。各状态位含义如表1.4所示。

#### 表1.4 主状态寄存器各位含义

|  |  |
| --- | --- |
| 值 | 说明 |
| 0x01 | 命令执行错误 |
| 0x02 | 收到索引 |
| 0x04 | ECC校验错误 |
| 0x08 | 请求服务，表示可传输数据 |
| 0x10 | 寻道结束 |
| 0x20 | 驱动器故障 |
| 0x40 | 驱动器准备好 |
| 0x80- | 控制器忙碌 |

在写时，该端口地址对应命令寄存器，该寄存器接收并保存CPU发送到控制器的命令。命令一旦发送到命令寄存器中，该命令就会开始执行。

#### 表1.5 命令寄存器命令值

|  |  |  |  |
| --- | --- | --- | --- |
| 命令码 | | 命令默认值 | 命令说明 |
| 高4位 | D3 D2 D1 D0 |
| 0x1 | R R R R | 0x10 | 驱动器重新校正 |
| 0x2 | 0 0 L T | 0x20 | 读操作 |
| 0x3 | 0 0 L T | 0x30 | 写操作 |
| 0x4 | 0 0 0 T | 0x40 | 扇区检验 |
| 0x5 | 0 0 0 0 | 0x50 | 格式化磁道 |
| 0x6 | 0 0 0 0 | 0x60 | 控制器初始化 |
| 0x7 | R R R R | 0x70 | 寻道 |
| 0x8 | 0 0 0 0 | 0x80 | 控制器诊断 |
| 0x9 | 0 0 0 1 | 0x90 | 建立驱动器参数 |

在表1.5中命令码的低4位是附加参数：

R是步进速率。R=0，则步进速率35us；R=1为0.5ms，以此量递增。L是数据模式。L=0表示读/写扇区为512B；L=1表示读/写为512B加4B的ECC码。T是重试模式。T=0表示允许重试；T=1则禁止重试。

然而实际上大多数情况下我们只需要使用默认的命令值即可。即对于读操作命令值为0x20，写操作对应命令值0x30。

1. **辅助状态寄存器(0x3f6，读)/ 磁盘控制寄存器(0x3f6，写) (未使用)**

在读时，它与状态寄存器的内容相同，但读该寄存器时不会清除中断请求信号。在写时，用于存放磁盘控制字节并控制复位操作。

1. **驱动器地址寄存器(0x3f7，读) (未使用)**

在ATA-1标准中该寄存器用来保存命令执行后的某些信息。在ATA-2标准中没有定义该寄存器。

### 对ATA磁盘控制器编程

在上面介绍了ATA磁盘控器的寄存器接口之后，我们就可以根据硬件规范中给出的命令参数，对控制器发出适当的命令，指挥其进行需要的读/写操作了。

对磁盘控制器进行编程时，需要发送参数和命令两部分信息。参数部分包括：要读/写的起始扇区号、扇区个数、操作对象是主盘还是从盘；命令部分需要指定当前I/O操作方式是读或写。具体的流程如下：

1. 设置要读取的扇区数为1，每次读入一个扇区；
2. 使用LBA28寻址方式，分别将扇区号的Bits 7~0写入端口为0x1f3的寄存器，扇区号的Bits 15~8写入端口为0x1f4的寄存器，扇区号的Bits 23~16写入端口为0x1f5的寄存器，扇区号的Bits 27~24写入端口为0x1f6寄存器的低4比特；
3. 设置驱动器/磁头寄存器(0x1f6)的高4比特，Bit(7和5)为1表示是IDE接口，Bit(6)为1表示开启LBA28模式，Bit(4)为1表示访问主盘；
4. 向命令寄存器输出0x20表示启动一次读操作；
5. 由于采用Polling方式检查I/O操作是否完成，CPU在此期间周期性地检查主状态寄存器(0x1f7)，看数据是否就绪；如果未就绪则忙等待直至就绪；
6. 检查是否出错，如果磁盘数据读取过程发生错误，则立即退出；
7. 若数据已经就绪在控制器缓冲区中，则设置好目标内存地址，使用rep insd指令将控制器缓冲区中的数据经由数据寄存器(0x1f0)读入内存。

上述通过控制器编程每次读取1个扇区大小的数据便是\_load\_sector函数(代码1.8)的执行过程，其接受目标内存地址和目标读取扇区号两个参数，能够在关中断情况下读取扇区数据，配合\_load\_kernel函数就实现了加载磁盘中占据连续多个扇区的内核映像到内存中的任务。