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：

学生学号：

学生姓名：

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

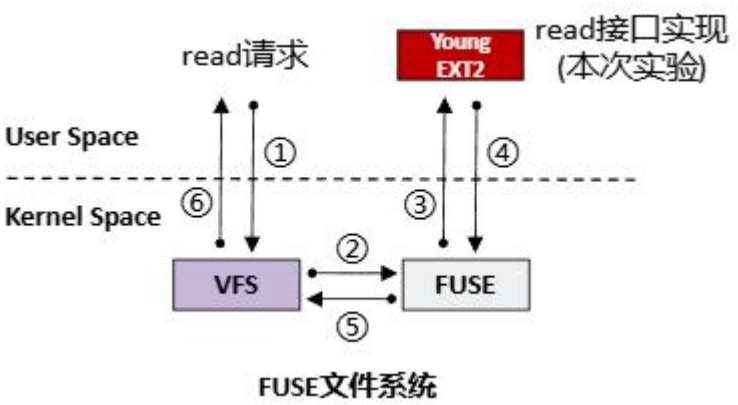
1、 总体设计方案

本实验基于 FUSE 架构实现了一个 EXT2 的文件系统。

1.FUSE 架构

VFS 是 Linux 虚拟文件系统，是所有内核文件系统的统一接口。

来自用户态的文件请求到达内核文件系统，就会调用对应的接口，在 FUSE 架构中，内核态并不负责实现对应的接口，它会额外注册到 FUSE 的用户态部分，也就是图中红色的部分。本次实验的任务就是在红色部分中实现内核态的各种文件接口。



FUSE文件系统

2.EXT2 文件系统的磁盘布局

磁盘由以下五部分组成：超级块、索引节点位图、数据块位图、索引节点区和数据块区。需要设计各部分在磁盘中所占空间的大小。

虚拟磁盘的大小为 4MB，每个逻辑块的大小为 1024B。超级块只需要一个逻辑块即可容纳。即使每个文件只占一个逻辑块，磁盘中最多可存储的文件数量也只有 4096，不超过一个逻辑块的大小，因此索引节点位图和数据块位图各自只占用一个逻辑块。

一个逻辑块中存放 16 个索引节点（为什么是 16 个在下一个部分中说明），一个文件最多占用 4 个逻辑块，最少占用一个逻辑块，也就是说最多磁盘中可以存放 4096-1-1-1 = 4093 个文件（忽略索引节点区），而这么多的文件需要 4093/16≈256 个逻辑块存放索引节点，也就是说数据块的数量为 4093-256 = 3837（这只是粗略的估计，但与准确计算的结果相差不大）。

最终，磁盘布局的设计如下

```
| BSIZE = 1024 B |
| Super(1) | Inode Map(1) | DATA Map(1) | INODE(256) | DATA(*) |
```

2、 功能详细说明

1.数据结构

内存超级块：

```
struct newfs_super {
    uint32_t magic;
    int      fd;
    /* TODO: Define yourself */
    int                 sz_io;        //驱
    int                 sz_disk;        //
    int                 sz_usage;
    int                 sz_blks;        /
    //索引节点
    int                 max_ino;        /
    uint8_t*            map_inode;
    int                 map_inode_blks; //
    int                 map_inode_offset;
    int                 inode_offset;    /
    //数据块
    uint8_t*            map_data;
    int                 max_data;        /
    int                 data_offset;     /
    int                 map_data_offset; /
    int                 map_data_blks;   /

    boolean             is_mounted;
    struct newfs_dentry* root_dentry;
};
```

磁盘超级块：

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

    int                 map_inode_blks;
    int                 map_inode_offset;

    int                 map_data_offset;
    int                 map_data_blks;

    int                 data_offset;
    int                 inode_offset;
};
```

超级块中存放了磁盘布局的信息，例如逻辑块的大小，各分区的起始地址等。

内存 inode：

```
struct newfs_inode {
    int           ino;
    int                size;                              /*
    int                link;
    int                block_pointer[NEWFS_DATA_PER_FILE];
    int                dir_cnt;                           //
    struct newfs_dentry* dentry;                          /*
    struct newfs_dentry* dentrys;                         /*
    uint8_t*           data[NEWFS_DATA_PER_FILE];         //
    NEWFS_FILE_TYPE          ftype;
};
```

磁盘 inode：

```
struct newfs_inode_d
{
    uint32_t           ino;                               /* 
    uint32_t           size;                              /* 
    int                link;
    int                block_pointer[NEWFS_DATA_PER_FILE];
    uint32_t           dir_cnt;
    NEWFS_FILE_TYPE      ftype;
};
```

inode 中存放了文件的关键信息，例如文件的类型、大小和数据块指针等。磁盘 inode 的大小为 36 字节，一个逻辑块的大小为 1024 字节，也就是说一个逻辑块可以存放 28 个磁盘 inode，在我的设计中，一个逻辑块中存放了 16 个 inode（因为 16 是 2 的次幂）。

内存目录项：

```
struct newfs_dentry {
    int ino;
    char               fname[NEWFS_MAX_FILE_NAME];
    struct newfs_dentry* parent;
    struct newfs_dentry* brother;
    struct newfs_inode*  inode;
    NEWFS_FILE_TYPE      ftype;
};
```

磁盘目录项：

```
struct newfs_dentry_d
{
    char               fname[NEWFS_MAX_FILE_NAME];
    NEWFS_FILE_TYPE      ftype;
    uint32_t           ino;
};
```

目录项是文件夹中的内容，主要存放了 inode 和对应的文件名。

2.功能函数说明

（1）new_dentry

创建一个新的目录项并返回，指定目录项中文件的名字和类型。

```c
static inline 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->parent  = NULL;
    dentry->brother = NULL;
    return dentry;
}
```

（2）newfs_driver_read 和 newfs_driver_write

驱动读写函数，因为文件系统是按块读写的，一个 io 块的大小为 512B，如果要读取的文件内容的地址不是从 512 的倍数开始的，则将读取的内容扩大并对齐到 512 的倍数，读取更大的文件部分到内存中，然后再对更大的部分进行操作。

其中 NEWFS_ROUND_DOWN 函数将偏移量 offset 对齐到更低的 512 倍数地址，NEWFS_ROUND_UP 将要读取的部分向上对齐到更高的 512 倍数地址。

```c
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;
}
```

```
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）newfs_alloc_inode

为输入的 dentry 分配一个 inode，并修改位图。首先遍历位图检查 inode 位图是否有空位，如果找到空位，则将空位置为 1，表示已被占用，并设置好 inode 对应的 ino。然后将 inode 与 dentry 关联起来。

因为我设计的文件系统的内存是动态分配的，所以在这里不需要为文件分配数据段，数据段在写文件内容时才分配。

```
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;

    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;

    //普通文件也不需要分配数据块了，分配数据块的过程会在写入文件时进行
```

（4）newfs_sync_inode

将内存 inode 及其下方的结构全部刷回磁盘。首先创建磁盘的 inode 结构，将成员赋值，写回 inode 本身。然后再写 inode 下方的数据。

```
struct newfs_inode_d  inode_d;
struct newfs_dentry*  dentry_cursor;
struct newfs_dentry_d dentry_d;
int ino             = inode->ino;

int offset;
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/16) + ino%16*sizeof(struct newfs_inode_d), (uint8_t *)&inode_d,
                       sizeof(struct newfs_inode_d)) != NEWFS_ERROR_NONE) {
    NEWFS_DBG("[%s] io error\n", __func__);
    return -NEWFS_ERROR_IO;
}
```

如果 inode 对应的文件是目录，那么要将目录项写回磁盘，并且目录项的 inode

也要写回磁盘。每个文件最多使用 4 个逻辑块，因此如果要写回的 dentry 超过一个逻辑块，那么根据 block_pointer 找到下一个逻辑块继续写。

```
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;
    int blk_number = 0;

    while (dentry_cursor != NULL && blk_number < NEWFS_DATA_PER_FILE)
    {
        offset = NEWFS_DATA_OFS(inode->block_pointer[blk_number]);
        while ((dentry_cursor != NULL) && (offset < NEWFS_DATA_OFS(inode->block_pointer[blk_number] + 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;
                }

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

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

如果 inode 对应的文件是普通文件，那么直接写回其中的数据。根据 inode 中的 size 字段计算普通文件的数据占用了几个逻辑块，并将这些逻辑块写回。

```
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<inode->size/NEWFS_BLK_SZ(); i++){
            if (newfs_driver_write(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 -NEWFS_ERROR_IO;
    }
    }

}
```

（5）read_inode

从磁盘中读取指定 ino 的 inode，并将之与传入的 dentry 关联。

首先从磁盘中读取 inode 本身，并初始化内存中的 inode。

```
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/16) + ino%16*sizeof(struct newfs_inode_d), (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;
for(int i = 0; i < NEWFS_DATA_PER_FILE; i++){
    inode->block_pointer[i] = inode_d.block_pointer[i];
}
```

如果 inode 对应的文件类型是文件夹，那么一个个地读取数据区中的 dentry。

```
if (NEWFS_IS_DIR(inode)) {
    dir_cnt = inode_d.dir_cnt;
    int blk_number = 0;
    int offset;

    while(dir_cnt > 0 && blk_number < NEWFS_DATA_PER_FILE){
        offset = NEWFS_DATA_OFS(inode->block_pointer[blk_number]);

        while((dir_cnt > 0) && (offset + sizeof(struct newfs_dentry_d) < NEWFS_DATA_OFS(inode->block_pointer[blk_number] + 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--;
        }
        blk_number++;
    }
}
```

如果 inode 对应的文件类型是普通文件，那么根据 size 字段判断读取的数据块数量，直接整块地读入数据块。

```
else if (NEWFS_IS_REG(inode)) {
    for (int i = 0; i < inode->size/NEWFS_BLK_SZ(); i++){
        inode->data[i] = (uint8_t *)malloc(NEWFS_BLK_SZ());
        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;
        }
    }
}
```

（6）newfs_alloc_dentry

首先将 dentry 加入到 inode 的 dentrys 链表中，并将 dir_cnt++，表示目录下的文件数量+1.

```
if (inode->dentrys == NULL) {
    inode->dentrys = dentry;
}
else {
    dentry->brother = inode->dentrys;
    inode->dentrys = dentry;
}
inode->dir_cnt++;
```

然后，因为我设计的文件系统的内存是动态分配的，因此需要在为文件添加了新的目录项之后判断是否需要为文件分配新的数据块，或者文件占用的内存是否超出了规定的文件最大大小。如果超过了最大大小，需要报错。

```
int cur_blk = inode->dir_cnt / NEWFS_MAX_DENTRY_BLK();
if(inode->dir_cnt % NEWFS_MAX_DENTRY_BLK() == 1){
    if(cur_blk == NEWFS_DATA_PER_FILE){ //超出文件最大大小
        return -1;
    }
    int byte_cursor = 0;
    int bit_cursor  = 0;
    int dno_cursor  = 0;
    boolean is_find_free_blk = FALSE;
    //根据数据块位图查找空闲数据块
    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) {
                newfs_super.map_data[byte_cursor] |= (0x1 << bit_cursor);
                inode->block_pointer[cur_blk] = dno_cursor;
                is_find_free_blk = TRUE;
                break;
            }
            dno_cursor++;
        }
        if (is_find_free_blk) {
            break;
        }
    }
    if (!is_find_free_blk || dno_cursor == newfs_super.max_data)
        return -NEWFS_ERROR_NOSPACE;
}
```

（7）newfs_lookup

根据输入的路径查找目录项，如果找到，返回该目录项；如果没找到，则返回上一个有效的路径。

（8）newfs_calc_lvl

计算输入路径的层级。

（9）newfs_get_fname

根据输入的路径获取文件的名字，即最后一个’/’之后的内容。

3.mount 实现

mount 过程的核心代码在函数 newfs_mount 中，其关键部分如下。

首先，将内存中的超级块的 is_mounted 设为 FALSE，表示此时还未挂载。然后打开磁盘文件。

```
newfs_super.is_mounted = FALSE;

// driver_fd = open(options.device, O_RDWR);
driver_fd = ddriver_open(options.device);

if (driver_fd < 0) {
    return driver_fd;
}
```

然后从磁盘文件中读入超级块。

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

接着，根据读入的磁盘超级块的 magic_num 判断是否是第一次挂载，如果是的话要

进行初始化，为磁盘中超级快的各个字段赋值。

```
if (newfs_super_d.magic_num != NEWFS_MAGIC_NUM) {
    /* 估算各部分大小 */
    super_blks = NEWFS_SUPER_BLKS;
    map_inode_blks = NEWFS_MAP_INODE_BLKS;
    map_data_blks = NEWFS_MAP_DATA_BLKS;
    inode_num  = NEWFS_INODE_BLKS;
    data_num = NEWFS_DATA_BLKS;

    newfs_super.max_ino = inode_num;
    newfs_super.max_data = data_num;

    newfs_super_d.map_inode_blks = map_inode_blks;
    newfs_super_d.map_data_blks = map_data_blks;

    newfs_super_d.map_inode_offset = NEWFS_SUPER_OFS + NEWFS_BLKS_SZ(super_blks);
    newfs_super_d.map_data_offset = newfs_super_d.map_inode_offset + NEWFS_BLKS_SZ(map_inode_blks);

    newfs_super_d.inode_offset = newfs_super_d.map_data_offset + NEWFS_BLKS_SZ(map_data_blks);
    newfs_super_d.data_offset = newfs_super_d.inode_offset + NEWFS_BLKS_SZ(inode_num);

    newfs_super_d.sz_usage = 0;
    newfs_super_d.magic_num = NEWFS_MAGIC_NUM;

    is_init = TRUE;
}
```

然后，无论是不是第一次挂载，均要为内存超级块的各个字段赋值，建立内存超级块。

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

newfs_super.map_inode = (uint8_t *)malloc(NEWFS_BLKS_SZ(newfs_super_d.map_inode_blks));
newfs_super.map_inode_blks = newfs_super_d.map_inode_blks;
newfs_super.map_inode_offset = newfs_super_d.map_inode_offset;
newfs_super.inode_offset = newfs_super_d.inode_offset;
newfs_super.map_data = (uint8_t *)malloc(NEWFS_BLKS_SZ(newfs_super_d.map_data_blks));
newfs_super.map_data_blks = newfs_super_d.map_data_blks;
newfs_super.map_data_offset = newfs_super_d.map_data_offset;
newfs_super.data_offset = newfs_super_d.data_offset;

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

4. umount 实现

将根目录 inode 及其下方的数据写回到磁盘中，将超级块，inode 位图和数据块位图写回到磁盘中，最后释放内存中的部分，再关闭打开的磁盘文件。

```c
int newfs_umount() {
    struct newfs_super_d  newfs_super_d;

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

    newfs_sync_inode(newfs_super.root_dentry->inode);

    newfs_super_d.magic_num           = NEWFS_MAGIC_NUM;
    newfs_super_d.sz_usage            = newfs_super.sz_usage;

    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.inode_offset        = newfs_super.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.data_offset         = newfs_super.data_offset;

    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_BLKS_SZ(newfs_super_d.map_inode_blks)) != NEWFS_ERROR_NONE) {
        return -NEWFS_ERROR_IO;
    }

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

    free(newfs_super.map_inode);
    free(newfs_super.map_data);

    ddriver_close(NEWFS_DRIVER());
```

5. mkdir 实现

首先调用 newfs_lookup 查找要创建的文件的上一级有效目录，如果找到了当前要创建的文件，说明文件已经创建，报错；如果上一级的有效文件的文件类型是普通文件，说明输入的路径有误，报错。

如果没有错误，则创建文件的 inode 和 dentry，并将二者关联。

```c
int newfs_mkdir(const char* path, mode_t mode) {
    (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);
    newfs_alloc_dentry(last_dentry->inode, dentry);

    return 0;
}
```

6. mknod 实现

创建文件。

与创建文件夹的逻辑基本相同。

```c
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_ISDIR(mode)) {
        dentry = new_dentry(fname, NEWFS_DIR);
    }else {
        dentry = new_dentry(fname, NEWFS_FILE);
    }

    dentry->parent = last_dentry;
    inode = newfs_alloc_inode(dentry);
    newfs_alloc_dentry(last_dentry->inode, dentry);

    return NEWFS_ERROR_NONE;
}
```

7. newfs_getattr 实现

获取文件或目录的属性。

首先查找当前文件的 dentry，如果没有找到，那么报错。如果找到了，那么根据文件的类型设置 stat 结构体中的 st_mode 字段和 st_size 字段。其他字段固定，特别的，如果当前文件是根目录，那么需要额外设置一些字段，并且根目录的 link 数固定为 2.

```
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/newfs.c的newfs_getat
    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 = NEWFS_DEFAULT_PERM | S_IFDIR;
        newfs_stat->st_size = dentry->inode->dir_cnt * sizeof(struct newfs_dentry_d);
    }
    else if (NEWFS_IS_REG(dentry->inode)) {
        newfs_stat->st_mode = NEWFS_DEFAULT_PERM | S_IFREG;
        newfs_stat->st_size = dentry->inode->size;
    }

    newfs_stat->st_nlink   = 1;
    newfs_stat->st_uid     = getuid();          //文件所有者的用户ID
    newfs_stat->st_gid     = getgid();          //文件所有者的组ID
    newfs_stat->st_atime   = time(NULL);        //文件最近一次访问时间
    newfs_stat->st_mtime   = time(NULL);        //文件最近一次修改时间
    newfs_stat->st_blksize = NEWFS_BLK_SZ();    //文件所在文件系统的逻辑块大小

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

8. newfs_readattr 实现

给定路径，路径中的目录项，并交给 FUSE 输出。

首先查找路径，如果找到了，那么调用 filter 打印文件名，如果没有找到，那么报错。

```
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/simple
    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;
}
```

3、 实验特色

1. 按需内存分配

在创建文件和文件夹时，不会直接分配内存，而是在向文件和文件夹中写入内容时按需分配内存。这种按需分配内存的方法能提高对内存的利用率，减少文件内存分配中的内部碎片。

2. 一个逻辑块存放 16 个 inode

在 simplefs 中，一个 inode 就要占用一个逻辑块。而一个 inode 的大小不过几十个字节，占用一个逻辑块造成了非常大的空间浪费，因此经过计算，我将一个逻辑块中容纳 16 个 inode，极大地节省了存储空间。

二、遇到的问题及解决方法

在最开始我并没有采用按需分配内存策略，而是在在创建文件时直接为每个文件分配了相同的内存空间，并同时修改了数据位图。这导致了我的代码在测试中无法通过测试 5.2 和 5.3.我根据 5.2 的报错“数据位图不匹配”推测出了代码的错误是在创建文件时直接分配了内存空间，并且修改了位图。在我将分配空间的代码删除后，通过了测试。经过后来的分析，我发现 5.3 也报错的原因是我并没有设置 block_pointer 的值，但是在写回和读入的时候却使用了 block_pointer，写和读了错误的地址。

三、实验收获和建议

万事开头难，开始上手写实验时最大的问题是不知道如何设计 inode，dentry 和 super

结构体，而在 mount 函数中就需要用到 super 和 super_d，可能是有些强迫症，如果不能把这些数据结构很好地设置好，我在写后面的代码时总是心慌，效率很低，因此我硬着头皮先把结构体声明好了，但是在后面写代码时才发现结构体中缺少了一些必要的成员，又多了一些没有必要的成员，删删改改之间让我思绪很乱，导致效率低下。现在写完了回头来看，一开始就应该先把结构体的定义搁置，在后续写代码时发现需要什么成员再回头添加。

在上手写了一段时间的代码之后，发现实验并没有想象中那么复杂，代码很快就写完了，但最痛苦的过程就是 debug，因为没什么 debug 的经验，面对庞杂的代码根本不知道怎么高效地 debug，只能看着报错去找可能出错的地方，整个过程十分折磨，也是本次实验中花费我时间最多的部分。如果可以的话希望实验指导书中可以添加一些如何高效 debug 的内容（如果有帮助 debug 的工具的话）。

四、参考资料

主要参考了 simplefs 的代码实现以及实验指导书。