



東南大學
SOUTHEAST UNIVERSITY

操作系统实践实验报告

姓名 李宗辉

学号 61522122

教师 张柏礼

2024 年 12 月 31 日

目录

1	文件系统的实现	3
1.1	设计思路和流程	3
1.2	设备创建	3
1.3	分区格式化和加载磁盘	4
1.3.1	数据结构	5
1.3.2	格式化磁盘	6
1.3.3	加载磁盘	7
1.4	注册文件系统	8
1.5	文件夹和文件操作	8
1.5.1	辅助函数说明	8
1.5.2	文件夹操作	10
1.5.3	文件操作	12
1.5.4	获取文件属性	13
2	文件系统测试	14
3	实验疑难与体会	17
3.1	实验疑难	17
3.2	实验体会	17

1 文件系统的实现

实验目的：通过实验完整了解文件系统实现机制。

实验内容：实现具有设备创建、分区格式化、注册文件系统、文件夹操作、文件操作功能的完整文件系统。

1.1 设计思路和流程

由于时间比较紧，而实现完整的文件系统工程量大，需要规范构造磁盘组织和读写方式，且需要在 Linux 内核上修改编译，不方便调试测试，花费较多时间。因此结合网上资料，选择实现了基于 FUSE 架构的简单文件系统，自定义分区格式化，并具备

- 挂载
- 卸载
- 创建和删除文件
- 创建和删除文件夹
- 查看文件夹下文件
- 获取文件属性

的功能。

FUSE 全称为 Filesystem in Userspace，即运行在用户空间上的文件系统，是一种实现在用户态、由应用程序开发者为迎合用户空间需求而专门设计的文件系统。这种机制支持了应用程序开发者提供具有各式各样特性的文件系统，具有很高的灵活性。编写 FUSE 文件系统时，只需要内核加载了 fuse 内核模块即可，不需要重新编译内核。

FUSE 实现了一个对文件系统访问的回调。FUSE 分为内核态的模块和用户态的库两部分。其中用户态的库为程序开发提供接口，也是我们实际开发时用的接口，我们通过这些接口将请求处理功能注册到 FUSE 中。内核态模块是具体的数据流程的功能实现，它截获文件的访问请求，然后调用用户态注册的函数进行处理。具体流程如图 1。

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。因此，要接入 FUSE 框架，就需要通过一系列回调函数来完成对应操作。

1.2 设备创建

设备创建在磁盘上为文件系统分配空间并创建一个虚拟的磁盘映像。可以使用 dd 命令来生成一个磁盘映像文件。

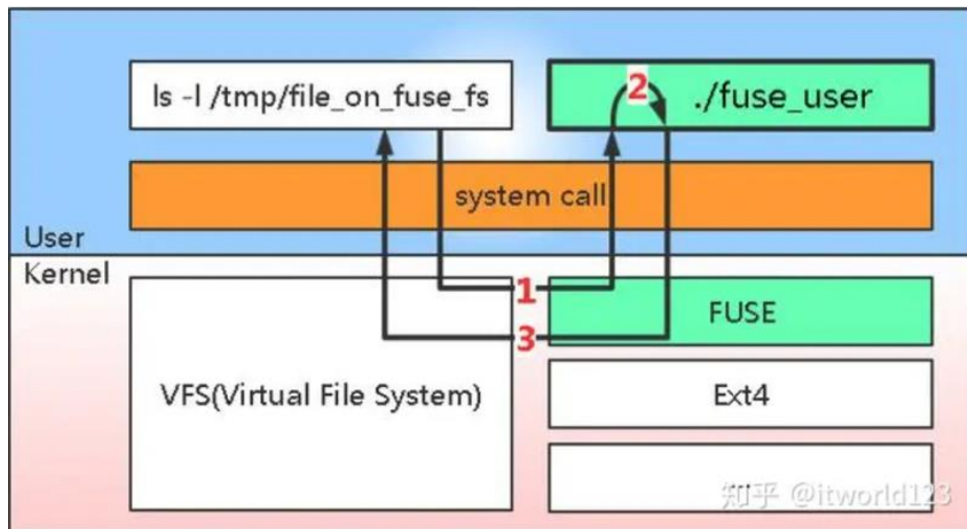


图 1: FUSE 框架

在这里我定义磁盘映像大小未 4096*1024 字节，即 4M，创建一个名为 disk.img 的文件，文件内容全部来自 /dev/zero，后者是一个特殊设备文件，可以生成无限的零字节。这是为了确保创建的磁盘映像是空白的。

```
1 dd if=/dev/zero of=disk.img bs=4096 count=1024
```

1.3 分区格式化和加载磁盘

文件系统设计由自定义完成。，可以参照图 2。系统各部分介绍如下：

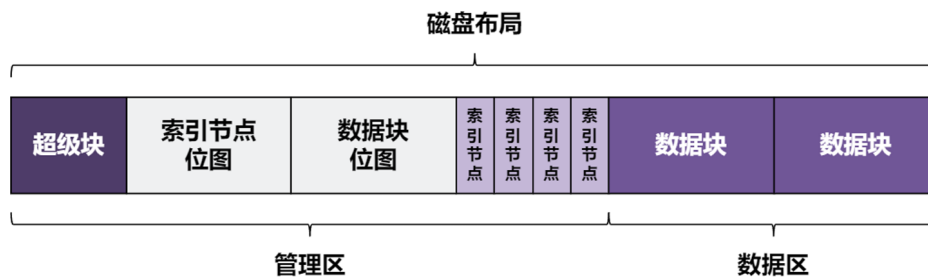


图 2: 文件系统布局

- 超级块：包含整个文件系统和磁盘布局的总体信息，比如文件系统的类型、大小、剩余空间、块大小、节点信息等。超级块通常存储在磁盘的固定位置，是文件系统加载时需要读取的首要结构。
- 索引节点位图：包含文件系统中所有文件的 inode 信息，比如文件名、文件类型、文件大小、起始数据块索引、占用块数、是否有效等信息。

- 数据块位图：数据块位图用于管理数据区的存储空间。每一位对应一个数据块，表示其是否被占用。
- 索引节点：是文件的元数据结构，存储着文件或目录的属性（如权限、大小、时间戳等）以及指向数据块的指针。
- 数据块：文件内容的存储单位。

1.3.1 数据结构

在这里我自主定义了索引节点（inode）和超级块（superblock）两个数据结构；索引节点位图、数据块位图使用数组模拟表示。

对于索引节点结构，我定义了文件名（在这里使用根目录’/’之后的文件路径名）、文件类型、文件大小、起始数据块索引、占用块数、是否有效等信息。对于超级块结构，我定义了当前 inode 数量、空闲块数量、块位图等信息。

```

1  #define BLOCK_SIZE 512          // 每块大小
2  #define TOTAL_BLOCKS 2048      // 块总数
3  #define MAX_FILES 128          // 最大文件数
4  #define MAX_NAME_LEN 255       // 文件/目录名最大长度
5
6  // 文件类型
7  typedef enum { FILE_TYPE, DIR_TYPE } FileType;
8
9  // 索引节点 inode
10 typedef struct {
11     char name[MAX_NAME_LEN];    // 用存储路径代替文件名
12     FileType type;
13     size_t size;                // 文件大小（字节）
14     int start_block;            // 起始数据块索引
15     int block_count;           // 占用块数
16     int is_valid;              // 是否有效
17 } Inode;
18
19 // 超级块
20 typedef struct {
21     int inode_count;            // 当前 inode 数量
22     int free_blocks;           // 空闲块数量
23     char block_bitmap[TOTAL_BLOCKS]; // 数据块位图
24 } SuperBlock;
25

```

```

26     static SuperBlock superblock; // 超级块
27     static Inode inodes[MAX_FILES]; // 索引节点表

```

1.3.2 格式化磁盘

格式化磁盘的过程包括初始化超级块、初始化索引节点表、初始化数据区、写入磁盘等步骤。主要就是：

- 初始化超级块，将 inode_count 设置为 0，free_blocks 设置为总块数，并清空块位图
- 将 inode 表清空，以初始化所有 inode 的状态。
- 创建一个空的数据块，并通过循环将其写入到所有数据块位置，以初始化数据区。
- 将初始化后的超级块和 inode 表写入虚拟磁盘。

```

1     // 模拟设备文件路径
2     static const char *disk_path = "disk.img";
3     static FILE *disk = NULL;
4
5     // 格式化磁盘
6     void init_disk() {
7         disk = fopen(disk_path, "wb+");
8         if (!disk) {
9             perror("Failed to create virtual disk");
10            exit(EXIT_FAILURE);
11        }
12
13        // 初始化超级块
14        superblock.inode_count = 0;
15        superblock.free_blocks = TOTAL_BLOCKS;
16        memset(superblock.block_bitmap, 0, sizeof(superblock.block_bitmap))
; //清
空
17
18        // 初始化 inode 表
19        memset(inodes, 0, sizeof(inodes));
20
21        // 将超级块和 inode 表写入磁盘
22        fwrite(&superblock, sizeof(SuperBlock), 1, disk);

```

```
23     fwrite(inodes, sizeof(Inode), MAX_FILES, disk);
24
25     // 初始化数据区
26     char empty_block[BLOCK_SIZE] = {0};
27     for (int i = 0; i < TOTAL_BLOCKS; i++) {
28         fwrite(empty_block, BLOCK_SIZE, 1, disk);
29     }
30 }
31
32 // 加载磁盘
33 void load_disk() {
34     disk = fopen(disk_path, "rb+");
35     if (!disk) {
36         perror("Failed to open virtual disk");
37         exit(EXIT_FAILURE);
38     }
39
40     fread(&superblock, sizeof(SuperBlock), 1, disk);
41     fread(inodes, sizeof(Inode), MAX_FILES, disk);
42 }
```

1.3.3 加载磁盘

加载磁盘主要是从磁盘中读取超级块和索引节点表，并将其加载到内存中，从而恢复文件系统的状态。

```
1 // 加载磁盘
2 void load_disk() {
3     disk = fopen(disk_path, "rb+");
4     if (!disk) {
5         perror("Failed to open virtual disk");
6         exit(EXIT_FAILURE);
7     }
8
9     fread(&superblock, sizeof(SuperBlock), 1, disk);
10    fread(inodes, sizeof(Inode), MAX_FILES, disk);
11 }
```

1.4 注册文件系统

FUSE 框架提供了 `fuse_main` 函数处理挂载、卸载等操作，并将文件系统注册到内核。用户需要先填充 `fuse_operations`，这是一个包含文件系统操作函数指针的结构体，可以将实现的回调函数赋值给该结构体的相应字段。在 `main` 函数中，调用 `fuse_main` 函数，将 `fuse_operations` 结构体和命令行参数传递给 FUSE 框架，即可完成文件系统注册，并且循环处理用户命令。

```

1 // FUSE 操作定义，右边字段即为自己实现文件系统的回调函数
2 static const struct fuse_operations OSfs_oper = {
3     .getattr = OSfs_getattr,    // 获取文件属性
4     .readdir = OSfs_readdir,   // 读取目录
5     .create = OSfs_create,     // 创建文件
6     .utimens = OSfs_utimens,   // 修改文件时间戳
7     .unlink = OSfs_unlink,     // 删除文件
8     .mkdir = OSfs_mkdir,       // 创建目录
9     .rmdir = OSfs_rmdir,       // 删除目录
10 };
11
12 // 主函数
13 int main(int argc, char *argv[]) {
14     if (access(disk_path, F_OK) != 0) { // 磁盘是否格式化过
15         init_disk();
16     } else {
17         load_disk();
18     }
19     // 启用 FUSE 框架，自动注册文件系统，并进入事件循环，处理用户的请求。
20     return fuse_main(argc, argv, &OSfs_oper, NULL);
21 }

```

1.5 文件夹和文件操作

1.5.1 辅助函数说明

由于文件夹和文件操作涉及到 inode 查找、inode 创建、inode 删除、写入磁盘等操作，因此需要定义了一些辅助函数。

inode 查找函数的思路就是遍历 inode 表，找到对应的 inode 并且 inode 有效，则返回索引。

```

1 // 查找 inode

```



```

2  int find_inode(const char *path) {
3      for (int i = 0; i < MAX_FILES; i++) {
4          if (inodes[i].is_valid && strcmp(inodes[i].name, path) == 0) {
5              return i;
6          }
7      }
8      return -ENOENT;
9  }

```

inode 创建函数的思路就是遍历 inode 表，找到第一个空闲的 inode，将完整文件路径赋值给它，并进行一些初始化操作，然后返回索引。

```

1  // 创建 inode
2  int create_inode(const char *path, FileType type) {
3      for (int i = 0; i < MAX_FILES; i++) {
4          if (!inodes[i].is_valid) {
5              strncpy(inodes[i].name, path, MAX_NAME_LEN);
6              inodes[i].type = type;
7              inodes[i].size = 0;
8              inodes[i].start_block = -1;
9              inodes[i].block_count = 0;
10             inodes[i].is_valid = 1;
11             superblock.inode_count++;
12             save_metadata();
13             return i;
14         }
15     }
16     return -ENOSPC; //无可用 inode, 错误代码
17 }

```

inode 删除函数的思路就是接收要删除的 inode 索引，找到对应的 inode，释放对应的块，并且将该 inode 清零，最后写入磁盘保存修改。

```

1  // 释放块
2  void free_block(int block_idx) {
3      superblock.block_bitmap[block_idx] = 0;
4      superblock.free_blocks++;
5  }
6  // 保存超级块和 inode 表到磁盘

```

```
7     void save_metadata() {
8         fseek(disk, 0, SEEK_SET);
9         fwrite(&superblock, sizeof(SuperBlock), 1, disk);
10        fwrite(inodes, sizeof(Inode), MAX_FILES, disk);
11        fflush(disk);
12    }
13    // 删除 inode
14    void delete_inode(int idx) {
15        for (int i = 0; i < inodes[idx].block_count; i++) {
16            free_block(inodes[idx].start_block + i);
17        }
18        memset(&inodes[idx], 0, sizeof(Inode));
19        superblock.inode_count--;
20        save_metadata();
21    }
```

1.5.2 文件夹操作

文件夹操作包括：

- mkdir: 创建文件夹
- rmdir: 删除文件夹
- ls: 查看文件夹下文件

创建和删除文件夹的思路很简单，文件夹可以被视作一种特殊的文件，因此可以用相同的 inode 结构表示。

mkdir 则创建一个新的 inode 即可，rmdir 则删除对应的 inode 即可。

```
1    // 创建文件夹
2    static int OSfs_mkdir(const char *path, mode_t mode) {
3        //printf(" 创建目录 mkdir: (void) mode;
4
5        if (find_inode(path + 1) != -ENOENT)
6            return -EEXIST;
7
8        return create_inode(path + 1, DIR_TYPE);
9    }
10
11    // 删除文件夹
```

```

12 static int OSfs_rmdir(const char *path) {
13     //printf(" 删除目录 rmdir: int idx = find_inode(path + 1);
14     if (idx == -ENOENT)
15         return -ENOENT;
16
17     if (inodes[idx].block_count > 0)
18         return -ENOTEMPTY; // 目录不为空
19
20     delete_inode(idx);
21     return 0;
22 }

```

查看文件夹下文件，也就是 ls 的功能，需要了解 filler 函数。filler 函数是在 FUSE 框架中使用的一个回调函数，用于填充目录内容。它的作用是将目录下的文件和子目录名称添加到输出缓冲区 buf 中，以使用户或其他系统调用能够读取这些信息。

首先需要添加当前目录和父目录。

接着解析传入的文件夹路径 path，判断是位于根目录还是子目录。

如果是根目录，则遍历 inode 表，通过查找是否只有一个 '/' 字符来判断是否有 inode 的文件路径和根目录相同，有则提取 '/' 之后的部分作为文件名添加到输出缓冲区。

如果是子目录，也就是带有多个 '/' 字符，存储第一个根目录 '/' 之后的路径，用于在遍历 inode 表时对每个 inode 进行前缀目录匹配，找到匹配的 inode，提取 '/' 之后的部分作为文件名添加到输出缓冲区。

```

1 //读取目录 ls
2 static int OSfs_readdir(const char *path, void *buf, fuse_fill_dir_t
filler, off_t offset, struct fuse_file_info *fi, enum fuse_readdir_flags
flags) {
3     // 添加当前目录和父目录
4     filler(buf, ".", NULL, 0, 0); // 当前目录
5     filler(buf, "..", NULL, 0, 0); // 上一级目录
6     // 解析路径
7     const char *trimmed_path = (path[0] == '/') ? path + 1 : path;
8     if (strlen(trimmed_path) == 0) {
9         trimmed_path = NULL; // 根目录
10    }
11    // 遍历 inode 表
12    for (int i = 0; i < MAX_FILES; i++) {
13        if (inodes[i].is_valid) {
14            const char *name = inodes[i].name;

```

```

15     const char *slash_pos = strrchr(name, '/'); // 指向路径中的最
    后一个斜杠，不为 NULL 则有/，表示位于某个目录下
16     // 判断当前文件是否属于指定目录
17     if ((trimmed_path == NULL && slash_pos == NULL) || // 根目
    录文件
18         (slash_pos != NULL && trimmed_path != NULL &&
19         strcmp(trimmed_path, name, slash_pos - name) == 0 &&
    //比较在/前的部分是否相
    同
20         strlen(trimmed_path) == (size_t)(slash_pos - name))) {
    // 子目录文
    件
21         // 提取当前目录下的文件名
22         const char *entry_name = slash_pos ? slash_pos + 1 :
    name;
23         filler(buf, entry_name, NULL, 0, 0);
24     }
25 }
26 }
27 return 0;
28 }

```

1.5.3 文件操作

文件操作包括：

- touch: 创建文件
- rm: 删除文件

创建文件的思路就是首先判断传入的文件路径是否已经被用了，如果没有被用，则创建一个新的 inode，返回成功即可。这里还有一个需要注意的地方：touch 命令不仅创建新文件，也会修改旧文件的时间戳从而实现“创建”新文件。因此还需要修改文件时间戳的回调函数。由于自定义的 inode 数据结构未考虑时间戳的属性，OSfs_utimens 函数直接返回成功（return 0）即可。

删除文件的思路同样也是判断传入的文件路径是否存在，存在则调用 delete_node 函数删除相应的 inode。

```

1     // FUSE 回调：创建文件
2     static int OSfs_create(const char *path, mode_t mode, struct
    fuse_file_info *fi) {

```

```

3      //printf(" 创建文件 create: const char *trimmed_path = (path[0] ==
      '/') ? path + 1 : path;
4      // 检查文件是否已存在
5      if (find_inode(trimmed_path) != -ENOENT)
6          return -EEXIST;
7
8      // 创建 inode, 并传入文件类型和权限
9      int res = create_inode(path, FILE_TYPE);
10     if (res < 0)
11         return res;
12
13     return 0; // 成功
14 }
15 // FUSE 回调: 文件时间戳
16 static int OSfs_utimens(const char *path, const struct timespec tv[2],
struct fuse_file_info *fi) {
17     return 0;
18 }
19
20 // FUSE 回调: 删除文件
21 static int OSfs_unlink(const char *path) {
22     const char *trimmed_path = (path[0] == '/') ? path + 1 : path;
23     // 查找 inode
24     int idx = find_inode(trimmed_path);
25     if (idx < 0) // 错误处理
26         return idx;
27     // 删除 inode
28     delete_inode(idx);
29     return 0; // 成功
30 }

```

1.5.4 获取文件属性

获取文件属性，对应 stat 命令，需要根据 inode 的类型设置文件属性。首先清空 stbuf 结构体，接着根据传入的文件路径判断是否是根目录，如果是，则设置其权限和硬链接数量。不是根目录，调用 find_inode 函数查找 inode 索引，根据查找到的 inode 的类型设置文件类型、权限、硬链接数量。对于目录，硬链接数量为 2，因为目录链接一个指向自己，一个指向父目录；对于普通文件，硬链接数量为 1。

```

1 //获取文件属性
2 static int OSfs_getattr(const char *path, struct stat *stbuf, struct
fuse_file_info *fi) {
3     // 清空 stbuf 结构体
4     memset(stbuf, 0, sizeof(struct stat));
5     // 检查是否请求的是根目录
6     if (strcmp(path, "/") == 0) {
7         stbuf->st_mode = S_IFDIR | 0755; //允许所有用户访问，但只有所有
者能够修改内容。
8         stbuf->st_nlink = 2; //硬链接数量
9         return 0;
10    }
11    //查找对应的 inode 索引
12    int idx = find_inode(path + 1);
13    if (idx == -ENOENT)
14        return -ENOENT;
15    // 根据 inode 的类型设置文件属性
16    if (inodes[idx].type == DIR_TYPE) { // 目录
17        stbuf->st_mode = S_IFDIR | 0755;
18        stbuf->st_nlink = 2;
19    } else { // 文件
20        stbuf->st_mode = S_IFREG | 0644; //允许所有用户读文件，但只有所
有者能够写。
21        stbuf->st_nlink = 1;
22        stbuf->st_size = inodes[idx].size;
23    }
24    return 0;
25 }

```

2 文件系统测试

本部分对文件系统的挂载、卸载、创建和删除文件夹、创建和删除文件、查看文件夹下文件、获取文件属性的功能进行了测试。

首先创建设备 disk.img。

```

1 dd if=/dev/zero of=disk.img bs=4096 count=1024

```

接着编译 OSfs.c 文件，也就是进行磁盘初始化以及 FUSE 回调函数的代码文件。

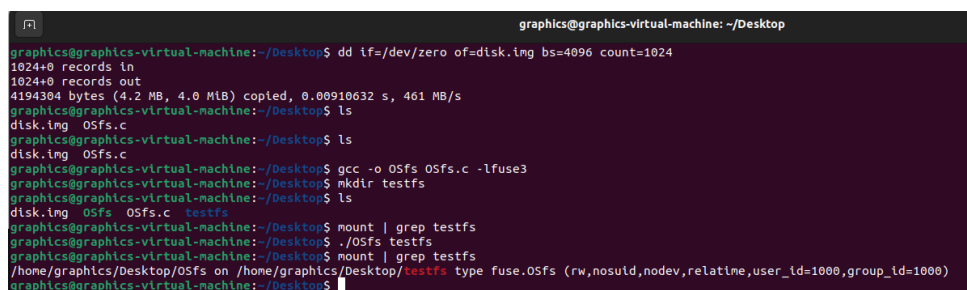
```
1 gcc -o OSfs OSfs.c -lfuse3
```

创建一个文件夹 testfs，并运行 OSfs，将 disk.img 作为文件系统的设备，挂载到 testfs 目录。

```
1 ./OSfs testfs
```

通过命令查看挂载情况，结果如图 3 所示，可以看到 testfs 目录已经挂载成功。

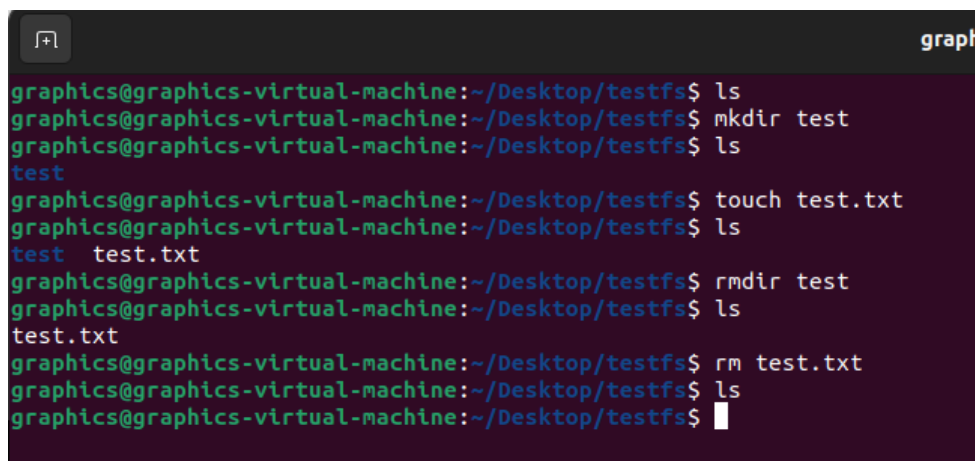
```
1 mount | grep testfs
```



```
graphics@graphics-virtual-machine: ~/Desktop
graphics@graphics-virtual-machine:~/Desktop$ dd if=/dev/zero of=disk.img bs=4096 count=1024
1024+0 records in
1024+0 records out
4194304 bytes (4.2 MB, 4.0 MiB) copied, 0.00910632 s, 461 MB/s
graphics@graphics-virtual-machine:~/Desktop$ ls
disk.img  OSfs.c
graphics@graphics-virtual-machine:~/Desktop$ ls
disk.img  OSfs.c
graphics@graphics-virtual-machine:~/Desktop$ gcc -o OSfs OSfs.c -lfuse3
graphics@graphics-virtual-machine:~/Desktop$ mkdir testfs
graphics@graphics-virtual-machine:~/Desktop$ ls
disk.img  OSfs  OSfs.c  testfs
graphics@graphics-virtual-machine:~/Desktop$ mount | grep testfs
graphics@graphics-virtual-machine:~/Desktop$ ./OSfs testfs
graphics@graphics-virtual-machine:~/Desktop$ mount | grep testfs
/home/graphics/Desktop/OSfs on /home/graphics/Desktop/testfs type fuse.OSfs (rw,nosuid,nodev,relatime,user_id=1000,group_id=1000)
graphics@graphics-virtual-machine:~/Desktop$
```

图 3: 挂载情况

由于 disk.img 是格式化后的，因此不包含任何文件，进入 testfs 空目录，测试创建/删除文件和文件夹以及查看文件夹下文件功能在图 4 中，我对 mkdir、touch、rmdir、rm、ls 命令进行了测试。可见都很好符合预期。



```
graphics@graphics-virtual-machine:~/Desktop/testfs$ ls
graphics@graphics-virtual-machine:~/Desktop/testfs$ mkdir test
graphics@graphics-virtual-machine:~/Desktop/testfs$ ls
test
graphics@graphics-virtual-machine:~/Desktop/testfs$ touch test.txt
graphics@graphics-virtual-machine:~/Desktop/testfs$ ls
test  test.txt
graphics@graphics-virtual-machine:~/Desktop/testfs$ rmdir test
graphics@graphics-virtual-machine:~/Desktop/testfs$ ls
test.txt
graphics@graphics-virtual-machine:~/Desktop/testfs$ rm test.txt
graphics@graphics-virtual-machine:~/Desktop/testfs$ ls
graphics@graphics-virtual-machine:~/Desktop/testfs$
```

图 4: 测试

随意创建几个文件和文件夹，如图 5 所示。

```

graphics@graphics-virtual-machine:~/Desktop/testfs$ ls
graphics@graphics-virtual-machine:~/Desktop/testfs$ mkdir test
graphics@graphics-virtual-machine:~/Desktop/testfs$ touch test.txt
graphics@graphics-virtual-machine:~/Desktop/testfs$ ls
test  test.txt
graphics@graphics-virtual-machine:~/Desktop/testfs$ cd test
graphics@graphics-virtual-machine:~/Desktop/testfs/test$ ls
graphics@graphics-virtual-machine:~/Desktop/testfs/test$ touch test2.txt
graphics@graphics-virtual-machine:~/Desktop/testfs/test$ ls
test2.txt
graphics@graphics-virtual-machine:~/Desktop/testfs/test$

```

图 5: 创建文件和文件夹

分别选取一个普通文件如 test.txt 和一个文件夹 test 查看其属性，结果如图 6 所示。

```

graphics@graphics-virtual-machine:~/Desktop/testfs/test$ cd ..
graphics@graphics-virtual-machine:~/Desktop/testfs$ ls
test  test.txt
graphics@graphics-virtual-machine:~/Desktop/testfs$ stat test
  File: test
  Size: 0          Blocks: 0          IO Block: 4096   directory
Device: 32h/50d Inode: 10          Links: 2
Access: (0755/drwxr-xr-x)  Uid: (  0/   root)   Gid: (  0/   root)
Access: 1970-01-01 08:00:00.000000000 +0800
Modify: 1970-01-01 08:00:00.000000000 +0800
Change: 1970-01-01 08:00:00.000000000 +0800
 Birth: -
graphics@graphics-virtual-machine:~/Desktop/testfs$ stat test.txt
  File: test.txt
  Size: 0          Blocks: 0          IO Block: 4096   regular empty file
Device: 32h/50d Inode: 11          Links: 1
Access: (0644/-rw-r--r--)  Uid: (  0/   root)   Gid: (  0/   root)
Access: 1970-01-01 08:00:00.000000000 +0800
Modify: 1970-01-01 08:00:00.000000000 +0800
Change: 1970-01-01 08:00:00.000000000 +0800
 Birth: -
graphics@graphics-virtual-machine:~/Desktop/testfs$

```

图 6: 查看文件属性

由文件属性可以看出 BlockSize 为 4096 字节，符合 disk.img 的初始创建设定。而 test 文件夹的 Inode 字段为 10，这正好符合其是第 10 个创建的 inode；而 test.txt 文件是第 11 个创建的 inode。文件夹的硬链接数量为 2，权限为 0755；文件的硬链接数量为 1，权限为 0644。而底下的时间戳由于本身也没有设定，更没有修改，访问、修改和变更时间戳均为 1970-01-01 08:00:00，这表明此文件在创建时未被修改过。可见属性都符合回调函数设定。

这也证明了在 FUSE 框架中使用终端命令调用的是框架内自实现的回调函数，而不是系统本身的调用。

最后，卸载 testfs 目录，并且重新挂载，结果如图 7 所示。

```
1  umount testfs
```

可见在卸载后，再次显示 testfs 目录内容，显示为空，说明已经卸载成功。


```
graphics@graphics-virtual-machine:~/Desktop/testfs$ ls
test  test.txt
graphics@graphics-virtual-machine:~/Desktop/testfs$ cd ..
graphics@graphics-virtual-machine:~/Desktop$ umount testfs
graphics@graphics-virtual-machine:~/Desktop$ ls testfs
graphics@graphics-virtual-machine:~/Desktop$ ls
disk.img  OSfs  OSfs.c  testfs
graphics@graphics-virtual-machine:~/Desktop$ ./OSfs testfs
graphics@graphics-virtual-machine:~/Desktop$ ls testfs
test  test.txt
graphics@graphics-virtual-machine:~/Desktop$
```

图 7: 卸载和重新挂载

重新挂载后，再次显示 testfs 目录内容，显示与卸载前内容相同，这说明内容是写到了虚拟磁盘上的，重新挂载会读取当前磁盘上的内容。文件系统的挂载和卸载是成功的。

3 实验疑难与体会

3.1 实验疑难

本实验遇到的主要困难主要是理解文件系统的原理结构，并简化设计一个自己的文件系统。在编写函数时我一直苦于判断使用什么数据结构？功能的具体实现逻辑是什么？此外，也需要查阅大量资料了解 FUSE 框架回调函数的接口具体形式。

实验参考资料：

- <https://blog.jeffli.me/blog/2014/08/30/use-gdb-to-understand-fuse-file-system/>
- <https://zhuanlan.zhihu.com/p/59354174>
- <https://blog.csdn.net/dillanzhou/article/details/82856358>
- <https://ouonline.net/building-your-own-fs-with-fuse-1>
- https://blog.csdn.net/m0_47524163/article/details/136284693

3.2 实验体会

由于时间紧、工程量大，采用了取巧的办法实现了一个简单的文件系统，功能也有限；基本的读写文件操作较为复杂，未能来得及实现。完整的文件系统是复杂的，而即使本阉割后的文件系统，实现过程中也出现了很多问题。不过，值得庆幸的是，因为使用的 FUSE 框架主要是在用户态空间编写运行，便于调试，这大大加快了文件系统的实现过程。