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 5 班

学生学号：220110519

学生姓名：邢瑞龙

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

本次实验基于 EXT2 文件系统和 FUSE 架构且利用 ddriver 驱动完成基于 FUSE 的青春版 EXT2 文件系统，包含磁盘的挂载和卸载，创建文件和目录，显示文件和目录功能。

（1）**EXT2 系统：**

系统主要包含以下五个部分：

超级块 ：包含整个文件系统和磁盘布局的总体信息。

索引节点位图 ：记录着 索引节点 表的使用情况， 用 1 个比特 记录某一个索引节点是否被使用。

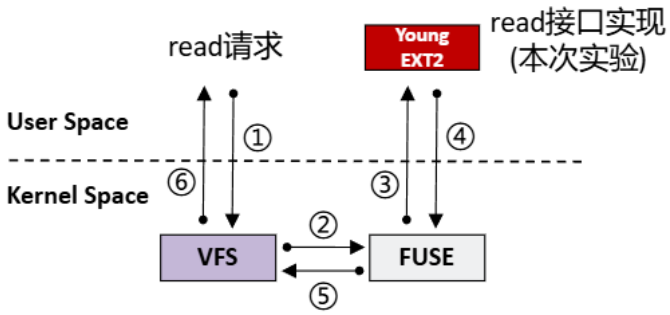
数据块位图 ：记录着 数据块 的使用情况， 用 1 个比特 记录某一个数据块是否被占用。

索引节点（inode）：记录着文件的元数据，每个文件都与一个 inode 对应。但一个 inode 可能对应多个文件（硬链接），我们实验不做考虑。

数据块 ：记录文件内容，数据块通常会被 inode 通过直接索引或者间接索引的方式找到。本次实验采用的是直接索引的方式。

（2）**FUSE 架构：**

FUSE 包括内核部分和用户态部分，FUSE 内核部分没有负责实现对应的接口，它会额外到注册到 FUSE 的用户态部分寻找对应的接口实现完成本次文件请求，最后将结果沿路返回。在图中红色的部分完成各种接口的实现 供 FUSE 内核调用。它将文件系统的实现从内核态搬到了用户态 ，从而我们可以在用户态实现一个文件系统 。



FUSE文件系统

（3）**DDRIVER 驱动**

本次实验在用户态下模拟了一个 容量为 4MB 的磁盘 ，并实现了对这个虚拟磁盘进行操作的 DDRIVER 驱动 。 DDRIVER 驱动，其原理是单次读取、写入这个数据文件时会按照固定的大小 512B 进行读取和写入，也就是 IO 大小为 512B ，从而来模拟达到磁盘操作。具体接口如下：

- int ddriver_open(char *path) ：

  根据路径 path 打开虚拟磁盘，返回文件描述符 fd。path 只能固定传入~/ddriver 的绝对路径。

- int ddriver_close(int fd)：

关闭打开的虚拟磁盘。

- int ddriver_seek(int fd, off_t offset, int whence)
  根据基准 whence 和偏移 offset 移动虚拟磁盘的磁盘头。whence 基准通常固定为 SEEK_SET，也就是 0；填写偏移 offset 即可。
- int ddriver_read(int fd, char *buf, size_t size)
  按照 IO 大小来从虚拟磁盘读取数据。size 只能固定传入 IO 大小。
- int ddriver_write(int fd, char *buf, size_t size)
  按照 IO 大小来往虚拟磁盘写入数据。size 只能固定传入 IO 大小。
- int ddriver_ioctl(int fd, unsigned long cmd, void *ret)
  根据传入命令 cmd 获取虚拟磁盘信息。cmd 为 IOC_REQ_DEVICE_SIZE，void *ret 处返回虚拟磁盘总大小。cmd 为 IOC_REQ_DEVICE_IO_SZ，void *ret 参数处返回虚拟磁盘的 IO 大小。

(4) 文件系统布局设计：

文件系统划分为 5 个部分：超级块，索引节点位图，数据位图，索引节点，数据块。

本次实验中，每个部分具体信息如下：

磁盘容量为 4MB，IO 块大小为 512B，逻辑块大小为两个 IO 块大小 1024B，逻辑块数总共有 4MB / 1024B = 4096 块。本次实验采用采用直接索引，每个文件最多直接索引 6 个逻辑块来填写文件数据，也就是每个文件数据上限是 6 * 1024B = 6KB。**本次实验中采取一个逻辑块中存储多个索引节点，避免浪费**。但为了估算文件数量，计算的时候按照最坏情况采用一个逻辑块存储一个索引节点。那么维护一个文件所需要的存储容量是 6KB + 1KB = 7KB。那么 4MB 磁盘，最多可以存放的文件数是 4MB / 7KB = 585。

- 超级块：一个逻辑块 1024B
- 索引节点位图：一个逻辑块 1024B 足以管理 8092 个索引节点远大于 585，足够
- 数据块位图：一个逻辑块 1024B 足以管理 8092 个数据块，足以覆盖整个文件系统，足够
- 索引节点：本次实验采取一个逻辑块存储多个索引节点的方式，固定为 16 个。故占用 585/16 向上取整共 37 个逻辑块

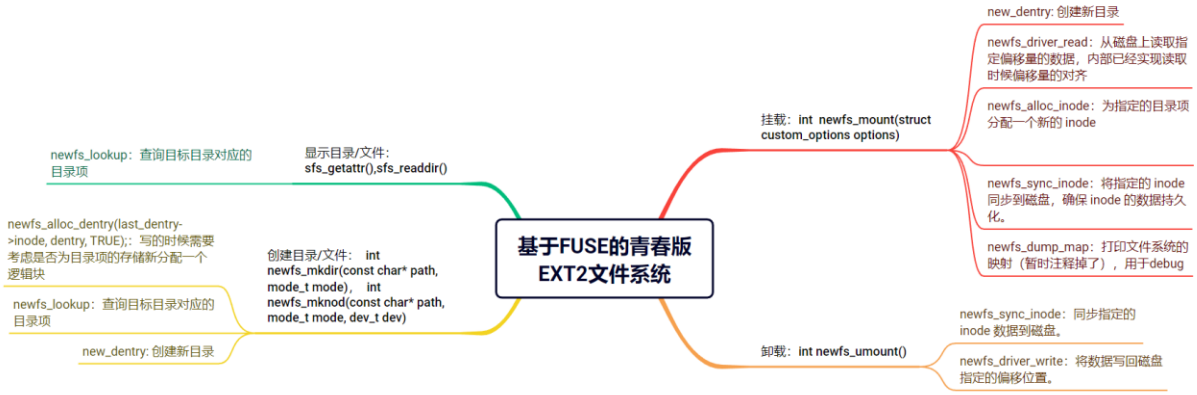
```
#define NEWFS_INODES_PER_BLK                16
```

- 数据块：4096 - 37 – 1 – 1 – 1 = 4056 块

填写 include/fs.layout 如下：

```
| BSIZE = 1024 B |
| Super(1) | Inode Map(1) | DATA Map(1) | INODE(37) | DATA(*) |
```

总体设计：



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）

2.1 数据结构：

（1）文件类型：

```
typedef enum newfs_file_type {
    NEWFS_REG_FILE,
    NEWFS_DIR,
    NEWFS_SYM_LINK
} NEWFS_FILE_TYPE;
```

- NEWFS_REG_FILE: 普通文件类型
- NEWFS_DIR: 目录项文件类型
- NEWFS_SYM_LINK:符号链接类型

(2) 超级块：

newfs_super_d 写回磁盘的结构：

```
struct newfs_super_d{

    uint32_t magic_num;             //幻数

    int blks_size;              // 逻辑块大小
    int io_size;                //IO大小
    int disk_size;              //磁盘大小
    int usage_size;             //已使用大小

    int sb_offset;              // 超级块于磁盘中的偏移，通常默认为0
    int sb_blks;                // 超级块于磁盘中的块数，通常默认为1

    int ino_map_offset;         //索引节点位图的偏移
    int ino_map_blks;           //索引节点位图占用逻辑块数量

    int data_map_offset;        //数据块位图偏移
    int data_map_blks;          //数据块位图占用逻辑块数量

    int ino_offset;             //索引节点的偏移
    int ino_blks;               //索引节点占用逻辑块数量

    int data_offset;            //数据块偏移
    int data_blks;              //数据块占用逻辑块数量

    int ino_max;                // 最大支持inode数
    int data_max;              //逻辑块块数
```

newfs_super 内存中的结构：

```
//超级块
struct newfs_super {
    uint32_t magic_num;             //幻数
    int driver_fd;                  //设备文件描述符

    boolean is_mounted;             //是否被挂载

    int blk_size;                   //逻辑块大小
    int io_size;                    //IO大小
    int disk_size;                  //磁盘大小
    int usage_size;                 //已使用大小

    int sb_offset;                  // 超级块于磁盘中的偏移，通常默认为0
    int sb_blks;                    // 超级块于磁盘中的块数，通常默认为1

    int ino_map_offset;             //索引节点位图的偏移
    int ino_map_blks;               //索引节点位图占用逻辑块数量
    uint8_t* ino_map;

    int data_map_offset;            //数据块位图偏移
    int data_map_blks;              //数据块位图占用逻辑块数量
    uint8_t* data_map;

    int ino_offset;                 //索引节点的偏移
    int ino_blks;                   //索引节点占用逻辑块数量

    int data_offset;                //数据块偏移
    int data_blks;                 //数据块占用逻辑块数量

    struct newfs_dentry* root_dentry; //根目录索引
    int ino_max;                    // 最大支持inode数
    int data_max;                  //逻辑块块数
};
```

（3）索引节点

newfs_inode_d 写回磁盘的结构：

```
struct newfs_inode_d {
    uint32_t           ino;                         // 在inode位图中的下标
    int                size;                        /* 文件已占用空间 */
    int                blk_pointers[NEWFS_DATA_PER_FILE];
    uint8_t*           data[NEWFS_DATA_PER_FILE];    /* 数据块指针（可固定分配）*/
    int                link;                        /* 链接数，默认为1 */
    char               target_path[NEWFS_MAX_FILE_NAME];/* store traget path when it is a symlink */
    NEWFS_FILE_TYPE    ftype;
    int                dir_cnt;                    // 如果是目录类型文件，下面有几个目录项
};
```

newfs_inode 内存中结构：

```
struct newfs_inode {
    uint32_t           ino;                        // 在inode位图中的下标
    int                size;                       /* 文件已占用空间 */
    int                blk_pointers[NEWFS_DATA_PER_FILE]; /*数据块地址指针*/
    uint8_t*           data[NEWFS_DATA_PER_FILE];     /* 数据块内容指针（可固定分配）*/
    int                link;                       /* 链接数，默认为1 */
    struct newfs_dentry* dentry;                     /* 指向该inode的目录dentrt或者文件dentry */
    struct newfs_dentry* dentrys;                    /* 如果是该inode是目录，dentrys指向其子目录的dentray链表的首个 */
    char               target_path[NEWFS_MAX_FILE_NAME];/* store traget path when it is a symlink */
    NEWFS_FILE_TYPE    ftype;
    int                dir_cnt;                   // 如果是目录类型文件，下面有几个目录项
};
```

（4）目录项

newfs_dentry_d 写回磁盘的结构：

```
struct newfs_dentry_d {
    char                fname[MAX_NAME_LEN];        /*该文件的名字*/
    uint32_t            ino;                        /*该目录项指向的ino节点*/
    NEWFS_FILE_TYPE     ftype;                      /*该目录文件或者普通文件*/
};
```

newfs_dentry 在内存中的结构：

```
struct newfs_dentry {
    char                fname[MAX_NAME_LEN];        /*该文件的名字*/
    uint32_t            ino;                        /*该目录项指向的ino节点*/
    NEWFS_FILE_TYPE     ftype;                      /*该目录文件或者普通文件*/
    struct newfs_dentry* parent;                    /* 父亲Inode的dentry */
    struct newfs_dentry* brother;                   /*同一级目录下的兄弟目录项*/
    struct newfs_inode*  inode;                     /*该目录项对应的inode*/
};
```

结构体中的成员含义注释已经标明

2.2 辅助函数

（1）计算 size 和逻辑块有关个数的简单函数：

```
/******************************************************************************/
#define NEWFS_IO_SZ()                   (newfs_super.io_size)
#define NEWFS_DISK_SZ()                 (newfs_super.disk_size)
#define NEWFS_DRIVER()                  (newfs_super.driver_fd)
#define NEWFS_BLK_SZ()                  (newfs_super.blk_size)
#define NEWFS_BLKS_SZ(blks)             ((blks) * NEWFS_BLK_SZ())


#define NEWFS_ROUND_DOWN(value, round)    ((value) % (round) == 0 ? (value) : ((value) / (round)) * (round))
#define NEWFS_ROUND_UP(value, round)      ((value) % (round) == 0 ? (value) : ((value) / (round) + 1) * (round))



#define NEWFS_ASSIGN_FNAME(psfs_dentry, _fname) \
(memcpy(psfs_dentry->fname, _fname, strlen(_fname)))

#define NEWFS_INODES_PER_BLK              16
#define NEWFS_INO_OFS(ino) \
    (newfs_super.ino_offset + ((ino) / NEWFS_INODES_PER_BLK) * NEWFS_BLK_SZ() + ((ino) % NEWFS_INODES_PER_BLK) * sizeof(struct newfs_inode_d))
#define NEWFS_DATA_OFS(ino)             (newfs_super.data_offset + NEWFS_BLKS_SZ(ino))

#define NEWFS_IS_DIR(pinode)            (pinode->dentry->ftype == NEWFS_DIR)
#define NEWFS_IS_REG(pinode)            (pinode->dentry->ftype == NEWFS_REG_FILE)
#define NEWFS_IS_SYM_LINK(pinode)       (pinode->dentry->ftype == NEWFS_SYM_LINK)
#define NEWFS_DENTRYS_PER_BLK           (NEWFS_BLK_SZ() / sizeof(struct newfs_dentry_d))
```

- NEWFS_IO_SZ()：返回文件系统的 I/O 大小（newfs_super.io_size）。
- NEWFS_DISK_SZ()：返回文件系统的磁盘大小（newfs_super.disk_size）
- NEWFS_DRIVER()：返回文件系统的驱动文件描述符（newfs_super.driver_fd）
- NEWFS_BLK_SZ()：返回文件系统的块大小（newfs_super.blk_size）
- NEWFS_BLKS_SZ(blks)：计算给定块数 blks 的总字节大小。公式为：(blks) * NEWFS_BLK_SZ()。
- NEWFS_ROUND_DOWN(value, round)：将 value 向下舍入到最接近的 round 倍数
- NEWFS_ROUND_UP(value, round)：将 value 向上舍入到最接近的 round 倍数。

- NEWFS_ASSIGN_FNAME(psfs_dentry, _fname):将字符串 _fname 复制到 psfs_dentry->fname 中。
- NEWFS_INODES_PER_BLK:每个块中的 inode 数量，固定为 16。
- NEWFS_INO_OFS(ino):计算给定 inode ino 的偏移量。公式为:newfs_super.ino_offset + ((ino) / NEWFS_INODES_PER_BLK) * NEWFS_BLK_SZ() + ((ino) % NEWFS_INODES_PER_BLK) * sizeof(struct newfs_inode_d)。
- NEWFS_DATA_OFS(ino):计算给定 inode ino 的数据块偏移量。公式为:newfs_super.data_offset + NEWFS_BLKS_SZ(ino)。
- NEWFS_IS_DIR(pinode):检查 pinode 是否是一个目录。通过检查其目录项类型是否为 NEWFS_DIR 来判断。
- NEWFS_IS_REG(pinode):检查 pinode 是否是一个常规文件。通过检查其目录项类型是否为 NEWFS_REG_FILE 来判断。
- NEWFS_IS_SYM_LINK(pinode):检查 pinode 是否是一个符号链接。通过检查其目录项类型是否为 NEWFS_SYM_LINK 来判断。
- NEWFS_DENTRYS_PER_BLK:每个块中可以存储的目录项数量，计算公式为:NEWFS_BLK_SZ() / sizeof(struct newfs_dentry_d)。

(2) 相关常量宏定义：

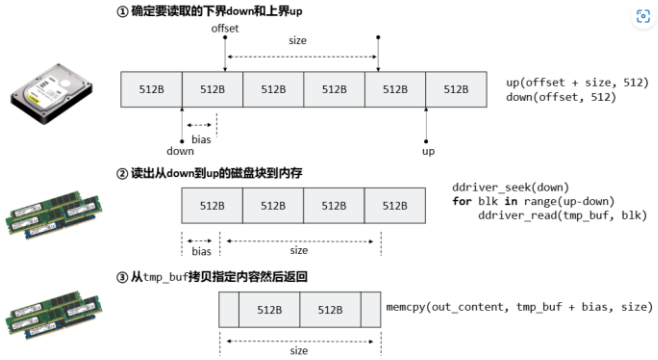
```
#define TRUE                    1
#define FALSE                   0
#define UINT32_BITS             32
#define UINT8_BITS              8

#define NEWFS_SUPER_OFS           0
#define NEWFS_ROOT_INO            0



#define NEWFS_ERROR_NONE          0
#define NEWFS_ERROR_ACCESS        EACCES
#define NEWFS_ERROR_SEEK          ESPIPE
#define NEWFS_ERROR_ISDIR         EISDIR
#define NEWFS_ERROR_NOSPACE       ENOSPC
#define NEWFS_ERROR_EXISTS        EEXIST
#define NEWFS_ERROR_NOTFOUND      ENOENT
#define NEWFS_ERROR_UNSUPPORTED   ENXIO
#define NEWFS_ERROR_IO            EIO     /* Error Input/Output */
#define NEWFS_ERROR_INVAL         EINVAL  /* Invalid Args */

#define NEWFS_MAX_FILE_NAME       128
#define NEWFS_INODE_PER_FILE      1
#define NEWFS_DATA_PER_FILE       6
#define NEWFS_DEFAULT_PERM        0777

#define NEWFS_IOC_MAGIC           'S'
#define NEWFS_IOC_SEEK            _IO(NEWFS_IOC_MAGIC, 0)

#define NEWFS_FLAG_BUF_DIRTY      0x1
#define NEWFS_FLAG_BUF_OCCUPY     0x2
```

（3）为文件系统主要功能提供辅助的函数：

1) int newfs_driver_read(int offset, uint8_t *out_content, int size)：

功能：从设备驱动指定偏移量，长度的字节读出数据。

```
int newfs_driver_read(int offset, uint8_t *out_content, int size) {
    int      offset_aligned = NEWFS_ROUND_DOWN(offset, NEWFS_IO_SZ());
    int      bias           = offset - offset_aligned;
    int      size_aligned   = NEWFS_ROUND_UP((size + bias), NEWFS_IO_SZ());
    uint8_t* temp_content   = (uint8_t*)malloc(size_aligned);
    uint8_t* cur            = temp_content;
    // lseek(NEWFS_DRIVER(), offset_aligned, SEEK_SET);
    ddriver_seek(NEWFS_DRIVER(), offset_aligned, SEEK_SET);
    while (size_aligned != 0)
    {
        // read(NEWFS_DRIVER(), cur, NEWFS_IO_SZ());
        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;
}
```



它从设备驱动读取数据，并将读取的内容存储到指定的缓冲区。首先通过对齐操作确保读取在 I/O 边界上对齐，分配临时缓冲区存储数据，然后从设备驱动按块读取数据，最终将读取的数据复制到目标缓冲区并释放临时资源。

2) int newfs_driver_write(int offset, uint8_t *in_content, int size):

功能：将指定偏移量，长度的字节数据写入到设备驱动。实现逻辑与上述 read 类似。

```
int newfs_driver_write(int offset, uint8_t *in_content, int size) {
    int      offset_aligned = NEWFS_ROUND_DOWN(offset, NEWFS_IO_SZ());
    int      bias           = offset - offset_aligned;
    int      size_aligned   = NEWFS_ROUND_UP((size + bias), NEWFS_IO_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);

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

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

3) int newfs_alloc_dentry(struct newfs_inode* inode, struct newfs_dentry* dentry, boolean judge)：

功能：采用头插法，将目录项插入父目录项对应链表的表头 inode->dentrys

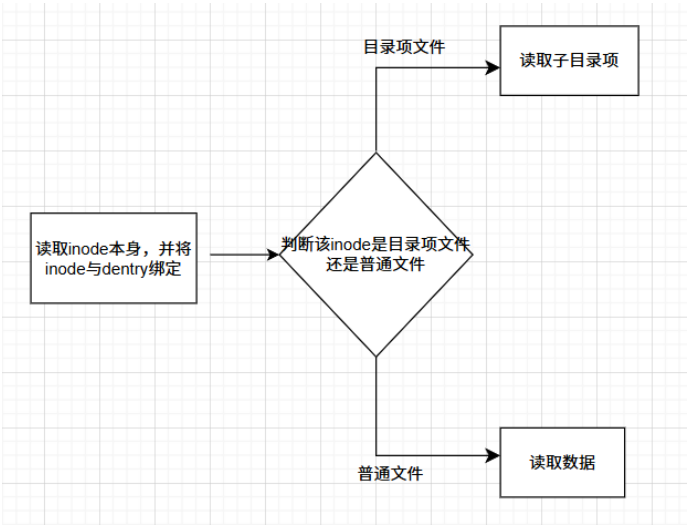
```
int newfs_alloc_dentry(struct newfs_inode* inode, struct newfs_dentry* dentry, boolean judge) {
    if (inode->dentrys == NULL) {
        inode->dentrys = dentry;
    }
    else {
        dentry->brother = inode->dentrys;          //原来的链表头目录变为当前目录的兄弟
        inode->dentrys = dentry;
    }
    inode->dir_cnt++;
    if(judge){
        /* 检查位图是否有空位 */
        if((inode->dir_cnt % NEWFS_DENTRYS_PER_BLK) == 1){ //需要找到新的逻辑块来存
            int byte_cursor = 0;
            int bit_cursor  = 0;
            int data_cursor  = 0;
            boolean is_find_free_entry = FALSE;
            for (byte_cursor = 0; byte_cursor < NEWFS_BLKS_SZ(newfs_super.data_map_blks); //字节位
                byte_cursor++)
            {
                for (bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {          //比特位
                    if((newfs_super.data_map[byte_cursor] & (0x1 << bit_cursor)) == 0) {
                                                        /* 当前data_cursor位置空闲 */
                        newfs_super.data_map[byte_cursor] |= (0x1 << bit_cursor);
                        is_find_free_entry = TRUE;
                        break;
                    }
                    data_cursor++;
                }
                if (is_find_free_entry) {
                    break;
                }
            }
            if (!is_find_free_entry || data_cursor == newfs_super.ino_max)
                return -NEWFS_ERROR_NOSPACE;
            /*这里只是为了记录数据块是否被占用*/
            int cur_blk = inode->dir_cnt / NEWFS_DENTRYS_PER_BLK;
            inode->blk_pointers[cur_blk] = data_cursor;
        }

    }
    return inode->dir_cnt;
}
```

由于一个逻辑块存储多个目录项，逻辑块的存储需要动态分配

参数 judge 用于判断此时是在挂载时读取的操作还是创建目录文件时的操作。

- 如果是读操作（judge==0），则不需要为 dentry 动态分配逻辑块。
- 如果是写操作（judge==1），则需要判断此时对应分配的逻辑块是否已被装满，如果装满则需要寻找一块空闲的逻辑块用于存储 dentry。最后空闲逻辑块的位置存入对应的 blk_pointer（**动态分配**）
    - 具体判断是否当前逻辑块是否被装满的逻辑便是如果当前索引节点对应的 dir_cnt 模 NEWFS_DENTRYS_PER_BLK == 1,则说明需要开辟一个新的逻辑块。

4) struct newfs_inode* newfs_read_inode(struct newfs_dentry * dentry, int ino)：
功能：读取目录项对应的索引节点




```
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, i;
    /* 从磁盘读索引结点 */
    if (newfs_driver_read(NEWFS_INO_OFS(ino), (uint8_t *)&inode_d,
                          sizeof(struct newfs_inode_d)) != 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;
    memcpy(inode->target_path, inode_d.target_path, NEWFS_MAX_FILE_NAME);
    inode->dentry = dentry;
    inode->dentrys = NULL;
    for(int i = 0;i <NEWFS_DATA_PER_FILE; i++){
        inode->blk_pointers[i] = inode_d.blk_pointers[i];
    }
```

```
    if (NEWFS_IS_DIR(inode)) {
        printf("READ INODE\n");
        printf("root ino:%d, name%s\n", inode->ino, dentry->fname);
        printf("root inode offset:%d\n", NEWFS_INO_OFS(ino));
        dir_cnt = inode_d.dir_cnt;
        int blk_num = 0;
        int offset;
        while(dir_cnt > 0 && blk_num < NEWFS_DATA_PER_FILE){
            offset = NEWFS_DATA_OFS(inode->blk_pointers[blk_num]);
            printf("    origin offset:%d\n", offset);
            while(dir_cnt > 0 && offset + sizeof(struct newfs_dentry_d) < NEWFS_DATA_OFS(inode->blk_pointers[blk_num] + 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;
                printf("    child ino:%d, child_fname%s, child dentry offset:%d\n",dentry_d.ino, dentry_d.fname, offset);
                newfs_alloc_dentry(inode, sub_dentry, FALSE); //读的时候不需要判断是否需要额外分配逻辑块给dentry
                dir_cnt --;
                offset += sizeof(struct newfs_dentry_d);
            }
            blk_num +=1;
        }
    }
```

```
    else if (NEWFS_IS_REG(inode)) {
        for(int i = 0;i <NEWFS_DATA_PER_FILE; i++){
            if(inode->blk_pointers[i] == -1) continue; //如果尚未分配，直接跳过
            inode->data[i] = (uint8_t *)malloc(NEWFS_BLK_SZ());
            if (newfs_driver_read(NEWFS_DATA_OFS(inode->blk_pointers[i]), (uint8_t *)inode->data[i], NEWFS_BLK_SZ()) != NEWFS_ERROR_NONE) {
                NEWFS_DBG("[%s] io error\n", __func__);
                return NULL;
            }
            printf("    read file:%s, offset:%d\n", inode->dentry->fname, NEWFS_DATA_OFS(inode->blk_pointers[i]));
        }
    }
    return inode;
}
```

5）struct newfs_inode* newfs_alloc_inode(struct newfs_dentry * dentry)：

功能：为目录项分配 inode

遍历 ino_map 索引节点位图，寻找空闲的索引节点分配给 dentry。

```
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.ino_map_blks); //字节位
         byte_cursor++)
    {
        for (bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {          //比特位
            if((newfs_super.ino_map[byte_cursor] & (0x1 << bit_cursor)) == 0) {
                                                    /* 当前ino_cursor位置空闲 */
                newfs_super.ino_map[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.ino_max){
        printf("分配失败！！\n");
        return -NEWFS_ERROR_NOSPACE;
    }
```

分配完成后初始化 inode，将其与 denry 绑定，子目录项个数初始化为 0。dentry 指向 inode，并将 ino 赋值。如果 inode 是普通文件类型分配 data 空间并且将 blk_pointers 都初始化为-1，代表尚未分配实际的逻辑块。

```
    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指回dentry */
    inode->dentry = dentry;
    inode->dir_cnt = 0;
    inode->dentrys = NULL;

    if (NEWFS_IS_REG(inode)) {
        for(int i = 0;i < NEWFS_DATA_PER_FILE; i++){
            inode->data[i] = (uint8_t *)malloc(NEWFS_BLK_SZ());
            inode->blk_pointers[i] = -1;
        }

    }
    printf("分配成功！！\n");
    return inode;
}
```

6）int newfs_sync_inode(struct newfs_inode * inode)：
功能：将内存 inode 及其下方结构全部刷回磁盘

先写 inode 本身：

```
int newfs_sync_inode(struct newfs_inode * inode) {
    struct newfs_inode_d  inode_d;
    struct newfs_dentry*  dentry_cursor;
    struct newfs_dentry_d dentry_d;
    int ino             = inode->ino;
    inode_d.ino         = ino;
    inode_d.size        = inode->size;
    memcpy(inode_d.target_path, inode->target_path, NEWFS_MAX_FILE_NAME);
    inode_d.ftype       = inode->dentry->ftype;
    inode_d.dir_cnt     = inode->dir_cnt;
    int offset;
    for(int i=0; i< NEWFS_DATA_PER_FILE; i++){
        inode_d.blk_pointers[i] = inode->blk_pointers[i];
    }
    /* 先写inode本身 */
    if (newfs_driver_write(NEWFS_INO_OFS(ino), (uint8_t *)&inode_d,
                           sizeof(struct newfs_inode_d)) != NEWFS_ERROR_NONE) {
        NEWFS_DBG("[%s] io error\n", __func__);
        return -NEWFS_ERROR_IO;
    }
```

之后再写 inode 下方的数据：

判断 inode 如果是目录项，则写完子目录项后继续递归。
遍历每个 blk_pointer[i]，同一个逻辑块内存储的目录项地址连续，offset 每次+=sizeof(struct newfs_dentry_d)。最内层循环写了 NEWFS_DENTRYS_PER_BLK 个后，切换到下一个 blk_pointer[i+1],并将 offset 重置为该逻辑块的初始地址。

```
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 */
    printf("START WRITE TO DISK\n");
    printf("root ino:%d\n", ino);
    printf("root inode offset:%d\n", NEWFS_INO_OFS(ino));
    printf("root dir_cnt:%d\n",inode_d.dir_cnt);
    printf("root fname:%s\n", inode->dentry->fname);
    printf("root type:%d\n", inode->dentry->ftype);
    int blk_num = 0;
    dentry_cursor = inode->dentrys;
    while(dentry_cursor != NULL && blk_num < NEWFS_DATA_PER_FILE){
        offset = NEWFS_DATA_OFS(inode->blk_pointers[blk_num]);
        printf("    origin offset:%d\n", offset);
        while (dentry_cursor != NULL && offset + sizeof(struct newfs_dentry_d) < NEWFS_DATA_OFS(inode->blk_pointers[blk_num] + 1))
        {
            memcpy(dentry_d.fname, dentry_cursor->fname, NEWFS_MAX_FILE_NAME);
            dentry_d.ftype = dentry_cursor->ftype;
            dentry_d.ino = dentry_cursor->ino;
            printf("    child ino:%d, child fname %s, child dentry offset:%d\n",dentry_d.ino, dentry_d.fname, offset);
            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;
            }

            if (dentry_cursor->inode != NULL) {
                newfs_sync_inode(dentry_cursor->inode);
            }

            dentry_cursor = dentry_cursor->brother;
            offset += sizeof(struct newfs_dentry_d);
        }
        blk_num += 1;
    }
}
```

判断如果 inode 是普通文件，则直接写入数据即可

```
    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->blk_pointers[i] == -1) continue; //如果尚未分配，直接跳过
            if (newfs_driver_write(NEWFS_DATA_OFS(inode->blk_pointers[i]), inode->data[i], NEWFS_BLK_SZ()) != NEWFS_ERROR_NONE) {
                NEWFS_DBG("[%s] io error\n", __func__);
                return -NEWFS_ERROR_IO;
            }
            printf("    write file:%s, offset:%d\n", inode->dentry->fname, NEWFS_DATA_OFS(inode->blk_pointers[i]));

        }
    }
    return NEWFS_ERROR_NONE;
}
```

7）int newfs_calc_lvl(const char * path)
功能：计算路径的层级（路径中'\'的个数）

```c
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;
}
```

8）char* newfs_get_fname(const char* path)
功能：获取文件名

```c
char* newfs_get_fname(const char* path) {
    char ch = '/';
    char *q = strrchr(path, ch) + 1;
    return q;
}
```

9）struct newfs_dentry* newfs_lookup(const char * path, boolean* is_find, boolean* is_root)
功能：从根目录节点开始往下寻找，寻找 path 对应的目录项。
如果找不到，则返回最后一次有效的目录项，is_find 置为 FALSE。
如果找到，返回对应的目录项，is_find 置为 TRUE

如果要找的是根目录，则直接返回根目录项即可，is_find 和 is_root 都置为 TRUE。

```c
struct newfs_dentry* newfs_lookup(const char * path, boolean* is_find, boolean* is_root){
    int total_lvl = newfs_calc_lvl(path);
    struct newfs_dentry* dentry_cursor = newfs_super.root_dentry;
    struct newfs_dentry* dentry_ret = NULL;
    struct newfs_inode*  inode;
    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;
    }
    printf("root_dentry ino: %d\n",dentry_cursor->inode->ino);
    printf("root_dentry fist child:%s\n", dentry_cursor->inode->dentrys->fname);
```

如果不是，则按照 path 一层一层地找。
如果当前 inode 是普通文件类型，但层数不是 path 的最后一层，则打印该 inode 不是目录项文件的信息，退出循环。
如果当前 inode 是目录项文件，fname 存储当前层对应的文件名。如果当前层遍历所有子目录项都没有名字与 fname 相同的目录项，则将返回值赋值为当前成功的父目录项的 dentry 并且打印没有找到的信息，退出整个循环。

```
fname = strtok(path_cpy, "/");
while (fname)
{
    lvl++;
    if (dentry_cursor->inode == NULL) {           /* Cache机制 */
        newfs_read_inode(dentry_cursor, dentry_cursor->ino);
    }

    inode = dentry_cursor->inode;

    if (NEWFS_IS_REG(inode) && lvl < total_lvl) { /*如果该文件为普通文件但却不在路径的末尾，则报错*/
        NEWFS_DBG("[%s] not a dir\n", __func__);
        dentry_ret = inode->dentry;
        break;
    }
    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;
            }
            printf("%s\n", dentry_cursor->fname);
            dentry_cursor = dentry_cursor->brother;
        }

        if (!is_hit) {
            *is_find = FALSE;
            NEWFS_DBG("[%s] not found %s\n", __func__, fname);
            dentry_ret = inode->dentry;
            break;
        }

        if (is_hit && lvl == total_lvl) {
            *is_find = TRUE;
            dentry_ret = dentry_cursor;
            break;
        }
    }
    fname = strtok(NULL, "/");
}
```

10）struct newfs_dentry* newfs_get_dentry(struct newfs_inode * inode, int dir)：
功能：返回子目录链表中的第 dir 个位置的子目录项。

```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;
}
```

11）int　newfs_mount(struct custom_options options):
功能：挂载文件系统

获取驱动程序的文件标识符，并获取设备的磁盘大小与 IO 块大小，逻辑块大小为 IO 块的两倍

```c
  int                 ret = NEWFS_ERROR_NONE;
  int                 driver_fd;
  struct newfs_super_d  newfs_super_d;
  struct newfs_dentry*  root_dentry;
  struct newfs_inode*   root_inode;
  int                 inode_num;
  int                 inode_blks;
  int                 ino_map_blks;

  int                 super_blks;
  boolean             is_init = FALSE;

  newfs_super.is_mounted = FALSE;

  driver_fd = ddriver_open(newfs_options.device);
   if (driver_fd < 0) {
      return driver_fd;
  }
  newfs_super.driver_fd = driver_fd;

  ddriver_ioctl(NEWFS_DRIVER(), IOC_REQ_DEVICE_SIZE, &newfs_super.disk_size);
  ddriver_ioctl(NEWFS_DRIVER(), IOC_REQ_DEVICE_IO_SZ, &newfs_super.io_size);
  newfs_super.blk_size = 2 * NEWFS_IO_SZ();
```

创建根目录：由于根目录不存在父目录，因此根目录的 dentry 比较特殊，不会保存在磁盘中，不会从父目录的数据块读出，而是每次在挂载文件系统时，为根目录新创建一个 dentry，然后再和根目录的 inode 关联起来。

```c
root_dentry = new_dentry("/", NEWFS_DIR);

if(newfs_driver_read(NEWFS_SUPER_OFS,  (uint8_t *)(&newfs_super_d), sizeof(struct newfs_super_d)) != NEWFS_ERROR_NONE){
    return -NEWFS_ERROR_IO;
}
```

如果当前超级块对应的幻数不是文件系统的 magic_num，则是第一次挂载，需要初始化布局设计相关信息。

```
if(newfs_super_d.magic_num != NEWFS_MAGIC_NUM){
    super_blks = NEWFS_ROUND_UP(sizeof(struct newfs_super_d), NEWFS_BLK_SZ())/ NEWFS_BLK_SZ();
    inode_num = NEWFS_DISK_SZ()/((NEWFS_DATA_PER_FILE + NEWFS_INODE_PER_FILE) * NEWFS_BLK_SZ());

    inode_blks = NEWFS_ROUND_UP(inode_num, NEWFS_INODES_PER_BLK) /NEWFS_INODES_PER_BLK;
    ino_map_blks = NEWFS_ROUND_UP(NEWFS_ROUND_UP(inode_blks, UINT32_BITS), NEWFS_BLK_SZ())/NEWFS_BLK_SZ();

    newfs_super_d.sb_offset = NEWFS_SUPER_OFS;
    newfs_super_d.sb_blks = super_blks;

    newfs_super_d.ino_map_offset = NEWFS_SUPER_OFS + NEWFS_BLKS_SZ(newfs_super_d.sb_blks);
    newfs_super_d.ino_map_blks = ino_map_blks;

    newfs_super_d.data_map_offset = newfs_super_d.ino_map_offset + NEWFS_BLKS_SZ(newfs_super_d.ino_map_blks);
    newfs_super_d.data_map_blks = 1;

    newfs_super_d.ino_offset = newfs_super_d.data_map_offset + NEWFS_BLKS_SZ(newfs_super_d.data_map_blks);
    newfs_super_d.ino_blks = inode_blks;

    newfs_super.ino_max = newfs_super_d.ino_blks;

    newfs_super_d.data_offset =newfs_super_d.ino_offset + NEWFS_BLKS_SZ(newfs_super_d.ino_blks);
    newfs_super_d.data_blks = NEWFS_ROUND_DOWN(NEWFS_DISK_SZ(), NEWFS_BLK_SZ())/NEWFS_BLK_SZ() - newfs_super_d.sb_blks - newfs_super_d.ino_map_blks - newfs_super_d.data_map_blks - newfs_super_d.ino_blks;
    newfs_super_d.usage_size = 0;

    newfs_super.data_max =  newfs_super_d.data_blks;
    NEWFS_DBG("inode map blocks: %d\n", ino_map_blks);
    NEWFS_DBG("disk_size: %d\n", newfs_super_d.disk_size);
    NEWFS_DBG("inode_num: %d\n", inode_num);
    NEWFS_DBG("inode_blks: %d\n", newfs_super_d.ino_blks);
    NEWFS_DBG("ino_map_blks: %d\n", newfs_super_d.ino_map_blks);
    NEWFS_DBG("ino_map_offset: %d\n", newfs_super_d.ino_map_offset);
    NEWFS_DBG("data_map_offset: %d\n", newfs_super_d.data_map_offset);
    NEWFS_DBG("ino_offset: %d\n", newfs_super_d.ino_offset);
    NEWFS_DBG("data_offset: %d\n", newfs_super_d.data_offset);
    is_init = TRUE;
}
```

将磁盘的超级块存储的布局信息读入内存中。

```
newfs_super.usage_size = newfs_super_d.usage_size;
/*超级块建立*/
newfs_super.sb_blks = newfs_super_d.sb_blks;
newfs_super.sb_offset = newfs_super_d.sb_offset;

/*索引位图建立*/
newfs_super.ino_map_blks = newfs_super_d.ino_map_blks;
newfs_super.ino_map_offset = newfs_super_d.ino_map_offset;
newfs_super.ino_map = (uint8_t *)malloc(NEWFS_BLKS_SZ(newfs_super_d.ino_map_blks));
/*索引节点建立*/
newfs_super.ino_blks = newfs_super_d.ino_blks;
newfs_super.ino_offset = newfs_super_d.ino_offset;
/*数据位图建立*/
newfs_super.data_map_blks = newfs_super_d.data_map_blks;
newfs_super.data_map_offset = newfs_super_d.data_map_offset;
newfs_super.data_map = (uint8_t *)malloc(NEWFS_BLKS_SZ(newfs_super_d.data_map_blks));
/*数据块建立*/
newfs_super.data_blks =  newfs_super_d.data_blks;
newfs_super.data_offset = newfs_super_d.data_offset;
```

将磁盘超级块的索引位图，数据位图读入内存中。

```
if (newfs_driver_read(newfs_super_d.ino_map_offset, (uint8_t *)(newfs_super.ino_map),
                      NEWFS_BLKS_SZ(newfs_super_d.ino_map_blks)) != NEWFS_ERROR_NONE) {
    return -NEWFS_ERROR_IO;
}

if (newfs_driver_read(newfs_super_d.data_map_offset, (uint8_t *)(newfs_super.data_map),
                      NEWFS_BLKS_SZ(newfs_super_d.data_map_blks)) != NEWFS_ERROR_NONE) {
    return -NEWFS_ERROR_IO;
}
```

分配根节点的 inode 并从磁盘中读取，将超级块中的 root_dentry 指针指向根目录项 dentry，。最后挂载成功，is_mounted 置为 TRUE。

```
if (is_init) {                                  /* 分配根节点 */
    root_inode = newfs_alloc_inode(root_dentry);
    newfs_sync_inode(root_inode);
}

root_inode            = newfs_read_inode(root_dentry, NEWFS_ROOT_INO);  /* 读取根目录 */
root_dentry->inode    = root_inode;
newfs_super.root_dentry = root_dentry;
newfs_super.is_mounted  = TRUE;
printf("FINISHED READING\n");

// newfs_dump_map();
return ret;
```

12）int newfs_umount()
功能：卸载文件系统
如果当前文件系统是没有挂载的，直接退出。

将内存的超级块信息赋值给写入磁盘的数据结构 newfs_super_d.使用 newfs_driver_write 写回磁盘。

```
if (!newfs_super.is_mounted) {
    return NEWFS_ERROR_NONE;
}
printf("START UNMOUNT!!!\n");

newfs_sync_inode(newfs_super.root_dentry->inode);     /* 从根节点向下刷写节点 */

newfs_super_d.magic_num          = NEWFS_MAGIC_NUM;
newfs_super_d.usage_size = newfs_super.usage_size;
/*超级块信息写回*/
newfs_super_d.sb_blks = newfs_super.sb_blks;
newfs_super_d.sb_offset = newfs_super.sb_offset;

/*索引位图信息写回*/
newfs_super_d.ino_map_blks = newfs_super.ino_map_blks;
newfs_super_d.ino_map_offset = newfs_super.ino_map_offset;

/*索引节点信息写回*/
newfs_super_d.ino_blks = newfs_super.ino_blks;
newfs_super_d.ino_offset = newfs_super.ino_offset;
/*数据位图信息写回*/
newfs_super_d.data_map_blks = newfs_super.data_map_blks;
newfs_super_d.data_map_offset = newfs_super.data_map_offset;
/*数据块信息写回*/
newfs_super_d.data_blks =  newfs_super.data_blks;
newfs_super_d.data_offset = newfs_super.data_offset;

/*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;
}
```

将索引位图，数据位图写回磁盘。并释放再内存中的空间。最后关闭驱动。

```
    /*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;
    }
    /*索引位图写回磁盘*/
    if (newfs_driver_write(newfs_super_d.ino_map_offset, (uint8_t *)(newfs_super.ino_map),
                           NEWFS_BLKS_SZ(newfs_super_d.ino_map_blks)) != NEWFS_ERROR_NONE) {
        return -NEWFS_ERROR_IO;
    }

    /*数据位图写回磁盘*/
    if (newfs_driver_write(newfs_super_d.data_map_offset, (uint8_t *)(newfs_super.data_map),
                           NEWFS_BLKS_SZ(newfs_super_d.data_map_blks)) != NEWFS_ERROR_NONE) {
        return -NEWFS_ERROR_IO;
    }

    free(newfs_super.ino_map);
    free(newfs_super.data_map);
    /*关闭驱动*/
    ddriver_close(NEWFS_DRIVER());
    printf("FINISH UNMOUNT!!!\n");

    return NEWFS_ERROR_NONE;
}
```

2.3 功能函数

（1）挂载钩子函数：newfs_init

直接使用写好的 newfs_mount 即可。

```
void* newfs_init(struct fuse_conn_info * conn_info) {
    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;
}
```

(2) 卸载钩子函数：newfs_destroy

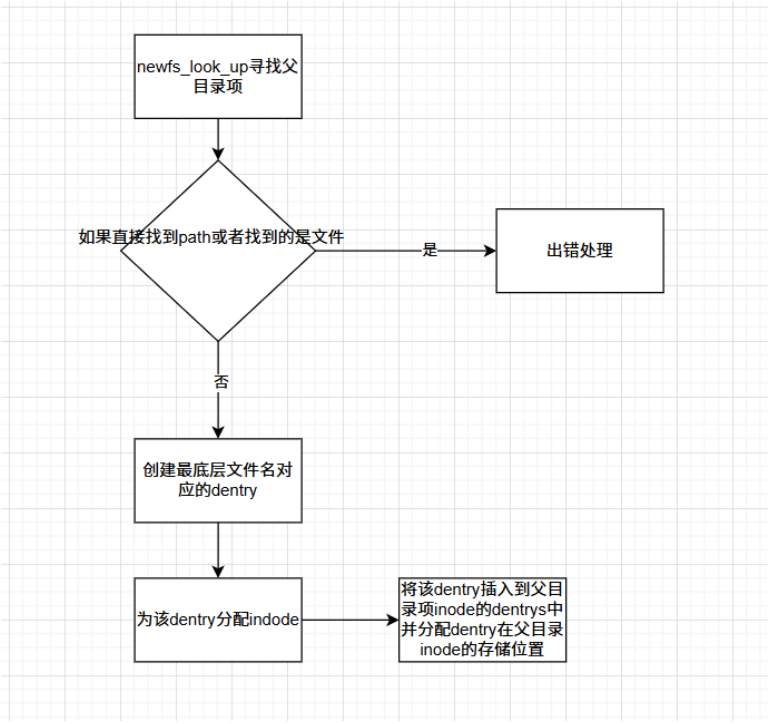
直接使用写好的 newfs_destroy 即可。

```
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;
}
```

（3）创建目录：newfs_mkdir

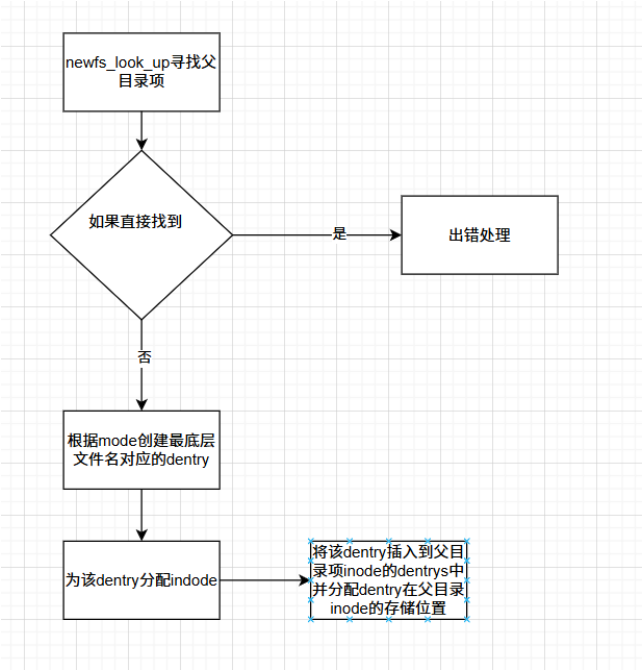
使用 newfs_lookup 寻找，如果直接找到或找到的是一个普通文件，返回错误。

否则在找到的目录项下，创建一个子目录项插入其中，并分配新的 inode。之后为 dentry 分配一个存储位置。此处需要考虑存储 dentry 逻辑块的动态分配




```
int newfs_mkdir(const char* path, mode_t mode) {
    /* TODO: 解析路径，创建目录 */
    boolean is_find, is_root;
    struct newfs_dentry* last_dentry = newfs_lookup(path, &is_find, &is_root); /*如果不到的会返回最深且有效的一层*/
    char* fname;
    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); //为该目录项分配一个索引来存储该目录项的所有子目录项
    newfs_alloc_dentry(last_dentry->inode, dentry, TRUE); ////写的时候需要考虑是否新分配一个逻辑块
    printf("Mkdir:\n");
    printf("Father ino: %d\n", last_dentry->ino);
    printf("    child ino: %d\n", last_dentry->inode->dentrys->ino);
    printf("    child fype: %d\n", dentry->ftype);
    printf("    inode:ino:%d\n", inode->ino);
    printf("    inode if is dir%d\n", NEWFS_IS_DIR(inode));
    return NEWFS_ERROR_NONE;
}
```

（4）创建文件：newfs_mknod
与创建目录类似。




```
int newfs_mknod(const char* path, mode_t mode, dev_t dev) {
    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, TRUE); //写的时候需要考虑是否新分配一个逻辑块
    printf("Touch:\n");
    printf("Father ino: %d\n", last_dentry->ino);
    printf("    child ino: %d\n", last_dentry->inode->dentrys->ino);
    printf("    child fype: %d\n", dentry->ftype);
    printf("    inode:ino:%d\n", inode->ino);
    printf("    inode if is REG%d\n", NEWFS_IS_REG(inode));

    return NEWFS_ERROR_NONE;
}
```

（5）显示文件和目录：newfs_getattr 和 newfs_readdir

int newfs_getattr(const char* path, struct stat * newfs_stat):获取文件属性

```c
int newfs_getattr(const char* path, struct stat * newfs_stat) {
    /* TODO: 解析路径，获取Inode，填充newfs_stat，可参考/fs/simplefs/sfs.c的sfs_getattr()函数实现 */
    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);
    }
    else if (NEWFS_IS_REG(dentry->inode)) {
        newfs_stat->st_mode = S_IFREG | NEWFS_DEFAULT_PERM;
        newfs_stat->st_size = dentry->inode->size;
    }
    else if (NEWFS_IS_SYM_LINK(dentry->inode)) {
        newfs_stat->st_mode = S_IFLNK | 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();

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

- st_size: 文件的大小
- st_mode: 文件的模式，包括文件的权限、文件类型
- st_nlink: 文件的连接数，即多少硬链接
- st_uid: 文件所有者的用户 ID
- st_gid: 文件所有者的组 ID
- st_blksize: 文件所在文件系统的逻辑块大小
- st_blocks: 文件所占的块数
- st_atime: 文件最近一次访问时间
- st_mtime: 文件最近一次修改时间

int newfs_readdir(const char * path, void * buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info * fi):读取对应目录项

先用 path 找到对应的目录项作为要找的父目录项，再通过 offset 得到该父目录项的第 offset 个子目录。不为空则用 filler 填充到 buf 区即可。Offset++移动到下一个子目录的位置。

```
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，可参考/fs/simplefs/sfs.c的sfs_readdir()函数实现 */
    boolean is_find, is_root;
    struct newfs_dentry * dentry = newfs_lookup(path, &is_find, &is_root);
    if(is_find){
        struct newfs_dentry* sub_dentry = newfs_get_dentry(dentry->inode, offset);
        if (sub_dentry) {
            /* 直接调用filler来装填结果 */
            filler(buf, sub_dentry->fname, NULL, ++offset);
        }
    }
    return 0;
}
```

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. 一个逻辑块中存放多个目录项文件或者多个索引节点，避免了空间的浪费。

- 目录项文件：

```
while(dir_cnt > 0 && blk_num < NEWFS_DATA_PER_FILE){
    offset = NEWFS_DATA_OFS(inode->blk_pointers[blk_num]);
    printf("    origin offset:%d\n", offset);
    while(dir_cnt > 0 && offset + sizeof(struct newfs_dentry_d) < NEWFS_DATA_OFS(inode->blk_pointers[blk_num] + 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;
        printf("    child ino:%d, child_fname%s, child dentry offset:%d\n",dentry_d.ino, dentry_d.fname, offset);
        newfs_alloc_dentry(inode, sub_dentry, FALSE); //读的时候不需要判断是否需要额外分配逻辑块给dentry
        dir_cnt --;
        offset += sizeof(struct newfs_dentry_d);
    }
    blk_num +=1;
}
```

- 索引节点：

体现在索引节点偏移的计算上：

#define NEWFS_INODES_PER_BLK 16

#define NEWFS_INO_OFS(ino) \

(newfs_super.ino_offset + ((ino) / NEWFS_INODES_PER_BLK) * NEWFS_BLK_SZ() + ((ino) % NEWFS_INODES_PER_BLK) * sizeof(struct newfs_inode_d))

2. 动态分配，只有当需要用的时候才会分配

初始化时都赋值为-1,代表尚未分配具体的逻辑块：

```
if (NEWFS_IS_REG(inode)) {
    for(int i = 0;i < NEWFS_DATA_PER_FILE; i++){
        inode->data[i] = (uint8_t *)malloc(NEWFS_BLK_SZ());
        inode->blk_pointers[i] = -1;            //采用动态分配，初始化-1
    }

}
printf("分配成功！！\n");
```

newfs_alloc_dentry 函数 中对于 dentry 存储位置的动态分配：

```
if(judge){
    /* 检查位图是否有空位 */
    if((inode->dir_cnt % NEWFS_DENTRYS_PER_BLK) == 1){ //需要找到新的逻辑块来存
        int byte_cursor = 0;
        int bit_cursor  = 0;
        int data_cursor  = 0;
        boolean is_find_free_entry = FALSE;
        for (byte_cursor = 0; byte_cursor < NEWFS_BLKS_SZ(newfs_super.data_map_blks); //字节位
            byte_cursor++)
        {
            for (bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {          //比特位
                if((newfs_super.data_map[byte_cursor] & (0x1 << bit_cursor)) == 0) {
                                                    /* 当前data_cursor位置空闲 */
                    newfs_super.data_map[byte_cursor] |= (0x1 << bit_cursor);
                    is_find_free_entry = TRUE;
                    break;
                }
                data_cursor++;
            }
            if (is_find_free_entry) {
                break;
            }
        }
        if (!is_find_free_entry || data_cursor == newfs_super.ino_max)
            return -NEWFS_ERROR_NOSPACE;
        /*这里只是为了记录数据块是否被占用*/
        int cur_blk = inode->dir_cnt / NEWFS_DENTRYS_PER_BLK;
        inode->blk_pointers[cur_blk] = data_cursor;
    }

}
```

这一点也体现在对文件的读取和写入中。如果尚未分配，直接跳过不读/写。

```
else if (NEWFS_IS_REG(inode)) {
    for(int i = 0;i <NEWFS_DATA_PER_FILE; i++){
        if(inode->blk_pointers[i] == -1) continue; //如果尚未分配，直接跳过
        inode->data[i] = (uint8_t *)malloc(NEWFS_BLK_SZ());
        if (newfs_driver_read(NEWFS_DATA_OFS(inode->blk_pointers[i]), (uint8_t *)inode->data[i], NEWFS_BLK_SZ()) != NEWFS_ERROR_NONE) {
            NEWFS_DBG("[%s] io error\n", __func__);
            return NULL;
        }
        printf("    read file:%s, offset:%d\n", inode->dentry->fname, NEWFS_DATA_OFS(inode->blk_pointers[i]));
    }
}
```

```
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->blk_pointers[i] == -1) continue; //如果尚未分配，直接跳过
        if (newfs_driver_write(NEWFS_DATA_OFS(inode->blk_pointers[i]), inode->data[i], NEWFS_BLK_SZ()) != NEWFS_ERROR_NONE) {
            NEWFS_DBG("[%s] io error\n", __func__);
            return -NEWFS_ERROR_IO;
        }
        printf("    write file:%s, offset:%d\n", inode->dentry->fname, NEWFS_DATA_OFS(inode->blk_pointers[i]));

    }
}
```

任务一和二结果

任务一代码：

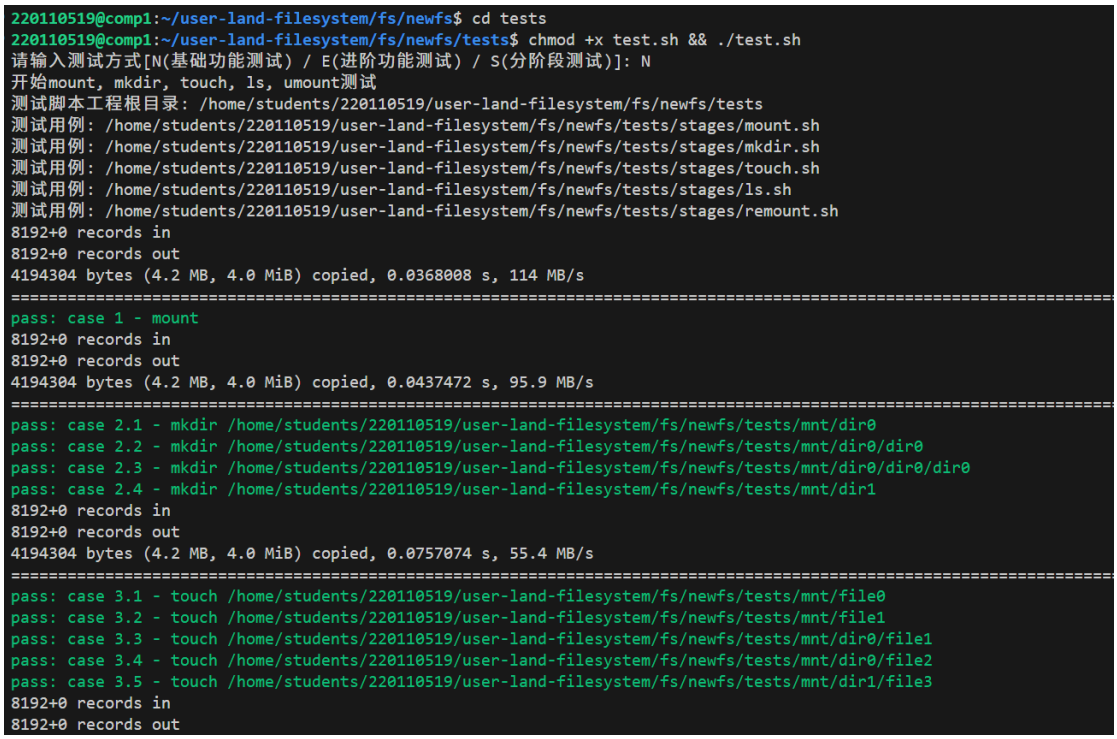
```c
static void* demo_mount(struct fuse_conn_info * conn_info){
    // 打开驱动
    char device_path[128] = {0};
    sprintf(device_path, "%s/" DEVICE_NAME, getpwuid(getuid())->pw_dir);
    super.driver_fd = ddriver_open(device_path);

    printf("super.driver_fd: %d\n", super.driver_fd);
    int size_disk, size_io;
    ddriver_ioctl(super.driver_fd, IOC_REQ_DEVICE_SIZE, &size_disk);
    ddriver_ioctl(super.driver_fd, IOC_REQ_DEVICE_IO_SZ, &size_io);
    /* 填充super信息 */
    super.sz_io = size_io;
    super.sz_disk = size_disk;
    super.sz_blks = 2 * size_io;
    printf("size_io: %d, size_disk: %d\n", size_io, size_disk);

    return 0;
}
```

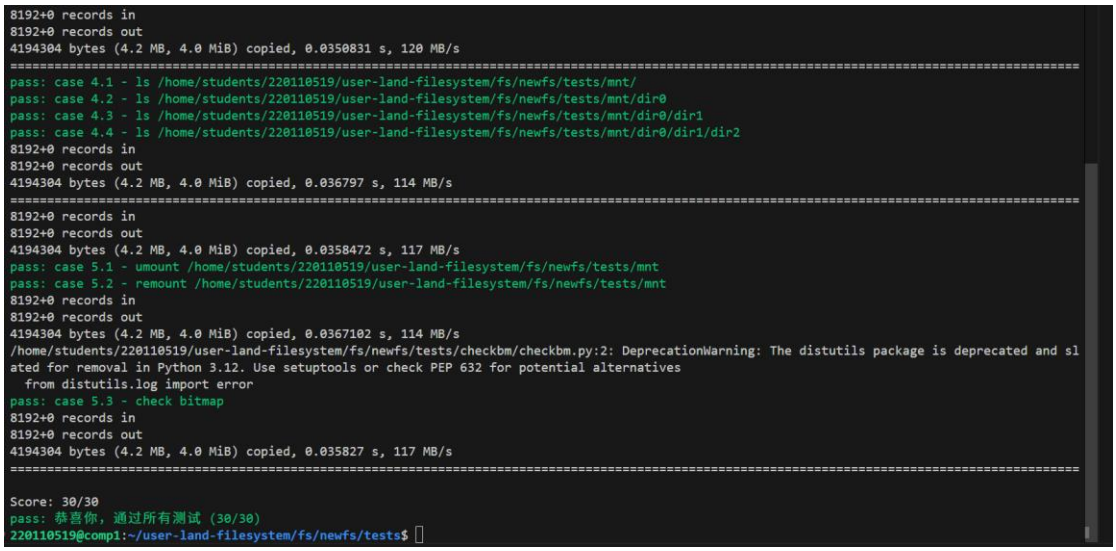
```c
static int demo_readdir(const char* path, void* buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info* f
{
    // 此处任务一同学无需关注demo_readdir的传入参数，也不要使用到上述参数

    char filename[128]; // 待填充的
    char buf_read[512];
    struct demo_dentry dentry;

    /* 根据超级块的信息，从第500逻辑块读取一个dentry，ls将只固定显示这个文件名 */

    /* TODO: 计算磁盘偏移off，并根据磁盘偏移off调用ddriver_seek移动磁盘头到磁盘偏移off处 */
    if(ddriver_seek(super.driver_fd, offset= 500 * super.sz_blks, SEEK_SET) < 0){
        printf("ddriver_seek error\n");
    }

    /* TODO: 调用ddriver_read读出一个磁盘块到内存，512B */
    if(ddriver_read(super.driver_fd, buf_read, super.sz_io) < 0){
        printf("ddriver_read error\n");
    }


    /* TODO: 使用memcpy拷贝上述512B的前sizeof(demo_dentry)字节构建一个demo_dentry结构 */
    memcpy(&dentry, buf_read, sizeof(struct demo_dentry));

    /* TODO: 填充filename */
    strcpy(filename, dentry.fname);
    // 此处大家先不关注filler，已经帮同学写好，同学填充好filename即可
    return filler(buf, filename, NULL, 0);
}
```

demo 测试结果:

```
220110519@comp4:~/user-land-filesystem/fs/demo$ cd tests
220110519@comp4:~/user-land-filesystem/fs/demo/tests$ chmod +x test.sh && ./test.sh
Test Pass!!!
220110519@comp4:~/user-land-filesystem/fs/demo/tests$
```

newfs 测试结果：





二、遇到的问题及解决方法

1. 阅读 simplefs 代码并理解：解决办法只能一边阅读 simplefs 代码一边阅读指导书。同时在写 newfs 的时候结合操作系统中对文件系统的基础知识加深自己的理解。
2. 实验过程主要问题在于 remount 后原来在挂载时候的创建的文件目录不能正常显示：Debug 的方法采用 printf 大法，最终发现是由于最开始没有为 blk_pointer 做初始化，以及在对文件读写时没有加以判断。因为在本次实验中没有为文件申请数据块，所以如果不显式地初始化 blk_pointer 的话，blk_pointer 默认指向为 0.这样在写文件时，会覆盖掉第一个数据块中的数据。因此后来我初始化每个 blk_pointer 为-1，

并在读写文件时做了特殊判断，一旦是-1 则直接跳过。最终结果成功通过了 test

三、实验收获和建议

收获与感受：

本次实验是我操作系统五个实验以来做的最久，但同时也是印象最深，成就感最高的一次实验。当 debug 并成功修改，跑过 test 时候的喜悦感是任何事情都无法比拟的。从 0 上手一个文件系统，其上手速度我个人感觉是越来越快。最开始确实一头雾水，但随着对文件系统理解的加深和阅读 simplefs，慢慢地有一种豁然开朗的感觉。本次实验我深刻理解了一个文件系统时如何运行的，特别是 EXT2 文件系统，以及索引节点和 dentry 分开之后为文件查询带来的便捷性。

建议：

实验五整体完成度很高，体验很好，若非说要建议的话可能就是课时能在多排一点，能够把整个文件系统完成我觉得就是完美了。再次感谢往届学长学姐以及老师的辛勤付出！祝实验课程越来越好。

四、参考资料

simplefs 文件夹下所有代码，实验指导书以及老师制作的 fuse 使用视频 。