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：4

学生学号：220110426

学生姓名：杨明宇

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。

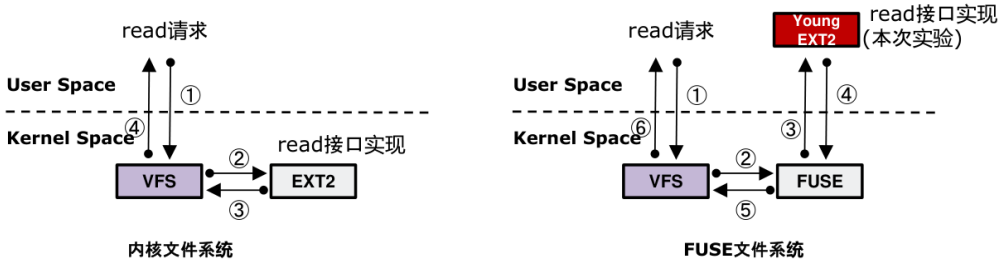
本实验基于 FUSE 架构，借鉴 EXT2 的 Linux 文件系统结构设计，实现了包含超级块、索引位图、数据块位图、索引节点、数据块等结构的文件系统，该系统支持创建文件目录(mkdir)、创建文件(touch)、展示文件(ls)、删除文件(rm)、写入文件(ls)、展示文件内容(cat)、移动文件(mv)、重命名文件(mv)、挂载卸载文件等操作，能够支持用户完成日常简易的文件内容操作。

关于 EXT2 文件系统设计：

EXT2 文件系统是 Linux 的核心文件系统，虽然如今 Linux 已经以 EXT4 作为主要文件系统版本，但是 EXT2 的文件系统架构设计思想仍然值得我们反复学习，它综合利用了索引、哈希表、链表等思想，将文件系统设计为了包含超级块、索引节点、位图等高效数据结构，它支持高效读写、查找文件、删除文件等，因此本次实验借鉴 EXT2 文件系统不仅能够加深我们对于 Linux 文件系统的认识，也能夯实我们其他各方面的知识。

关于 FUSE 架构设计：

FUSE(Filesystem in User SpacE)是一种可扩展、易于修改维护的架构设计，不同于一般的接口设计，一般来说，对于文件系统的操作会被封装到内核中，以保证文件系统数据的安全性，但是这种**宏内核**的架构设计使得它在修改维护时十分困难，同时如果用户只是想使用内核应用程序提供的几个功能，又不得不下载整个全部内核来实现期望功能，因此后来设计者们便设计了**微内核**架构，其主要思想就是将一些本应该属于内核的操作放到用户空间中实现，这样能够大大简便功能的实现以及空间的占用，而 FUSE 文件系统便类似于这种设计，通过将 EXT2 文件系统的基本操作放到用户态来实现，提高了代码的可维护性与可拓展性，但是这样做的同时带来的问题就是增加了用户态与内核态的切换开销，使得在高并发场景下，这种设计会有很大的性能问题，但是在中小系统中，这种模块化的设计思想非常实用。

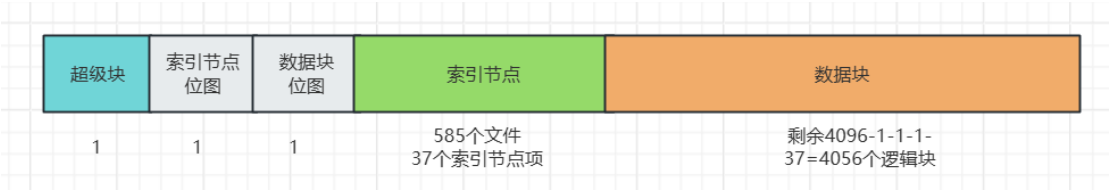


关于 DDriver 虚拟磁盘驱动：

关于这个没什么好说的，值得注意的就是它每次读取写入文件的大小为 512B，但是 EXT2 文件系统的逻辑块大小为 1024B，为两个 IO。

因此总体上看，我们要设计一个满足期望功能的逻辑块结构，同时实现基本的增删改查

等功能，逻辑块结构上，借鉴实验指导书的设计，磁盘容量为 4MB，逻辑块大小为 1024B，那么逻辑块数是 4096，采用直接索引，每个文件最多直接索引 6 个文件逻辑块填写文件数据，同时假设一个文件索引节点使用一个逻辑块存储，因为一个文件所需 7KB 的存储容量，那么 4MB 磁盘最多可存放 585 个文件数，一个索引节点可存储 16 个文件项，因此需要 585/16=37 个索引结点项，因此整体架构如下图：



而关于文件系统中基本的增删改查操作，借鉴 simplefs 中的函数实现即可，而 FUSE 中对于函数的调用通过钩子函数来实现：

```
struct newfs_super newfs_super;
/******************************************************************************
 * SECTION: FUSE操作定义
 *****************************************************************************/
static struct fuse_operations operations = {
    .init = newfs_init,       /* mount文件系统 */
    .destroy = newfs_destroy, /* umount文件系统 */
    .mkdir = newfs_mkdir,     /* 建目录，mkdir */
    .getattr = newfs_getattr, /* 获取文件属性，类似stat，必须完成 */
    .readdir = newfs_readdir, /* 填充dentrys */
    .mknod = newfs_mknod,     /* 创建文件，touch相关 */
    .write = newfs_write,           /* 写入文件 */
    .read = newfs_read,             /* 读文件 */
    .utimens = newfs_utimens, /* 修改时间，忽略，避免touch报错 */
    .truncate = newfs_truncate,         /* 改变文件大小 */
    .unlink = newfs_unlink,           /* 删除文件 */
    .rmdir = newfs_rmdir,           /* 删除目录，  rm -r */
    .rename = newfs_rename,           /* 重命名，mv */

    .open = newfs_open,
    .opendir = newfs_opendir,
    .access = newfs_access};
```

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）

数据结构：

文件类型数据结构：

```
typedef int         boolean;
typedef enum newfs_file_type {
    NEWFS_REG_FILE,
    NEWFS_DIR
} NEWFS_FILE_TYPE;
```

Simplefs 的文件结构中还有引用 SFS_SYM_LINK 引用类型文件结构，这里只定义简单的目录与文件两种文件类型

In_disk 与磁盘相关操作数据结构：

超级块数据结构：

```c
struct newfs_super_d {
    uint32_t magic_num;
    int sz_usage;

    /* 磁盘布局分区信息 */
    int super_blk_offset;           // 超级块于磁盘中的偏移, 0
    int super_blks;                 // 超级块于磁盘中的块数, 1

    int map_inode_offset;           // 索引节点位图于磁盘中的偏移 1
    int map_inode_blks;             // 索引节点位图于磁盘中的块数 1

    int map_data_offset;            // 数据块位图于磁盘中的偏移 2
    int map_data_blks;              // 数据块位图于磁盘中的块数 1

    int ino_offset;                 // 索引节点区于磁盘中的偏移 3
    int ino_blks;                   // 索引节点区于磁盘中的块数 37

    int data_offset;                // 数据块区于磁盘中的偏移 40
    int data_blks;                  // 数据块区于磁盘中的块数 4056

    /* 支持的限制 */
    int max_ino;                    // 最大支持inode数
    int max_data;                   // 最大支持数据块数

    /* 根目录索引 */
    int root_ino;                   // 根目录对应的inode
};
```

从上往下依次定义了超级块、索引节点位图、数据块位图、索引节点区、数据块区的偏移量以及逻辑块数信息

索引节点与目录项：

```c
struct newfs_inode_d {
    uint32_t ino;
    /* 文件的属性 */
    int                 size;               // 文件已占用空间
    int                 link;               // 链接数，默认为1
    NEWFS_FILE_TYPE     ftype;             // 文件类型（目录类型、普通文件类型）

    /* 数据块的索引 */
    int                 block_pointer[NEWFS_DATA_PER_FILE];   // 数据块块号（可固定分配）

    /* 其他字段 */
    int                 dir_cnt;            // 如果是目录类型文件，下面有几个目录项
};

struct newfs_dentry_d {
    /* 文件名称 */
    char     fname[MAX_NAME_LEN];

    /* inode编号 */
    uint32_t ino;

    /* TODO: Define yourself */
    /* 文件类型 */
    NEWFS_FILE_TYPE     ftype;              // 文件类型（目录类型、普通文件类型）
};
```

这两个结构体是索引节点与目录项的类型，主要借鉴 simplefs 定义方式，不同的是，这里的索引节点结构体中，存储了关于数据块的索引块号，能够方便查找。

In_mem 与内存相关操作数据结构：

超级块结构：

```
#ifndef _TYPES_H_
struct newfs_super {
    int      fd;
    /* TODO: Define yourself */
    int sz_disk;          // 磁盘大小
    int sz_usage;

    /* 逻辑块 */
    int sz_io;            // io大小
    int sz_blks;          // 逻辑块大小

    /* 磁盘布局分区信息 */
    int super_blk_offset;          // 超级块于磁盘中的偏移, 0
    int super_blks;             // 超级块于磁盘中的块数, 1

    int map_inode_offset;     // 索引节点位图于磁盘中的偏移 1
    int map_inode_blks;       // 索引节点位图于磁盘中的块数 1
    uint8_t* map_inode;

    int map_data_offset;     // 数据块位图于磁盘中的偏移 2
    int map_data_blks;       // 数据块位图于磁盘中的块数 1
    uint8_t* map_data;

    int ino_offset;     // 索引节点区于磁盘中的偏移 3
    int ino_blks;       // 索引节点区于磁盘中的块数 256

    int data_offset;     // 数据块区于磁盘中的偏移 131
    int data_blks;       // 数据块区于磁盘中的块数 3965

    /* 支持的限制 */
    int max_ino;            // 最大支持inode数
    int max_data;           // 最大支持数据块数

    /* 根目录索引 */
    int root_ino;           // 根目录对应的inode

    /* 其他信息 */
    boolean            is_mounted;
    struct newfs_dentry* root_dentry;     // 根目录
};
```

除了包含与磁盘操作定义的数据结构基本字段信息，这里又额外定义了索引节点位图偏移量信息以及数据块位图偏移量信息，同时记录了根目录对应索引

索引节点与目录项：

```
struct newfs_inode {
    uint32_t ino;
    /* TODO: Define yourself */
    /* 文件的属性 */
    int                 size;               // 文件已占用空间
    int                 link;               // 链接数，默认为1
    NEWFS_FILE_TYPE     ftype;             // 文件类型（目录类型、普通文件类型）

    /* 数据块的索引 */
    int                 block_pointer[NEWFS_DATA_PER_FILE];   // 数据块块号（可固定分配）
    uint8_t*            data[NEWFS_DATA_PER_FILE];            // 指向数据块的指针

    /* 其他字段 */
    int                 dir_cnt;            // 如果是目录类型文件，下面有几个目录项
    struct newfs_dentry* dentry;            // 指向该inode的dentry
    struct newfs_dentry* dentrys;           // 所有目录项
};

struct newfs_dentry {
    /* 文件名称 */
    char     fname[MAX_NAME_LEN];

    /* inode编号 */
    uint32_t ino;

    /* TODO: Define yourself */
    /* 文件类型 */
    NEWFS_FILE_TYPE     ftype;              // 文件类型（目录类型、普通文件类型）

    struct newfs_dentry* parent;
    struct newfs_dentry* brother;
    struct newfs_inode* inode;
};
```

这两处数据结构也与 simplefs 定义的类似，包含与磁盘操作相关数据结构的所有字段，而这里的索引节点项与文件目录项建立了映射的关系，索引节点项指向子目录项，文件目录项中记录着对应的索引节点，这样设计能够方便进行查找和修改。

核心代码：

首先是一些工具方法，这些方法在不同的文件操作中都有涉及，通过单独实现这些方法能够提高代码的维护性与复用性，它们主要定义在 newfs_utils.c 中：

```
/******************************************************************************
* SECTION: newfs_utils.c
*******************************************************************************/
char*               newfs_get_fname(const char *);
int                 newfs_calc_lvl(const char *);
int                 newfs_driver_read(int , uint8_t *, int);
int                 newfs_driver_write(int , uint8_t *, int);
int                 newfs_alloc_dentry(struct newfs_inode* , struct newfs_dentry*);
struct              newfs_inode* newfs_alloc_inode(struct newfs_dentry *);
int                 newfs_sync_inode(struct newfs_inode *);
struct              newfs_inode* newfs_read_inode(struct newfs_dentry * , int);
struct              newfs_dentry* newfs_get_dentry(struct newfs_inode * , int);
struct              newfs_dentry* newfs_lookup(const char * , boolean* , boolean*);
int                 newfs_mount(struct custom_options options);
int                 newfs_umount();
int                 newfs_alloc_data_blk(struct newfs_inode * inode, int blk_idx);
int                 newfs_drop_inode(struct newfs_inode * inode);
int                 newfs_drop_dentry(struct newfs_inode * inode, struct newfs_dentry * dentry);
```

获取指定路径的文件名(newfs_get_fname)：

```
/**
 * @brief 获取指定路径下面对应的文件名
 *
 * @param path
 * @return char*
 */
char* newfs_get_fname(const char* path) {
    char ch = '/';
    char *q = strrchr(path, ch) + 1;
    return q;
}
```

实现很简单，前面加上主目录地址’/’之后将字符串拼接起来返回即可

计算对应路径文件层级(newfs_calc-lvl)：

```
/**
 * @brief 计算路径的层级
 * @param path
 * @return int
 */
int newfs_calc_lvl(const char * path) {
    char* str = path;
    int   lvl = 0;
    if (strcmp(path, "/") == 0) {
        return lvl;
    }
    while (*str != NULL) {
        if (*str == '/') {
            lvl++;
        }
        str++;
    }
    return lvl;
}
```

由于每个层级都会有’/’作为分隔，因此只需要进行循环遍历，如果遇到了’/’就将 lvl 层级加一即可

读磁盘(newfs_driver_read)：

```c
/**
 * @brief 从磁盘中读取对应偏移地址的内容到输出内容中
 *
 * @param offset
 * @param out_content
 * @param size
 * @return int
 */
int newfs_driver_read(int offset, uint8_t *out_content, int size) {
    /* 读取逻辑块 */
    int      offset_aligned = NEWFS_ROUND_DOWN(offset, NEWFS_BLK_SZ());
    int      bias           = offset - offset_aligned;
    int      size_aligned   = NEWFS_ROUND_UP((size + bias), NEWFS_BLK_SZ());
    uint8_t* temp_content   = (uint8_t*)malloc(size_aligned);
    uint8_t* cur            = temp_content;

    ddriver_seek(NEWFS_DRIVER(), offset_aligned, SEEK_SET);

    while (size_aligned != 0)
    {
        ddriver_read(NEWFS_DRIVER(), cur, NEWFS_IO_SZ());
        cur          += NEWFS_IO_SZ();
        size_aligned -= NEWFS_IO_SZ();
    }
    memcpy(out_content, temp_content + bias, size);
    free(temp_content);
    return NEWFS_ERROR_NONE;
}
```

借鉴实验指导书提供的读取思想，首先确定读取的上界 up 与下界 down，先将磁盘的内容读到内存 mem 中，之后从内存中拷贝文件数据到 out_content 中即可

写磁盘(newfs_driver_write)：

```c
/**
 * @brief 将指定内容的数据写入到磁盘对应偏移地址中
 *
 * @param offset
 * @param in_content
 * @param size
 * @return int
 */
int newfs_driver_write(int offset, uint8_t *in_content, int size) {
    /* 读取逻辑块 */
    int      offset_aligned = NEWFS_ROUND_DOWN(offset, NEWFS_BLK_SZ());
    int      bias           = offset - offset_aligned;
    int      size_aligned   = NEWFS_ROUND_UP((size + bias), NEWFS_BLK_SZ());
    uint8_t* temp_content   = (uint8_t*)malloc(size_aligned);
    uint8_t* cur            = temp_content;
    newfs_driver_read(offset_aligned, temp_content, size_aligned);
    memcpy(temp_content + bias, in_content, size);

    ddriver_seek(NEWFS_DRIVER(), offset_aligned, SEEK_SET);
    while (size_aligned != 0)
    {
        ddriver_write(NEWFS_DRIVER(), cur, NEWFS_IO_SZ());
        cur          += NEWFS_IO_SZ();
        size_aligned -= NEWFS_IO_SZ();
    }

    free(temp_content);
    return NEWFS_ERROR_NONE;
}
```

写回步骤主要经过读-修改-写回的步骤，首先将对应的写入内容写入到内存中，之后将对应内存的内容依次写回到磁盘中。

插入 dentry 到 inode(newfs_alloc_dentry):

```
/**
 * @brief 将dentry插入inode中，采用头插法
 *
 * @param inode
 * @param dentry
 * @return int
 */
int newfs_alloc_dentry(struct newfs_inode* inode, struct newfs_dentry* dentry) {
    if (inode->dentrys == NULL) {
        inode->dentrys = dentry;
    }
    else {
        dentry->brother = inode->dentrys;
        inode->dentrys = dentry;
    }
    inode->dir_cnt++;
```

这里采用头插法将 dentry 目录项插入到索引节点中，我们会先判断索引节点中对应的所有目录项是否为空，为空那么直接将 dentry 插入即可，否则要更改 dentry 的兄弟指针，同时让索引节点中的 dentrys 指向最新的 dentry 头部指针

```
    // 分配数据块
    int cur_blk = inode->dir_cnt / MAX_DENTRY_PER_BLK();
    if (inode->dir_cnt % MAX_DENTRY_PER_BLK() == 1) {
        /* 当前数据块已满，需要寻找新的数据块*/
        int byte_cursor = 0;
        int bit_cursor  = 0;
        int dno_cursor  = 0;
        boolean is_find_free_data_blk = FALSE;
        /* 检查位图是否有空位 */
        for (byte_cursor = 0; byte_cursor < NEWFS_BLKS_SZ(newfs_super.map_data_blks);
            byte_cursor++)
        {
            for (bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {
                if((newfs_super.map_data[byte_cursor] & (0x1 << bit_cursor)) == 0) {
                                                        /* 当前dno_cursor位置空闲 */
                    newfs_super.map_data[byte_cursor] |= (0x1 << bit_cursor);
                    //在指定位置的数据块指针中插入当前数据块
                    inode->block_pointer[cur_blk] = dno_cursor;
                    is_find_free_data_blk = TRUE;
                    break;
                }
                dno_cursor++;
            }
            if (is_find_free_data_blk) {
                break;
            }
        }

        if (!is_find_free_data_blk || dno_cursor == newfs_super.max_data)
            return -NEWFS_ERROR_NOSPACE;
    }

    return inode->dir_cnt;
}
```

在插入目录项完毕后，要对当前目录项分配对应的地址空间，这里我们会首先通过已有的目录项个数计算当前目录项应该存储到哪个数据块中，之后进行遍历查看数据块位图中是否有对应的空闲位，如果有那么将这一位设置为 1 表示已使用，同时将对应数据块位置插入到索引节点的数据块指针数组中，最后分配成功返回目录项个数，否则返回错误

分配 inode 索引节点(newfs_alloc_inode)：

```c
/**
 * @brief 分配inode索引节点
 *
 * @param dentry
 * @return int
 */
struct newfs_inode* newfs_alloc_inode(struct newfs_dentry * dentry) {
    struct newfs_inode* inode;
    int byte_cursor = 0;
    int bit_cursor  = 0;
    int ino_cursor  = 0;
    boolean is_find_free_entry = FALSE;


    /* 检查位图是否有空位 */
    for (byte_cursor = 0; byte_cursor < NEWFS_BLKS_SZ(newfs_super.map_inode_blks);
         byte_cursor++)
    {
        for (bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {
            if((newfs_super.map_inode[byte_cursor] & (0x1 << bit_cursor)) == 0) {
                                                    /* 当前ino_cursor位置空闲 */
                newfs_super.map_inode[byte_cursor] |= (0x1 << bit_cursor);
                is_find_free_entry = TRUE;
                break;
            }
            ino_cursor++;
        }
        if (is_find_free_entry) {
            break;
        }
    }

    if (!is_find_free_entry || ino_cursor == newfs_super.max_ino)
        return -NEWFS_ERROR_NOSPACE;
```

在这个方法中，我们要为当前 inode 节点分配对应的数据块指针同时建立与 dentry 目录项的映射，首先我们要判断对应的索引节点位图中是否有空位，与上面的分配目录项方法类似，我们进行遍历判断，如果发现对应数据块空闲直接分配指针即可，而若并没有找到空闲位，则会返回错误

```c
    inode = (struct newfs_inode*)malloc(sizeof(struct newfs_inode));
    inode->ino  = ino_cursor;
    inode->size = 0;
    // dentry 指向 inode
    dentry->inode = inode;
    dentry->ino   = inode->ino;
    // inode 指向 dentry
    inode->dentry = dentry;

    inode->dir_cnt = 0;
    inode->dentrys = NULL;

    for (int i = 0; i < NEWFS_DATA_PER_FILE; i++){
        inode->block_pointer[i] = -1;
    }
    // inode指向文件类型，则分配数据指针
    if (NEWFS_IS_REG(inode)) {
        for (int i = 0; i < NEWFS_DATA_PER_FILE; i++) {
            inode->data[i] = (uint8_t *)malloc(NEWFS_BLK_SZ());
        }
    }
    return inode;
}
```

在找到索引节点位图的空闲位后，我们要建立索引节点与目录项之间的映射，目录项对应的索引节点项指向当前的索引节点，同时更改对应的索引节点号，索引节点中的目录项指

向传递的目录项参数，而新分配的索引节点项中还没有存储对应的文件目录项数据块指针，因此全部置为-1，同时如果 inode 为文件类型，要注意为 NEWFS_DATA_PER_FILE 个文件项分配数据指针，最后返回分配的索引节点项 inode 即可

将 inode 写回磁盘(newfs_sync_inode)：

```
/**
 * @brief 将内存inode及其下方结构全部刷回磁盘
 *
 * @param inode
 * @return int
 */
int newfs_sync_inode(struct newfs_inode * inode) {
    struct newfs_inode_d  inode_d;
    struct newfs_dentry*  dentry_cursor;
    struct newfs_dentry*  cur_dentry_cursor;
    struct newfs_dentry_d dentry_d;
    int ino             = inode->ino;
    int offset;

    // 将内存中的 inode 刷回 磁盘的 inode_d
    inode_d.ino         = ino;
    inode_d.size        = inode->size;
    inode_d.ftype       = inode->dentry->ftype;
    inode_d.dir_cnt     = inode->dir_cnt;
    for (int i = 0; i < NEWFS_DATA_PER_FILE; i++) {
        inode_d.block_pointer[i] = inode->block_pointer[i];
    }

    /* 先写inode本身 */
    if (newfs_driver_write(NEWFS_INO_OFS(ino), (uint8_t *)&inode_d,
        NEWFS_INODE_SZ()) != NEWFS_ERROR_NONE) {
        NEWFS_DBG("[%s] io error\n", __func__);
        return -NEWFS_ERROR_IO;
    }
```

这个方法主要应用于卸载文件系统的时候，将内存中的数据全部写回磁盘中，这个方法是写回 inode 索引节点项，由于内存的 inode 数据结构与磁盘的 inode 数据结构定义有差别，因此这里要重新定义对应的 inode_d 类型，同时将内存的 inode 字段刷回磁盘的 inode_d 结构体中，在刷回后，通过调用写磁盘方法，将 inode_d 本身节点写回

```
/* 再写inode下方的数据 */
if (NEWFS_IS_DIR(inode)) { /* 如果当前inode是目录，那么数据是目录项，且目录项的inode也要写回 */
    dentry_cursor = inode->dentrys;

    for (int i = 0; i < NEWFS_DATA_PER_FILE; i++) {
        offset = NEWFS_DATA_OFS(inode->block_pointer[i]);
        while ((dentry_cursor != NULL) && (offset < NEWFS_DATA_OFS(inode->block_pointer[i] + 1))) {
            memcpy(dentry_d.fname, dentry_cursor->fname, NEWFS_MAX_FILE_NAME);
            dentry_d.ftype = dentry_cursor->ftype;
            dentry_d.ino = dentry_cursor->ino;
            if (newfs_driver_write(offset, (uint8_t *)&dentry_d,
                sizeof(struct newfs_dentry_d)) != NEWFS_ERROR_NONE) {
                NEWFS_DBG("[%s] io error\n", __func__);
                return -NEWFS_ERROR_IO;
            }

            // 递归调用 将目录项的inode写回
            if (dentry_cursor->inode != NULL) {
                newfs_sync_inode(dentry_cursor->inode);
            }

            // 下一个目录项
            cur_dentry_cursor = dentry_cursor;
            dentry_cursor = dentry_cursor->brother;
            offset += sizeof(struct newfs_dentry_d);

            free(cur_dentry_cursor);
        }
    }
}
else if (NEWFS_IS_REG(inode)) { /* 如果当前inode是文件，那么数据是文件内容，直接写即可 */
```

之后写回 inode 里面包含的 dentrys 结构体字段，这里由于目录项与文件写回的策略不同，我们要进行条件判断分别讨论，对于目录项，我们通过 dentrys 的指针遍历存储的所有文件目录数，将 dentry_cursor 当前指向的目录项字段刷回 dentry 中，之后调用写磁盘方法将当前目录项写回，由于目录项里面还包含其他的文件或者目录，因此对于儿子节点，我们要进行递归调用写回该目录下面的索引节点项，对于兄弟节点，我们直接通过链表直线下一个目录项即可，这里注意的是将对应的偏移量实时更新

```
}
else if (NEWFS_IS_REG(inode)) { /* 如果当前inode是文件，那么数据是文件内容，直接写即可 */
    for (int i = 0; i < NEWFS_DATA_PER_FILE; i++) {
        if(inode->block_pointer[i] == -1) continue;
        if (newfs_driver_write(NEWFS_DATA_OFS(inode->block_pointer[i]), inode->data[i],
            NEWFS_BLK_SZ()) != NEWFS_ERROR_NONE) {
            NEWFS_DBG("[%s] io error\n", __func__);
            return -NEWFS_ERROR_IO;
        }
        free(inode->data[i]);
    }
}
free(inode);
return NEWFS_ERROR_NONE;
```

对于普通的文件类型，我们只需要将对应的文件内容写回到磁盘中即可

读取指定编号的索引节点信息(newfs_read_inode)：

```c
/**
 * @brief 读取指定编号的索引节点信息
 *
 * @param dentry dentry指向ino, 读取该inode
 * @param ino inode唯一编号
 * @return struct newfs_inode*
 */
struct newfs_inode* newfs_read_inode(struct newfs_dentry * dentry, int ino) {
    struct newfs_inode* inode = (struct newfs_inode*)malloc(sizeof(struct newfs_inode));
    struct newfs_inode_d inode_d;
    struct newfs_dentry* sub_dentry;
    struct newfs_dentry_d dentry_d;
    int dir_cnt = 0;
    int offset;

    if (newfs_driver_read(NEWFS_INO_OFS(ino), (uint8_t *)&inode_d,
                          NEWFS_INODE_SZ()) != NEWFS_ERROR_NONE) {
        NEWFS_DBG("[%s] io error\n", __func__);
        return NULL;
    }
    inode->dir_cnt = 0;
    inode->ino = inode_d.ino;
    inode->size = inode_d.size;
    inode->dentry = dentry;
    inode->dentrys = NULL;

    for(int i = 0; i < NEWFS_DATA_PER_FILE; i++) {
        inode->block_pointer[i] = inode_d.block_pointer[i];
    }
```

当我们进行索引节点的读取时，与写回索引节点相反，我们这里要将磁盘的数据结构 inode_d 的字段写回到内存的数据结构 inode 中，比如对应的索引节点编号、大小、目录项、文件的数据块指针等

```c
    /* 内存中的inode的数据或子目录项部分也需要读出 */
    if (NEWFS_IS_DIR(inode)) {
        dir_cnt = inode_d.dir_cnt;
        // 节点指向的数据块
        for (int i = 0; i < NEWFS_DATA_PER_FILE; i++) {
            offset = NEWFS_DATA_OFS(inode->block_pointer[i]);
            // 磁盘无指针, 按照 dentry 大小遍历
            while ((dir_cnt > 0) && (offset + sizeof(struct newfs_dentry_d) < NEWFS_DATA_OFS(inode->block_pointer[i] + 1))) {
                if (newfs_driver_read(offset, (uint8_t *)&dentry_d,
                    sizeof(struct newfs_dentry_d)) != NEWFS_ERROR_NONE) {
                    NEWFS_DBG("[%s] io error\n", __func__);
                    return NULL;
                }
                sub_dentry = new_dentry(dentry_d.fname, dentry_d.ftype);
                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 --;
            }
        }
    }
    else if (NEWFS_IS_REG(inode)) {
        for (int i = 0; i < NEWFS_DATA_PER_FILE; i++) {
            inode->data[i] = (uint8_t *)malloc(NEWFS_BLK_SZ());
            if(inode->block_pointer[i] == -1) continue;
            if (newfs_driver_read(NEWFS_DATA_OFS(inode->block_pointer[i]), (uint8_t *)inode->data[i],
                NEWFS_BLK_SZ()) != NEWFS_ERROR_NONE) {
                NEWFS_DBG("[%s] io error\n", __func__);
                return NULL;
            }
        }
    }
    return inode;
}
```

之后由于索引节点项中包含对应文件目录项的指针，因此我们还要依次读入 inode 结构体中的 dentry 指针字段，对于目录项类型，我们首先得到对应的目录项个数，通过偏移量以及目录项个数的判断条件进行遍历，将读取的 dentry_d 字段写回到临时目录项 sub_dentry 中，之后调用分配目录项数据块指针的方法，分配目录项的数据块指针并将当前目录项插入到索引节点中；对于文件类型，我们只需要为对应的数据块申请对应空间，将文件内容写回

到对应的数据块地址中即可，最后返回读取的 inode 索引节点

获得第 dir 个 dentry(newfs_get_dentry)：

```
/**
 * @brief 获得第 dir 个 dentry
 *
 * @param inode
 * @param dir [0...]
 * @return struct newfs_dentry*
 */
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;
}
```

由于目录项在索引节点中通过 brother 指针连接形成了一个双向链表，因此我们通过使用 cnt 计数，直接对双向链表遍历计数即可

查找文件或者目录(newfs_lookup)：

```
struct newfs_dentry* newfs_lookup(const char * path, boolean* is_find, boolean* is_root) {
    struct newfs_dentry* dentry_cursor = newfs_super.root_dentry;
    struct newfs_dentry* dentry_ret = NULL;
    struct newfs_inode*  inode;
    int   total_lvl = newfs_calc_lvl(path);
    int   lvl = 0;
    boolean is_hit;
    char* fname = NULL;
    char* path_cpy = (char*)malloc(sizeof(path));
    *is_root = FALSE;
    strcpy(path_cpy, path);

    if (total_lvl == 0) {                           /* 根目录 */
        *is_find = TRUE;
        *is_root = TRUE;
        dentry_ret = newfs_super.root_dentry;
    }
```

这个方法会查找当前索引节点下对应的文件名匹配的目录项，若不存在会返回外层目录项，我们在进行查找时，传递了 is_find 与 is_root 指针，方便于外层函数的查找判断，首先通过 newfs_calc_lvl 计算对应路径的文件层级，如果层级为零说明是根目录，那么将对应的 is_find 与 is_root 设置为 true，将 dentry_ret 设置为根目录即可

```c
// 最外层文件夹名称
fname = strtok(path_cpy, "/");
while (fname)
{
    lvl++;
    if (dentry_cursor->inode == NULL) {           /* Cache机制 */
        newfs_read_inode(dentry_cursor, dentry_cursor->ino);
    }

    // 当前 dentry 对应的 inode
    inode = dentry_cursor->inode;

    // 若出现文件类型但层数未到
    if (NEWFS_IS_REG(inode) && lvl < total_lvl) {
        NEWFS_DBG("[%s] not a dir\n", __func__);
        dentry_ret = inode->dentry;
        break;
    }

    // 若为文件夹类型
    if (NEWFS_IS_DIR(inode)) {
        dentry_cursor = inode->dentrys;
        is_hit        = FALSE;

        while (dentry_cursor)   /* 遍历子目录项 */
        {
            if (memcmp(dentry_cursor->fname, fname, strlen(fname)) == 0) {
                is_hit = TRUE;
                break;
            }
            dentry_cursor = dentry_cursor->brother;
        }
```

之后若当前目录项的索引节点为空，那么说明目录项还没被加载到内存中，从磁盘中读取目录项，如果当前是文件层级但层数小于对应计算层级，那么说明查找失败，终止循环返回即可，如果是文件夹类型，那么就对所有的子目录项进行遍历，如果文件名匹配，那么查找成功

```c
            // 未找到该文件 or 文件夹
            // mkdir mknod
            if (!is_hit) {
                *is_find = FALSE;
                NEWFS_DBG("[%s] not found %s\n", __func__, fname);
                dentry_ret = inode->dentry;
                break;
            }

            // 找到正确的文件
            if (is_hit && lvl == total_lvl) {
                *is_find = TRUE;
                dentry_ret = dentry_cursor;
                break;
            }
        }
        fname = strtok(NULL, "/");
    }

    // 从磁盘中读出 inode
    if (dentry_ret->inode == NULL) {
        dentry_ret->inode = newfs_read_inode(dentry_ret, dentry_ret->ino);
    }

    return dentry_ret;
}
```

如果没有找到对应名称的文件或者文件夹，返回错误，反之设置 is_find 为真，将

dentry_ret 设置为当前目录项，如果目录项中的索引节点为空，从磁盘中读取对应的索引节点，最后返回名称匹配或者外层的目录项。

挂载文件系统(newfs_mount)：

```
int newfs_mount(struct custom_options options) {
    int ret = NEWFS_ERROR_NONE;
    struct newfs_super_d newfs_super_d;
    struct newfs_dentry* root_dentry;
    struct newfs_inode* root_inode;
    boolean is_init = FALSE;

    // 1. 初始化基本信息
    newfs_super.is_mounted = FALSE;
    int fd = ddriver_open(options.device);
    if (fd < 0) {
        return fd;
    }
    newfs_super.fd = fd;

    // 2. 获取设备信息
    ddriver_ioctl(NEWFS_DRIVER(), IOC_REQ_DEVICE_SIZE, &newfs_super.sz_disk);
    ddriver_ioctl(NEWFS_DRIVER(), IOC_REQ_DEVICE_IO_SZ, &newfs_super.sz_io);
    newfs_super.sz_blks = NEWFS_IO_SZ() * 2;

    // 3. 创建根目录项
    root_dentry = new_dentry("/", NEWFS_DIR);

    // 4. 读取并检查超级块
    if (newfs_driver_read(NEWFS_SUPER_OFS, (uint8_t*)(&newfs_super_d),
        sizeof(struct newfs_super_d)) != NEWFS_ERROR_NONE) {
        return -NEWFS_ERROR_IO;
    }

    if (newfs_super_d.magic_num != NEWFS_MAGIC_NUM) {
        init_newfs_super_d(&newfs_super_d);
        is_init = TRUE;
    }
```

挂载文件系统主要分为 7 步，参考 sys_utils 的挂载方法实现，首先初始化对应的挂载状态，文件描述符信息，之后获取设备信息，其次创建根目录项，再读取并检查超级块对应的信息，如果超级块对应的魔数并没有设置，那么要初始化超级块结构体的字段信息，设置初始化标识为真

```
    // 5. 同步超级块到内存
    sync_super_to_memory(&newfs_super_d);

    // 6. 分配并读取位图
    newfs_super.map_inode = (uint8_t*)malloc(NEWFS_BLKS_SZ(newfs_super_d.map_inode_blks));
    newfs_super.map_data = (uint8_t*)malloc(NEWFS_BLKS_SZ(newfs_super_d.map_data_blks));

    if (newfs_driver_read(newfs_super_d.map_inode_offset, newfs_super.map_inode,
        NEWFS_BLKS_SZ(newfs_super_d.map_inode_blks)) != NEWFS_ERROR_NONE) {
        return -NEWFS_ERROR_IO;
    }

    if (newfs_driver_read(newfs_super_d.map_data_offset, newfs_super.map_data,
        NEWFS_BLKS_SZ(newfs_super_d.map_data_blks)) != NEWFS_ERROR_NONE) {
        return -NEWFS_ERROR_IO;
    }

    // 7. 处理根目录
    if (is_init) {
        root_inode = newfs_alloc_inode(root_dentry);
        newfs_sync_inode(root_inode);
    }

    root_inode = newfs_read_inode(root_dentry, NEWFS_ROOT_INO);
    root_dentry->inode = root_inode;
    newfs_super.root_dentry = root_dentry;
    newfs_super.is_mounted = TRUE;

    return ret;
}
```

之后同步超级块信息到内存中，由于位图中存储了目录项、索引节点到数据块地址的映射，因此预先分配并且读取对应的位图信息，然后处理根目录对应的索引节点映射，超级块中包含的目录项信息等，最后更新挂载状态即可

卸载文件系统(newfs_umount)：

```
/**
 * @brief 卸载文件系统
 */
int newfs_umount() {
    struct newfs_super_d newfs_super_d;

    // 1. 检查文件系统是否已挂载
    if (!newfs_super.is_mounted) {
        return NEWFS_ERROR_NONE;
    }

    // 2. 从根节点开始,递归地将所有inode写回磁盘
    newfs_sync_inode(newfs_super.root_dentry->inode);

    // 3. 将内存中的超级块信息同步到磁盘超级块结构
    sync_super_to_disk(&newfs_super_d);

    // 4. 将超级块写回磁盘
    if (newfs_driver_write(NEWFS_SUPER_OFS,
                           (uint8_t *)&newfs_super_d,
                           sizeof(struct newfs_super_d)) != NEWFS_ERROR_NONE) {
        return -NEWFS_ERROR_IO;
    }

    // 5. 将位图写回磁盘
    int ret = sync_maps_to_disk(&newfs_super_d);
    if (ret != NEWFS_ERROR_NONE) {
        return ret;
    }

    // 6. 清理资源
    free(newfs_super.map_inode);
    free(newfs_super.map_data);
    ddriver_close(NEWFS_DRIVER());

    return NEWFS_ERROR_NONE;
}
```

卸载文件系统函数进行的主要操作就是将内存中的所有数据全部写回到磁盘中，首先我们判断是否挂载，未挂载直接返回，之后依次将 inode 索引节点、超级块信息、位图信息写回到磁盘中，我们调用 sync_super_to_disk 将与内存相关的超级块数据结构刷回到与磁盘相关的超级块数据结构中，最后释放对应的对象内存，关闭 ddriver 连接，返回即可

```c
/**
 * @brief 从内存超级块同步到磁盘超级块
 */
static void sync_super_to_disk(struct newfs_super_d* newfs_super_d) {
    newfs_super_d->magic_num = NEWFS_MAGIC_NUM;
    newfs_super_d->sz_usage = newfs_super.sz_usage;
    newfs_super_d->super_blks = newfs_super.super_blks;
    newfs_super_d->super_blk_offset = newfs_super.super_blk_offset;
    newfs_super_d->map_inode_blks = newfs_super.map_inode_blks;
    newfs_super_d->map_inode_offset = newfs_super.map_inode_offset;
    newfs_super_d->map_data_blks = newfs_super.map_data_blks;
    newfs_super_d->map_data_offset = newfs_super.map_data_offset;
    newfs_super_d->ino_blks = newfs_super.ino_blks;
    newfs_super_d->ino_offset = newfs_super.ino_offset;
    newfs_super_d->data_blks = newfs_super.data_blks;
    newfs_super_d->data_offset = newfs_super.data_offset;
    newfs_super_d->max_ino = newfs_super.max_ino;
    newfs_super_d->max_data = newfs_super.max_data;
}
```

为 inode 分配数据块(newfs_alloc_data_blk)：

```c
/**
 * @brief 为inode分配一个数据块
 *
 * @param inode 需要分配数据块的inode
 * @param blk_no 分配的数据块编号
 * @return int 成功返回NEWFS_ERROR_NONE, 失败返回错误码
 */
int newfs_alloc_data_blk(struct newfs_inode* inode, int blk_no) {
    // 检查参数
    if (blk_no >= NEWFS_DATA_PER_FILE) {
        return -NEWFS_ERROR_NOSPACE;
    }

    // 在数据块位图中寻找空闲块
    int byte_cursor = 0;
    int bit_cursor = 0;
    int data_blk_cursor = 0;
    boolean is_find_free_blk = FALSE;

    // 遍历数据块位图
    for (byte_cursor = 0; byte_cursor < NEWFS_BLKS_SZ(newfs_super.map_data_blks); byte_cursor++) {
        for (bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {
            if ((newfs_super.map_data[byte_cursor] & (0x1 << bit_cursor)) == 0) {
                // 找到空闲数据块
                newfs_super.map_data[byte_cursor] |= (0x1 << bit_cursor);
                is_find_free_blk = TRUE;
                break;
            }
            data_blk_cursor++;
        }
        if (is_find_free_blk) {
            break;
        }
    }
```

与分配索引节点方法类似，我们首先要在数据块位图中查找对应的空闲块信息，如果能够找到对应的空闲块，那么就将设置对应的空闲块指针

```
    // 检查是否找到空闲块
    if (!is_find_free_blk || data_blk_cursor >= newfs_super.max_data) {
        return -NEWFS_ERROR_NOSPACE;
    }

    // 分配数据块
    inode->block_pointer[blk_no] = data_blk_cursor;

    // 为数据块分配内存
    if (inode->data[blk_no] == NULL) {
        inode->data[blk_no] = (uint8_t *)malloc(NEWFS_BLK_SZ());
        if (inode->data[blk_no] == NULL) {
            // 分配失败，回退位图标记
            newfs_super.map_data[byte_cursor] &= ~(0x1 << bit_cursor);
            return -NEWFS_ERROR_NOSPACE;
        }
    }

    return NEWFS_ERROR_NONE;
}
```

之后如果没有查找到空闲数据块，返回错误即可，反之为数据块申请内存，设置对应的数据块指针地址，若申请空间失败返回错误

删除 dentry 目录项(newfs_drop_dentry)：

```
/**
 * @brief 删除dentry目录项
 *
 * @param inode 包含当前dentry目录项的索引节点
 * @param dentry 要删除的dentry目录项
 * @return int 成功返回NEWFS_ERROR_NONE，失败返回错误码
 */
int newfs_drop_dentry(struct newfs_inode* inode, struct newfs_dentry* dentry) {
    if (inode->dentrys == dentry) {
        inode->dentrys = dentry->brother;
    }
    else {
        struct newfs_dentry* dentry_cursor = inode->dentrys;
        while (dentry_cursor) {
            if (dentry_cursor->brother == dentry) {
                //若下一个兄弟节点为删除目录项，更新指针信息
                dentry_cursor->brother = dentry->brother;
                break;
            }
            dentry_cursor = dentry_cursor->brother;
        }
    }
    inode->dir_cnt--;
    free(dentry);
    return NEWFS_ERROR_NONE;
}
```

在删除目录项时，由于目录项与索引节点之间建立了映射的关系，因此我们要解除索引

节点与目录项之间指针的指向关系，让索引节点指向要删除的目录项的兄弟目录项，这里分为两种情况，一种为删除的目录项为头节点，另一种为中间节点，在指针更新成功后，文件数量 dir_cnt 减一，返回

删除内存中的 inode 及其对应的 dentry 和 data(newfs_drop_inode)：

```
/**
 * @brief 删除内存中的一个inode及其对应的dentry和data
 *
 * @param inode 要删除的inode
 * @return int 成功返回NEWFS_ERROR_NONE, 失败返回错误码
 */
int newfs_drop_inode(struct newfs_inode* inode) {
    struct newfs_dentry* dentry_cursor;
    struct newfs_dentry* dentry_to_free;
    int byte_cursor = 0;
    int bit_cursor = 0;
    int ino_cursor = 0;
    boolean is_find = FALSE;

    if (inode == NULL) {
        return NEWFS_ERROR_NONE;
    }

    if (NEWFS_IS_DIR(inode)) {
        dentry_cursor = inode->dentrys;
        /* 递归删除目录下的所有目录项 */
        while (dentry_cursor) {
            dentry_to_free = dentry_cursor;
            dentry_cursor = dentry_cursor->brother;
            newfs_drop_inode(dentry_to_free->inode);
            free(dentry_to_free);
        }
    }
    else if (NEWFS_IS_REG(inode)) {
```

删除 inode 信息时，由于 inode 索引节点包含了目录项的指针，因此要从内到外联动删除，而删除目录项时分为两种情况考虑，一种为文件夹类型，这时要不断更新释放的目录项指针，同时递归删除当前目录下的所有目录项

```
int newfs_drop_inode(struct newfs_inode* inode) {
    if (NEWFS_IS_DIR(inode)) {
    else if (NEWFS_IS_REG(inode)) {
        /* 释放文件对应的数据块 */
        for (int i = 0; i < NEWFS_DATA_PER_FILE; i++) {
            if (inode->block_pointer[i] != -1) {
                /* 清除数据块位图 */
                int blk_byte = inode->block_pointer[i] / UINT8_BITS;
                int blk_bit = inode->block_pointer[i] % UINT8_BITS;
                newfs_super.map_data[blk_byte] &= ~(0x1 << blk_bit);

                /* 释放数据块内存 */
                if (inode->data[i]) {
                    free(inode->data[i]);
                    inode->data[i] = NULL;
                }
            }
        }
    }

    /* 清除inode位图中对应的位 */
    for (byte_cursor = 0; byte_cursor < NEWFS_BLKS_SZ(newfs_super.map_inode_blks); byte_cursor++) {
        for (bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {
            if (ino_cursor == inode->ino) {
                newfs_super.map_inode[byte_cursor] &= (uint8_t)(~(0x1 << bit_cursor));
                is_find = TRUE;
                break;
            }
            ino_cursor++;
        }
        if (is_find) {
            break;
        }
    }

    /* 释放inode内存 */
    free(inode);

    return NEWFS_ERROR_NONE;
}
```

对于普通文件类型，由于文件类型会建立对应于数据块的映射，因此在释放数据块内存的同时也要清除对应的数据块位图信息，最后内部的目录项删除完毕后，删除 inode 位图对应的位，最后再释放内存

至此，newfs_utils.c 中的函数方法全部介绍完毕了，它们是文件系统增删改查的基础

下面介绍核心方法 newfs.c：

挂载文件系统(newfs_init)：

```
/**
 * @brief 挂载（mount）文件系统
 *
 * @param conn_info 可忽略，一些建立连接相关的信息
 * @return void*
 */
void *newfs_init(struct fuse_conn_info *conn_info)
{
    /* TODO: 在这里进行挂载 */
    if (newfs_mount(newfs_options) != NEWFS_ERROR_NONE)
    {
        NEWFS_DBG("[%s] mount error\n", __func__);
        fuse_exit(fuse_get_context()->fuse);
        return NULL;
    }
    return NULL;

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

直接调用 newfs_mount 方法挂载即可

卸载文件系统(newfs_destroy)：

```c
/**
 * @brief 卸载 (umount) 文件系统
 *
 * @param p 可忽略
 * @return void
 */
void newfs_destroy(void *p)
{
    /* TODO: 在这里进行卸载 */
    if (newfs_umount() != NEWFS_ERROR_NONE)
    {
        NEWFS_DBG("[%s] unmount error\n", __func__);
        fuse_exit(fuse_get_context()->fuse);
        return;
    }
    return;
}
```

调用 newfs_umount 卸载方法即可

创建目录(newfs_mkdir)：

```c
/**
 * @brief 创建目录
 *
 * @param path 相对于挂载点的路径
 * @param mode 创建模式（只读？只写？），可忽略
 * @return int 0成功，否则返回对应错误号
 */
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 -NEWFS_ERROR_EXISTS;
    }

    if (NEWFS_IS_REG(last_dentry->inode))
    {
        return -NEWFS_ERROR_UNSUPPORTED;
    }

    fname = newfs_get_fname(path);
    dentry = new_dentry(fname, NEWFS_DIR);
    dentry->parent = last_dentry;
    inode = newfs_alloc_inode(dentry);              // son 的 inode
    newfs_alloc_dentry(last_dentry->inode, dentry); // parent 的 inode

    return NEWFS_ERROR_NONE;
}
```

首先通过 newfs_lookup 函数找到对应路径的外层目录，之后创建新的 dentry 目录项，更新对应的父目录项信息，调用 newfs_allock_inode 与 newfs_alloc_dentry 方法建立 inode 索

引节点与 dentry 目录项的映射，让父目录项的索引节点指向子目录

获取文件或者目录的属性(newfs_getattr)：

```c
int newfs_getattr(const char *path, struct stat *newfs_stat)
{
    /* TODO: 解析路径，获取Inode，填充newfs_stat*/
    boolean is_find, is_root;
    struct newfs_dentry *dentry = newfs_lookup(path, &is_find, &is_root);
    if (is_find == FALSE)
    {
        return -NEWFS_ERROR_NOTFOUND;
    }

    if (NEWFS_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); // 文件大小
    }
    else if (NEWFS_IS_REG(dentry->inode))
    {
        newfs_stat->st_mode = S_IFREG | NEWFS_DEFAULT_PERM;
        newfs_stat->st_size = dentry->inode->size;
    }

    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 = NEWFS_BLK_SZ(); // 逻辑块大小

    if (is_root)
    {
        newfs_stat->st_size = newfs_super.sz_usage;
        newfs_stat->st_blocks = NEWFS_DISK_SZ() / NEWFS_BLK_SZ(); // 文件块数
        newfs_stat->st_nlink = 2;                                 /* !特殊，根目录link数为2 */
    }
    return NEWFS_ERROR_NONE;
}
```

参考 simplefs 的方法实现，首先查找文件，如果查找不到文件直接返回错误，之后根据文件类型的不同（文件夹还是文件），来设置 newfs_stat 的 st_mode 与 st_size 字段，如果是根节点还要设置 newfs_stat 的 st_size、st_blocks、st_nlink 字段，注意的是为了区分根目录与普通文件，根目录的 st_nlink 设置为 2

获取文件属性并通过 filler 填充到 buf 中(newfs_readdir)：

```c
int newfs_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset,
                  struct fuse_file_info *fi)
{
    /* TODO: 解析路径，获取目录的Inode，并读取目录项，利用filler填充到buf */
    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;
    struct newfs_inode *inode;
    if (is_find)
    {
        inode = dentry->inode;
        sub_dentry = newfs_get_dentry(inode, cur_dir);
        if (sub_dentry)
        {
            filler(buf, sub_dentry->fname, NULL, ++offset);
        }
        return NEWFS_ERROR_NONE;
    }
    return -NEWFS_ERROR_NOTFOUND;
}
```

该方法主要应用于 ls 展示文件中，我们要读取当前路径下面对应的所有目录属性信息，

每个 dentry 属性中有下层的索引节点 inode 属性，又通过 newfs_get_dentry 得到对应索引节点指向的子目录文件，readdir 每次回处理一个目录项文件

创建文件(newfs_mknod)：

```
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 -NEWFS_ERROR_EXISTS;
    }

    fname = newfs_get_fname(path);

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

    return NEWFS_ERROR_NONE;
}
```

根据给定的路径以及指定文件类型，创建目录项还是普通文件，最后建立当前创建文件与上层目录的映射关系

写入文件(newfs_write)：

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

    if (!is_find)
    {
        return -NEWFS_ERROR_NOTFOUND;
    }

    inode = dentry->inode;
    if (NEWFS_IS_DIR(inode))
    {
        return -NEWFS_ERROR_ISDIR;
    }

    // 计算需要的数据块
    int blk_start = offset / NEWFS_BLK_SZ();
    int blk_end = (offset + size - 1) / NEWFS_BLK_SZ();
    int write_size = 0;
```

该方法用于 vim 写入文件，当我们使用 vim 写入文件时，如果文件不存在会首先创建文件，之后通过该方法写入文件，首先判断是否找到该文件以及该文件是否可写，之后计算对应需要的数据块上下界

```
// 检查并分配所需的数据块
for (int i = blk_start; i <= blk_end && i < NEWFS_DATA_PER_FILE; i++)
{
    if (inode->block_pointer[i] == -1)
    {
        // 分配新的数据块
        int ret = newfs_alloc_data_blk(inode, i);
        if (ret != NEWFS_ERROR_NONE)
        {
            return ret;
        }
    }

    int cur_offset = (i == blk_start) ? offset % NEWFS_BLK_SZ() : 0;
    int cur_size = NEWFS_BLK_SZ() - cur_offset;
    if (i == blk_end)
    {
        cur_size = ((offset + size) % NEWFS_BLK_SZ()) - cur_offset;
        if (cur_size <= 0)
            cur_size = NEWFS_BLK_SZ() - cur_offset;
    }
    if (write_size + cur_size > size)
    {
        cur_size = size - write_size;
    }

    memcpy(inode->data[i] + cur_offset, buf + write_size, cur_size);
    write_size += cur_size;
}

if (offset + size > inode->size)
{
    inode->size = offset + size;
}

return write_size;
```

在确定了数据块的上下界后，就只需要遍历这些数据块，判断对应的数据块指针映射是否为-1，若为-1 表示还没有分配数据块，为它们分配新的数据块，之后计算对应的数据块大小，如果是第一个数据块，那么要计算对应于边界的偏移量 cur_offset，否则 cur_offset 为 0，如果为最后一个块，要计算对应写入的块大小 cur_size，同时每次写入一个块后，更新对应的已经写入的文件数据大小 write_size，仿照 simplefs 更新对应的索引节点大小，最后返回写入数据大小

读文件方法(newfs_read)：

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

    if (!is_find)
    {
        return -NEWFS_ERROR_NOTFOUND;
    }

    inode = dentry->inode;
    if (NEWFS_IS_DIR(inode))
    {
        return -NEWFS_ERROR_ISDIR;
    }

    if (inode->size < offset)
    {
        return -NEWFS_ERROR_SEEK;
    }

    // 计算要读取的数据块
    int blk_start = offset / NEWFS_BLK_SZ();
    int blk_end = (offset + size - 1) / NEWFS_BLK_SZ();
    int read_size = 0;
```

在完成写文件方法后，读文件方法如法炮制即可，也是要读取文件，判断是否存在，判断文件是否合法，计算对应的数据块上下界

```
    // 读取每个数据块的内容
    for (int i = blk_start; i <= blk_end && i < NEWFS_DATA_PER_FILE; i++)
    {
        int cur_offset = (i == blk_start) ? offset % NEWFS_BLK_SZ() : 0;
        int cur_size = NEWFS_BLK_SZ() - cur_offset;
        if (i == blk_end)
        {
            cur_size = ((offset + size) % NEWFS_BLK_SZ()) - cur_offset;
            if (cur_size <= 0)
                cur_size = NEWFS_BLK_SZ() - cur_offset;
        }
        if (read_size + cur_size > size)
        {
            cur_size = size - read_size;
        }
        memcpy(buf + read_size, inode->data[i] + cur_offset, cur_size);
        read_size += cur_size;
    }

    return read_size;
}
```

得到上下界后遍历读取每个数据块的内容，读取过程中也是要维护对应读取偏移量 cur_offset 与对应读取的大小 cur_size，读入数据块的时候维护对应的 read_size 大小

删除文件(newfs_unlink)：

```c
/**
 * @brief 删除文件
 *
 * @param path 相对于挂载点的路径
 * @return int 0成功，否则返回对应错误号
 */
int newfs_unlink(const char *path) {
    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 -NEWFS_ERROR_NOTFOUND;
    }

    if (is_root) {
        return -NEWFS_ERROR_INVAL;  // 不能删除根目录
    }

    inode = dentry->inode;
    if (NEWFS_IS_DIR(inode)) {
        return -NEWFS_ERROR_ISDIR;  // 不能用unlink删除目录
    }

    newfs_drop_inode(inode);                        // 删除inode及其对应的数据块
    newfs_drop_dentry(dentry->parent->inode, dentry);// 从父目录的目录项链表中删除该目录项

    return NEWFS_ERROR_NONE;
}
```

删除文件的核心就是解除 inode 与 dentry 之间的映射，首先查找文件，如果没找到返回错误，根目录不能删除，文件夹类型不能删除，之后解除 inode 与 dentry 之间的映射关系即可

删除文件夹类型(newfs_rmdir)：

```c
 */
int newfs_rmdir(const char *path) {
    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 -NEWFS_ERROR_NOTFOUND;
    }

    if (is_root) {
        return -NEWFS_ERROR_INVAL;  // 不能删除根目录
    }

    inode = dentry->inode;
    if (!NEWFS_IS_DIR(inode)) {
        return -NEWFS_ERROR_NOTDIR;  // 不是目录
    }

    if (inode->dir_cnt != 0) {
        return -NEWFS_ERROR_NOTEMPTY;  // 目录不为空
    }

    newfs_drop_inode(inode);                        // 删除inode及其对应的数据块
    newfs_drop_dentry(dentry->parent->inode, dentry);// 从父目录的目录项链表中删除该目录项

    return NEWFS_ERROR_NONE;
}
```

与删除文件同理，值得注意的是在删除文件夹的时候，我们必须首先递归删除文件夹中的子文件，之后删除顶层的文件夹

重命名文件(newfs_rename)：

```c
int newfs_rename(const char *from, const char *to)
{
    /* 选做 */
    int ret = NEWFS_ERROR_NONE;
    boolean is_find, is_root;
    struct newfs_dentry* from_dentry = newfs_lookup(from, &is_find, &is_root);
    struct newfs_inode*  from_inode;
    struct newfs_dentry* to_dentry;
    mode_t mode = 0;
    if (is_find == FALSE) {
        return -NEWFS_ERROR_NOTFOUND;
    }

    if (strcmp(from, to) == 0) {
        return NEWFS_ERROR_NONE;
    }

    from_inode = from_dentry->inode;

    if (NEWFS_IS_DIR(from_inode)) {
        mode = S_IFDIR;
    }
    else if (NEWFS_IS_REG(from_inode)) {
        mode = S_IFREG;
    }

    ret = newfs_mknod(to, mode, NULL);
    if (ret != NEWFS_ERROR_NONE) {                  /* 保证目的文件不存在 */
        return ret;
    }

    to_dentry = newfs_lookup(to, &is_find, &is_root);
    newfs_drop_inode(to_dentry->inode);             /* 保证生成的inode被释放 */
    to_dentry->ino = from_inode->ino;               /* 指向新的inode */
    to_dentry->inode = from_inode;

    newfs_drop_dentry(from_dentry->parent->inode, from_dentry);
    return ret;
}
```

这里我们重命名文件，并不是真的修改文件名，而是另外创建一个新的文件名为指定文件名的文件，创建的新文件已经被父目录包含，因此只需要将原来节点的 inode 指针修改即可，并且将原来的文件释放即可

改变文件大小(newfs_truncate)：

```c
/**
 * @brief 改变文件大小
 *
 * @param path 相对于挂载点的路径
 * @param offset 改变后文件大小
 * @return int 0成功，否则返回对应错误号
 */
int newfs_truncate(const char *path, off_t offset) {
    boolean is_find, is_root;

    // 1. 查找文件
    struct newfs_dentry *dentry = newfs_lookup(path, &is_find, &is_root);
    if (!is_find) {
        return -NEWFS_ERROR_NOTFOUND;
    }

    // 2. 检查是否为目录
    struct newfs_inode *inode = dentry->inode;
    if (NEWFS_IS_DIR(inode)) {
        return -NEWFS_ERROR_ISDIR;
    }

    // 3. 检查新大小是否需要的数据块数超出限制
    int new_blks = NEWFS_ROUND_UP(offset, NEWFS_BLK_SZ()) / NEWFS_BLK_SZ();
    if (new_blks > NEWFS_DATA_PER_FILE) {
        return -NEWFS_ERROR_NOSPACE;
    }

    // 4. 更新文件大小
    inode->size = offset;

    return NEWFS_ERROR_NONE;
}
```

改变文件大小要做的就是要判断大小是否超出限制，若不超出直接修改即可

判断文件权限(newfs_access)：

```
int newfs_access(const char *path, int type)
{
    boolean is_find, is_root;
    struct newfs_dentry *dentry = newfs_lookup(path, &is_find, &is_root);
    return is_find ? NEWFS_ERROR_NONE : -NEWFS_ERROR_NOTFOUND;
}
/*****************************************************************************
```

这个函数我实现的比较简单，只是判断有没有找到，如果找到了返回成功即可，真正实现需要通过 getattr 获取文件属性，判断文件属性的 st_mode 中是否满足即可，例如+735 来说，就代表文件所有者可读可写可执行，文件所属组可写可执行，其他用户可读可执行

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。

对于 inode 指向的数据块，采用按需分配的方法实现，使得在仅仅使用一个数据块的情况下不必分配全部指向的四个数据块

```
int newfs_alloc_dentry(struct newfs_inode* inode, struct newfs_dentry* dentry) {
    if (inode->dentrys == NULL) {
        inode->dentrys = dentry;
    }
    else {
        dentry->brother = inode->dentrys;
        inode->dentrys = dentry;
    }
    inode->dir_cnt++;

    // 分配数据块
    int cur_blk = inode->dir_cnt / MAX_DENTRY_PER_BLK();
    if (inode->dir_cnt % MAX_DENTRY_PER_BLK() == 1) {
        /* 当前数据块已满，需要寻找新的数据块*/
        int byte_cursor = 0;
        int bit_cursor  = 0;
        int dno_cursor  = 0;
        boolean is_find_free_data_blk = FALSE;
        /* 检查位图是否有空位 */
        for (byte_cursor = 0; byte_cursor < NEWFS_BLKS_SZ(newfs_super.map_data_blks);
            byte_cursor++)
        {
            for (bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {
                if((newfs_super.map_data[byte_cursor] & (0x1 << bit_cursor)) == 0) {
                                                        /* 当前dno_cursor位置空闲 */
                    newfs_super.map_data[byte_cursor] |= (0x1 << bit_cursor);
                    //在指定位置的数据块指针中插入当前数据块
                    inode->block_pointer[cur_blk] = dno_cursor;
                    is_find_free_data_blk = TRUE;
                    break;
                }
```

在文件重命名时，并不是直接修改名称，而是创建一个新的同名文件，再修改下面的 inode 指针即可（其实是借鉴了 simplefs 的实现方法）

```
    boolean is_find, is_root;
    struct newfs_dentry* from_dentry = newfs_lookup(from, &is_find, &is_root);
    struct newfs_inode*  from_inode;
    struct newfs_dentry* to_dentry;
    mode_t mode = 0;
    if (is_find == FALSE) {
        return -NEWFS_ERROR_NOTFOUND;
    }

    if (strcmp(from, to) == 0) {
        return NEWFS_ERROR_NONE;
    }

    from_inode = from_dentry->inode;

    if (NEWFS_IS_DIR(from_inode)) {
        mode = S_IFDIR;
    }
    else if (NEWFS_IS_REG(from_inode)) {
        mode = S_IFREG;
    }

    ret = newfs_mknod(to, mode, NULL);
    if (ret != NEWFS_ERROR_NONE) {                  /* 保证目的文件不存在 */
        return ret;
    }

    to_dentry = newfs_lookup(to, &is_find, &is_root);
    newfs_drop_inode(to_dentry->inode);             /* 保证生成的inode被释放 */
    to_dentry->ino = from_inode->ino;               /* 指向新的inode */
    to_dentry->inode = from_inode;

    newfs_drop_dentry(from_dentry->parent->inode, from_dentry);
    return ret;
}
```

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。*

从 0 到 1 很难，但是从 0.5 到 1 就简单了很多，其实大部分 demo 都可以从 simplefs 中获得灵感，遇到最困难的问题还是读写函数的书写由于数据块以及边界的限制，调用函数拷贝时非常容易报错，这时需要画图知晓对应起始块以及最终块的边界以及写入大小，同时借鉴网上的优秀代码，得到最终的成品代码

另外再 dentry 与 inode 之间的指向关系时，一开始并不清楚，写到后面切记 dentry 里面的 inode 索引节点包含了子目录信息，这样在每次书写分配索引节点、分配目录项代码时，就能轻松很多，同时创建文件遵循从外到内的思想，删除目录遵循从内到外的思想，能够使得解决问题容易很多

简易操作截图：

```
220110426@comp2:~/user-land-filesystem/fs/newfs/tests$ ls
checkbm  fs_test.sh  main.sh  mnt  stages  test.sh
220110426@comp2:~/user-land-filesystem/fs/newfs/tests$ cd mnt/
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$ ls
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$ mkdir a
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$ touch b
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$ ls
a  b
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$ cd a
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt/a$ mkdir c
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt/a$ ls
c
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt/a$ cd ..
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$ ls
a  b
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$ vim d
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$ cat d
ymy
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$ ls
a  b  d
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$ rm a
rm: cannot remove 'a': Is a directory
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$ rm -r a
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$ ls
b  d
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$ rm b
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$ ls
d
220110426@comp2:~/user-land-filesystem/fs/newfs/tests/mnt$
```

测评截图：

```
8192+0 records in
8192+0 records out
4194304 bytes (4.2 MB, 4.0 MiB) copied, 0.0723793 s, 57.9 MB/s
pass: case 5.1 - umount /home/students/220110426/user-land-filesystem/fs/newfs/tests/mnt
pass: case 5.2 - remount /home/students/220110426/user-land-filesystem/fs/newfs/tests/mnt
8192+0 records in
8192+0 records out
4194304 bytes (4.2 MB, 4.0 MiB) copied, 0.0442367 s, 94.8 MB/s
pass: case 5.3 - check bitmap
8192+0 records in
8192+0 records out
4194304 bytes (4.2 MB, 4.0 MiB) copied, 0.059671 s, 70.3 MB/s
8192+0 records in
8192+0 records out
4194304 bytes (4.2 MB, 4.0 MiB) copied, 0.0558416 s, 75.1 MB/s
pass: case 6.1 - write /home/students/220110426/user-land-filesystem/fs/newfs/tests/mnt/file0
pass: case 6.2 - read /home/students/220110426/user-land-filesystem/fs/newfs/tests/mnt/file0
8192+0 records in
8192+0 records out
4194304 bytes (4.2 MB, 4.0 MiB) copied, 0.0484953 s, 86.5 MB/s
pass: case 7.1 - prepare content of /home/students/220110426/user-land-filesystem/fs/newfs/tests/mnt/file9
pass: case 7.2 - copy /home/students/220110426/user-land-filesystem/fs/newfs/tests/mnt/file9 to /home/students/220110426/user

Score: 34/34
pass: 恭喜你，通过所有测试 (34/34)
220110426@comp2:~/user-land-filesystem/fs/newfs/tests$
```

三、实验收获和建议

实验中的收获、感受、问题、建议等。

通过本次实验，让我对于文件系统的认识更加深刻，我不仅从零到一动手实现了一个可

以增删改查的文件系统（虽然大部分都是参考 simplefs 板子搞的），同时也加深了对于文件系统架构设计的认识，从简单的数据结构定义，到后面串联起目录项与索引节点的映射关系，使用位图来查看对应数据块地址的使用情况等，尽管中间很不知所措，但是好在先前有看过 mit 的文件系统视频，对于实验指导书的内容有一种亲切的感觉，而且其实文件系统设计的精髓就在于一开始的架构设计，参考 EXT2 实现了基础的架构设计，明白了各个部分的功能后，代码的书写只需要将对应的关系映射实现即可

四、参考资料

实验过程中查找的信息和资料

Simplefs 板子代码、EXT 文件系统相关文章、mit 官网视频及配套文档