

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	2022.10.17
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
李心杨	2020302181022	信息安全	2
王宇骥	2020302181008	信息安全	2
林锬扬	2020302181032	信息安全	2
郑炳捷	2020302181024	信息安全	2

1 实验目的及实验内容

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。

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。

2 实验环境及实验步骤

2.1 实验环境

- Ubuntu 16.04.1;
- VMWare Workstation 16 player;
- bochs 2.7。

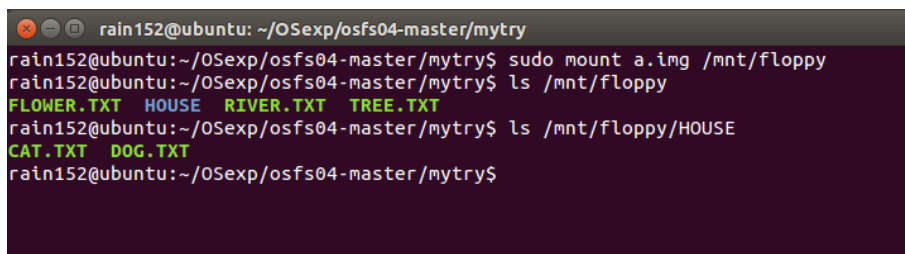
2.2 实验步骤

1. 按照参考教材创建软盘并且写入相应文件，手工分析扇区信息。
2. 加载 loader 入内存，记录原理与步骤。
3. 熟悉使用 xxd 命令，用前两步的实验进行验证。

3 实验过程分析

3.1 手工分析扇区信息

首先创建待分析的软盘 a.img。根目录下依次存放 FLOWER.TXT, TREE.TXT, RIVER.TXT, HOUSE 文件夹。HOUSE 文件夹下有 CAT.TXT 和 DOG.TXT。创建好后的软盘内容如下图所示：



```
rain152@ubuntu: ~/OSexp/osfs04-master/mytry
rain152@ubuntu:~/OSexp/osfs04-master/mytry$ sudo mount a.img /mnt/floppy
rain152@ubuntu:~/OSexp/osfs04-master/mytry$ ls /mnt/floppy
FLOWER.TXT  HOUSE  RIVER.TXT  TREE.TXT
rain152@ubuntu:~/OSexp/osfs04-master/mytry$ ls /mnt/floppy/HOUSE
CAT.TXT  DOG.TXT
rain152@ubuntu:~/OSexp/osfs04-master/mytry$
```

图 1: a.img 内容

在二进制查看器中查看该软盘的二进制信息。这里我们选择 010Editor。010Editor 支持盘符 Drive 的模版，可以辅助我们的人工分析。

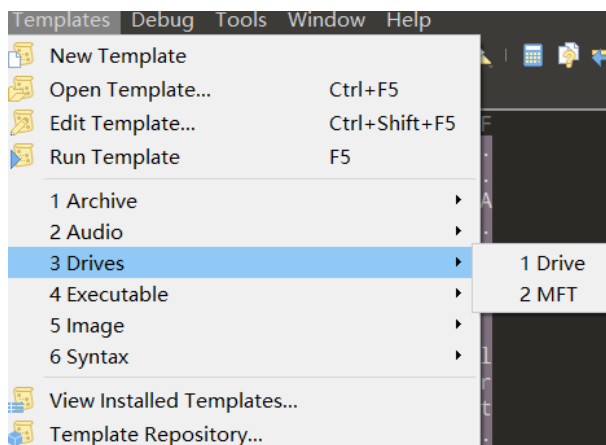


图 2: 010Editor 的模板

3.1.1 引导扇区分析

引导扇区在 0 号扇区，也就是第 0-1FFH 字节。a.img 的引导扇区信息如图3。

对照引导扇区结构，可以得到如下一些有用的信息：厂商名为 mkfs.fat（这与我们创建时采用的格式化操作相符）；每扇区的字节数是 200H（即 512 字节）；每簇有 1 个扇区；根目录可以容纳的目录项数为 E0H，进而算出根目录区占用 14 个扇区；FAT 表的数量为 2，每个 FAT 的扇区数是 9，FAT 总共占用 18 个扇区。所以数据区从第 33 个扇区开始。

在引导扇区的末尾，我们也看到了结束表示 0xAA55。

图 3: a.img 的引导扇区信息

FAT1 从第一个扇区开始，即从 0200H 字节开始。a.img 的相关信息如下：

图 4: a.img 的 FAT1 信息

在后面的定位文件步骤中，会结合 FAT 项进行分析。

根据 FAT 表的份数与每个 FAT 扇区数可以计算出根目录区从第 19 个扇区开始, 即从 2600H 字节开始。a.img 的根目录区的十六进制信息如下图所示:

2600h:	46 4C 4F 57 45 52 20 20	54 58 54 20 00 00 7D 28	FLOWER TXT ..}(
2610h:	4A 55 4A 55 00 00 7D 28	4A 55 03 00 08 07 00 00	JUJU..}(JU.....
2620h:	54 52 45 45 20 20 20 20	54 58 54 20 00 64 82 28	TREE TXT .d,(
2630h:	4A 55 4A 55 00 00 82 28	4A 55 07 00 0D 00 00 00	JUJU..,(JU.....
2640h:	52 49 56 45 52 20 20 20	54 58 54 20 00 64 85 28	RIVER TXT .d...(
2650h:	4A 55 4A 55 00 00 85 28	4A 55 08 00 11 00 00 00	JUJU.....(JU.....
2660h:	48 4F 55 53 45 20 20 20	20 20 20 10 00 00 BD 28	HOUSE ...½(
2670h:	4A 55 4A 55 00 00 BD 28	4A 55 09 00 00 00 00 00	JUJU..½(JU.....
2680h:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
2690h:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
26A0h:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

图 5: a.img 的根目录区信息

- 定位 TREE.TXT

对照表2，整理出根目录表中 TREE.TXT 的相关信息如下：

名称	内容
DIR_Name	TREE TXT
DIR_Attr	0x20
DIR_WrtTime	0x2882
DIR_WrtDate	0x554A
DIR_FstClus	0x07
DIR_FileSize	0x0D

时间的格式在微软关于 FAT 系统的白皮书中指定,我们不需要深究,通过 010Editor 的分析可以知道该文件最后一次的更新时间是 05:04:04 10/10/2022。属性值 0x20 表示它是一个文件。

从 DIR_FstClus 看出该文件的内容从 7 号簇开始。再去观察 FAT 表，发现第 7 个 FAT 项值为 FFF，可知 7 号簇就是文件最后一个簇，所以该文件只占用一个簇。计算可知 7 号簇对应第 38 个扇区，从 0x4C00 字节开始。跳转到对应位置，再根据文件大小为 13 字节读出文件内容：“treetreetree”。验证我们的推导是正确的。

4C00h:	74 72 65 65 74 72 65 65	74 72 65 65 0A 00 00 00	treetreetree....
4C10h:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
4C20h:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00
4C30h:	00 00 00 00 00 00 00 00	00 00 00 00 00 00 00 00

图 6: TREE.TXT 文件内容

- 定位 FLOWER.TXT

对照表2，整理出根目录表中 FLOWER.TXT 的相关信息如下：

名称	内容
DIR_Name	FLOWER TXT
DIR_Attr	0x20
DIR_WrtTime	0x287D
DIR_WrtDate	0x554A
DIR_FstClus	0x03
DIR_FileSize	0x0708

DIR_Attr 值为 0x20，表示该项仍然为文件。然后该文件的首簇是 3。查询 FAT 表，发现第 3 项值为 0x04，第 4 项值为 0x05，第 5 项值为 0x06，第 6 项值为 0x07。所以整个文件依次存放在 3、4、5、6 四个簇中。



图 7: FLOWER.TXT 的簇链

这四个簇分别对应第 34~37 号扇区。第 34 号扇区从 0x4400 字节开始。跳转到对应位置，依次读出 0x708 字节的内容，最终读取 300 遍 flower。

43F0h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
4400h:	66 6C 6F 77 65 72 66 6C 6F 77 65 72 66 6C 6F 77	flowerflowerflow
4410h:	65 72 66 6C 6F 77 65 72 66 6C 6F 77 65 72 66 6C	erflowerflowerfl
4420h:	6F 77 65 72 66 6C 6F 77 65 72 66 6C 6F 77 65 72	owerflowerflower
4430h:	66 6C 6F 77 65 72 66 6C 6F 77 65 72 66 6C 6F 77	flowerflowerflow
4440h:	65 72 66 6C 6F 77 65 72 66 6C 6F 77 65 72 66 6C	erflowerflowerfl

图 8: FLOWER.TXT 开始

4AA0h:	65 72 66 6C 6F 77 65 72 66 6C 6F 77 65 72 66 6C	erflowerflowerf
4AB0h:	6F 77 65 72 66 6C 6F 77 65 72 66 6C 6F 77 65 72	owerflowerflowe
4AC0h:	66 6C 6F 77 65 72 66 6C 6F 77 65 72 66 6C 6F 77	flowerflowerflo
4AD0h:	65 72 66 6C 6F 77 65 72 66 6C 6F 77 65 72 66 6C	erflowerflowerf
4AE0h:	6F 77 65 72 66 6C 6F 77 65 72 66 6C 6F 77 65 72	owerflowerflowe
4AF0h:	66 6C 6F 77 65 72 66 6C 6F 77 65 72 66 6C 6F 77	flowerflowerflo
4B00h:	65 72 66 6C 6F 77 65 72 00 00 00 00 00 00 00 00	erflower.....
4B10h:	00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00

图 9: FLOWER.TXT 结束

• 定位 HOUSE/CAT.TXT

前两个文件都是直接存放在根目录下。现在来探究 HOUSE/CAT.TXT 如何定位。首先在根目录表中找到 HOUSE 相关信息。

名称	内容
DIR_Name	HOUSE
DIR_Attr	0x10
DIR_WrtTime	0x28BD
DIR_WrtDate	0x554A
DIR_FstClus	0x09
DIR_FileSize	0x0

属性 0x10 表示该项是一个子目录。起始簇为 0x09。FAT 表中第 9 项值为 FFF，所以该子目录内容只占一个扇区。9 号簇对应 40 扇区，即 0x5000-0x5200 字节。跳转到相应位置，信息如下：

5000h:	2E 20 20 20	20 20 20 20	20 20 20 10	00 64 91 28	.	..d'('
5010h:	4A 55 4A 55	00 00 91 28	4A 55 09 00	00 00 00 00	JUJU..'(JU.....	
5020h:	2E 2E 20 20	20 20 20 20	20 20 20 10	00 64 91 28d'('
5030h:	4A 55 4A 55	00 00 91 28	4A 55 00 00	00 00 00 00	JUJU..'(JU.....	
5040h:	43 41 54 20	20 20 20 20	54 58 54 20	00 64 BA 28	CAT	TXT .d°('
5050h:	4A 55 4A 55	00 00 BA 28	4A 55 0A 00	0A 00 00 00	JUJU..°(JU.....	
5060h:	44 4F 47 20	20 20 20 20	54 58 54 20	00 00 BD 28	DOG	TXT ..½('
5070h:	4A 55 4A 55	00 00 BD 28	4A 55 0B 00	0A 00 00 00	JUJU..½(JU.....	
5080h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	

图 10: HOUSE 对应扇区的内容

在该扇区中，查找到 CAT.TXT 相关信息。仍然按照根目录区条目格式来分析该扇区的信息。结果如下：

名称	内容
DIR_Name	CAT TXT
DIR_Attr	0x20
DIR_WrtTime	0x28BA
DIR_WrtDate	0x554A
DIR_FstClus	0x0A
DIR_FileSize	0x0A

故 CAT.TXT 存放在第 10 号簇中，即 41 扇区的位置。0x5200 字节处可以查看到该文件的信息，再结合文件长度得到文件内容为“catcatcat”。

51F0h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
5200h:	63 61 74 63	61 74 63 61	74 0A 00 00	00 00 00 00	catcatcat.....
5210h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
5220h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
5230h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

图 11: CAT.TXT 文件内容

通过对上述几个文件的定位，基本上涉及到 FAT12 系统中定位文件内容的细节，包括首簇的定位、簇链的形成、子目录的处理等。

3.2 加载 loader 入内存并执行

3.2.1 加载 loader 入内存

将文件加载到内存要读取软盘，要用到 BIOS 的 13h 号中断，其功能如下：

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	复位硬盘控制器
15h	获取驱动器类型
16h	获取软驱中盘片的状态

其他参数如下：

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

可以看出是从 DL 号驱动器的 DH 磁头 CH 柱面 CL 扇区开始连续读取 AL 个扇区并存入 ES:BX 指向的缓冲区中。软盘有 2 个磁头、80 个柱面，每个柱面有 18 个扇区，所以对于我们的模拟软盘可以通过如下方式求得各项参数：

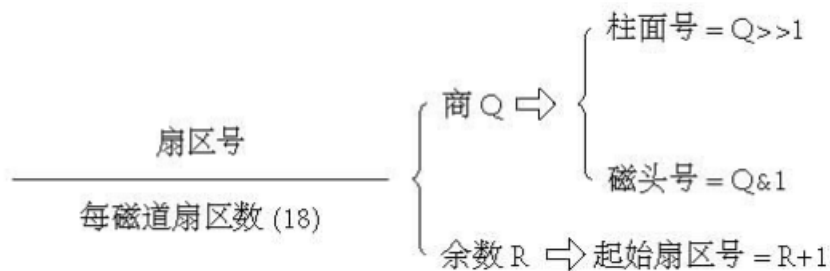


图 12: 参数的计算方法

我们可以依此写出读扇区函数：

```

1 ReadSector:
2     ; -----
3     ; 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)
4     ; -----
5     ; 设扇区号为 x
6     ;                               柱面号 = y >> 1
7     ;           x           商 y
8     ; ----- =>           磁头号 = y & 1
9     ; 每磁道扇区数
10    ;           余 z => 起始扇区号 = z + 1
11    push    bp
12    mov bp, sp
13    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]
14
15    mov byte [bp-2], cl
16    push    bx ; 保存 bx
17    mov bl, [BPB_SecPerTrk] ; bl: 除数
18    div bl ; y 在 al 中, z 在 ah 中
19    inc ah ; z ++
20    mov cl, ah ; cl <- 起始扇区号
21    mov dh, al ; dh <- y
22    shr al, 1 ; y >> 1 (y/BPB_NumHeads)
23    mov ch, al ; ch <- 柱面号
24    and dh, 1 ; dh & 1 = 磁头号
25    pop bx ; 恢复 bx
26    ; 至此, "柱面号, 起始扇区, 磁头号" 全部得到
27    mov dl, [BS_DrvNum] ; 驱动器号 (0 表示 A 盘)
28    .GoOnReading:
29    mov ah, 2 ; 读
30    mov al, byte [bp-2] ; 读 al 个扇区
31    int 13h
32    jc .GoOnReading ; 如果读取错误 CF 会被置为 1,
33    ; 这时就不停地读, 直到正确为止
34    add esp, 2
35    pop bp
36
37    ret

```

为了将 loader 读到内存中，我们需要知道 loader 的起始簇号（在根目录中）和簇

链（在文件分区表中），所以我们要遍历根目录来找到 loader 的目录项来确定起始簇号，再带着起始簇号在 FAT 中得到簇链，从而可以将 loader 读入内存。

在根目录中寻找 loader：

```

1      xor ah, ah ;
2      xor dl, dl ; 软驱复位
3      int 13h ;
4
5      ; 下面在 A 盘的根目录寻找 LOADER.BIN
6      mov word [wSectorNo], SectorNoOfRootDirectory
7  LABEL_SEARCH_IN_ROOT_DIR_BEGIN:
8      cmp word [wRootDirSizeForLoop], 0 ;
9      jz LABEL_NO_LOADERBIN ; 判断根目录区是不是已经读完
10     dec word [wRootDirSizeForLoop] ; 如果读完表示没有找到 LOADER.BIN
11     mov ax, BaseOfLoader
12     mov es, ax ; es <- BaseOfLoader
13     mov bx, OffsetOfLoader ; bx <- OffsetOfLoader 于是, es:bx = BaseOfLoader:
        OffsetOfLoader
14     mov ax, [wSectorNo] ; ax <- Root Directory 中的某 Sector 号
15     mov cl, 1
16     call ReadSector
17
18     mov si, LoaderFileName ; ds:si -> "LOADER BIN"
19     mov di, OffsetOfLoader ; es:di -> BaseOfLoader:0100 = BaseOfLoader*10h+100
20     cld
21     mov dx, 10h
22  LABEL_SEARCH_FOR_LOADERBIN:
23     cmp dx, 0 ; 循环次数控制,
24     jz LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR ; 如果已经读完了一个 Sector,
25     dec dx ; 就跳到下一个 Sector
26     mov cx, 11
27  LABEL_CMP_FILENAME:
28     cmp cx, 0
29     jz LABEL_FILENAME_FOUND ; 如果比较了 11 个字符都相等, 表示找到
30  dec cx
31     lodsb ; ds:si -> al
32     cmp al, byte [es:di]
33     jz LABEL_GO_ON
34     jmp LABEL_DIFFERENT ; 只要发现不一样的字符就表明本 DirectoryEntry 不是
35 ; 我们要找的 LOADER.BIN
36  LABEL_GO_ON:
37     inc di
38     jmp LABEL_CMP_FILENAME ; 继续循环
39
40  LABEL_DIFFERENT:
41     and di, 0FFE0h ; else di &= E0 为了让它指向本条目开头
42     add di, 20h ;
43     mov si, LoaderFileName ; di += 20h 下一个目录条目
44     jmp LABEL_SEARCH_FOR_LOADERBIN;
45
46  LABEL_GOTO_NEXT_SECTOR_IN_ROOT_DIR:
47     add word [wSectorNo], 1
48     jmp LABEL_SEARCH_IN_ROOT_DIR_BEGIN
49
50  LABEL_NO_LOADERBIN:

```

```

51     mov dh, 2          ; "No LOADER."
52     call DispStr       ; 显示字符串
53 %ifdef _BOOT_DEBUG_
54     mov ax, 4c00h      ;
55     int 21h           ; 没有找到 LOADER.BIN, 回到 DOS
56 %else
57     jmp $             ; 没有找到 LOADER.BIN, 死循环在这里
58 %endif

```

在根目录中找到 loader 之后，可以看到上面的代码中保存了簇号，然后调用了过程 GetFATEntry，其作用是查找当前簇号在 FAT 表中对应的值（来判断当前簇是不是最后一簇，如果不是的话可以得到下一个簇号），从而可以得到簇链：

```

1  ;-----
2  ; 函数名: GetFATEntry
3  ;-----
4  ; 作用:
5  ;   找到序号为 ax 的 Sector 在 FAT 中的条目, 结果放在 ax 中
6  ;   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
7  GetFATEntry:
8      push es
9      push bx
10     push ax
11     mov ax, BaseOfLoader;
12     sub ax, 0100h      ; | 在 BaseOfLoader 后面留出 4K 空间用于存放 FAT
13     mov es, ax        ; /
14     pop ax
15     mov byte [bOdd], 0
16     mov bx, 3
17     mul bx            ; dx:ax = ax * 3
18     mov bx, 2
19     div bx            ; dx:ax / 2 ==> ax <- 商, dx <- 余数
20     cmp dx, 0
21     jz LABEL_EVEN
22     mov byte [bOdd], 1
23 LABEL_EVEN:;偶数
24     ; 现在 ax 中是 FATEntry 在 FAT 中的偏移量,下面来
25     ; 计算 FATEntry 在哪个扇区中(FAT占用不止一个扇区)
26     xor dx, dx
27     mov bx, [BPB_BytsPerSec]
28     div bx ; dx:ax / BPB_BytsPerSec
29     ; ax <- 商 (FATEntry 所在的扇区相对于 FAT 的扇区号)
30     ; dx <- 余数 (FATEntry 在扇区内的偏移)。
31     push dx
32     mov bx, 0 ; bx <- 0 于是, es:bx = (BaseOfLoader - 100):00
33     add ax, SectorNoOfFAT1 ; 此句之后的 ax 就是 FATEntry 所在的扇区号
34     mov cl, 2
35     call ReadSector ; 读取 FATEntry 所在的扇区, 一次读两个, 避免在边界
36     ; 发生错误, 因为一个 FATEntry 可能跨越两个扇区
37     pop dx
38     add bx, dx
39     mov ax, [es:bx]
40     cmp byte [bOdd], 1
41     jnz LABEL_EVEN_2
42     shr ax, 4

```

```

43 LABEL_EVEN_2:
44     and ax, 0FFFh
45
46 LABEL_GET_FAT_ENRY_OK:
47
48     pop bx
49     pop es
50     ret
51 ;-----

```

可以看到在上面的函数中区分了扇区号的奇偶性，原因是 FAT12 是一个簇号占 12 位，两扇区才对齐一次。也就是说扇区号为奇数的时候，该扇区第一个字节为一个新的簇号的开始；而当扇区号为偶数的时候，该扇区的第一个字节和上一个字节的高四位共同组成一个簇号，第二字节才是新的簇号，所以要进行区分，这同时也是代码中一次读两个扇区的原因。

在实现在根目录中找到首簇、在 FAT 找到簇链的功能后就可以加载 loader 了：

```

1 LABEL_FILENAME_FOUND:           ; 找到 LOADER.BIN 后便来到这里继续
2     mov ax, RootDirSectors
3     and di, 0FFE0h              ; di -> 当前条目的开始
4     add di, 01Ah                ; di -> 首 Sector
5     mov cx, word [es:di]
6     push cx                    ; 保存此 Sector 在 FAT 中的序号
7     add cx, ax
8     add cx, DeltaSectorNo       ; cl <- LOADER.BIN 的起始扇区号 (0-based)
9     mov ax, BaseOfLoader
10    mov es, ax                  ; es <- BaseOfLoader
11    mov bx, OffsetOfLoader      ; bx <- OffsetOfLoader
12    mov ax, cx                  ; ax <- Sector 号
13
14 LABEL_GOON_LOADING_FILE:
15    push ax                     ;
16    push bx                     ; |
17    mov ah, 0Eh                 ; | 每读一个扇区就在 "Booting " 后面
18    mov al, '.'                 ; | 打一个点，形成这样的效果：
19    mov bl, 0Fh                 ; | Booting .....
20    int 10h                     ; |
21    pop bx                      ; |
22    pop ax                      ; /
23
24    mov cl, 1
25    call ReadSector
26    pop ax                      ; 取出此 Sector 在 FAT 中的序号
27    call GetFATEntry
28    cmp ax, 0FFFh
29    jz LABEL_FILE_LOADED
30    push ax                     ; 保存 Sector 在 FAT 中的序号
31    mov dx, RootDirSectors
32    add ax, dx
33    add ax, DeltaSectorNo
34    add bx, [BPB_BytsPerSec]
35    jmp LABEL_GOON_LOADING_FILE
36 LABEL_FILE_LOADED:
37    mov dh, 1                   ; "Ready."

```

```
38 | call DispStr ; 显示字符串
```

此时 loader 已经被加载到内存中，接下来就可以移交控制权（跳转到 loader 开始处）开始执行 loader 了。

```
1 | jmp BaseOfLoader:OffsetOfLoader
2 | ; 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。
```

3.2.2 调试运行

loader 的功能为打印字符'L'，运行结果如下：

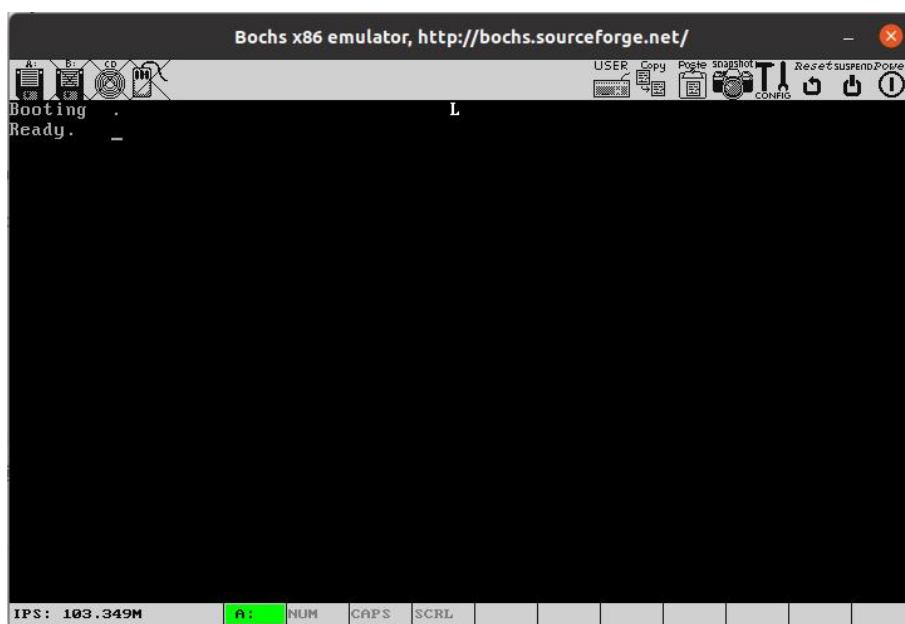


图 13: loader 执行结果

可以看到第一行的“Booting”后有一个点，loader 确实占用一个扇区，并且打印出了字符'L'，实验结果符合预期，loader 被正确加载并执行。

3.3 使用 xxd 读取二进制文件

xxd 的具体用法见4.2.2一节，这里尝试使用 xxd 工具，对3.1 和3.2 两节的二进制信息进行分析。

3.3.1 分析 FAT12 文件系统

使用 xxd -a 指令可以二进制文件的非 0 部分打印出来，对于本次实验中这种磁盘文件有大量空闲的情况有很大帮助，便于查找各项内容。

```

1 00000000: eb3c 906d 6b66 732e 6661 7400 0201 0100  .<.mkfs.fat.....
2 00000010: 02e0 0040 0bf0 0900 1200 0200 0000 0000  ...@.....
3 00000020: 0000 0000 0000 2925 85ac 2b4e 4f20 4e41  .....)%..+NO NA
4 00000030: 4d45 2020 2020 4641 5431 3220 2020 0e1f  ME    FAT12  ..
5 00000040: be5b 7cac 22c0 740b 56b4 0ebb 0700 cd10  .[|".t.V.....
6 00000050: 5eeb f032 e4cd 16cd 19eb fe54 6869 7320  ^..2.....This
7 00000060: 6973 206e 6f74 2061 2062 6f6f 7461 626c  is not a bootabl
8 00000070: 6520 6469 736b 2e20 2050 6c65 6173 6520  e disk.  Please
9 00000080: 696e 7365 7274 2061 2062 6f6f 7461 626c  insert a bootabl
10 00000090: 6520 666c 6f70 7079 2061 6e64 0d0a 7072  e floppy and..pr
11 000000a0: 6573 7320 616e 7920 6b65 7920 746f 2074  ess any key to t
12 000000b0: 7279 2061 6761 696e 202e 2e2e 200d 0a00  ry again ... ..
13 000000c0: 0000 0000 0000 0000 0000 0000 0000 0000  .....
14 *
15 000001f0: 0000 0000 0000 0000 0000 0000 0000 55aa  .....U.
16 00000200: f0ff ff00 0000 0000 0000 0000 0000 0000  .....
17 00000210: f0ff 0000 0000 0000 0000 0000 0000 0000  .....
18 00000220: 0000 0000 0000 0000 0000 0000 0000 0000  .....
19 *
20 00001400: f0ff ff00 0000 0000 0000 0000 0000 0000  .....
21 00001410: f0ff 0000 0000 0000 0000 0000 0000 0000  .....
22 00001420: 0000 0000 0000 0000 0000 0000 0000 0000  .....
23 *
24 00002600: 4168 0065 006c 006c 006f 000f 00f1 2e00  Ah.e.l.l.o.....
25 00002610: 7400 7800 7400 0000 ffff 0000 ffff ffff  t.x.t.....
26 00002620: 4845 4c4c 4f20 2020 5458 5420 000f e996  HELLO  TXT ....
27 00002630: 5055 5055 0000 e996 5055 0b00 0c00 0000  PUPU....PU.....
28 00002640: e570 0000 00ff ffff ffff ff0f 000e ffff  .p.....
29 00002650: ffff ffff ffff ffff ffff 0000 ffff ffff  .....
30 00002660: e52e 0068 0065 006c 006c 000f 000e 6f00  ...h.e.l.l....o.
31 00002670: 2e00 7400 7800 7400 2e00 0000 7300 7700  ..t.x.t.....s.w.
32 00002680: e545 4c4c 4f54 7e31 5357 5020 004c e396  .ELLOT~1SWP .L..
33 00002690: 5055 5055 0000 e396 5055 0000 0000 0000  PUPU....PU.....
34 000026a0: e568 0065 006c 006c 006f 000f 00ed 2e00  .h.e.l.l.o.....
35 000026b0: 7400 7800 7400 7e00 0000 0000 ffff ffff  t.x.t.~.....
36 000026c0: e545 4c4c 4f7e 3120 5458 5420 000f e996  .ELLO~1 TXT ....
37 000026d0: 5055 5055 0000 e996 5055 0000 0000 0000  PUPU....PU.....
38 000026e0: e545 4c4c 4f54 7e31 5357 5820 004c e396  .ELLOT~1SWX .L..
39 000026f0: 5055 5055 0000 e396 5055 0000 0000 0000  PUPU....PU.....
40 00002700: 0000 0000 0000 0000 0000 0000 0000 0000  .....
41 *
42 00005400: 4865 6c6c 6f20 576f 726c 640a 0000 0000  Hello World.....
43 00005410: 0000 0000 0000 0000 0000 0000 0000 0000  .....
44 *
45 00167ff0: 0000 0000 0000 0000 0000 0000 0000 0000  .....

```

如上所示，二进制文件被清楚地显示成了 DBR，FAT，根目录、数据区这些不同的区域。通过 xxd 输出的内容，可以快速从 FAT12 磁盘文件中手工定位文件。

3.3.2 加载 loader 入内存

我们通过 bochs 的 writemem 指令，获取了虚拟机物理内存的二进制文件。使用 xxd -a 来查看这个内存文件。可以看到，内存 0x8f000 的位置 (BaseOfLoader:OffsetOfLoader

- 0x1000) 的位置存放了 FAT 表，这是程序在执行过程中用到的。而在 0x90100 位置 (BaseOfLoader:OffsetOfLoader)，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，实现了实验目的。

```
1 *
2 0008f000: 0000 0000 f0ff 0000 0000 0000 0000 0000 .....
3 0008f010: 0000 0000 0000 0000 0000 0000 0000 0000 .....
4 *
5 00090100: b800 b88e e8b4 0fb0 4c65 a34e 00eb fe00 .....Le.N....
6 00090110: 0000 0000 0000 0000 0000 0000 0000 0000 .....
```

3.3.3 使用 xxd 的一些小技巧

1. xxd binfile | vim -: 使用 vim 查看 xxd 的输出
2. 在 vim 中编辑二进制文件: 使用 vim 打开二进制文件，然后输入:%!xxd，即可将文件转换为可编辑的文本形式。在编辑结束后输入:%!xxd -r 可以将编辑后的文本重新保存为二进制形式。

4 实验结果总结

4.1 FAT12 结构

FAT 文件系统包含四个部分: 引导扇区, FAT 表, 根目录区和数据区。其结构如下图所示:



图 14: FAT12 结构

下面分别介绍各个部分的结构。

4.1.1 引导扇区

引导扇区占据一个扇区，它不仅包含有引导程序，还有 FAT12 文件系统的整个组成结构信息。引导扇区的结构如表1。

表 1: 引导扇区结构

名称	偏移	长度	内容
BS_jumpBoot	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

在计算数据区扇区位置时, 主要用到的是 BPB_RootEntCnt 和 BPB_BytesPreSec。

4.1.2 FAT 表

FAT 文件系统以簇来为单位来分配数据区的存储空间 (扇区), 每个簇的长度为: $\text{BPB_BytesPreSec} \times \text{BPB_SecPreClus}$ 字节, 数据区的簇号与 FAT 表的表项是一一对应关系。因此, 文件在 FAT 类文件系统的存储单位是簇, 而非字节或扇区。这种设计方式可以将磁盘存储空间按固定储片 (页) 有效管理起来, 进而按照文件偏移, 分片段访问文件内的数据, 就不必一次将文件里的数据全部读取出来。

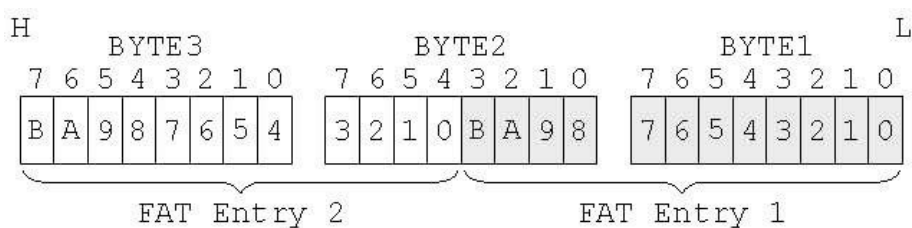


图 15: FAT12 的 FAT 项

在 FAT12 中，每个 FAT 项占 12bit，第一项是 0 号（但是注意第 0 和第 1 个 FAT 项始终不使用，从 2 号开始表示数据区的簇）。多数情况下，FAT 项的值表示代表文件的下一簇的簇号。具体的含义分为以下几类：

- 0x000：可用簇
- 0x002 ~ 0xFEFF：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

当文件的体积增大时，其所需的磁盘存储空间也会增加，随时间的推移，文件系统将无法保证文件中的数据存储在连续的磁盘扇区内，文件往往被分成若干个片段。借助 FAT 表项，可将这些不连续的文件片段按簇号链接起来，这个链接原理与链表数据结构相似。

根据图14，我们发现 FAT12 系统中有 FAT1 和 FAT2，它们分别占 9 个扇区，是相互备份的关系。

4.1.3 根目录区

根目录位于 FAT2 表之后，从第 19 个扇区开始。它由若干个目录条目组成。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、文件内容起始地址以及其他一些元数据。根目录中每一个条目占用 32 个字节。具体结构如下表所示。

表 2: 根目录区的条目格式

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

4.1.4 数据区

数据区的某一个扇区如果是一个目录，那么这个目录被称为子目录。子目录分为一级、二级……子目录，他们的组织方法和根目录基本一样。如果某个扇区储存的是文件普通文件，那么就没有什么特定的规则，根据每个文件的不同会有不同的数据组织结构。

4.2 读取软盘信息

4.2.1 汇编语言读取

13H 号系统中断能够对磁盘进行操作。其中功能号 2 为读取扇区信息。参数如下表所示：

参数	内容
AH	0x02
AL	读取的扇区数
CH	磁道号
CL	扇区号
DH	磁头号
DL	驱动器号
ES:BX	接收从扇区读入数据的内存区的指针

扇区内容最终将被写道 ES:BX 指向的内存区中。通过该方法可以得到软盘信息。

4.2.2 终端命令查看

xxd 命令用于用二进制或十六进制显示文件的内容，如果没有指定 outfile 参数，则把结果显示在屏幕上，如果指定了 outfile 则把结果输出到 outfile 中；如果 infile 参数为“-”或者没有指定 infile 参数，则默认从标准输入读入。指令的形式如下：

· xxd [options] [infile [outfile]]

其中 options 有以下一些参数：

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. 加号表示从地址处开始的内容，减号表示距末尾 address 开始的内容

4.2.3 查看器查看

PEView、010Editor 等工具都能够方便地查看文件的十六进制信息。文中用到了 010Editor，并且它提供模板，可以帮助分析数据信息。

4.3 定位指定文件

定位根目录下的文件步骤如图16所示。

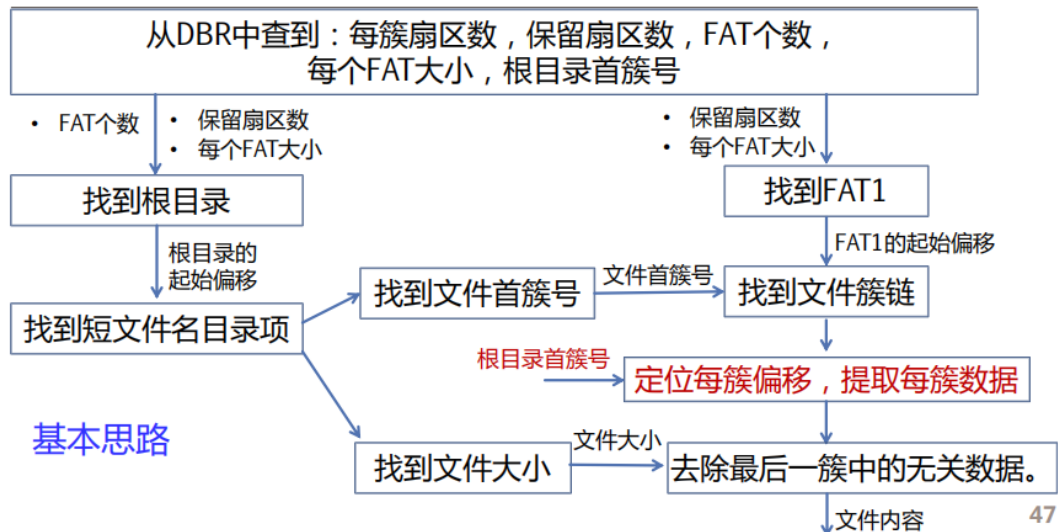


图 16: 定位根目录文件步骤

这里补充一下如何从簇号计算出对应的扇区号。根据引导扇区信息，可以直接得到 FAT 占用的扇区和根目录表的条目数量。根据下面的这个公式可以得到根目录表占用的扇区数。

$$RootDirSectors = \frac{BPB_RootEntCnt \times 32 + (BPB_BytePerSec - 1)}{BPB_BytePerSec} \quad (1)$$

上式中 RootDirSectors 为根目录区占用扇区数，BPB_RootEntCnt 为根目录最大条目数，BPB_BytePerSec 为每个扇区的字节数，除法是整数除法。

引导扇区占一个扇区，再加上 FAT 表占用扇区和根目录区占用扇区，则可以推算出数据区的初始扇区了。这里尤其注意，数据区的初始扇区对应的是 2 号簇。

另外，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，整体的步骤与上图是十分类似的，这里不再详述。

4.4 在系统引导过程中，读取加载可执行文件到内存并移交控制权的流程

本问题在3.2节中已经有所涉及，简单的讲就是读 BPB 找到根目录和 FAT 表的位置，然后读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指定地址，再带入 FAT 表得到簇链的下一项，然后重复读入内存、带入 FAT 表，直至簇链结束，就可以将文件加载进入内存（同时也是定位文件的过程，可以回答上一个问题），然后直接跳转到加载的位置就可以移交控制权。

4.5 loader 程序不包含 dos 系统调用的原因

引导扇区中代码的作用就是加载 loader，而 loader 用于加载操作系统内核，然后移交控制权给操作系统，这说明引导扇区代码在执行时，操作系统一定还没有被加载，那么操作系统的调用自然也不可用。dos 也是一个操作系统，如果我们在 loader 中使用 dos 系统调用，则需要先把 dos 加载到内存中。

4.6 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？

4.6.1 硬盘启动

在较为早期的硬盘上，主要采用 MBR 的方式实现 bootstrap，如图 17 所示。其中，Bootstrap 代码部分负责将启动分区的 DBR 加载到内存中，并跳转到该 DBR。

Structure of a classical generic MBR			
Address	Description		Size (bytes)
0x0000 (0)	Bootstrap code area		446
0x01BE (446)	Partition entry №1	Partition table (for primary partitions)	16
0x01CE (462)	Partition entry №2		16
0x01DE (478)	Partition entry №3		16
0x01EE (494)	Partition entry №4		16
0x01FE (510)	0x55	Boot signature ^[a]	2
0x01FF (511)	0xAA		
Total size: 446 + 4×16 + 2			512

图 17: MBR 结构

4.6.2 FAT32 系统

FAT32 和 FAT12/16 相比，对于 bootloader 来说，最大的差别在于 FAT32 中，根目录簇可能从任意簇号开始，即根目录不一定紧接在 FAT 区域后面。所以在 FAT32 中，根目录与其它文件、目录并没有明显区别。在 BPB 中定义了 BPB_RootClus，指示根目录的簇号。因此程序需要像访问文件一样顺着簇链循环获取各个簇，并在簇中根目录的各个目录项。

4.6.3 NTFS 系统

NTFS 分区的首扇区也存储了 DBR 信息，0x0 处是一个跳转指令，跳转到 BPB 结束的位置，这个位置存有加载 Loader 的程序。对于 NTFS 文件系统而言，这个位置被称为 \$Boot 文件，这个文件占用了 16 个扇区。所以 NTFS 文件系统中启动代码可用的空间是要大于 FAT 的。

4.6.4 其它方法

书上提到，我们之所以使用 FAT12 是为了兼容 DOS，便于调试。实际上我们也可以不把内核放到文件系统中，而是直接把 loader 或者内核放到一个特定的位置，在 Bootstrap 代码中直接加载该位置的数据。由于 MBR 中指定了各个分区的起始地址和结束地址，则完全可以在将起始地址整体后移，空出一部分区域专门用于存放 loader。这样加载 loader 就不需要解析文件系统了。

5 实验问题汇总

1. 在查看 010Editor 的辅助信息时，发现书中对根目录项的结构描述与模板分析内容不一致。进一步搜索资料，在官方文档中找到更为精确的结构表述。

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)

图 18: 根目录区条目格式修正

2. 如果文件名是小写，根目录表中会存放什么内容？

在 Linux 下，我们新建了一个软盘，并使用 Free dos 的 format 命令对其进行了格式化。被 Free dos 格式化的软盘，会被自动设置为 FAT12 格式。接下来，我们通过 Linux 的 loop 设备，将软盘挂载到 Linux 上，并写入了一些以小写字母命名的文件。其根目录区的内容如下图所示。

2600h:	41 66 00 6C	00 6F 00 77	00 65 00 0F	00 57 72 00	Af.l.o.w.e...Wr.
2610h:	2E 00 74 00	78 00 74 00	00 00 00 00	FF FF FF FF	..t.x.t....ÿÿÿ
2620h:	46 4C 4F 57	45 52 20 20	54 58 54 20	00 64 E5 1D	FLOWER TXT .dä.
2630h:	4A 55 4A 55	00 00 E5 1D	4A 55 03 00	08 07 00 00	JUJU..ä.JU.....
2640h:	41 74 00 72	00 65 00 65	00 2E 00 0F	00 28 74 00	At.r.e.e....(t.
2650h:	78 00 74 00	00 00 FF FF	FF FF 00 00	FF FF FF FF	x.t...ÿÿÿ..ÿÿÿ
2660h:	54 52 45 45	20 20 20 20	54 58 54 20	00 00 0A 1E	TREE TXT
2670h:	4A 55 4A 55	00 00 0A 1E	4A 55 07 00	0D 00 00 00	JUJU....JU.....
2680h:	41 72 00 69	00 76 00 65	00 72 00 0F	00 7D 2E 00	Ar.i.v.e.r...}..
2690h:	74 00 78 00	74 00 00 00	FF FF 00 00	FF FF FF FF	t.x.t...ÿÿ..ÿÿÿ
26A0h:	52 49 56 45	52 20 20 20	54 58 54 20	00 64 0C 1E	RIVER TXT .d..
26B0h:	4A 55 4A 55	00 00 0C 1E	4A 55 08 00	11 00 00 00	JUJU....JU.....
26C0h:	48 4F 55 53	45 20 20 20	20 20 20 10	00 00 3B 1E	HOUSE;
26D0h:	4A 55 4A 55	00 00 3B 1E	4A 55 09 00	00 00 00 00	JUJU.;.JU.....
26E0h:	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00

图 19: 小写字母文件名的根目录区存放形式

可以看到 flower.txt 等文件依然会转换成对应的大写来生成对应的目录项。不过在 这些目录项前面会有 20 个字节，用于描述文件的真实名称。这 20 个字节遵循长文件名条目的结构。

这与我们预料的、软件安全中学到的一致。但是，Free dos 真的支持长文件名吗？于是，我们又进行了下面的实验：

首先，为了保证功能的完整，我们取了 Free dos 映像压缩包中的 c.img 作为启动盘。它拥有 FreeDOS kernel version 1.1.17 的完整功能。并给 bochs 虚拟机挂上了空白的软盘。利用 Free dos 格式化后，尝试创建长的、小写字母的文件名。

```
C:\>echo "test for lfn" > abcdefghijklmn.opqrst
C:\>echo "test for me" > b.txt
```

图 20: Free dos 下创建长文件名

然而，它并没有产生长目录项，只是简单的截断了它。将文件名转换成了小写，并把文件名截剩 8 个字符，扩展名截剩 3 个字符。

000050e0	41 42 43 44 45 46 47 48	4f 50 51 20 00 00 00 00	ABCDEFHOPQ
000050f0	00 00 00 00 00 00 ac 40	50 55 8b 02 10 00 00 00@PU.....
00005100	42 20 20 20 20 20 20 20	54 58 54 20 00 00 00 00	B TXT
00005110	00 00 00 00 00 00 b0 40	50 55 8c 02 0f 00 00 00@PU.....

图 21: 长文件名被截断

然后，我们又进行了第二个实验。这次，我们还是在 Linux 下创建了软盘，并利用 Free dos 格式化。只是将写入文件的步骤移动到了 Linux 上。我们发现，Linux 如我们之前做的，为文件名分配了长目录项，这个长文件名在 Linux 下能够被正常识别。

```
→ MyCodingDir sudo mount -o loop a.img /mnt/floppy
→ MyCodingDir cd /mnt/floppy && sudo touch abcdefghijklmn.opqrst
→ floppy ls
abcdefghijklmn.opqrst
```

图 22: Linux 下创建长文件名

但是，当在 Free Dos 下时，长目录项并不会得到处理，而会被简单的忽略。

```
B:\>dir

Volume in drive B has no label
Directory of B:\*.*

ABCDEF~1 OPQ                0 10-16-122   3:17p
      1 file                  0 bytes
      0 dirs                 1,458,176 bytes free

B:\>del abcdefghijklmn.opqrst
File not found.

B:\>del ABCDEF~1.OPQ

B:\>
```

图 23: Free dos 无法识别 LFN

据我们所知，长目录项其实是 FAT 的扩展功能，FreeDos 本身并不支持。

表 3: 长文件名条目结构

名称	偏移	长度	内容
Sequence	0	1	序
First 5 Characters	1	10	文件名的前 5 个字符
Attributes	11	1	属性 (值为 0F)
Reserved	12	1	保留位
Checksum	13	1	校验码
Next 6 characters	14	12	接下来的 6 个字符
Reserved	26	2	保留位
Last 2 characters	28	4	文件名最后两个字符

参考资料汇总

1. 配套教材《Orange's 一个操作系统的实现》。
2. 操作系统学习笔记——FAT12 文件系统与 Loader 的加载
3. 【实现操作系统 02】FAT12 文件系统（摆脱术语用实际例子介绍）
4. fat12_description.pdf

6 各人实验贡献与体会

在手工定位文件中，我们对 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，我们了解了 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。通过本次实验，我们熟悉了各种二进制编辑工具的使用，尤其是在 linux 下使用 xxd 进行二进制读取与编辑的方法。

本次实验分工大致如下：李心杨负责调研硬盘及其它文件结构下的 loader 加载；王宇骥负责梳理 FAT12 的各个部分结构以及整理手工定位文件的步骤；林锟扬负责了资料的收集，补充了一些实验；郑炳捷负责调试 boot.asm，对加载可执行程序入内存的原理进行解释。

教师评语		
姓名	学号	分数
李心杨	2020302181022	
王宇骥	2020302181008	
林锟扬	2020302181032	
郑炳捷	2020302181024	
教师签名：		
年 月 日		