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

实验报告

开课学期：2022 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：6

学生学号：200110618

学生姓名：邓皓元

评阅教师：

报告成绩：

实验与创新实践教育中心制

2022 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

对实现整个文件系统的分析说明

1.文件系统的结构(在 type.h 中有描述) 逻辑块大小：1024B，索引 inode 大小：32B

1 1 1 1 个索引对应 8 个数据块

(1)逻辑块：该文件系统每个逻辑块的大小为 1024B

(2)inode 索引：每个 inode 大小设置为 32B，1 个 inode 索引最多能对应 8 个数据块即 1024B*8=8192B

(3)在 4GB 的硬盘中，计算最多能容下的 inode 数量，因为每个 inode 最多对应 8 个数据块，每个数据块为 1 个逻辑块 1024B，因此可以计算得到最多有 4GB/(32B + 1024B*8)=509(向下取整)个 inode

(4)超级块(1 个逻辑块，起始位置 0)，超级块的结构如下：

真实磁盘上的超级块：

```
struct newfs_super_d {
    /**
    用于识别文件系统。
    比如说，如果实现的文件系统幻数为 0x20011005
    那么如果读到的幻数不等于 0x20011005
    则表示当前磁盘中无系统，系统损坏，或者是其他不识别的文件系统。
    */
    uint32_t            magic_num;              // 幻数
    int                 sz_disk;                //磁盘大小
    int                 sz_io;                  //物理磁盘块大小
    int                 driver_fd;

    int                 max_ino;                // 最多支持的inode数
    int                 map_inode_blks;         // inode位图占用的块数
    int                 map_inode_offset;       // inode位图在磁盘上的偏移

    int                 map_data_blks;          // data位图占用的块数
    int                 map_data_offset;        // data位图在磁盘上的偏移
    int                 inode_offset;           // inode在磁盘上的偏移
    int                 data_offset;            // data在磁盘上的偏移

    int                 sz_usage;
};
```

文件系统上的超级块：增加了 1 个 root_dentry 文件结构体指针，指向根目录，便于根目录的快速访问

```
struct newfs_super {

    int                 sz_io;
    int                 sz_disk;
    int                 sz_usage;

    int                 driver_fd;
    int                 max_ino;                    // 最多支持的文件数

    uint8_t*            map_inode;                  // inode位图
    int                 map_inode_blks;
    int                 map_inode_offset;
    uint8_t*            map_data;                   // data位图
    int                 map_data_blks;
    int                 map_data_offset;
    int                 inode_offset;               // inode在磁盘上的偏移
    int                 data_offset;                // data在磁盘上的偏移
    bool                is_mounted;

    struct newfs_dentry*root_dentry;                // 根目录dentry
};
```

(5)inode 位图(1 个逻辑块，起始位置 1024)，因为最多有 3968 个 inode，又因为 inode 位图用 1 个 bit 表示 1 个 inode 是否被使用，则 1 个逻辑块 1024B 可以表示 1024*8=8192 个 inode 是否被使用，因此给 inode 位图分配 1 个逻辑块，inode 位图起始位置为 0+1024(超级块)=1024

(6)data 位图(1 个逻辑块，起始位置 2048)，因为最多有 3968 个 inode，1 个 inode 最少对应 1 个数据块，因此最多有 3968 个数据块，又因为 data 位图用 1 个 bit 表示 1 个 inode 是否被使用，则 1 个逻辑块 1024B 可以表示 1024*8=8192 个 data 块是否被使用，因此给 data 位图分配 1 个逻辑块，inode 位图起始位置为 0+1024(超级块)+1024(inode 位图)=2048

(7)inode 索引(16 个逻辑块，起始位置 3072)，超级块、inode 位图、data 位图占用了 3 个逻辑块，计算最多有多少个 inode，由于 1 个 inode 最多对应 8 个数据块，因此最多可以有(4GB - 3*1024B)/(32B + 1024B)=3968(向下取整)个 inode，又因为每个 inode 大小为 32B，因此需要 32B*3968/1024B=124(向上取整)个数据块来保存所有 inode 索引节点，inode 索引起始位置为 0+1024(超级块)+1024(inode 位图)+1024(data 位图)=3072

(8)data 数据块(起始位置 19456)，由于最多有 3968 个 inode 索引节点，因此最多有 3968 个数据块，data 数据块起始位置为 0+1024(超级块)+1024(inode 位图)+1024(data 位图)+124*1024=130048

(9)inode 结构如下：仿照了 simple 的 inode 结构

真实磁盘上的 inode：

```
struct newfs_inode_d {  // 32B
    int                 ino;                    // 在inode位图中的下标
    int                 size;                   // 文件已占用空间
    int                 link;                   // 链接数
    NEWFS_FILE_TYPE     ftype;                  // 文件类型（目录类型、普通文件类型）
    int                 dir_cnt;                // 如果是目录类型文件，下面有几个目录项
    uint8_t*            data;                 // 数据块指针（可固定分配）
};
```

文件系统上的 inode：

```c
struct newfs_inode {
    int                 ino;                    // 在inode位图中的下标
    int                 size;                   // 文件已占用空间
    int                 dir_cnt;                // 目录项数量
    struct newfs_dentry*dentry;                 // 指向该inode的dentry
    struct newfs_dentry*dentrys;                // 所有目录项
    uint8_t*            data;                  // 数据块指针
};
```

(10)dentry 结构如下：仿照了 simple 的 dentry 结构

真实磁盘上的 dentry：

```c
struct newfs_dentry_d { //
    char                fname[NEWFS_MAX_FILE_NAME];      // 指向的ino文件名
    NEWFS_FILE_TYPE     ftype;                          // 指向的ino文件类型
    int                 ino;                            // 指向的ino号
    int                 valid;                          // 该目录项是否有效
};
```

文件系统上的 dentry：

```c
struct newfs_dentry {
    char                fname[NEWFS_MAX_FILE_NAME];      // 指向的inode文件名
    NEWFS_FILE_TYPE     ftype;                          // 指向的inode文件类型
    struct newfs_dentry*parent;                         /* 父亲Inode的dentry */
    struct newfs_dentry*brother;                        /* 兄弟 */
    int                 ino;                            // 指向的inode号
    struct newfs_inode*inode;                           /* 指向inode */
    int                 valid;                          // 该目录项是否有效
};
```

2.一些静态函数和定义：

静态常量：

```c
#define NEWFS_MAGIC_NUM           0x20011005
#define NEWFS_MAX_FILE_NAME       128
#define NEWFS_SUPER_BLOCKS        1       // super超级块包含的逻辑块数量
#define NEWFS_MAP_INODE_BLOCKS    1       // inode位图包含的逻辑块数量
#define NEWFS_MAP_DATA_BLOCKS     1       // data位图包含的逻辑块数量
#define NEWFS_INODE_PER_FILE      1       // 每个inode最多对应的file文件数量
#define NEWFS_DATA_PER_FILE       8       // 每个file文件最多包含的数据块数量
#define NEWFS_BLOCK_SIZE          1024    // 逻辑块大小
#define NEWFS_SUPER_OFS           0       // 超级块起始位置
#define NEWFS_MAP_INODE_OFS       1024    // inode位图起始位置 0 + 1024
#define NEWFS_MAP_DATA_OFS        2048    // data位图起始位置 1024 + 1 * 1024
#define NEWFS_INODE_OFS           3072    // inode起始位置 2048 + 1 * 1024
#define NEWFS_INODE_SIZE          32      // 每个inode的大小 每个块存1024 / 32 = 32个INODE 一共需要NEW_ROUND_UP(3968 * 32, 1024) / 1024 = 124块存取INODE
#define NEWFS_INODE_NUM           3968    // inode数量 ((4096 - 1 - 1 - 1) * 1024 / (1024 + 32) = 3968
#define NEWFS_DATA_OFS            130048  // data起始位置 3072 + 124 * 1024
#define NEWFS_DATA_SIZE           1024    // 每个数据块大小
#define NEWFS_DATA_NUM            3968    // 数据块数量 3968
```

静态函数返回参数，仿照了 simple 的一些函数返回参数的定义

```c
#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 */
```

静态函数的定义：

```
#define NEWFS_IOBLOCK_SZ()              (newfs_super.sz_io) // IO块大小
#define NEWFS_DISK_SZ()                 (newfs_super.sz_disk) // 磁盘容量大小
#define NEWFS_DRIVER()                  (newfs_super.driver_fd) // 磁盘号

#define NEWFS_ROUND_DOWN(value, round)     (value % round == 0 ? value : (value / round) * round) // 向下取整计算对应的逻辑块号
#define NEWFS_ROUND_UP(value, round)       (value % round == 0 ? value : (value / round + 1) * round) // 向上取整计算对应的逻辑块号
#define NEWFS_ASSIGN_FNAME(pnewfs_dentry, _fname)\
                                        memcpy(pnewfs_dentry->fname, _fname, strlen(_fname))
#define NEWFS_BLKS_SZ()                 (NEWFS_ROUND_UP(NEWFS_BLOCK_SIZE, NEWFS_IOBLOCK_SZ())) // 逻辑块大小
#define NEWFS_INO_OFS(ino)              (NEWFS_INODE_OFS + ino * NEWFS_INODE_SIZE) // 对应的inode位置
#define NEWFS_DA_OFS(data)              (NEWFS_DATA_OFS + data * NEWFS_DATA_PER_FILE * NEWFS_DATA_SIZE) // 对应的data位置
#define NEWFS_IS_DIR(pinode)            (pinode->dentry->ftype == NEWFS_DIR) // 是否是dir文件
#define NEWFS_IS_REG(pinode)            (pinode->dentry->ftype == NEWFS_FILE) // 是否是file文件
#define NEWFS_IS_SYM_LINK(pinode)       (pinode->dentry->ftype == NEWFS_SYM_LINK) // 是否是symlink文件
```

出错调用该函数进行错误提示输出：

```
#define NEWFS_DBG(fmt, ...) do { printf("NEWFS_DBG: " fmt, ##__VA_ARGS__); } while(0)
```

文件枚举类型：

```
typedef enum file_type {
    NEWFS_FILE,             // 普通文件
    NEWFS_DIR,              // 目录文件
    NEWFS_SYM_LINK          // 链接文件
} NEWFS_FILE_TYPE;
```

2、 功能详细说明

每个功能点的详细说明（关键的数据结构、代码、流程等）

函数功能的实现：

1.驱动读函数：用以从磁盘中读取数据，其中要读取的范围为逻辑块大小，每次读取的范围为 IO 块大小

```
int newfs_driver_read(int offset, uint8_t *out_content, int size) {
    int      offset_aligned = NEWFS_ROUND_DOWN(offset, NEWFS_BLKS_SZ());
    int      bias           = offset - offset_aligned;
    int      size_aligned   = NEWFS_ROUND_UP((size + bias), NEWFS_BLKS_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_IOBLOCK_SZ());
        cur          += NEWFS_IOBLOCK_SZ();
        size_aligned -= NEWFS_IOBLOCK_SZ();
    }
    memcpy(out_content, temp_content + bias, size);
    free(temp_content);
    return NEWFS_ERROR_NONE;
}
```

2.驱动写函数：用以向磁盘中写入数据，其中要写入的范围为逻辑块大小，每次写入的范围为 IO 块大小

```
int newfs_driver_write(int offset, uint8_t *in_content, int size) {
    int      offset_aligned = NEWFS_ROUND_DOWN(offset, NEWFS_BLKS_SZ());
    int      bias           = offset - offset_aligned;
    int      size_aligned   = NEWFS_ROUND_UP((size + bias), NEWFS_BLKS_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_IOBLOCK_SZ());
        cur          += NEWFS_IOBLOCK_SZ();
        size_aligned -= NEWFS_IOBLOCK_SZ();
    }

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

3.newfs_init 函数：挂载函数

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

    /*定义磁盘各部分结构*/
    int                 driver_fd;
    struct newfs_super_d  newfs_super_d;
    struct newfs_dentry*  root_dentry;
    struct newfs_inode*   root_inode;
    bool                is_init = false;

    /*打开驱动*/
    driver_fd = ddriver_open(newfs_options.device);

    if (driver_fd < 0) {
        return driver_fd;
    }

    /*向内存超级块中标记驱动并写入磁盘大小和单次io大小*/
    int sz_io;

    newfs_super.driver_fd = driver_fd;
    ddriver_ioctl(newfs_super.driver_fd, IOC_REQ_DEVICE_SIZE,  &newfs_super.sz_disk);
    ddriver_ioctl(newfs_super.driver_fd, IOC_REQ_DEVICE_IO_SZ, &newfs_super.sz_io);

    /*创建根目录项并读取磁盘超级块到内存*/
    root_dentry = new_dentry("/", NEWFS_DIR);

    if (newfs_driver_read(NEWFS_SUPER_OFS, (uint8_t *)(&newfs_super_d), // 读取磁盘中超级块并赋给文件系统超级块
                          sizeof(struct newfs_super_d)) != NEWFS_ERROR_NONE) {
        return -NEWFS_ERROR_IO;
    }

    if (newfs_super_d.magic_num != NEWFS_MAGIC_NUM) {     /* 幻数无 */
        newfs_super.max_ino = NEWFS_INODE_NUM;
        newfs_super.max_data = NEWFS_DATA_NUM;
        newfs_super_d.map_inode_offset = NEWFS_MAP_INODE_OFS;
        newfs_super_d.map_data_offset = NEWFS_MAP_DATA_OFS;
        newfs_super_d.map_inode_blks  = NEWFS_MAP_INODE_BLOCKS;
        newfs_super_d.map_data_blks  = NEWFS_MAP_DATA_BLOCKS;
        newfs_super_d.inode_offset = NEWFS_INODE_OFS;
        newfs_super_d.data_offset = NEWFS_DATA_OFS;
        newfs_super_d.sz_usage  = 0;
        NEWFS_DBG("inode map blocks: %d\n", newfs_super_d.map_inode_blks);
        is_init = true;
    }

    newfs_super.sz_usage   = newfs_super_d.sz_usage;      /* 建立 in-memory 结构 */
```

```
    newfs_super.map_inode = (uint8_t *)malloc(newfs_super_d.map_inode_blks * NEWFS_BLKS_SZ()); // 给文件系统inode位图分配空间
    newfs_super.map_inode_blks = newfs_super_d.map_inode_blks;
    newfs_super.map_inode_offset = newfs_super_d.map_inode_offset;

    newfs_super.map_data = (uint8_t *)malloc(newfs_super_d.map_data_blks * NEWFS_BLKS_SZ()); // 给文件系统data位图分配空间
    newfs_super.map_data_blks = newfs_super_d.map_data_blks;
    newfs_super.map_data_offset = newfs_super_d.map_data_offset;

    if (newfs_driver_read(newfs_super_d.map_inode_offset, (uint8_t *)(newfs_super.map_inode),  // 读取磁盘inode位图给文件系统inode位图
                          newfs_super_d.map_inode_blks * NEWFS_BLKS_SZ()) != NEWFS_ERROR_NONE) {
        return -NEWFS_ERROR_IO;
    }

    if (newfs_driver_read(newfs_super_d.map_data_offset, (uint8_t *)(newfs_super.map_data),    // 读取磁盘data位图给文件系统data位图
                          newfs_super_d.map_data_blks * NEWFS_BLKS_SZ()) != NEWFS_ERROR_NONE) {
        return -NEWFS_ERROR_IO;
    }

    if (is_init) {                                  /* 若尚未初始化，分配根节点 */
        root_inode = newfs_alloc_inode(root_dentry); // 为根目录项分配inode
        newfs_sync_inode(root_inode);               // 将根目录inode下的文件结构刷回磁盘
    }

    root_inode            = newfs_read_inode(root_dentry, 0); // 读取根节点
    root_dentry->inode    = root_inode;                       // 连接根目录和根节点
    newfs_super.root_dentry = root_dentry;
    newfs_super.is_mounted  = true;

    return NULL;
}
```

其中用到的函数：

new_dentry 函数:创建一个对应文件类型的 dentry 文件

```
struct newfs_dentry* new_dentry(char * fname, NEWFS_FILE_TYPE ftype) {
    struct newfs_dentry * dentry = (struct newfs_dentry *)malloc(sizeof(struct newfs_dentry));
    memset(dentry, 0, sizeof(struct newfs_dentry));
    NEWFS_ASSIGN_FNAME(dentry, fname);
    dentry->ftype   = ftype;
    dentry->ino     = -1;
    dentry->inode   = NULL;
    dentry->valid   = 0;
    return  dentry;
}
```

newfs_alloc_inode 函数:为对应 dentry 分配一个 inode 并修改 inode 位图

```
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;
    bool is_find_free_entry = false;

    for (byte_cursor = 0; byte_cursor < newfs_super.map_inode_blks * NEWFS_BLKS_SZ();
         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;

    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)) {
        inode->data = (uint8_t*)malloc(NEWFS_DATA_PER_FILE * NEWFS_BLKS_SZ());
    }
    return inode;
}
```

newfs_sync_inode 函数:把对应 inode 以下的结构全部刷回磁盘

```
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;
    inode_d.ftype       = inode->dentry->ftype;
    inode_d.dir_cnt     = inode->dir_cnt;
    int offset;
    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;
    }
                                                    /* Cycle 1: 写 INODE */
                                                    /* Cycle 2: 写 数据 */
    if (NEWFS_IS_DIR(inode)) {
        dentry_cursor = inode->dentrys;
        offset        = NEWFS_DA_OFS(ino);
        while (dentry_cursor != NULL)
        {
            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;
            }

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

            dentry_cursor = dentry_cursor->brother;
            offset += sizeof(struct newfs_dentry_d);
        }
    }
    else if (NEWFS_IS_REG(inode)) {
        if (newfs_driver_write(NEWFS_DA_OFS(ino), inode->data,
                               NEWFS_ROUND_UP(inode->size, NEWFS_BLKS_SZ())) != NEWFS_ERROR_NONE) {
            NEWFS_DBG("[%s] io error\n", __func__);
            return -NEWFS_ERROR_IO;
        }
    }
    return NEWFS_ERROR_NONE;
}
```

newfs_read_inode 函数:将对应 ino 号的 inode 指向对应 dentry，返回读取到的 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, i;
    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;
    inode->dentry = dentry;
    inode->dentrys = NULL;
    if (NEWFS_IS_DIR(inode)) {
        dir_cnt = inode_d.dir_cnt;
        for (i = 0; i < dir_cnt; i++)
        {
            if (newfs_driver_read(NEWFS_DA_OFS(ino) + i * sizeof(struct newfs_dentry_d),
                                  (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);
        }
    }
    else if (NEWFS_IS_REG(inode)) {
        inode->data = (uint8_t *)malloc(NEWFS_DATA_PER_FILE * NEWFS_BLKS_SZ());
        if (newfs_driver_read(NEWFS_DA_OFS(ino), (uint8_t*)inode->data,
                              NEWFS_DATA_PER_FILE * NEWFS_BLKS_SZ()) != NEWFS_ERROR_NONE) {
            NEWFS_DBG("[%s] io error\n", __func__);
            return NULL;
        }
    }
    return inode;
}
```

newfs_alloc_dentry 函数;为对应 inode 分配一个 dentry

```
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;
    }
    //如果分配dentry给inode之后当前拥有的数据块大小不够则分配一个数据块并修改数据位图
    int data_cursor = 0;
    bool is_find_free_entry = false;
    if(NEWFS_ROUND_UP(inode->size,NEWFS_BLKS_SZ()) != NEWFS_ROUND_UP(inode->size + sizeof(struct newfs_dentry),NEWFS_BLKS_SZ()))
    {
        for (int byte_cursor = 0; byte_cursor < newfs_super.map_data_blks * NEWFS_BLKS_SZ(); byte_cursor++)
        {
            for (int bit_cursor = 0; bit_cursor < UINT8_BITS; bit_cursor++) {
                if((newfs_super.map_data[byte_cursor] & (0x1 << bit_cursor)) == 0) {
                                                    /* 当前ino_cursor位置空闲 */
                    newfs_super.map_data[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.max_data)
        return -NEWFS_ERROR_NOSPACE;

    inode->dir_cnt++;
    inode->size+=sizeof(struct newfs_dentry);
    return inode->dir_cnt;
}
```

4.newfs_getattr 函数：解析路径，获取文件或者目录的属性

```
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 */
    bool    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_BLKS_SZ();

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

其中用到的函数：

newfs_new_lookup 函数:判断该路径文件是不是根目录，是不是已经存在，返回路径的上级目录文件

```
struct newfs_dentry* newfs_lookup(const char * path, bool* is_find, bool* 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;
    bool 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;
    }
    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) {
            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)
            {
                if (memcmp(dentry_cursor->fname, fname, strlen(fname)) == 0) {
                    is_hit = true;
                    break;
                }
                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, "/");
    }

    if (dentry_ret->inode == NULL) {
        dentry_ret->inode = newfs_read_inode(dentry_ret, dentry_ret->ino);
    }

    return dentry_ret;
}
```

newfs_calc_lvl 函数:计算路径层级

```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;
}
```

5.newfs_mkdir 函数：在对应路径创建一个目录文件

```c
int newfs_mkdir(const char* path, mode_t mode) {
    (void)mode;
    bool is_find, is_root;
    char* fname;
    struct newfs_dentry* last_dentry = newfs_lookup(path, &is_find, &is_root);//寻找上级目录项
    struct newfs_dentry* dentry;
    struct newfs_inode*  inode;

    if (is_find) {//目录存在
        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);
    newfs_alloc_dentry(last_dentry->inode, dentry);

    return NEWFS_ERROR_NONE;
}
```

6.newfs_mknod 函数：在对应路径创建一个普通文件

```
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 */
    bool is_find, is_root;

    struct newfs_dentry* last_dentry = newfs_lookup(path, &is_find, &is_root);//找到创建文件所在的目录
    struct newfs_dentry* dentry;
    struct newfs_inode* inode;
    char* fname;

    if (is_find == true) {//文件存在
        return -NEWFS_ERROR_EXISTS;
    }

    fname = newfs_get_fname(path);//获取文件名字

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

    return NEWFS_ERROR_NONE;
}
```

7.newfs_readdir 函数:读取 dentry 目录文件

```
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 */
    bool 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;
}
```

8.newfs_destroy 函数:卸载

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

    if (!newfs_super.is_mounted) {
        return NEWFS_ERROR_NONE;
    }

    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.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.inode_offset         = newfs_super.inode_offset;
    newfs_super_d.data_offset         = newfs_super.data_offset;
    newfs_super_d.sz_usage            = newfs_super.sz_usage;

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

    if (newfs_driver_write(newfs_super_d.map_inode_offset, (uint8_t *)(newfs_super.map_inode),
                           newfs_super_d.map_inode_blks * NEWFS_BLKS_SZ()) != NEWFS_ERROR_NONE) {
        return -NEWFS_ERROR_IO;
    }

    if (newfs_driver_write(newfs_super_d.map_data_offset, (uint8_t *)(newfs_super.map_data),
                           newfs_super_d.map_data_blks * NEWFS_BLKS_SZ()) != NEWFS_ERROR_NONE) {
        return -NEWFS_ERROR_IO;
    }

    free(newfs_super.map_inode);
    free(newfs_super.map_data);
    ddriver_close(NEWFS_DRIVER());
    return;
}
```

9.newfs_read 函数:读取文件

```
int newfs_read(const char* path, char* buf, size_t size, off_t offset,
               struct fuse_file_info* fi) {
    bool 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;
    }

    inode = dentry->inode;

    if (NEWFS_IS_DIR(inode)) {
        return -NEWFS_ERROR_ISDIR;
    }

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

    memcpy(buf, inode->data + offset, size);

    return size;
}
```

10.newfs_write 函数:写入文件

```
int newfs_write(const char* path, const char* buf, size_t size, off_t offset,
                struct fuse_file_info* fi) {
    bool 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;
    }

    inode = dentry->inode;

    if (NEWFS_IS_DIR(inode)) {
        return -NEWFS_ERROR_ISDIR;
    }

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

    memcpy(inode->data + offset, buf, size);
    inode->size = offset + size > inode->size ? offset + size : inode->size;

    return size;
}
```

3、 *实验特色*

实验中你认为自己实现的比较有特色的部分

1.文件系统结构：整个文件系统按照 EXT2 文件系统的文件系统布局进行结构设计，仿照 simple 文件系统进行修改，将 simple 一个文件对应固定数量的数据块的 inode 对应文件的存储方式改成了按照文件大小按需分配数据块的存储方式。

2.对整个磁盘空间的高效利用，每个 inode 对应一个文件，文件大小为 1-8 数据块，对每个文件按照数据块数灵活分配能够最大程度地利用到整个磁盘空间，使得文件系统可以存取更多的文件(增加了 max_ino 的值)

二、用户手册

实现的文件系统中的所有命令使用方式

1.mount ：挂载，没有参数，读取磁盘并挂载到对应文件系统
2.mkdir 文件路径：创建目录文件，参数为希望创建目录文件的路径
3.touch 文件路径：查看对应路径文件的文件属性
4.ls 文件路径：查看对应路径文件下的子文件
5.mknod 文件路径：创建普通文件，参数为希望创建普通文件的路径
6.umount 文件路径：卸载，参数为想要卸载的文件系统挂载位置路径

三、实验收获和建议

实验中的收获、感受、问题、建议等。

做完五个实验之后，我对于操作系统的文件管理有了完全不同于理论课的认识，对于 xv6 系统的底层实现逻辑和运行方法有了更加深刻的理解。在学习文件系统实验课之前，我对于文件系统和 IO 存储的知识还停留在表面理论上，并没有将其运用于代码实操的能力，对于整个文件系统的设计和构造也只停留在表面，在自己设计和完成文件系统的结构和功能之后，我对于文件系统有了更加深刻的认识，也明白了开始真正实施一个项目之前学习相应知识和搜索对应资料的重要性。

在设计和实现文件系统的过程中，我对于许多知识点的理解不够深刻导致多次对先前的设计进行更改，大大增加了整个文件系统的实现耗时，在一次又一次查看指导网站的过程中，我巩固了有关文件系统的知识，锻炼了自己的代码实操能力，在操作系统这门课程的学习中受益良多。

总而言之，操作系统实验令我受益匪浅，希望操作系统实验课越做越好。

四、参考资料

实验过程中查找的信息和资料

参考了以下的博客

(9 条消息) 模拟实现 EXT2 文件系统_berry_juice 的博客-CSDN 博客_模拟磁盘文件系统实现

、实验指导网站、simple 文件系统例子