

实验四 FAT文件系统的实现

实验目标

- 熟悉FAT16的存储结构，利用FUSE实现一个FAT文件系统

实验环境

- vmware/virtual box
- OS: Ubuntu 18.04LTS
- Linux内核版本: 5.9.0

实验时间安排

注：此处为实验发布时的安排计划，请以课程主页和课程群内最新公告为准

- 6.4 晚实验课，讲解及检查实验
- 6.11晚实验课，检查实验
- 6.18晚实验课，检查实验
- 6.25晚实验课，实验补检查

实验代码提交

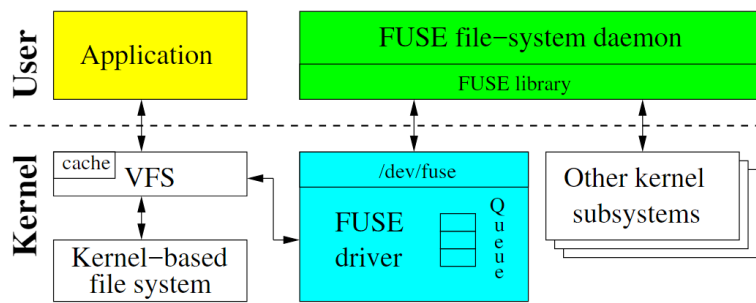
- 1 本次实验以现场检查为主，主要在于考察实验内容、过程思路及确认结果。本实验只需提交代码，无需写实验报告。提交代码仅用于检查实验是否为自己独立完成，不另算分。

- 提交代码simple_fat16.c
- 将上述文件压缩
 - 格式为 .7z/.rar/.zip
 - 命名格式为 学号姓名实验4，如果上传后需要修改，请将文件重新命名为学号姓名实验4_修改n (n为修改版本)，以最后修改版本为准。
- 如PB10001000张三实验4.zip, PB10001000张三实验4_修改n.zip

实验评分

1. 任务一，满分5分
 - 能够运行tree/ls命令查看文件目录结构 (2.5分)
 - 能够运行cat/head/tail命令查看目录下文件内容 (2.5分)
2. 任务二，满分5分
 - 能够运行touch命令创建新文件，要保证文件属性的正确填写 (2.5分)
 - 能够运行rm命令删除已有文件，要保证簇的正确释放 (2.5分)
3. 实验进阶分(满分2分)
 - 待定

背景知识：FUSE概述



- FUSE (Filesystem in Userspace, 用户态文件系统) 是一个实现在用户空间的文件系统框架，通过FUSE内核模块的支持，使用者只需要根据fuse提供的接口实现具体的文件操作就可以实现一个文件系统。
- FUSE主要由三部分组成：FUSE内核模块、用户空间库libfuse以及挂载工具fusermount：
 1. fuse内核模块：实现了和VFS的对接，实现了一个能被用户空间进程打开的设备。
 2. fuse库libfuse：负责和内核空间通信，接收来自/dev/fuse的请求，并将其转化为一系列的函数调用，将结果写回到/dev/fuse；提供的函数可以对fuse文件系统进行挂载卸载、从linux内核读取请求以及发送响应到内核。
 3. 挂载工具：实现对用户态文件系统的挂载。
- 更多详细内容可参考[这篇文章](#)。

配置FUSE环境

- linux kernel在2.6.14后添加了FUSE模块，因此对于目前的大多数发行版来说只需要安装libfuse库即可。
- 在<https://github.com/libfuse/libfuse>里下载libfuse 2.9.5，然后编译安装：

```

1 $ sudo apt install pkg-config pkgconf
2 $ wget -O libfuze-2.9.5.zip
  https://codeload.github.com/libfuse/libfuse/zip/fuse_2_9_5
3 unzip libfuze-2.9.5.zip
4 $ cd libfuse-fuse_2_9_5/
5 $ ./makeconf.sh
6 $ ./configure --prefix=/usr
7 $ make -j4
8 $ sudo make install

```

- **编译出错处理** 在执行./makeconf.sh可能会遇到“libtoolize: not found”或“autoreconf: not found”，需要安装libtool和autoconf软件包，ubuntu下可以通过下面的命令安装：

```

1 sudo apt install libtool
2 sudo apt install autoconf

```

测试FUSE

- 通过libfuse-fuse_2_9_5/example下的fusexmp进行测试：

```

1 cd example
2 mkdir fuse_test_dir
3 ./fusexmp -d fuse_test_dir

```

- 这时候在文件管理器中打开fuse_test_dir目录，可以看到当前Linux系统的“/”被挂载到这个目录下。
- 结束测试可以直接在当前终端中Ctrl + C结束程序，或者在新的终端中输入：

```
1 | fusermount -u fuse_test_dir
```

- **提示** 当执行用户自己实现的fuse程序时，如果出现错误 ("fuse: bad mount point ...Transport endpoint is not connected")，可通过执行上面这条命令卸载对应的文件夹来解决。

任务一：实现FAT文件系统读操作

1.1 FAT文件系统初识

FAT(File Allocation Table)是“文件分配表”的意思。顾名思义，就是用来记录文件所在位置的表格，它对于硬盘的使用是非常重要的，假若丢失文件分配表，那么硬盘上的数据就会因无法定位而不能使用了。不同的操作系统所使用的文件系统不尽相同，在个人计算机上常用的操作系统中，MS-DOS 6.x及以下版本使用FAT16。操作系统根据表现整个磁盘空间所需要的簇数量来确定使用多大的FAT。所谓簇就是磁盘空间的配置单位，就象图书馆内一格一格的书架一样。FAT16使用了16位的空间来表示每个扇区(Sector)配置文件的情形，故称之为FAT16。

从上述描述中我们得知，一个FAT分区或者磁盘能够使用的簇数是 $2^{16}=65536$ 个，因此簇大小是一个变化值，通常与分区大小有关，计算方式为：(磁盘大小/簇个数)向上按2的幂取整。在FAT16文件系统中，由于兼容性等原因，簇大小通常不超过32K，这也是FAT分区容量不能超过2GB的原因。

1.2 专有名词

名词	释义
簇	文件的最小空间分配单元，通常为若干个扇区，每个文件最小将占用一个簇
扇区	磁盘上的磁道被等分为若干个弧段，这些弧段被称为扇区，硬盘的读写以扇区为基本单位

1.3 磁盘分布

一个FAT分区或磁盘的结构布局							
内容	主引导区	文件系统信息扇区	额外的保留空间	文件分配表1(FAT表1)	文件分配表2(FAT表2)	根目录(Root directory)	数据区()
大小(Bytes)	保留扇区数*扇区大小			FAT扇区数*扇区大小	FAT扇区数*扇区大小	根目录条目数*文件条目大小	剩下的磁盘空间

一个FAT文件系统包括四个不同的部分。

- **保留扇区**，位于最开始的位置。第一个保留扇区是引导扇区（分区启动记录）。它包括一个称为基本输入输出参数块的区域（包括一些基本的文件系统信息尤其是它的类型和其它指向其它扇区的指针），通常包括操作系统的启动调用代码。保留扇区的总数记录在引导扇区中的一个参数中。引导扇区中的重要信息可以被DOS和OS/2中称为驱动器参数块的操作系统结构访问。
- **FAT区域**。它包含有两份文件分配表，这是出于系统冗余考虑，尽管它很少使用。它是分区信息的映射表，指示簇是如何存储的。
- **根目录区域**。它是在根目录中存储文件和目录信息的目录表。在FAT32下它可以存在分区中的任何位置，但是在FAT16中它永远紧随FAT区域之后。
- **数据区域**。这是实际的文件和目录数据存储的区域，它占据了分区的绝大部分。通过简单地在FAT中添加文件链接的个数可以任意增加文件大小和子目录个数（只要有空簇存在）。然而需要注意的

是每个簇只能被一个文件占有：如果在32KB大小的簇中有一个1KB大小的文件，那么31KB的空间就浪费掉了。

1.3.1 启动扇区详解

名称	偏移(字节)	长度(字节)	说明
BS_jmpBoot	0x00	3	跳转指令（跳过开头一段区域）
BS_OEMName	0x03	8	OEM名称，Windows操作系统没有针对这个字段做特殊的逻辑，理论上说这个字段只是一个标识字符串，但是有一些FAT驱动程序可能依赖这个字段的指定值。常见值是"MSWIN4.1"和"FrLdr1.0"
BPB_BytsPerSec	0x0b	2	每个扇区的字节数。基本输入输出系统参数块从这里开始。
BPB_SecPerClus	0x0d	1	每簇扇区数
BPB_RsvdSecCnt	0x0e	2	保留扇区数（包括主引导区）
BPB_NumFATS	0x10	1	文件分配表数目，FAT16文件系统中为0x02,FAT2作为FAT1的冗余
BPB_RootEntCnt	0x11	2	最大根目录条目个数
BPB_TotSec16	0x13	2	总扇区数（如果是0，就使用偏移0x20处的4字节值）
BPB_Media	0x15	1	介质描述：F8表示为硬盘，F0表示为软盘
BPB_FATSz16	0x16	2	每个文件分配表的扇区数（FAT16专用）
BPB_SecPerTrk	0x18	2	每磁道的扇区数
BPB_NumHeads	0x1a	2	磁头数
BPB_HiddSec	0x1c	4	隐藏扇区数
BPB_TotSec32	0x20	4	总扇区数（如果0x13处不为0，则该处应为0）
BS_DrvNum	0x24	1	物理驱动器号，指定系统从哪里引导。
BS_Reserved1	0x25	1	保留字段。这个字段设计时被用来指定引导扇区所在的
BS_BootSig	0x26	1	扩展引导标记（Extended Boot Signature）
BS_VolIID	0x27	4	卷序列号，例如0
BS_VolLab	0x2B	11	卷标，例如"NO NAME "
BS_FilSysType	0x36	8	文件系统类型，例如"FAT16"
Reserved2	0x3E	448	引导程序
Signature_word	0x01FE	2	引导区结束标记

根据上面的DBR扇区，我们可以算出各FAT的偏移地址，根目录的偏移地址，数据区的偏移地址。

FAT1偏移地址：保留扇区之后就是FAT1。因此可以得到，FAT1的偏移地址就是

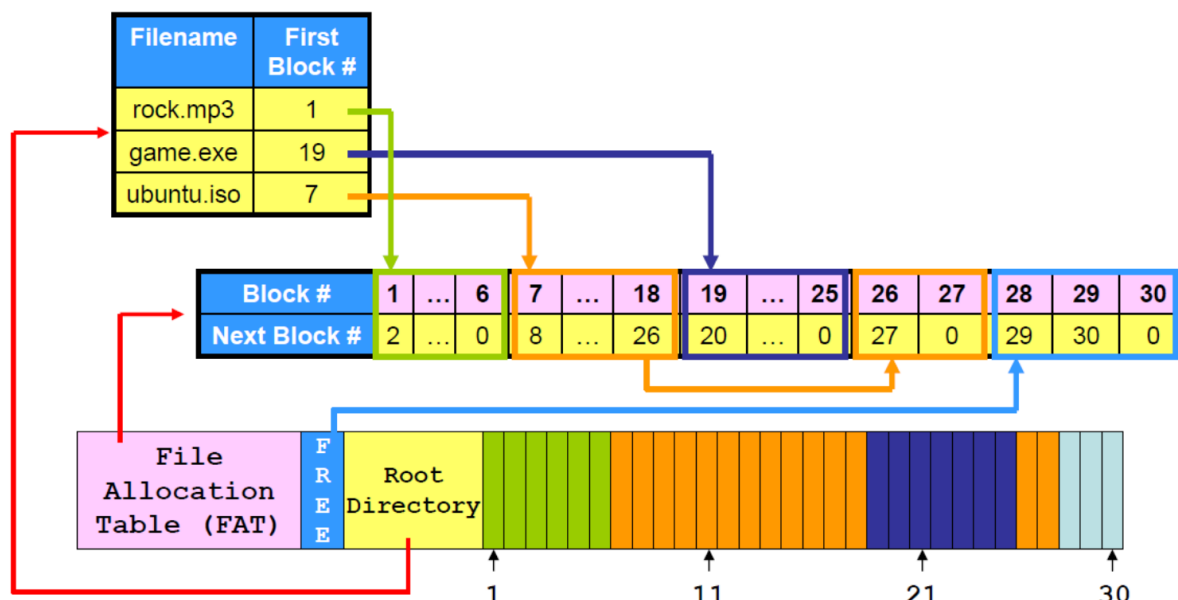
$\text{BPB_RsvdSecCnt} * \text{BPB_BytsPerSec}$

根目录偏移地址：FAT1表后两个FAT表地址就是根目录区，即FAT1偏移地址

$+ \text{BPB_NumFATS} * \text{BPB_FATSz16} * \text{BPB_BytsPerSec}$ 。

数据区的偏移地址：根目录偏移地址+根目录大小，即根目录偏移地址+ $\text{BPB_RootEntCnt} * 32$

1.4 FAT文件系统磁盘分布详解

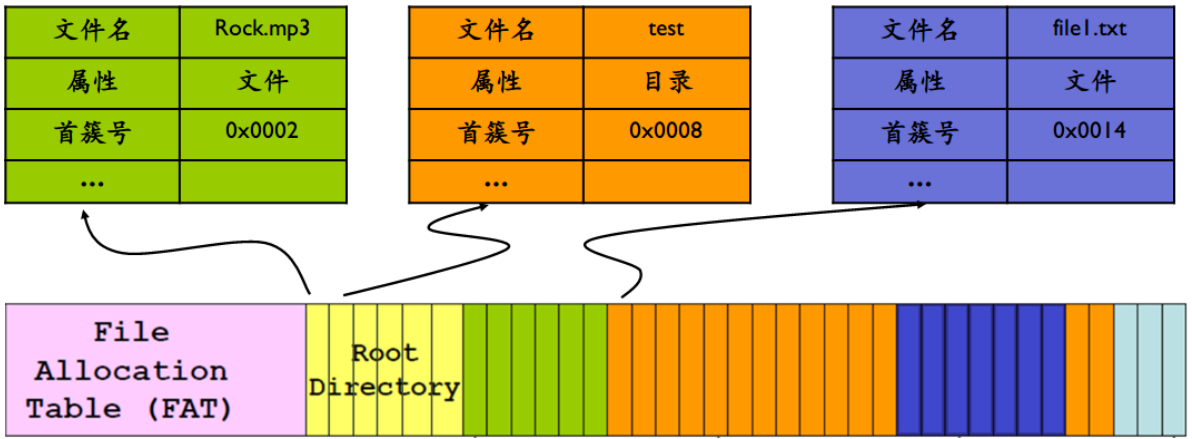


在FAT文件系统中，文件分为两个部分，第一个部分为目录项，记录该文件的文件名，拓展名，以及属性，首簇号等元数据信息，注：文件夹也是一种文件，它的目录项和普通文件结构相同；第二部分为实际存储内容，若为文件，则存储文件内容，若为文件夹，则存储子文件夹的文件目录项。

- 例一，文件处于Root目录下 (假设该文件路径为/rock.mp3)
若文件处于Root目录下，那么该文件的目录项将会存储在 Root Directory 区域中，其中记录了该文件的文件名，拓展名，以及第一个文件簇的地址。
FAT Table中存储了所有可用的簇，通过簇地址查找使用情况，我们能够得知该文件的下一个簇地址，若下一个簇地址为END标记符，则表示该簇为最后一个簇。通过查询FAT Table，我们能够得知该文件所有簇号。
- 情况二，文件处于Root目录下子文件夹中 (假设该文件路径为/test/file1.txt)，那么可知test目录项在 Root Directory 中，而rock.mp3的目录项将会存储在test文件的数据区域。

1.4.1 文件目录项是如何存储的

文件目录是文件的元数据信息，存储在 Root directory 以及数据区中，下图是两个文件/rock.mp3和 /test/file1.txt的文件目录项存储示例。



文件的目录项结构

名称	偏移(字节)	长度(字节)	说明
DIR_Name	0x00	11	文件名（前8个字节为文件名，后3个为拓展名
DIR_Attr	0x0B	1	文件属性，取值为0x10表示为目录，0x20表示为文件
DIR_NTRes	0x0C	1	保留
DIR_CrtTimeTenth	0x0D	1	保留(FAT32中用作创建时间，精确到10ms))
DIR_CrtTime	0x0E	2	保留(FAT32中用作创建时间，精确到2s)
DIR_CrtDate	0x10	2	保留(FAT32中用作创建日期)
DIR_LstAccDate	0x12	2	保留(FAT32中用作最近访问日期)
DIR_FstClusHI	0x14	2	保留(FAT32用作第一个簇的两个高字节)
DIR_WrtTime	0x16	2	文件最近修改时间
DIR_WrtDate	0x18	2	文件最近修改日期
DIR_FstClusLO	0x1A	2	文件首簇号(FAT32用作第一个簇的两个低字节)
DIR_FileSize	0x1C	4	文件大小

1.4.2 文件分配表(FAT表)详解

- FAT表由FAT表项构成的，我们把FAT表项简称为FAT项，本次实验中，FAT项为 2 个字节大小。每个FAT项的大小为12位，16位，32位，三种情况，对应的分别FAT12，FAT16，FAT32文件系统。
- 每个FAT项都有一个固定的编号，这个编号是从0开始。
- FAT表的前两个FAT项有专门的用途：0号FAT项通常用来存放分区所在的介质类型，例如硬盘的介质类型为“F8”，那么硬盘上分区FAT表第一个FAT项就是以“F8”开始，1号FAT项则用来存储文件系统的脏标志，表明文件系统被非法卸载或者磁盘表面存在错误。
- 分区的数据区每一个簇都会映射到FAT表中的唯一一个FAT项。因为0号FAT项与1号FAT项已经被系统占用，无法与数据区的簇形成映射，所以从2号FAT项开始跟数据区中的第一个簇映射，正因为如此，数据区中的第一个簇的编号为2，这也是没有0号簇与1号簇的原因，然后2号簇与3号FAT项映射，3号簇与4号FAT项映射

- 分区格式化后，用户文件以簇为单位存放在数据区中，一个文件至少占用一个簇。当一个文件占用多个簇时，这些簇的簇号不一定是连续的，但这些簇号在存储该文件时就确定了顺序，即每一个文件都有其特定的“簇号链”。在分区上的每一个可用的簇在FAT中有且只有一个映射FAT项，通过在对应该簇号的FAT项内填入“FAT项值”来表明数据区中的该簇是已占用，空闲或者是坏簇三种状态之一。

取值意义

簇取值	对应含义
0x0000	空闲簇
0x0001	保留簇
0x0002 - 0xFFEF	被占用的簇；指向下一个簇
0xFFF0 - 0xFFF6	保留值
0xFFF7	坏簇
0xFFF8 - 0xFFFF	文件最后一个簇

1.5 完成simple fat16中关于读文件的部分

- 代补全部分：**完成simple_fat16.c文件中TODO部分，主要包括以下功能
 - path_split 将输入路径按“/”分割成多个字符串，并按照FAT文件名格式转换字符串
 - pre_init_fat16 读取启动扇区，获得文件系统相关参数，初始化fat16_ins的其余成员变量
 - fat_entry_by_cluster 返回簇号为ClusterN对应的FAT表项
 - find_subdir 从子目录开始查找path对应的文件或目录
 - fat16_readdir 遍历给定目录下的所有文件名，返回给fuse
 - fat16_read 从给定文件中读取一定长度的数据，返回给fuse
- 代码下载**

```

1  mkdir -p lab4-code
2  wget http://staff.ustc.edu.cn/~ykli/os2021/lab/lab4-code.tar.gz
3  # 备选链接1: https://raw.githubusercontent.com/ZacharyLiu-
    CS/USTC_OS/master/term2021/Lab4-File-System/lab4-code.tar.gz
4  # 备选链接2:
    https://git.lug.ustc.edu.cn/gloomy/ustc_os/-/raw/master/term2021/Lab4-
    File-System/lab4-code.tar.gz
5  tar -zxvf lab4_code.tar.gz -C lab4-code

```

- 代码编译调试**
请在main函数中的指定位置编写调试相关代码(如对上面的补充好了的函数进行调用, 并观察其返回的结果是否符合你的预期). 可以使用

```

1  #进入源码目录
2  cd lab4-code
3  make clean
4  make
5  ./simple_fat16 --test

```

测试一是为了验证程序的FAT相关部分的代码正确性（以fat16_test.img作为磁盘镜像文件），一共有6个测试目标（#2和#6不需要你们完成）：


```
(base) zach@v-pc:~/oslab/lab4_code/fuse_test_dir$ ./simple_fat16 --test
-----
running test
-----
#1 running test_path_split
test case 1: OK
test case 2: OK
test case 3: OK
success in test_path_split

#2 running test_path_decode
test case 1: OK
test case 2: OK
test case 3: OK
success in test_path_decode

#3 running test_pre_init_fat16
success in test_pre_init_fat16

#4 running test_fat_entry_by_cluster
test case 1: OK
test case 2: OK
test case 3: OK
success in test_fat_entry_by_cluster

#5 running test_find_root
test case 1: OK
test case 2: OK
test case 3: OK
success in test_find_root

#6 running test_find_subdir
test case 1: OK
test case 2: OK
test case 3: OK
success in test_find_subdir
```

通过测试一后，运行如下的命令进行FUSE功能的测试（以fat16.img作为磁盘镜像文件）：

```
1  #测试二
2  ./simple_fat16 -d fat_dir
```

在另一个shell中进入该目录

1. 能够运行tree/ls命令查看文件目录结构。

```
(base) zach@v-pc:~/oslab/lab4_code/fuse_test_dir$ tree fat_dir/
fat_dir/
|-- dir1
|   |-- dir2
|   |   |-- dir3
|   |   |   |-- test.c
|-- log.c
|-- makefile

3 directories, 3 files
```


2. 能够运行cat/head/tail命令查看目录下文件内容。

```
(base) zach@v-pc:~/oslab/lab4_code/fuse_test_dir$ tail fat_dir/log.c
// Set logfile to line buffering
setvbuf(logfile, NULL, _IONBF, 0);
}

void log_msg(const char *format, ...)
{
    va_list ap;
    va_start(ap, format);
    vfprintf(logfile, format, ap);
}
```

任务二：实现FAT文件系统创建/删除文件操作

在实现了支持读操作的FAT 16文件系统基础上，实现创建文件和删除文件的功能。

2.1 创建文件

为保证大部分同学能够按时完成实验，我们只要求能够简单创建空文件，对文件夹的创建和文件的写入操作不做要求。也就是说，在你实现的文件系统中，能够支持touch命令。

首先，我们来考虑文件系统中创建文件需要完成哪些工作：

- 找到新建文件的父目录
- 在父目录中找到可用的entry
- 将新建文件的信息填入该entry

思路似乎很简单，但实现起来有很多细节问题，为了让大家顺利通过检查，有必要透露更多的细节。你需要思考这些细节问题：

1. 父目录有可能是根目录或者子目录，后续操作是否相同？不同的话分别如何处理？
2. 什么是可用的entry？如何在目录文件中查找可用的entry？
3. 一个entry记录了很多值，在装填entry时我们应该写入哪些内容？

最后一个问题实现起来是比较琐碎，难点主要在于不熟悉相关的系统时间结构体以及FAT 16文件系统中时间的记录方式，这里我给出一些介绍：

```
1 struct tm{
2     int tm_sec; //代表目前秒数，正常范围为0-59，但允许至61秒
3     int tm_min; //代表目前分数，范围0-59
4     int tm_hour; //从午夜算起的时数，范围为0-23
5     int tm_mday; //目前月份的日数，范围01-31
6     int tm_mon; //代表目前月份，从一月算起，范围从0-11
7     int tm_year; //从1900 年算起至今的年数
8     int tm_wday; //一星期的日数，从星期一算起，范围为0-6
9     int tm_yday; //从今年1月1日算起至今的天数，范围为0-365
10    int tm_isdst; //日光节约时间的旗标，也就是夏令时
11 };
12
13 // 用timer的值来填充tm结构。timer的值被分解为tm结构，并用本地时区表示。
14 struct tm *localtime(const time_t *timer)
15
16 偏移地址0x16: 长度2，表示时间=小时*2048+分钟*32+秒/2。也就是：0x16地址的0~4位是以2秒为
    单位的量值，0x16地址的5~7位和0x17地址的0~2位是分钟，3~7位是小时
17
18 偏移地址0x18: 长度为2，表示日期=(年份-1980)*512+月份*32+日。也就是：0x18地址的0~4位是
    日期数，0x18地址5~7位和0x19地址的0位是月份；0x19地址的1~7位为年号
```

为了不做重复劳动，时间上面我们只要求填写最近修改的时间和日期，创建时间、访问时间等由感兴趣的同学们自主添加。

细心的同学可能会发现一些问题：

1. FAT 16文件系统根目录大小是固定的，也就意味着最多只能有512个entry。如果我在根目录下不断新建文件，超过了512个entry该如何处理呢？
2. 在FAT 16文件系统中，子目录和正常文件一样存放在数据区，这意味着子目录空间是可变的。如果我在某个子目录下新建一个文件，但是空间不够，找不到可用的entry，该怎么办？

对于第1个问题，可以忽略，既然文件系统固定了根目录大小，我们只能避免使用的entry数量不超过512。

第2个问题是可以解决的，我们可以给该子目录分配新的簇，增加该目录文件的空间，自然也就有了可用的entry。但为了控制实验难度，我们并不做强制要求，有余力的同学可以去实现这项功能。

总而言之，上述这两个问题都可以忽略，只要在找到可用entry的时候填入信息即可，如果没找到那就不填，文件创建失败。当然，我们鼓励同学们去解决第2个问题。

如果你看了源码，还应该注意，我们创建新文件时，该文件的首簇号我们填入了0xffff。这显然是错误的，但为了简化实验，降低难度，创建新文件时簇的分配我们不做要求，大家掌握创建文件的流程即可。

2.2 删除文件

同样地，我们只要求实现删除文件的操作，对文件夹的删除不做要求。也就是说，在你实现的文件系统中，能够支持rm指令。

删除一个文件，我们并不需要改变文件具体的内容，其实就是做一些信息标记，让系统知道该文件已经被删除了以及该文件占有的空间已经被释放了。这里，给出删除文件的思路：

- 释放该文件使用的簇（考虑迭代或递归，修改FAT表项即可，）
- 在父目录里释放该文件的entry（此操作和创建文件类似）

代码流程上和创建文件很相似，不再给予更多的提示。

注意：有些同学可能在添加了创建和删除文件功能后，前面的只读文件系统代码上会出现一些bug（逻辑bug，不会报错的），因人而异，仔细想一想，主要看看在添加功能后，查找文件时判断语句会不会出错，没问题就可以来检查了！

举个例子吧，比如在查找文件时，实现只读文件系统的时候是这么判定找不到entry的：

```
1  /* If the directory entry is free, all the next directory entries are also
2   * free. So this file/directory could not be found
3   */
4   if (Root->DIR_Name[0] == 0x00){
5       return 1;
6   }
```

2.3 辅助函数

为了让你们专注于文件系统的实现，助教给出了一些简单的辅助函数可供使用：

```
1  char **org_path_split(char *pathInput)
2  char * get_prt_path(const char *path, const char **orgPaths, int pathDepth)
```

函数具体的功能可以查看你simple_fat16.c文件。

2.4 实验要求

你需要补全下面4个函数：

```
1 int fat16_mknod(const char *path, mode_t mode, dev_t devNum)
2 int fat16_unlink(const char *path)
3 int dir_entry_create(FAT16 *fat16_ins, int sectorNum, int offset, char *Name,
   BYTE attr, WORD firstClusterNum, DWORD fileSize)
4 int freecluster(FAT16 *fat16_ins, int clusterNum)
```

函数的主体框架已经给出，你需要实现每个函数中的TODO标记部分。如果觉得给出的框架不合适，也可以自己重新实现，保证功能的正确性即可。

参考资料

- [FAT文件系统实现](#)
- [文件分配表](#)
- [fat32文件系统示例](#)