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：计科 7 班

学生学号：220110720

学生姓名：刘睿

评阅教师：

报告成绩：

实验与创新实践教育中心制

2024 年 9 月

一、实验详细设计

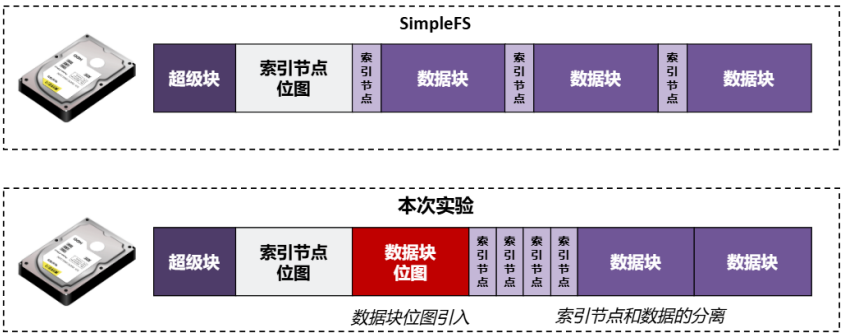
图文并茂地描述实验实现的所有功能和详细的设计方案及实验过程中的特色部分。

1、 总体设计方案

详细阐述文件系统的总体设计思路，包括系统架构图和关键组件的说明。

本次实验旨在通过以 Linux 系统中的 EXT2 文件系统为例，熟悉该文件系统内部数据结构的组织方式和基本处理流程。通过基于 FUSE（Filesystem in Userspace）的设计与实现，我们将开发一个可以在 Linux 上运行的文件系统。该系统应具备挂载、卸载、创建文件、创建文件夹以及查看文件夹下文件等基本的的功能。

1.1 磁盘布局设计



1.1.1 逻辑块大小

首先，我们确定了逻辑块的大小为 1024B。这意味着每个逻辑块可以存储 1024 字节的数据。由于磁盘总容量为 4MB，逻辑块数为：

逻辑块数 = 4MB/1024B = 4096

1.1.2 磁盘容量估算

为了估算磁盘可以存放的文件数量，我们进行了如下计算：

- 每个文件最多直接索引 4 个逻辑块，即每个文件的数据上限为：
  4 × 1024B = 4KB
- 假设一个逻辑块可以存储 16 个文件的索引节点，每个索引节点占用 64B，因此维护一个文件所需的存储容量为：
  4KB + 64B = 4096B + 64B = 4160B
- 因此，4MB 磁盘最多可以存放的文件数为：
  4MB / 4160B = 4×1024KB / 4160B ≈ 1008

1.1.3 与提供的 simplefs 实现的对比

Simplefs：

采用固定分配的方式来维护文件，因此无需数据块位图，一个文件固定分配 1 个逻辑块当索引节点，16 个逻辑块当数据块，索引节点和数据块一起放置，简单便于索引，但利用率不高

本次实验：

将一个文件的索引节点和数据进行了分离，形成索引节点区和数据块区，灵活为每

个文件按需分配数据块，但也需要数据块位图来记录数据块分配情况。由于后续布局检查会基于 1024B 逻辑块进行检查，因此我们不能简单照搬 simplefs 的实现

1.1.4 各组件的功能与大小设计

超级块

超级块是存放文件系统全局性的数据结构。它包含了文件系统的重要信息，用于识别磁盘上是否有文件系统、文件系统类型、文件是否损坏等。超级块包含整个文件系统和磁盘布局的总体信息 占用 1 个逻辑块即 1 024B

索引节点位图

索引节点位图用于记录索引节点表的使用情况，每个索引节点的使用状态通过一个比特位表示。由于文件系统最多支持 1008 个文件维护，一个逻辑块（1024B）的位图可以管理 1024 * 8 = 8192 个索引节点，因此索引节点位图仅占用 1 个逻辑块（1024B）便足够

数据块位图

数据块位图用于记录数据块的使用情况，每个数据块的使用状态通过一个比特位表示。根据索引节点位图可知，数据块位图也仅占用 1 个逻辑块（1024B）便足够

索引节点

索引节点记录文件的元数据，每个文件都与一个 inode 对应。一个 inode 可能对应多个文件（硬链接）。假设一个逻辑块存储可以存储 16 个文件的索引节点，因此 1008 个文件最多需要 64 个逻辑块便可以全部存储

数据块

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（间接索引本次实验不做考虑）。磁盘中的剩余空间均作为数据块，还剩 4096 - 1 - 1 - 1 - 64 = 4029 个逻辑块

综上所述，我们的整体磁盘布局设计如图：

```
50    // 磁盘布局设计,一个逻辑块能放8个inode
51    #define NFS_INODE_PER_BLK       8     // 一个逻辑块能放8个inode
52    #define NFS_SUPER_BLKS          1
53    #define NFS_INODE_MAP_BLKS      1
54    #define NFS_DATA_MAP_BLKS       1
55    #define NFS_INODE_BLKS          64    // 4096/8/8(磁盘大小为4096个逻辑块，维护一个文件需要8个逻辑块即一个索引块+七个数据块)
56    #define NFS_DATA_BLKS           4029  // 4096-64-1-1-1
```

由此填写 layout 文件如下：

```
22      | BSIZE = 1024 B |
23      | Super(1) | Inode Map(1) | DATA Map(1) | INODE(64) | DATA(*) |
```

1.2 模拟磁盘与模拟驱动程序

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 。 虚拟磁盘 是通过一个普通的数据文件来充当，这个数据文件的路径位于 ~/ddriver 。我们封装了一层 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

DDRIVER 驱动封装实现了 open，close，seek，read，write，ioctl 的方法，我们仅需要在文件系统实现中直接调用这些驱动接口来和 磁盘进行读写交互 即可。相关的代

码位于 user-land-filesystem/driver/user_ddriver/ddriver.c

本次实验与 simplefs 不同的是，simplefs 的逻辑块大小与驱动读写 IO 相等，即按 512B 来封装，而 EXT2 文件系统的逻辑块大小为 1024B，也就是两个 IO 单位

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

2.1.1 超级块

内存中的超级块：

为运行时使用而设计，包含了文件系统的全局信息，并且添加了一些仅在内存中需要的信息或指针，例如：

- 文件描述符 (fd): 用于标识打开的虚拟磁盘文件
- 位图的内存起点 (map_inode, map_data): 指向实际的位图数据，这些数据是在内存中操作的
- 根目录项 (root_dentry): 指向根目录的 dentry，方便快速访问文件系统的根
- 是否挂载 (is_mounted): 标记文件系统当前是否已挂载

这些字段对于在用户空间中实现文件系统来说是非常必要的，因为它们允许文件系统在运行时动态地管理和访问文件系统的资源

内存中超级块的具体结构定义如下：

```
90  struct nfs_super {
91      uint32_t          magic;            // 幻数
92      int               fd;               // 文件描述符
93
94      int               sz_io;            // 512B
95      int               sz_blks;          // 1KB
96      int               sz_disk;          // 4MB
97      int               sz_usage;         // 已使用空间大小
98
99      int               max_ino;          // 索引节点最大数量
100     uint8_t*          map_inode;        // inode位图内存起点
101     int               map_inode_blks;   // inode位图占用的逻辑块数量
102     int               map_inode_offset; // inode位图在磁盘中的偏移
103
104     int               max_dno;          // 数据位图最大数量
105     uint8_t*          map_data;         // data位图内存起点
106     int               map_data_blks;    // data位图占用的逻辑块数量
107     int               map_data_offset;  // data位图在磁盘中的偏移
108
109     int               inode_offset;     // inode在磁盘中的偏移
110     int               data_offset;      // data在磁盘中的偏移
111
112     boolean           is_mounted;       // 是否挂载
113     struct nfs_dentry* root_dentry;     // 根目录dentry
114 };
```

磁盘上的超级块：

相比之下，磁盘上的超级块则只包含持久化存储所需的基本信息，如：

- 幻数 (magic): 用于识别文件系统类型
- 已使用的空间大小 (sz_usage): 表示文件系统已经使用的空间
- 索引节点位图、数据位图以及 inode 和 data 的偏移量等信息

磁盘上的超级块不包括任何指向其他数据结构的指针，因为这些信息是不需要持久

化的。当文件系统被卸载后，所有在内存中的状态都会丢失，因此只有那些真正需要保存到磁盘上的信息才会出现在磁盘超级块中

磁盘上超级块的具体结构定义如下：

```
struct nfs_super_d {
    uint32_t        magic;          // 幻数
    int             sz_usage;       // 已使用空间大小

    int             max_ino;        // 索引节点最大数量
    int             map_inode_blks; // inode位图占用的逻辑块数量
    int             map_inode_offset; // inode位图在磁盘中的偏移

    int             max_dno;        // 数据位图最大数量
    int             map_data_blks;  // data位图占用的逻辑块数量
    int             map_data_offset; // data位图在磁盘中的偏移

    int             inode_offset;   // inode在磁盘中的偏移
    int             data_offset;    // data在磁盘中的偏移
};
```

2.1.2 索引节点

内存中的索引节点：

为运行时使用而设计，不仅包含了文件的基本元数据信息，还包含了一些额外的信息或指针，这些信息主要用于内存中的快速访问和管理

内存中的索引节点结构允许文件系统快速查找和更新文件的相关信息，并且可以方便地遍历文件树结构。此外，它还维护了与文件内容相关的指针，如数据块指针等，这对于实现高效的文件读写操作至关重要

内存中的索引节点具体定义如下：

```
struct nfs_inode {
    uint32_t        ino;                        // 索引编号
    int             size;                       // 文件占用空间
    int             link;                       // 连接数默认为1(不考虑软链接和硬链接)
    int             block_pointer[NFS_DATA_PER_FILE]; // 数据块索引
    int             dir_cnt;                    // 如果是目录型文件，则代表有几个目录项
    struct nfs_dentry* dentry;                  // 指向该inode的父dentry
    struct nfs_dentry* dentrys;                 // 指向该inode的所有子dentry
    NFS_FILE_TYPE   ftype;                      // 文件类型
    uint8_t*        data[NFS_DATA_PER_FILE];    // 指向数据块的指针
    int             block_allocated;            // 已分配数据块数量
};
```

磁盘上的索引节点：

相比之下，磁盘上的索引节点则只包含持久化存储所需的基本信息，即那些必须保存到磁盘上以确保文件系统状态可以在重启后恢复的信息

磁盘上的索引节点结构简化了很多，它不包含任何指向其他数据结构的指针（例如，没有对父 dentry 或子 dentry 的引用），因为这些信息不需要持久化。当文件系统被卸载或者发生崩溃时，所有在内存中的状态都会丢失，因此只有那些真正需要保存到磁盘上的信息才会出现在磁盘索引节点中

磁盘上的索引节点具体定义如下：

```
struct nfs_inode_d {
    uint32_t        ino;                        // 索引编号
    int             size;                       // 文件占用空间(用了多少个逻辑块)
    int             link;                       // 连接数默认为1(不考虑软链接和硬链接)
    int             block_pointer[NFS_DATA_PER_FILE]; // 数据块索引
    int             dir_cnt;                    // 如果是目录型文件，则代表有几个目录项
    NFS_FILE_TYPE   ftype;                      // 文件类型
    int             block_allocated;            // 已分配数据块数量
};
```

2.1.3 目录项

内存中的目录项：

为运行时使用而设计的，它不仅包含了目录项的基本信息，还包含了一些额外的信息或指针，这些信息主要用于内存中的快速访问和管理

内存中的目录项结构允许文件系统快速查找、添加、删除或更新目录项，并且可以方便地遍历文件树结构。此外，通过维护对父目录和其他同级目录项的引用，实现了高效的路径解析和目录遍历功能

内存中的目录项具体定义如下：

```
129  struct nfs_dentry {
130      char               fname[NFS_MAX_FILE_NAME];    // dentry指向的文件名
131      struct nfs_dentry* parent;                      // 父目录的dentry
132      struct nfs_dentry* brother;                     // 兄弟dentry
133      int                ino;                         // 指向的inode编号
134      struct nfs_inode*  inode;                       // 指向的inode
135      NFS_FILE_TYPE      ftype;                       // 文件类型
136  };
```

磁盘上的目录项：

相比之下，磁盘上的目录项则主要包含持久化存储所需的基本信息，即那些必须保存到磁盘上以确保文件系统状态可以在重启后恢复的信息

磁盘上的目录项结构简化了很多，它不包含任何指向其他数据结构的指针（例如，没有对父 dentry 或兄弟 dentry 的引用），因为这些信息不需要持久化。当文件系统被卸载或者发生崩溃时，所有在内存中的状态都会丢失，因此只有那些真正需要保存到磁盘上的信息才会出现在磁盘目录项中

磁盘上的目录项具体定义如下：

```
180  struct nfs_dentry_d {
181      char               fname[NFS_MAX_FILE_NAME];    // dentry指向的文件名
182      int                ino;                         // 指向的inode编号
183      NFS_FILE_TYPE      ftype;                       // 文件类型
184  };
```

2.2 核心代码

2.2.1 newfs.c

newfs.c 实现了一个基于 FUSE（用户空间文件系统）的自定义文件系统，允许在用户空间中进行文件和目录的创建、删除、读写等操作。提供了挂载和卸载文件系统的功能，支持获取文件属性、遍历目录内容，并简化了时间戳更新的操作。

核心函数的功能和设计如下：

newfs_init()：

在文件系统被挂载时调用。它首先尝试通过 nfs_mount 函数来初始化文件系统。如果初始化失败（返回非零值），则打印错误信息并通过 fuse_exit 终止 FUSE 进程

```
54      void* newfs_init(struct fuse_conn_info *conn_info) {
55              /* 初始化文件系统 */
56              if (nfs_mount(nfs_options) != 0) {
57                      NFS_DBG("[%s] mount error\n", __func__);
58                      fuse_exit(fuse_get_context()->fuse);
59                      return NULL;
60              }
61
62              return NULL;
63      }
```

newfs_destroy()：

当文件系统卸载时调用。使用 nfs_umount 卸载文件系统，并检查是否成功。如果有错误发生，则记录日志并退出 FUSE 进程

```
71      void newfs_destroy(void* p) {
72              /* 清理文件系统资源 */
73              if (nfs_umount() != NFS_ERROR_NONE) {
74                      NFS_DBG("[%s] unmount error\n", __func__);
75                      fuse_exit(fuse_get_context()->fuse);
76                      return;
77              }
78      }
```

newfs_mkdir():

接收一个路径参数和模式位，查找父级 dentry（目录条目）。如果父级是普通文件，则不允许创建子目录；否则，创建新的 dentry 对象并将其链接到父级，同时分配 inode（索引节点）

```
87      int newfs_mkdir(const char* path, mode_t mode) {
88              boolean is_find, is_root;
89              char* fname;
90              struct nfs_dentry* last_dentry = nfs_lookup(path, &is_find, &is_root);
91              struct nfs_dentry* dentry;
92              struct nfs_inode* inode;
93
94              // 如果目录已经存在，返回错误
95              if (is_find) {
96                      return -NFS_ERROR_EXISTS;
97              }
98
99              // 如果上级文件是普通文件，则不能创建子目录
100             if (NFS_IS_REG(last_dentry->inode)) {
101                     return -NFS_ERROR_UNSUPPORTED;
102             }
103
104             // 提取文件名并创建新目录
105             fname  = nfs_get_fname(path);
106             dentry = new_dentry(fname, NFS_DIR);
107             dentry->parent = last_dentry;
108             inode  = nfs_alloc_inode(dentry);
109             nfs_alloc_dentry(last_dentry->inode, dentry);
110
111             return NFS_ERROR_NONE;
112     }
```

newfs_mknod():

根据提供的模式决定创建普通文件还是目录。提取文件名并创建相应的 dentry 和 inode。对于普通文件，默认处理方式是创建常规文件节点

```c
int newfs_mknod(const char* path, mode_t mode, dev_t dev) {
        boolean is_find, is_root;

        struct nfs_dentry* last_dentry = nfs_lookup(path, &is_find, &is_root);
        struct nfs_dentry* dentry;
        struct nfs_inode* inode;
        char* fname;

        // 如果文件已经存在，返回错误
        if (is_find == TRUE) {
                return -NFS_ERROR_EXISTS;
        }

        // 提取文件名并创建新文件
        fname = nfs_get_fname(path);

        if (S_ISREG(mode)) {
                dentry = new_dentry(fname, NFS_REG_FILE);
        } else if (S_ISDIR(mode)) {
                dentry = new_dentry(fname, NFS_DIR);
        } else {
                dentry = new_dentry(fname, NFS_REG_FILE);
        }
        dentry->parent = last_dentry;
        inode = nfs_alloc_inode(dentry);
        nfs_alloc_dentry(last_dentry->inode, dentry);

        return NFS_ERROR_NONE;
}
```

newfs_getattr():

通过 nfs_lookup 查找指定路径对应的 dentry。然后基于该 dentry 的类型（目录或普通文件）设置适当的属性，如权限、大小等。对于根目录，特别设置了链接数和其他特定属性

```c
int newfs_getattr(const char* path, struct stat *nfs_stat) {
        boolean is_find, is_root;
        struct nfs_dentry* dentry = nfs_lookup(path, &is_find, &is_root);

        // 如果找不到对应文件，返回错误
        if (!is_find) {
                return -NFS_ERROR_NOTFOUND;
        }

        // 根据文件类型设置相应的属性
        if (NFS_IS_DIR(dentry->inode)) {
                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)) {
                nfs_stat->st_mode = S_IFREG | NFS_DEFAULT_PERM;
                nfs_stat->st_size = dentry->inode->size;
        }

        // 设置其他属性
        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_IO_SZ();
        nfs_stat->st_blocks  = NFS_DATA_PER_FILE;

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

newfs_readdir():

利用 offset 参数确定从哪个位置开始读取目录项。每次调用都会尝试获取下一个 dentry，并通过 filler 回调函数将结果添加到输出缓冲区中。如果找不到更多的 dentry，则停止

```
175     int newfs_readdir(const char *path, void *buf, fuse_fill_dir_t filler, off_t offset,
176                                             struct fuse_file_info *fi) {
177         boolean is_find, is_root;
178             int cur_dir = offset;
179 
180             struct nfs_dentry* dentry = nfs_lookup(path, &is_find, &is_root);
181             struct nfs_dentry* sub_dentry;
182             struct nfs_inode* inode;
183 
184             // 如果找到指定路径下的文件
185             if (is_find) {
186                     inode = dentry->inode;
187                     // 根据当前offset找到对应的dentry
188                     sub_dentry = nfs_get_dentry(inode, cur_dir);
189                     if (sub_dentry) {
190                             // 调用filler填装结果到buf中
191                             filler(buf, sub_dentry->fname, NULL, ++offset);
192                     }
193                     return NFS_ERROR_NONE;
194             }
195             return -NFS_ERROR_NOTFOUND;
196     }
```

2.2.2 newfs_utils.c

newfs_utils.c 中的各种功能函数共同构成了一个自定义文件系统 newfs 的核心操作集，这些函数负责从底层磁盘 I/O 到高层路径解析的各个方面。以下是主要功能函数的分析

nfs_get_fname():

实现从路径字符串中提取文件名。使用 strrchr 函数找到路径字符串中最后一个出现的斜杠（/），然后返回其后的部分作为文件名。如果路径以斜杠结尾，则返回斜杠后的空字符串。在创建新文件或目录时（如 newfs_mkdir 和 newfs_mknod）使用此函数

```
12  char* nfs_get_fname(const char* path) {
13      // strrchr函数返回一个指针，指向字符串中最后一个出现的字符'/'的位置(逆向搜索)
14      char *q = strrchr(path, '/') + 1;
15      return q;
16  }
17
18  /**
19   * @brief 计算路径的层级
20   * 示例: /av/c/d/f -> lvl = 4
21   * @param path 路径字符串
22   * @return 路径层级数
23   */
24  int nfs_calc_lvl(const char *path) {
25      int lvl = 0;
26      // 根目录
27      if (strcmp(path, "/") == 0) {
28          return lvl;
29      }
30      for (const char *str = path; *str != '\0'; str++) {
31          if (*str == '/') {
32              lvl++;
33          }
34      }
35      return lvl;
36  }
```

nfs_calc_lvl():

实现计算路径的层级数。遍历路径字符串中的每个字符，每当遇到斜杠（/）时增加计数器。对于根目录（"/"），直接返回 0。辅助进行路径解析

```
24  int nfs_calc_lvl(const char *path) {
25      int lvl = 0;
26      // 根目录
27      if (strcmp(path, "/") == 0) {
28          return lvl;
29      }
30      for (const char *str = path; *str != '\0'; str++) {
31          if (*str == '/') {
32              lvl++;
33          }
34      }
35      return lvl;
36  }
```

nfs_driver_read() 和 nfs_driver_write():

实现底层磁盘驱动的读写操作。

• 对齐偏移量和大小到逻辑块边界，确保每次读写都是按照固定大小的块进行。

• 使用临时缓冲区来处理未对齐的数据部分，并通过多次调用磁盘驱动的读写接口完成整个操作。

• nfs_driver_read 先定位磁盘头，然后分段读取数据；nfs_driver_write 则先读出旧内容，修改后再写回磁盘

```
46  int nfs_driver_read(int offset, uint8_t *out_content, int size) {
47      // 对齐，按一个逻辑块大小(两个IO大小)进行读写
48      int offset_aligned = NFS_ROUND_DOWN(offset, NFS_BLK_SZ());
49      int bias           = offset - offset_aligned;
50      int size_aligned   = NFS_ROUND_UP((size + bias), NFS_BLK_SZ());
51      uint8_t* temp_content   = (uint8_t*)malloc(size_aligned);
52      uint8_t* cur            = temp_content;
53
54      // 磁盘头定位到down位置
55      ddriver_seek(NFS_DRIVER(), offset_aligned, SEEK_SET);
56      // 按照IO大小进行读，从down开始读size_aligned大小的内容
57      while (size_aligned > 0) {
58          ddriver_read(NFS_DRIVER(), cur, NFS_IO_SZ());
59          cur          += NFS_IO_SZ();
60          size_aligned -= NFS_IO_SZ();
61      }
62      // 从down+bias开始拷贝size大小的内容到out_content
63      memcpy(out_content, temp_content + bias, size);
64      free(temp_content);
65      return NFS_ERROR_NONE;
66  }
```

```
76   int nfs_driver_write(int offset, uint8_t *in_content, int size) {
77       // 对齐，按一个逻辑块大小(两个IO大小)进行读写
78       int offset_aligned = NFS_ROUND_DOWN(offset, NFS_BLK_SZ());
79       int bias           = offset - offset_aligned;
80       int size_aligned   = NFS_ROUND_UP((size + bias), NFS_BLK_SZ());
81       uint8_t* temp_content   = (uint8_t*)malloc(size_aligned);
82       uint8_t* cur            = temp_content;
83       // 读出被写磁盘块到内存
84       nfs_driver_read(offset_aligned, temp_content, size_aligned);
85       // 从down+bias开始覆盖size大小的内容
86       memcpy(temp_content + bias, in_content, size);
87       // 磁盘头定位到down
88       ddriver_seek(NFS_DRIVER(), offset_aligned, SEEK_SET);
89
90       // 内容在内存中修改后写回磁盘
91       while (size_aligned > 0) {
92           ddriver_write(NFS_DRIVER(), cur, NFS_IO_SZ());
93           cur          += NFS_IO_SZ();
94           size_aligned -= NFS_IO_SZ();
95       }
96
97       free(temp_content);
98       return NFS_ERROR_NONE;
99   }
```

nfs_alloc_dentry():

实现将新的目录项（dentry）插入到指定的索引节点（inode）中。采用头插法将新 dentry 添加到 inode 的 dentry 链表头部，并更新 inode 的目录项计数和其他相关属性。当目录项数量达到一定阈值时，分配新的数据块。用于建立新实体与父级的关系

```
108     int nfs_alloc_dentry(struct nfs_inode* inode, struct nfs_dentry* dentry) {
109         dentry->brother = inode->dentrys;
110         inode->dentrys = dentry;
111         inode->dir_cnt++;
112         inode->size += sizeof(struct nfs_dentry);
113 
114         // 判断是否需要重新分配一个数据块
115         if (inode->dir_cnt % NFS_DENTRY_PER_BLK() == 1) {
116             inode->block_pointer[inode->block_allocated] = nfs_alloc_data();
117             inode->block_allocated++;
118         }
119         return inode->dir_cnt;
120     }
```

nfs_alloc_inode():

实现分配一个新的数据块。遍历数据位图寻找空闲的数据块槽位，设置位图标记该数据块已被占用。返回分配的数据块编号。当需要为 inode 分配额外的数据存储空间时（例如，在创建大文件或扩展目录时）使用此函数

```
128     struct nfs_inode* nfs_alloc_inode(struct nfs_dentry *dentry) {
129         struct nfs_inode* inode;
130         int byte_cursor   = 0;
131         int bit_cursor    = 0;
132         int ino_cursor    = 0;
133         boolean is_find_free_entry   = FALSE;
134 
135         inode = (struct nfs_inode*)malloc(sizeof(struct nfs_inode));
136 
137         // 先按字节寻找空闲的inode位图
138         for (; byte_cursor < NFS_BLKS_SZ(nfs_super.map_inode_blks); byte_cursor++) {
139             // 再在该字节中遍历8个bit寻找空闲的inode位图
140             for (; bit_cursor < UINT8_BITS; bit_cursor++) {
141                 if((nfs_super.map_inode[byte_cursor] & (0x1 << bit_cursor)) == 0) {
142                     // 当前ino_cursor位置空闲
143                     nfs_super.map_inode[byte_cursor] |= (0x1 << bit_cursor);
144                     is_find_free_entry = TRUE;
145                     break;
146                 }
147                 ino_cursor++;
148             }
149             if (is_find_free_entry) {
150                 break;
151             }
152         }
153 
154         // 未找到空闲结点
155         if (!is_find_free_entry || ino_cursor >= nfs_super.max_ino) {
156             free(inode);
157             return NULL;
158         }
159 
160         // 为目录项分配inode节点并初始化相关属性
161         inode->ino  = ino_cursor;
162         inode->size = 0;
163         inode->dir_cnt = 0;
164         inode->block_allocated = 0;
165         inode->dentrys = NULL;
166 
167         // dentry指向分配的inode
168         dentry->inode = inode;
169         dentry->ino   = inode->ino;
170         // inode指回dentry
171         inode->dentry = dentry;
172 
173         // 文件类型需要分配空间，目录项已经在dentrys里，普通文件不要求额外分配数据块的操作，一次性分配完就好了
174         if (NFS_IS_REG(inode)) {
175             for (int i = 0; i < NFS_DATA_PER_FILE; i++) {
176                 inode->data[i] = (uint8_t *)malloc(NFS_BLK_SZ());
177             }
178         }
179 
180         return inode;
181     }
```

nfs_sync_inode():

实现将内存中的 inode 及其关联结构刷回到磁盘上。将 inode 的内容复制到一个临时结构体中，并将其写入磁盘。如果 inode 是目录，则递归地将所有子 dentry 及其对应的 inode 同步到磁盘。对于普通文件，直接将预分配的数据块内容写入磁盘

```c
int nfs_sync_inode(struct nfs_inode *inode) {
    struct nfs_inode_d  inode_d;
    struct nfs_dentry*  dentry_cursor;
    struct nfs_dentry_d dentry_d;
    int offset;
    int ino             = inode->ino;

    // 把inode的内容拷贝到inode_d中
    inode_d.ino            = ino;
    inode_d.size           = inode->size;
    inode_d.ftype          = inode->dentry->ftype;
    inode_d.dir_cnt        = inode->dir_cnt;
    inode_d.block_allocated = inode->block_allocated;
    for (int i = 0; i < NFS_DATA_PER_FILE; i++) {
        inode_d.block_pointer[i] = inode->block_pointer[i];
    }

    // 将inode_d刷回磁盘
    if (nfs_driver_write(NFS_INO_OFS(ino), (uint8_t *)&inode_d,
                         sizeof(struct nfs_inode_d)) != NFS_ERROR_NONE) {
        NFS_DBG("[%s] io error\n", __func__);
        return -NFS_ERROR_IO;
    }

    // 刷回inode的数据块
    if (NFS_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 data_blks_num = 0;
        // 要将dentry的内容刷回磁盘，目录有block_allocated个数据块
        while ((dentry_cursor != NULL) && (data_blks_num < inode->block_allocated)) {
            offset = NFS_DATA_OFS(inode->block_pointer[data_blks_num]);
            while ((dentry_cursor != NULL) && (offset + sizeof(struct nfs_dentry_d) < NFS_DATA_OFS(inode->block_pointer[data_blks_num] + 1))) {
                // dentry的内容复制到dentry_d中
                memcpy(dentry_d.fname, dentry_cursor->fname, NFS_MAX_FILE_NAME);
                dentry_d.ftype = dentry_cursor->ftype;
                dentry_d.ino = dentry_cursor->ino;
                // dentry_d的内容刷回磁盘
                if (nfs_driver_write(offset, (uint8_t *)&dentry_d, sizeof(struct nfs_dentry_d))!= NFS_ERROR_NONE) {
                    NFS_DBG("[%s] io error\n", __func__);
                    return -NFS_ERROR_IO;
                }

                // 递归处理目录项的内容
                if (dentry_cursor->inode != NULL) {
                    nfs_sync_inode(dentry_cursor->inode);
                }

                dentry_cursor  = dentry_cursor->brother;
                offset += sizeof(struct nfs_dentry_d);
            }
            data_blks_num++;
        }
    }
    // 如果当前inode是文件，那么数据是文件内容，直接写即可
    else if (NFS_IS_REG(inode)) {
        // 这里也要保证i < inode->block_allocated
        // 必做并不需要实现普通文件的write操作，虽然我们前面在alloc_node中为普通文件申请了数据块内存
        // 但我们并没有为其申请对应的数据位图，所以这里不需要写回去
        for (int i = 0; i < inode->block_allocated; i++) {
            if (nfs_driver_write(NFS_DATA_OFS(inode->block_pointer[i]), inode->data[i], NFS_BLK_SZ()) != NFS_ERROR_NONE) {
                NFS_DBG("[%s] io error\n", __func__);
                return -NFS_ERROR_IO;
            }
        }
    }

    return NFS_ERROR_NONE;
}
```

nfs_read_inode():

实现从磁盘读取指定 inode 的内容到内存中。

• 分配一个新的 nfs_inode 结构体用于存储读取的数据。

• 使用 nfs_driver_read 函数从磁盘读取 inode 数据（nfs_inode_d），如果读取失败则释放分配的内存并返回 NULL。

• 将读取到的数据复制到新的 nfs_inode 结构体中，并初始化其属性。

```
299  struct nfs_inode* nfs_read_inode(struct nfs_dentry *dentry, int ino) {
300      struct nfs_inode* inode = (struct nfs_inode*)malloc(sizeof(struct nfs_inode));
301      struct nfs_inode_d inode_d;
302      struct nfs_dentry* sub_dentry;
303      struct nfs_dentry_d dentry_d;
304      int dir_cnt = 0;
305
306      // 从磁盘读取索引节点
307      if (nfs_driver_read(NFS_INO_OFS(ino), (uint8_t *)&inode_d, sizeof(struct nfs_inode_d)) != NFS_ERROR_NONE) {
308          NFS_DBG("[%s] io error\n", __func__);
309          free(inode);
310          return NULL;
311      }
312
313      // 根据inode_d的内容初始化inode
314      inode->dir_cnt = inode_d.dir_cnt;
315      inode->ino = inode_d.ino;
316      inode->size = inode_d.size;
317      inode->dentry = dentry;
318      inode->dentrys = NULL;
319      inode->block_allocated = inode_d.block_allocated;
320      for (int i = 0; i < NFS_DATA_PER_FILE; i++) {
321          inode->block_pointer[i] = inode_d.block_pointer[i];
322      }
323
```

•如果该 inode 是一个目录，则继续读取其包含的所有子 dentry，并为每个子 dentry 创建新的 nfs_dentry 对象，使用 nfs_alloc_dentry 将其链接到父级 inode。

• 如果该 inode 是一个普通文件，则直接读取其所有数据块内容到内存。

返回指向新分配的 nfs_inode 的指针。

```
324      // 内存中的inode的数据或子目录项部分也需要读出
325      if (NFS_IS_DIR(inode)) {
326          dir_cnt = inode_d.dir_cnt;
327          int data_blks_num = 0;
328          int offset;
329
330          // 对所有的目录项都进行处理（先按数据块处理）
331          while (dir_cnt > 0 && data_blks_num < NFS_DATA_PER_FILE) {
332              offset = NFS_DATA_OFS(inode->block_pointer[data_blks_num]);
333
334              // 再按单独的目录项进行处理
335              while (dir_cnt > 0 && offset + sizeof(struct nfs_dentry_d) < NFS_DATA_OFS(inode->block_pointer[data_blks
336                  if (nfs_driver_read(offset, (uint8_t *)&dentry_d, sizeof(struct nfs_dentry_d)) != NFS_ERROR_NONE) {
337                      NFS_DBG("[%s] io error\n", __func__);
338                      free(inode);
339                      return NULL;
340                  }
341
342                  // 用从磁盘中读出的dentry_d更新内存中的sub_dentry
343                  sub_dentry = new_dentry(dentry_d.fname, dentry_d.ftype);
344                  sub_dentry->parent = inode->dentry;
345                  sub_dentry->ino = dentry_d.ino;
346                  nfs_alloc_dentry(inode, sub_dentry);
347
348                  offset += sizeof(struct nfs_dentry_d);
349                  dir_cnt--;
350              }
351              data_blks_num++;
352          }
353      }
354      // inode是普通文件则直接读取数据块
355      else if (NFS_IS_REG(inode)) {
356          for (int i = 0; i < NFS_DATA_PER_FILE; i++) {
357              inode->data[i] = (uint8_t *)malloc(NFS_BLK_SZ());
358              if (nfs_driver_read(NFS_DATA_OFS(inode->block_pointer[i]), (uint8_t *)inode->data[i], NFS_BLK_SZ()) != N
359                  NFS_DBG("[%s] io error\n", __func__);
360                  free(inode);
361                  return NULL;
362              }
363          }
364      }
```

nfs_get_dentry():

实现获取指定位置的目录项。通过遍历 inode->dentrys 链表，直到找到索引匹配的 dentry。如果找到了对应索引的 dentry，则返回其指针；否则返回 NULL

```
376     struct nfs_dentry* nfs_get_dentry(struct nfs_inode *inode, int dir) {
377         struct nfs_dentry* dentry_cursor = inode->dentrys;
378         int cnt = 0;
379 
380         // 获取某个inode的第dir个目录项
381         while (dentry_cursor) {
382             if (dir == cnt) {
383                 return dentry_cursor;
384             }
385             cnt++;
386             dentry_cursor = dentry_cursor->brother;
387         }
388         return NULL;
389     }
```

nfs_lookup():

实现根据给定路径查找对应的文件或目录。

- 计算路径的层级数，并准备辅助变量。
- 对于根目录的情况，直接设置标志位并返回根目录的 dentry。
- 使用 strtok 分割路径字符串，逐层查找对应的 dentry。

```
419     struct nfs_dentry* nfs_lookup(const char *path, boolean* is_find, boolean* is_root) {
420         struct nfs_dentry* dentry_cursor = nfs_super.root_dentry;
421         struct nfs_dentry* dentry_ret = NULL;
422         struct nfs_inode* inode;
423         int total_lvl = nfs_calc_lvl(path);
424         int lvl = 0;
425         boolean is_hit;
426         char* fname = NULL;
427         char* path_cpy = (char*)malloc(strlen(path) + 1);
428         *is_find = FALSE;
429         *is_root = FALSE;
430         strcpy(path_cpy, path);
431 
432         // 根目录
433         if (total_lvl == 0) {
434             *is_find = TRUE;
435             *is_root = TRUE;
436             dentry_ret = nfs_super.root_dentry;
437         }
438 
439         // strtok用来分割路径以此获得最外层文件名
440         fname = strtok(path_cpy, "/");
```

- 每次找到一个 dentry 后，检查是否需要进一步读取其 inode 内容（缓存机制）。
- 如果当前 inode 是普通文件且未到达路径末尾，则停止查找并返回上一级有效的 dentry。
- 如果当前 inode 是目录，则在其子 dentry 列表中查找下一个路径组件。
- 如果在某一层找不到对应的 dentry，则设置标志位并返回最后找到的有效 dentry。
- 确保最终返回的 dentry 对应的 inode 不为空。
- 释放临时分配的内存（如路径副本）。

```
441         while (fname) {
442             lvl++;
443 
444             // Cache机制，如果当前dentry的inode为空则从磁盘读出来
445             if (dentry_cursor->inode == NULL) {
446                 dentry_cursor->inode = nfs_read_inode(dentry_cursor, dentry_cursor->ino);
447             }
448 
449             inode = dentry_cursor->inode;
450 
451             // 还没到对应层数就查到普通文件，无法继续往下查询
452             if (NFS_IS_REG(inode) && lvl < total_lvl) {
453                 NFS_DBG("[%s] not a dir\n", __func__);
454                 dentry_ret = inode->dentry;
455                 break;
456             }
457 
458             // 当前inode对应的是一个目录
459             if (NFS_IS_DIR(inode)) {
460                 dentry_cursor = inode->dentrys;
461                 is_hit = FALSE;
462 
463                 // 遍历子目录项找到对应文件名称的dentry
464                 while (dentry_cursor) {
465                     if (memcmp(dentry_cursor->fname, fname, strlen(fname)) == 0) {
466                         is_hit = TRUE;
467                         break;
468                     }
469                     dentry_cursor = dentry_cursor->brother;
470                 }
471 
472                 // 没有找到对应文件夹名称的目录项则返回最后找到的文件夹的dentry
473                 if (!is_hit) {
474                     *is_find = FALSE;
475                     NFS_DBG("[%s] not found %s\n", __func__, fname);
476                     dentry_ret = inode->dentry;
477                     break;
478                 }
479 
480                 if (is_hit && lvl == total_lvl) {
481                     *is_find = TRUE;
482                     dentry_ret = dentry_cursor;
483                     break;
484                 }
485             }
486             // 继续使用strtok获取下一层文件的名称
487             fname = strtok(NULL, "/");
488         }
489 
490         // 确保dentry_ret对应的inode不为空
491         if (dentry_ret && dentry_ret->inode == NULL) {
492             dentry_ret->inode = nfs_read_inode(dentry_ret, dentry_ret->ino);
493         }
494 
495         free(path_cpy);
496         return dentry_ret;
```

nfs_mount():

实现挂载自定义文件系统 newfs，并初始化其内部结构。

• 打开设备：通过 ddriver_open 打开指定的设备文件，并检查是否成功。

```
507     int nfs_mount(struct custom_options options) {
508         int ret = NFS_ERROR_NONE;
509         int driver_fd;
510         struct nfs_super_d nfs_super_d;
511         struct nfs_dentry* root_dentry;
512         struct nfs_inode* root_inode;
513 
514         int inode_num;
515         int map_inode_blks;
516         int data_num;
517         int map_data_blks;
518 
519         int super_blks;
520         boolean is_init = FALSE;
521 
522         nfs_super.is_mounted = FALSE;
523         driver_fd = ddriver_open(options.device);
524 
525         if (driver_fd < 0) {
526             return driver_fd;
527         }
```

• 设置超级块信息：初始化 nfs_super 结构体中的字段，包括磁盘大小、I/O 大小等，并计算逻辑块大小（两个 I/O 大小）。

• 创建根目录 dentry：为根目录（"/"）创建一个新的 nfs_dentry 对象。

• 读取超级块：从磁盘读取超级块数据（nfs_super_d），如果读取失败则返回错误码。

```
529         // 向超级块中写入相关信息
530         nfs_super.fd = driver_fd;
531         ddriver_ioctl(NFS_DRIVER(), IOC_REQ_DEVICE_SIZE, &nfs_super.sz_disk);
532         ddriver_ioctl(NFS_DRIVER(), IOC_REQ_DEVICE_IO_SZ, &nfs_super.sz_io);
533         nfs_super.sz_blks = 2 * nfs_super.sz_io;  // 两个IO大小
534 
535         // 新建根目录
536         root_dentry = new_dentry("/", NFS_DIR);
537 
538         // 读取磁盘超级块内容
539         if (nfs_driver_read(NFS_SUPER_OFS, (uint8_t *)(&nfs_super_d), sizeof(struct nfs_super_d)) != NFS_ERROR
540             return -NFS_ERROR_IO;
541         }
```

• 首次挂载检测：根据超级块中的幻数（magic）判断是否是第一次挂载。如果是首次挂载，则计算各部分的大小（如 inode 数量、数据块数量等），并初始化超级块布局。设置新的幻数值以标记文件系统已初始化。

```
543         // 根据磁盘超级块的幻数判断是否是第一次挂载
544         if (nfs_super_d.magic != NFS_MAGIC_NUM) {
545             // 计算各部分的大小
546             // 宏定义在type.h中
547             super_blks      = NFS_SUPER_BLKS;
548             map_inode_blks  = NFS_INODE_MAP_BLKS;
549             map_data_blks   = NFS_DATA_MAP_BLKS;
550             inode_num       = NFS_INODE_BLKS;
551             data_num        = NFS_DATA_BLKS;
552 
553             // 布局layout
554             nfs_super.max_ino               = inode_num;
555             nfs_super.max_dno               = data_num;
556             nfs_super_d.map_inode_blks      = map_inode_blks;
557             nfs_super_d.map_data_blks       = map_data_blks;
558             nfs_super_d.map_inode_offset    = NFS_SUPER_OFS + NFS_BLKS_SZ(super_blks);
559             nfs_super_d.map_data_offset     = nfs_super_d.map_inode_offset + NFS_BLKS_SZ(map_inode_blks);
560             nfs_super_d.inode_offset        = nfs_super_d.map_data_offset + NFS_BLKS_SZ(map_data_blks);
561             nfs_super_d.data_offset         = nfs_super_d.inode_offset + NFS_BLKS_SZ(inode_num);
562 
563             nfs_super_d.sz_usage            = 0;
564             nfs_super_d.magic               = NFS_MAGIC_NUM;
565 
566             is_init = TRUE;
567         }
```

• 建立内存结构：初始化超级块（nfs_super）的属性，并分配内存用于存储索引位图和数据位图。

• 读取位图：从磁盘读取索引位图和数据位图到内存中，确保文件系统的状态与磁盘一致。

```
569         // 建立 in-memory 结构
570         // 初始化超级块
571         nfs_super.sz_usage   = nfs_super_d.sz_usage;
572 
573         // 建立索引位图
574         nfs_super.map_inode         = (uint8_t *)malloc(NFS_BLKS_SZ(nfs_super_d.map_inode_blks));
575         nfs_super.map_inode_blks    = nfs_super_d.map_inode_blks;
576         nfs_super.map_inode_offset  = nfs_super_d.map_inode_offset;
577         nfs_super.inode_offset      = nfs_super_d.inode_offset;
578 
579         // 建立数据位图
580         nfs_super.map_data          = (uint8_t *)malloc(NFS_BLKS_SZ(nfs_super_d.map_data_blks));
581         nfs_super.map_data_blks     = nfs_super_d.map_data_blks;
582         nfs_super.map_data_offset   = nfs_super_d.map_data_offset;
583         nfs_super.data_offset       = nfs_super_d.data_offset;
584 
585         // 从磁盘中读取索引位图
586         if (nfs_driver_read(nfs_super_d.map_inode_offset, (uint8_t *)(nfs_super.map_inode),
587                             NFS_BLKS_SZ(nfs_super_d.map_inode_blks)) != NFS_ERROR_NONE) {
588             free(nfs_super.map_inode);
589             free(nfs_super.map_data);
590             return -NFS_ERROR_IO;
591         }
592 
593         // 从磁盘中读取数据位图
594         if (nfs_driver_read(nfs_super_d.map_data_offset, (uint8_t *)(nfs_super.map_data),
595                             NFS_BLKS_SZ(nfs_super_d.map_data_blks)) != NFS_ERROR_NONE) {
596             free(nfs_super.map_inode);
597             free(nfs_super.map_data);
598             return -NFS_ERROR_IO;
599         }
```

• 分配根节点：如果是首次挂载，则为根目录分配一个 inode，并同步到磁盘；否则，直接读取根节点的 inode。

• 更新根目录 dentry：将读取或分配的根节点 inode 链接到根目录 dentry，并将其设

置为超级块的一部分。

• 标记挂载状态：设置 is_mounted 标志为 TRUE，表示文件系统已成功挂载

```
601         // 分配根节点
602         if (is_init) {
603             root_inode = nfs_alloc_inode(root_dentry);
604             if (!root_inode) {
605                 free(nfs_super.map_inode);
606                 free(nfs_super.map_data);
607                 return -NFS_ERROR_NOSPACE;
608             }
609             nfs_sync_inode(root_inode);
610         }
611 
612         root_inode            = nfs_read_inode(root_dentry, NFS_ROOT_INO);
613         if (!root_inode) {
614             free(nfs_super.map_inode);
615             free(nfs_super.map_data);
616             return -NFS_ERROR_IO;
617         }
618         root_dentry->inode    = root_inode;
619         nfs_super.root_dentry = root_dentry;
620         nfs_super.is_mounted  = TRUE;
621 
622         return ret;
623     }
```

nfs_umount():

实现卸载自定义文件系统 newfs，并清理相关资源。

• 检查挂载状态：如果文件系统未挂载，则直接返回成功。

• 同步所有更改：调用 nfs_sync_inode 将根节点及其子树的所有更改刷回磁盘，确保所有数据都被持久化。

• 更新超级块：将内存中的超级块信息（如位图偏移量、大小等）复制到临时结构体 nfs_super_d 中，并写入磁盘。

• 刷回位图：将内存中的索引位图和数据位图写回到磁盘相应位置，保证位图的一致性。

• 释放内存资源：释放为位图分配的内存。

• 关闭驱动：关闭底层磁盘驱动接口，确保所有资源被正确释放。

• 返回结果：如果所有操作都成功，则返回成功码；否则返回相应的错误码。

```
630     int nfs_umount() {
631         struct nfs_super_d nfs_super_d;
632 
633         // 没有挂载直接报错
634         if (!nfs_super.is_mounted) {
635             return NFS_ERROR_NONE;
636         }
637 
638         // 从根节点向下刷写节点
639         if (nfs_sync_inode(nfs_super.root_dentry->inode) != NFS_ERROR_NONE) {
640             return -NFS_ERROR_IO;
641         }
642 
643         // 将内存中的超级块刷回磁盘
644         nfs_super_d.magic               = NFS_MAGIC_NUM;
645         nfs_super_d.sz_usage            = nfs_super.sz_usage;
646 
647         nfs_super_d.map_inode_blks      = nfs_super.map_inode_blks;
648         nfs_super_d.map_inode_offset    = nfs_super.map_inode_offset;
649         nfs_super_d.inode_offset        = nfs_super.inode_offset;
650 
651         nfs_super_d.map_data_blks       = nfs_super.map_data_blks;
652         nfs_super_d.map_data_offset     = nfs_super.map_data_offset;
653         nfs_super_d.data_offset         = nfs_super.data_offset;
654 
655         if (nfs_driver_write(NFS_SUPER_OFS, (uint8_t *)&nfs_super_d,
656                              sizeof(struct nfs_super_d)) != NFS_ERROR_NONE) {
657             return -NFS_ERROR_IO;
658         }
659 
660         // 将索引位图刷回磁盘
661         if (nfs_driver_write(nfs_super_d.map_inode_offset, (uint8_t *)(nfs_super.map_inode),
662                              NFS_BLKS_SZ(nfs_super_d.map_inode_blks)) != NFS_ERROR_NONE) {
663             return -NFS_ERROR_IO;
664         }
665 
666         // 将数据位图刷回磁盘
667         if (nfs_driver_write(nfs_super_d.map_data_offset, (uint8_t *)(nfs_super.map_data),
668                              NFS_BLKS_SZ(nfs_super_d.map_data_blks)) != NFS_ERROR_NONE) {
669             return -NFS_ERROR_IO;
670         }
671 
672         // 释放内存中的位图
673         free(nfs_super.map_inode);
674         free(nfs_super.map_data);
675 
676         // 关闭驱动
677         ddriver_close(NFS_DRIVER());
678 
679         return NFS_ERROR_NONE;
680     }
```

2.3 整体流程

整个 newfs 程序的运行流程可以概述如下：

程序启动与挂载

- 初始化：程序启动时调用 nfs_mount，打开并初始化磁盘驱动，读取或初始化超级块。
- 加载数据：将根节点和位图加载到内存中，设置文件系统为已挂载状态。

文件系统操作

- 路径解析：用户请求访问文件或目录时，通过 nfs_lookup 解析路径，查找对应的

dentry 和 inode。

• 文件操作：创建、修改或删除文件/目录时，分配或更新相应的 inode 和 dentry，并同步更改到磁盘。读写文件内容时，使用底层 I/O 函数进行数据块的操作。

程序结束与卸载

• 同步更改：在卸载前，确保所有更改被持久化到磁盘。

• 清理资源：释放内存中的位图和其他资源，关闭磁盘驱动。

• 完成卸载：调用 nfs_umount，标记文件系统为未挂载状态。

这里简单贴一下 test 脚本的结果：

```
220110720@comp2:~/Fuse_HITSZ/fs/demo$ cd tests/
220110720@comp2:~/Fuse_HITSZ/fs/demo/tests$ chmod +x test.sh && ./test.sh
Test Pass!!!


4194304 bytes (4.2 MB, 4.0 MiB) copied, 0.0700027 s, 59.9 MB/s
================================================================================
8192+0 records in
8192+0 records out
4194304 bytes (4.2 MB, 4.0 MiB) copied, 0.0680943 s, 61.6 MB/s
pass: case 5.1 - umount /home/students/220110720/Fuse_HITSZ/fs/newfs/tests/mnt
pass: case 5.2 - remount /home/students/220110720/Fuse_HITSZ/fs/newfs/tests/mnt
8192+0 records in
8192+0 records out
4194304 bytes (4.2 MB, 4.0 MiB) copied, 0.0731509 s, 57.3 MB/s
pass: case 5.3 - check bitmap
8192+0 records in
8192+0 records out
4194304 bytes (4.2 MB, 4.0 MiB) copied, 0.0494514 s, 84.8 MB/s
================================================================================

Score: 30/30
pass: 恭喜你，通过所有测试 (30/30)
```

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。

为了不浪费空间并提高存储效率，我们设计了一个精细的空间分配方案。具体来说：

实验规定定了逻辑块的大小为 1024B。这意味着每个逻辑块可以存储 1024 字节的数据。由于磁盘总容量为 4MB，逻辑块数为： 逻辑块数 = 4MB/1024B = 4096

为了估算磁盘可以存放的文件数量，我们进行了如下计算：

每个文件最多直接索引 4 个逻辑块，即每个文件的数据上限为： 4 × 1024B = 4KB

假设一个逻辑块可以存储 16 个文件的索引节点，每个索引节点占用 64B，因此维护一个文件所需的存储容量为： 4KB + 64B = 4096B + 64B = 4160B

因此，4MB 磁盘最多可以存放的文件数为： 4MB / 4160B = 4×1024KB / 4160B ≈ 1008

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。*

在实现文件路径解析时，需要能够正确处理给定的路径字符串，并支持多层级目录的创建和访问。这对路径字符串的逐层解析提出了较高的要求，特别是在处理嵌套结构时，确保路径解析的准确性至关重要。虽然实验明确不要求处理相对路径中的.和..等特殊情况，但专注于实现基本功能需求也带来了不少挑战

面对这些挑战，我首先学习了现有文件系统中路径解析的设计思路。基于这些研究，我设计并实现了一套适合本实验需求的路径解析逻辑。这套逻辑可以逐层解析路径字符串，通过分割路径字符串，逐级查找对应的目录项（dentry），从而确定目标文件或目录的位置

在这一过程中，我不仅解决了基本的路径解析问题，还深入理解了路径解析的核心原理，这对我在实验中其他涉及路径解析的地方提供了宝贵的指导

三、实验收获和建议

实验中的收获、感受、问题、建议等。

通过本次基于 FUSE 的青春版 EXT2 文件系统实验，我不仅加深了对文件系统内部结构的理解，还提升了实际编程能力和解决问题的技巧。具体来说：

实验帮助我将课堂上学到的文件系统概念应用于实际项目中，例如超级块、索引节点（inode）、数据块位图等概念不再只是书本上的知识点，而是变成了我可以亲手实现的功能模块。

另外，由于缺乏有效的调试工具，整个实验过程中我不得不依赖日志输出和错误信息来定位问题。这一过程虽然充满挑战，但也极大地锻炼了我的问题分析和解决能力。

值得一提的是，实验过程中还学到了许多编程细节，比如 C 语言中 extern 和 static 关键字的具体用法，这些都为我未来的开发工作打下了坚实的基础。

对于本实验课程，我还是如下建议：

1. 希望能在正式开始实验前，提供更多关于 FUSE 和 EXT2 文件系统的预习资料或讲座，帮助学生更好地理解相关背景知识和技术要点
2. 建议组织定期的公开线上讨论会，鼓励同学们分享各自的思考过程和解决方案。这不仅有助于避免个体在解决问题时陷入困境，还能促进知识共享和团队合作精神，提升整体学习效果

总之，这次实验是一次宝贵的学习经历，它不仅提高了我的技术水平，也让我更加热爱计算机科学这个领域。祝愿本课程越办越好，未来的学生们也能从中受益匪浅！

四、参考资料

实验过程中查找的信息和资料

主要是指导书和 simplefs 的实现