北京交通大学实验报告

课程名称 ： **操作系统**
实验题目 ： **FAT文件系统模拟与实现**
学号 ： **21281280**
姓名 ： **柯劲帆**
班级 ： **物联网2101班**
指导老师 ： **何永忠**
报告日期 ： **2023年12月31日**

目录

1. 开发运行环境和工具

操作系统	Linux内核版本	处理器	GCC版本
Deepin 20.9	5.18.17-amd64-desktop-hwe	Intel(R) Core(TM) i3-2330M CPU @ 2.20GHz	gcc (Uos 8.3.0.3-3+rebuild) 8.3.0

- **其他工具**:
  - **编辑**：VSCode
  - **编译和运行**：Terminal

2. FAT磁盘格式化操作和映像文件生成

FAT的引导扇区各个字段的定义和说明如下：

字节位移	字段长度(字节)	字段名称
0x00	3	跳转指令(Jump Instruction)
0x03	8	OEM ID
0x0B	25	BPB

字节位移	字段长度(字节)	字段名称
0x24	26	扩展BPB
0x3E	448	引导程序代码(Bootstrap Code)
0x01FE	4	扇区结束标识符(0xAA55)

FAT16分区的BPB字段为：

字节位移	字段长度(字节)	例值	名称、定义和描述
0x0B	2	0x0200	扇区字节数(Bytes Per Sector) 硬件扇区的大小。本字段合法的十进制值有512、1024、2048和4096。对大多数磁盘来说，本字段的值为512
0x0D	1	0x10	每簇扇区数(Sectors Per Cluster) 一个簇中的扇区数。由于FAT16文件系统只能跟踪有限个簇(最多为65536个)。因此，通过增加每簇的扇区数可以支持最大分区数。分区的缺省的簇的大小取决于该分区的大小。本字段合法的十进制值有1、2、4、8、16、32、64和128。导致簇大于32KB(每扇区字节数*每簇扇区数)的值会引起磁盘错误和软件错误
0x0e	2	0x0006	保留扇区数(Reserved Sector) 第一个FAT开始之前的扇区数，包括引导扇区。
0x10	1	0x02	FAT数(Number of FAT)该分区上FAT的副本数。本字段的值一般为2
0x11	2	0x0200	根目录项数(Root Entries) 能够保存在该分区的根目录文件夹中的32个字节长的文件和文件夹名称项的总数。在一个典型的硬盘上，本字段的值为512。其中一个项常常被用作卷标号(Volume Label)，长名称的文件和文件夹每个文件使用多个项。文件和文件夹项的最大数一般为511，但是如果使用的长文件名，往往都达不到这个数。
0x13	2	0x0000	小扇区数(Small Sector) 该分区上的扇区数，表示为16位(<65536)。对大于65536个扇区的分区来说，本字段的值为0，而使用大扇区数来取代它。
0x15	1	0xF8	媒体描述符(Media Descriptor)提供有关媒体被使用的信息。值0xF8表示硬盘，0xF0表示高密度的3.5寸软盘。媒体描述符要用于MS-DOS FAT16磁盘，在Windows 2000中未被使用
0x16	2	0x00F5	每FAT扇区数(Sectors Per FAT) 该分区上每个FAT所占用的扇区数。计算机利用这个数和FAT数以及隐藏扇区数来决定根目录在哪里开始。计算机还可以根据根目录中的项数(512)决定该分区的用户数据区从哪里开始
0x18	2	0x003F	每道扇区数(Sectors Per Trark)

字节位移	字段长度(字节)	例值	名称、定义和描述
0x1A	2	0x00FF	磁头数(Number of head)
0x1C	4	0x000000400	隐藏扇区数(Hidden Sector) 该分区上引导扇区之前的扇区数。在引导序列计算到根目录和数据区的绝对位移的过程中使用了该值
0x20	4	0x000F4C00	大扇区数(Large Sector) 如果小扇区数字段的值为0，本字段就包含该FAT16分区中的总扇区数。如果小扇区数字段的值不为0，那么本字段的值为0

FAT16分区的扩展BPB字段为：

字节位移	字段长度(字节)	图8对应取值	名称、定义和描述
0x24	1	0x80	物理驱动器号(Physical Drive Number) 与BIOS物理驱动器号有关。软盘驱动器被标识为0x00，物理硬盘被标识为0x80，而与物理磁盘驱动器无关。一般地，在发出一个INT13h BIOS调用之前设置该值，具体指定所访问的设备。只有当该设备是一个引导设备时，这个值才有意义
0x25	1	0x01	保留(Reserved) FAT16分区一般将本字段的值设置为1
0x26	1	0x29	扩展引导标签(Extended Boot Signature) 本字段必须要有能被Windows 2000所识别的值0x28或0x29
0x27	2	0xABA13358	卷序号(Volume Serial Number) 在格式化磁盘时所产生的一个随机序号，它有助于区分磁盘
0x2B	11	"NO NAME"	卷标(Volume Label) 本字段只能使用一次，它被用来保存卷标号。现在，卷标被作为一个特殊文件保存在根目录中
0x36	8	"FAT16"	文件系统类型(File System Type) 根据该磁盘格式，该字段的值可以为FAT、FAT12或FAT16

因此，各参数初始化如下：

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <time.h>
#include <sys/time.h>

#define BS_jmpBoot_Offset 0x00  // 跳转指令偏移
#define BS_jmpBoot_Len 3    // 跳转指令长度
#define BS_OEMName_Offset 0x03  // 厂商名称偏移
#define BS_OEMName_Len 8    // 厂商名称长度
```

```
#define BPB_BytsPerSec_Offset 0x0B  // 每扇区字节数偏移
#define BPB_BytsPerSec_Len 2    // 每扇区字节数长度
#define BPB_SecPerClus_Offset 0x0D  // 每簇扇区数偏移
#define BPB_SecPerClus_Len 1    // 每簇扇区数长度
#define BPB_RsvdSecCnt_Offset 0x0E  // 保留扇区数 (引导扇区的扇区数）偏移
#define BPB_RsvdSecCnt_Len 2    // 保留扇区数 (引导扇区的扇区数）长度
#define BPB_NumFATs_Offset 0x10 // FAT的份数偏移
#define BPB_NumFATs_Len 1   // FAT的份数长度
#define BPB_RootEntCnt_Offset 0x11  // 根目录可容纳的目录项数偏移
#define BPB_RootEntCnt_Len 2    // 根目录可容纳的目录项数长度
#define BPB_TotSec16_Offset 0x13    // 扇区总数偏移
#define BPB_TotSec16_Len 2  // 扇区总数长度
#define BPB_Media_Offset 0x15   // 介质描述符偏移
#define BPB_Media_Len 1 // 介质描述符长度
#define BPB_FATSz16_Offset 0x16 // 每个FAT表扇区数偏移
#define BPB_FATSz16_Len 2   // 每个FAT表扇区数长度
#define BPB_SecPerTrk_Offset 0x18   // 每磁道扇区数偏移
#define BPB_SecPerTrk_Len 2 // 每磁道扇区数长度
#define BPB_NumHeads_Offset 0x1A    // 磁头数偏移
#define BPB_NumHeads_Len 2  // 磁头数长度
#define BPB_HiddSec_Offset 0x1C // 隐藏扇区数偏移
#define BPB_HiddSec_Len 4   // 隐藏扇区数长度
#define BPB_TotSec32_Offset 0x20    // 偏移 如果BPB_TotSec16是0，由这个值记录扇区数
#define BPB_TotSec32_Len 4   // 长度 如果BPB_TotSec16是0，由这个值记录扇区数
#define BS_DrvNum_Offset 0x24   // int 13h的驱动器号偏移
#define BS_DrvNum_Len 1 // int 13h的驱动器号长度
#define BS_Reservedl_Offset 0x25    // 偏移 未使用
#define BS_Reservedl_Len 1  // 长度 未使用
#define BS_BootSig_Offset 0x26  // 扩展引导标记偏移
#define BS_BootSig_Len 1    // 扩展引导标记长度
#define BS_VolID_Offset 0x27    // 卷序列号偏移
#define BS_VolID_Len 4  // 卷序列号长度
#define BS_VolLab_Offset 0x2B   // 卷标偏移
#define BS_VolLab_Len 11    // 卷标长度
#define BS_FileSysType_Offset 0x36  // 文件系统类型偏移
#define BS_FileSysType_Len 8    // 文件系统类型长度
#define DBR_BootCode_Offset 0x3E  // 引导代码及其他偏移
#define DBR_BootCode_Len 448    // 引导代码及其他长度
#define DBR_EndMark_Offset 0x1FE    // 结束标志偏移
#define DBR_EndMark_Len 2   // 结束标志长度
#define DBR_EndMark 0xAA55  // 结束标志

typedef struct {
    short BytsPerSec;   // 每扇区字节数
    char SecPerClus;    // 每簇扇区数
    short RsvdSecCnt;   // 保留扇区数 (引导扇区的扇区数）
    char NumFATs;   // FAT的份数
    short RootEntCnt;   // 根目录可容纳的目录项数
    short TotSec16; // 扇区总数
    char Media; // 介质描述符
    short FATSz16;  // 每个FAT表扇区数
    short SecPerTrk;    // 每磁道扇区数
    short NumHeads; // 磁头数
    int HiddSec;    // 隐藏扇区数
    int TotSec32;   // 替代扇区数
```

```
    char DrvNum;    // int 13h的驱动器号
    char Reservedl; // 未使用
    char BootSig;   // 扩展引导标记
    int VolID;  // 卷序列号
    int TotalSize;  // 总磁盘大小
    int current_dir;    // 当前目录的首地址
} DISK;


DISK init_disk_attr() {
    DISK disk;
    disk.BytsPerSec = 0x0200;
    disk.SecPerClus = 0x01;
    disk.RsvdSecCnt = 0x0001;
    disk.NumFATs = 0x01;
    disk.RootEntCnt = 0x0200;
    disk.TotSec16 = 0x2000;
    disk.Media = 0xf8;
    disk.FATSz16 = 0x0020;
    disk.SecPerTrk = 0x0020;
    disk.NumHeads = 0x0040;
    disk.HiddSec = 0x00000000;
    disk.TotSec32 = 0x00000000;
    disk.DrvNum = 0x80;
    disk.Reservedl = 0x00;
    disk.BootSig = 0x29;
    disk.VolID = 0x00000000;
    disk.TotalSize = 0x400000;
    disk.current_dir = 0x4200;
    return disk;
}
```

初始化和格式化磁盘代码如下：

```
void init_file_system(char disk_name[13], DISK disk_attr) {
    FILE* fp = fopen(disk_name, "wb");
    // 跳转指令
    fseek(fp, BS_jmpBoot_Offset, 0);
    char jmpBoot[BS_jmpBoot_Len] = { 0xEB, 0x3C, 0x90 };
    fwrite(jmpBoot, 1, BS_jmpBoot_Len, fp);

    // 厂商名称
    fseek(fp, BS_OEMName_Offset, 0);
    char OEMName[BS_OEMName_Len] = { 'K', 'e', 'J', 'F', ' ', ' ', ' ', ' '
};
    fwrite(OEMName, 1, BS_OEMName_Len, fp);

    // 每扇区字节数
    fseek(fp, BPB_BytsPerSec_Offset, 0);
    fwrite(&disk_attr.BytsPerSec, BPB_BytsPerSec_Len, 1, fp);

    // 每簇扇区数
    fseek(fp, BPB_SecPerClus_Offset, 0);
    fwrite(&disk_attr.SecPerClus, BPB_SecPerClus_Len, 1, fp);

```

```
    // 保留扇区数（引导扇区的扇区数）
    fseek(fp, BPB_RsvdSecCnt_Offset, 0);
    fwrite(&disk_attr.RsvdSecCnt, BPB_RsvdSecCnt_Len, 1, fp);

    // FAT的份数
    fseek(fp, BPB_NumFATs_Offset, 0);
    fwrite(&disk_attr.NumFATs, BPB_NumFATs_Len, 1, fp);

    // 根目录可容纳的目录项数
    fseek(fp, BPB_RootEntCnt_Offset, 0);
    fwrite(&disk_attr.RootEntCnt, BPB_RootEntCnt_Len, 1, fp);

    // 扇区总数
    fseek(fp, BPB_TotSec16_Offset, 0);
    fwrite(&disk_attr.TotSec16, BPB_TotSec16_Len, 1, fp);

    // 介质描述符
    fseek(fp, BPB_Media_Offset, 0);
    fwrite(&disk_attr.Media, BPB_Media_Len, 1, fp);

    // 每个FAT表扇区数
    fseek(fp, BPB_FATSz16_Offset, 0);
    fwrite(&disk_attr.FATSz16, BPB_FATSz16_Len, 1, fp);

    // 每磁道扇区数
    fseek(fp, BPB_SecPerTrk_Offset, 0);
    fwrite(&disk_attr.SecPerTrk, BPB_SecPerTrk_Len, 1, fp);

    // 磁头数
    fseek(fp, BPB_NumHeads_Offset, 0);
    fwrite(&disk_attr.NumHeads, BPB_NumHeads_Len, 1, fp);

    // 隐藏扇区数
    fseek(fp, BPB_HiddSec_Offset, 0);
    fwrite(&disk_attr.HiddSec, BPB_HiddSec_Len, 1, fp);

    // 替代扇区数
    fseek(fp, BPB_TotSec32_Offset, 0);
    fwrite(&disk_attr.TotSec32, BPB_TotSec32_Len, 1, fp);

    // int 13h的驱动器号
    fseek(fp, BS_DrvNum_Offset, 0);
    fwrite(&disk_attr.DrvNum, BS_DrvNum_Len, 1, fp);

    // 未使用
    fseek(fp, BS_Reserved1_Offset, 0);
    fwrite(&disk_attr.Reserved1, BS_Reserved1_Len, 1, fp);

    // 扩展引导标记
    fseek(fp, BS_BootSig_Offset, 0);
    fwrite(&disk_attr.BootSig, BS_BootSig_Len, 1, fp);

    // 卷序列号
    fseek(fp, BS_VolID_Offset, 0);
    fwrite(&disk_attr.VolID, BS_VolID_Len, 1, fp);

```

```
    // 卷标
    fseek(fp, BS_VolLab_Offset, 0);
    char VolLab[BS_VolLab_Len] = {'K', 'e', 'J', 'i', 'n', 'g', 'f', 'a', 'n', ' ', ' '};
    fwrite(VolLab, 1, BS_VolLab_Len, fp);

    // 文件系统类型
    fseek(fp, BS_FileSysType_Offset, 0);
    char FileSysType[BS_FileSysType_Len] = {'F', 'A', 'T', '1', '6', ' ', ' ', ' '};
    fwrite(FileSysType, 1, BS_FileSysType_Len, fp);

    // 引导代码等
    fseek(fp, DBR_BootCode_Offset, 0);
    char BootCode[DBR_BootCode_Len] = { 0 };
    fwrite(BootCode, 1, DBR_BootCode_Len, fp);

    // 结束标志
    fseek(fp, DBR_EndMark_Offset, 0);
    short EndMark = DBR_EndMark;
    fwrite(&EndMark, DBR_EndMark_Len, 1, fp);

    // FAT分区表
    fseek(fp, 0x0200, 0);
    int FAT_Len = disk_attr.FATSz16 * disk_attr.BytsPerSec * disk_attr.NumFATs; // 每个FAT表扇区数 * 扇区字节数 * FAT的份数
    char* FAT_data = (char*)malloc(sizeof(char) * FAT_Len);
    fwrite(FAT_data, 1, FAT_Len, fp);

    // DATA段
    fseek(fp, 0x0200 + FAT_Len, 0);
    int DATA_Len = disk_attr.TotalSize - 0x0200 - FAT_Len;
    char* DATA_data = (char*)malloc(sizeof(char) * DATA_Len);
    fwrite(DATA_data, 1, DATA_Len, fp);

    fclose(fp);
}
```

调用这个函数生成.img虚拟磁盘镜像挂载到linux上：

```
> gcc ./main.c -o main
> ./main
> mkdir /media/cdrom
> sudo mount -o loop KJF_disk.img /media/cdrom
> cd /media/cdrom
```

发现可以正常认盘和使用。

查看其二进制：

```
> hexdump KJF_disk.img -C
00000000  eb 3c 90 4b 65 4a 46 20  20 20 20 00 02 01 01 00  |.<.KeJF
.....|
00000010  01 00 02 00 20 f8 20 00  20 00 40 00 00 00 00 00  |.... . .
.@.....|
00000020  00 00 00 00 80 00 29 00  00 00 00 4b 65 4a 69 6e
|......)....KeJin|
00000030  67 66 61 6e 20 20 46 41  54 31 36 20 20 20 00 00  |gfan  FAT16
..|
00000040  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.|
00000200  00 00 00 00 00 00 00 00  00 00 00 00 00 00 00 00
|................|
*
00400000
```

FAT文件格式正确。

3. 目录

目录项字段定义：

偏移地址	字节数	说明	
0x00~0x07	8	文件名	
0x08~0x0A	3	扩展名	
0x0B	1	属 性	00000000(读写) 00000001(只读) 00000010(隐藏) 00000100(系统) 00001000(卷标)00010000(目录) 00100000(归档)
0x0C~0x15	10	保留	
0x16~0x17	2	文件最近修改时间	
0x18~0x19	2	文件最近修改日期	
0x1A~0x1B	2	文件的首簇号	
0x1C~0x1F	4	文件长度大小	

因此代码如下：

```
int mkdir(char dir_name[11], char disk_name[13], DISK disk_attr) {
    FILE* fp = fopen(disk_name, "rb+");
    unsigned short dir_firstBlockNum = find_free_blocks(disk_attr, fp);
    if (dir_firstBlockNum == 0xFFFF) return -1;

    fseek(fp, 0x200 + dir_firstBlockNum * 2, 0);
    short new_index = 0xFFFF;
    fwrite(&new_index, 2, 1, fp);

    int start_byte = disk_attr.current_dir;
    int byte_line_num = 0;
    while (byte_line_num < disk_attr.BytsPerSec) {
        fseek(fp, start_byte + byte_line_num, 0);
        int byte_line[4] = { 0 };
        fread(byte_line, 4, 3, fp);
        if (!byte_line[0] && !byte_line[1] && !byte_line[2]) break;
        byte_line_num += 0x20;
    }
    if (byte_line_num >= disk_attr.BytsPerSec) return -1;
    fseek(fp, start_byte + byte_line_num, 0);
    fwrite(dir_name, 1, 11, fp);
    short dir_attr_byte = 0b00010000;
    fwrite(&dir_attr_byte, 1, 1, fp);

    short dir_reserved_byte[6] = { 0 };
    fwrite(dir_reserved_byte, 2, 5, fp);

    struct tm time = get_time();
    short dir_time_byte = time.tm_hour * 2048 + time.tm_min * 32 +
time.tm_sec / 2;
    fwrite(&dir_time_byte, 2, 1, fp);
    short dir_date_byte = (time.tm_year + 1900 - 1980) * 512 + time.tm_mon *
32 + time.tm_mday;
    fwrite(&dir_date_byte, 2, 1, fp);
    fwrite(&dir_firstBlockNum, 2, 1, fp);
    short dir_size = disk_attr.BytsPerSec * disk_attr.SecPerClus;
    fwrite(&dir_size, 2, 1, fp);

    int new_block = 0x7800 + disk_attr.BytsPerSec * disk_attr.SecPerClus *
dir_firstBlockNum;
    fseek(fp, new_block, 0);
    char dot[11] = { '.', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' '
};
    fwrite(dot, 1, 11, fp);
    fwrite(&dir_attr_byte, 1, 1, fp);
    fwrite(dir_reserved_byte, 2, 5, fp);
    fwrite(&dir_time_byte, 2, 1, fp);
    fwrite(&dir_date_byte, 2, 1, fp);
    short dir_oriBlock_byte = 0;
    fwrite(&dir_oriBlock_byte, 2, 1, fp);
    unsigned int dir_oriSize = 0;
    fwrite(&dir_oriSize, 4, 1, fp);

    fseek(fp, new_block + 0x20, 0);
```

```
    char dotdot[11] = { '.', '.', ' ', ' ', ' ', ' ', ' ', ' ', ' ', ' ', '
' };
    fwrite(dotdot, 1, 11, fp);
    fwrite(&dir_attr_byte, 1, 1, fp);
    fwrite(dir_reserved_byte, 2, 5, fp);
    short dir_oriTime_byte = 0;
    fwrite(&dir_oriTime_byte, 2, 1, fp);
    short dir_oriDate_byte = (1970 - 1980 + 1900) * 512 + 1 * 32 + 1;
    fwrite(&dir_oriDate_byte, 2, 1, fp);
    fwrite(&dir_oriBlock_byte, 2, 1, fp);
    fwrite(&dir_oriSize, 4, 1, fp);

    fclose(fp);
    return 0;
}

struct tm get_time() {
    struct timeval tv;
    struct timezone tz;
    struct tm *t;

    gettimeofday(&tv, &tz);
    t = localtime(&tv.tv_sec);
    return *t;
}
```

4. 文件

```
int touch(char file_name[11], char disk_name[13], DISK disk_attr) {
    FILE* contentFile = fopen("./data.txt", "r");
    char *content = (char*)malloc(2000 * sizeof(char));
    fread(content, 1, 2000, contentFile);
    fclose(contentFile);
    FILE* fp = fopen(disk_name, "rb+");
    int file_blockNum = strlen(content) / (disk_attr.BytsPerSec *
disk_attr.SecPerClus) + 1;
    unsigned short* index = (short*)malloc(sizeof(short) * file_blockNum);
    for (int i = 0; i < file_blockNum; i++) {
        index[i] = find_free_blocks(disk_attr, fp);
    }
    for (int i = 0; i < file_blockNum - 1; i++) {
        fseek(fp, 0x200 + index[i] * 2, 0);
        fwrite(&index[i + 1], 2, 1, fp);
    }
    fseek(fp, 0x200 + index[file_blockNum - 1] * 2, 0);
    short last_index = 0xFFFF;
    fwrite(&last_index, 2, 1, fp);

    int start_byte = disk_attr.current_dir;
    int byte_line_num = 0;
    while (byte_line_num < disk_attr.BytsPerSec) {
        fseek(fp, start_byte + byte_line_num, 0);
```

```
        int byte_line[4] = { 0 };
        fread(byte_line, 4, 3, fp);
        if (!byte_line[0] && !byte_line[1] && !byte_line[2]) break;
        byte_line_num += 0x20;
    }
    if (byte_line_num >= disk_attr.BytsPerSec) return -1;
    fseek(fp, start_byte + byte_line_num, 0);
    fwrite(file_name, 1, 11, fp);
    short file_attr_byte = 0b00000000;
    fwrite(&file_attr_byte, 1, 1, fp);

    short file_reserved_byte[6] = { 0 };
    fwrite(file_reserved_byte, 2, 5, fp);

    struct tm time = get_time();
    short file_time_byte = time.tm_hour * 2048 + time.tm_min * 32 +
time.tm_sec / 2;
    fwrite(&file_time_byte, 2, 1, fp);
    short file_date_byte = (time.tm_year + 1900 - 1980) * 512 + time.tm_mon
* 32 + time.tm_mday;
    fwrite(&file_date_byte, 2, 1, fp);
    fwrite(&index[0], 2, 1, fp);
    unsigned int file_size = strlen(content);
    fwrite(&file_size, 4, 1, fp);

    for (int i = 0; i < file_blockNum; i++) {
        int new_block = 0x7800 + disk_attr.BytsPerSec * disk_attr.SecPerClus
* index[i];
        fseek(fp, new_block, 0);
        fwrite(content,
            file_size > disk_attr.BytsPerSec * disk_attr.SecPerClus ?
disk_attr.BytsPerSec * disk_attr.SecPerClus : file_size,
            1,
            fp
        );
        content += disk_attr.BytsPerSec * disk_attr.SecPerClus;
        file_size -= disk_attr.BytsPerSec * disk_attr.SecPerClus;
    }
    fclose(fp);
    return 0;
}
```