实验四 FAT文件系统的实现 Part2

> 提示：本文档涉及一些对Part1问题的更正。所以只要你计划完成Part1，就建议你阅读本文档的部分内容。

实验目标

- 熟悉文件系统的基本功能与工作原理（理论基础）
- 熟悉FAT16的存储结构，利用FUSE实现一个FAT文件系统：
  - 文件目录与文件的读（只读文件系统，基础）
  - 文件的创建与删除（基础）
  - 文件目录的创建与删除（进阶）
  - 文件的写（进阶）
- 在实现fat16文件系统的基础上，引入文件系统的重要问题研究（进阶，选做）
  - 如何做文件系统的错误修复——三备份与纠删码
  - 如何做文件系统的日志恢复——元数据日志

实验环境

- VMware / VirtualBox
- OS: Ubuntu 20.04.4 LTS

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

- 5月13日晚实验课，讲解Part1及检查实验
- 5月20日晚实验课，讲解Part2检查实验
- 5月27日晚实验课，检查实验
- 6月3日晚实验课，实验补检查

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600余行注释，以解决实验文档对于代码实现的提示量不足的问题，代码实现过程中务必注意看注释！看注释！看注释！**
- 实验多个任务难度依次递增，靠前的任务代码提示较为详尽，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。
- **本实验文档较为详尽，是为了尽可能提前解决大家实验中可能遇到的问题。你并不一定需要完全阅读所有实验文档才能完成实验、获得分数。**

第一部分：对Part1的补充说明

1.0 关于代码版本的说明

我们在Part2中发布了完整的实验代码和文档包 `lab4-all.zip`，由于其中的 `simple_fat16_part1.c` 相比于Part1发布的版本稍有修改，我们建议你根据目前的实验情况，选择以下一种方式处理你的代码：

- 如果你**还未开始进行实验**：请进行忽略之前发布的 `lab4-part1.zip`，**使用 `lab4-all.zip` 中的文件完成实验即可。**
- 如果你**已经开始了Part1的实验**：请使用 `lab4-all.zip` 中的 `simple_fat16_part2.c` 替换Part1发布的同名文件，**并按照1.1节的方法，修正 `simple_fat16_part1.c` 中的计算错误。**

1.1 修正RootOffset的计算错误

若你还未开始进行实验，你可以忽略这一节的内容，使用5月20日发布的 `lab4-all.zip` 中的代码完成实验即可。

在5月13日发布的 `simple_fat16_part1.c` 的第293行，函数 `pre_init_fat16` 的倒数第5行，对 `fat16_ins->RootOffset`（即根目录区域在镜像中的偏移量）的计算出现了错误，误将第一个运算符由 `+` 误打为 `*`。
这会导致使用以下字段和函数时可能出现错误：

- `fat16_ins->RootOffset`
- `fat16_ins->DataOffset`
- `find_root` 函数的 `offset_dir` 输出参数

我们在Part2的代码中，添加了 `get_fat16_ins_fix` 函数修正该错误，所以在Part2中使用这些字段和函数不会出现问题。但由于这些字段和函数，特别是 `find_root` 函数在实验中大量使用，为防止该错误导致程序出现难以检查的问题，我们强烈建议你**按如下方式手动修正 `simple_fat16_part1.c` 的代码：**

- 在 `simple_fat16_part1.c` 中，查找 `fat16_ins->RootOffset =`，定位到错误代码（在没有添加代码的文件中，该错误行在293行，函数 `pre_init_fat16` 的倒数第5行），如下图红色行所示。
- 将该行代码中第一个 `*` 改为 `+`，修改后的代码如下图绿色行所示。
- 你也可以自行阅读改行代码，并对比Part1文档的2.3节，理解该行代码，并检查代码的正确性。

```
291   292       fat16_ins->FatOffset =  fat16_ins->Bpb.BPB_RsvdSecCnt * fat16_ins->Bpb.BPB_BytsPerSec;
292   293       fat16_ins->FatSize = fat16_ins->Bpb.BPB_BytsPerSec * fat16_ins->Bpb.BPB_FATSz16;
293         -   fat16_ins->RootOffset = fat16_ins->FatOffset * fat16_ins->FatSize * fat16_ins->Bpb.BPB_NumFATS;
      294   +   fat16_ins->RootOffset = fat16_ins->FatOffset + fat16_ins->FatSize * fat16_ins->Bpb.BPB_NumFATS;
294   295       fat16_ins->ClusterSize = fat16_ins->Bpb.BPB_BytsPerSec * fat16_ins->Bpb.BPB_SecPerClus;
295   296       fat16_ins->DataOffset = fat16_ins->RootOffset + fat16_ins->Bpb.BPB_RootEntCnt * BYTES_PER_DIR;
```

1.2 强制FUSE使用单线程模式运行

在Part1的3.0.4节，我们给出如下命令用于运行我们的程序，将我们的文件系统挂载到 `fat_dir` 目录下：

```
./simple_fat16 -d fat_dir
```

该命令会导致FUSE使用默认的多线程模式运行，在一般情况下，这不会导致什么问题，但由于我们提供的代码并不保证线程安全，当发生对文件系统并发访问时，有可能因为数据争用而导致错误。例如，某些pdf阅读器为了加速渲染等过程，或由多个线程同时对读取文件，这可能会导致我们的文件系统出现偶然的读取错误，使得pdf阅读器显示不完全、出现乱码或者因发生错误而崩溃。这些错误因为数据争用的不确定性而难以复现。

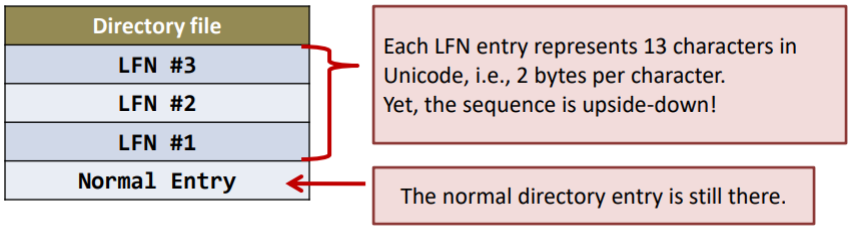
为了避免该问题，**我们推荐使用单线程模式运行FUSE，即使用下列命令运行我们的程序**：

```
./simple_fat16 -s -d fat_dir
```

`-s` 参数代表 `single-thread`，会强制FUSE使用单线程模式运行，一次只能有一个线程访问挂载的文件系统，这避免了数据争用带来的Bug，也简化了我们的调试。

1.3 忽略长文件名项（LFN entries）

在实现任务一的 `readdir` 函数时，一些程序可能会读取到文件名类似 `A1`，`Am`，`B0` 的目录项。如果检查该目录项的 `DIR_Attr` 字段，会发现这些目录项的属性均为 `0x0F`。这些项是FAT文件系统的长文件名项（LFN entries），用于存储文件的长文件名。长文件名可以占据一个或多个目录项的位置，在文件的普通的目录项之前，如图所示：



我们提供的镜像中，所有文件和目录都有对应的长文件名项（包括文件名没有超过8+3字节的文件）。本次实验，我们不要求对LFN进行处理，但是程序要能正确忽略这些LFN项，即，在实现 `readdir`，`mknod`，`rmdir` 等功能时，要主动忽略所有 `Dir_Attr` 为 `0x0F` 的目录项。即：

- 在实现 `readdir` 时，你应该跳过所有LFN项，不将它们读出。
- 在实现 `mknod`、`mkdir` 时，你能正确识别出LFN项是非空的目录项，不将它们用于创建新文件、文件夹。（一般来说，这条不需要特别实现，因为LFN项的首个字节不会为 `0x00` 或 `0xe5` 等代表未使用或已删除条目字符）。
- 在实现 `rmdir`，`unlink` 时，你不需要将文件或目录对应的LFN项删除，忽略它们即可。
- 由于上一条，在实现 `rmdir` 中，判断目录是否为空时，你需要跳过所有LFN项，也就是说，即使被删除的目录的目录项中下仍有LFN项，只有没有正常的目录项存在，你就可以认为该目录为空。

有兴趣的同学也可以思考如何正确读出文件的长文件名和创建LFN。如我们提供的镜像文件中pdf文件，全名是 `hamlet_PDF_FolgerShakespeare.pdf`，这个名称（包括大小写），完整的保存在了LFN中。

1.4 其它补充说明

1.4.1 安装 `tree` 指令

在测试读目录时，若出现找不到 `tree` 命令的情况，请使用包管理器手动安装 `tree` 命令。（不知道什么是包管理器？请回顾Lab1的实验文档。）

1.4.2 POSIX错误代码

本次实验中要求实现的大部分文件系统接口，出现错误时都要求返回POSIX错误代码的负值。你其实可以在错误时返回任意负数，但是这会导致终端显示的错误信息出现问题。如，假设你在出现所有错误时都返回 `-1`，那终端给出的错误提示将永远是 `Operation not permitted` 或 `权限不够`。所以我们推荐你在程序中返回正确的POSIX错误代码。

POSIX错误代码已经以宏定义的形式设置在 `error.h` 头文件中，fuse已经包含了这个头文件，你可以直接使用。错误的宏名和对应的意义可以在**这个链接**中找到，你也可以运行 `man errno` 或直接到 `error.h` 中查看这些宏定义的名字。这里给出一些可能用到的错误：

- `ENOENT` 文件或目录不存在
- `ENOSPC` 空间不足
- `EISDIR` 路径是目录
- `ENOTDIR` 路径不是目录
- `ENOTEMPTY` 目录非空
- `EINVAL` 参数非法
- `EIO` 读写错误
- `EBUSY` 设备或路径被占用（用于 `rmdir` 删除根目录时）
- ……

我们【**不要求，但推荐**】你准确使用这些错误代码，你可以按照自己理解随意使用。

例如，当你的程序执行 `readdir` 时，发现参数的 `path` 不存在，可以返回 `-ENOENT`；发现 `path` 是一个文件（自然没法“读目录”），可以返回 `EISDIR`。

第二部分：实验任务

2.0 前置任务：FAT中空闲簇的分配

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

你也可以不实现本节中的函数，使用自己的逻辑完成任务三和任务四。

2.0.1 空闲簇分配

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。

2.0.2 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，我们建议实现 `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, WORD clusterN, WORD data);
```

该函数作用是，将 `clusterN` 对应的FAT表项修改为 `data` 。这个函数和Part1中给出的 `fat_entry_by_cluster` 函数相似，后者作用为读出 `clusterN` 对应的表项，所以你可以参考 `fat_entry_by_cluster` 函数来实现该函数。该函数的实现思路如下：

1. 计算出 `ClusterN` 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。

2.1 任务三：实现FAT文件系统创建/删除文件夹操作

2.1.1 创建文件夹

该部分的主要任务是实现以下函数：

```
int fat16_mkdir(const char *path, mode_t mode);
```

`fat16_mkdir` 函数的功能是，创建 `path` 指定的新目录，注意传入的 `path` 必须保证父目录存在，如父目录不存在，直接返回 `-ENOENT` （文件不存在即可）。 `mode` 参数指定了创建文件夹时的一些选项，**在本次实验中，我们可以忽略 `mode` 参数。**

当在fuse挂载的目录中调用 `mkdir` 等命令时，fuse会将设计的文件系统操作转换为对函数 `fat16_mkdir` 的调用。所以我们只需实现 `fat16_mkdir` 函数，就能支持文件夹的创建功能。
`fat16_mkdir` 的实现思路：

1. 找到新建文件的父目录；
2. 在父目录中找到可用的entry；
3. 将新建文件夹的信息填入该entry：

   这里的实现思路看起来与 `fat16_mknod` 完全一致（事实上，找到父目录以及在父目录中找到可用entry两部分的思路确实一致）。但是，我们在填入entry一步还需要其他操作：
4. 在FAT表中申请一个空闲簇，将其作为新建文件夹的初始簇，并在该簇中插入 `.` 和 `..` 两个目录项：

   这一操作是有其必要性的。第一，在课上你已经学过，目录文件内前两条entry分别是该文件本身和父目录文件的符号链接，它们在目录文件创建时就已经存在；第二，我们本次实验中不要求在目录的目录项空间不足时自动拓展新簇，如果不在创建时就分配簇的话，会导致无法在该文件夹中创建文件和子文件夹，这是不可容忍的bug。如果你实现了前置任务中的 `alloc_clusters` 函数，你可以在此处使用它，简单地为你分配一个空簇，你只需要正确调用 `dir_entry_create` 就可以将这个簇连接到新目录下。

   但是，由于本次实验不涉及链接操作，所以你不需要把 `.` 和 `..` 链接到对应目录，FUSE会在涉及这些目录时自动识别。填写它们的entry时，按照实际涉及的两个目录信息填写参数。（目录的文件大小设置为0即可。）

2.1.2 删除文件夹

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Part1中的这两个函数来实现文件夹删除。

2.2 任务四：实现FAT文件系统写操作

为了支持 FAT16 文件的写操作，需要实现如下两个函数：

```
int fat16_write(const char *path, const char *data, size_t size, off_t offset, struct
fuse_file_info *fi);
int fat16_truncate(const char *path, off_t size);
```

函数解释说明：

- `fat16_write` ：将 `data[0..size-1]` 的数据写入到 `path` 对应的文件的偏移为 `offset` 的位置，如果写入的数据超出了文件末尾，则应相应增大文件大小。 `fi` 参数在本次实验中可忽略。
- `fat16_truncate` ：将 `path` 对应的文件大小设置为 `size` 。如果 `size` 比原文件的大小要小，则从末尾处截断文件；如果 `size` 大于原文件的大小，那么将新增加部分的数据初始化为 `0` 。

当我们在终端中使用类似 `echo hello >> file` 的命令，通过 `>>` 重定向符向文件末尾追加写时，只会调用 `fat16_write` 函数。若我们使用 `>` 重定向符，如运行 `echo hello > file` 时，则会先调用 `fat16_truncate` 将文件大小设置为0，再调用 `write` 向文件中写入。所以为了支持文件写，两个函数都需要实现。我们也可以使用 `truncate` 命令来直接测试 `fat16_truncate` 的功能。

2.2.1 文件写入

首先看一下 `fat16_write` 的实现思路：

1. 通过 `path` 获取到对应的目录项，即 `DIR_ENTRY` 结构体，通过调用 `find_root` 即可。
2. 比较新写入的数据和文件原来的数据所需的簇数量，如果新写入的数据所需的簇数量大于原来所需的簇数量，则需要为文件扩容簇数量。即在FAT表中查找未分配的簇，并链接在文件末尾。如果你完成了前置任务中的 `alloc_clusters` ，你可以在此处使用它，为你分配你所需要的簇数量，你只需要正确地将分配好的簇链连接在你文件的末尾即可。（如果文件原来没有任何簇，你要把簇链连接在目录项的 `DIR.FstClusLo` ，否则，连接在原文件簇链的末尾。）
3. 按需扩容后，只需要实现写文件操作即可，通过 `DIR.FstClusLo` 获取第一个簇号，之后可以通过链接依次遍历，找到要写入簇的位置，并在读取相应的扇区，修改扇区的正确位置，并写回扇区。（这个过程和 `read` 的实现很类似。）
4. 实现目录项的更新，通过 `find_root` 可以获取到目录项对应的偏移量，更新对应的目录项数据写入即可；
5. 返回成功写入的数据字节数。

`fat16_write` 函数的流程有些复杂，为了简化代码逻辑，在提供的代码中，我们将这一过程拆分成了以下多个函数：

```
// 要实现的函数本身，完成流程的第一步，并正确调用write_file
int fat16_write(const char *path, const char *data, size_t size, off_t offset,
                struct fuse_file_info *fi);

// 写入文件的主要功能在此实现
// 先调用file_new_cluster完成第二步
// 然后循环调用write_to_cluster_at_offset实现第三步
// 最后更新目录项并返回正确的值
int write_file(FAT16 *fat16_ins, DIR_ENTRY *Dir, off_t offset_dir, const void *buff,
               off_t offset, size_t length);

// 完成第三步，调用alloc_clusters，并把新分配的簇链接在文件末尾
```

```
int file_new_cluster(FAT16 *fat16_ins, DIR_ENTRY *Dir, WORD last_cluster, DWORD count);

// 将数据写入指定簇的对应偏移量，用于简化第三步的实现。
// 所有写入扇区的细节被封装在这个函数中，write_file函数只需要考虑簇一级即可。
size_t write_to_cluster_at_offset(FAT16 *fat16_ins, WORD clusterN, off_t offset,
                                  const BYTE* data, size_t size);

// 辅助函数，找到文件最后一个簇，并计算文件当前簇的数目。
// 在计算文件是否需要分配新簇，和将新簇连接至文件末尾时很有用。
WORD file_last_cluster(FAT16 *fat16_ins, DIR_ENTRY *Dir, int64_t *count);
```

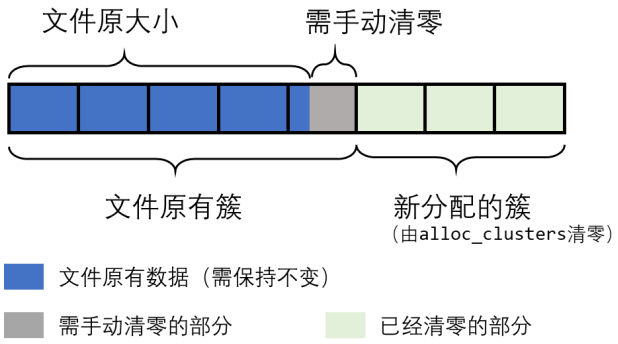
你可以在我们提供的代码中找到这些函数的详细功能即参数说明。你可以通过正确补充这些函数来实现文件写入的功能。当然，和前面的任务一样，你也可以不实现这些函数，而通过自己喜欢的逻辑实现 `fat16_write`。

2.2.2 文件截断/拓展

`truncate` 意为“截断”，但实际上这个函数也能实现文件拓展，所有下文我们也用“截断”一词统称改变文件大小的操作。

接下来是 `fat16_truncate` 的实现思路：

1. 通过 path 获取到对应的目录项，即 `DIR_ENTRY` 结构体，通过调用 `find_root` 即可
2. 计算并比较原文件拥有的簇数量 `n1`，和文件截断后需要的新簇数量 `n2`。通过比较 `n1` 和 `n2` 的大小，来判断文件是否需要新增或释放簇：
    1. `n1 == n2`：此时不需要任何操作
    2. `n1 < n2`：此时需要扩容文件大小，可以参照 `fat_write` 实现思路。
    3. `n1 > n2`：此时需要截断文件，找到文件的第 `n2` 个簇，更改FAT表项，将此处改为文件末尾，并释放后续所有簇。（释放可使用Part1中的 `free_clusters` 函数。）
3. 按需写入，比较新文件的大小 `new_size`（就是参数中的 `size`）和原来的文件大小 `old_size`：
    1. `new_size > old_size`：需要从 `old_size` 位置开始，将后续文件数据清空为 `0x00`。如果你正确实现了 `alloc_clusters`，那么新分配的簇应该已经清空，你只需要将文件原末尾，至原文件的最后一个簇的结尾的一小段手动清零即可。（如图所示，你也可用自己的方式清零。）



    2. `new_size <= old_size`：不需要进行任何操作
3. 实现目录项的更新，通过 `find_root` 可以获取到目录项对应的偏移量，更新对应的目录项数据写入即可
4. 返回 0 表示正常结束，否则表示异常

2.3 选做：实现FAT文件系统的容错与日志机制

2.3.1 实现文件系统的三备份容错机制（1'）

存储介质在使用过程中有概率会出现不可预测的错误事件，比如单比特反转，在特定场景下会产生严重的影响（某个参数的量级改变）。因此现代计算机系统中会考虑各种技术中的出错现象并设计一定的方案来保证出错后系统的正常运行，如我们课上学到的RAID组织方式下，通常会有冗余或者纠删码来提供出错检测及出错后的修复能力。本次实验中我们可以尝试实现一种最易于理解的纠错设计——三备份冗余，即一个磁盘上的内容，总共保存三份副本，这样当任意一个副本中出现错误时，根据少数服从多数的原则，将两个没有错误的副本中的内容作为正确内容，对于出错副本做修复。一般三份副本会存在三块独立的存储设备上，因此在同一处同时发生两次或三次错误的可能性较低，在一定程度上保证了磁盘的容错能力。

本部分的实现思路为：

- 将原有的img文件复制额外复制两份，在初始化时读入三份img文件
- 维护一个三副本的扇区写接口，每次写入时同时对三份img文件写（视为一个原子操作，但是这里暂时不要求多线程，单线程下能正常运行即可）
- 维护一个三副本的扇区读接口，每次读入三份扇区并做比较，如果有不同则进行修复，最后返回三份一致的扇区内容（同样视为一个原子操作）

本部分的检查要求：人工干扰单个磁盘的内容后（如某个位反转），文件系统可以正确修复该错误。为了验证此功能，你需要先找到一种干扰方式，使得三份img中只有两份可以读出正确内容，而被干扰的img读出的内容存在问题（或者可能直接无法工作），然后在支持三备份的文件系统中挂载三份img后，可以正确读出内容，并修复了存在问题的那个img，保证之后单独读时三份都正确。

2.3.2 实现文件系统的日志机制 (1')

磁盘作为非易失存储，具有掉电后依然保持有数据的特征。但是若设备在持久化存储过程中掉电，则会出现错误，如涉及到多个扇区写的原子操作，在写完一部分扇区后设备断电，那么重新通电时，已写完的部分就会成为错误的信息，且整个原子操作并未实际完成。为了确保掉电后可以正确恢复磁盘内的数据组织，现代文件系统如ext4通常会采用日志来记录每次磁盘操作，并在掉电恢复时先读磁盘操作，如果有尚未完成的操作则尝试恢复或是直接丢弃，从而保证磁盘数据的正确性。

考虑到代码实现的难度，我们对于日志的实现要求做了简化，完成本部分选做只需要额外生成一个log文件，log文件中存储有每次原子操作的记录，并分为begin-end两步，在原子操作开始时记录原子操作对应的参数，并在结束时再加入一条结束记录。

定义fuse的一个operation操作为一个原子操作，我们需要在这部分保证各种操作的原子性，保证多线程下fuse文件系统依然可以保证正确的运行顺序（不会在一个写操作执行的同时执行另一个写操作），之后围绕每个原子操作，做log的记录。在保证原子性的前提下，log的记录中不会出现一个原子操作过程中发生其他原子操作的情况。

本部分的检查要求：需要在多线程下正确打开pdf文件，并且产生的log中没有违反原子性的记录。log中的格式可以自行设计，需要至少有操作名、操作传入的参数，操作的begin-end记录。传入参数含写入数据的可以只记录前30个字符。

第三部分：知识问答

我们将从下列题目随机挑选 3 道，答对 2 题即可获得该部分满分：

1. FAT16 文件的目录名等元数据存储在哪？数据存储在哪，是怎么组织的？
2. FAT16 系统中需要两个 FAT 表的目的是什么？
3. FAT16 中的根目录和子目录在磁盘存储上有什么区别？
4. FAT16 的文件名存储格式是怎样的，如何转化成实际用户看到的文件名？
5. 不考虑LFN，FAT16 文件系统中支持的最大文件名长度是多少，为什么？
6. FAT16 文件系统的最大单文件的大小是多少，受限于什么因素？
7. FAT16 文件系统分配空间的基本单位是什么，文件系统读写磁盘的基本单位是什么？
8. FAT16 文件系统的根目录数量达到最大值的时候，再次创建目录时，会自动分配空闲簇吗？
9. FAT16 文件系统某个文件夹下的所有目录项都是紧密排列的吗？
10. FAT16文件系统的布局是什么样的？分为哪几个部分，它们的位置在哪？

第四部分 检查内容 & 评分标准

1. 任务一：实现FAT文件系统读操作（**满分2分**）
    - 能够运行 `tree` ， `ls` 命令查看文件目录结构 （**1分**）
    - 能够用过 `cat` / `head` / `tail` 命令，或直接打开文件（如直接打开pdf）查看目录下文件内容（**1分**）
2. 任务二：实现FAT文件系统创建/删除文件操作（**满分2分**）
    - 能够运行 `touch` 命令创建新文件，且保证文件属性的正确填写（**1分**）
    - 能够运行 `rm` 命令删除已有文件，且保证簇的正确释放（**1分**）
3. 任务三：实现FAT文件系统创建/删除文件夹操作（**满分2分**）
    - 能够运行 `mkdir` 命令创建新目录，且新目录中默认具有 `.` 和 `..` 两项（**1分**）
    - 能够运行 `rmdir` 和 `rm -r` 命令删除已存在的目录（**1分**）
4. 任务四：实现FAT文件系统写操作（**满分2分**）
    - 能够使用 `echo` 命令和重定向符 `>` ， `>>` 正确进行文件写入（**2分**）
5. 知识问答（**满分2分**）
    - 回答从知识问答一章中随机抽取的三道问题，每回答正确一题得一分。回答对两题即可得到满分（**2分**）
6. 选做：实现三备份和日志机制（**额外加分，满分2分**）
    - 请参考2.3.1和2.3.2中的检查标准，你需要自行设计方法，展示你的成果，并说明你的实现方案。