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 / VirtualBox
- OS: Ubuntu 20.04 LTS
- Linux内核版本: 5.9.0+
- libfuse3

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月6日晚实验课，讲解第一部分及检查实验
- 6月13日晚实验课，讲解第二部分及检查实验
- 6月20日晚实验课，检查实验
- 6月27日晚实验课，检查实验
- 7月2日 18:00 前，提交实验报告及代码

实验报告及实验代码提交

- 提交代码文件 `simple_fat16.c`
- 提交你的实验报告（推荐为 PDF 格式）
- 上传至 **BB系统**

快速开始

实验内容

本次实验要求你补全 `simple_fat16.c` 文件，使用实现一个简单的 FAT16 文件系统。如图，具体地说，助教提供的代码，用一个格式化为 FAT16 文件系统的镜像文件模拟了磁盘的行为，你只能通过助教提供的 `sector_read` 、 `sector_write` 函数，以扇区为粒度访问该镜像文件。你需要通过读写扇区，访问和维护模拟磁盘中的文件系统结构，实现 FAT16 文件系统的基本功能，例如：读取文件系统中的目录（ `readdir` ）、读取文件 `read` 、创建和删除文件、目录（ `mknod/unlink/mkdir/rmdir` ）、写文件（ `write` ）等。

为了方便实验，我们采用了 FUSE3 作为我们的实验平台，所以你只需要完成 `simple_fat16.c` 中格式为 `fat16_*` 的各个函数即可。为了降低实验难度，助教已经编写了大部分文件系统代码，你只需要按注释补全剩余部分即可。根据难度，推荐以以下顺序阅读和补全代码：

- 理解 `sector_read` 和 `sector_write` 的用法。
- 参考本文档对 FAT16 的说明，阅读 `fat16_init` 的代码，理解 `BPB_BS` 和 `DIR_ENTRY` （定义在 `fat16.h` ）中重要字段的含义。
- 阅读并补全 `fat16_readdir` 函数
  - 在 `fat16_readdir` 中，用到了 `find_entry` 函数，后者又调用了 `find_entry_internal` 函数，请阅读这两个函数，并补全后者。
- 阅读并补全 `fat16_read` 函数
- 阅读并补全 `fat16_mknod` 函数
- ……

测试方法

在编译前，你需要安装以下软件包：

- `libfuse3-dev`
- `pkg-config`
- `python3`

在实验目录下，使用 `make` 编译程序。

然后，使用 `./simple_fat16 -d ./fat16/ --img=fat16.img` ，即可将镜像文件 `fat16.img` 挂载到 `./fat16` 目录下。你可以打开另一个终端，进入这个目录运行 `ls` 等命令进行简单测试。在运行 `./simple_fat16` 的终端中，会打印调试信息。

为了简化测试，助教编写了自动化脚本进行自动测试，运行 `./test/run_test.sh` 即可自动完成测试。

当实现的文件系统有Bug时，可能会导致终端卡死、无法终止进程等问题，这时候可以使用以下命令强制取消挂载：

```
fusermount3 -zu ./fat16  # 也可将./fat16换为你挂载的目录
```

友情提示

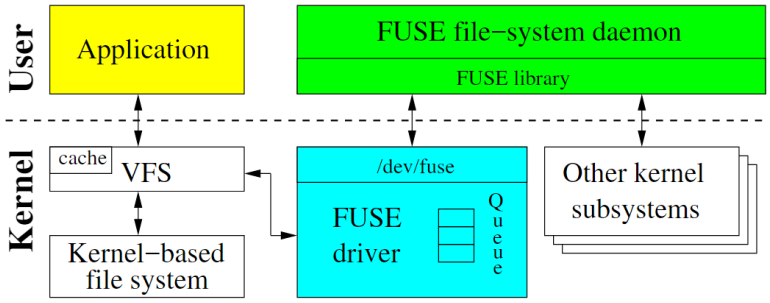
- **本次实验总工作量较大，请尽早开始实验。**
- **本次实验有300余行注释，注释量超过代码量的1/2，代码实现过程中务必注意看注释！看注释！看注释！**
- 实验多个任务难度梯度较大，靠前的任务代码提示较为详尽，且分值较高，你可以相对容易地获得这部分分数。
- 实验多个任务之间较为独立，你也可以选择只完成靠后的任务。
- **本实验文档可能不够详尽，如果你有什么疑问，可以在在线文档中提出，或联系助教询问，我们后续可能会更新实验文档，请注意群通知。**

评分标准

注：为平衡实验难度，减轻同学们的实验压力，本次实验任务一、二难度较低分值较高，任务三、四较为复杂但分值较低，同学们可根据自身时间情况选择性完成。实验任务三、四的具体评分标准将在下周发布。

1. 任务一：实现FAT文件系统读操作（**满分5分**）
    - 能够运行 `tree` ， `ls` 命令查看文件目录结构 （**3分**）
    - 能够正确读取小于一个簇的文件，通过短文件读取测试（**1分**）
    - 能够正确读取长文件，通过长文件读取测试（**1分**）
2. 任务二：实现FAT文件系统创建/删除文件、目录操作（**满分2分**）
    - 能够运行 `touch` 、 `mkdir` 命令创建新文件、目录，要保证文件属性的正确填写（**1分**）
    - 能够运行 `rm` 、 `rmdir` 、 `rm -r` 命令删除已有文件、目录，要保证簇的正确释放（**1分**）
3. 任务三：实现FAT文件系统写操作（**满分2分**）
    - 能够正确写入文件，通过文件写入、截断测试（**2分**）
4. 任务四：FAT 文件系统性能优化（**满分1分**）
    - 在模拟磁盘环境下，性能达到基准测试 50% （**1分**）
    - 在模拟磁盘环境下，性能达到基准测试的 200% （**额外+1分**）

背景知识：FUSE简介及环境配置



- 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 **这个文章**。

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。本实验使用fuse3，请使用以下命令安装libfuse3，和实验需要的其它库：

```
sudo apt install libfuse3-dev pkg-config python3 python3-pip
pip install pytest
```

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。
从上述描述中我们够得知，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

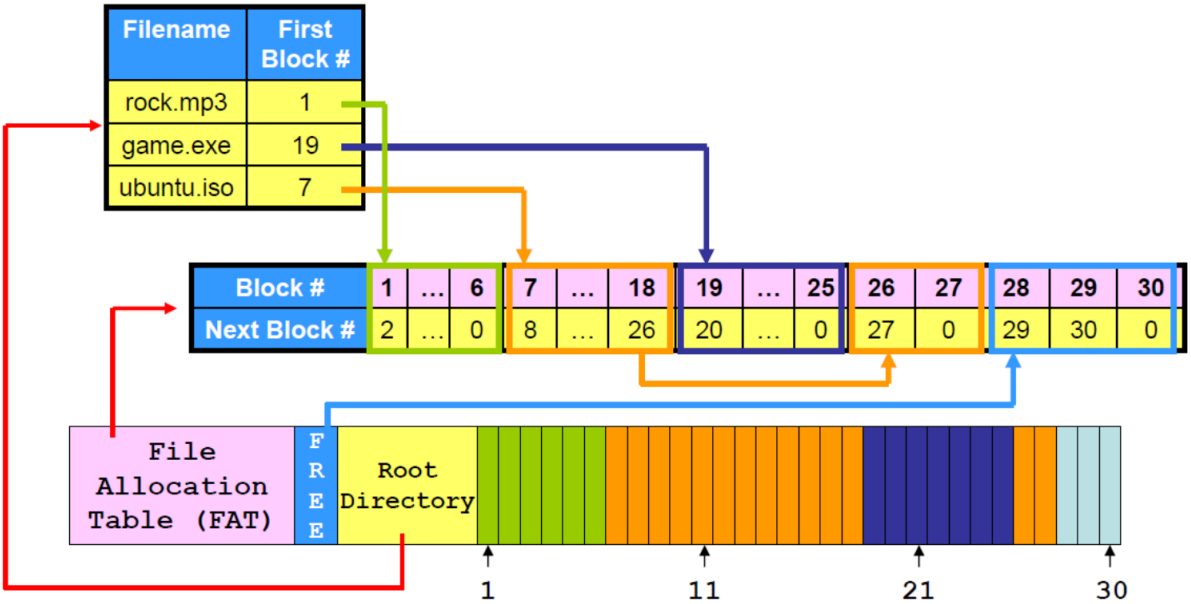
名称	偏移(字节)	长度(字节)	说明
BS_jmpBoot	0x00	3	跳转指令（跳过开头一段区域）
BS_OEMName	0x03	8	OEM名称，Windows操作系统没有针对这个字段做特殊的逻辑，理论上说这个字段只是一个标识字符串，但是有一些FAT驱动程序可能依赖这个字段的指定值。常见值是"MSWIN4.1"和"FrLdr1.0"
BPB_BytsPerSec	0x0b	2	**每个扇区的字节数。基本输入输出系统参数块从这里开始。**
BPB_SecPerClus	0x0d	1	**每簇扇区数**
BPB_RsvdSecCnt	0x0e	2	**保留扇区数（包括主引导区）**
BPB_NumFATS	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_VollID	0x27	4	卷序列号，例如0
BS_VollLab	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`BPB_RsvdSecCnt` * `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 + `BPB_NumFATS` * `BPB_FATSz16` * `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+ `BPB_RootEntCnt` *32

1.4 FAT文件系统磁盘分布详解

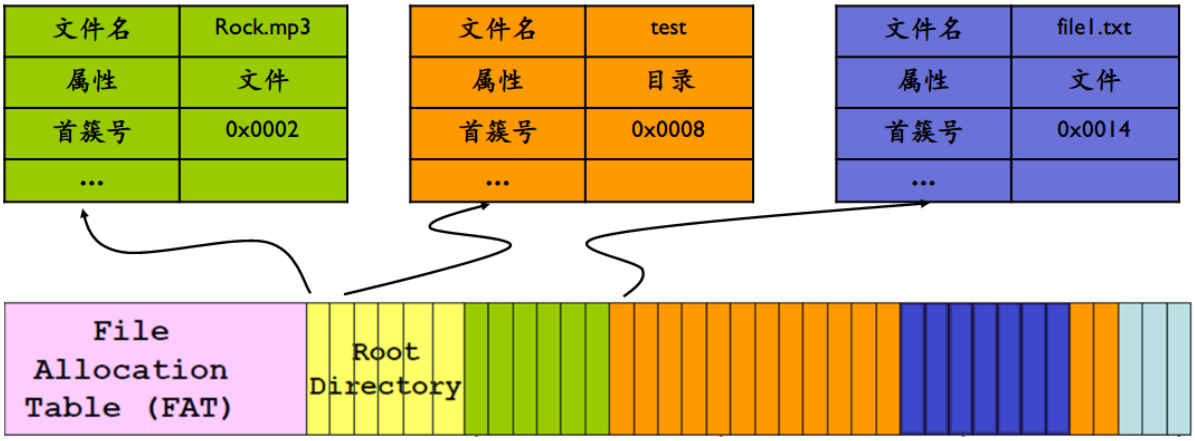


在FAT文件系统中，文件分为两个部分，第一个部分为目录项，记录该文件的文件名，拓展名，以及属性，首簇号等元数据信息，注：文件夹也是一种文件，它的目录项和普通文件结构相同；第二部分为实际存储内容，若为文件，则存储文件内容，若为文件夹，则存储子文件夹的文件目录项。

- 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项的大小有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肮脏标志，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，无法与数据区的族形成映射，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，这些族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中关于读目录/读文件的部分

- **代补全部分**：完成 `simple_fat16.c` 文件中关于读文件的TODO部分，主要包括以下功能
  1. `find_entry_internal`：依次读取路径每一级目录的内容，找到path所对应的目录项
  2. `find_entry_in_sectors`：读取一系列连续扇区内的目录项，并找到与给定文件名匹配的目录项
  3. `read_fat_entry`：读取指定簇号的 FAT 表项
  4. `fat16_readdir`：实现文件系统的读目录功能，即利用上述函数，找到目录对应的目录项，并读取并填写目录下的内容。
  5. `fat16_read`：实现文件系统的读文件功能，即利用上述函数，找到文件对应的目录项，并读取文件指定位置和长读的内容。

  函数补全的具体思路，请参考代码注释。
- **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符不符合你的预期). 可以使用

  ```
  #进入源码目录
  cd lab4-code
  make clean
  make
  ```

  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）：

  ```
  #测试二
  ./simple_fat16 -s -d fat16 --img=fat16.img
  ```

  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。

```
# ldeng @ DL-LAPTOP-G14 in ~/oslab4/fat16 [0:42:57]
$ ls
large.txt  small  tree
(base)
# ldeng @ DL-LAPTOP-G14 in ~/oslab4/fat16 [0:43:09]
$ tree small
small
├── s00.txt
├── s01.txt
├── s02.txt
├── s03.txt
├── s04.txt
├── s05.txt
├── s06.txt
├── s07.txt
├── s08.txt
├── s09.txt
├── s10.txt
├── s11.txt
├── s12.txt
├── s13.txt
├── s14.txt
├── s15.txt
├── s16.txt
├── s17.txt
├── s18.txt
└── s19.txt
```

2. 能够运行cat/head/tail命令查看目录下文件内容。

```
# ldeng @ DL-LAPTOP-G14 in ~/oslab4/fat16 [0:43:16]
$ head large.txt
0_______
1_______
2_______
3_______
4_______
5_______
6_______
7_______
8_______
9_______
```

自动化测试

为了方便实验，助教提供了用于自动化测试的 python 脚本，使用以下命令运行：

```
./test/run_test.sh
```

脚本中某些命令需要管理员权限，所以你可能需要输入密码运行。

该脚本会会运行13个测试样例，包含任务一和任务二的各个部分，当测试通过时，你会看到如下结果：

```
============================================= test session starts ==============================================
platform linux -- Python 3.10.10, pytest-7.3.1, pluggy-1.0.0 -- /home/ldeng/miniconda3/bin/python
cachedir: .pytest_cache
rootdir: /home/ldeng/oslab4/test
collected 13 items

fat16_test.py::TestFat16List::test1_list_tree PASSED                                                  [  7%]
fat16_test.py::TestFat16Read::test1_file_exists PASSED                                                [ 15%]
fat16_test.py::TestFat16Read::test2_seq_read_large_file PASSED                                        [ 23%]
fat16_test.py::TestFat16Read::test3_rand_read_large_file PASSED                                       [ 30%]
fat16_test.py::TestFat16Read::test4_seq_read_small_files PASSED                                       [ 38%]
fat16_test.py::TestFat16CreateRemove::test1_file_create_root PASSED                                   [ 46%]
fat16_test.py::TestFat16CreateRemove::test2_file_remove_root PASSED                                   [ 53%]
fat16_test.py::TestFat16CreateRemove::test3_dir_create_root PASSED                                    [ 61%]
fat16_test.py::TestFat16CreateRemove::test4_dir_remove_root PASSED                                    [ 69%]
fat16_test.py::TestFat16CreateRemove::test5_file_create_subdir PASSED                                 [ 76%]
fat16_test.py::TestFat16CreateRemove::test6_file_remove_subdir PASSED                                 [ 84%]
fat16_test.py::TestFat16CreateRemove::test7_create_tree PASSED                                        [ 92%]
fat16_test.py::TestFat16CreateRemove::test8_remove_tree PASSED                                        [100%]

============================================== 13 passed in 0.75s ==============================================
```

当测试失败时，你会看到类似下图的提示，提示中包含了你的文件系统在哪个样例中出现错误（图中为`test2_seq_read_large_file`，说明你的程序在读取大文件时出现错误），并提示了你错误的内容（如果你熟悉python，你可以查看`./test/fat16_test.py`来查看具体的测试流程，否则，你需要手动调试出错的功能，如果遇到复杂的问题，请及时联系助教。）

```
fat16_test.py::TestFat16List::test1_list_tree PASSED                                                  [  7%]
fat16_test.py::TestFat16Read::test1_file_exists PASSED                                                [ 15%]
fat16_test.py::TestFat16Read::test2_seq_read_large_file FAILED                                        [ 23%]

=================================================== FAILURES ===================================================
_____________________________________ TestFat16Read.test2_seq_read_large_file _____________________________________

self = <fat16_test.TestFat16Read testMethod=test2_seq_read_large_file>

    def test2_seq_read_large_file(self):
        os.chdir(FAT_DIR)
        self.assertTrue(os.path.getsize(LARGE_FILE) == LARGE_FILE_SIZE,
                        f'{LARGE_FILE} is not {LARGE_FILE_SIZE} bytes')
        with open(LARGE_FILE, 'r') as f:
>           content = f.read()
E           IsADirectoryError: [Errno 21] Is a directory

/home/ldeng/oslab4/test/fat16_test.py:46: IsADirectoryError
============================================= short test summary info ==============================================
FAILED fat16_test.py::TestFat16Read::test2_seq_read_large_file - IsADirectoryError: [Errno 21] Is a directory
!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!! stopping after 1 failures !!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!!
============================================ 1 failed, 2 passed in 0.06s ============================================
```

任务二：实现FAT系统文件/文件夹创建/删除操作

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？
4. 我们只能以扇区为粒度修改镜像，但一个entry小于一个扇区，我们如何只修改一个entry？

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（但没有加分）。**

如果你看了源码，还应该注意到，我们创建新文件时，该文件的首簇号我们填入了0。这显然是错误的，但为了简化实验，降低难度，大家掌握创建文件的流程即可。（你也可以自行修改创建时的首簇号，也许0xFFFF也是个不错的选择。）

2.2 创建文件夹（目录）

创建文件夹和创建文件的要求很像，但文件夹创建的时候，就包含了 `.` 和 `..` 两个目录，因此我们不能像创建文件时那样，不给新目录分配簇。所以，在创建文件夹时，我们也要从FAT16文件系统中找到空簇，并分配给新创建的文件夹，并把 `.` 和 `..` 两个特殊的目录写在新文件夹的头两个目录项内。

2.2.1 空闲簇的分配

在完成本部分实验前，我们建议实现 FAT 文件系统的空闲簇分配，这不仅是创建文件夹的必须条件，也会极大简化在任务三和任务四的代码复杂度。空闲簇分配的使用场景如下：

- 在 FAT 中创建文件夹的时候，需要申请一个空闲簇，作为新建文件夹的起始簇，存储 `.` 和 `..` 目录项
- 在 FAT 中写文件的时候，如果写入量过大，需要申请额外的空间以防止写数据丢失问题

为实现空闲簇的分配，，我们提前定义了如下函数：

```
WORD alloc_clusters(FAT16 *fat16_ins, uint32_t n);
```

该函数作用是，在文件系统中分配n个未被使用的簇，将它们连接起来，返回首个分配的簇号。如图，如果我们需要分配四个簇，并且在FAT表中找到了 `0x06`，`0x07`，`0x18`，`0x27` 等四个空簇。我们需要将这四个空簇的表项连接在一起，并返回 `0x0006`：

分配前：

Cluster #	…	0x02	…	0x06	0x07	…	0x18	…	0x27	…
Next Cluster #	…	0x03	…	0x00	0x00	…	0x00	…	0x00	…

分配后：

Cluster #	…	0x02	…	0x06	0x07	…	0x18	…	0x27	…
Next Cluster #	…	0x03	…	0x07	0x18	…	0x19	…	0xFFFF	…

该函数实现思路如下：

1. 扫描FAT表，找到n个空闲的簇（空闲簇的FAT表项为 `0x0000`）

2. 若找不到n个空闲簇，直接返回 `CLUSTER_END` 作为错误提示。注意，找不到n个簇时，不应修改任何FAT表项。
3. 依次清零n个簇，这需要将0写入每个簇的所有扇区
4. 依次修改n个簇的FAT表项，将每个簇通过FAT表项指向下一个簇，第n个簇的FAT表项应该指向 `CLUSTER_END`
5. 返回第1个簇的簇号

如果你不喜欢该函数的定义，你也可以选择不实现该函数，自行定义函数或用别的方式完成空闲簇分配的任务。

2.2.2 修改 FAT 表项

在实现 `alloc_clusters` 的过程中，需要修改多个簇的FAT表项，为了简化代码，我们建议你补全 `write_fat_entry` 函数，用于修改单个簇对应的FAT表项。同样，你可以选择不实现这个函数，用自己的方式完成实验。

```
int write_fat_entry(FAT16 *fat16_ins, cluster_t clus, cluster_t data);
```

该函数作用是，将 `clusterN` 对应的FAT表项修改为 `data` 。这个函数 `read_fat_entry` 函数逻辑上比较相似，后者作用为读出 `clus` 对应的表项，所以你可以参考 `read_fat_entry` 函数来实现该函数。该函数的实现思路如下：

1. 计算出 `clus` 这个簇对应的FAT表项所在的扇区号和偏移量。
2. 读取所在扇区，将前述偏移量位置的FAT表项修改为 `data` ，然后写回该扇区。

2.3 删除文件/文件夹

在你实现的文件系统中，需要能够支持**rm**和**rmdir**指令，事实上，如果支持了这两个指令，你的文件系统同时也就支持了**rm -r**，后者实际上就是递归的调用前面两个指令。

2.3.1 删除文件

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！另外，我们的测试功能进行了很复杂的创建和删除操作，如果能通过pytest测试，那么功能上应该没有太大问题。

2.3.2 删除文件夹（目录）

该部分的主要任务是实现以下函数：

```
int fat16_rmdir(const char *path);
```

`fat16_rmdir` 函数需要实现删除 `path` 对应的目录的功能，注意，传入的 `path` 对应的目录必须存在，且为空目录，否则应该直接返回错误。另外，文件系统的根目录 `/` 无法被删除。

在终端中运行 `rmdir` 指令时，fuse会将对应的文件系统操作转换为对 `fat16_rmdir` 的调用。而当在终端中调用 `rm -r` 时， `rm` 命令会依次递归的删除目录下的文件和子文件夹（转换为fuse中的 `unlink` 和 `rmdir` 操作），将目录清空后，在删除目录本身。这就是为什么 `fat16_rmdir` 要求 `path` 对应的目录为空，而使用 `rm -r` 时，却不需要保证文件夹为空。

`fat16_rmdir` 的实现思路如下：

1. 找到目录，确认目录为空。
2. 释放目录占用的所有簇。
3. 删除目录在父目录中的目录项。

实现思路中的第一步和 `readdir` 有一些相似，而第2、3步则类似 `unlink` 。你可以参考之前这两个函数来实现文件夹删除。

2.4 实验要求

你需要补全下面几个函数：

```
int dir_entry_write(DirEntrySlot slot);
int write_fat_entry(cluster_t clus, cluster_t data);
int alloc_clusters(size_t n, cluster_t* first_clus);

int fat16_mknod(const char *path, mode_t mode, dev_t dev); // 已实现，供参考
int fat16_unlink(const char *path); // 已实现，供参考
int fat16_mkdir(const char *path, mode_t mode); // 参考 fat16_mknod
int fat16_rmdir(const char *path);   // 参考 fat16_unlink
```

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

1. **FAT文件系统实现**
2. **文件分配表**
3. **fat32文件系统示例**
4. **Linux kernel 中的FAT实现**
5. **Linux kernel VFAT文档**
6. **维基百科：8.3文件名**
7. **Ubuntu Manpage: mount(8)**
8. **维基百科：FAT文件系统的设计**
9. **微软文档：文件名**
10. **POSIX locale (C locale) 定义**