哈爾濱工業大學（深圳）
HARBIN INSTITUTE OF TECHNOLOGY

实验报告

开课学期：2024 秋季

课程名称：操作系统

实验名称：基于 FUSE 的青春版 EXT2 文件系统

学生班级：22 级 4 班

学生学号：220110430

学生姓名：吴梓滔

评阅教师：

报告成绩：

实验与创新实践教育中心制

2024 年 9 月

一、实验详细设计

图文并茂地描述实验实现的所有功能和详细的设计方案及实验过程中的特色部分。

1、 总体设计方案

详细阐述文件系统的总体设计思路，包括系统架构图和关键组件的说明。

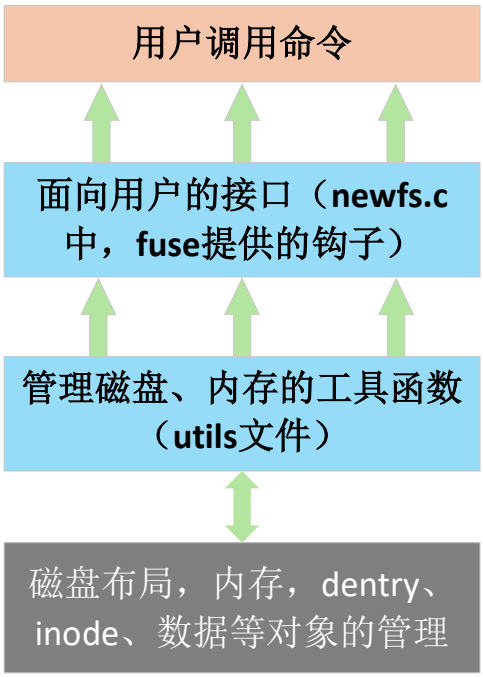

图 1 系统架构图

如图所示，fuse 架构文件系统的设计可以分为 2 个大部分，一个部分是面向用户调用命令的由 fuse 提供的钩子函数。这一部分包含 mkdir、mknod、init、destroy、rename、write、read 等功能的函数，这一部分不依赖于磁盘布局的设计，只需要调用下一层实现好的工具函数构建。另一部分是为了实现上述函数，基于具体磁盘布局、内存中对象管理方案来实现的工具函数，这些工具函数和磁盘布局等紧密相关，负责具体实现磁盘的读写、内存中对象的管理，被上一层接口调用去实现面向用户的特定功能。

(一)虚拟磁盘和磁盘布局

本实验使用了一个 4MB 大小的虚拟磁盘以及模拟驱动 ddriver，通过 ddriver 提供的一些调用来模拟磁盘操作的调用。磁盘的 IO 块大小为 512B。虚拟磁盘挂载在 tests/mnt/目录下，当虚拟磁盘挂载后，磁盘内的文件经过正常的读取就会显示在 mnt/目录下。磁盘卸载后，mnt/下就没有磁盘里的文件了。由于文件系统规定的逻辑块大小为 1024B，两次磁盘 IO 读取 1 个逻辑块。下文的“磁盘块”均代表 1024B 的逻辑块。

超级块 1块	Inode位图 1块	数据块位图 1块	Inode块 315块	数据块 3778块

图 2 磁盘块分布示意图

本设计实现的是 ext2 文件系统，支持全部的用户操作，包含挂载、卸载，创建、显示、读写、删除、重命名文件或目录。在磁盘布局上，规定块大小为 1kB，对 4MB 大小的模拟磁盘分为超级块、inode 位图、数据块位图、inode、数据五个部分，分别占用 1 块、1 块、1 块、315 块、3778 块，如图 2 所示。各部分设计如下：

1) 超级块：用于存储磁盘的元数据，包括磁盘布局信息，文件系统的一些约定。这些信息不会超过一个块的大小，使用 1 个磁盘块来存储。
2) Inode 位图：索引节点位图，使用位图来标记 inode 块的使用情况，每一个 bit 对应到一个 inode 的使用情况。对于 1024B 的位图，能够表示最多 8192 个索引节点，已经足够该磁盘和文件系统使用。
3) 数据块位图：使用位图来标记数据块的使用情况，每一个 bit 对应到一个数据块的使用情况。位图 1024B，能够最多表示 8192 个数据块，已经足够该磁盘和文件系统使用。
4) Inode 块：连续存放 inode。在该文件系统中，一个 inode 的长度设计为 128B，在磁盘中，一个磁盘块能够恰好存放 8 个 inode。考虑到存放尽可能多的文件，inode 块准备了 315 块，即最多可以创建 2520 个文件（包括根目录在内）。
5) 数据块：存放文件和目录的数据。在本设计中，规定一个文件最多使用 12 个数据块，即一个文件的大小最大为 12KB。前面四个部分未使用的剩余磁盘块全部作为数据块，共有 3778 个数据块。

```
20  /*******************************************************************
21  * SECTION: FUSE操作定义
22  *******************************************************************
23  static struct fuse_operations operations = {
24      .init = newfs_init,                     /* mount文件系统 */
25      .destroy = newfs_destroy,               /* umount文件系统 */
26      .mkdir = newfs_mkdir,                   /* 建目录，mkdir */
27      .getattr = newfs_getattr,               /* 获取文件属性，类似stat，必须完
28      .readdir = newfs_readdir,               /* 填充dentrys */
29      .mknod = newfs_mknod,                   /* 创建文件，touch相关 */
30      .write = newfs_write,                  /* 写入文件 */
31      .read = newfs_read,                    /* 读文件 */
32      .utimens = newfs_utimens,               /* 修改时间，忽略，避免touch报错
33      .truncate = newfs_truncate,            /* 改变文件大小 */
34      .unlink = newfs_unlink,                /* 删除文件 */
35      .rmdir  = newfs_rmdir,                /* 删除目录， rm -r */
36      .rename = newfs_rename,                 /* 重命名，mv */
37
38      .open = newfs_open,
39      .opendir = newfs_opendir,
40      .access = newfs_access
41  };
```

(二)文件系统的实现方案

向用户提供的功能只需实现 fuse_operations 的接口。如图所示，本设计实现全部钩子函数。

具体实现功能的工具函数要负责 inode、dentry 等对象的磁盘结构、内存结构的管理和操作。在内存中，表示目录的 inode 可以通过链表来管理该目录下的子目录项 dentry，表示文件的 inode 可以通过指针来管理该文件的数据。为了灵活地管理文件的关系，内存中的 inode 和 dentry 拥有多种指针字段来记录相互关系，这些指针字段在写回磁盘时不需要写回。

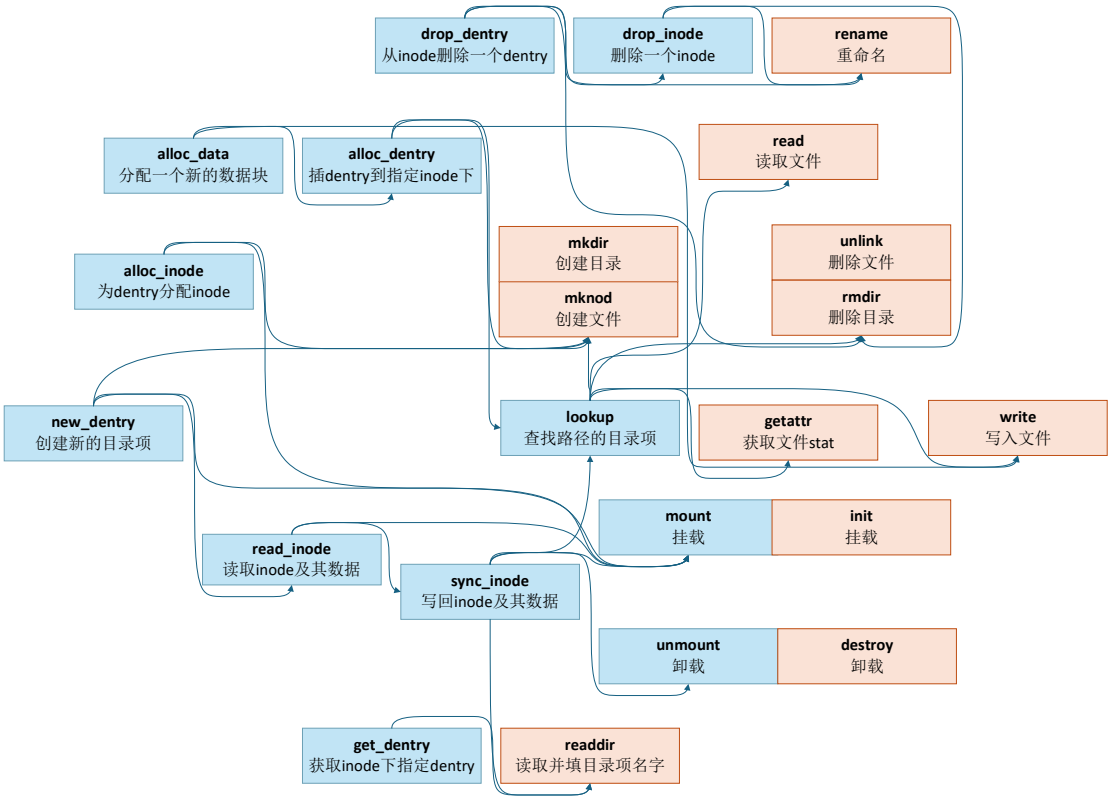

图 3 关键示意图函数

如图所示，设计的关键工具函数由蓝色标注，面向用户操作的函数由红色标注（即 fuse 的钩子函数），函数设计结构如图 3 所示。如果一个函数 A 指向了函数 B 有一条有向边，说明 B 的实现依赖于 A。实现 fuse 框架的全部钩子函数后，fuse 框架将能够提供对各种用户操作的支持。

2、 功能详细说明

每个功能点的详细说明（关键的数据结构、核心代码、流程等）

(一)关键数据结构

文件系统涉及到超级块、索引节点 inode、目录项 dentry 三种重要对象，在磁盘中存储的形式和在内存中操作的形式不相同，需要根据内存和磁盘分别创建数据结构。

普通文件和目录文件采用枚举类型来定义。

```
typedef enum file_type {
    WZT_FILE,           // 普通文件,不能叫FILE，否则和标准库撞车
    WZT_DIR             // 目录文件
} FILE_TYPE;
```

图 4 文件类型枚举

内存中的超级块结构，需要存储一些文件系统的元数据之外，还需要比磁盘内存储的结

构增加一个根目录项的指针，并通过指针直接指向内存中的 inode 位图、数据块位图。

```
57  struct newfs_super {
58      uint32_t magic;
59      int      fd;
60      uint32_t sz_io;          // io大小
61      // 逻辑块大小和块数
62      uint32_t block_size;     // 块大小
63      uint32_t block_nums;     // 块数
64      // 磁盘大小
65      uint32_t disk_size;
66      int                 sz_usage;           /* ioctl 相关信息 */
67      // 索引节点位图起始块号
68      uint32_t inode_bitmap_block;
69      uint8_t*            map_inode;           // 指向 inode 位图的内存起点
70      // inode位图块数
71      uint32_t map_inode_blks;
72      // inode位图偏移
73      uint32_t map_inode_offset;
74      // 数据块位图起始块号
75      uint32_t data_bitmap_block;
```

```
76      uint8_t*            map_data;           // 指向 data 位图的内存起点
77      // 数据块位图块数
78      uint32_t map_data_blks;
79      // 数据块位图偏移
80      uint32_t map_data_offset;
81      // 索引节点起始offset
82      uint32_t inode_offset;
83      // inode信息
84      uint32_t inode_size;     // inode大小
85      uint32_t inode_nums;     // inode数(最大支持的inode数量)
86      // 数据块起始块号
87      uint32_t data_offset;
88      uint32_t data_block;
89      //数据块最大数目
90      uint32_t max_data;
91      // 根目录索引
92      uint32_t root_inode;
93      struct newfs_dentry* root_dentry;
94      // 是否挂
95      boolean is_mounted;
96  };
```

图 5 内存中超级块结构

磁盘中的超级块结构需要存储幻数、inode 位图块数，数据块位图块数，以及 inode 块的起始位置 offset，数据块起始位置 offset。存储这些信息就足够在挂载时读入并设置好内存中超级块结构。

```c
struct newfs_super_d {
    uint32_t          magic;
    uint32_t          sz_usage;

    uint32_t          map_inode_blks;     /* inode 位图占用的块数 */
    uint32_t          map_inode_offset;   /* inode 位图在磁盘上的偏移 */

    uint32_t          map_data_blks;      /* data 位图占用的块数 */
    uint32_t          map_data_offset;    /* data 位图在磁盘上的偏移 */

    uint32_t          inode_offset;       /* 索引结点的偏移 */
    uint32_t          data_offset;        /* 数据块的偏移*/
};
```

图 6 磁盘中超级块结构

索引节点用于指向文件或目录对应的数据块，每个文件的索引节点都由该文件的目录项（dentry）指向它。在磁盘中，这种指向是通过保存对应数据块的块号或者索引节点号来实现的。当索引节点和目录项、数据块读入内存后，为了更方便地管理，使用指针来快速地指向内存中的对象。

```c
111 struct newfs_inode {
112     uint32_t ino;
113     /* TODO: Define yourself */
114     // 文件大小
115     uint32_t size;
116     // 连接数
117     uint32_t link;
118     // 文件类型
119     FILE_TYPE type;
120     int                 dir_cnt;  // 维护目录项数量
121     struct newfs_dentry* dentry;                        /* 指向该inode的dentry */
122     struct newfs_dentry* dentrys;                       /* 所有目录项 */
123 
124     // 数据块索引
125     uint32_t data_block[DATA_PER_FILE];     // 数据块索引，就是一个整数块号
126     uint8_t* data_ptr[DATA_PER_FILE];           // 指向数据块的内存起点
127 
128     // 已经使用的块数
129     int block_used;
130 };
```

图 7 内存中的 inode 结构

内存中的 inode 保存了指向该 inode 的目录项，即该文件对应的目录项 dentry。如果该 inode 对应的是目录，那么在内存中通过一个 dentrys 链表来管理该目录所拥有的目录项。如果该 inode 对应的是文件，则通过一系列指针 data_ptr 来指向内存中的数据块。此外，通过一个 block_used 记录已经分配给该 inode 对应文件的数据块数。

值得注意的是，当数据从磁盘读入内存时，一个文件的所有数据在内存中是连续存储的，对于 data_ptr 这个指针数组，总是有 data_ptr[i+1]=data_ptr[i]+1024。

磁盘中的索引节点如图 8 所示。磁盘中的索引节点比内存中去除了指针字段，只保存最大数目（12 个）的数据块号，用于指向该文件对应的数据块位置。磁盘中索引节点结构还增加了 56 字节的填充，不保存任何有意义信息，仅用于把索引节点的大小凑满 128 字节。为了让索引节点的大小与 1024B 对齐，使一个磁盘块恰好能够存放整数个索引节点，将索引节点凑满 128B，从而一个磁盘块中能够恰好存放 8 个索引节点。由于是对齐存放，这样的优势是只要知道索引节点号 ino 和磁盘中索引节点起始偏移 inode_offset(保存在超级块)，

即可通过 inode_offset + ino * 128 来定位目标索引节点在磁盘中的位置。inode 在磁盘中的存储如图 9 所示。

```c
// inode的大小是128B，所以一个块可以放8个inode
struct newfs_inode_d {
    uint32_t           ino;                        // 索引编号
    uint32_t           size;                       // 文件已占用空间
    uint32_t           link;                       // 链接数，默认为1
    FILE_TYPE          type;                       // 文件类型（目录类型、普通文件类型）
    int                data_block[DATA_PER_FILE];  // 数据块指针（可固定分配）
    int                dir_cnt;                    // 如果是目录类型文件，下面有几个目录
    int                block_used;                 // 已经使用的块数

    // 前面18个int， 72字节
    // 凑成可以让1024整除的字节数，凑到128B，所以还差56字节
    char               padding[56];                // 填充字节
};
```

图 8 磁盘中的索引节点

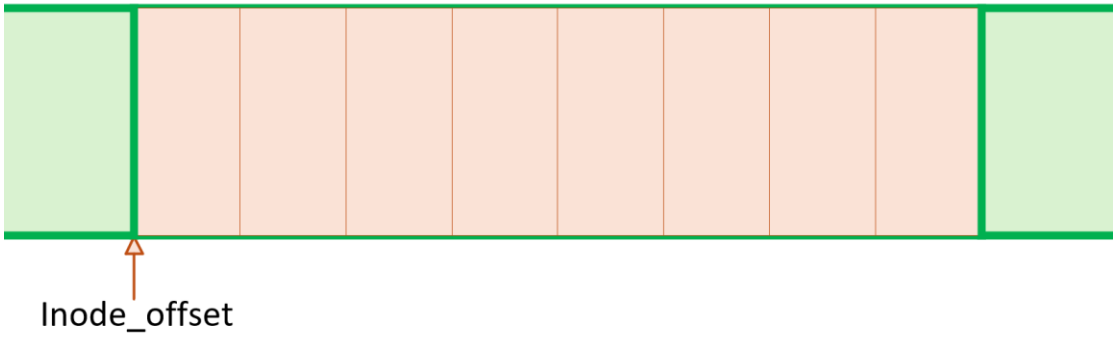

图 9 inode 在磁盘中的存储示意

目录项用于指向该文件的索引节点。在内存中，目录项由父目录索引节点直接通过链表来管理，同时目录项也能指向自己的兄弟目录项和父目录项，以及对应的索引节点。

```c
struct newfs_dentry {
    char     name[MAX_NAME_LEN];
    uint32_t ino;
    /* TODO: Define yourself */

    struct newfs_dentry* brother;
    struct newfs_dentry* parent;

    struct newfs_inode*  inode;                    /* 指向inode */

    // 文件类型
    FILE_TYPE type;
};
```

图 10 目录项 dentry 在内存中的结构

在磁盘中，目录项只需要记录文件名、索引节点号、文件类型即可。一个目录项在磁盘中的大小是 136 字节。

```c
struct newfs_dentry_d {
    char            name[MAX_NAME_LEN];                 // 文件名
    uint32_t        ino;                                // 指向的ino号
    FILE_TYPE       type;                              // 文件类型
};
```

图 11 目录项在磁盘中的结构

(二)文件的表示

一个文件或目录，是先有 dentry，再有 inode。在磁盘挂载后，会直接在内存中创建根目录项，根目录项只会出现在内存中而不会出现在磁盘。再通过读取或者新建跟索引节点，通过根目录的数据块，可以检索到根目录下的各种目录项，从这些目录项出发再找到对应 inode，再找到对应数据块。操作一个文件时，是通过 dentry 索引到 inode，再通过 inode 索引到子目录项等。通过实现相应的工具函数，可以在读入一个 inode 时，将对应的子目录项/数据也读入内存。

(三)详细设计

(1)常量宏定义和宏定义函数

为一些常用的量进行宏定义。如磁盘布局，报错信息等。

```c
#define MAX_NAME_LEN    128

#define LOGIC_BLOCK_SIZE 1024
#define IO_BLOCK_SIZE    512
#define DISK_SIZE        1024*1024*4
#define DATA_PER_FILE 12   // 每个文件数据块数

#define DRIVER_FD()       (super.fd)     // 暂时放着

typedef int          boolean;
#define TRUE                  1
#define FALSE                 0


#define SUPER_OFFSET          0
#define SUPER_BLKS            1
#define MAP_INODE_BLKS        1
#define MAP_DATA_BLKS         1
#define INODE_BLKS           315 // 暂时放着
#define DATA_BLKS            3778      //2024/12/7 补上宏定义

// 定义MAGIC
#define MAGIC_NUM             0x12345678
```

图 12 常量宏定义

为高度复用的计算进行函数宏定义。

```c
// 求偏移量下界
#define OFFSET_ROUND_DOWN(value, round) ((value) % (round) == 0 ? (value) : ((value) / (round)) * (round))
// 求偏移量上界
#define OFFSET_ROUND_UP(value, round) ((value) % (round) == 0 ? (value) : ((value) / (round) + 1) * (round))
// 块数求字节数
#define BLOCKS_SIZE(blks)               ((blks) * LOGIC_BLOCK_SIZE)

// 计算偏移
#define INODE_OFFSET(ino)               (super.inode_offset + (ino) * sizeof(struct newfs_inode_d)) // 这个
#define DATA_OFFSET(bno)              (super.data_offset + BLOCKS_SIZE(bno))
// 判断inode类型
#define INODE_IS_DIR(pinode)              (pinode->dentry->type == WZT_DIR)
#define INODE_IS_REG(pinode)              (pinode->dentry->type == WZT_FILE)

#define DENTRY_PER_BLK()            ((BLK_SZ()) / sizeof(struct newfs_dentry_d)) // 这边问题：要用dentry还是den



#define BLK_SZ()                    (super.block_size)

// 复制文件名到某个dentry中
#define WZT_ASSIGN_FNAME(pjfs_dentry, _fname) memcpy(pjfs_dentry->name, _fname, strlen(_fname))
```

图 13 宏定义函数

这一部分宏定义了一些在代码中多次使用的函数。关键的有磁盘偏移量计算的 INODE_OFFSET(ino),能够根据输入的索引节点号 ino，直接确定该索引节点在磁盘中的偏移量。是通过访问内存 super 块的 inode_offset，加上 ino 和索引节点大小 128B 的乘积来实现的。在需要读写索引节点时，都会调用这个计算来确定磁盘中读写的位置。DATA_OFFSET 根据数据块号 bno，计算出这个数据块起点在磁盘中的偏移量，也是通过访问 super 块的 data_offset，获取到数据块的起点，再加上数据块号 bno 和块长 1024B 的乘积实现的。

此外，还实现了计算每个数据块能够存放的 dentry 个数的函数 DENTRY_PER_BLK,这个函数在读写目录项到磁盘时有重要作用。简单地通过磁盘块大小除以目录项长度来实现。值得注意的是，这里所计算的偏移或者存放个数，均是在磁盘中讨论的，使用的都是磁盘中存储的数据结构。

OFFSET_ROUND_DOWN 和 OFFSET_ROUND_UP 用于确定给定的偏移量上下界，在封装磁盘读写操作时使用。在磁盘驱动封装一块详细说明。

(2)磁盘驱动封装

由于磁盘驱动提供的读写只能按照 IO 块来读，并且 IO 块和逻辑磁盘块大小不同，一次磁盘读取包括通过 seek 进行磁盘头的定位，且 seek 的定位也需要与 IO 块对齐。再通过 read 读取需要的 IO 块，在从这些 IO 块中读取需要的信息。这样的磁盘操作太过于底层。在文件系统的实现中，理想的读写调用是给定读写的起点位置，给定读写数据大小，给定读或写数据的内存地址，就可以完成这样的调用。因此，要对磁盘 IO 操作做一个封装，提供给后续的操作来调用。理想的接口声明如图 14 所示。

```c
int newfs_driver_read(int offset, uint8_t *out_content, int size);
int newfs_driver_write(int offset, uint8_t *in_content, int size);
```

图 14 理想的磁盘 IO 接口

对磁盘操作进行封装需要进行磁盘数据的 IO 块定位。给定磁盘中数据的起点 offset 和 size，要通过 offset 和 size 定位数据所在若干个的 IO 块，把这些 IO 块全部读入内存中暂存，在从这些数据中取出目标数据。宏定义函数 OFFSET_ROUND_DOWN 可以找到一个地址的按块对齐的下界，OFFSET_ROUND_UP 可以找到上界。

```c
int newfs_driver_read(int offset, uint8_t *out_content, int size) {
    int      offset_aligned = OFFSET_ROUND_DOWN(offset, LOGIC_BLOCK_SIZE);  //对齐按1024
    int      bias           = offset - offset_aligned;
    int      size_aligned   = OFFSET_ROUND_UP((size + bias), LOGIC_BLOCK_SIZE);
    uint8_t* temp_content   = (uint8_t*)malloc(size_aligned);   // 申请一个上下都对齐了的缓冲
    uint8_t* cur            = temp_content;                     // 缓冲区指针


    ddriver_seek(DRIVER_FD(), offset_aligned, SEEK_SET);
    while (size_aligned != 0)
    {
        ddriver_read(DRIVER_FD(), cur, IO_BLOCK_SIZE);
        cur          += IO_BLOCK_SIZE;              // 读写磁盘按照io块大小512来读
        size_aligned -= IO_BLOCK_SIZE;
    }
    memcpy(out_content, temp_content + bias, size);
    free(temp_content);

    return ERROR_NONE;  // return 0;

} // absolutely no problem
```

图 15 读磁盘操作的封装

```c
int newfs_driver_write(int offset, uint8_t *in_content, int size) {
    int      offset_aligned = OFFSET_ROUND_DOWN(offset, LOGIC_BLOCK_SIZE);
    int      bias           = offset - offset_aligned;
    int      size_aligned   = OFFSET_ROUND_UP((size + bias), LOGIC_BLOCK_SIZE);

    uint8_t* temp_content   = (uint8_t*)malloc(size_aligned);
    uint8_t* cur            = temp_content;

    newfs_driver_read(offset_aligned, temp_content, size_aligned);  // 此处直接调用上面接口

    memcpy(temp_content + bias, in_content, size);
    ddriver_seek(DRIVER_FD(), offset_aligned, SEEK_SET);
    while (size_aligned != 0)
    {
        ddriver_write(DRIVER_FD(), cur, IO_BLOCK_SIZE);
        cur          += IO_BLOCK_SIZE;
        size_aligned -= IO_BLOCK_SIZE;
    }
    free(temp_content);

    return ERROR_NONE;  // return 0;
} // absolutely no problem
```

图 16 写磁盘操作的封装

这两个函数的代码逻辑模仿参考实现 simplefs。在后续的工具函数设计中，磁盘读写全部调用这两个函数。

(3)工具函数的实现

这一部分为工具函数（utils 文件）的实现，即图 3 中蓝色标注的部分。

1. new_dentry 函数

函数声明为 struct newfs_dentry* new_dentry(const char* name, FILE_TYPE type)，根据指定的名称和类型，创建一个新的内存中目录项并返回。这个函数再挂载时要用来创建根目录项。每次新建一个文件或目录时，也需要调用来创建相应的目录项。核心代码如图 17 所示。

```c
struct newfs_dentry* new_dentry(const char* name, FILE_TYPE type) {
    struct newfs_dentry* dentry = (struct newfs_dentry*)malloc(sizeof(struct newfs_dentry));
    memset(dentry, 0, sizeof(struct newfs_dentry));
    WZT_ASSIGN_FNAME(dentry, name);
    dentry->type = type;
    dentry->ino = -1;
    dentry->inode   = NULL;
    dentry->parent  = NULL;
    dentry->brother = NULL;  // 一共六个字段
    return dentry;
}
```

图 17 new_dentry 核心代码

新创建一个 dentry 时，是通过 malloc 申请内存空间，创建在堆中；内存空间的释放发生在其他过程。初始先完成名称和类型的设置。其他字段如索引节点 ino，inode，兄弟目录项或父目录项的设置，都在函数的外层调用者处完成。

2. newfs_alloc_dentry

这一函数的作用是在内存中，将一个 dentry 通过头插法，插入到父目录 inode 的 dentrys 链表中。在内存中，目录的 inode 通过链表管理该目录下的目录项，当该 inode 需要写回时，会对这些目录项对应数据块写回。在分配的时候，还同时考虑当前 inode 对应的数据块是否够存储目录项，如果不够时会申请新的目录项。将目录项插入时采用头插法。

目录类型的数据块申请是在为其插入目录项时按需申请的。该函数关键代码如下：

```
if (inode->dentrys == NULL) {
    inode->dentrys = dentry;
}
else {
    dentry->brother = inode->dentrys;
    inode->dentrys = dentry;
}
inode->dir_cnt++;
inode->size += sizeof(struct newfs_dentry_d);
if (inode->dir_cnt % MAX_DENTRY_PER_BLOCK == 1)
   inode->data_block[inode->block_used++] = wztfs_alloc_data();
return inode->dir_cnt;
```

以上代码先用头插法完成 dentry 的插入，之后将 inode 的目录项个数 dir_cnt 增加 1，再将 size 增加一个目录项。之后使用 dir_cnt 和每个数据块能够存放的目录项个数比较，当 dir_cnt 对每个数据块最多能存放的目录项个数取余恰好为 1 时，说明在插入这一个目录项前，已经恰好存满，新插入的这个目录项需要申请一块新的数据块用于存放。于是调用分配数据块的函数 wztfs_alloc_data()申请一个数据块并登记到 data_block 数组中。

inode->data_block[inode->block_used++] = wztfs_alloc_data()这一行代码能够同时完成新数据块的登记和已使用数据块的增加。已使用数据块数恰好是记录下一块数据块要使用的 data_blocl 数组的下标。

3. wztfs_alloc_data

这一函数的作用是根据数据块位图，选取一个空闲的数据块，将位图置 1，返回数据块号给上层，完成数据块的分配。外层调用该函数后，需将返回值设置为数据块号。

该函数通过对数据块位图的遍历完成。如图 18 所示。

```c
int wztfs_alloc_data() {
    int byte_cursor       = 0;
    int bit_cursor        = 0;
    int dno_cursor        = 0;
    int is_find_free_data = 0;

    for (byte_cursor = 0; byte_cursor < BLOCKS_SIZE(super.map_data_blks); byte_cursor++) {
        // 再在该字节中遍历8个bit寻找空闲的inode位图
        for (bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {
            if((super.map_data[byte_cursor] & (0x1 << bit_cursor)) == 0) {

                super.map_data[byte_cursor] |= (0x1 << bit_cursor);
                is_find_free_data = 1;
                break;
            }
            dno_cursor++;
        }
        if (is_find_free_data) {
            break;
        }
    }

    if (!is_find_free_data || dno_cursor >= super.max_data) {
        return -ERROR_NOSPACE;
    }

    return dno_cursor;
} // definitely no problem
```

图 18 分配数据块的核心代码

4. newfs_alloc_inode

这一函数的作用是为一个已有的 dentry 分配索引节点。在上层创建文件或目录时，步骤是先调用 new_dentry()创建新目录项，再调用 alloc_dentry()将 dentry 插入一个父目录的 inode 下，最后调用 alloc_inode 为该目录项分配索引节点，完成一个新文件或子目录的创建。在这一个函数内，需要做的工作包括 inode 位图的管理，inode 的申请，inode 和 dentry 相互连接的建立。

首先是通过 inode 位图找到可用的 inode，确定 inode 号 ino 并创建一个内存中的 inode。Inode 位图管理的核心代码如图 19 所示。

```c
// 在索引节点位图上查找空闲的索引节点
for (byte_cursor = 0; byte_cursor < BLOCKS_SIZE(1); byte_cursor++)
{
    for (bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {
        if((super.map_inode[byte_cursor] & (0x1 << bit_cursor)) == 0) {
                                                /* 当前ino_cursor位置空闲 */
            super.map_inode[byte_cursor] |= (0x1 << bit_cursor);
            is_find_free_entry = TRUE;
            break;
        }
        ino_cursor++;
    }
    if (is_find_free_entry) {
        break;
    }
}
```

图 19 在 inode 位图中寻找空闲 inode 并分配

这段代码遍历索引节点位图，当第一次找到可用的（值为 0）则将该值置为 1，并退出循环。此时的 ino_cursor 保存着的就是将要分配出去的 inode 号。

之后根据这个 inode 号，创建内存中的 inode 对象并将其与 dentry 建立对应关系。核心代码如图 20 所示。

```
// 为目录项分配inode节点并建立他们之间的连接
inode = (struct newfs_inode*)malloc(sizeof(struct newfs_inode));
inode->ino  = ino_cursor;
inode->size = 0;
inode->dir_cnt = 0;
inode->dentrys = NULL;
inode->block_used = 0;  //2024/12/7 23:50, 增加了新的字段，同步更新

/* dentry指向inode */
dentry->inode = inode;
dentry->ino   = inode->ino;
/* inode指回dentry */
inode->dentry = dentry;
```

图 20 创建 inode 并和 dentry 建立对应关系

这段代码中使用 malloc 在内存中创建新的 inode，设置 ino 为 ino_cursor，由于是新创建的文件，初始的 size、dir_cnt 等字段均设置为初始值 0 或者 NULL。最后将 inode 的信息同步到 dentry 中，完成 dentry 中 ino、inode 字段的设置。

由于文件创建后，可能马上需要写入操作。因此，如果是文件类型，应该在 inode 创建后为其分配内存中的数据块，用于可能的写数据。这需要先对 inode 的文件类型进行判断，对于文件类型，一次性申请最大所需的连续内存空间，以备写入数据使用。

```
282     /* 如果inode指向文件类型，则需要分配数据指针。如果是目录则不需要，目录项已存在dentrys中*/
283     // inode是内存里面的索引节点，分配指向内存中数据块的指针
284
285     // 应该改一下，改成直接分配一个大的连续数据块，使得数据块是连续的。
286     if (INODE_IS_REG(inode)) {
287         uint8_t *large_block = (uint8_t *)malloc(DATA_PER_FILE * LOGIC_BLOCK_SIZE);
288
289         for(int i = 0; i < DATA_PER_FILE; i++){
290             // inode->data_ptr[i] = (uint8_t *)malloc(LOGIC_BLOCK_SIZE);
291             inode->data_ptr[i] = large_block + i * LOGIC_BLOCK_SIZE;
292         }
293
294     }
295
296     return inode;
297 }
```

图 21 为文件分配内存数据空间

值得注意的是，在分配的时候，一次性分配连续的 12KB，再将指针依次间隔 1KB 设置。设置多个指针而不是只使用一个 data 指针，是为了方便写回磁盘时写回对应数据块。分配的数据在内存中连续而不是每一个指针 data_ptr[i]分别 malloc 申请一块内存，是为了在上一层面向用户的读写函数中调用接口方便。如果采取每一个数据块对应一个指针，分别申请内存空间的话，在上层的读写中，需要复杂的判断来找到对应数据的位置；如果是所有数据块连续分配，就可以对这个内存数据空间直接进行一次 memcpy 来完成读写。

5. newfs_sync_inode

这一个函数的声明是 int newfs_sync_inode(struct newfs_inode * inode)，它的作用是将指定的 inode 以及其下属所有 inode 以及数据全部写回磁盘。在磁盘卸载时，需要通过该函数将根节点及所有所属数据写回磁盘。在磁盘初次挂载时，初次创建的根节点也会通过该函数写回一次。

该函数的第一步是制作写回磁盘的 inode 结构，并将当前 inode 写回磁盘。

```
    struct newfs_inode_d  inode_d;   // 申请一个写回磁盘的inode对象（不是指针），把inode先
    struct newfs_dentry*  dentry_cursor;
    struct newfs_dentry_d dentry_d;
    /* inode_d 6个字段*/
    int ino             = inode->ino;
    inode_d.ino         = ino;
    inode_d.size        = inode->size;
    inode_d.type       = inode->dentry->type;
    inode_d.dir_cnt     = inode->dir_cnt;
    inode_d.block_used  = inode->block_used;

    int blk_cnt = 0;
    for(blk_cnt = 0; blk_cnt < DATA_PER_FILE; blk_cnt++) {
        inode_d.data_block[blk_cnt] = inode->data_block[blk_cnt]; // 赋值数据块索引
    }
    int offset, offset_limit;  //写回 dentry 使用

    // inode_d 刷回磁盘
    if (newfs_driver_write(INODE_OFFSET(ino), (uint8_t *)&inode_d,
                           sizeof(struct newfs_inode_d)) != ERROR_NONE) {
        NFS_DBG("[%s] io error\n", __func__);
        return -ERROR_IO;
    }
```

图 22 将当前 inode 写回磁盘

当前 inode 选出部分字段组成 inode_d 结构，通过封装后的磁盘 write 写回磁盘中。

第二步骤是根据 inode 是文件或目录类型，采取不同的操作。如果是文件，则需要将当前 inode 所对应的数据块写回磁盘。该文件已经使用的数据块由 block_used 记录，可以直接根据这个值写回相应数量的数据块。

```
} else if (INODE_IS_REG(inode)){  /* 如果是文件 */
    // 如果全部写回去，可能并不恰当。因为没有校验。原本判断条件for(blk_cnt = 0; blk_cnt
    printf("%s 被写回磁盘",inode->dentry->name);
    // int used_blocks = (inode->size + LOGIC_BLOCK_SIZE - 1) / LOGIC_BLOCK_S
    int used_blocks = inode->block_used; //2024/12/7 23:52 加了新的字段，已经使用
    for(blk_cnt = 0; blk_cnt < used_blocks; blk_cnt++){
        if (newfs_driver_write(DATA_OFFSET(inode->data_block[blk_cnt]),
                inode->data_ptr[blk_cnt], LOGIC_BLOCK_SIZE) != ERROR_NONE) {
            NFS_DBG("[%s] io error\n", __func__);
            return -ERROR_IO;
        }
    }
}
return ERROR_NONE;
```

图 23 将使用了的数据块写回

如图 23 所示，通过 data_block[i]可以确定一个数据块号，进而通过 DATA_OFFSET 宏函数确定磁盘中的偏移量；通过 data_ptr[i]可以确定内存中对应的数据块进行写入。

如果是目录类型，则当前 inode 可以通过链表 dentrys 来管理所拥有的子目录项，需要将子目录项依次写回磁盘，对子目录项对应的 inode 依次递归调用该函数，将它们进行写回。核心代码如下：

```
while(dentry_cursor != NULL && blk_cnt < inode->block_used) {
    offset = DATA_OFFSET(inode->data_block[blk_cnt]); // dentry 从 inode 分配的首个数据块开始存
    offset_limit = DATA_OFFSET(inode->data_block[blk_cnt] + 1);
```

```
        while ((dentry_cursor != NULL) && (offset + sizeof(struct newfs_dentry_d) < offset_limit))
        {
            memcpy(dentry_d.name, dentry_cursor->name, MAX_NAME_LEN);
            dentry_d.type = dentry_cursor->type;
            dentry_d.ino = dentry_cursor->ino;
            if (newfs_driver_write(offset, (uint8_t *)&dentry_d,
                                   sizeof(struct newfs_dentry_d)) != ERROR_NONE) {
                NFS_DBG("[%s] io error\n", __func__);
                return -ERROR_IO;
            }
            if (dentry_cursor->inode != NULL) {
                newfs_sync_inode(dentry_cursor->inode); /* 完成子目录写回 */
            }
            dentry_cursor = dentry_cursor->brother; // 写下一个 子目录 ，内存中下一个 dentry，实现内存中的遍历
            offset += sizeof(struct newfs_dentry_d); // 磁盘上的偏移
        }
        blk_cnt++; // 访问下一个指向的数据块
    }
```

这段代码是一个两层 while 循环，外层是用于遍历磁盘块和子目录项，内层是在同一个从磁盘块中写回子目录项。当同一个数据块写满目录项后，将结束内层循环，访问下一个数据块，再次进入内层循环写回子目录项。在内层循环中，先制作子目录项的磁盘中结构，并将它们写回磁盘；最后递归调用该函数，将子目录项对应的索引节点也写回。

6. newfs_read_inode

这一函数的作用是读入一个索引节点以及它对应的数据块。对于目录，不会递归读取全部的数据，只会读取当前索引节点指向的数据。

函数声明是 struct newfs_inode* newfs_read_inode(struct newfs_dentry * dentry, int ino) ，读取指定 dentry 的 inode，读入参数 dentry 和 ino 是对应的，调用者负责确定对应关系，在函数内不会进行检查。

首先是将 inode 本身读进内存。有 ino 即可确定 inode 在磁盘中的位置，可以直接根据 ino 读入 inode_d 结构，并制作出内存结构来。这一部分关键代码如图 24 所示。

```
if (newfs_driver_read(INODE_OFFSET(ino), (uint8_t *)&inode_d,
                      sizeof(struct newfs_inode_d)) != ERROR_NONE) {
    NFS_DBG("[%s] io error\n", __func__);
    return NULL;
}

inode->dir_cnt = 0; // inode是文件，目录项数量为0，是目录，这个值才有意义，到后面会重新赋值
inode->ino = inode_d.ino;
inode->size = inode_d.size;
inode->dentry = dentry;     /* 指回父级 dentry*/
inode->block_used = inode_d.block_used;
inode->dentrys = NULL;
// 数据块块号赋值
for(blk_cnt = 0; blk_cnt < DATA_PER_FILE; blk_cnt++)
    inode->data_block[blk_cnt] = inode_d.data_block[blk_cnt];
```

图 24 读入 inode

随后的数据读取，需要根据 inode 所对应文件类型的不同做出不同处理。对于文件类型，只需要读入该文件所拥有的数据块到内存。由于后续可能会有写入，应当先在内存中申请连续的最大数据空间，再设定好指针，并将数据读入。如图 25 所示。这一部分的逻辑和创建 inode 的时候类似。

```
// 改个方法，不是每个数据块分别申请内存空间，而是一次性给所有数据块申请内存空间，然后分别用指针去指
// 这样使得一个文件的数据在内存中连续。 外层读取就方便一点。
uint8_t *large_block = (uint8_t *)malloc(DATA_PER_FILE * LOGIC_BLOCK_SIZE);
for (int i = 0; i < DATA_PER_FILE; i++) {
    inode->data_ptr[i] = large_block + i * LOGIC_BLOCK_SIZE;
    if (newfs_driver_read(DATA_OFFSET(inode->data_block[i]), inode->data_ptr[i],
                          LOGIC_BLOCK_SIZE) != ERROR_NONE) {
        NFS_DBG("[%s] io error\n", __func__);
        return NULL;
    }
}
```

图 25 文件读入数据块

对于目录类型，不需要递归读取，而是只需要读取该 inode 所指向的数据块，从数据块中读出子目录项，将这些子目录项用头插法插入该 inode 即可，这可以调用 alloc_dentry 实现。如图 26 所示。

```
while((dir_cnt > 0) && (blk_cnt < DATA_PER_FILE)){
    // offset是磁盘中的偏移量
    offset = DATA_OFFSET(inode->data_block[blk_cnt]); // dentry 从 inode 分配的首个数据
    offset_limit = DATA_OFFSET(inode->data_block[blk_cnt] + 1);
    /* 写满一个 blk 时换到下一个 bno */
    // 下面这个循环在同一个数据块 data_block[blk_cnt]里面连续读取目录项
    while ((dir_cnt > 0) && (offset + sizeof(struct newfs_dentry_d) < offset_limit))
    {
        if (newfs_driver_read(offset, (uint8_t *)&dentry_d,
                              sizeof(struct newfs_dentry_d)) != ERROR_NONE) {
            NFS_DBG("[%s] io error\n", __func__);
            return NULL;
        }

        sub_dentry = new_dentry(dentry_d.name, dentry_d.type);
        sub_dentry->parent = inode->dentry;
        sub_dentry->ino    = dentry_d.ino;
        newfs_alloc_dentry(inode, sub_dentry);

        offset += sizeof(struct newfs_dentry_d);
        dir_cnt--;

    }
    blk_cnt++; /* 访问下一个指向的数据块 */
}
```

图 26 目录类型读入数据块

读入 dentrys 的过程中，采用了和写回类似的两层 while 循环来完成。外层循环会负责切换到下一个数据块，内层 while 循环负责在同一个数据块里面连续读取 dentry，并在读取后制作出内存 dentry 结构，调用 alloc_dentry 函数插入当前 inode 中。

7. wztfs_drop_dentry

这一函数的作用是从内存中删去一个 dentry。函数声明为 int wztfs_drop_dentry(struct newfs_inode * inode , struct newfs_dentry * dentry)，需要读入一个 inode 和一个 dentry，dentry 一定是 inode 所拥有的子目录项。这个函数将会把 dentry 从 inode 的子目录项链表中移除。

该函数的核心代码如图 27 所示。通过链表操作删除一个 dentry，并将 dir_cnt 减去 1。值得注意的是，删除后的 dentry 并不会在这里调用 free 释放，而是在外层调用者释放。

```
int wztfs_drop_dentry(struct newfs_inode * inode , struct newfs_dentry * dentry){
    boolean is_find = FALSE;
    struct newfs_dentry* dentry_cursor = inode->dentrys;

    if (dentry_cursor == dentry) {
        inode->dentrys = dentry->brother;  // 如果上来第一个就是，直接可以删除掉
        is_find = TRUE;
    } else {
        while (dentry_cursor) {
            if (dentry_cursor->brother == dentry) {
                dentry_cursor->brother = dentry->brother;
                is_find = TRUE;
                break;
            }
            dentry_cursor = dentry_cursor->brother;
        }
    }
    if (!is_find) {
        return ERROR_NOTFOUND;
    }
    inode->dir_cnt--;
    return inode->dir_cnt;
}
```

图 27 drop_dentry 代码

8. wztfs_drop_inode

这一函数的作用是从内存中删除一个 inode。对于目录需要递归删除其下的所有子目录项以及对应 inode，同时要管理 inode 位图和数据块位图。

drop_inode 和 drop_dentry 这两个函数是配合使用。当需要删除文件时，会先调用 drop_inode 删去待删除文件对应的 inode，在调用 drop_dentry 删除带删除文件对应的目录项。在 drop_inode 内部，会涉及到当前 inode 是目录时，需要删除所有子目录项及对应 inode，也是按照先删除 inode 在删除 dentry 的顺序。

进入该函数首先要进行判断，不允许删除根节点。随后会进入类型判断，根据不同类型采取不同的删除操作。当前 inode 如果对应文件类型，则需要删除对应的数据，通过设置对应的位图为 0 来实现。之后删除当前 inode，通过设置对应 inode 位图为 0 来实现。

设置数据块位图为 0 的核心代码如下：

```
 for (int i = 0; i < inode->block_used; i++) {
            byte_cursor = inode->data_block[i] / UINT8_BITS;
            bit_cursor  = inode->data_block[i] % UINT8_BITS;
            super.map_data[byte_cursor] &= (uint8_t)(~(0x1 << bit_cursor));
}
```

设置 inode 位图为 0 的核心代码如下：

```
        for(byte_cursor = 0; byte_cursor < BLOCKS_SIZE(1); byte_cursor++) {
            for(bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {
                if (ino_cursor == inode->ino) {
                    super.map_inode[byte_cursor] &= (uint8_t)(~(0x1 << bit_cursor));
                    is_find = TRUE;
                    break;
                }
```

```
            ino_cursor++;
        }
        if(is_find) {
            break;
        }
    }
```

之后，会通过 free 来释放内存中的数据空间和 inode。核心代码如下：

```
    if (inode->data_ptr[0])
        free (inode->data_ptr[0]);
    free(inode);
```

以上代码完成了文件类型的 inode 删除工作。对于目录类型，同样需要用相同的方法完成数据块位图以及 inode 位图的管理。此外，还需要对每一个子目录项进行删除，包括删除索引节点，删除目录项。这段核心代码如下：

```
    dentry_cursor = inode->dentrys;
    while (dentry_cursor) {
        inode_cursor = dentry_cursor->inode;
        wztfs_drop_inode(inode_cursor);
        wztfs_drop_dentry(inode, dentry_cursor);
        dentry_to_free = dentry_cursor;
        dentry_cursor = dentry_cursor->brother;
        free(dentry_to_free);
    }
```

通过以上代码就可以完成 drop_inode 函数的设计。

9. newfs_get_dentry

这一函数的声明为 struct newfs_dentry* newfs_get_dentry(struct newfs_inode * inode, int dir)，作用是获取指定 inode 的 dentrys 中的第 dir 个 dentry。实现上十分简单，只需要对 inode 的 dentrys 链表进行遍历，直至第 dir 个，返回即可。核心代码如图 28 所示。

```c
struct newfs_dentry* newfs_get_dentry(struct newfs_inode * inode, int dir) {
    struct newfs_dentry* dentry_cursor = inode->dentrys;
    int    cnt = 0;
    while (dentry_cursor)
    {
        if (dir == cnt) {
            return dentry_cursor;
        }
        cnt++;
        dentry_cursor = dentry_cursor->brother;
    }
    return NULL;
} //definitely no problem
```

图 28 获取指定序数的 dentry

10. newfs_lookup

lookup 函数用于根据目录查找指定的 dentry，在创建文件或目录、读写文件、显示文件时都起到重要作用。lookup 是由多个工具函数共同构成，lookup 函数内部还附带实现了 newfs_calc_lvl，用于计算路径层级，即查找的深度。

该函数的查找路径所对应的目录项，若存在则返回对应文件的目录项，若不存在则返回

其最近的外层目录项。进行文件的创建时，由于不存在指定的目录项，将返回父目录项，用于创建新的文件。进行文件的读写等操作时，将找到指定文件并返回。

newfs_calc_lvl 函数用于计算给定路径的层级。如图 29 所示。

```
int newfs_calc_lvl(const char * path) {
    // char* path_cpy = (char *)malloc(strlen(path));
    // strcpy(path_cpy, path);
    char* str = path;
    int   lvl = 0;
    if (strcmp(path, "/") == 0) {
        return lvl;
    }
    while (*str != NULL) {
        if (*str == '/') {
            lvl++;
        }
        str++;
    }
    return lvl;
} // definitely no problem
```

图 29 计算路径层级代码

calc_lvl 计算路径层级的核心原理就是统计’/’字符的数量，特别地，根目录层级为 0。

否则将逐层查找目录项，用 memcmp(dentry_cursor->name, fname, strlen(fname)) == 0)将目录项和当前层文件名对比，找到当前层目标文件后，从 dentry 索引到 inode，再从 inode 索引到数据块，继续查找下一层文件名的目录项。

查找到后，将会置 is_find 为 True，返回给上层判断使用。

事实上，lookup 函数是基于 inode 和 dentry 在内存中的关系的，和磁盘布局没有关系。Simplefs 中设计的 lookup，可以在本设计中直接使用。

11. newfs_mount

mount 函数用于挂载磁盘块。挂载磁盘块需要使用此前的工具函数，经过较多的流程。通过流程图来表示。

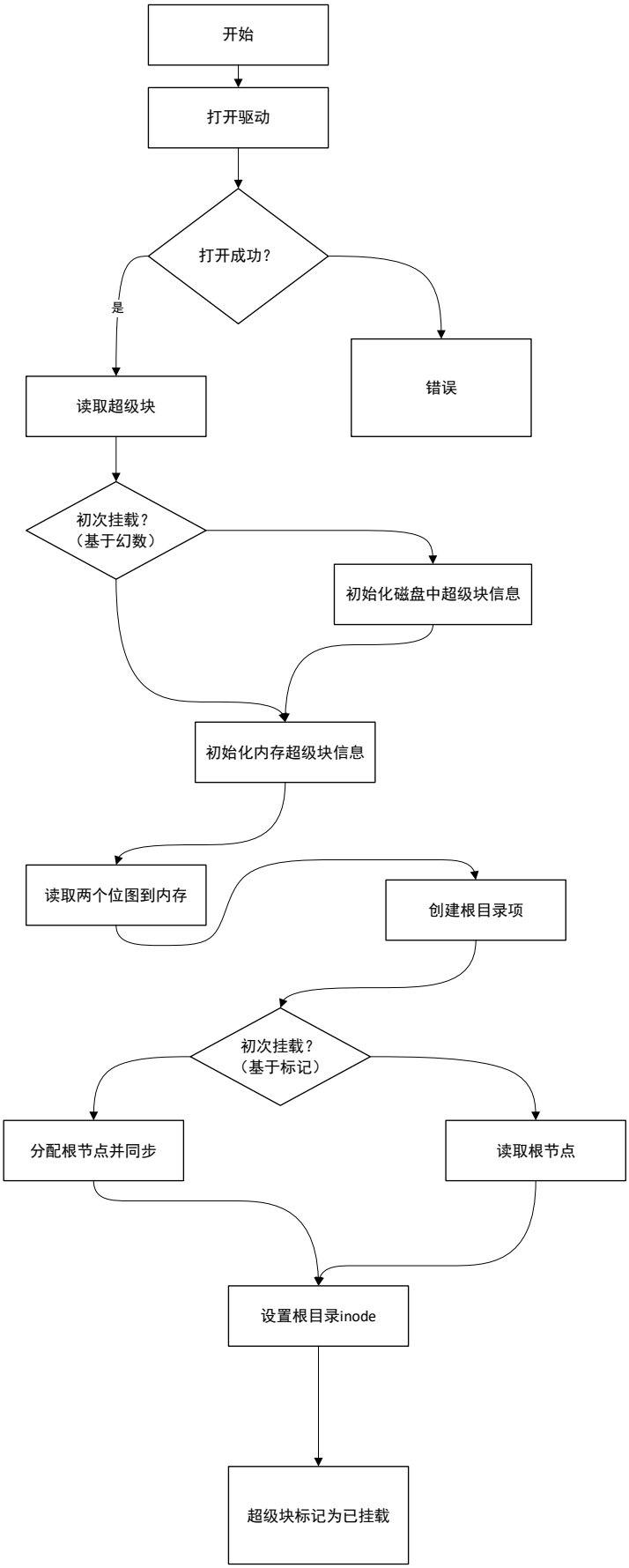
开始
打开驱动
打开成功?
是
错误
读取超级块
初次挂载?
（基于幻数）
初始化磁盘中超级块信息
初始化内存超级块信息
读取两个位图到内存
创建根目录项
初次挂载?
（基于标记）
分配根节点并同步
读取根节点
设置根目录inode
超级块标记为已挂载

图 30 挂载流程图

挂载函数的核心功能就是通过幻数判断是不是初次挂载，完成相关初始化，并创建根目录项。挂在函数会直接在偏移为 0 的磁盘块中读取超级块，文件系统会先预设一组值，如果是初次挂载，将会把这组值赋值到磁盘超级块结构中。随后，将磁盘超级块的值赋给内存超级块，建立起内存超级块结构。再完成 inode 位图、数据块位图的读入。最后创建根目录，由内存超级块持有根目录的指针，以便在任何时候通过根目录查找文件。

12. newfs_umount

这一个函数的功能是卸载磁盘块，卸载时，要将根节点调用 sync_inode 函数，将根节点及其所有数据递归写回磁盘，这一部分包括了索引节点、数据块两部分的写回。随后将超级块、索引节点位图、数据块位图也写回磁盘，就完成了五个部分写回磁盘，完成卸载。核心代码如下所示：

```
newfs_sync_inode(super.root_dentry->inode);       /* 从根节点向下递归将节点写回 */
 /*代码省略，该处将内存超级块赋值到磁盘超级块*/
 if (newfs_driver_write(SUPER_OFFSET, (uint8_t *)&super_d,
                    sizeof(struct newfs_super_d)) != ERROR_NONE) {
     return -ERROR_IO;
 }
 if (newfs_driver_write(super_d.map_inode_offset, (uint8_t *)(super.map_inode),
                        BLOCKS_SIZE(super_d.map_inode_blks)) != ERROR_NONE) {
     return -ERROR_IO;
 }
 if (newfs_driver_write(super_d.map_data_offset, (uint8_t *)(super.map_data),
                        BLOCKS_SIZE(super_d.map_data_blks)) != ERROR_NONE) {
     return -ERROR_IO;
 }
 free(super.map_inode);
 free(super.map_data);
 ddriver_close(DRIVER_FD());
```

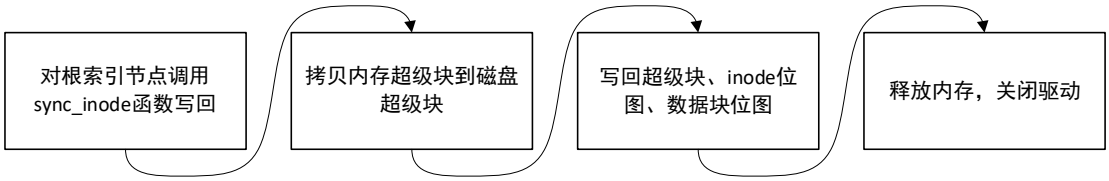

图 31 卸载流程图

(4)fuse 钩子函数的实现

这一部分实现的是面向用户命令的函数，即图 3 中红色标注的部分。

1. newfs_init

这一函数实现挂载，直接调用工具函数 newfs_mount 即可完成，同时包装了一些报错信息。

```c
void* newfs_init(struct fuse_conn_info * conn_info) {
    /* TODO: 在这里进行挂载 */

    /* 下面是一个控制设备的示例 */
    // super.fd = ddriver_open(newfs_options.device);

    if (newfs_mount(newfs_options) != ERROR_NONE) {
        NFS_DBG("[%s] mount error\n", __func__);
        fuse_exit(fuse_get_context()->fuse);
        return NULL;
    }
    return NULL;


}
```

图 32 init 函数

2. newfs_destroy

这一函数实现文件系统卸载，直接调用 unmount 函数，并包装一些报错信息。

```c
void newfs_destroy(void* p) {
    /* TODO: 在这里进行卸载 */

    if (newfs_umount() != ERROR_NONE) {
        NFS_DBG("[%s] unmount error\n", __func__);
        fuse_exit(fuse_get_context()->fuse);

    }

    return;
}
```

图 33 destroy 函数

3. newfs_mkdir

这一函数用于创建目录。函数声明为 int newfs_mkdir(const char* path, mode_t mode)。流程是先用 lookup 找到父目录项，为待创建的目录创建新的目录项，和父目录项及其 inode 建立联系，最后为其分配 inode，就完成了新目录的创建。

```c
int newfs_mkdir(const char* path, mode_t mode) {
    /* TODO: 解析路径，创建目录 */
    (void)mode;
    boolean is_find, is_root;
    char* fname;
    struct newfs_dentry* last_dentry = newfs_lookup(path, &is_find, &is_root);
    struct newfs_dentry* dentry;
    struct newfs_inode*  inode;

    if (is_find) {
        return -ERROR_EXISTS;
    }

    if (INODE_IS_REG(last_dentry->inode)) {
        return -ERROR_UNSUPPORTED;
    }

    fname  = newfs_get_fname(path);
    dentry = new_dentry(fname, WZT_DIR);
    dentry->parent = last_dentry;
    inode  = newfs_alloc_inode(dentry);
    newfs_alloc_dentry(last_dentry->inode, dentry);


    return ERROR_NONE;  // return 0;
}
```

图 34 mkdir 函数

如图 34 所示，调用 lookup 函数获取到父目录项 last_dentry。由于传入的 path 是带创建的目录的路径，该目录暂时还不存在，所以 lookup 将返回父目录路径。get_fname 函数将从 path 中取出将要创建的文件或目录的名称，并用该名称创建一个类型为目录的新目录项。随后通过 alloc_inode 为新目录项申请索引节点，当这一步完成并没有报错，就可以将目录项插入到父目录的 inode 中。

3. newfs_mknod

这一函数与 mkdir 基本类似。

```c
int newfs_mknod(const char* path, mode_t mode, dev_t dev) {
    /* TODO: 解析路径，并创建相应的文件 */
    boolean is_find, is_root;

    struct newfs_dentry* last_dentry = newfs_lookup(path, &is_find, &is_root);
    struct newfs_dentry* dentry;
    struct newfs_inode* inode;
    char* fname;

    if (is_find == TRUE) {
        return -ERROR_EXISTS;
    }

    fname = newfs_get_fname(path);

    if (S_ISREG(mode)) {
        dentry = new_dentry(fname, WZT_FILE);
    }
    else if (S_ISDIR(mode)) {
        dentry = new_dentry(fname, WZT_DIR);
    }
    dentry->parent = last_dentry;
    inode = newfs_alloc_inode(dentry);
    newfs_alloc_dentry(last_dentry->inode, dentry);

    return ERROR_NONE;
```

图 35 mknod 函数

4. newfs_getattr

函数声明为 int newfs_getattr(const char* path, struct stat * newfs_stat)这一函数用于获取指定路径 path 的文件或目录属性，填充 stat 结构体返回给上层。先使用 lookup 函数获取目标文件或目录的目录项，从目录项中获取相应的信息填充至 stat 结构体中。

```c
boolean is_find, is_root;
struct newfs_dentry* dentry = newfs_lookup(path, &is_find, &is_root);
if (is_find == FALSE) {
    return -ERROR_NOTFOUND;
}

if (INODE_IS_DIR(dentry->inode)) {
    newfs_stat->st_mode = S_IFDIR | NEWFS_DEFAULT_PERM;
    newfs_stat->st_size = dentry->inode->dir_cnt * sizeof(struct newfs_dentry_d);
}
else if (INODE_IS_REG(dentry->inode)) {
    newfs_stat->st_mode = S_IFREG | NEWFS_DEFAULT_PERM;
    newfs_stat->st_size = dentry->inode->size;
}
```

图 36 填充 stat 结构体

```
newfs_stat->st_nlink = 1;
newfs_stat->st_uid   = getuid();
newfs_stat->st_gid   = getgid();
newfs_stat->st_atime   = time(NULL);
newfs_stat->st_mtime   = time(NULL);
newfs_stat->st_blksize = LOGIC_BLOCK_SIZE;  // 逻辑块大小
newfs_stat->st_blocks = DATA_PER_FILE;
if (is_root) {
    newfs_stat->st_size = super.sz_usage;
    newfs_stat->st_blocks = DISK_SIZE / LOGIC_BLOCK_SIZE;  // 磁盘块数/块大小 没有错
    newfs_stat->st_nlink  = 2;      /*根目录link数为2 */
}
return ERROR_NONE;
```

图 37 填充 stat 结构体

核心代码如图 36、图 37 所示。调用 lookup 获取目录项 dentry 后，会先用 is_find 判断查找是否成功，不成功将报错。进而使用判断 dentry 对应的是文件还是目录类型，根据不同类型完成 st_mode 字段的填充，在这里，st_mode 的权限位默认给 0777。并根据目录项索引到 inode，从 inode 中获取文件或目录的大小，填充 st_size。

随后，将向 stat 中填充一些元数据性质的信息，如连接数，在本实验系统中连接数为 1。随后还用到 is_root 进行判断，如果是根节点，会进行额外处理，因为根节点的连接数为 2，blocks 和 size 都改为整个磁盘的对应数据，如 blocks 改为磁盘总块数，在这里是 4096；st_size 改为超级块中记录的已使用空间。

5. newfs_readdir

该函数声明为 int newfs_readdir(const char * path, void * buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info * fi) ，作用是读取指定 path 的目录项所对应的目录下面的指定目录项。换言之，这一函数先找到 path 对应的目录项 dentry，再通过 dentry 找到 inode，再通过 inode 索引到数据块，读取指定位置的子目录项 sub_dentry。其中，filler 是 fuse 框架提供的函数指针，使用 filler 可以完成将目录项填充到 buf 的功能。最终的实现，我们只需考虑用 lookup 找到 dentry，找到 inode 以及其子目录项，调用 filler 即可。

```
boolean is_find, is_root;
int     cur_dir = offset;

struct newfs_dentry* dentry = newfs_lookup(path, &is_find, &is_root);
struct newfs_dentry* sub_dentry;

if (is_find) {
    struct newfs_inode* inode;
    inode = dentry->inode;
    sub_dentry = newfs_get_dentry(inode, cur_dir);    // 根据当前偏移找到对应dentry
    if (sub_dentry) {
        filler(buf, sub_dentry->name, NULL, ++offset);
    }
    return ERROR_NONE;
}
return -ERROR_NOTFOUND;
```

图 38 readdir 函数核心代码

6. newfs_utimens

这一函数用于更改时间，在创建文件中有作用。该函数直接返回 0，使得 touch 不报错。

7. newfs_write

该函数的声明为 int newfs_write(const char* path, const char* buf, size_t size, off_t offset,struct fuse_file_info* fi)，作用是将 buf 内的数据写入到路径为 path 的文件。

根据“文件的表示”一节，该文件系统中操作文件总是通过找到目录项来完成。先通过 lookup 找到目标文件的目录项 dentry 与 inode。在调用了 lookup 找到 dentry 的时候，lookup 内部已经通过调用 read_inode，将目标文件的 inode 读入内存，并且创建了最大 12kB 的连续内存空间，将目标文件的数据块读入内存中。这个连续的数据块可以通过 inode->data_ptr[0]来访问。因此，除开一些错误检查外，只需要通过一句 memcpy 语句，就可以将 buf 写入目标文件的数据中。在目标文件写回的时候，数据会更新到磁盘。

再写入的同时，还要做好文件大小的管理。write 写入数据只会使得文件大小增加而不会减少。写入后，根据写入的终点偏移量和原本文件大小进行比较，判断写入是否导致了文件总大小增加，并更新文件大小 size 字段。核心代码如下所示：

```
memcpy(inode->data_ptr[0] + offset, buf, size);
inode->size = offset + size > inode->size ? offset + size : inode->size;
```

此外，还需要完成文件数据块的分配。由于本设计采用按需分配数据块，在写文件的时候检查是否需要新的数据块。使用更新后的文件大小 inode->size 计算实际所需的数据块数，与已经分配到文件的磁盘块数 inode->block_used 进行比较，判断是否需要新分配数据块。在需要时，只需调用 alloc_data 的函数即可。这一部分核心代码如下：

```
int required_block = (inode->size + LOGIC_BLOCK_SIZE - 1) / LOGIC_BLOCK_SIZE;
    if (required_block > inode->block_used) {
        inode->data_block[inode->block_used++] = wztfs_alloc_data();
    }
```

8. newfs_read

读入文件与写文件基本类似，但不需要文件大小的更新和磁盘块的分配操作。逻辑更加简单，只需要用 lookup 找到目录项，除开错误检查之外，只需调用 memcpy 把数据读入 buf 即可。此处的数据也是直接使用 data_ptr[0]来访问。代码如图 39 所示。

```
int newfs_read(const char* path, char* buf, size_t size, off_t offset,
               struct fuse_file_info* fi) {
    /* 选做 */
    boolean is_find, is_root;
    struct newfs_dentry* dentry = newfs_lookup(path, &is_find, &is_root);
    struct newfs_inode*  inode;

    if (is_find == FALSE) {
        return -ERROR_NOTFOUND;
    }
    // 找到对应文件的目录项后，判断文件类型
    inode = dentry->inode;
    if (INODE_IS_DIR(inode)) {
        return -ERROR_ISDIR;
    }
    // 文件大小判断，若文件大小比offset小报错
    if (inode->size < offset) {
        return -ERROR_SEEK;
    }

    // 由于修改了文件读入的方式，文件数据读入内存后一定是连续的。因此data_ptr[0]就是内存中数据块的起点
    memcpy(buf, inode->data_ptr[0] + offset, size);
    return size;
}
```

图 39 read 核心代码

值得注意的是，此处的逻辑简单，直接使用 data_ptr[0]可以访问数据，是因为同一个文件的多个数据块在内存中是连续存储的。每一个文件读入内存，都会直接在内存中申请最大长度的数据空间，连续存储读入的数据块，这样的设计方便了面向用户接口的实现。

9. newfs_unlink

该函数的声明为 int newfs_unlink(const char* path)，作用是删除指定路径的文件。在删除文件时，会用到 drop_inode 和 drop_dentry 函数。对于指定路径的文件，先使用 lookup 找到目录项 dentry，进而找到 inode，先删除 inode，再删除 dentry 即可。

```
int newfs_unlink(const char* path) {
    /* 选做 */
    boolean is_find, is_root;
    struct newfs_dentry* dentry = newfs_lookup(path, &is_find, &is_root);
    struct newfs_inode*  inode;

    if (is_find == FALSE) {
        return -ERROR_NOTFOUND;
    }

    inode = dentry->inode;

    wztfs_drop_inode(inode);
    wztfs_drop_dentry(dentry->parent->inode, dentry);

    return ERROR_NONE;
}
```

图 40 unlink 核心代码

创建一个文件的时候，是先有目录项 dentry，再有 inode；删除文件的时候，先删除 inode，再删除目录项。

10. newfs_rmdir

该函数用于删除指定路径的目录，直接调用 unlink 函数即可。

```
int newfs_rmdir(const char* path) {
    return newfs_unlink(path);
}
```

11. newfs_rename

该函数用于修改一个指定路径，读入源路径和目标路径，函数运行后，源路径的文件会变成目标路径。用于实现用户的 mv 命令。

重命名路径的原理，是先调用 mknod 在新路径创建一个目标文件，用新的目录项取代原本的目录项，而不替换索引节点和数据。只需要使用新的 to_dentry 指向原本的 inode，并将原本的 dentry 删除。由于 mknod 创建新文件的时候，也会分配新的 inode，要将新的 inode 也删掉。核心代码如图 41 所示。

```
if (strcmp(from, to) == 0) {
    return ERROR_NONE;
}
from_inode = from_dentry->inode;
if (INODE_IS_DIR(from_inode)) {
    mode = S_IFDIR;
} else if(INODE_IS_REG(from_inode)) {
    mode = S_IFREG;
}

ret = newfs_mknod(to, mode, NULL);

if (ret != ERROR_NONE) {                        /* 保证目的文件不存在 */
    return ret;
}
to_dentry = newfs_lookup(to, &is_find, &is_root);
wztfs_drop_inode(to_dentry->inode);
to_dentry->ino = from_inode->ino;
to_dentry->inode = from_inode;
to_dentry->inode->dentry = to_dentry;


wztfs_drop_dentry(from_dentry->parent->inode, from_dentry);
```

图 41 rename 的核心代码

核心代码首先判断 mv 传入的两个路径是否完全一致，如果完全一致就不需要做任何处理。如果不一致，则会调用 mknod 创建目标路径的文件，并删去新创建文件的索引节点，因为这一索引节点并无意义；用新的目录项与原有的索引节点建立联系，再删去原有的目录项。就完成了文件路径的更改。

12. 其他函数

除此之外还包括 fuse 框架的 open、opendir、truncate、access 钩子需要实现，这些函数较为简单，可以直接使用 simplefs 中的实现。其中，open 和 opendir 直接返回 0，truncate 用于改变文件大小，access 用于判断文件是否可以访问。

```
int newfs_truncate(const char* path, off_t offset) {
    /* 选做 */
    boolean is_find, is_root;
    struct newfs_dentry* dentry = newfs_lookup(path, &is_find, &is_root);
    struct newfs_inode*  inode;


    if (is_find == FALSE) {
        return -ERROR_NOTFOUND;
    }
    inode = dentry->inode;
    if (INODE_IS_DIR(inode)) {
        return -ERROR_ISDIR;
    }
    inode->size = offset;

    return ERROR_NONE;
}
```

图 42 truncate 代码

```c
int newfs_access(const char* path, int type) {
    /* 选做：解析路径，判断是否存在 */
    boolean is_find, is_root;
    boolean is_access_ok = FALSE;
    struct newfs_dentry* dentry = newfs_lookup(path, &is_find, &is_root);
    struct newfs_inode*  inode;
    switch (type)
    {
    case R_OK:
        is_access_ok = TRUE;
        break;
    case F_OK:
        if (is_find) {
            is_access_ok = TRUE;
        }
        break;
    case W_OK:
        is_access_ok = TRUE;
        break;
    case X_OK:
        is_access_ok = TRUE;
        break;

    default:
        break;
    }
    return is_access_ok ? ERROR_NONE : -ERROR_ACCESS;
```

图 43 access 代码

3、 实验特色

实验中你认为自己实现的比较有特色的部分，包括设计思路、实现方法和预期效果。

1. 索引节点在磁盘中存放的设计

指导书提供的思路一个磁盘块只存放一个索引节点，造成了大量的浪费。一个磁盘块只存放一个索引节点的优势是寻址方便，获得 ino，根据块大小进行偏移就可以找到指定的索引节点，但是存在大量浪费，这样文件系统能够维护的文件数量很少。

设计思路：为了使一个磁盘块存放多个索引节点又方便寻找索引节点，想到的思路就是**对齐**，原本有用的字段长度已经超过 64B，思路就是**凑成 1024 的因数**，凑满 128B。

实现方法：通过在磁盘 inode_d 结构体里面加了一些没有意义的填充单元，把 inode_d 凑满 128B，这样 inode_d 结构体能够和磁盘块对齐，一个磁盘块恰好存放 8 个 inode_d。寻址时，使用基址 inode_offset+ino*128 就可以找到 ino 对应的索引节点位置。

预期效果：文件系统能够容纳的文件数，在相同磁盘布局下，比指导书提供的方法增大了 8 倍。文件系统可以管理很大量的文件并且不牺牲大量数据空间。

2. 数据块在内存连续存放

在 ext2 文件系统中，一个索引节点指向多个数据块，很容易让人提出每个数据块分别申请一个内存空间来存放的方法，这样可以灵活申请内存空间。但是在用户读写的时候，面向用户的接口是使用一个 memcpy 来完成数据读写，如果每个数据块分别申请内存空间，分别用指针指向，那么读写的时候会有很大困难。

设计思路：在数据块读入内存时，还是像 simplefs 那样处理，在内存直接预分配最大的数据空间给文件使用，保证数据在内存中连续，在磁盘中数据块可以随意分布。这样读写的

时候，一个 memcpy 就完成，面向用户的接口和 simplefs 比不需要很多修改，几乎可以直接使用 simplefs 的实现。同时，每个数据块对应一个指针的数组仍保留，在写回的时候就比较方便。

实现方式：在创建 inode 和读入 inode 的时候都需要为 inode 中的数据指针分配空间，先一次性分配 12KB 空间，再依次把指针指向对应地方。在释放内存空间的时候，释放 data_ptr[0]即可。读写的时候，也只通过 data_ptr[0]访问数据。同时，有可以通过 data_ptr[i]把数据写回 data_block[i]块里。

预期效果：这样实现下，用户读写文件操作的实现简化了。

3. 分配数据块的函数

在将 dentry 插入 inode 的时候，会按需分配数据块；在写文件的时候，也会按需分配数据块。寻找数据块位图、分配数据块这个操作是相对独立且复用的，写成一个专门的函数，在需要的时候直接调用，就可以完成数据块的分配。

4. 位图查询逻辑的改进 S

实验指导书和示例代码提供的位图，是从头到尾遍历的方式。实际上，在删除 inode 和数据位图的时候，不需要用遍历的方式，因为索引节点号 ino，数据块位图号 bno 都是可以反应顺序的，例如已知要删去的数据块位图号 bno，那么 bno/8 就是位图中的字节偏移，bno%8 就是位图中那个字节里的位偏移，而不需要向示例代码一样用两层遍历来做。

```
// 管理inode位图。把inode对应的位图位置0
// for(byte_cursor = 0; byte_cursor < BLOCKS_SIZE(1); byte_cursor++) {
//     for(bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {
//         if (ino_cursor == inode->ino) {
//             super.map_inode[byte_cursor] &= (uint8_t)(~(0x1 << bit_cursor));
//             is_find = TRUE;
//             break;
//         }
//         ino_cursor++;
//     }
//     if(is_find) {
//         break;
//     }
// } // 两层循环的逻辑，可以简化为下面的逻辑
byte_cursor = inode->ino / UINT8_BITS;
bit_cursor  = inode->ino % UINT8_BITS;
super.map_inode[byte_cursor] &= (uint8_t)(~(0x1 << bit_cursor));
// 通过inode->block_used来判断有多少个数据块，对于inode->data_block[0]到inode->data_block[bl
for (int i = 0; i < inode->block_used; i++) {
    byte_cursor = inode->data_block[i] / UINT8_BITS;
    bit_cursor  = inode->data_block[i] % UINT8_BITS;
    super.map_data[byte_cursor] &= (uint8_t)(~(0x1 << bit_cursor));
}
```

图 44 位图置 0 逻辑的优化，注释代码为 simplefs 的实现方式

二、遇到的问题及解决方法

*列出实验过程中遇到的主要问题，包括技术难题、设计挑战等。对应每个问题，提供采取的**解决策略**，以及解决问题后的效果评估。*

技术难题：

索引节点在磁盘中存放的设计。如果索引节点要在同一个磁盘块中存放多个，但是最后

可能对不齐，在寻址上就有很大的难题。最终我偶然间想到有看过一些奇葩 bug，因为删了一个没有用到的变量，导致内存刚好没对齐，出了 bug。而本来那个变量出现在那里恰好让内存对齐了，所以就能跑通。

我从中获得灵感，可以往磁盘索引节点里面填一些填充变量，把 inode_d 填充到 128B，这样刚好能够对齐，在寻址上也比较简单。这样就可以实现在一个磁盘块存放 8 个索引节点。

设计挑战：

因为代码是从 simplefs 里面复制过来再改，有一个地方改错了一直没发现，remount 死活不成功。我最初判断 bug 是重新挂载能否看到文件和挂载后创建的文件/目录顺序有关，后来觉得 bug 是纯粹随机，最后调试了很久后定位到 bug 是而是创建第一、第二个文件或目录后，remount 读取不到；第三个和第四个创建后，可以读取第一个、第二个。

解决策略：为了进行 bug 调试，我先清空磁盘，然后每次挂载、卸载只做一次操作，先在纸上预测输出的磁盘内容会长什么样，再和输出的 dump 比较，哪一个操作后 dump 不符合预期，就是哪一次操作出了问题。

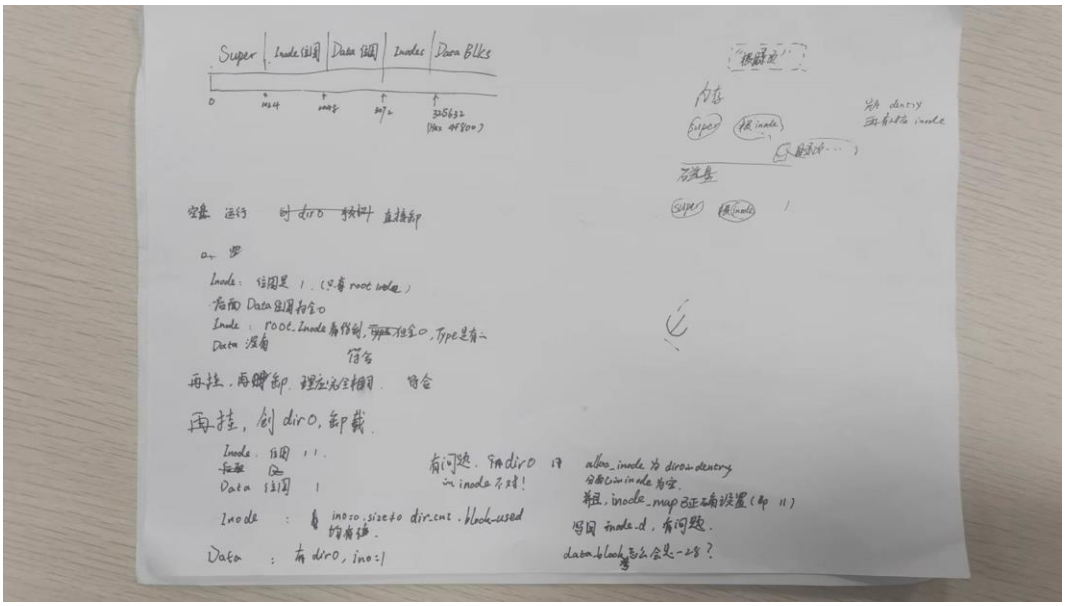

图 45　调试使用的纸

发现创建第一个文件后，输出的位图就不符合预期，inode 根本没有出现在输出中。后来开断点调试去看创建第一个文件的过程，发现一处错误检查写错了，导致 inode 号为 1 的 inode 没有被设置正确的值，但是位图已经占用了。修改后就跑通了。这个 bug 调试了十个小时左右。

三、实验收获和建议

实验中的收获、感受、问题、建议等。

经过这次实验，感觉在心态上成熟了很多，我花了十二个小时以上调试一个 bug，最终进行小于 12 个字符的代码修改，解决了这个 bug。经过这样漫长的历练，心性长进了很多，面对问题也更加平和。我学会了你得接受人的不足之处，直面自己的缺陷，我现在确信并且坦然接受我的智商在这里有点不够用，但不会为此有心情上的波动，我觉得人来人往，做自

己就好。面对技术问题，也不能轻易波动自己的情绪，心态要放平放宽，包容一切。例如不必唏嘘一个细小的错误造成大量的时间投入在上面纠错，也许原本就该花这么多时间。你总会因为某个原因赔上工作量，遇到任何一种 bug 都是你和它的缘分。

我的感受也许会感谢坚持不依靠别人力量做实验的自己。这次试验最让人印象深刻的是一天晚上看代码看到块 4 点，睡了三个小时就起来继续调试，最终解决了问题，让人不可思议。不能说从这段经历里面学到多少东西，可能没有成正比地学到很多技术上的收获，但是通常人都会感激自己拼搏过的时间。

在技术上我通过这次实验了解到了 EXT2 文件系统的原理，同时也了解到文件系统在对磁盘进行操作的同时，在内存中可以是另外一种结构来进行高效的组织，这让我对操作系统本身有了更深的认识。

建议上，我觉得可以给出一些 fuse 文件系统的钩子函数和用户命令之间的关系。做完实验只知道钩子函数按照 sfs 的示例做就可以了，其实不是很清楚到用户之间的关系。如果能够进一步的，从用户给出某个命令，文件系统是走过了哪些函数来完成了这个功能，在指导书里面拿示例的做一个介绍，做起来会轻松很多也会学到更多东西。

四、参考资料

实验过程中查找的信息和资料

工具函数和 fuse 接口、文件格式参考 simplefs 和指导书。此外纯粹自研，没有参考网络资料。