实验九：FAT12文件系统引导和加载

1 实验题目

初步实现软盘的 FAT12 文件系统，使存储规范，mbr 和内核能够通过文件系统来加载程序。
具体来说，共有五项：

1. 利用十六进制查看器分析一个 FAT12 文件系统映像文件，验证结构各部分的位置、大小和标志；
2. 生成为一个带个人标识的 1.44MB 完整 FAT12 文件系统盘的映像文件，并用 WinImage 或 FAT12 仿真操作程序或其他同类工具，在其中加进 3~5 个文件；
3. 修改 mbr，实现从 FAT12 文件系统盘中加载内核执行文件；
4. 修改内核代码，实现从 FAT12 文件系统加载特定名字的用户程序文件，并执行用户程序；
5. 编写一个程序，用于列出一个 FAT12 的根目录中一个文件分配到的所有簇号和对应的逻辑扇区号等等，如果文件太大，只要列出前 5 个簇即可。

2 实验目的

1. 掌握 FAT12 文件系统结构和组织文件方法；
2. 掌握 FAT12 文件系统的引导扇区程序设计方法；
3. 掌握 FAT12 文件系统中加载内核执行文件的引导方法；
4. 掌握 FAT12 文件系统中加载用户程序执行文件的方法。

3 实验要求

1. 确保多进程等模型还能正常工作。

4 实验方案

工具与环境：Windows（x86_64），gcc，g++，NASM，ld 2.25.1，bochs 虚拟机（1MB 内存、1.44MB 软驱），Hex Editor

FAT12 文件系统是一种存储格式，实现之后能够打破以前的无格式存储，更规范、高效地实现存储设备的文件管理。实验首先需要掌握 FAT12 文件系统的规范，然后使用映像管理工具改造原来的软盘，最后依次修改 mbr、内核中的“加载执行用户程序”。

对于映像管理工具，由于 WinImage 等软件没有免费正版，我将自行用 c++ 写一个简易的管理工具。

对于 FAT12 中的 FAT 表、根目录项，这次实验是把它们视为只读的，因此没有写磁盘操作，相对简单些。

扇区号、簇号统一使用逻辑编号（从 0 开始），仅在 `int 13h` 读磁盘时使用物理扇区号。

本次实验重点为掌握 FAT12 文件系统的规范，难点为根据 FAT 表、根目录项寻找并加载文件。

实验过程部分将以流畅的方式叙述实验细节，更多试错过程写在实验总结部分。

5 实验过程

5.1 分析一个FAT12文件系统映像文件

在 1.44MB 软盘中，FAT12 将软盘分为 1+9+9+14+2847=2880 个扇区（逻辑编号 0 ～ 2879），每个扇区 512 字节。文件储存以“簇”为单位，一簇大小为一个扇区，一个文件的所有簇用链表来串起来。

首扇区为软盘信息及引导程序；接下来 9 个扇区为 FAT 表，除了头两个元素以外每个元素对应一个簇，该元素的值为“下一簇是谁”；接下来 9 个扇区为 FAT 表的备份，即这两个 FAT 表是一模一样的；然后 14 个扇区为根目录项，即根目录下的文件信息；最后 2847 个扇区为数据区。

实验一已通过 VMware 软件获得一张格式化的 FAT12 空盘，现对这张空盘进行分析验证。

先看首扇区。网课课件给出了 FAT12 首扇区的格式[1]，如左图；空盘的首扇区如右图：

名称	偏移	长度	内容	软盘参考值
BS_jmpBoot	0	3		转移指令
BS_OEMName	3	8	厂商名	'ForrestY'
BPB_BytsPerSec	11	2	每扇区字节数	0x200（即十进制512）
BPB_SecPerClus	13	1	每簇扇区数	0x01
BPB_RsvdSecCnt	14	2	Boot记录占用多少扇区	0x01
BPB_NumFATs	16	1	共有多少FAT表	0x02
BPB_RootEntCnt	17	2	根目录文件数最大值	0xE0（224）
BPB_TotSec16	19	2	扇区总数	0xB40（2880）
BPB_Media	21	1	介质描述符	0xF0
BPB_FATSz16	22	2	每FAT扇区数	0x09
BPB_SecPerTrk	24	2	每磁道扇区数	0x12
BPB_NumHeads	26	2	磁头数	0x02
BPB_HiddSec	28	4	隐藏扇区数	0
BPB_TotSec32	32	4	值记录扇区数	0xB40（2880）
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	0x29
BS_VolD	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	62	448	引导代码、数据及其他填充字符等	
结束标志	510	2		0xAA55

图 1: 首扇区格式

```
00000000  00 01 02 03 04 05 06 07  08 09 0a 0b 0c 0d 0e 0f
00000000  eb 3c 90 49 50 52 54 20  36 2e 32 00 02 01 01 00   ë < I P R T   6 . 2 . . . . .
00000010  02 e0 00 40 0b f0 09 00  12 00 02 00 00 00 00 00   . à . @ . ð . . . . . . . . .
00000020  00 00 00 00 00 00 29 7a  36 e3 f3 20 20 20 20 20   . . . . . . ) z 6 ã ó
00000030  20 20 20 20 20 20 46 41  54 31 32 20 20 20 cd 19           F A T 1 2       Í .
00000040  cc cc 00 00 00 00 00 00  00 00 00 00 00 00 00 00   Ì Ì . . . . . . . . . . . . . . . .
00000050  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
00000060  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
00000070  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
00000080  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
00000090  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
000000a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
000000b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
000000c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
000000d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
000000e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
000000f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
00000100  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
00000110  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
00000120  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
00000130  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
00000140  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
00000150  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
00000160  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
00000170  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
00000180  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
00000190  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
000001a0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
000001b0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
000001c0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
000001d0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
000001e0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00   . . . . . . . . . . . . . . . .
000001f0  00 00 00 00 00 00 00 00  00 00 00 00 00 00 55 aa   . . . . . . . . . . . . . . U ª
```

图 2: 空盘首扇区

偏移 3 ～ 64 是软盘基本信息，规定了软盘的存储格式。开头 3 个字节为指令，分别是 `jmp +60(0x7c3e)`（代码为 `eb3c`）和 `nop`（代码为 `90`），这样 BIOS 来到 `0x7c00` 后，就能跳过软盘信息，直接到达引导程序的位置 `0x7c3e`。这张空盘的引导程序就只有一句 `int 0x19`（代码为 `cd19`），功能为重启，以致死循环。首扇区末尾为合法首扇区标志 `0xaa55`。

接下来是 FAT 表。这里每一个表项占 1.5 字节（12 位）。头两个表项分别代表介质描述符、结束簇标记。其他表项分别代表数据区中的簇（这里有用的簇号从 2 开始，而数据区逻辑扇区号从 33 开始，因此簇号=逻辑扇区号+31），表项的值表示“下一簇是谁”，这样就形成了链表。

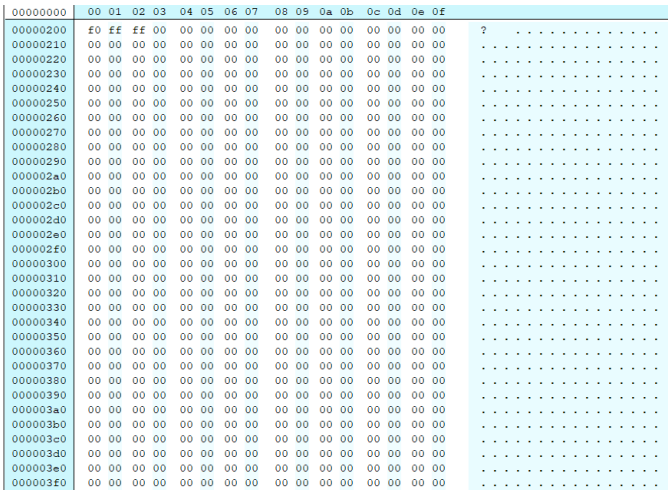

图 3: FAT表

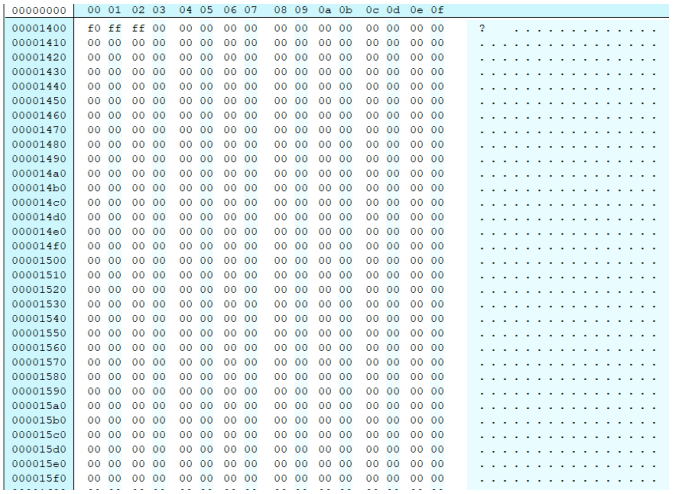

图 4: 备份FAT表

可以看到，空盘没有数据，因此除了头两个元素以外都是 0。两份 FAT 表是一样的。（第二份的起始地址为 (1+9)×0x200=0x1400）

然后是根目录项，起始地址 (1+9+9)×0x200=0x2600。每项 32 字节，表示根目录下的一个文件的信息，格式如左图：

图 5: 根目录项格式

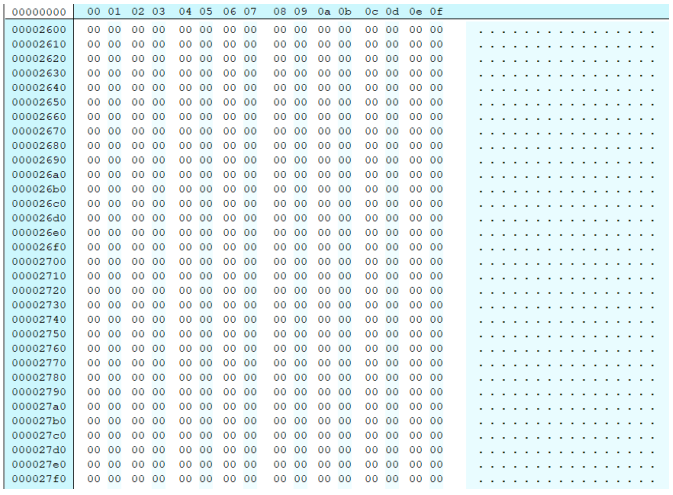

图 6: 空盘根目录项

空盘没有文件，因此根目录项为空。

最后是数据区，起始地址 (1+9+9+14)×0x200=0x4200。这里用了 0xf6 作为空数据填充：

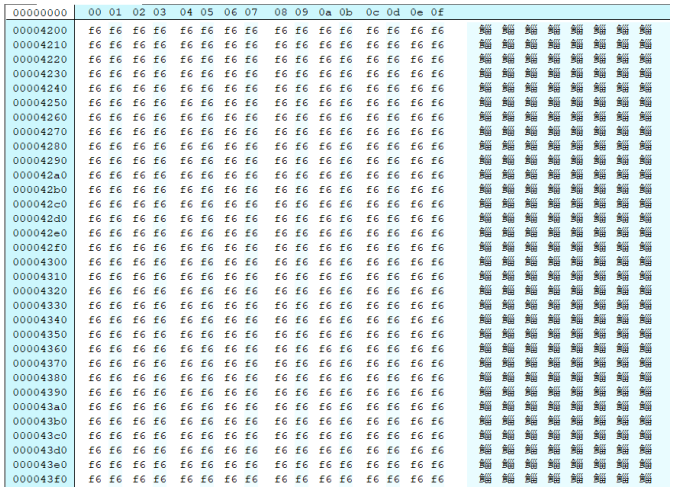

图 7: 数据区

5.2 制作映像管理工具及格式化软盘

现要将以前实验的软盘改造成 FAT12。此处用 c++ 写一个程序，功能为将一个文件按 FAT12 的格式写入软盘。

参考网课时提供的 FAT12 仿真程序的设计思路[2]：读软盘的 FAT 表、根目录项，分别用一个大数组存起来。以参数的形式传入要写入的文件名，先在根目录表里找到一个空项，建立该文件的信息，然后从 FAT 表中找到足够多的空项，依次链起来，并在对应的数据区簇中写入该文件。

伪代码如下：（完整代码为 `\src\tools\imgOP.cpp`）

Algorithm 1 FAT12文件写入软盘

Require: 写入的文件名 `fileName`、软盘名 `flpName.img`

Ensure: 写入了该文件的软盘

$SECSIZE \leftarrow 512$

读取 `flpName.img` 第 $1 \sim 9$ 逻辑扇区至数组 fat

读取 `flpName.img` 第 $19 \sim 32$ 逻辑扇区至数组 $root$

读取文件 `fileName`，获得其大小 len

$cluNum \leftarrow \lceil \frac{len}{SECSIZE} \rceil$

在 fat 中寻找 $cluNum$ 个空项，记入临时数组 $d_1, \cdots, d_{cluNum}$

在 $root$ 中寻找 1 个空项，记录偏移地址为 $rtPos$

更新 $root[rtPos], \cdots, root[rtPos+31]$ 为文件信息

for $i \leftarrow 1$ to $cluNum$ **do**

    将 `fileName` 第 d_i 簇写入 `flpName.img` 第 $d_i + 31$ 逻辑扇区

    if $i < cluNum$ **then** MODIFYFATENTRY(fat,d_i,d_{i+1}) ▷ 非最后簇，修改 fat 表第 d_i 项的值为 d_{i+1}

    else MODIFYFATENTRY(fat,d_i,0xFFF) ▷ 最后簇，修改 fat 表第 d_i 项的值为 0xFFF

    end if

end for

fat 数组写入 `flpName.img` 第 $1 \sim 9$ 逻辑扇区、第 $10 \sim 18$ 逻辑扇区

$root$ 数组写入 `flpName.img` 第 $19 \sim 32$ 逻辑扇区

这里 *root* 和 *fat* 都是以字节为类型的数组（方便读入和存储，避免 `struct` 内存对齐），而其表项大小分别为 32B、1.5B，因此查找某一下标的表项有些麻烦。其中 *root* 数组查找较为简单，偏移指针每次往后移 32B 即可；而 *fat* 数组的查找较为复杂，因为涉及到了跨字节的操作。

对于 *fat* 数组，每 3 个字节总是由一个偶数下标项和一个奇数下标项构成，如原理图，偶数下标项从高位到低位依次为：中间字节的低 4 位、左边字节；奇数下标项从高位到低位依次为：右边字节、中间字节高 4 位。

因此，查找或修改的时候，针对下标的奇偶性进行分类讨论：设查找第 j 项，则涉及到的两个字节分别为 $i=\lfloor\frac{3j}{2}\rfloor$ 和 $i+1$，若 j 为偶数，则取i 字节作为 0~7 位、$i+1$ 字节低 4 位作为 $8\sim11$ 位；若 j 为奇数，则取 i 字节高 4 位作为 $0\sim3$ 位、$i+1$ 字节作为 $4\sim11$ 位。

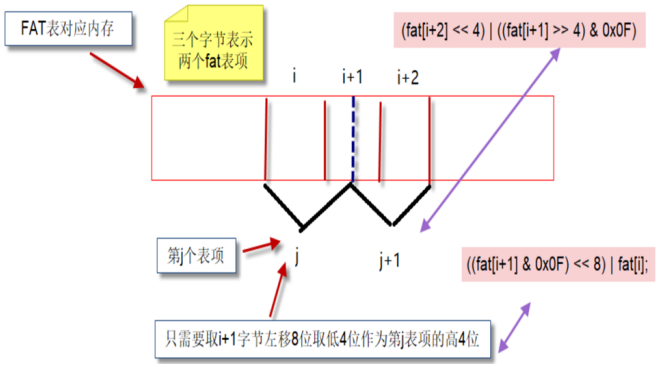

图 8: FAT表查找原理

```
int getFatEntry(int n)  // 查找FAT表第n项
{
    int pos=(n*3)>>1;
    return (n&1) ?((int)fat[pos+1]<<4)|(fat[pos]>>4) :((int)(fat[pos+1]&15)<<8)|fat[pos] ;
}

void modifyFatEntry(int n,int val)  // 修改FAT表第n项为val
{
    int pos=(n*3)>>1;
    if (n&1)
    {
        fat[pos+1]=val>>4;
        fat[pos]=(fat[pos]&(0x0F)) | ((val&15)<<4);
    } else
    {
        fat[pos+1]=(fat[pos+1]&(0xF0)) | (val>>8) ;
        fat[pos]=val&255;
    }
}
```

现来测试该工具。将之前的 5 个用户程序，连同内核 `myOS.com`，都用该工具写入 FAT12 空盘，结果如下：

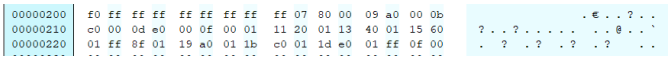

图 9: FAT表1

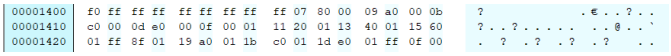

图 10: FAT表2

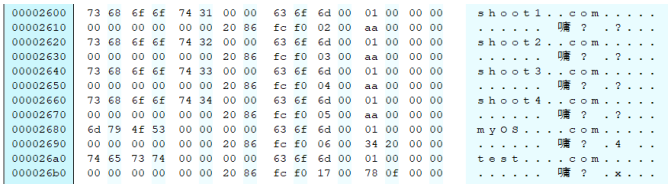

图 11: 根目录项

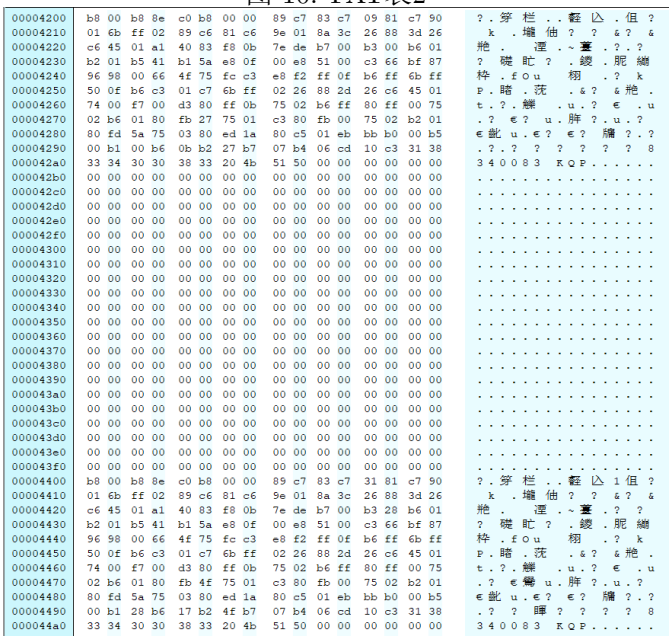

图 12: 数据区

可以看到：

1. FAT 表上，从表项 2 开始，连续 4 个 0xFFF，这是四个短的设字母程序；接下来一串 17 个表项，是内核；接下来一串 8 个表项，是系统调用测试程序；
2. 根目录表上，已经把 6 个程序按格式记录下来了；
3. 数据区，程序正确写入。

5.3 修改mbr

现修改 mbr.asm，使其能生成整个首扇区（包括软盘信息），并且引导程序能从 FAT12 文件系统里找到内核并加载。

首先是增加引导程序前的部分，开头两条指令 jmp 和 nop，接下来一大串软盘信息，这在汇编里很好实现，对着图 1 定义数据即可：

```
1  jmp boot_loader
2  nop
3  BS_OEMName       DB 'MyOS    '
4  BPB_BytsPerSec   DW 512
5  BPB_SecPerClus   DB 1
6  BPB_RsvdSecCnt   DW 1
7  BPB_NumFATs      DB 2
8  ...
9
10 boot_loader:
11 ...
```

接下来改造引导程序。引导程序的逻辑为：屏幕初始化，加载 FAT 表、根目录项到内存，从根目录项中找到 myOS.com，根据 FAT 表把内核加载到内存，跳转。

将 FAT 表加载到 0:0x8000 的位置，将根目录项加载到 0:0xA000 的位置，这只需要写一个通用的加载扇区函数（load，传入参数为 cx 逻辑扇区号、es:bx 目标内存地址），用一个循环来不断调用就可以了。例如加载 FAT 表：

```
fat_offset_addr equ 0x8000

; load FAT to 0x8000
mov word [next_offset], fat_offset_addr
mov byte [i], 1
mbr_load_FAT:
    mov ax, 0
    mov es, ax
    mov bx, [next_offset]
    add word [next_offset], 512
    movzx cx, byte [i]
    call load
    add byte [i], 1
    cmp byte [i], 10
    jnz mbr_load_FAT
```

从根目录中找内核，只需遍历所有的根目录表项，逐一比较其名字是否为 myOS.com 即可。这里是简单的遍历+字符串比较，不赘述代码。

假设找到内核文件在根目录项中的偏移地址为 bx，那么从 root[bx+26] 中获得内核的首簇号，接下来遍历这个链表，把内核加载到内存：

```
load_os:
    ; here bx is the offset of myOS.com in root directory
    mov cx, [bx+26]                 ; first cluster offset
    mov [curClu], cx
    mov word [next_offset], 100h
    load_os_while:
        mov ax, os_seg_addr         ; load os
        mov es, ax
        mov bx, [next_offset]
        add word [next_offset], 512
        mov cx, [curClu]
        add cx, 31
        call load

        mov cx, [curClu]            ; find next cluster
        ; here calculate dx = fat[cx]
        mov [curClu], dx
        cmp dx, 0xFFF
        jnz load_os_while
```

此处需要计算 dx 为下一簇的簇号（即 FAT 表第 curClu 项），这里相当于 5.2 节的 getFatEntry 函数翻译成汇编。

执行完 load_os 以后就是跳转过去，沿用旧的跳转方式就可以了。这样就改造完成了。

编译后复制粘贴到 5.2 节制作的软盘首扇区，运行虚拟机如下：

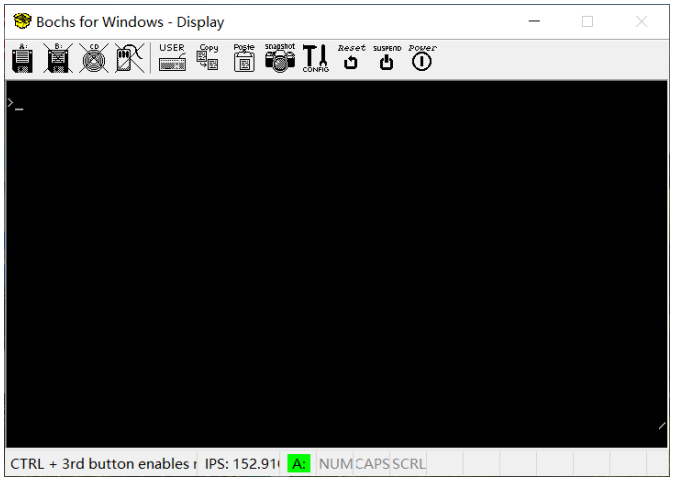

图 13: mbr改造

可以看到，屏幕出现 > 字符，这是内核的 shell 界面，说明内核被成功加载、跳转、执行。

5.4 修改内核

接下来修改内核，使其能根据输入的程序名来查找、加载、执行程序。这里有两个地方要改，一是命令解释器，二是加载方式。

对于命令解释器，输入的命令字符串以“|”为分界，每部分是一个单独的命令，可以是程序名，或其他命令。每得到一个命令，先从根目录项中匹配它，若匹配成功，说明是程序名，则进行正常的加载过程；若无法匹配，则再判断是否为 shell 命令。

```
// input command string buf[0..buflen-1]
for(int l=0, r=0; l<buflen; l=++r)
{
    while (r<buflen && buf[r]!='|' && buf[r]!='\r') r++;    // current command is buf[l..r-1]
    if (l==r) continue;
    if (find(l,r))      // match file name in root directory
    {
        // here load program
    } else if (bufcmp(l,r,"dir",3))     // match shell command name
    {
        // here dir
    } else
    {
        myprintf("undefined command!\r\n");
    }
}
```

FAT 表和根目录项已经被引导程序加载到内存第一个 64KB 段了，这里方便起见，直接拿来用，不再重新加载。当 C 代码无法直接调用这段内存时，用汇编写一个函数，取出一个指定表项。例如取出根目录项的指定表项（参数在栈里）存于 dirEnt 中：

```
global _read_dir                ; C statement: read_dir(i);
extern _dirEnt
```

```
_read_dir:
    mov ax, 0
    mov es, ax
    mov di, [esp+4]              ; di = &rootDir[i]
    shl di, 5
    add di, dir_offset_addr
    mov bx, 0                    ; loop var
    _read_dir_for:
        mov ah, [es:di+bx]
        mov [_dirEnt+bx], ah
        add bx, 1
        cmp bx, 32
        jnz _read_dir_for
    o32 ret
```

对于根目录项文件名匹配，这也是简单的遍历+字符串比较，不赘述代码。

匹配成功后，从根目录项信息得到其首簇号，通过 FAT 表得到其所有簇。这里跟 5.2 节的“通过 FAT 表项将文件写入相应的数据区”是一样的，只不过把“文件写入数据区”换成“数据区加载到内存”：

```
for(short st=(dirEnt[27]<<8)+dirEnt[26], bx=0x100;  // st: cluster id, bx: offset address
    st!=0xFFF;
    st=next_cluster(st), bx+=0x200)
        load_client(seg_addr,bx,st);
```

这里同样用汇编实现“查找下一簇”的过程，因为 C 代码无法直接使用 FAT 表所在的内存。这个过程也是相当于 5.2 节的 `getFatEntry` 翻译成汇编。

“用户程序加载执行”的其他部分，就跟以往一样了。

现在可以正规地实现 `dir` 命令了。遍历一遍根目录项，把非空项的相关信息输出即可，这里只输出名字和大小：

```
const int ROOTNUM=224;

for(int i=0; i<ROOTNUM; i++)
{
    read_dir(i);
    if (dirEnt[12]==1) myprintf("%8d  %.8s.%s\r\n",*(int*)(dirEnt+28),dirEnt,dirEnt+8);
        else break;
}
```

这里用表项第 12 字节是否为 0 来区分空项与占用项。

至此内核修改完成，测试效果如下：（助教可运行 `\src\os\myOS.bxrc` 查看动态效果）

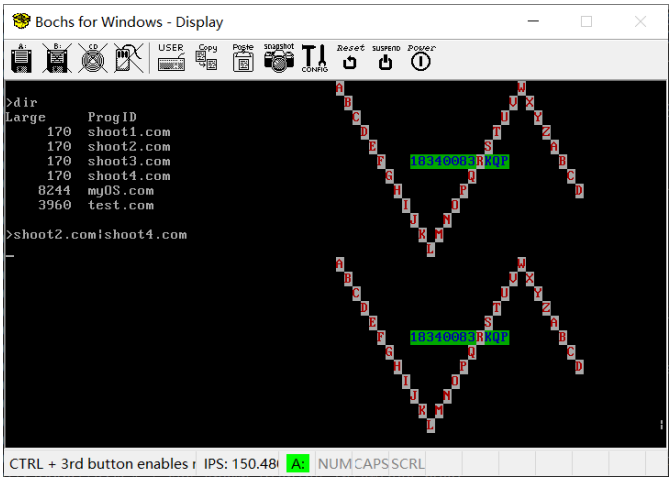

图 14: dir及多进程测试

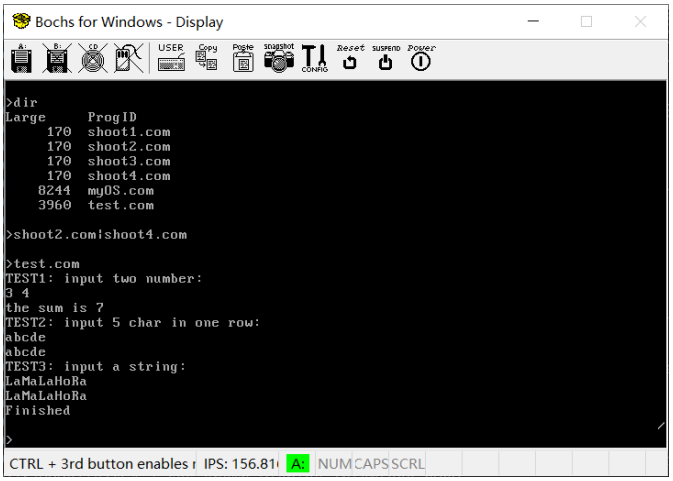

图 15: 系统调用测试

可以看到，dir 命令正确，多个射字母进程可以同时执行，系统调用服务正常，说明用户程序的查找、加载、执行正确。

5.5 输出簇号程序

此处对要求做一些修改，原来要求在用户程序中完成簇号的输出，但正常来说用户程序不应该有这个权限（即便用户程序需要，也应当由操作系统代为完成），且内核中的“加载用户程序”已经实现了这个“读链表”的功能，不必再重复一次。

因此，在“数据区加载到内存”部分，增加一个输出：

```
1 myprintf("%s: ",proList[j].pname);
2 for(short st=(dirEnt[27]<<8)+dirEnt[26], bx=0x100;
3     st!=0xFFF;
4     st=next_cluster(st), bx+=0x200)
5     {
6         load_client(seg_addr,bx,st);
7         myprintf("%d ",st);
8     }
9 myprintf("\r\n");
```

这样就实现了“列出一个文件的所有簇号”。效果如下：

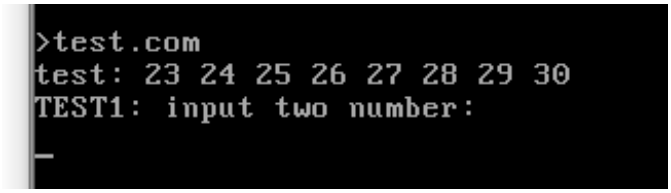

图 16: 输出簇号测试

可以看到，test.com 的 8 个簇号正确输出。

6 实验总结

这次实验是做基础的 FAT12 文件系统，未涉及到 FAT 表、根目录项的修改，只是读取，因此较为简单，但编程工作量也不小。

6.1 写映像管理工具

吸引我做这个实验的原因是老师说的：“实现了文件系统之后，就可以把一个程序直接拖到映像盘中。”这是开发过程极为便利的操作，如果还有后续实验的话我会果断选择先做这个。但后来发现能实现这一功能的 WinImage 软件没有免费正版。又基于一些其他原因，于是决定自己写一个：

1. 在理解了 FAT12 标准的基础上，这就是一些文件读写操作（把软盘、文件读进来改改再写回去），之前的计网实验多少做了些铺垫，甚至有些操作是能复用的（如提取文件后缀名）；
2. 网课时没有做 FAT12 仿真，而自己写一个映像管理工具的话，一定程度上相当于仿真；
3. 在这里实现的文件操作，将来还可以继续复用到内核里。

事实上仅这一映像管理工具就花了差不多一天时间，通过这一天，把 `getFatEntry`、`modifyFatEntry` 这两个最抠细节的过程写好了，把链表操作、根目录项文件信息记录写好了，这样一写，对 FAT12 格式的印象就更深刻了。

当然，这个工具现在还是只有一个功能的简陋工具。

6.2 试错细节

常见的错误有：

1. FAT 表项的下标计算，这里涉及了跨字节的操作，需要保持清晰的逻辑，才能处理好各种位运算；
2. 逻辑编号与物理编号可能混用，例如公式“逻辑扇区号=簇号+31”，这里两个编号都是从 0 开始的，因此最好的做法是，全部采用从 0 开始的逻辑编号，只有在 `int 13h` 写磁盘时才用物理扇区号（从 1 开始）；
3. 文件名匹配，若根目录项中存文件名的 8 个字节全占满，则需要注意字符串以 \000 结束的问题。

调试时上来就 bochs 是低效的，尽量先肉眼检查软盘中的存储是否遵循格式、FAT 链表是否有错。

参考文献

[1] 凌应标,网课2,第13页

[2] 凌应标,网课4根目录,第15页