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：

非常不建议写个名字就作为你的实验报告

实验与创新实践教育中心制

2024 年 9 月

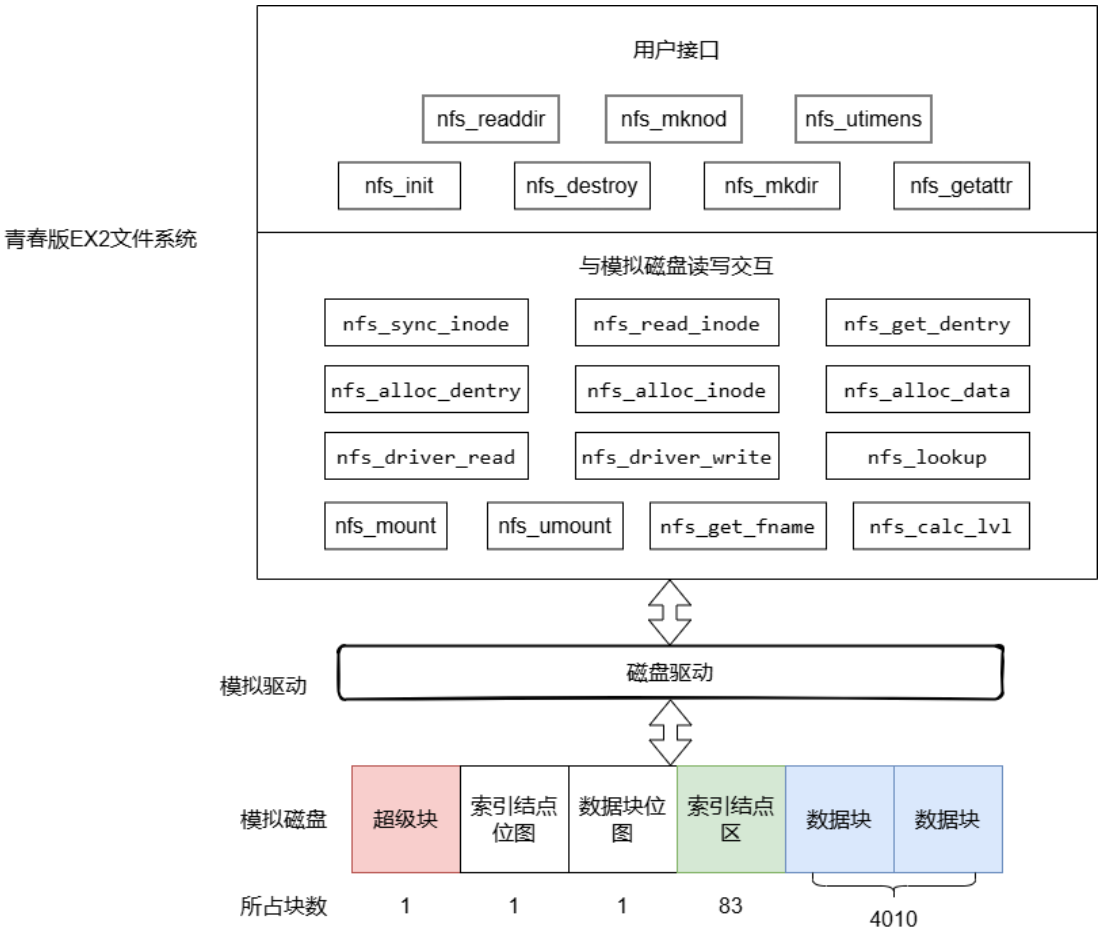
一、实验详细设计

图文并茂地描述实验实现的所有功能和详细的设计方案及实验过程中的特色部分。

1、 总体设计方案

详细阐述文件系统的总体设计思路，包括系统架构图和关键组件的说明。

本次青春版 EX2 文件系统设计主要分为三部分：模拟磁盘布局设计、与模拟磁盘读写交互设计和文件系统接口设计。系统架构图如下：



在模拟磁盘布局设计中，磁盘容量为 4MB，单个逻辑块的大小为 1024B，每个文件最多直接索引 6 个逻辑块来填写文件数据。由于设计的单个 inode 结点大小为 104B，故设计一个逻辑块最多可存放 8 个 inode 结点。8 个文件占用的空间大小为 8 * 6 + 1 = 49KB，则 4MB 的磁盘最多可以存放的文件个数是(4MB / 49KB) * 8 = 664 个文件，需要 4MB / 49KB = 83 个逻辑块存储索引节点；而超级块的大小小于 1024B，用一个逻辑块存储；一个逻辑块可以管理 1024 * 8 = 8192 个索引结点/数据块，大于文件系统最大的索引结点数/逻辑块数，故索引结点位图和数据块位图分别使用一个逻辑块存储，剩余的 4010 个逻辑块作为数据块。在模拟磁盘存储中，亦按照图中顺序存储相应的信息，如第一个逻辑块存储超级块信息。

超级块的幻数约定为 0x22011022。

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）

与模拟磁盘读写交互部分

- nfs_mount：

nfs_mount 函数实现文件系统的挂载。文件系统的挂载需要填写超级块中相关字段信息，超级块 nfs_super 的 in-Mem 结构如下图所示：

```
struct nfs_super {
    uint32_t magic;
    int      fd;
    /* TODO: Define yourself */
    int sz_io;   // io大小
    int sz_disk;   // 磁盘容量大小
    int sz_blks;   // 磁盘逻辑块大小，为1024B
    int sz_usage;

    int max_ino;   // inode数目
    uint8_t* map_inode;   // inode位图
    int map_inode_blks;   // inode位图所占的数据块
    int map_inode_offset;   // inode位图的起始地址

    int max_data;   // 数据块数目
    uint8_t* map_data;   // 数据块位图
    int map_data_blks;   // 数据块位图所占的数据块
    int map_data_offset;   // 数据块位图的起始地址

    int inode_offset;   // 索引节点块的起始地址
    int data_offset;   // 数据块的起始地址

    boolean is_mounted;

    struct nfs_dentry* root_dentry;   // 根目录

};
```

超级块 nfs_super 的 to-Disk 结构如下图所示：

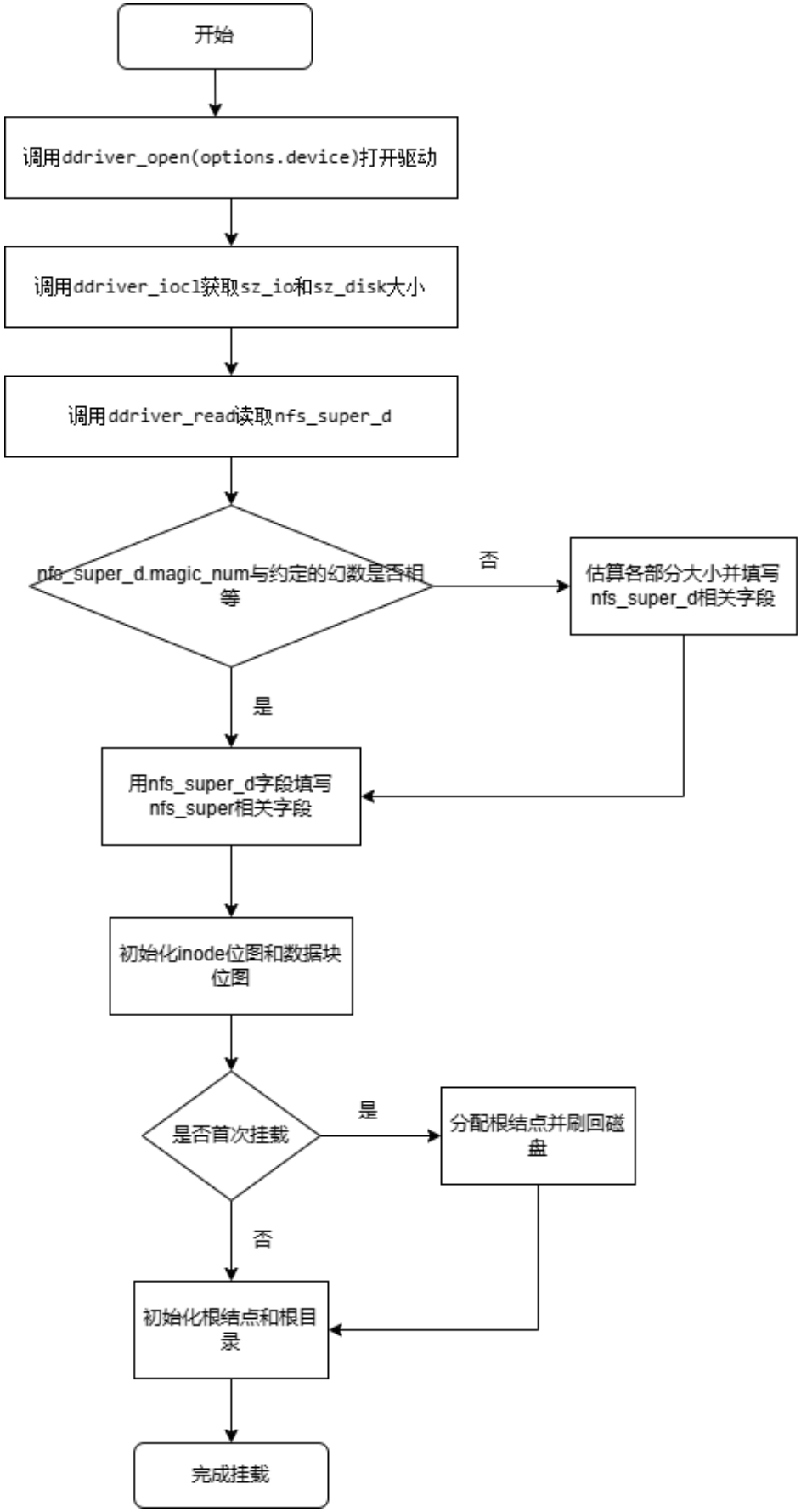
```
struct nfs_super_d{
    uint32_t magic_num;
    uint32_t sz_usage;

    int max_ino;   // inode数目
    int map_inode_offset;   // inode位图的起始地址
    int map_inode_blks;   // inode位图所占的数据块

    int max_data;   // 数据块数目
    int map_data_blks;   // 数据块位图所占的数据块
    int map_data_offset;   // 数据块位图的起始地址

    int inode_offset;   // 索引节点块的起始地址
    int data_offset;   // 数据块的起始地址

};
```

nfs_mount 函数首先会通过磁盘驱动 ddriver_open 函数尝试打开驱动，并通过磁盘驱动 ddriver_ioctl 函数向超级块 in-Mem 结构 nfs_super 中写入磁盘大小和单次 IO 大小，而 nfs_super.sz _blks 定义为 2 * nfs_super.sz_io，表明逻辑块大小为 2 个磁盘 io 大小，即 1024B。然后 nfs_mount 读取第一个逻辑块信息，大小为 sizeof(struct nfs_super_d)，并判断幻数是否与约定的幻数相等，若不等，则估算各部分所占空间大小，先填写 nfs_super_d 部分的信息，后再用 nfs_super_d 的相关字段填写 nfs_super 相关字段；若相等，则直接用 nfs_super_d 字段填写 nfs_super 字段。估算部分代码如下图所示：

```
if (nfs_super_d.magic_num != NFS_MAGIC_NUM) {     /* 幻数不正确，初始化 */
                                                  /* 估算各部分大小 */
    super_blks = NFS_SUPER_BLOCK_NUM;   // 超级块占用逻辑块数量

    inode_block_num  =  NFS_INODE_BLOCK_NUM;   // 索引节点占用逻辑块数量

    map_inode_blks = NFS_INODE_MAP_BLOCK_NUM;   // 索引节点位图占用逻辑块数量
                                                /* 布局layout */
    // 先填充nfs_super_d，后赋值给nfs_super
    nfs_super_d.max_ino = inode_block_num * 8;    // inode数目(一个逻辑块放置8个索引节点)
    nfs_super_d.magic_num = NFS_MAGIC_NUM;   // 幻数
    nfs_super_d.max_data = NFS_DATA_BLOCK_NUM;   // 数据块数量

    nfs_super_d.map_inode_offset = NFS_SUPER_OFS + NFS_BLKS_SZ(super_blks);   // inode位图起始地址
    nfs_super_d.map_data_offset = nfs_super_d.map_inode_offset + NFS_BLKS_SZ(map_inode_blks);   // 数据块位图起始地址

    nfs_super_d.map_inode_blks  = map_inode_blks;   // inode位图所占数据块数量
    nfs_super_d.map_data_blks = NFS_DATA_MAP_BLOCK_NUM;   // 数据块位图所占逻辑块数量

    nfs_super_d.inode_offset = nfs_super_d.map_data_offset + NFS_BLKS_SZ(nfs_super_d.map_data_blks);   // 索引节点块起始地址
    nfs_super_d.data_offset = nfs_super_d.inode_offset + NFS_BLKS_SZ(inode_block_num);   // 数据块起始地址

    nfs_super_d.sz_usage    = 0;
    NFS_DBG("inode map blocks: %d\n", map_inode_blks);
    is_init = TRUE;
}
```

后调用 nfs_driver_read 从磁盘对应位置读取索引结点位图和数据块位图，完成索引结点位图和数据块位图的初始化。若是初次挂载，则需要调用 nfs_alloc_inode 函数分配根结点并刷回磁盘保存，否则直接读取根目录完成根结点和根目录的初始化。

函数流程图如下图所示：


开始
调用ddriver_open(options.device)打开驱动
调用ddriver_iocl获取sz_io和sz_disk大小
调用ddriver_read读取nfs_super_d
nfs_super_d.magic_num与约定的幻数是否相等
否
估算各部分大小并填写nfs_super_d相关字段
是
用nfs_super_d字段填写nfs_super相关字段
初始化inode位图和数据块位图
是否首次挂载
是
分配根结点并刷回磁盘
否
初始化根结点和根目录
完成挂载

- nfs_umount：

nfs_umount 函数实现文件系统的卸载。文件系统卸载需要将内存中数据结构的信息写回到磁盘中。在 nfs_umount 函数中，首先调用 nfs_sync_inode(nfs_super.root_dentry->inode)从根结点向下刷写节点，将其刷回磁盘；后通过 nfs_super 字段填写 nfs_super_d 字段，并将 nfs_super_d 写回磁盘中；再将 inode 位图和数据块位图写入磁盘中，并释放内存中的 inode 位图和数据块位图，最后使用磁盘驱动 ddriver_close 函数关闭驱动，完成文件系统的卸载。

- nfs_get_fname：

nfs_get_fname 函数实现获取路径中的文件名。函数通过 strrchr 函数获取路径中最右侧"/"后面的字符串实现获取文件名。

- nfs_calc_lvl：

nfs_calc_lvl 函数实现计算路径的层级。nfs_calc_lvl 函数通过路径中"/"的数量确定路径层级，如果"/"数量为 1，则为根目录，路径层级为 0，否则路径层级为路径中"/"的数量加 1。

- nfs_driver_read：

nfs_driver_read 实现驱动读。nfs_driver_read 函数根据传入的待读取数据段在磁盘中的偏移 offset 和读取的数据段大小 size，通过 NFS_ROUND_DOWN 和 NFS_ROUND_UP 函数计算出要读取的逻辑块的下界 down 和上界 up，使用磁盘驱动 ddriver_read 函数将需要访问逻辑块的内容全部读取出来，然后从这一部分读出的数据中读出相应的数据，存放到 out_content 中。其中 NFS_ROUND_DOWN 和 NFS_ROUND_UP 函数定义如下图所示：

```
#define NFS_ROUND_DOWN(value, round)    ((value) % (round) == 0 ? (value) : ((value) / (round)) * (round))   // 不超过value中round的最大倍数
#define NFS_ROUND_UP(value, round)      ((value) % (round) == 0 ? (value) : ((value) / (round) + 1) * (round))   // 不小于value中round的最小倍数
```

- nfs_driver_write：

nfs_driver_write 实现驱动写。nfs_driver_write 函数根据传入的写回的目标地址 offset 和写回的数据段大小 size，通过 NFS_ROUND_DOWN 和 NFS_ROUND_UP 函数计算出要读取的逻辑块的下界 down 和上界 up，使用 nfs_driver_read 先将逻辑块内容读取出来，后用写回的数据 in_content 替换逻辑块中对应位置的内容，最后使用磁盘驱动 ddriver_write 函数将读取出的内容全部写回到磁盘中。

- nfs_alloc_dentry：

nfs_alloc_dentry 函数实现将 dentry 插入到 inode 中。nfs_alloc_dentry 函数采用头插法插入 dentry，并更新 inode 的目录项个数 dir_cnt 和文件所占用空间大小 size。若 inode 原有的数据块已满或初始时未分配数据块（即 inode->dir_cnt % (NFS_BLKS_SZ(1) / sizeof(struct nfs_dentry)) == 1），则调用 nfs_alloc_data 函数分配一个新的数据块，让 block_index[i]存储分配的数据块号，并更新 inode 中的 block_num 字段。

- nfs_alloc_inode：

nfs_alloc_inode 函数实现分配一个 inode，占用 inode 位图。索引结点 inode 的 in-Mem 结构如下图所示：

```c
struct nfs_inode {
    uint32_t ino;   // 在inode位图中的下标
    /* TODO: Define yourself */
    int size;   // 文件已占用空间大小
    int  dir_cnt;   // 目录项个数
    struct nfs_dentry* dentry;    // 指向该inode的dentry
    struct nfs_dentry* dentrys;   // 所有目录项
    int block_num;   // 已分配数据块数量
    int block_index[6];   // 数据块在磁盘中的块号
    uint8_t* block_pointer[6];   // 数据块指针(假设每个文件最多直接索引6个逻辑块来填写文件数据)
};
```

nfs_alloc_inode 函数首先会检查 inode 位图中的每一位，查找是否有值为 0 的空闲位，若有则将该位置 1；若无空闲位或空闲位超出 inode 位图范围则报错。然后为分配的 inode 开辟内存空间，并填写 inode 相关信息完成 inode 初始化，inode 初始化部分代码如下图所示：

```c
inode->ino  = ino_cursor;
inode->size = 0;
inode->block_num = 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;
```

- nfs_alloc_data：

nfs_alloc_data 函数实现分配一个数据块，占用数据块位图。nfs_alloc_data 函数首先会检查数据块位图中的每一位，查找是否有值为 0 的空闲位，若有则将该位置 1；若无空闲位或空闲位超出数据块位图范围则报错，最后返回分配的数据块号。

- nfs_sync_inode：

nfs_sync_inode 函数实现将内存 inode 及其下方结构全部刷回磁盘中。索引结点 inode 的 to-Disk 结构如下图所示：

```c
struct nfs_inode_d{
    uint32_t ino;   // 在inode位图中的下标
    int size;   // 文件已占用空间大小
    int  dir_cnt;   // 目录项个数
    int block_num;   // 已分配数据块数量
    int block_index[6];   // 数据块在磁盘中的块号
    NFS_FILE_TYPE      ftype;   // 文件类型
};
```

目录项 dentry 的 to-Disk 结构如下图所示：

```
struct nfs_dentry_d{
    char      name[MAX_NAME_LEN];
    uint32_t ino;
    NFS_FILE_TYPE      ftype;    // 文件类型

};
```

nfs_sync_inode 函数首先会根据 inode 的字段填写 inode_d 相关字段信息，把 inode_d 写回磁盘中。如果 inode 是目录，那么数据是目录项，则依次遍历 inode 中的目录项，填写 dentry_d 的相关字段信息，将 dentry_d 写回到对应的数据块中，若目录项不为空，则递归调用 nfs_sync_inode，写回目录项对应的 inode；如果 inode 是文件，那么数据是文件内容，直接把已分配的数据块内容写回对应的磁盘逻辑块中。

- nfs_read_inode：

nfs_read_inode 函数实现从磁盘中读取 inode 节点。nfs_read_inode 函数先从磁盘中读取索引结点 inode_d，然后根据 inode_d 的字段填写 inode 的相应字段信息。如果 inode 是目录，则将其子目录项部分从磁盘中依次读出，建立 dentry 的 in-Mem 结构，并通过 nfs_alloc_dentry 函数将 dentry 插入到 inode 中；如果 inode 是文件，则将其有效的数据块内容读取出来。

目录项 dentry 的 in-Mem 结构如下图所示：

```
struct nfs_dentry {
    char      name[MAX_NAME_LEN];
    uint32_t ino;
    /* TODO: Define yourself */
    struct nfs_dentry* parent;    // 父亲Inode的dentry
    struct nfs_dentry* brother;    // 兄弟
    struct nfs_inode*  inode;    // 指向inode
    NFS_FILE_TYPE      ftype;
};
```

- nfs_get_dentry：

nfs_get_dentry 函数实现获取 inode 的第 dir（从 0 开始）个目录项。nfs_get_dentry 函数遍历 inode 的 dentry 链表，找到第 dir 个目录项并返回。

- nfs_lookup：

nfs_lookup 函数实现查找路径对应的文件或目录，如果能查找到，返回该目录项；如果查找不到，返回的是上一个有效的路径。nfs_lookup 函数从根目录开始查找，首先计算路径的层级，若路径层级为 0，则查找的是根目录，返回根目录；若不是，则获取

当前 dentry_cursor 指向的 inode，如果其是普通文件且未遍历到目标层数，则报错，返回上一级路径，如果是目录，遍历其子目录项，查找是否匹配当前的目录名 fname，其中 fname 为当前层级下路径对应的文件名。在每一层目录中，如果找到了对应的目录项，检查是否已经到达目标层级，如果是，设置 is_find = TRUE 并返回该目录项；否则继续查找下一层级；如果在目标层级没有找到目标目录项，设置 is_find = FALSE，并返回上一个有效的路径。如果 dentry_ret 最终未找到有效的 inode，调用 nfs_read_inode 函数读取该 inode。

用户接口部分

- nfs_init：

  nfs_init 通过调用 nfs_mount 实现文件系统的挂载。

- nfs_destory:

  nfs_destory 函数通过调用 nfs_umount 函数实现文件系统的卸载。

- nfs_mkdir：

  nfs_mkdir 函数实现创建目录。nfs_mkdir 函数首先通过 nfs_lookup 函数解析路径，若目录已存在，则报错，否则获取上一级路径的目录项，如果上一级路径是普通文件，则不能创建目录，报错；然后通过 new-dentry 函数创建一个新的目录项，并使用 nfs_alloc_inode 函数为该目录项分配一个索引结点，通过 nfs_alloc_dentry 函数将 dentry 插入到上一级索引结点中，完成目录的创建。

  new_dentry 函数实现创建目录项，函数代码如下图所示：

```
// 函数功能：创建目录项
static inline struct nfs_dentry* new_dentry(char * fname, NFS_FILE_TYPE ftype) {
    struct nfs_dentry * dentry = (struct nfs_dentry *)malloc(sizeof(struct nfs_dentry));
    memset(dentry, 0, sizeof(struct nfs_dentry));
    NFS_ASSIGN_FNAME(dentry, fname);
    dentry->ftype   = ftype;
    dentry->ino     = -1;
    dentry->inode   = NULL;
    dentry->parent  = NULL;
    dentry->brother = NULL;
    return dentry;
}
```

- nfs_getattr：

  nfs_getattr 函数实现获取文件或目录的属性。nfs_getattr 函数首先通过 nfs_lookup 函数获取当前路径对应的目录项，然后以此填写 nfs_stat 结构体相关字段信息，相关代码如下图所示：

```
if (NFS_IS_DIR(dentry->inode)) {    // inode对应的是普通文件，设置其属性
    nfs_stat->st_mode = S_IFDIR | NFS_DEFAULT_PERM;
    nfs_stat->st_size = dentry->inode->dir_cnt * sizeof(struct nfs_dentry_d);
}
else if (NFS_IS_REG(dentry->inode)) {    // inode对应的是目录，设置其属性
    nfs_stat->st_mode = S_IFREG | NFS_DEFAULT_PERM;
    nfs_stat->st_size = dentry->inode->size;
}
// else if (SFS_IS_SYM_LINK(dentry->inode)) {    // 实验无需考虑软链接和硬链接的实现
//  sfs_stat->st_mode = S_IFLNK | SFS_DEFAULT_PERM;
//  sfs_stat->st_size = dentry->inode->size;
// }

nfs_stat->st_nlink = 1;
nfs_stat->st_uid     = getuid();
nfs_stat->st_gid     = getgid();
nfs_stat->st_atime   = time(NULL);
nfs_stat->st_mtime   = time(NULL);
nfs_stat->st_blksize = NFS_BLKS_SZ(1);
nfs_stat->st_blocks = NFS_DATA_PER_FILE;

if (is_root) {
    nfs_stat->st_size   = nfs_super.sz_usage;
    nfs_stat->st_blocks = NFS_DISK_SZ() / (NFS_IO_SZ() * 2);
    nfs_stat->st_nlink  = 2;        /* !特殊，根目录link数为2 */
}
```

- nfs_readdir：

nfs_readdir 函数实现遍历目录项，填充至 buf，并交给 FUSE 输出。nfs_ readdir 函数首先通过 nfs_lookup 函数获取当前路径对应的目录项，并找到目录项指向的索引结点 inode，然后调用 nfs_get_dentry 函数遍历 inode 的 dentry 链表，使用 filler(buf, sub_dentry->name, NULL, ++offset);进行输出。

- nfs_mknod：

nfs_mknod 函数实现创建文件。nfs_mknod 函数首先通过 nfs_lookup 函数获取当前路径对应的目录项，后使用 nfs_get_fname 函数获取文件名。根据创建文件的模式，使用 new_dentry 函数为文件创建目录项，并使用 nfs_alloc_inode 函数为该目录项分配一个索引结点，通过 nfs_alloc_dentry 函数将 dentry 插入到上一级索引结点中，完成文件的创建。

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。

相较于指导书上的磁盘布局，本文件系统采取一个逻辑块存放 8 个索引结点，提高了磁盘空间利用效率，使文件系统能支持存放更多的文件。此外，对于索引结点数据结构 inode，设置了 int block_num 成员记录 inode 已分配的数据块数量，在读入和写回 inode 时均只读取和写回已分配的数据块，提高了读取和写回效率，并更直观地实现按需分配数据块。

二、遇到的问题及解决方法

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

在设计超级块等数据结构时，由于初次接触文件系统的设计，不是很清楚结构体内应该包含哪些成员。对于这个问题，我的解决策略是参照提供的样例 simplefs，先大致设计结构体内应该包含的成员，然后在具体功能实现时，在根据代码的要求进一步优化数据结构，同时，参照 simplefs 的超级块中有关 inode 位图的成员变量，设计有关数据块位图的成员变量。

在编写代码过程中，刚开始对于 inode 的读取和写回都是简单粗暴的直接读取和写回 NFS_DATA_PER_FILE 个数据块（即不管是否有分配数据块，均读取或写回），导致位图检查出错以及 remount 测试不通过。我的解决策略是通过提供的磁盘管理工具，利用 Hex Editor 插件查看虚拟磁盘的内容，同时结合 VScode 的调试工具，查看 inode 读取和写回时相关结构体的内容。通过上述策略，我发现之前的读取和写回方式会使空白的数据块覆盖先前写回磁盘的内容，从而导致错误的发生。因此，在索引结点数据结构中，我新增了 block_num 成员，记录 inode 已分配的数据块数量，在读入和写回 inode 时均只读取和写回已分配的数据块，解决了上述问题。

三、实验收获和建议

实验中的收获、感受、问题、建议等。

在这次操作系统实验中，我通过设计和实现基于 FUSE 的青春版 EXT2 文件系统，不仅加深了对文件系统工作原理的理解，也提升了自己在实际开发中的动手能力。从文件系统的基本构造、磁盘数据块的管理到实现相关功能，我经历了一个完整的开发过程。通过实现这些功能，我更加清楚地认识到文件系统在操作系统中的核心地位，理解了 I/O 操作、硬件交互等关键概念。同时，实验过程中遇到的调试问题让我提高了对代码的分析和排查能力，让我对 VScode 的调试工具更加熟悉。

我的建议是实验中的一些文档和资料可以更加清晰，也可以提供更多有关 EX2 文件系统结构的资料，辅助学生更快地理解代码框架，有助于提高开发效率。

总的来说，这次实验是我参加过最具挑战性的实验之一，当 debug 了很久的代码终于通过测试时，成就感满满。

四、参考资料

实验过程中查找的信息和资料

操作系统实验指导书 - 2024 秋 | 哈工大（深圳）

https://os-labs.pages.dev/lab5/part1/