



福昕高级PDF编辑器

高效 · 安全 · 专业

立即下载

点击购买



OFFICE格式互转



OCR文字识别



文本图像编辑



加密和签署



交互式动态表单



互联PDF文档

BOOT.ASM

运行一个操作系统需要：

- 计算机（虚拟机bochs）

安装bochs虚拟机： `sudo apt-get install bochs`

安装bochs的GUI库： `sudo apt-get install bochs-sdl`

- 硬盘（装有操作系统）

创建虚拟软盘映像： `bximage -> fd -> 1.44 -> a.img`

fd: 软盘映像

1.44: 映像大小 1.44MB

a.img: 映像名称（在往映像中写入操作系统时以及配置计算机时会用到）

开始我们自己的操作系统

boot.asm

	7	6	5	4	3	2	1	0
含义	<u>BL</u>	<u>R</u>	<u>G</u>	<u>B</u>	<u>I</u>	<u>R</u>	<u>G</u>	<u>B</u>
	闪烁	背景		高亮	前景			

```
org 07c00h          ; 告诉编译器程序加载到7c00处 org程序起始地址
mov ax, cs          把当前段地址放到ax
mov ds, ax          在放到段寄存器ds、es
mov es, ax
call DispStr        ; 调用显示字符串例程 调用子程序
jmp $               ; 无限循环 $:当前行地址,所以无线循环,程序不断,
                    ; 窗口一直处于打开状态
DispStr:
mov ax, BootMessage 加载数据到ax:一次2B
mov bp, ax           ; ES:BP = 串地址 显示字符串的地址
mov cx, 16           ; CX = 串长度 字符串长度
mov ax, 01301h       ; AH = 13, AL = 01h AH=13H入口参数 AL:输出方式
mov bx, 000ch        ; 页号为0(BH = 0) 黑底红字(BL = 0Ch,高亮)
mov dl, 0            BH =页号 BL=属性=00001010BG=000黑色 高亮=1 前景=010红色
int 10h              ; 10h 号中断 对屏幕进行操作 对寄存器配置即可自动显示
ret
BootMessage:         db "Hello, OS world!" 定义字节型字符串数据
times 510-($-$$)     db 0 ; 填充剩下的空间,使生成的二进制代码恰好为512
dw 0xaa55            ; 结束标志
```

AL=01H:字符串中只含显示字符,其显示属性在BL中。显示后,光标位置改变

Org 07c00h

- 为什么要这么一行代码？
 - Bios将软盘内容放在07c00h位置，并不是由这行代码决定的。那这行代码的作用是什么？
- Org是伪指令。
 - 伪指令是指，不生成对应的二进制指令，只是汇编器使用的。也就是说，boot.bin文件里面，压根没有07c00h这个东西，bios不是因为这条指令才把代码放在07c00h
 - Mov这种指令，就会生成二进制代码，可以直接告诉cpu该做什么。
- 那么org 07c00h到底是做什么用的？

Org 07c00h

- 告诉汇编器，当前这段代码会放在07c00h处。所以，如果之后遇到需要绝对寻址地指令，那么绝对地址就是07c00h加上相对地址。
 - 绝对地址：内存的实际位置（先不考虑内存分页一类逻辑地址）
 - 相对地址：当前指令相对第一行代码的位置。
-
- 在第一行加上org 07c00h只是让编译器从相对地址07c00h处开始编译第一条指令，相对地址被编译加载后就正好和绝对地址吻合

boot.bin

使用nasm来汇编boot.asm生成“操作系统”(boot.bin)的二进制的代码

- 首先安装nasm工具:
`sudo apt-get install nasm`
- 生成boot.bin:
`nasm boot.asm -o boot.bin`
(boot.o->boot.bin)

写入boot.bin

- 有了“计算机”，也有了“软盘”，以及“操作系统”（boot.bin），是时候将boot.bin写入“软盘”了

• 我们需要把boot.bin放在软盘的第一个扇区，为什么？

- 需要理解BIOS与OS的加载

BIOS

- 开机，从ROM运行BIOS程序，BIOS是厂家写好的。
- BIOS程序检查软盘0面0磁道1扇区，如果扇区以0xaa55结束，则认定为引导扇区，将其512字节的数据加载到内存的07c00处，然后设置PC，跳到内存07c00处开始执行代码。
- 以上的0xaa55以及07c00都是一种约定，BIOS程序就是这样做的，所以我们就需要把我们的OS放在软盘的第一个扇区，填充，并在最末尾写入0xaa55

```
17 times 510-($-$) db 0 ; 填充剩下的空间，使生成的二进制代码恰好为512字节
18 dw 0xaa55 ; 结束标志
```

写入boot.bin

- `nasm -f bin boot.asm`生成的其实是一个512字节的二进制文件，如何把它放在软盘的第一个扇区？
- 直接拷贝是不可以的（普通的读写操作（`mv`, `rm`, `cp`）是基于文件系统的，文件系统是一个逻辑概念。引导扇区，是磁盘第一个磁道的第一个扇区，他是一个物理概念，在文件系统中，这个扇区是不可见的。）
- 用特殊的命令：
 - `dd if=boot.bin of=a.img bs=512 count=1 conv=notrunc`
 - （`if`代表输入文件，`of`代表输出设备，`bs`代表一个扇区大小，`count`代表扇区数，`conv`代表不作其它处理）
- `a.img`是软盘？
 - 虚拟软盘
 - 使用：`bximage -> fd -> 1.44 -> a.img`这条命令就是为了生成一个1.44m大小的虚拟磁盘软盘文件。

继续BIOS

- 将其512字节的数据加载到内存的07c00处，然后设置PC，跳到内存07c00处开始执行代码。
 - 为什么是0700c处？才说了是bios厂家约定好的。当然，其实不只是bios厂家，这涉及整个计算机的设计和架构地一种约定，进一步学习这门课程后你就能理解为什么是07c00而不是07c000了。

理解Org 07c00h 后

- 可以去掉org 07c00h吗?
- 让我们来试试

```
代码 (boot1.asm)  mov ax,cs
                   mov ds,ax
                   mov es,ax
                   call DispStr
                   jmp $
DispStr:
  mov ax,BootMessage
  mov bp,ax
  mov cx,16
  mov ax,01301h
  mov bx,000ch
  mov dl,0
  int 10h
  ret
BootMessage:  db "Hello, OS world!"
times 510-($-$$) db 0
dw 0xaa55
```

编译得到 boot1.bin

反编译boot1.bin

第一条指令将从0地址开始编译：

反编译： ndisasm boot2.bin -o 0

查看2.txt:

00000000	8CC8	mov ax,cs	0000000B B81E00	mov ax,0x1e
00000002	8ED8	mov ds,ax	取字符串，由之前所说，第一行可执行代码	
00000004	8EC0	mov es,ax	加载到内存的0x7c00处，故字符串地址应该	
00000006	E80200	call word 0xb	为0x7c00+0x1e	
00000009	EBFE	jmp short 0x9		
0000000B	B81E00	mov ax,0x1e		
0000000E	89C5	mov bp,ax		
00000010	B91000	mov cx,0x10		
00000013	B80113	mov ax,0x1301		
00000016	BB0C00	mov bx,0xc		
00000019	B200	mov dl,0x0		
0000001B	CD10	int 0x10		
0000001D	C3	ret		
0000001E	48	dec ax		

修改boot1.asm

```
mov ax,cs
mov ds,ax
mov es,ax
call DispStr
jmp $
DispStr:
mov ax,BootMessage+07c00h
mov bp,ax
mov cx,16
mov ax,01301h
mov bx,000ch
mov dl,0
int 10h
ret
BootMessage: db "Hello, OS world!"
times 510-($-$$) db 0
dw 0xaa55
```

反编译:

```
000 8CC8
00000002 8ED8
00000004 8EC0
00000006 E80200
00000009 EBFE
0000000B B81E7C
0000000E 89C5
00000010 B91000
00000013 B80113
00000016 BB0C00
00000019 B200
0000001B CD10
0000001D C3
```

```
mov ax,cs
mov ds,ax
mov es,ax
call word 0xb
jmp short 0x9
mov ax,0x7c1e
mov bp,ax
mov cx,0x10
mov ax,0x1301
mov bx,0xc
mov dl,0x0
int 0x10
ret
```

正确

启动？

- 现在准备就绪，可以启动了吗？
- 还需要一样重要的东西，那就是Bochs的配置文件。
- 为什么要配置有配置文件？因为你要告诉Bochs，你希望你的虚拟机是什么样子的，比如内存多大啊，硬盘映像和软盘映像都是哪些文件啊等内容。

启动!

- 配置文件bochsrc:

```
megs:32  
display_library: sdl  
floppya: 1_44=a.img, status=inserted  
boot: floppy
```

display_library: bochs使用的GUI库, 在Ubuntu下面是sdl

megs: 虚拟机内存大小 (MB)

floppya: 虚拟机外设, 软盘为a.img文件

boot: 虚拟机启动方式, 从软盘启动

配置文件保存为bochsrc, 和a.img以及boot.bin放在同一目录下

- 启动bochs:

bochs -f bochsrc

OS的加载

- 首先回忆一下我们在BIOS中提到的“操作系统”的加载过程：

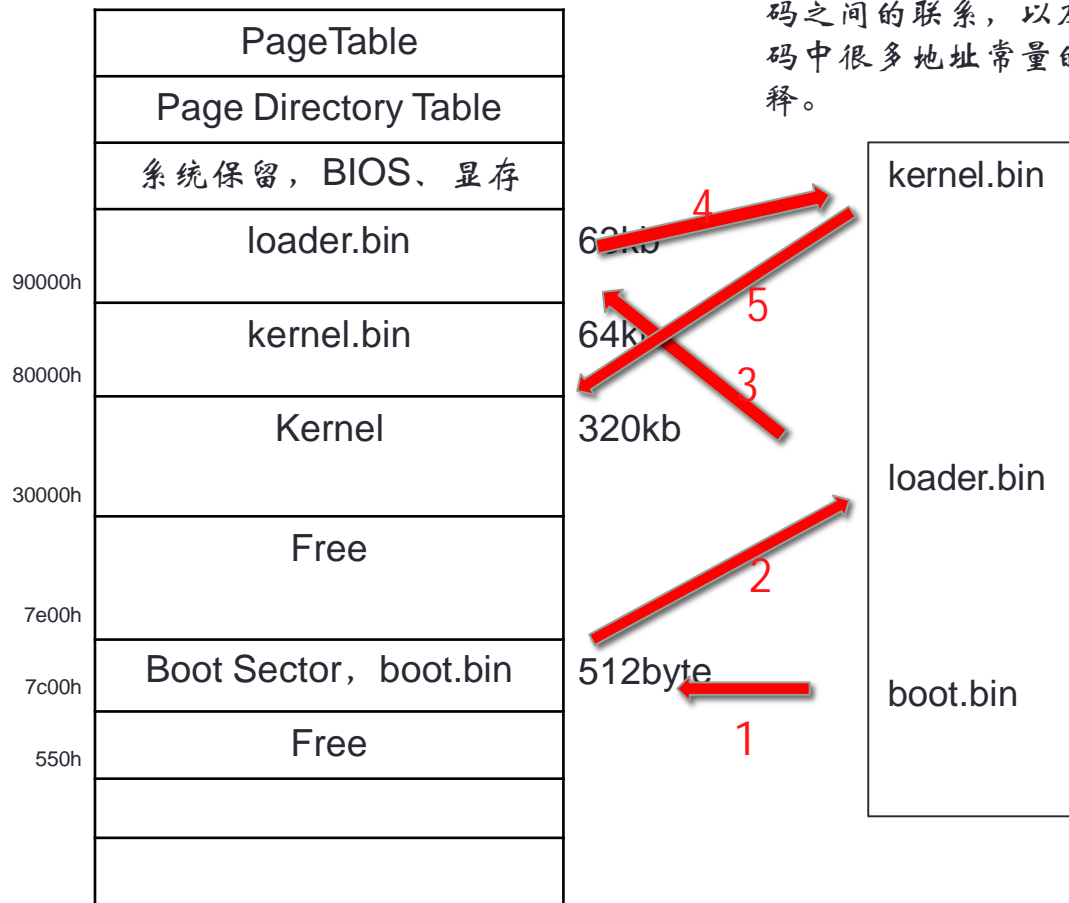
BIOS程序检查软盘0面0磁道1扇区，如果扇区以0xaa55结束，则认定为引导扇区，将其512字节的数据加载到内存的07c00处，然后设置PC，跳到内存07c00处开始执行代码。

- 其实我们所提到的只是真正的操作系统内核加载过程中的第一步，真正的加载过程大概是这样：“引导—加载内核入内存—跳入保护模式—开始执行内核”，那是不是只要我们将加载内核入内存的代码写入我们之前写的boot.asm中就行了呢？

突破512字节的限制

- 其实，除了加载内核，我们要做的事还有准备保护模式（等讲到保护模式后再来理解）等，512字节显然不够，为了突破512字节的限制，我们引入另外一个重要的文件，loader.asm,引导扇区只负责把loader加载入内存并把控制权交给他，这样将会灵活得多。
- 最终，由loader将内核kernel加载入内存，才开始了真正操作系统内核的运行。

启动流程和内存分布



这张图对于理解代码有着非常重要的作用。包括boot, loader, kernel代码之间的联系, 以及代码中很多地址常量的解释。

loader

- 跳入保护模式
 - 最开始的x86处理器16位，寄存器用ax, bx等表示，称为实模式。后来扩充成32位，eax, ebx等，为了向前兼容，提出了保护模式
 - 必须从实模式跳转到保护模式，才能访问1M以上的内存。
- 启动内存分页
- 从kernel.bin中读取内核，并放入内存，然后跳转到内核所在的开始地址，运行内核
 - 跟boot类似，使用汇编直接在软盘下搜索kernel.bin
 - 但是，不能把整个kernel.bin放在内存，而是要以ELF文件的格式读取并提取代码。
 - 接下来的幻灯片继续解释

kernel

- 这才是真正的操作系统
- 内存管理，进程调度，图像显示，网络访问等等，都是内核的功能。
- 内核的开发使用高级语言
 - C语言可以更高效的编写内核
 - 但是我们是在操作系统层面上编写另一个操作系统，于是生成的内核可执行文件是和当前操作系统平台相关的。比如linux下是elf格式，有许多无关信息，于是，内核并不能像boot.bin或loader.bin那样直接放入内存中，需要loader从kernel.bin中提取出需要放入内存中的部分。

总结

环境搭建-Linux

- 安装GCC(系统已经自带, 如果没有, `sudo apt-get install gcc`)
- 安装NASM(`sudo apt-get install nasm`)
- 安装虚拟机Bochs(`sudo apt-get install bochs`)
- 安装bochs的GUI库bochs-sdl (`sudo apt-get install bochs-sdl`)
- 如果出现问题, 可以将软件源改成163源, 参考: [Ubuntu 163 源](#)

环境搭建-windows

- 安装虚拟机VirtualBox（或VMWare）
- 在虚拟机中安装Linux(Ubuntu或其它发行版本)
- 接下和Linux中搭建环境一样

环境搭建-其它说明

- 与《Orange'S》不一样，彻底摒弃Windows环境。
- 由于Bochs是简单的虚拟机而且只用来运行我们自己写的简单代码，因此即使在VirtualBox虚拟机中安装Linux再使用Bochs也不会存在性能问题。

Hello, OS

- Step 1. 使用nasm汇编boot.asm生成“操作系统”的二进制代码。

```
nasm boot.asm -o boot.bin
```

- Step 2. 使用bximage命令生成虚拟软盘.

```
bximage -> fd -> 1.44 -> a.img
```

- Step 3. 使用dd命令将操作系统写入软盘

```
dd if=boot.bin of=a.img bs=512 count=1
```

```
conv=notrunc
```

Hello, OS

- Step 4. 配置bochs。

```
display_library: sdl  
floppya: 1_44=a.img, status=inserted  
boot: floppy
```

display_library: bochs使用的GUI库，在Ubuntu下面是sdl

megs: 虚拟机内存大小

floppya: 虚拟机外设，软盘为a.img文件

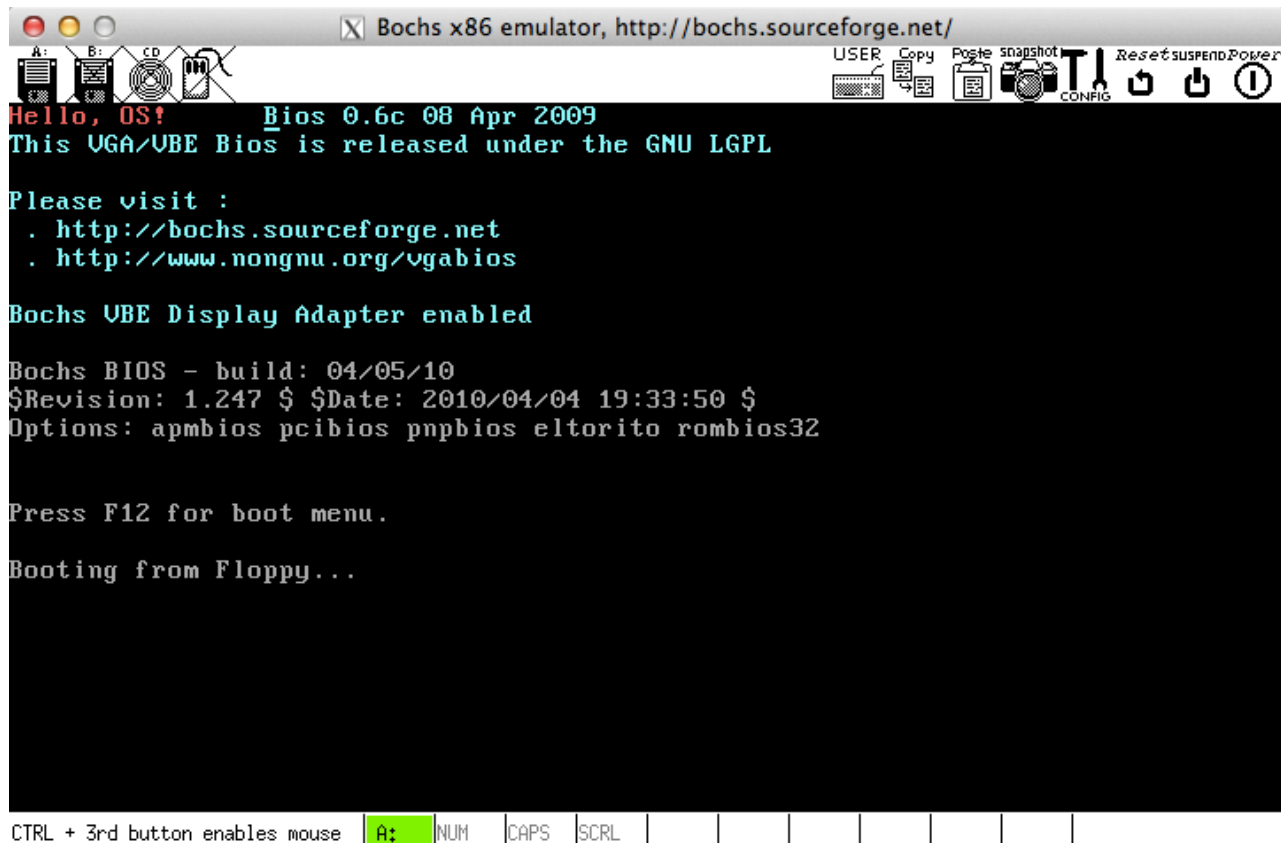
boot: 虚拟机启动方式，从软盘启动

配置文件保存为bochsrc，和a.img以及boot.bin放在同一目录下

- Step 5. 启动bochs.

bochs -f bochsrc

Hello, OS



The screenshot shows a Bochs x86 emulator window. The title bar reads 'Bochs x86 emulator, http://bochs.sourceforge.net/'. The window contains a black terminal area with green and red text. The text reads: 'Hello, OS! Bios 0.6c 08 Apr 2009', 'This UGA/VE BIOS is released under the GNU LGPL', 'Please visit :', '. http://bochs.sourceforge.net', '. http://www.nongnu.org/ugabios', 'Bochs VBE Display Adapter enabled', 'Bochs BIOS - build: 04/05/10', '\$Revision: 1.247 \$ \$Date: 2010/04/04 19:33:50 \$', 'Options: apmbios pcibios pnpbios eltorito rombios32', 'Press F12 for boot menu.', and 'Booting from Floppy...'. The bottom status bar shows 'CTRL + 3rd button enables mouse' and a row of keyboard function keys: A:, NUM, CAPS, SCRL, and several empty slots.

```
Bochs x86 emulator, http://bochs.sourceforge.net/
Hello, OS! Bios 0.6c 08 Apr 2009
This UGA/VE BIOS is released under the GNU LGPL

Please visit :
. http://bochs.sourceforge.net
. http://www.nongnu.org/ugabios

Bochs VBE Display Adapter enabled

Bochs BIOS - build: 04/05/10
$Revision: 1.247 $ $Date: 2010/04/04 19:33:50 $
Options: apmbios pcibios pnpbios eltorito rombios32

Press F12 for boot menu.

Booting from Floppy...
```

CTRL + 3rd button enables mouse A: NUM CAPS SCRL

如果一开始没显示，说明是debug模式，那么只需要按**C!**即可显示！