

# □ 武汉大学国家网络安全学院教学实验报告

课程名称	操作系统设计与实践	实验日期	2024.10.18
实验名称	由盘上结构实现程序加载	实验周次	第五周
姓名	学号	专业	班级
黄东威	2022302181148	信息安全	5 班
王浚杰	2022302181143	网络空间安全	5 班
程序	2022302181131	信息安全	4 班
周业营	2022302181145	信息安全	5 班

Table 1

## ✳ 一、实验目的及实验内容

(本次实验所涉及并要求掌握的知识；实验内容；必要的原理分析)

i

(实验目的、内容及相关的理论知识)

### 1.1 实验目的

- 如何从软盘读取并加载一个 Loader 程序到操作系统，然后转交系统控制权
- 对应章节：第四章

### 1.2 实验内容

- 1 向软盘镜像文件写入一个你自己任意创建的文件，手工方式在软盘中找到指定的文件，读取其扇区信息，记录你的步骤。

- ② 将指定的可执行文件装入指定内存区，并执行，记录原理与步骤
- ③ 学会使用 xxd 读取二进制信息，通过 1、2 来验证。

## 1.3 原理分析

### 1.3.1 FAT12

详细的概述在 [此处](#)

- 引导扇区和 Loader:

- 引导扇区是位于硬盘上的第一个扇区，通常是 0 号扇区，大小为 512 字节。它的主要功能是存储一段小型的引导程序，用于将控制权交给操作系统或其他引导加载器。

FAT12 文件系统中引导扇区的格式如下：

名称	偏移	长度	内容	Orange'S的值
BS_jmpBoot	0	3	一个短跳转指令	jmp LABEL_START nop
BS_OEMName	3	8	厂商名	'ForrestY'
BPB_BytsPerSec	11	2	每扇区字节数	0x200
BPB_SecPerClus	13	1	每簇扇区数	0x1
BPB_RsvdSecCnt	14	2	Boot记录占用多少扇区	0x1
BPB_NumFATs	16	1	共有多少 FAT 表	0x2
BPB_RootEntCnt	17	2	根目录文件数最大值	0xE0
BPB_TotSec16	19	2	扇区总数	0xB40
BPB_Media	21	1	介质描述符	0xF0
BPB_FATSz16	22	2	每 FAT 扇区数	0x9
BPB_SecPerTrk	24	2	每磁道扇区数	0x12
BPB_NumHeads	26	2	磁头数 (面数)	0x2
BPB_HiddSec	28	4	隐藏扇区数	0
BPB_TotSec32	32	4	如果BPB_TotSec16是0, 由这个值记录扇区数	0
BS_DrvNum	36	1	中断 13 的驱动器号	0
BS_Reserved1	37	1	未使用	0
BS_BootSig	38	1	扩展引导标记 (29h)	0x29
BS_VolID	39	4	卷序列号	0
BS_VolLab	43	11	卷标	'OrangeS0.02'
BS_FileSysType	54	8	文件系统类型	'FAT12'
引导代码及其他	62	448	引导代码、数据及其他填充字符等	引导代码 (剩余空间被0填充)
结束标志	510	2	0xAA55	0xAA55

其中一个很重要的数据结构叫做 BPB(BIOS ParameterBlock)。引导扇区需

要有 BPB 等头信息才能被 DOS 以及 Linux 识别。

- Loader 是指引导加载器，它的功能为：加载内核进入内存、跳入保护模式。这些工作如果全都交给引导扇区来做，512 字节很可能是不够用的，所以这些过程通常交给另外的 Loader 模块来完成。

因此，一个操作系统从开机到开始运行大致经历：引导 → 控制权转移给 Loader → 加载内核入内存 → 跳入保护模式 → 开始执行内核的过程。

- FAT12 文件系统：

FAT12 (File Allocation Table 12) 是由微软开发的早期文件系统之一，最初用于在 1980 年代的个人计算机上管理文件。FAT12 使用 12 位地址来指向磁盘上的簇，因此每个簇可以被表示为 0 到 4095 之间的一个值，主要用于早期的软盘和小型存储设备。

FAT12 文件系统包含四个部分：引导扇区、FAT1、FAT2、根目录区、数据区。

- 引导扇区的格式上文已经介绍。
- FAT1、FAT2 是两个完全相同的 FAT 表，FAT2 是 FAT1 的备份。每个表占用 9 个扇区。FAT 表维护着磁盘上所有簇的使用情况，每个 FAT 项为 12 个字节，称为一个 FATEntry，记录着文件的下一个簇号，当簇号  $\geq 0xFF8$  时为最后一个簇，当簇号为 0xFF7 时为坏簇。第 0、1 个簇通常不使用。
- 根目录区从 19 号扇区开始，由若干个 32 字节长的目录条目组成，最多有 BPB\_RootEntCnt 个。FAT 文件系统的每一个文件和文件夹都被分配到一个目录项，目录项中记录着文件名、大小、文件内容起始地址等数据。条目的格式如下：

名称	偏移	长度	描述
DIR_Name	0	0xB	文件名 8 字节，扩展名 3 字节
DIR_Attr	0xB	1	文件属性
保留位	0xC	10	保留位
DIR_WrtTime	0x16	2	最后一次写入时间
DIR_WrtDate	0x18	2	最后一次写入日期
DIR_FstClus	0x1A	2	此条目对应的开始簇号
DIR_FileSize	0x1C	4	文件大小

- 数据区存放着文件的数据。

### 1.3.2 读取扇区

int 13h: BIOS 中断 **int 13h** 通常用于加载一个文件进入内存。

参数如下:

表4.4 BIOS中断int 13h的用法

中断号	寄存器		作用
13h	ah=00h	dl=驱动器号 (0 表示 A 盘)	复位软驱
	ah=02h	al=要读扇区数	从磁盘将数据读入es:bx指向的缓冲区中
	ch=柱面 (磁道) 号	cl=起始扇区号	
	dh=磁头号	dl=驱动器号 (0 表示 A 盘)	
	es:bx→ 数据缓冲区		

使用 **int 13h** 时需要将线性编号的扇区转化为 **(磁头号, 柱面号, 扇区号)** 的 CHS 寻址方式，对于 1.44MB 的软盘，总共有两面（磁头号 0 和 1），因此磁头号、柱面磁道号和起始扇区号计算方式如下：

$$\begin{array}{c}
 \text{扇区号} \\
 \hline
 \text{每磁道扇区数 (18)} \\
 \end{array}
 \left\{
 \begin{array}{l}
 \text{商 } Q \Rightarrow \left\{ \begin{array}{l} \text{柱面号} = Q \gg 1 \\ \text{磁头号} = Q \& 1 \end{array} \right. \\
 \text{余数 } R \Rightarrow \text{起始扇区号} = R + 1
 \end{array}
 \right.$$

## \* 二、实验环境及实验步骤

**i** (本次实验所使用的器件、仪器设备等的情况；具体实验步骤)

### 2.1 实验环境

- Windows Subsystem for Linux 2 (WSL 2)
- Ubuntu 20.04
- NASM 2.14.02
- Bochs 2.7
- Visual Studio Code (VSCode)

### 2.2 实验步骤

- ① 向软盘镜像文件写入一个你自己任意创建的文件，手工方式在软盘中找到指定的文件，读取其扇区信息，记录你的步骤。
- ② 将指定的可执行文件装入指定内存区，并执行，记录原理与步骤
- ③ 学会使用 xxd 读取二进制信息，通过 1、2 来验证。

## \* 三、实验过程分析

**i** (详细记录实验过程中发生的故障和问题，进行故障分析，说明故障排除的过程及方法。根据具体实验，记录、整理相应的数据表格等)

### 3.1 手工分析扇区信息

#### 3.1.1 创建文件

- 1 使用 `bximage` 创建一个新的虚拟软盘，命名为 `pm.img`，并在 `bochsrc` 中添加 `floppyb: 1_44=pm.img, status=inserted`

```
● (base) rimeheart@Rimeheart:~/OSlab/lab4/osfs04-master$ bximage
=====
          bximage
Disk Image Creation / Conversion / Resize and Commit Tool for Bochs
$Id: bximage.cc 13481 2018-03-30 21:04:04Z vruppert $

=====
1. Create new floppy or hard disk image
2. Convert hard disk image to other format (mode)
3. Resize hard disk image
4. Commit 'undoable' redolog to base image
5. Disk image info

0. Quit

Please choose one [0] 1

Create image

Do you want to create a floppy disk image or a hard disk image?
Please type hd or fd. [hd] fd

Choose the size of floppy disk image to create.
Please type 160k, 180k, 320k, 360k, 720k, 1.2M, 1.44M, 1.68M, 1.72M, or 2.88M.
[1.44M] 1.44M

What should be the name of the image?
[a.img] pm.img

Creating floppy image 'pm.img' with 2880 sectors

The following line should appear in your bochsrc:
floppyb: image="pm.img", status=inserted
```

Figure 1

- 2 分别创建如下文件，第一个是教材示例文件，第二个是自己任意创建的文件，第三个是跨越多个扇区的文件

- `RIVER.TXT` : `riverriverriver`
- `hdw.txt` : `Huang Dongwei, or Hou Duan Wang, which one is the true HDW?`
- `HDWS.TXT` : `Huang Dongwei, or Hou Duan Wang, which one is the true HDW? * 9` (即 9 行句子)

- 3 进入 bochs，使用 `format` 指令对 B 盘进行格式化操作

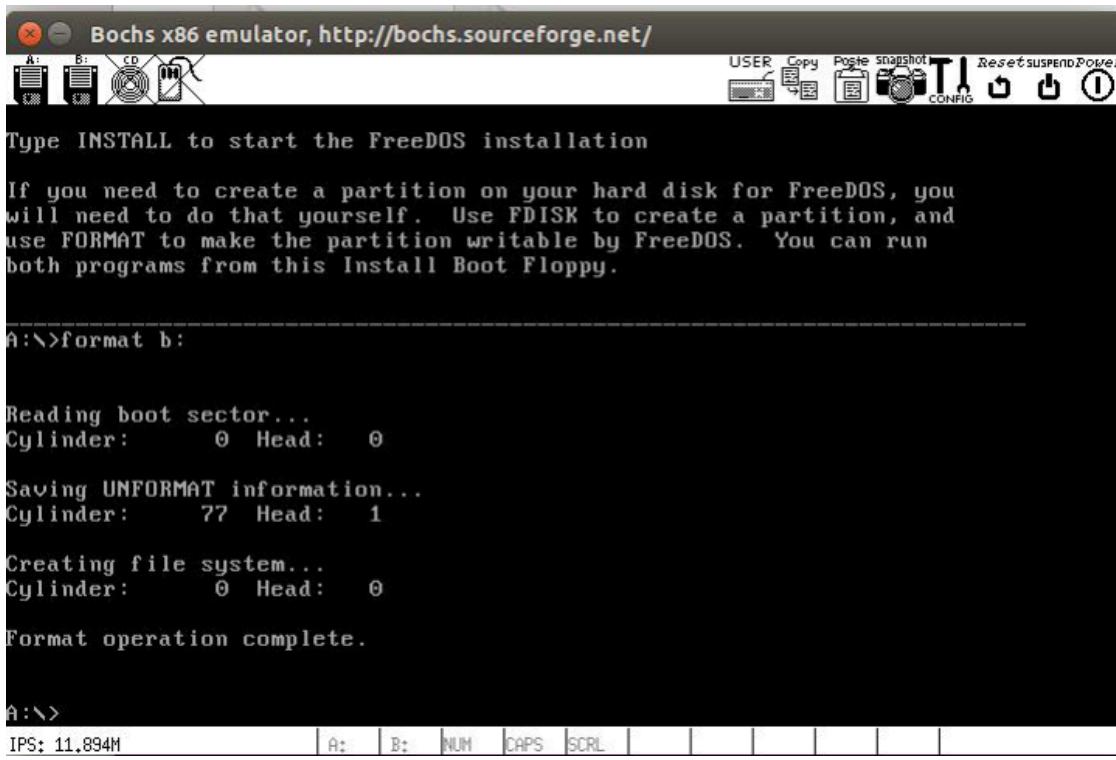


Figure 2

- ④ 将上述三个文件写入软盘中

```
sudo mkdir /mnt/floppy
sudo mount -o loop pm.img /mnt/floppy
sudo cp HDW.txt /mnt/floppy
sudo umount /mnt/floppy
```

Fence 1

### 3.1.2 寻找文件

根目录区存放着根目录的内容，根目录下所有文件和子目录的文件控制块，一步一步找下去，可以找到文件对应的文件控制块，文件控制块的表项记录了该文件存放在磁盘的第一个起始簇号。再由该起始簇号，通过文件分配表 FAT，找出所有该文件存放的簇链，如下所示：

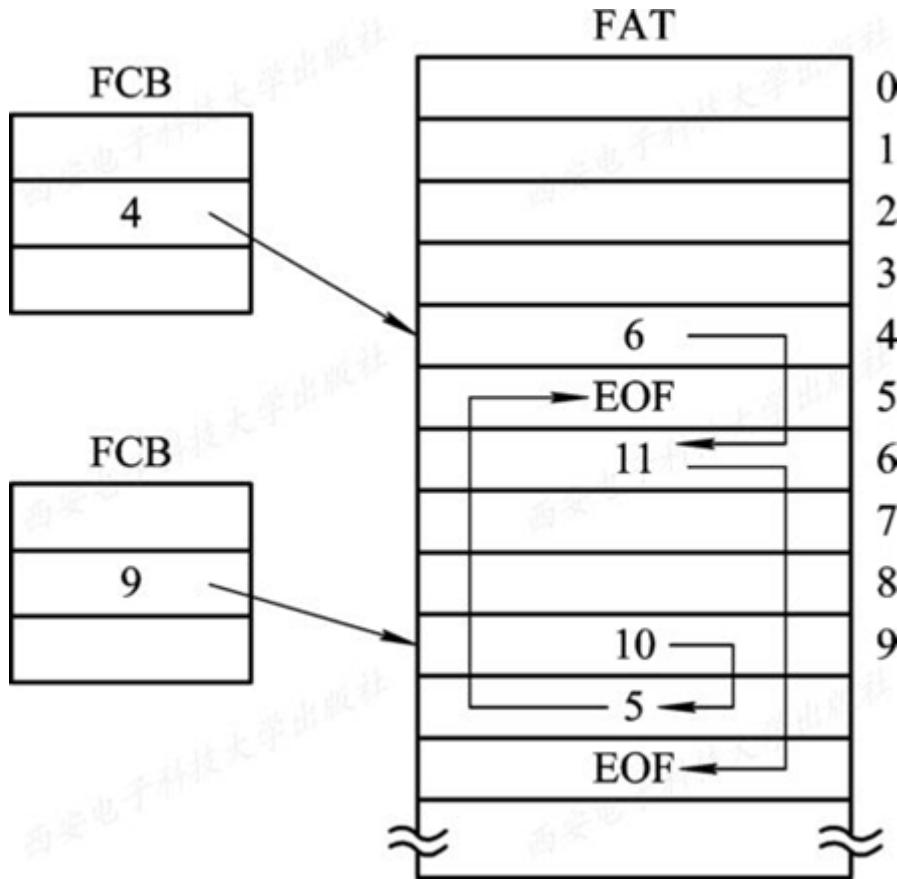


Figure 3

- 1 根目录区读取起始簇号：由于根目录区从第 19 扇区开始，故其第一个字节位于偏移  $19 \times 512 = 0x2600$  处。使用命令 `xxd -u -a -g 1 -c 16 -s +0x2600 -l 512 pm.img` 得到以下结果

```
(base) rimeheart@Rimeheart ~/0/1/o/a> xxd -u -a -g 1 -c 16 -s +0x2600 -l 512 pm.img
00002600: 52 49 56 45 52 20 20 20 54 58 54 20 00 07 5D 7B RIVER TXT ...]{ 
00002610: 50 59 50 59 00 00 5D 7B 50 59 04 00 0F 00 00 00 PYPY...]{PY.....
00002620: 41 48 00 44 00 57 00 2E 00 74 00 0F 00 64 78 00 AH.D.W...t...dx.
00002630: 74 00 00 00 FF FF FF FF FF FF 00 00 FF FF FF FF t..... 
00002640: 48 44 57 20 20 20 20 54 58 54 20 00 9C 65 7B HDW TXT ..e{ 
00002650: 50 59 50 59 00 00 65 7B 50 59 05 00 3B 00 00 00 PYPY..e{PY..;...
00002660: 48 44 57 53 20 20 20 54 58 54 20 00 8B 6F 7B HDWS TXT ..o{ 
00002670: 50 59 50 59 00 00 6F 7B 50 59 06 00 1B 02 00 00 PYPY..o{PY.....
00002680: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... 
*
000027f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... 
```

Figure 4

可以看到三个文件的目录项，读取每个项的第 27、28 字节，得到三个文件的开始簇号分别为 4、5、6。

- 2 FAT 表读取簇链：FAT1 表位于引导扇区之后的第一个扇区，使用命令 `xxd` 从偏移量 0x200 开始读取。

```

(base) rimeheart@Rimeheart ~/0/1/o/a> xxd -u -a -g 1 -c 16 -s +0x200 -l 512 pm.img
00000200: F0 FF FF 00 00 00 FF FF FF 07 F0 FF 00 00 00 00 ..... .
00000210: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
*
000003f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .

```

Figure 5

通过查看 FAT 表，我们发现前两个文件的 FAT 表项值为 0xFFFF，这表示这些文件的簇链已经结束。而第三个文件的 FAT 表项值为 7，这意味着该文件的内容继续存储在第 7 个簇中，并且第 7 个簇是该文件的最后一个簇。

- ③ 数据区读取内容：数据区开始扇区号 = 根目录区开始扇区号 + 14 = 19 + 14 = 33。数据区的第一个簇的簇号是 2，其偏移量为 0x4200。随着簇号的增大，偏移量每次增加 512 字节。为了读取文件内容，我们依次读取簇号为 4、5、6 和 7 的数据。每个簇的偏移量如下：

- 簇号 4 的偏移量： $0x4200 + (4 - 2) * 512 = 0x4600$
- 簇号 5 的偏移量： $0x4200 + (5 - 2) * 512 = 0x4800$
- 簇号 6 的偏移量： $0x4200 + (6 - 2) * 512 = 0x4A00$
- 簇号 7 的偏移量： $0x4200 + (7 - 2) * 512 = 0x4C00$

通过这些偏移量，使用 `xxd` 命令成功读取到文件的内容。

```

(base) rimeheart@Rimeheart ~/0/1/o/a> xxd -u -a -g 1 -c 16 -s +0x4600 -l 512 pm.img
00004600: 72 69 76 65 72 72 69 76 65 72 72 69 76 65 72 00 riverriverriver.
00004610: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
*
000047f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
(base) rimeheart@Rimeheart ~/0/1/o/a> xxd -u -a -g 1 -c 16 -s +0x4800 -l 512 pm.img
00004800: 48 75 61 6E 67 20 44 6F 6E 67 77 65 69 2C 20 6F Huang Dongwei, o
00004810: 72 20 48 6F 75 20 44 75 61 6E 20 57 61 6E 67 2C r Hou Duan Wang,
00004820: 20 77 68 69 63 68 20 6F 6E 65 20 69 73 20 74 68 which one is th
00004830: 65 20 74 72 75 65 20 48 44 57 3F 00 00 00 00 00 e true HDW?.....
00004840: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .
*
000049f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ..... .

```

Figure 6

```

(base) rimeheart@Rimeheart ~/0/1/o/a> xxd -u -a -g 1 -c 16 -s +0x4A00 -l 512 pm.img
00004a00: 48 75 61 6E 67 20 44 6F 6E 67 77 65 69 2C 20 6F Huang Dongwei, o
00004a10: 72 20 48 6F 75 20 44 75 61 6E 20 57 61 6E 67 2C r Hou Duan Wang,
00004a20: 20 77 68 69 63 68 20 6F 6E 65 20 69 73 20 74 68 which one is th
00004a30: 65 20 74 72 75 65 20 48 44 57 3F 0A 48 75 61 6E e true HDW?.Huan
00004a40: 67 20 44 6F 6E 67 77 65 69 2C 20 6F 72 20 48 6F g Dongwei, or Ho
00004a50: 75 20 44 75 61 6E 20 57 61 6E 67 2C 20 77 68 69 u Duan Wang, whi
00004a60: 63 68 20 6F 6E 65 20 69 73 20 74 68 65 20 74 72 ch one is the tr
00004a70: 75 65 20 48 44 57 3F 0A 48 75 61 6E 67 20 44 6F ue HDW?.Huang Do
00004a80: 6E 67 77 65 69 2C 20 6F 72 20 48 6F 75 20 44 75 ngwei, or Hou Du
00004a90: 61 6E 20 57 61 6E 67 2C 20 77 68 69 63 68 20 6F an Wang, which o
00004aa0: 6E 65 20 69 73 20 74 68 65 20 74 72 75 65 20 48 ne is the true H
00004ab0: 44 57 3F 0A 48 75 61 6E 67 20 44 6F 6E 67 77 65 Dw?.Huang Dongwe
00004ac0: 69 2C 20 6F 72 20 48 6F 75 20 44 75 61 6E 20 57 i, or Hou Duan W
00004ad0: 61 6E 67 2C 20 77 68 69 63 68 20 6F 6E 65 20 69 ang, which one i
00004ae0: 73 20 74 68 65 20 74 72 75 65 20 48 44 57 3F 0A s the true HDW?.
00004af0: 48 75 61 6E 67 20 44 6F 6E 67 77 65 69 2C 20 6F Huang Dongwei, o
00004b00: 72 20 48 6F 75 20 44 75 61 6E 20 57 61 6E 67 2C r Hou Duan Wang,

```

Figure 7

```
(base) rimeheart@Rimeheart ~/0/1/o/a> xxd -u -a -g 1 -c 16 -s +0x4C00 -l 512 pm.img
00004c00: 20 77 68 69 63 68 20 6F 6E 65 20 69 73 20 74 68 which one is th
00004c10: 65 20 74 72 75 65 20 48 44 57 3F 00 00 00 00 00 e true HDW?.....
00004c20: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....*
00004df0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

Figure 8

## 3.2 装入可执行文件

为加载 loader.bin 到软盘，需要修改 boot.asm，让引导扇区寻找 Loader.bin，即依次寻找 **Load.bin** 的起始簇号和簇链，通过读取 **Load.bin** 的各个扇区，将 **Loader** 加载入内存，最后把控制权交给 **Loader**。

### 3.2.1 读取软盘扇区

要将一个文件加载进入内存的话，需要读取软盘，那么就会用到 BIOS 的 13h 号中断，该中断的 02H 功能号的作用是读取扇区，不同 AH 取值对应的功能如下：

AH 取值	功能
00h	复位磁盘驱动器
01h	检查磁盘驱动器状态
02h	读扇区
03h	写扇区
04h	校验扇区
05h	格式化磁道
08h	获取驱动器参数
09h	初始化硬盘驱动器参数
0Ch	寻道
0Dh	复位磁盘控制器
15h	获取驱动器类型

Table 2

在 BIOS 中断 13h 的磁盘读操作中，参数 DL 指定了驱动器号，DH 指定了磁头号，CH 指定了柱面号，CL 指定了扇区号。该操作从指定的驱动器、磁头、柱面和扇区开始，连续读取 AL 个扇区的数据，并将这些数据存储到由 ES: BX 指向的缓冲区中。需要的参数如下：

参数	功能
AL	处理对象扇区数（连续的扇区）
CH	柱面号
CL	扇区号
DH	磁头号
DL	驱动器号
ES: BX	缓冲地址（校验及寻道时不使用）
CF	判断是否读盘成功

Table 3

对于 1.44MB 的软盘，总共有两面（磁头号 0 和 1），每面有 80 个磁道（柱面号 0 到 79），每个磁道有 18 个扇区（扇区号 1 到 18）。

因此，对于一个线性扇区号 L，计算步骤如下：

- ① 计算扇区号 S:  $S = (L \% 18) + 1$
- ② 计算柱面号 C :  $C = L / (2 * 18)$
- ③ 计算磁头号 H:  $H = (L / 18) \% 2$

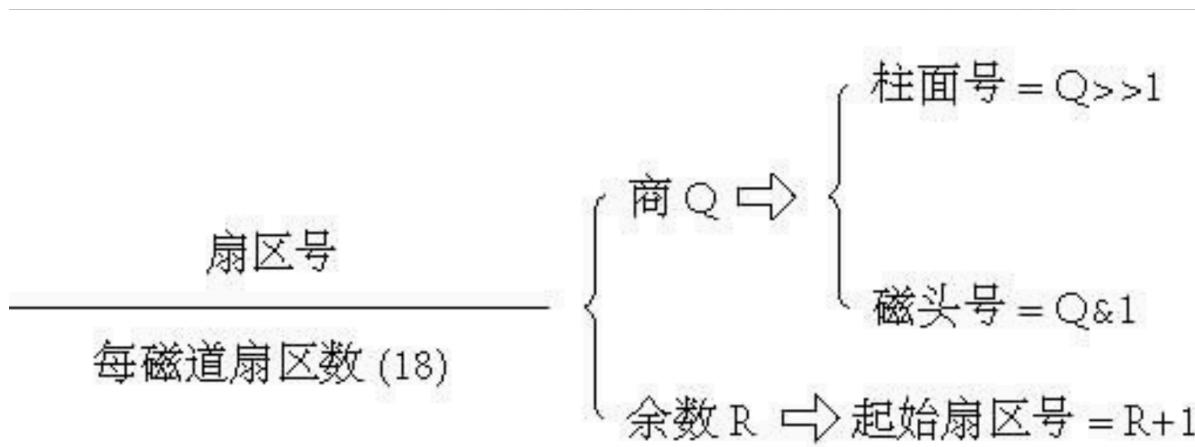


Figure 9

根据计算过程，使用函数 ReadSector 实现计算参数并调用 int 13h 读取扇区的功能。

```
;-----  
; 函数名: ReadSector  
;  
;  
; 作用:  
; 从第 ax 个 Sector 开始, 将 cl 个 Sector 读入 es:bx 中  
ReadSector:  
;  
;  
; 怎样由扇区号求扇区在磁盘中的位置 (扇区号 → 柱面号, 起始扇区, 磁头号)  
;  
;  
; 设扇区号为 x  
;  
;           | 柱面号 = y >> 1  
;   x       | 商 y |  
; ----- ⇒ |       | 磁头号 = y & 1  
; 每磁道扇区数 |  
;           | 余 z ⇒ 起始扇区号 = z + 1  
;  
push    bp  
mov     bp, sp  
sub    esp, 2      ; 辟出两个字节的堆栈区域保存要读的扇区数: byte [bp-2]  
  
mov    byte [bp-2], cl  
push    bx          ; 保存 bx  
mov    bl, [BPB_SecPerTrk] ; bl: 除数  
div    bl          ; y 在 al 中, z 在 ah 中  
inc    ah          ; z ++  
mov    cl, ah        ; cl ← 起始扇区号  
mov    dh, al        ; dh ← y  
shr    al, 1         ; y >> 1 (其实是 y/BPB_NumHeads, 这里  
BPB_NumHeads=2)  
mov    ch, al        ; ch ← 柱面号  
and    dh, 1         ; dh & 1 = 磁头号  
pop    bx          ; 恢复 bx  
; 至此, "柱面号, 起始扇区, 磁头号" 全部得到 ^^^^^^^^^^^^^^  
mov    dl, [BS_DrvNum]      ; 驱动器号 (0 表示 A 盘)  
.GoOnReading:  
mov    ah, 2          ; 读  
mov    al, byte [bp-2]    ; 读 al 个扇区  
int    13h
```

```

jc .GoOnReading      ; 如果读取错误 CF 会被置为 1, 这时就不停地读,
直到正确为止

add esp, 2
pop bp

ret

```

Fence 2

### 3.2.2 寻找 Load.bin 的起始簇号

为了将 Loader 读取到内存中，需要知道 Loader 的起始簇号，因此通过遍历根目录来找到 loader 的目录项来确定起始簇号。

#### ① 软驱复位，调用 int13h

```

xor ah, ah ; `.
xor dl, dl ; | 软驱复位
int 13h ; /

```

Fence 3

#### ② 遍历根目录：依次读取根目录的每个扇区，并将读取的扇区数据存入内存的 `BaseOfLoader:OffsetOfLoader` 地址。

```

LABEL_SEARCH_IN_ROOT_DIR_BEGIN:
    cmp word [wRootDirSizeForLoop], 0      ; 判断根目录区是否已读完
    jz LABEL_NO_LOADERBIN      ; 如果读完, 跳转到未找到文件的处理
    dec word [wRootDirSizeForLoop]   ; 递减根目录剩余的扇区数
    mov ax, BaseOfLoader
    mov es, ax          ; ES 段寄存器指向 Loader 文件的加载地址
    mov bx, OffsetOfLoader ; BX 设置为 Loader 文件的偏移量
    mov ax, [wSectorNo]     ; 将当前扇区号存入 AX
    mov cl, 1          ; 准备读取 1 个扇区
    call ReadSector      ; 调用 ReadSector 函数, 读取扇区数据到内
    存

```

Fence 4

#### ③ 遍历根目录扇区：`si` 被初始化为 `LoaderFileName`，即要查找的文件名 "LOADER BIN"。设置 `di`，让 `es:di` 指向 Loader 加载的内存偏移地址。`dx` 用于记录剩余项的数量，控制每个扇区内的文件项遍历。一个扇区包含 16 个文件项，每个项长度为 32 字节。

```

mov si, LoaderFileName ; DS:SI → "LOADER BIN" 这个是要查找
的文件名
mov di, OffsetOfLoader ; ES:DI → Loader 加载的内存偏移地址

```

```

cld          ; 清除方向标志，保证字符串比较时是向前读取
mov dx, 10h

LABEL_SEARCH_FOR_LOADERBIN:
    cmp dx, 0           ; 判断根目录扇区是否读完
    jz LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR ; 如果读完一个扇区，跳转
到下一个扇区
    dec dx             ; 递减计数
    mov cx, 11          ; 文件名长度为 11 个字符

LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR:
    add word [wSectorNo], 1 ; 将根目录扇区号递增，读取下一个扇区
    jmp LABEL_SEARCH_IN_ROOT_DIR_BEGIN ; 返回根目录查找循环

```

- ④ 文件名比较：循环逐字符比较读取到的文件名和 "LOADER BIN"。`lodsb` 指令会从 `DS:SI` 中取出一个字节（文件名的一个字符），并存入 `AL`，然后与 `ES:DI` 指向的内存中的字节（根目录中的文件名）比较。如果完全匹配 11 个字符，跳转到 `LABEL_FILENAME_FOUND`；否则，跳转到 `LABEL_DIFFERENT`，调整 `di` 的值，让其跳过当前文件项，转到下一个文件项继续查找。

```

LABEL_CMP_FILENAME:
    cmp cx, 0
    jz LABEL_FILENAME_FOUND ; 如果 11 个字符都匹配，表示找到
Loader.bin
    dec cx
    lodsb           ; 取出 DS:SI 指向的字节放入 AL 寄存器
    cmp al, byte [es:di] ; 比较 AL 中的字符和 ES:DI 指向的字符
    jz LABEL_GO_ON
    jmp LABEL_DIFFERENT ; 如果字符不匹配，跳到下一个文件项

LABEL_GO_ON:
    inc di
    jmp LABEL_CMP_FILENAME ; 继续比较下一个字符

LABEL_DIFFERENT:
    and di, 0FFE0h      ; 将 DI 寄存器的指针对齐到下一个目录项
    add di, 20h         ; 让 DI 指向下一个目录项
    mov si, LoaderFileName ; 重新指向 Loader 文件名
    jmp LABEL_SEARCH_FOR_LOADERBIN

```

- ⑤ 查找结束：

- 文件未找到： 显示 "No LOADER." 消息。如果定义了 `_BOOT_DEBUG_` 宏，则程序会通过中断返回 DOS，否则进入死循环

```

LABEL_NO_LOADERBIN:
    mov dh, 2           ; 显示 "No LOADER." 的消息
    call DispStr        ; 显示消息
%ifdef _BOOT_DEBUG_
    mov ax, 4c00h       ; 调用 DOS 中断返回
    int 21h
%else
    jmp $              ; 死循环，表示没有找到文件
%endif

```

Fence 7

- 文件找到：进入 `LABEL_FILENAME_FOUND` 函数，将会加载 Loader，在后两节提到。

```

LABEL_FILENAME_FOUND:          ; 找到 LOADER.BIN 后便来到这里继续
    mov ax, RootDirSectors
    and di, 0FFE0h      ; di → 当前项的开始
    add di, 01Ah         ; di → 首 Sector
    mov cx, word [es:di]
    push cx             ; 保存此 Sector 在 FAT 中的序号
    add cx, ax
    add cx, DeltaSectorNo ; cl ← LOADER.BIN的起始扇区号(0-based)
    mov ax, BaseOfLoader
    mov es, ax           ; es ← BaseOfLoader
    mov bx, OffsetOfLoader ; bx ← OffsetOfLoader
    mov ax, cx             ; ax ← Sector 号

```

Fence 8

### 3.2.3 寻找 Load.bin 的簇链

找到起始簇号后，还需要从 FAT 中得到簇链，从而可以将 loader 读入内存。因此我们调用函数 `GetFATEntry`，查找当前簇号在 FAT 表中对应的值。

- ① 保存寄存器和初始化：es 寄存器设置为指向 FAT 读取后的内存地址。在 `BaseOfLoader` 的后面留出 4KB 空间，用于存放 FAT 表。

```

; -----
; 函数名: GetFATEntry
;
; -----

```

```

; 作用:
; 找到序号为 ax 的 Sector 在 FAT 中的项, 结果放在 ax 中
; 需要注意的是, 中间需要读 FAT 的扇区到 es:bx 处, 所以函数一开始保存了
es 和 bx
GetFATEntry:
    push    es
    push    bx
    push    ax
    mov ax, BaseOfLoader; `.
    sub ax, 0100h ; | 在 BaseOfLoader 后面留出 4K 空间用于存放
FAT
    mov es, ax ; /
    pop ax

```

## ② 计算指定簇号在 FAT 中的偏移 : Fence 9

- **ax** 中的初始值是要查找的簇号。FAT12 文件系统中的每个簇号占用 1.5 字节, 因此将簇号乘以 3 并除以 2, 以确定它的位置。
- **dx** 中的余数用于区分簇号是奇数还是偶数。如果余数为 0, 表示这是偶数簇; 否则, 它是奇数簇。

如果是奇数簇 (**dx ≠ 0**), 会设置 **b0dd** 标志, 表示稍后需要特殊处理。

```

mov byte [b0dd], 0
mov bx, 3
mul bx          ; ax = ax * 3 (一个簇占用1.5字节, 因此乘以3)
mov bx, 2
div bx          ; ax = ax / 2 ==> ax ← 商, dx ← 余数
cmp dx, 0
jz LABEL_EVEN
mov byte [b0dd], 1

```

Fence 10

## ③ 计算扇区号和偏移量

- **ax** 中存储的是 FATT 表中的字节偏移量, **BPB\_BytsPerSec** 是每个扇区的字节数。将 **ax** 除以每扇区字节数, 以确定项所在的具体扇区。
- **dx** 中保存项在该扇区内的偏移量 (即项在扇区内的位置)。

```

LABEL_EVEN:
    xor dx, dx
    mov bx, [BPB_BytsPerSec]
    div bx ; ax / 每扇区字节数, 计算出簇号所在的扇区
            ; ax ← 商 (FATEntry 在 FAT 中的扇区号)
            ; dx ← 余数 (FATEntry 在该扇区内的偏移)

```

Fence 11

#### ④ 读取 FAT 项所在的扇区

- 将 `ax` 中的扇区号加上 FAT1 的起始扇区号，得出 FAT 项所在的具体扇区。
- 调用 `ReadSector` 函数读取这个扇区。由于 FAT12 中的簇号占 12 位，因此一次读取两个扇区，以防项跨越扇区边界。
- 读取完成后，计算项在扇区内的偏移，使用 `es:bx` 获取 FAT 表中的项值。

```
push    dx
mov bx, 0 ; 设置 bx 为 0, 准备从 es:bx 地址读取数据
add ax, SectorNoOfFAT1 ; 将 FAT 的偏移量加上 FAT1 的起始扇区号
mov cl, 2
call    ReadSector ; 读取该扇区 (一次读取两个扇区, 避免跨扇区读取时出错)
pop dx
add bx, dx
mov ax, [es:bx] ; 取出 FATEntry 的值
```

Fence 12

#### ⑤ 处理奇偶簇号

如果 `b0dd` 标志为 1，说明是奇数簇号，需要将 `ax` 右移 4 位，因为奇数簇号的高 4 位位于前一个字节的末尾。否则，通过 `and 0FFFh` 保留低 12 位。最后保存寄存器并返回。

```
cmp byte [b0dd], 1
jnz LABEL EVEN_2
shr ax, 4
LABEL EVEN_2:
and ax, 0FFFh

LABEL GET_FAT_ENTRY_OK:
pop bx
pop es
ret
```

Fence 13

### 3.2.4 加载 Loader.bin

实现在根目录中找到首簇、在 FAT 找到簇链的功能后，通过调用 `ReadSector` 读取扇区，即可将 Loader 读取进入内存。

- 1 每次读取扇区时，输出一个点以指示加载进度。
- 2 调用 `ReadSector` 将扇区读入 `BaseOfLoader: OffsetOfLoader` 中
- 3 通过 `GetFATEntry` 获取下一个扇区的 FAT 项。如果 `ax` 的值为 `0FFFh`，表示已达到文件末尾，程序跳转到 `LABEL_FILE_LOADED`。如果还未结束，继续读取下一个扇区。

```

LABEL_FILENAME_FOUND:           ; 找到 LOADER.BIN 后便来到这里继续
    mov ax, RootDirSectors
    and di, 0FFE0h      ; di → 当前项的开始
    add di, 01Ah        ; di → 首 Sector
    mov cx, word [es:di]
    push cx            ; 保存此 Sector 在 FAT 中的序号
    add cx, ax
    add cx, DeltaSectorNo ; cl ← LOADER.BIN的起始扇区号(0-
                           based)
    mov ax, BaseOfLoader
    mov es, ax          ; es ← BaseOfLoader
    mov bx, OffsetOfLoader ; bx ← OffsetOfLoader
    mov ax, cx          ; ax ← Sector 号

LABEL_GOON_LOADING_FILE:
    push ax             ; `.
    push bx             ; |
    mov ah, 0Eh          ; | 每读一个扇区就在 "Booting" 后面
    mov al, '.'          ; | 打一个点，形成这样的效果：
    mov bl, 0Fh          ; | Booting .....
    int 10h             ; |
    pop bx              ; |
    pop ax              ; /

    mov cl, 1
    call ReadSector
    pop ax              ; 取出此 Sector 在 FAT 中的序号
    call GetFATEntry
    cmp ax, 0FFFh
    jz LABEL_FILE_LOADED
    push ax            ; 保存 Sector 在 FAT 中的序号
    mov dx, RootDirSectors
    add ax, dx
    add ax, DeltaSectorNo
    add bx, [BPB_BytsPerSec]

```

```

        jmp LABEL_GOON_LOADING_FILE
LABEL_FILE_LOADED:

        mov dh, 1           ; "Ready."
        call DispStr        ; 显示字符串
```

```

Fence 14

- ④ 最后跳转到 LOADER.BIN 的开始处，将控制权交给 Loader

```

; **** * **** * **** * **** * **** * **** * **** * **** * **** * ****
**** * **** * **** * **** * **** * **** * **** * **** * **** * ****
*****  

        jmp BaseOfLoader: OffsetOfLoader      ; 这一句正式跳转到已加载到内
  ; 存中的 LOADER.BIN 的开始处,
  ; 开始执行 LOADER.BIN 的代码。
  ; Boot Sector 的使命到此结束
; **** * **** * **** * **** * **** * **** * **** * **** * **** * ****
**** * **** * **** * **** * **** * **** * **** * **** * **** * ****
*****  


```

Fence 15

### 3.2.5 实验结果

- ① 编译 boot.asm 得到 boot.bin，将其写入引导扇区

```

nasm boot.asm -o boot.bin
dd if = boot.bin of = x.img bs = 512 count = 1 conv = notrunc

```

Fence 16

- ② 新建一个 loader.asm，实现简单的功能：在屏幕第一行中央出现字符“L”。之后将其编译并挂载到镜像。

```

org 0100h

        mov ax, 0B800h
        mov gs, ax
        mov ah, 0Fh          ; 0000: 黑底    1111: 白字
        mov al, 'L'
        mov [gs:((80 * 0 + 39) * 2)], ax    ; 屏幕第 0 行, 第 39 列

        jmp $                ; 到此停住

```

```
nasm boot.asm -o boot.bin Fence 17
sudo mount -o loop a.img /mnt/floppy
sudo cp loader.bin /mnt/floppy
sudo umount /mnt/floppy
```

- ③ 启动 bochs Fence 18

```
bochs -f bochsrc
```

Fence 19

最后打印字符'L'，结果符合预期。

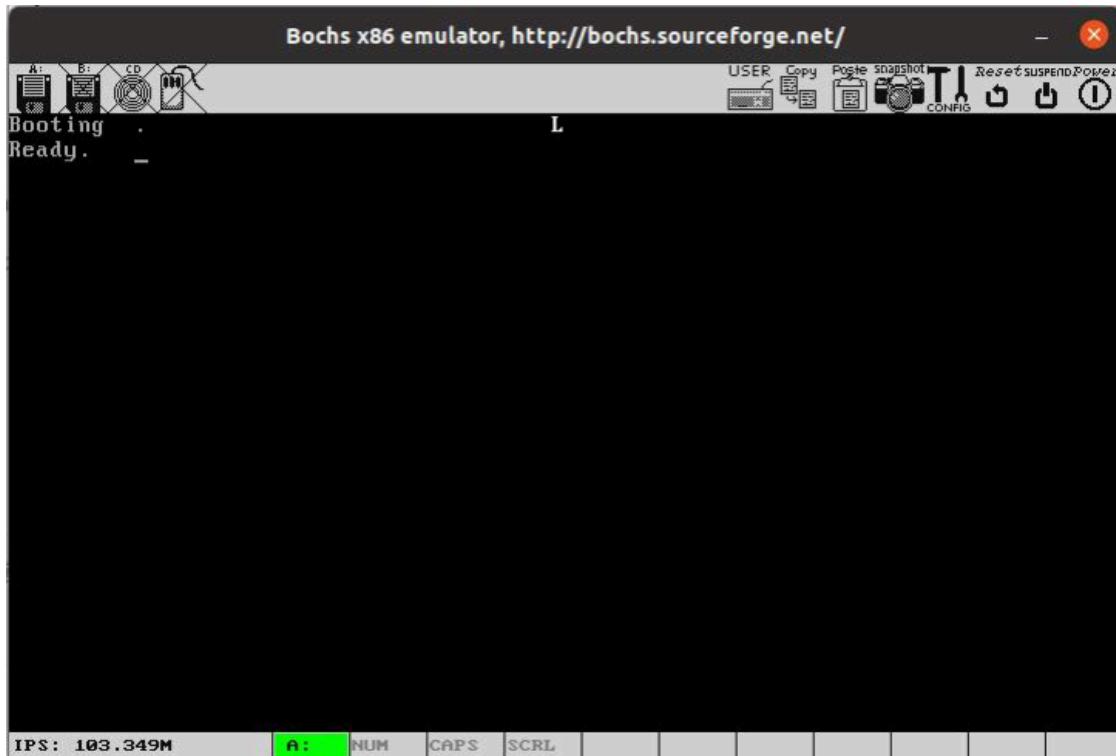


Figure 10

### 3.3 使用 xxd 读取二进制信息

**xxd** 是一个强大的命令行工具，用于查看和操作二进制文件的内容。它可以将二进制数据转换为十六进制和 ASCII 格式，反之亦然。以下是一些常用的选项和用法：

#### 3.3.1 常用选项

| 参数 | 内容                                                          |
|----|-------------------------------------------------------------|
| -a | 作用是自动跳过空白内容， 默认是关闭的                                         |
| -c | 加上数字表示每行显示多少字节的十六进制数， 默认是 16 字节                             |
| -g | 设定以几个字节为一块， 默认为 2 字节                                        |
| -l | 显示多少字节的内容                                                   |
| -s | 后面接 【±】 和 address 。 “+” 表示从地址处开始的内容，“-” 表示距末尾 address 开始的内容 |
| -b | 以二进制（0 or 1）的形式查看文件内容                                       |

Table 4

### 3.3.2 基本用法示例

#### ① 查看整个文件的内容：

```
-bash-3.2$ xxd demon.c
0000000: 2369 6e63 6c75 6465 203c 7374 6469 6f2e #include <
stdio.
0000010: 683e 0a2f 2f74 6869 7320 6973 206d 6169 h >./this is
mai
0000020: 6e20 6675 6e63 0a69 6e74 206d 6169 6e28 n func.int
main(
0000030: 290a 7b0a 2020 2020 696e 7420 6e75 6d20 ).{. int
num
0000040: 3d31 303b 0a20 2020 2070 7269 6e74 6628 = 10;.
printf(
0000050: 226e 756d 6265 7220 6973 2025 645c 6e22 "number is
%d\n"
0000060: 2c6e 756d 293b 0a20 2020 2072 6574 7572 , num);.
return
0000070: 6e20 303b 0a7d 0a0a n 0;..}
```

Fence 20

#### ② 从特定偏移量读取：

```
(base) rimeheart@Rimeheart ~/0/l/o/c > xxd -s +0x100 boot.bin
00000100: 1e0b 7ceb d7b6 01e8 3100 ea00 0100 900e
.. |.....1.....
```

```
00000110: 0000 0000 4c4f 4144 4552 2020 4249 4e00 ....LOADER
BIN.
00000120: 426f 6f74 696e 6720 2052 6561 6479 2e20 Booting
Ready.
00000130: 2020 4e6f 204c 4f41 4445 52b8 0900 f6e6 No
LOADER.....
00000140: 0520 7d89 c58c d88e c0b9 0900 b801 13bb .
}.....
00000150: 0700 b200 cd10 c355 89e5 6683 ec02 884e
.....U..f....N
00000160: fe53 8a1e 187c f6f3 fec4 88e1 88c6 d0e8
.S...|.....
00000170: 88c5 80e6 015b 8a16 247c b402 8a46 fecd .....
[...$|...F..
00000180: 1372 f766 83c4 025d c306 5350 b800 902d
.r.f....]..SP....
00000190: 0001 8ec0 58c6 0613 7d00 bb03 00f7 e3bb
....X...}.....
000001a0: 0200 f7f3 83fa 0074 05c6 0613 7d01 31d2
.....t....}.1.
000001b0: 8b1e 0b7c f7f3 52bb 0000 83c0 01b1 02e8
...|..R.....
000001c0: 95ff 5a01 d326 8b07 803e 137d 0175 03c1
..Z..&...>.}.u..
000001d0: e804 25ff 0f5b 07c3 0000 0000 0000 0000 ..%..
[.....
000001e0: 0000 0000 0000 0000 0000 0000 0000 0000
......
000001f0: 0000 0000 0000 0000 0000 0000 0000 55aa
.....U.
```

③ 指定字节数: Fence 21

```
-bash-3.2$ xxd -l 16 demon.c
0000000: 2369 6e63 6c75 6465 203c 7374 6469 6f2e #include <
stdio.
```

Fence 22

④ 结合多个选项:

```
(base) rimeheart@Rimeheart ~/0/l/o/c > xxd -u -a -g 1 -c 16 -s
+0x0 -l 64 boot.bin
00000000: EB 3C 90 46 6F 72 72 65 73 74 59 00 02 01 01 00 .
<.ForrestY.....
00000010: 02 E0 00 40 0B F0 09 00 12 00 02 00 00 00 00 00
...@.....
00000020: 00 00 00 00 00 00 29 00 00 00 00 4F 72 61 6E 67
.....)....Orang
00000030: 65 53 30 2E 30 32 46 41 54 31 32 20 20 20 8C C8
eS0.02FAT12 ..
```

Fence 23

### 3.3.3 在实验1、2中验证

实验一 `xxd` 验证已经在实验过程中使用过，[点击此处跳转](#)，接下来是实验2调试和验证过程：

- 在搜寻根目录 `LABEL_SEARCH_IN_ROOT_DIR_BEGIN` 时，调用 `ReadSector` 加载根目录扇区内存到 `BaseOfLoader:OffsetOfLoader`，此时添加断点，并使用 `x` 命令读取

```
(0) Magic breakpoint
Next at t=14534392
(0) [0x000000007c8b] 0000:7c8b (unk. ctxt): cld ; fc
<bochs:2> reg
CPU0:
eax: 00000000_00000001
ebx: 00000000_00000100
ecx: 00000000_00090002
edx: 00000000_00000100
esp: 00000000_00007c00
ebp: 00000000_00007d24
esi: 00000000_000e7d18
edi: 00000000_00000100
eip: 00000000_00007c8b
eflags 0x00000006: id vip vif vm rf nt IOPL=0 of df if tf sf zf af PF cf
<bochs:3>[]
```

Figure 11

```
<bochs:22> x /16xcb es:di +32
[bochs]:
0x0000000000090120 <bogus+> 0>: L O A D E R
0x0000000000090128 <bogus+> 8>: B I N \0 \x15 \xC7 ,
```

Figure 12

发现在 `es:di (0x9000:0100)` 往后 32 字节偏移处读取到 `loader.bin` 文件名，即文件项开始处。

- 在成功进入 `LABEL_FILENAME_FOUND` 后添加断点，此时 `es:0x9000, di: 0x0120`，与前一次断点 `es:di` 相差 32 字节偏移，符合结果预期。

```

00014534506i[CPU0 ] [14534506] Stopped on MAGIC BREAKPOINT
(0) Magic breakpoint
Next at t=14534506
(0) [0x000000007cca] 0000:7cca (unk. ctxt): add di, 0x001a ; 83c71a
<bochs:4> reg
CPU0:
eax: 00000000_0000000e
ebx: 00000000_00000100
ecx: 00000000_00090000
edx: 00000000_0000000e
esp: 00000000_00007c00
ebp: 00000000_00007d24
esi: 00000000_000e7d23
edi: 00000000_00000120
eip: 00000000_00007cca
eflags 0x00000002: id vip vif ac vm rf nt IOPL=0 of df if tf sf zf af PF cf
<bochs:5>[]

```

Figure 13

单步执行 `and di, 0FFE0h`, `add di, 01Ah` 后, `es:di` 指向该项的第 26 字节处, 此处保存着起始簇号。通过 `x` 命令读取, 起始簇号为 3。

```

<bochs:10> x /2xb es:di
[bochs]:
0x0000000000009013a <bogus+          0>:      0x03      0x00

```

Figure 14

- ③ 读取簇链时, 内存 `0x8f000` 的位置存放了 FAT 表, 通过 `x` 命令读取如下:

```

<bochs:49> x/16xb 0x8f000
[bochs]:
0x000000000008f000 <bogus+      0>:  0x00  0x00  0x00  0x00  0x00  0xf0  0xff  0x00  0x00
0x000000000008f008 <bogus+      8>:  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00  0x00

```

Figure 15

- ④ 最后通过 bochs 的 `writemem` 指令, 获取了虚拟机物理内存的在 `0x90100` 处的二进制文件。使用 `xxd` 查看, 存放了 Loader 程序的代码, 证明 Loader 已经成功地被加载到内存中了。

```

writemem "load.bin" 0x90100 32
xxd -u -a -g 1 -c 16 load.bin

```

Fence 24

```

(base) rimeheart@Rimeheart:~/OSlab/lab4/osfs04-master/c$ xxd -u -a -g 1 -c 16 load.bin
00000000: B8 00 B8 8E E8 B4 0F B0 4C 65 A3 4E 00 EB FE 00 .....Le.N....
00000010: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

```

Figure 16

## 3.4 实验问题与故障分析

### 3.4.1 根目录项描述存在出入

在查阅资料时，发现网上与书中对根目录项的字段长度及含义存在出入，为此翻阅官方文档并通过实验验证，给出了更精确的分析，写在了FAT12结构概述的[根目录区](#)，[点击跳转](#)。

### 3.4.2 文件名大小写探讨

在手工创建文件时，不小心以小写的 `txt` 为扩展名，在 `xxd` 命令查看时产生了奇怪的现象。为此在 `format` 之后，分别创建 `HDW.TXT`、`HDW.txt`、`hdw.TXT` `hdw.txt` 进行比较，下面是依次查看根目录区的结果：

```
(base) rimeheart@Rimeheart ~/0/1/o/a> xxd -u -a -g 1 -c 16 -s +0x2600 -l 512 pm.img
00002600: 48 44 57 20 20 20 20 20 54 58 54 20 00 7F 4E 6D HDW TXT ..Nm
00002610: 53 59 53 59 00 00 4E 6D 53 59 03 00 3B 00 00 00 SYSY..NmSY..;...
00002620: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....*

```

Figure 17

```
(base) rimeheart@Rimeheart ~/0/1/o/a> xxd -u -a -g 1 -c 16 -s +0x2600 -l 512 pm.img
00002600: 41 48 00 44 00 57 00 2E 00 74 00 0F 00 64 78 00 AH.D.W...t...dx.
00002610: 74 00 00 00 FF FF FF FF FF 00 00 FF FF FF FF t.....
00002620: 48 44 57 20 20 20 20 20 54 58 54 20 00 99 CD 6E HDW TXT ...n
00002630: 53 59 53 59 00 00 CD 6E 53 59 03 00 3B 00 00 00 SYSY...nSY..;...
00002640: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....*
000027f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

Figure 18

```
(base) rimeheart@Rimeheart ~/0/1/o/a> sudo mount -o loop pm.img /mnt/floppy
[sudo] password for rimeheart:
(base) rimeheart@Rimeheart ~/0/1/o/a> sudo cp hdw.txt /mnt/floppy/
(base) rimeheart@Rimeheart ~/0/1/o/a> xxd -u -a -g 1 -c 16 -s +0x2600 -l 512 pm.img
00002600: 41 68 00 64 00 77 00 2E 00 74 00 0F 00 64 78 00 Ah.d.w...t...dx.
00002610: 74 00 00 00 FF FF FF FF FF 00 00 FF FF FF FF t.....
00002620: 48 44 57 20 20 20 20 20 54 58 54 20 00 30 CA 6B HDW TXT .0.k
00002630: 53 59 53 59 00 00 CA 6B 53 59 03 00 3B 00 00 00 SYSY...kSY..;...
00002640: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....*
```

Figure 19

```
(base) rimeheart@Rimeheart ~/0/1/o/a> sudo mount -o loop pm.img /mnt/floppy
(base) rimeheart@Rimeheart ~/0/1/o/a> sudo cp HDW.TXT /mnt/floppy/
(base) rimeheart@Rimeheart ~/0/1/o/a> xxd -u -a -g 1 -c 16 -s +0x2600 -l 512 pm.img
00002600: 41 68 00 64 00 77 00 2E 00 54 00 0F 00 64 58 00 Ah.d.w...T...dX.
00002610: 54 00 00 00 FF FF FF FF FF 00 00 FF FF FF FF T.....
00002620: 48 44 57 20 20 20 20 20 54 58 54 20 00 B2 99 6E HDW TXT ...n
00002630: 53 59 53 59 00 00 99 6E 53 59 03 00 3B 00 00 00 SYSY...nSY..;...
00002640: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....*
```

Figure 20

可以看到FAT12文件名不区分大小写字母：当文件名和扩展名有小写字母时，系统会保存一份以大写字母A开头、真正文件名紧随其后的目录项（长文件名项），而文件目录项保存大写的文件名。

通过翻阅wiki。找到了不同文件系统区分，下表列出了各现有文件系统是否区分大小写以及是否保留大小写：

|       | 大小写敏感                                                                              | 大小写不敏感                                                                 |
|-------|------------------------------------------------------------------------------------|------------------------------------------------------------------------|
| 保留大小写 | <u>UFS</u> 、 <u>ext3</u> 、 <u>ext4</u> 、 <u>HFS Plus</u> （可选）， <u>NTFS</u> （Unix下） | <u>VFAT</u> 和 <u>FAT32</u> 基本上始终随长文件名支持， <u>NTFS</u> 、 <u>HFS Plus</u> |
| 不留大小写 | 不可能                                                                                | <u>FAT12</u> 、 <u>FAT16</u> ，仅在无长文件名支持下。                               |

同时此表进一步引入了长文件名的探讨

### 3.4.3 长文件名探讨

在格式化后，进一步创建文件 `hdwhdwhdw.txt`、`HDWHDWHDW.TXT`

```
(base) rimeheart@Rimeheart ~/0/1/o/a> xxd -u -a -g 1 -c 16 -s +0x2600 -l 512 pm.img
00002600: 41 68 00 64 00 77 00 68 00 64 00 0F 00 48 77 00 Ah.d.w.h.d...Hw.
00002610: 68 00 64 00 77 00 2E 00 74 00 00 00 78 00 74 00 h.d.w...t...x.t.
00002620: 48 44 57 48 44 57 7E 31 54 58 54 20 00 51 2F 7A HDWHDW~1TXT .Q/z
00002630: 53 59 53 59 00 00 2F 7A 53 59 03 00 3B 00 00 00 SYSY../zSY..;...
00002640: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
*
000027f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

Figure 21

```
(base) rimeheart@Rimeheart ~/0/1/o/a> xxd -u -a -g 1 -c 16 -s +0x2600 -l 512 pm.img
00002600: 41 48 00 44 00 57 00 48 00 44 00 0F 00 48 57 00 AH.D.W.H.D...HW.
00002610: 48 00 44 00 57 00 2E 00 54 00 00 00 58 00 54 00 H.D.W...T...X.T.
00002620: 48 44 57 48 44 57 7E 31 54 58 54 20 00 41 D7 79 HDWHDW~1TXT .A.y
00002630: 53 59 53 59 00 00 D7 79 53 59 04 00 3B 00 00 00 SYSY...ySY..;...
00002640: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
*
000027f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....
```

Figure 22

发现FAT12 使用了多个目录项来存储文件名的不同部分：

- 短文件名（SFN）项：自动生成符合8.3 格式的短文件名 `HDWHDW~1.TXT`，其中出现有损转转换标识 `~`，以 `~1` 结尾

- 长文件名（LFN）项：长文件名通过LFN条目存储，每个LFN条目存储最多13个字符。

在使用LFN长文件名的系统中，会自动生成SFN短文件名以确保此文件在短文件名的文件系统中可使用。同时为了防止生成的短文件名冲突，SFN的生成采用名称+数字后缀+扩展的格式，同时采用以下规则生成SFN：

- ① 小写自动转大写
- ② 如果存在空格，则删去空格，并设置有损转换标识
- ③ 已 . 开头的文件删除头部的 .，并设置有损转换标识
- ④ 存在多个 . 的文件名，仅保留最后一个作为文件名与扩展的分隔，并设置有损转换标识
- ⑤ 其他不支持的字符，采用 \_ 替代，并设置有损转换标识
- ⑥ 文件名如果是Unicode编码，则转化为ANSI/OEM编码；不能转换的字符采用 \_ 替代，并设置有损转换标识
- ⑦ 长度超过8字节的部分，截断，并设置有损转换标识
- ⑧ 扩展名字段超过3字节的，截断，并设置有损转换标识

## ✿ 四、实验结果总结

- ① (对实验结果进行分析，完成思考题目，并提出实验的改进意见)

### 4.1 思考题

#### 4.1.1 FAT12格式是怎样的？

FAT12是一种经典的文件系统，自DOS时代起广泛使用，尤其在软盘（如1.44MB软盘）中仍然保持使用。它通过将磁盘划分为多个层次来便于数据的组织与管理，这些层次包括：

- ① 扇区（Sector）：磁盘上的最小数据单元，通常为512字节。

② 簇 (Cluster) : 一个或多个扇区的组合, 用于更有效地管理空间。

③ 分区 (Partition) : 通常指整个文件系统的划分。

当软盘以 **FAT12** 格式组织格式化后将会以如下标准设定:

- 两个磁头;
- 每个磁头有 **80** 个磁道;
- 每个磁道有 **18** 个扇区;
- 每个扇区大小为 **512** 字节。

标准 **FAT12** 软盘空间:  $2 * 80 * 18 * 512 = 1474560\text{B} = 1440\text{KB} = 1.44\text{MB}$

因此一个标准的 **1.44MB** 大小的 **FAT12** 格式软盘共有  $2 * 80 * 18 = 2880$  个扇区。这 **2880** 个扇区被分为 **5** 个部分, 如下:



图 1 软盘 (1.44MB, FAT12)

Figure 23

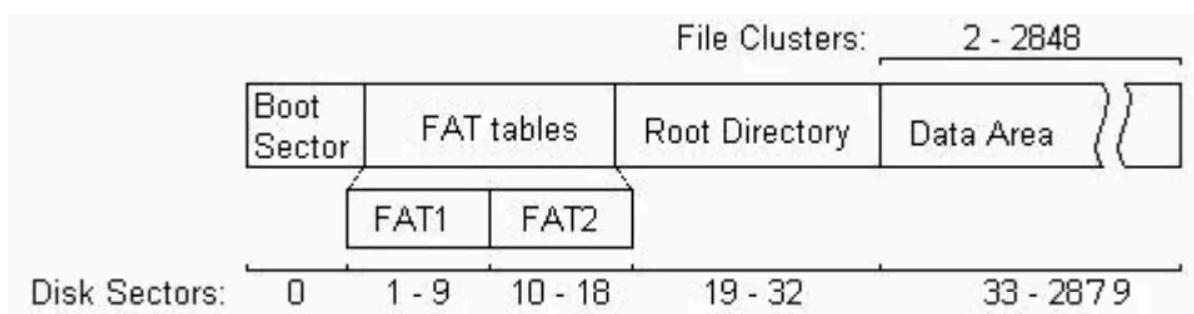


Figure 24

#### 4.1.1.1 引导扇区

引导扇区是整个软盘的第一个扇区，包含了重要的数据结构，如 **BPB ( BIOS Parameter Block )**。引导扇区的格式包括多个以 BPB\_ 开头的字段，这些字段用于描述文件系统的特性和参数，记录了整个文件系统的组织结构信息和引导程序两部分内容。

|                |        |                                 |                |
|----------------|--------|---------------------------------|----------------|
|                |        |                                 | jmp LABEL_STAR |
| BS_jmpBoot     | 0 3    | 一个跳转指令                          | T              |
|                |        |                                 | nop            |
| BS_OEMName     | 3 8    | 厂商名                             | 'ForrestY'     |
| BPB_BytPerSec  | 11 2   | 每扇区字节数                          | 0x200          |
| BPB_SecPerClus | 13 1   | 每簇扇区数                           | 0x1            |
| BPB_RsvdSecCnt | 14 2   | Boot记录占用多少扇区<br>t               | 0x1            |
| BPB_NumFATs    | 16 1   | 共有多少FAT表                        | 0x2            |
| BPB_RootEntCnt | 17 2   | 根目录文件数最大值                       | 0xE0           |
| BPB_TotSec16   | 19 2   | 扇区总数                            | 0xB40          |
| BPB_Media      | 21 1   | 介质描述符                           | 0xF0           |
| BPB_FATSz16    | 22 2   | 每FAT扇区数                         | 0x9            |
| BPB_SecPerTrk  | 24 2   | 每磁道扇区数                          | 0x12           |
| BPB_Numheads   | 26 2   | 磁头数(面数)                         | 0x2            |
| BPB_HiddSec    | 28 4   | 隐藏扇区数                           | 0              |
| BPB_TotSec32   | 32 4   | 如果BPB_TotSec16是0, 由这个值记录扇区<br>数 | 0              |
| BS_DrvNum      | 36 1   | 中断13的驱动器号                       | 0              |
| BS_Reserved1   | 37 1   | 未使用                             | 0              |
| BS_BootSig     | 38 1   | 扩展引用标记(29h)                     | 0x29           |
| BS_VolID       | 39 4   | 卷序列号                            | 0              |
| BS_VolLab      | 43 11  | 卷标                              | 'OrangesS0.02' |
| BS_FileSysType | 54 8   | 文件系统类型                          | 'FAT12'        |
| 引导代码及其他        | 62 448 | 引导代码、数据及其他填充字符等                 | 引导代码(其余为0)     |
| 结束标志           | 510 2  | 0xAA55                          | 0xAA55         |

Figure 25

| 名称              | 偏移  | 长度  | 内容                         |
|-----------------|-----|-----|----------------------------|
| BS_jmpBoot      | 0   | 3   | 跳转指令                       |
| BS_OEMName      | 3   | 8   | 生产厂商名                      |
| BPB_BytesPreSec | 11  | 2   | 每扇区字节数                     |
| BPB_SecPreClus  | 13  | 1   | 每簇扇区数                      |
| BPB_RsvSecCnt   | 14  | 2   | 保留扇区数                      |
| BPB_NumFATs     | 16  | 1   | FAT 表的份数                   |
| BPB_RootEntCnt  | 17  | 2   | 根目录可容纳的目录项数                |
| BPB_TotSec16    | 19  | 2   | 总扇区数                       |
| BPB_Media       | 21  | 1   | 介质描述符                      |
| BPB_FATSz16     | 22  | 2   | 每 FAT 扇区数                  |
| BPB_SecPreTrk   | 24  | 2   | 每磁道扇区数                     |
| BPB_NumHeads    | 26  | 2   | 磁头数                        |
| BPB_HiddSec     | 28  | 4   | 隐藏扇区数                      |
| BPB_TotSec32    | 32  | 4   | 若 BPB_Tot16 为 0，则由这个值记录扇区数 |
| BS_DrvNum       | 36  | 1   | int 13h 的驱动器号              |
| BS_Reserved1    | 37  | 1   | 未使用                        |
| BS_BootSig      | 38  | 1   | 扩展引导标记 (0x29)              |
| BS_VolID        | 39  | 4   | 卷序列号                       |
| BS_VolLab       | 43  | 11  | 卷标                         |
| BS_FileSysType  | 54  | 8   | 文件系统类型                     |
| 引导代码            | 62  | 448 | 引导代码、数据及其他信息               |
| 结束标志            | 510 | 2   | 结束标志 0xAA55                |

Table 5

引导代码：

- BIOS在启动时将引导代码读取到0x7C00 -0x7DFF处
- 然后跳转到0x7C00处继续执行指令（引导代码）
- 开始执行该引导程序段，其主要功能是完成操作系统的自举并将控制权交给操作系统

#### 4.1.1.2 FAT 表

FAT 表（File Allocation Table）用于跟踪文件在数据区中所占用的簇，表项的值存放的是逻辑簇号。每个 FAT 项表示一个簇的状态，从第二个 FAT 项开始，代表数据区中的每一个簇。FAT 项的值可以指向下一个簇的簇号，或者指示当前簇是最后一个簇（值大于等于 0xFF8）。具体的含义分为以下几类：

| Value           | Meaning                                |
|-----------------|----------------------------------------|
| 0x00            | Unused                                 |
| 0xFF0-0xFF6     | Reserved cluster                       |
| 0xFF7           | Bad cluster                            |
| 0xFF8-0xFFFF    | Last cluster in a file                 |
| (anything else) | Number of the next cluster in the file |

Figure 26

- 0x000：可用簇
- 0x002 ~ 0xFE0：已用簇标识下一个簇的簇号
- 0xFF0 ~ 0xFF6：保留簇
- 0xFF7：坏簇
- 0xFF8 ~ 0xFFFF：文件的最后一个簇

**FAT** 表项的取值如下：

| FAT 项 | 可取值       | 描述                           |
|-------|-----------|------------------------------|
| 0     | BPB_Media | 磁盘标识字，低字节需与 BPB_Media 数值保持一致 |
| 1     | FFFh      | 表示第一个簇已占用                    |
|       | 000h      | 可用簇                          |
|       | 002h~FEFh | 已用簇                          |
| 2 ~ N | FF0h~FF6h | 保留簇                          |
|       | FF7h      | 坏簇                           |
|       | FF8h~FFFh | 文件的最后一个簇                     |

[注]: FAT[0] 和 FAT[1] 始终不作为数据区的索引使用。

当文件的体积较大时，通常会被分成若干个片段，分散存储在磁盘扇区。FAT 表项用于指示文件在多个簇中的分布，按簇号链接起来。

### FAT的读取规则

- ① 大小端：FAT文件系统是为IBM PC机设计的，字节存储采用的是小端格式（little-endian），即最低有效字节（Least Significant Byte, LSB）放在内存的低地址，举例，一个数 **0x12345678**，最低有效字节为 **0x78**。

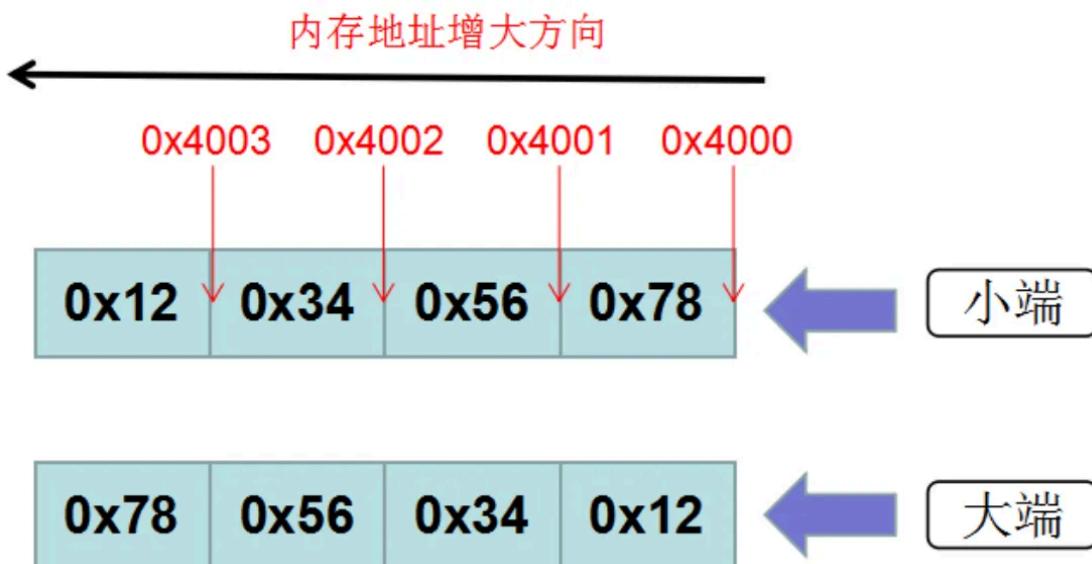


Figure 27

- ② **12位与8位转换**: 在 FAT12 文件系统中, 每个 FAT 表项 (FAT Entry) 占用 12 位 (1.5 字节)。由于每个 FAT 表项不是整字节对齐的, 因此需要跨字节存储。具体来说, 连续的 3 个字节包含了 2 个 FAT 表项。以下是详细的存储方式:

假设有连续的 3 个字节, 分别为 BYTE1、BYTE2 和 BYTE3。这 3 个字节包含了两个 FAT 表项 (FAT Entry1 和 FAT Entry2), 具体存储方式如下:

- **FAT Entry1**: 由 BYTE1 和 BYTE2 的低 4 位组成。
- **FAT Entry2**: 由 BYTE2 的高 4 位和 BYTE3 组成

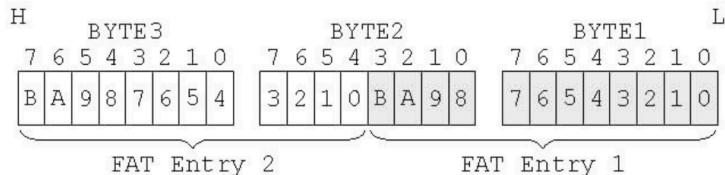


图4.2 从FAT表中得到一个FAT项的值

Figure 28

#### 4.1.1.3 根目录区

在 FAT12 文件系统中, 根目录区紧接在第二个 FAT 表之后, 根目录的起始扇区号为 **19**。根目录区包含多个目录项, 每个目录项占用 **32** 字节, 这些目录项存储了文件的名称、属性、大小、日期及其在磁盘上的位置。

根目录区的大小取决于 **BPB\_RootEntCnt**, 即 根目录可容纳的目录项数。每个目录项占用 32 字节, 根目录区最多可以容纳 **BPB\_RootEntCnt** 个目录项。在计算数据区扇区位置时, 主要用到两个参数: **BPB\_RootEntCnt** 和 **BPB\_BytsPerSec** (每个扇区的字节数)。

根目录区占用的扇区数可以通过以下公式计算:

$$RootDirSectors = \frac{BPB\_RootEntCnt \times 32 + (BPB\_BytsPerSec - 1)}{BPB\_BytsPerSec}$$

公式解释:

- **BPB\_RootEntCnt**: 根目录中目录项的数量 (即目录项数目)。
- 每个目录项的大小为 **32** 字节。
- **BPB\_BytsPerSec**: 每个扇区的字节数 (一般为 512 字节)。
- **BPB\_BytsPerSec - 1**: 用于实现向上取整。

根据这个公式，我们可以计算出根目录区占用的总扇区数。由于每个扇区通常有 512 字节，而每个目录项占用 32 字节，一个扇区可以存储 16 个目录项。

一般来说，根目录区的起始扇区号为 19，共占用 14 个扇区（扇区号 19 - 32），用于存放根目录区中的文件或目录。因此，FAT12 文件系统最多只能容纳  $14 \times 16 = 224$  个文件或目录。

根目录项：

每个目录项表示一个文件控制块，32字节。文件控制块示意图如下：

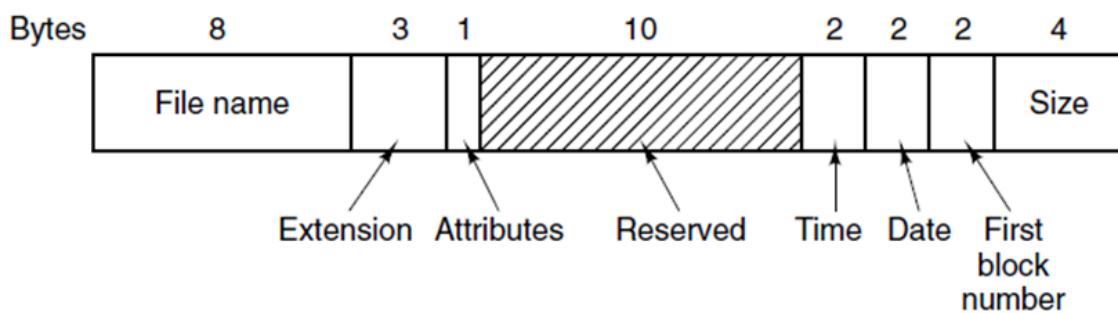


Figure 29

书上的字段长度及含义如下：

| 名称           | 偏移 | 长度 | 内容             |
|--------------|----|----|----------------|
| DIR_Name     | 0  | 11 | 文件名 8B, 扩展名 3B |
| DIR_Attr     | 11 | 1  | 文件属性           |
| Reserved     | 12 | 10 | 保留位            |
| DIR_WrtTime  | 22 | 2  | 最后一次写入时间       |
| DIR_WrtDate  | 24 | 2  | 最后一次写入日期       |
| DIR_FstClus  | 26 | 2  | 起始簇号           |
| DIR_FileSize | 28 | 4  | 文件大小           |

Table 6

与上述略有不同，官网查到的如下：

| Offset (in bytes) | Length (in bytes) | Description                                                       |
|-------------------|-------------------|-------------------------------------------------------------------|
| 0                 | 8                 | Filename (but see notes below about the first byte in this field) |
| 8                 | 3                 | Extension                                                         |
| 11                | 1                 | Attributes (see details below)                                    |
| 12                | 2                 | Reserved                                                          |
| 14                | 2                 | Creation Time                                                     |
| 16                | 2                 | Creation Date                                                     |
| 18                | 2                 | Last Access Date                                                  |
| 20                | 2                 | Ignore in FAT12                                                   |
| 22                | 2                 | Last Write Time                                                   |
| 24                | 2                 | Last Write Date                                                   |
| 26                | 2                 | First Logical Cluster                                             |
| 28                | 4                 | File Size (in bytes)                                              |

Figure 30

实际上，在实验过程中 **0D** 位并非总是像 **0C** 位作为保留位一直为 **00**，再区分细一点如下：

- 短文件名的存储格式：

| 偏移   | 大小 | 描述               |
|------|----|------------------|
| 0x00 | 11 | 8.3格式的文件名和文件扩展名。 |
| 0x0B | 1  | 文件属性             |
| 0x0C | 1  | 保留不用             |
| 0x0D | 1  | 文件创建时间毫秒数的10倍    |
| 0x0E | 2  | 文件创建时间           |
| 0x10 | 2  | 文件创建日期           |
| 0x12 | 2  | 最后访问日期           |
| 0x14 | 2  | 文件首簇号高16位        |
| 0x16 | 2  | 最后写操作的时间         |
| 0x18 | 2  | 最后写操作的日期         |
| 0x1A | 2  | 首簇号的低16位         |
| 0x1C | 4  | 文件大小             |

Table 7

- 长文件名的存储格式:

| 偏移   | 大小 | 描述             |
|------|----|----------------|
| 0x00 | 1  | 长文件名目录项的顺序     |
| 0x01 | 10 | 长文件名的第一个部分     |
| 0x0B | 1  | 属性，值必须为0x0F    |
| 0x0C | 1  | 类型，必须为0        |
| 0x0D | 1  | 校验和            |
| 0x0E | 12 | 长文件名的第二个部分     |
| 0x1A | 2  | 文件首簇号低16位，必须为0 |
| 0x1C | 4  | 长文件名的第三个部分     |

Table 8

重要字段：

① 文件名和文件扩展名：

在 FAT12 文件系统中，文件名和文件扩展名均为短文件名（8.3 格式）。这意味着文件名最多为 **8** 个字符，扩展名最多为 **3** 个字符，且不包括表示文件扩展名的句点 “.”。

- 文件名不足 **8** 字节 的会使用空格填充，超过 **8** 字节 的文件名会使用长文件名（LFN）的方法来存储。
- 扩展名最多为 3 个字符。
- FAT12 不区分文件名的大小写，总是以大写的形式存在。
- 如果文件名的第一个字节是 **0xE5**，表示该目录项是空闲的，未被使用。
- 如果文件名的第一个字节是 **0x00**，表示该目录项及后续的目录都是空闲的

② 文件属性：

该字节表示文件的属性信息。文件属性字节的每个位（bit）表示不同的文件属性。文件属性字节的结构如下所示：

| <b>Bit</b> | <b>Mask</b> | <b>Attribute</b> |
|------------|-------------|------------------|
| 0          | 0x01        | Read-only        |
| 1          | 0x02        | Hidden           |
| 2          | 0x04        | System           |
| 3          | 0x08        | Volume label     |
| 4          | 0x10        | Subdirectory     |
| 5          | 0x20        | Archive          |
| 6          | 0x40        | Unused           |
| 7          | 0x80        | Unused           |

Figure 31

| 位   | 含义                 | 说明                             |
|-----|--------------------|--------------------------------|
| 位 0 | 只读位，值为 1 表示只读文件    | 如果设置为 1，表示文件是只读的，不能被修改。        |
| 位 1 | 隐藏位，值为 1 表示隐藏文件    | 如果设置为 1，表示该文件是隐藏文件，通常用户不可见。    |
| 位 2 | 系统文件，值为 1 表示系统文件   | 如果设置为 1，表示该文件是操作系统的系统文件。       |
| 位 3 | 卷标，值为 1 表示目录项是卷标   | 如果设置为 1，该目录项的文件名部分表示卷标，而不是文件名。 |
| 位 4 | 子目录，值为 1 表示目录项是子目录 | 如果设置为 1，该目录项代表一个文件夹，而不是普通文件。   |
| 位 5 | 存档，值为 1 表示文件未被备份   | 表示该文件自上次备份以来被修改过。通常用于增量备份的标识。  |
| 位 6 | 保留位，值为 0           | 该位保留，不使用。                      |
| 位 7 | 保留位，值为 0           | 该位保留，不使用。                      |

Table 9

特别说明：

- 如果低 4 位（位 0 到位 3）值都是 1（即属性值为 0x0F），则该目录项为长文件名（LFN）目录项，用于存储长文件名。

#### 4.1.1.4 数据区

数据区是存放实际文件内容的区域。数据区的第一个簇号为 2，文件的数据从此位置开始。需要注意的是，数据区的开始扇区号可以通过计算根目录区所占的扇区数来确定。每个簇包含一个或多个扇区，因此对于大于 512 字节的文件，需要使用 FAT 表来找到所有的簇。

### 4.1.2 如何读取一张软盘的信息

- 在汇编程序中读取：

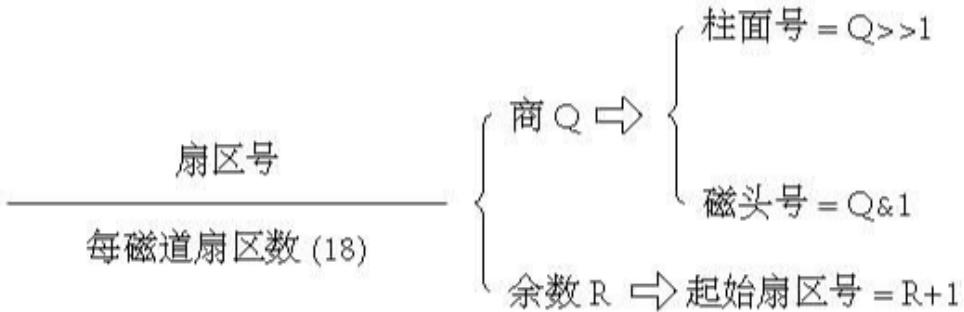
可以使用 `int 13h` 来读取软盘信息。BIOS中断 `int 13h` 通常用于加载扇区进入内存，功能号为2。

参数如下：

表4.4 BIOS中断`int 13h`的用法

| 中断号 | 寄存器          | 作用                                |
|-----|--------------|-----------------------------------|
| 13h | ah=00h       | d1=驱动器号 (0 表示 A 盘)                |
|     | ah=02h       | a1=要读扇区数                          |
|     | ch=柱面 (磁道) 号 | c1=起始扇区号                          |
|     | dh=磁头号       | d1=驱动器号 (0 表示 A 盘)<br>es:bx→数据缓冲区 |

使用 `int 13h` 读取扇区时需要将线性编号的扇区转化为 **(磁头号, 柱面号, 扇区号)** 的CHS寻址方式，对于1.44MB的软盘，总共有两面（磁头号0和1），因此磁头号、柱面磁道号和起始扇区号计算方式如下：



扇区内容最终将被写到 `ES:BX` 指向的缓冲区中。通过该方法可以得到软盘信息。

- 通过命令行查看：

使用xxd命令可以显示一个扇区的内容。使用 `man xxd` 可以查看使用手册：

```

xxd(1)                                     General Commands Manual                                     xxd(1)

NAME
    xxd - make a hexdump or do the reverse.

SYNOPSIS
    xxd [-h[elp]]
    xxd [options] [infile [outfile]]
    xxd -[revert] [options] [infile [outfile]]

DESCRIPTION
    xxd creates a hex dump of a given file or standard input. It can also convert a hex dump back to its original binary form. Like uuencode(1) and uudecode(1) it allows the transmission of binary data in a 'mail-safe' ASCII representation, but has the advantage of decoding to standard output. Moreover, it can be used to perform binary file patching.

OPTIONS
    If no infile is given, standard input is read. If infile is specified as a '-' character, then input is taken from standard input. If no outfile is given (or a '-' character is in its place), results are sent to standard output.

    Note that a "lazy" parser is used which does not check for more than the first option letter, unless the option is followed by a parameter. Spaces between a single option letter and its parameter are optional. Parameters to options can be specified in decimal, hexadecimal or octal notation. Thus -c8, -c 8, -c 010 and -cols 8 are all equivalent.

    -a | -autoskip
        toggle autoskip: A single '*' replaces nul-lines. Default off.

    -b | -bits
        switch to bits (binary digits) dump, rather than hexdump. This option writes octets as eight digits "1"s and "0"s instead of a normal hexadecimal dump. Each line is preceded by a line number in hexadecimal and followed by an ascii (or ebcDIC) representation. The command line switches -r, -p, -l do not work with this mode.

    -c cols | --cols cols
        format <cols> octets per line. Default 16 (-l: 12, -ps: 30, -b: 6). Max 256.

    -E | -EBCDIC
        change the character encoding in the righthand column from ASCII to EBCDIC. This does not change the hexadecimal representation. The option is meaningless in combinations with -r, -p or -l.

    -g bytes | -groupsize bytes
        separate the output of every <bytes> bytes (two hex characters or eight bit-digits each) by a whitespace. Specify -g 0 to suppress grouping. <Bytes> defaults to 2 in normal mode and 1 in bits mode. Grouping does not apply to postscript or include style.

    -h | -help
        print a summary of available commands and exit. No hex dumping is performed.

    -i | -include
        output in C include file style. A complete static array definition is written (named after the input file), unless xxd reads from stdin.

Manual page xxd(1) line 1 (press h for help or q to quit)

```

指令的通用格式为：`xxd [options] [infile [outfile]]`， outfile表示内容输出到哪个文件，当不指定时显示至屏幕上。

以a.img磁盘映像文件为例，查看0x2600开始的512字节内容：

```

ubuntu-32@ubuntu:~/Desktop/osfs04/c$ xxd -u -a -g 1 -c 16 -s +0x2600 -l 512 a.img
0002600: 41 6C 00 6F 00 61 00 64 00 65 00 0F 00 AB 72 00 A.l.o.a.d.e....r.
0002610: 2E 00 62 00 69 00 6E 00 00 00 00 00 FF FF FF FF ..b.i.n.....
0002620: 4C 4F 41 44 45 52 20 20 42 49 4E 20 00 00 59 08 LOADER BIN ..Y.
0002630: 4E 59 4E 59 00 00 59 08 4E 59 03 00 0F 00 00 00 NYNY..Y.NY.....
0002640: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 .....*
*00027f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
ubuntu-32@ubuntu:~/Desktop/osfs04/c$ 

```

- 除了上述两种课本上介绍的方法外，还可以采用 `ls` 和 `df` 指令读取软盘信息。

使用 `sudo mount -o loop a.img /mnt/floppy` 挂载软盘，然后使用 `ls /mnt/floppy` 查看软盘内容，也可以使用 `df -h /mnt/floppy` 来查看文件系统的信息，最后用 `sudo umount /mnt/floppy` 卸载软盘。

### 4.1.3 如何在软盘中找到指定的文件

在软盘中找到指定的文件需要以下几个步骤：

- 查看引导扇区，查询根目录最大条目数、每个扇区的字节数、每磁道扇区数等信息。这些信息将会在后续计算过程中使用，如将线性编号的扇区号转化为CHS寻址需要使用每磁道扇区数。
- 查看根目录。根目录通常位于19号扇区。对每个目录条目进行搜索，直到该条目的文件名为指定文件。读取该条目得到起始簇号。
- 搜索FAT表。FAT1位于1号扇区，根据起始簇号找到第一个簇对应的FAT项，根据其内容找到第二个FAT项，直至找到最后一簇（FAT项为0xFFFF）。

- 获取文件数据。将簇号转化为在磁盘中的扇区号，使用 `int 13h` 指令将文件数据加载入内存。
- 此外，如果需要寻找的文件在子目录下，则按照文件路径依次定位每一层目录的所在扇区，并在该扇区中寻找下一层目录的位置，步骤与上述类似。

#### 4.1.4 如何在系统引导过程中，从读取并加载一个可执行文件到内存，并移交控制权？

读取和加载文件的过程如上所说。通过根目录找到首簇簇号，将首簇簇号对应的扇区的数据读入内存，再根据 FAT 表得到文件的簇链，然后使用 `int 13h` 将剩余扇区加载入内存。移交控制权的方法通过一个 `jmp` 指令实现，通过跳转到加载入内存的位置来移交控制权。

#### 4.1.5 为什么需要这个Loader程序不包含dos系统调用？

一个操作系统从开机到开始运行大致经历：引导扇区→控制权转移给 Loader→加载内核入内存→跳入保护模式→开始执行内核的过程。

因此，Loader 程序主要用于加载操作系统内核，跳入保护模式，然后移交控制权给操作系统内核。当 Loader 程序在执行时，操作系统还没有被加载，因此没有可用的系统调用。DOS 也是一个操作系统，Loader 程序必须直接与硬件交互，而不是依赖于操作系统提供的功能。DOS 系统调用需要 DOS 操作系统已经加载并运行。

#### 4.1.6 为什么在前面几个章节中 `a.img`，不能直接 `mount`，在本章代码里面却可以？

`mount` 指令是 Linux 操作系统中的一个关键命令，用于将文件系统挂载到目录树上，使用户能够访问存储设备（如硬盘分区）中的文件。

而在前面几个章节中，我们只是使用 `dd` 命令将 bin 文件写入引导扇区，并没有文件系统结构，因此不能 `mount`。

在本章代码中，我们在 `boot.bin` 中加入了 BPB 头信息，使得它可以被 Dos、Linux 识别，并向软盘镜像文件写入了 `boot.bin`。此时我们已经为这些文件创建了有效的文件系统结构，操作系统能够识别并理解文件系统结构，从而正确加载文件。因此，在本章代码中，这些文件可以直接 `mount`。

## 4.1.7 扩展提高：调研在硬盘上，文件系统格式为FAT32或者NTFS，应该怎么来实现类似功能呢？（可粗略参阅第9章）

### 7.1 FAT32

- FAT32文件系统的格式：
  - 引导扇区：包含文件系统的基本信息，如总大小、扇区大小、保留扇区数等。FAT32文件系统的DBR有5部分组成，分别为跳转指令，OEM代号，BPB，引导程序和结束标志。
  - 文件分配表（FAT）：记录文件占用的簇（cluster）信息，类似链表结构。与FAT12不同，每个FAT项占据4字节，以0xFFFFFFF表示文件的最后一簇。
  - 根目录区（Root Directory Area）：存储根目录的信息。一个目录项为32字节，字节偏移位置00-07H为文件名，14-15H为文件首簇高16位，1A-1BH为文件首簇低16位，1C-1FH为文件长度。
  - 数据区：存储实际文件数据，数据以簇为单位进行存储。由于FAT32使用32位地址，最大支持 $2^{28}$ 个簇，簇大小更大，可以处理更大的存储文件。
  - 通常也把根目录区看成数据区的一部分。数据区的内容主要由三部分组成：根目录，子目录和文件内容。数据区的2号簇被分配给根目录使用。
- 查找指定文件：

FAT32 与本实验的 FAT12 相比，最大的区别在于根目录不再是固定大小、固定区域，而是改成了根目录文件，是数据区的一部分。因此，文件查找方式应修改如下：

  - ① 定位引导扇区，获取BPB\_RootClus用于指示根目录的簇号。
  - ② 根据首簇号FAT表得到根目录的簇链，得到根目录的数据。
  - ③ 在根目录中通过目录项条目的文件名来匹配目标文件的名称。如果路径中包含子目录，则进入下一级目录继续查找，直到找到对应文件。
  - ④ 找到对应文件目录项后，获取其首簇号，在FAT表中根据首簇号得到完整的簇链。
  - ⑤ 根据簇链将簇号转化为扇区号，在软盘中读取。

- 加载Loader:

首先准备好引导设备。与 FAT12 不同，FAT32 文件系统通常存储在硬盘上而不是软盘上。准备一张硬盘映像文件，将 Loader 程序复制入硬盘，然后利用上述方法从文件系统中找到 Loader 后加载入内存然后将控制权转交给 Loader 程序。这通常是通过一个跳转指令或函数调用来实现的，将控制权传递给 Loader 程序的入口点。

## 7.2 NTFS

- NTFS文件系统的格式:

NTFS (New Technology File System)，是 WindowsNT 环境的文件系统。NTFS 的结构更加复杂，能够提供更高的灵活性和安全性。

在NTFS文件系统中，磁盘上的所有数据都是以文件的形式存储，其中包括元文件。NTFS将文件作为属性/属性值的集合来处理，这一点与其他文件系统不一样。文件数据就是未命名属性的值，其他文件属性包括文件名、文件拥有者、文件时间标记等。

- 引导扇区（Boot）：包含跳转指令、OEM代号、BPB、引导程序、结束标志。
- 主文件表（MFT）：MFT是每个文件的索引。每个文件都有一个或多个文件记录，每个文件记录占用两个扇区，而MFT元文件就是专门记录每个文件的文件记录。文件系统通过MFT来确定文件在磁盘上的位置以及文件的属性。MFT的前16个文件记录总是元文件的，并且顺序是固定不变的。这些元文件为：

| 序 号 | 元 文 件                       | 功 能                           |
|-----|-----------------------------|-------------------------------|
| 0   | \$MFT                       | 主文件表本身，是每个文件的索引               |
| 1   | \$MFTMirr                   | 主文件表的部分镜像                     |
| 2   | \$LogFile                   | 事务型日志文件                       |
| 3   | \$Volume                    | 卷文件，记录卷标等信息                   |
| 4   | \$AttrDef                   | 属性定义列表文件                      |
| 5   | \$Root                      | 根目录文件，管理根目录                   |
| 6   | \$Bitmap                    | 位图文件，记录了分区中簇的使用情况             |
| 7   | \$Boot                      | 引导文件，记录了用于系统引导的数据情况           |
| 8   | \$BadClus                   | 坏簇列表文件                        |
| 9   | \$Quota (NTFS4)             | 在早期的 Windows NT 系统中此文件为磁盘配额信息 |
| 10  | \$Secure                    | 安全文件                          |
| 11  | \$UpCase                    | 大小写字符转换表文件                    |
| 12  | \$Extend metadata directory | 扩展元数据目录                       |
| 13  | \$Extend\\$Reparse          | 重解析点文件                        |
| 14  | \$Extend\\$UsnJrn           | 加密日志文件                        |
| 15  | \$Extend\\$Quota            | 配额管理文件                        |
| 16  | \$Extend\\$ObjId            | 对象 ID 文件                      |

在元文件后是数据文件的文件记录。文件记录的格式如下：

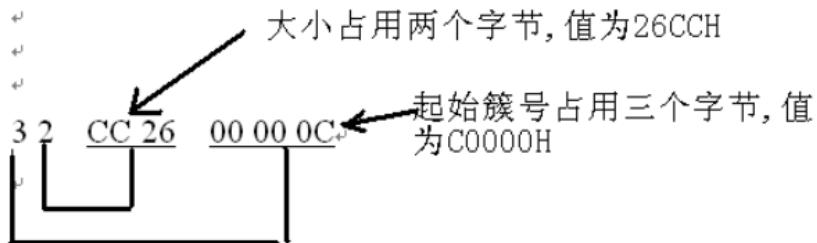
- 文件记录：文件记录由两部分构成，一部分是文件记录头，另一部分是属性列表，最后结尾是FFFFFF。在同一系统中，文件记录头的长度和具体偏移位置的数据含义是不变的，而属性列表是可变的，其不同的属性有着不同的含义。文件记录头的格式如下：

| 偏移   | 长度   | 描述                                                                        |
|------|------|---------------------------------------------------------------------------|
| 0X0  | 4    | 固定值，一定是“FILE”                                                             |
| 0X4  | 2    | 更新序列号的偏移                                                                  |
| 0X6  | 2    | 更新序列号与更新数组以字为单位大小 (S)                                                     |
| 0X8  | 8    | 日志文件序列号（每次记录被修改，都将导致该序列号加 1）                                              |
| 0X10 | 2    | 序列号（用于记录本文件记录被重复使用的次数，每次文件删除时加 1，跳过 0 值，如果为 0，则保持为 0）                     |
| 0X12 | 2    | 硬连接数，只出现在基本文件记录中，目录所含项数要使用到它                                              |
| 0X14 | 2    | 第一个属性流的偏移地址                                                               |
| 0X16 | 2    | 标志字节，1 表示记录使用中，2 表示该记录为目录                                                 |
| 0X18 | 4    | 文件记录实际大小（填充到 8 字节，即以 8 字节为边界）                                             |
| 0X1C | 4    | 文件记录分配大小（填充到 8 字节，即以 8 字节为边界）                                             |
| 0X20 | 8    | 所对应的基本文件记录的文件参考号（扩展文件记录中使用，基本文件记录中为 0，在基本文件记录的属性列表 0X20 属性存储中扩展文件记录的相关信息） |
| 0X28 | 2    | 下一个自由 ID 号，当增加新的属性时，将该值分配给新属性，然后该值增加，如果 MFT 记录重新使用，则将它置 0，第一个实例总是 0       |
| 0X2A | 2    | 边界，WINDOWS XP 中使用，也就是本记录使用的两个扇区的最后两个字节的值                                  |
| 0X2C | 4    | WINDOWS XP 中使用，本 MFT 记录号                                                  |
|      | 2    | 更新序列号                                                                     |
|      | 2S-2 | 更新序列数组                                                                    |

属性是指NTFS文件系统中所有与文件相关得数据结构，包括文件的内容。每个文件记录中都有多个属性，它们相对独立，有各自的类型和名称。每个属性都由两部分组成：属性头和属性体。属性头包含了该属性的重要信息，如属性类型，属性大小，名字及是否为常驻属性等。属性头的格式通常如下：

| 偏移(16进制) | 长度     | 常用值(16进制) | 含义                  |     |
|----------|--------|-----------|---------------------|-----|
| 00-03    | 4      |           | 属性类型                |     |
| 04-07    | 4      |           | 属性的长度, 8的整数倍 整个属性长度 |     |
| 08       | 1      | 00        | 是否为常驻属性, 00表示为常驻    |     |
| 09       | 1      | 00        | 属性名的长度, 00表示没有属性名   |     |
| 0A       | 2      | 1800      | 属性值开始的偏移            |     |
| 0C-0D    | 2      | 00        | 标志,                 |     |
| 0E-0F    | 2      | 00        | 标识                  |     |
| 10-13    | 4      | Length    | 属性长度                | 属性体 |
| 14-15    | 2      | 18        | 属性体开始位置             |     |
| 16       | 1      |           | 索引标志                |     |
| 17       | 1      |           | 填充                  |     |
| 18       | Length |           | 属性体开始               |     |

属性体存放属性的具体值。属性的种类有很多，因此各属性体的含义也不同。当属性类型为80H时，该属性为文件的数据内容。当该属性的空间不能存放完数据，系统就会在NTFS数据区域开辟一个空间存放，这个区域是以簇为单位的，并在属性结尾用runlist记录这个数据区域的起始簇号和大小。runlist记录这个数据区域的起始簇号和大小，高4位表示起始簇号占用多少个字节，低位4表示大小占用的字节数。例如：



runlist将以一个00字节结尾，表示结束。

- 数据区：存储文件数据，可以是分散的，支持压缩和加密。
- 日志区：用于记录文件系统的变更，提高数据完整性。
- 查找指定文件：
  - ① 定位引导扇区，获取MFT的起始簇号及簇的大小。
  - ② 找到MFT，在其中寻找根目录的文件记录。NTFS的根目录像其他文件一样存储在MFT中，具有固定的记录位置。
  - ③ 通过MFT条目的文件名属性来匹配目标文件的名称。如果路径中包含子目录，则进入下一级目录继续查找，直到完整匹配给定的文件路径。每个MFT条目可以使用索引属性来加速文件查找过程。NTFS使用B+树索引结构对文件进行排序和管理，因此文件查找更为高效。

- ④ 通过属性中的RunList定位到其数据流，查找到文件所有的数据簇，并返回文件的内容。
- 加载Loader：

首先准备好引导设备。与 FAT12 不同，NTFS 文件系统通常存储在硬盘上而不是软盘上。准备一张硬盘映像文件，将 Loader 程序复制入硬盘，然后利用上述方法从文件系统中找到 Loader 后加载入内存然后将控制权转交给 Loader 程序。这通常是通过一个跳转指令来实现的，以此将控制权传递给 Loader 程序的入口点。

## 4.2 实验结果分析

- Loader.bin 文件成功加载：通过编译和挂载 Loader.bin，启动 bochs，在屏幕第一行中央观察到字符“L”，验证了 boot.bin 能正确在 FAT12 中找到 Loader.bin 并转交控制权。
- 二进制信息的正确读取：通过 `xxd` 命令读取 FAT 表、目录项和数据区，读取的结果与预期一致，验证了 `xxd` 命令可以用于查看和操作二进制文件的内容。

## 4.3 实验改进意见

- 长文件名的处理：在本实验中，我们以 Loader.bin 为例介绍了如何在软盘中找到指定的文件。但是根据前面的介绍，FAT12 文件系统中，一个根目录项只有 8 字节用于存储文件名，3 字节存储拓展名。我们可以进一步查阅资料，了解文件名长于 8 字节时是如何在软盘中存储的以及它的查找方式，实现更加完备的文件查找代码。
- 错误排除指南：实验可以提供学生在遇到常见问题时如何进行自我排查的指南，这可以包括常见错误输出的解释以及解决这些问题的步骤。

# ✿ 五、各人实验贡献与体会

- ① 程序：

- 承担任务：独立完成实验，完成思考题共七题以及实验改进建议部分。
- 个人体会：本次实验是Loader 程序的读取和加载，介绍了FAT12文件系统的格式，软盘信息的读取方式，文件的查找方式等。这次实验与软件安全实验的结合比较紧密，都涉及到FAT文件系统中指定文件的读取，因此代码阅读和上手难度较小。同时，通过本次实验，我对操作系统的启动过程，引导扇区、Loader和内核之间的联系有了更深入的理解，为后续进行内核的设计以及其他操作系统实验打下基础。

## ② 周业营：

- 承担任务：完成实验大部分内容，向软盘写入自己创建的文件并手动找到文件，调试执行载入内存的可执行文件。
- 个人体会：在手工定位文件的过程中，我对 FAT12 系统整体的结构进行了梳理，理清了构建 FAT12 文件系统的逻辑。结合FAT12结构和存储的信息，我学会了如何计算数据区开始扇区号，并总结出定位一个文件的一般步骤。同时，我进一步了解了通过bios 硬盘中断 int 13读取磁盘信息的使用方法、使用汇编代码读取 FAT12 文件系统中的文件的方法。通过创建 Loader 程序，我学会了如何将一个可执行文件加载到内存中，并将控制权转交给它，这是操作系统启动的关键步骤，在这个步骤中我也深刻体会到了操作系统的功能和启动流程。除此之外，我还熟悉了二进制编辑工具 xxd 的使用，为加载操作系统打下了坚实的基础。

## ③ 黄东威：

- 承担任务：在手工定位文件中，我对FAT12系统各部分的结构有了一个系统的梳理，能够理清其中的逻辑，并总结出定位文件的一般步骤。在3.2 中，我了解了bios硬盘中断 的使用功能方法、使用汇编代码读取FAT12中文件的方法，成功将Loader程序读入了内存，为加载操作系统打下了坚实的基础。我手动的生成了flower.txt并将其挂载，然后使用分析FAT12文件的方法，找到了文件的开始与结束。
- 个人体会：通过本次实验，我深入地分析、理解并掌握了以下的内容：FAT12文件系统格式和工作方式、向软盘镜像文件写入文件的方法、读取文件在磁盘中的信息的方法、学会使用xxd读取二进制信息、学会通过int13h读磁盘、读取其扇区信息、学会将指定文件装入指定内存区并执行……对其各种相关知识都有了一定程度的理解和自我的掌握；同时，对汇编代码的阅读和分析过程，也对我自己汇编的语法和代码编写的知识和能力带来的极大的锻炼和提升。

## ④ 王浚杰：

- 承担任务：在本次实验中，我独立完成了所有实验任务，包括对 FAT12 文件系统结构的分析、手动分析软盘镜像中的扇区信息以及实现装入 `Loader.bin` 的过程。首先，我学会了使用 `xxd` 工具解析软盘镜像中的根目录和 FAT 表，通过分析 FAT 表的工作机制，确定了文件的起始簇号和簇链。在理解了 FAT12 文件系统的基础上，我通过手动查找和读取文件数据，实现了对软盘文件信息的准确提取。为了将 `Loader.bin` 成功加载到内存中，我修改了 `boot.asm`，确保引导扇区能够正确寻找 `Loader.bin` 文件的起始簇号和簇链，并逐个读取相应的扇区。最终，我通过 `bochs` 虚拟机调试了引导扇区的执行过程，并验证了 `Loader.bin` 程序的成功加载与执行。整个实验过程中，我负责了各个调试步骤，并通过 `bochs` 的 `writemem` 指令提取二进制信息，使用 `xxd` 工具验证了文件的读取与加载情况。
- 个人体会：这次实验让我对 FAT12 文件系统有了更深入的理解，掌握了软盘中的数据组织与管理方式。通过手动分析软盘镜像中的文件扇区信息，计算簇链，我成功理清了对根目录、FAT 表和数据区三者之间的关系。此外，我熟练掌握了如何调试引导程序。在装载 `Loader.bin` 的过程中，我逐步理清了文件的加载过程，确保了引导程序能够顺利找到文件的起始簇号并读取簇链，从而将 `Loader.bin` 文件正确加载到内存中。在使用 `xxd` 工具分析二进制数据的过程中，我对于如何查看和验证底层文件结构有了实际的经验。这次实验不仅强化了我对 FAT12 文件系统的理解，也提升了我在使用工具进行调试和验证方面的能力。