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：5 班

学生学号：200110513

学生姓名：宗晴

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

本实验基于 FUSE 设计并实现了一个可以真正在 Linux 上跑的青春版 EXT2 文件系统，包括：挂载文件系统、卸载文件系统、创建文件/文件夹（数据块可预分配）和查看文件夹下的文件（读取文件夹内容）等功能。

（1）FUSE（Filesystem in User SpacE）架构

FUSE 架构将文件系统的实现从内核态搬到了用户态，也就是我们可以在用户态实现一个文件系统。因此本次实验我们基于 FUSE 架构，在用户态实现青春版 EXT2 文件系统，从而完美接入 Linux。

FUSE 实现了一个对文件系统访问的回调，它分为内核态的模块和用户态的库两部分。其中用户态的库为用户态程序开发提供接口，我们通过这些接口将请求处理功能注册到 FUSE 中。内核态模块是具体的数据流程的功能实现，它截获文件的访问请求，然后调用用户态注册的函数进行处理。基于此原理，我们就可以在用户态实现我们的文件系统。

（2）DDRIVER 虚拟磁盘驱动

DDRIVER 驱动模拟了对一个容量为 4MB（可以通过 ddriver_ioctl 函数访问得到）的磁盘的操作。它保留了磁盘的访问特性，即**按块访问**，每次读写都需要读写完整的 512B 的磁盘块。这使得更新某个磁盘块中的某个 inode 时，需要先把整个磁盘块读出来，更新相应 inode，再写回去，而不是直接往磁盘里写（除非一个 inode 比一个磁盘块大）。

DDRIVER 驱动提供的头文件 ddriver.h 中定义了一系列虚拟磁盘的接口，包括：打开 ddriver 设备、移动 ddriver 磁盘头、写入数据、读出数据、ddriver IO 控制和关闭 ddriver 设备。

我们需要通过 ddriver（disk driver）驱动来访问 ddriver 设备，从而简化磁盘访问操作，来保证后续可以利用 ddriver 驱动来编写文件系统。因此，需要封装对 ddriver 的访问代码，方便设备读写，具体函数 int nfs_driver_read(int offset, uint8_t *out_content, int size)和 int nfs_driver_write(int offset, uint8_t *in_content, int size)见第 2 节功能详细说明。驱动读写 IO 为 512B，此处与 simplefs 文件系统不同的是，simplefs 的块大小与驱动读写 IO 相等，按一个块大小为 512B 来封装的，而 EXT2 文件系统一个块大小为 1024B，也就是两个 IO 单位。

（3）设计介质数据结构（即磁盘数据结构）和内存数据结构

介质数据结构包括超级块 nfs_super_d、索引节点 nfs_inode_d 和目录项 nfs_dentry_d，内存数据结构同样也包括超级块 nfs_super、索引节点 nfs_inode 和目录项 nfs_dentry。

介质数据结构与内存数据结构有较大的不同，主要在于内存数据结构不仅要存储相

应数据在磁盘中的相关信息，还要存储它们在内存中的相关信息。在代码中，我们用“_d”来区分该数据结构或变量是在磁盘中还是在内存中（带有该标志的表示在磁盘中）。数据结构的具体设计见第 2 节功能详细说明。

（4）设计文件系统布局

在 include/fs.layout 文件中设计填写文件系统布局。

首先，该文件系统包含 5 个部分，包括超级块、索引节点位图、数据块位图、索引节点和数据块。各部分的功能以及大小如下：

1）**超级块 superblock**：包含整个系统的总体信息。占用 **1 个块**大小，即 1024B。

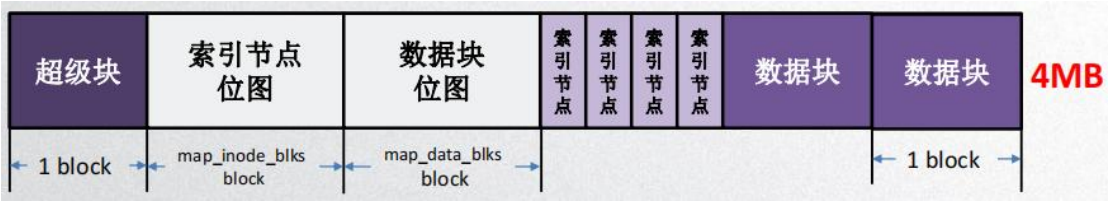
2）**索引节点位图 map_inode**：记录着索引节点表的使用情况，用 1 个比特记录某一个索引节点是否被使用。已知磁盘容量为 4MB，一个块大小为 1024B，因此，最多有 4MB/1KB=4096 个块（事实上更少）。由于 4096bit=512Byte<1024B，所以数据块位图只需要 1 块。又因为一个索引节点对应多个数据块，因此所需的索引节点数量小于数据块数量，所以索引节点位图也只需要 **1 块**，即 1024B。

3）**数据块位图 map_data**：记录着数据块的使用情况，用 1 个比特记录某一个数据块是否被占用。由 2）可知数据块位图只需要 **1 块**，即 1024B。

4）**索引节点 inode**：记录着文件的元数据，每个文件都与一个 inode 对应，但一个 inode 可能对应多个文件（硬链接）。在本实验中可以认为一个 inode 对应一个文件。在此处，我设置索引节点在磁盘中连续存储，仅限制所有索引节点的总体大小与块大小对齐。经计算（计算过程见第 3 节实验特色），我设置了索引节点的最大个数 nfs_max_ino 为 **700 个**，占用块数由代码计算得到。

5）**数据块 datablock**：记录文件原始内容。磁盘中的剩余空间均用于存储数据块。

由上所述，可以得出文件系统的布局如下：



其中 map_inode_blks 和 map_data_blks 均为 1。

由此，可以填写 include/fs.layout 文件中的布局：

```
| BSIZE = 1024 B |
| Super(1) | Inode Map(1) | DATA Map(1) | INODE(28) | DATA(*) |
```

其中存储 inode 的块数由程序根据 inode 的最大个数计算并输出得到。

（5）完成钩子以及相应的工具函数

钩子函数（也可称为函数指针），是一系列函数的抽象，帮助我们接入 FUSE 框架。FUSE 框架通过向 fuse_main 函数传入一个被复制的 fuse_operations 结构体即可完成操作的注册，从而使得文件系统能够按照我们既定的方式来处理命令。需要完成的钩子函数定义在 src/nfs.c 的 operations 数据结构中，主要包括：

钩子	对应的函数	主要功能
.init	void* nfs_init(struct fuse_conn_info *	挂载（mount）文件系统，完成超级块的

	conn_info)	读取、位图的建立、驱动的初始化等操作
.destroy	void nfs_destroy(void* p)	卸载（umount）文件系统，完成超级块回写设备、驱动的关闭、更多必要结构的回写等操作，以保证下一次挂载能够恢复 ddriver 设备中的数据
.mkdir	int nfs_mkdir(const char* path, mode_t mode)	创建目录
.getattr	int nfs_getattr(const char* path, struct stat * nfs_stat)	获取文件或目录的属性
.readdir	int nfs_readdir(const char * path, void * buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info * fi)	遍历目录项，填充至 buf，并交给 FUSE 输出
.mknod	int nfs_mknod(const char* path, mode_t mode, dev_t dev)	创建文件
.utimens	int nfs_utimens(const char* path, const struct timespec tv[2])	修改时间

函数的具体实现流程见第 2 节功能详细说明。

为实现这些钩子还需要完成相应的工具函数：

工具函数	主要功能
char* nfs_get_fname(const char* path)	获取文件名
int nfs_calc_lvl(const char * path)	计算路径的层级
int nfs_driver_read(int offset, uint8_t *out_content, int size)	驱动读
int nfs_driver_write(int offset, uint8_t *in_content, int size)	驱动写
int nfs_alloc_dentry(struct nfs_inode* inode, struct nfs_dentry* dentry)	为一个 inode 分配 dentry，采用头插法
struct nfs_inode* nfs_alloc_inode(struct nfs_dentry * dentry)	分配一个 inode，占用位图
int nfs_sync_inode(struct nfs_inode * inode)	将内存 inode 及其下方结构全部刷回磁盘
struct nfs_inode* nfs_read_inode(struct nfs_dentry * dentry, int ino)	dentry 指向 ino，读取该 inode
struct nfs_dentry* nfs_get_dentry(struct nfs_inode * inode, int dir)	返回 inode 中的第 dir 个 dentry
struct nfs_dentry* nfs_lookup(const char * path, boolean* is_find, boolean* is_root)	找到路径所对应的目录项，或者返回上一级目录项
int nfs_mount(struct custom_options options)	挂载 nfs
int nfs_umount()	卸载 nfs

上述工具函数的具体实现流程见第 2 节功能详细说明。

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）

2.1. 数据结构

1、超级块 nfs_super_d

介质数据结构（磁盘数据结构）如下所示：

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

    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                 inode_offset;       // 第一个索引节点在磁盘上的偏移

    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                 data_offset;        // 第一个数据块在磁盘上的偏移
};
```

内存数据结构如下图所示：

```
struct nfs_super {
    uint32_t            magic;              // 幻数，用于标识文件系统
    int                 fd;                 // 设备描述符
    /* TODO: Define yourself */

    int                 sz_io;              // 设备单次IO大小，512B
    int                 sz_disk;            // 磁盘大小，4MB
    int                 sz_usage;
    int                 sz_blk;             // EXT2文件系统的块大小，1024B

    int                 max_ino;            // 最多支持的文件数，即索引节点最大个数
    uint8_t*            map_inode;          // inode位图，用1bit记录某一个索引节点是否被使用
    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                 inode_offset;       // 第一个索引节点在磁盘上的偏移

    int                 max_data;           // 最多支持的数据块数
    uint8_t*            map_data;           // data位图，用1bit记录某一个数据块是否被使用
    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                 data_offset;        // 第一个数据块在磁盘上的偏移

    boolean             is_mounted;         // 是否挂载

    struct nfs_dentry*  root_dentry;        // 根目录项
};
```

超级块的内存数据结构比介质数据结构复杂很多，这是因为内存中的超级块不仅要记录磁盘中整个系统的总体信息，还要记录磁盘与内存交换数据时的相关参数，以及两个位图在内存中的地址。各变量的含义已基本在上图的注释中给出，在此不再赘述。

对于内存中的超级块数据结构中的变量，若在磁盘中的超级块数据结构中同样存在，那么在挂载时会使用磁盘中的值初始化内存中的，其它的变量则是通过计算得到的，在挂载函数中会进行具体的解释。

2、索引节点 nfs_inode_d

介质数据结构（磁盘数据结构）如下所示：

```
struct nfs_inode_d
{  // 每个inode对应一个文件
    int                 ino;                                // 在inode位图中的下标
    int                 size;                               // 文件已占用空间
    NFS_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
    int                 data_pointer[NFS_DATA_PER_FILE];    // 数据块块号数组（可固定分配）
};
```

内存数据结构如下图所示：

```
struct nfs_inode {  // 每个inode对应一个文件
    uint32_t            ino;                                    // 在inode位图中的下标
    /* TODO: Define yourself */
    int                 size;                                   // 文件已占用的空间
    int                 dir_cnt;                                // 目录项数量
    struct nfs_dentry*  dentry;                                 // 指向该inode的dentry
    struct nfs_dentry*  dentrys;                                // 当inode是文件夹时，存储所有目录项
    uint8_t *           data_addr_pointer[NFS_DATA_PER_FILE];   // 数据块指针数组，指向内存地址
    int                 data_pointer[NFS_DATA_PER_FILE];        // 数据块块号数组 （可固定分配）
};
```

各变量的含义已基本在上图的注释中给出，在此不再赘述。需注意的是，在此系统中，我们规定每个 inode 对应 NFS_DATA_PER_FILE=6 个数据块，因此需设置一个 data_pointer 数组用于存放所有的块号。

内存数据结构比介质数据结构拥有更多的变量，因为在内存中还需要额外的空间用于存储 inode 对应的数据块内容，所以在 inode 数据结构中设置了 data_addr_pointer 数组用于指向这些空间的内存地址。

同时，对于内存中对应文件和目录的 inode 有不同的设置。对应目录的 inode，全部目录项仅以链表的形式保存在 dentrys 变量中，而没有另外存储在 data_addr_pointer 数组指向的地址中，因此对于**目录对应的 inode，事实上不用在内存中给其分配固定的 block**；对应文件的 inode，文件原始内容存储在 data_addr_pointer 数组指向的数据块内存地址中，而 dentrys 变量并未使用。**这样可以减小对于内存的使用量，同时便于内存中的访问**。

此外，在**规定每个文件最多使用 6 个数据块的基础上**，新增设计，**规定每个目录最多使用 2 个数据块**（具体原因见第 3 节实验特色）。

3、目录项 nfs_dentry_d

介质数据结构（磁盘数据结构）如下所示：

```
struct nfs_dentry_d
{
    char                fname[MAX_NAME_LEN];    // 指向的ino文件名
    NFS_FILE_TYPE       ftype;                  // 指向的ino文件类型
    int                 ino;                    // 指向的ino号
};
```

内存数据结构如下图所示：

```c
struct nfs_dentry {
    char                name[MAX_NAME_LEN];
    uint32_t            ino;                // dentry所指向的inode在位图中的下标
    /* TODO: Define yourself */
    struct nfs_dentry*  parent;             // 父亲inode的dentry
    struct nfs_dentry*  brother;            // 兄弟inode的dentry
    struct nfs_inode*   inode;              // 指向的inode
    NFS_FILE_TYPE       ftype;              // dentry所指向的inode对应的文件类型
};
```

同样，各变量的含义已基本在上图的注释中给出，在此不再赘述。此处，对于内存中的 dentry，一个目录 inode 对应的所有 dentry 都存在该 inode 的 dentrys 链表中，这些 dentry 之间通过 brother 指针相连，它们的 parent 指针均指向这个 inode 的 dentry。

2.2. 工具函数

为便于对每个功能点进行说明，我们首先介绍用到的所有工具函数。

1、基础的获取变量数值的函数以及简单计算函数

这部分函数被定义在 include/types.h 中，如下图所示：

```c
// io大小
#define NFS_IO_SZ()                     (nfs_super.sz_io)
// 磁盘大小
#define NFS_DISK_SZ()                   (nfs_super.sz_disk)
// 设备描述符
#define NFS_DRIVER()                    (nfs_super.fd)
// 一个block的大小
#define NFS_BLK_SZ()                    (nfs_super.sz_blk)
// 一个索引节点在磁盘上的大小
#define NFS_INODE_SZ()                  (sizeof(struct nfs_inode_d))
// blks个block的大小
#define NFS_BLKS_SZ(blks)               (blks * NFS_BLK_SZ())
// 以round为单位向下取整
#define NFS_ROUND_DOWN(value, round)    (value % round == 0 ? value : (value / round) * round)
// 以round为单位向上取整
#define NFS_ROUND_UP(value, round)      (value % round == 0 ? value : (value / round + 1) * round)
// 设置文件名
#define NFS_ASSIGN_FNAME(pnfs_dentry, _fname)   memcpy(pnfs_dentry->name, _fname, strlen(_fname))
// 计算第ino个索引节点的偏移，事实上inode连续存储在磁盘中，因此只会在最后一个存储inode的数据块中存在碎片
#define NFS_INO_OFS(ino)                (nfs_super.inode_offset + ino * NFS_INODE_SZ())
// 计算第dno个数据块的偏移
#define NFS_DATA_OFS(dno)               (nfs_super.data_offset + NFS_BLKS_SZ(dno))
// 判断该inode是否对应目录
#define NFS_IS_DIR(pinode)              (pinode->dentry->ftype == N_DIR)
// 判断该inode是否对应文件
#define NFS_IS_FILE(pinode)             (pinode->dentry->ftype == N_FILE)
```

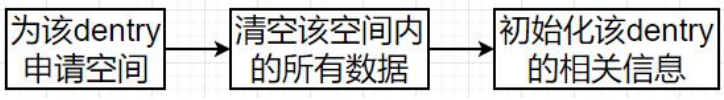
各函数的作用均已在注释中体现，不再赘述。

其中 NFS_ROUND_DOWN 和 NFS_ROUND_UP 比较重要，它们的作用是将输入的数值 value 按照 round 的大小进行对齐，这将会在读写磁盘中起到重要的作用。

NFS_INO_OFS 和 NFS_DATA_OFS 也与 simplefs 文件系统中的相差较多。对于 NFS_INO_OFS，它用于计算第 ino 个索引节点在磁盘上的偏移，因为此处的 inode 在磁盘上为连续存储，因此只需要在基础偏移量上加上 ino 个索引节点的大小即可。而对于 NFS_DATA_OFS，因为每个 inode 对应的数据块是离散存储的，所以该函数所求的是第

dno 个数据块在磁盘上的偏移。

此外，include/types.h 中还定义了 static inline struct nfs_dentry* new_dentry(char * fname, NFS_FILE_TYPE ftype)函数，它相较于 simplefs 文件系统没有进行较大的改变，逻辑也较为简单。它的作用是为名叫 fname 的文件创建一个新的 dentry，返回值为该 dentry。具体流程如下：



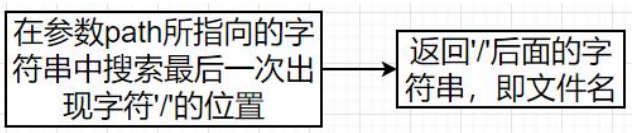
（该报告中的流程图均较为粗糙，仅表示函数的大体逻辑与流程，并不够规范）

2、辅助实现钩子函数的工具函数

（1）char* nfs_get_fname(const char* path)：获取文件名

该函数逻辑较为简单，作用是从给定的路径 path 中获取该文件的文件名，返回值为文件名。

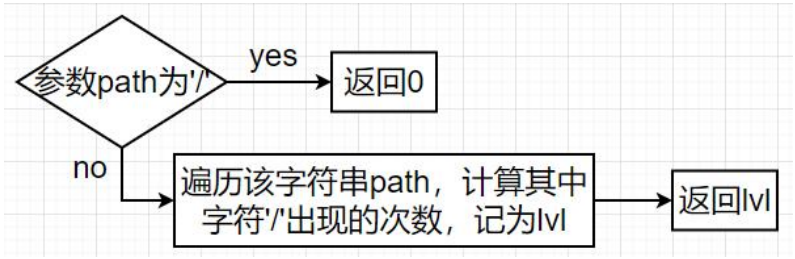
主要流程如下：



（2）int nfs_calc_lvl(const char * path)：计算路径的层级

该函数逻辑较为简单，作用是计算给定路径对应的文件或目录所处的层级，返回值为层数。

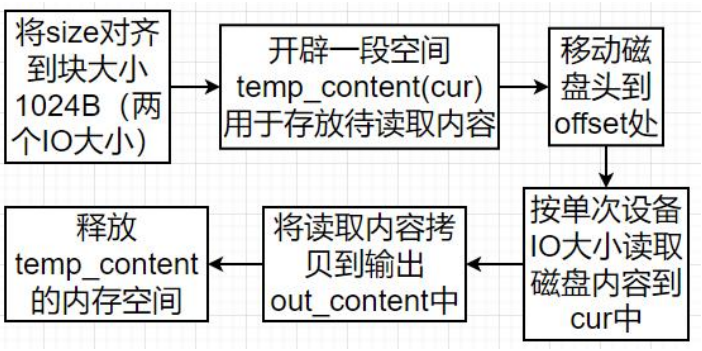
主要流程如下：



（3）int nfs_driver_read(int offset, uint8_t *out_content, int size)：驱动读

该函数的作用是从磁盘中 offset 的位置读取大小为 size 的内容到 out_content 中。由于 inode 采用连续存储，因此与 simplefs 文件系统中不同，此处 offset 不按照 IO 大小对齐，只将 size 对齐到 IO 大小，若成功则返回值为 0。

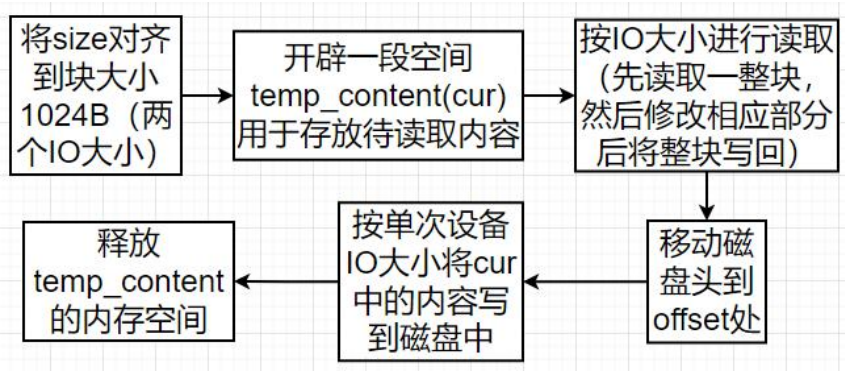
主要流程如下：



（4）int nfs_driver_write(int offset, uint8_t *in_content, int size)：驱动写

该函数的作用是向磁盘中 offset 的位置写大小为 size 的内容 in_content。由于 inode 采用连续存储，因此与 simplefs 文件系统中不同，此处 offset 不按照 IO 大小对齐，只将 size 对齐到 IO 大小，若成功则返回值为 0。

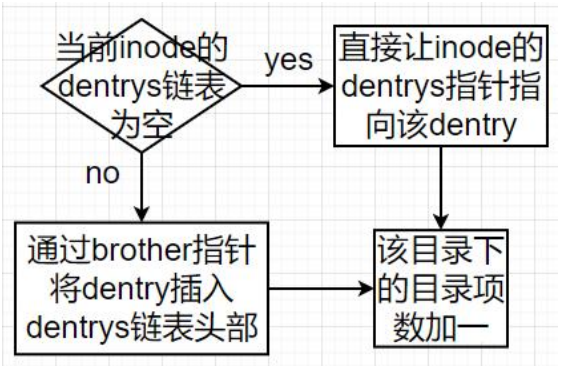
主要流程如下：



（5）int nfs_alloc_dentry(struct nfs_inode* inode, struct nfs_dentry* dentry)：为一个 inode 分配 dentry，采用头插法

该函数逻辑较为简单，作用是采用头插法将 dentry 插入 inode 指向的 dentrys 链表中，返回值为该 inode 对应的目录项数量。

主要流程如下：

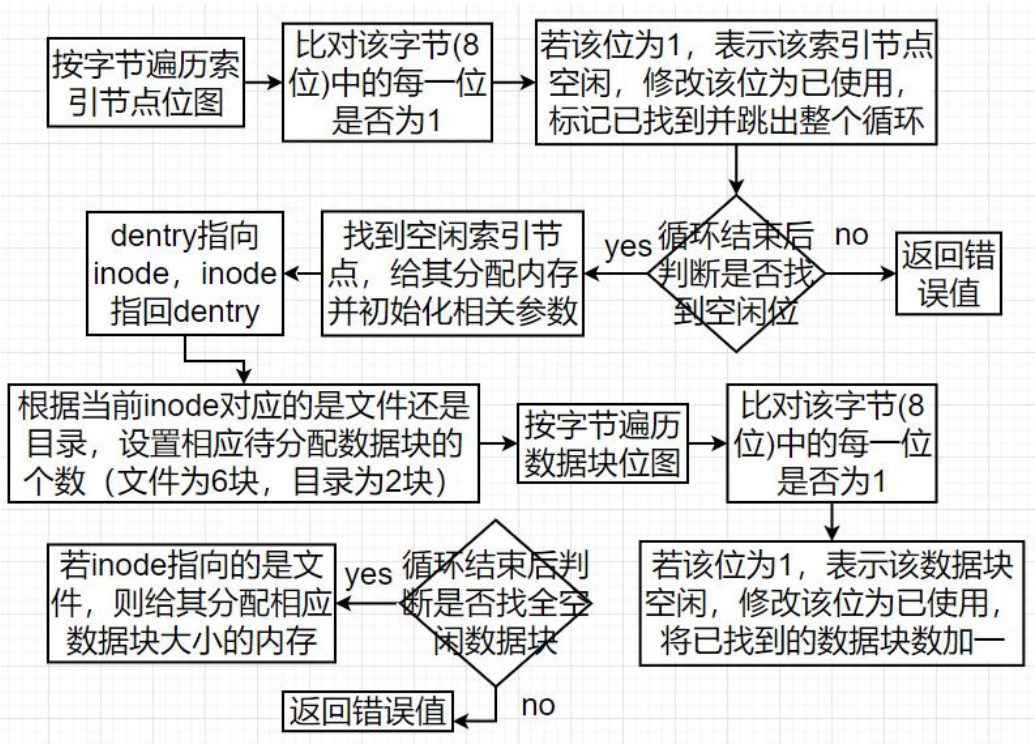


（6）struct nfs_inode* nfs_alloc_inode(struct nfs_dentry * dentry)：分配一个 inode，占用位图

该函数的作用是为作为参数的目录项 dentry 分配一个空闲索引节点，同时需要更新索引节点位图。此外，还需要根据当前 dentry 对应的是文件还是目录分配相应的空

闲数据块，需要更新数据块位图，并且若是文件的话，需要给它分配相应数据块大小的内存空间，若成功则返回值该分配的空闲 inode，否则返回错误号。

主要流程如下：

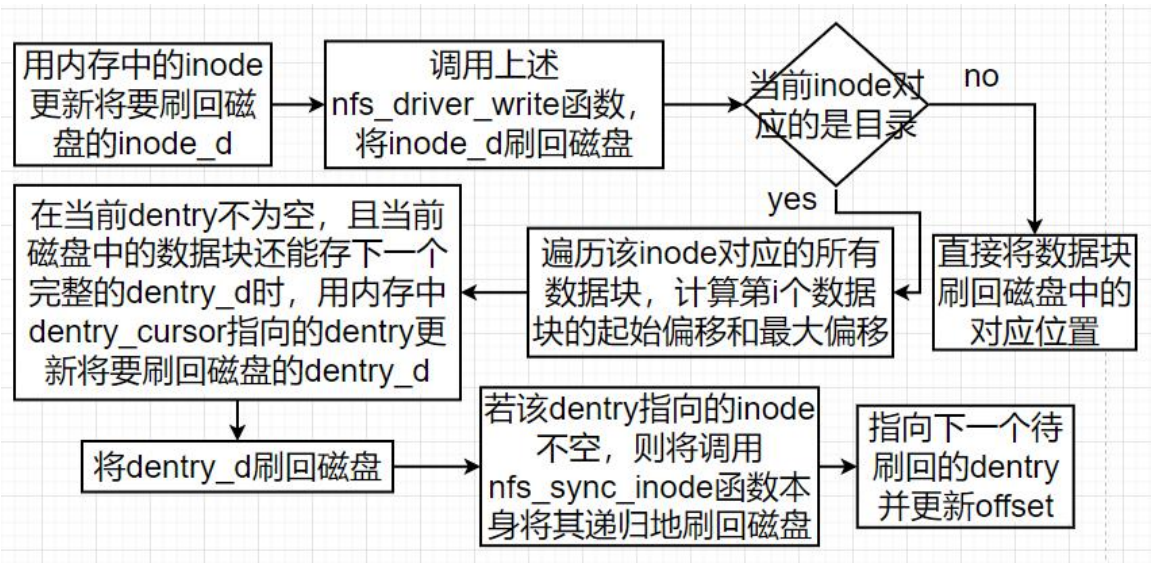


（7）int nfs_sync_inode(struct nfs_inode * inode)：将内存 inode 及其下方结构全部刷回磁盘

该函数的作用是将作为参数的索引节点 inode 以及它下方的所有结构全部写回磁盘的对应位置中，若成功则返回 0，否则返回错误号。

需要特别注意 inode 对应的是**目录**的情况，此时需要将其对应的所有 dentrys 刷回磁盘，由于在磁盘中，dentrys 要存储在 inode_d 对应的数据块中，所以需要根据内存中 inode 中的数据块号数组 data_pointer 计算出磁盘中每个数据块的偏移，从而将每一个 dentry_d 存储到磁盘中的对应位置。由于在 simplefs 文件系统中数据块为一片连续的存储空间，所以 dentrys 可直接存储到数据块中，但此处的 nfs 中，数据块是分散存储的，因此 dentrys 需要按照数据块大小存储，不可出现一个 dentry 存储到两个数据块中的情况，所以需要遍历所有的数据块，按照数据块大小进行存储。

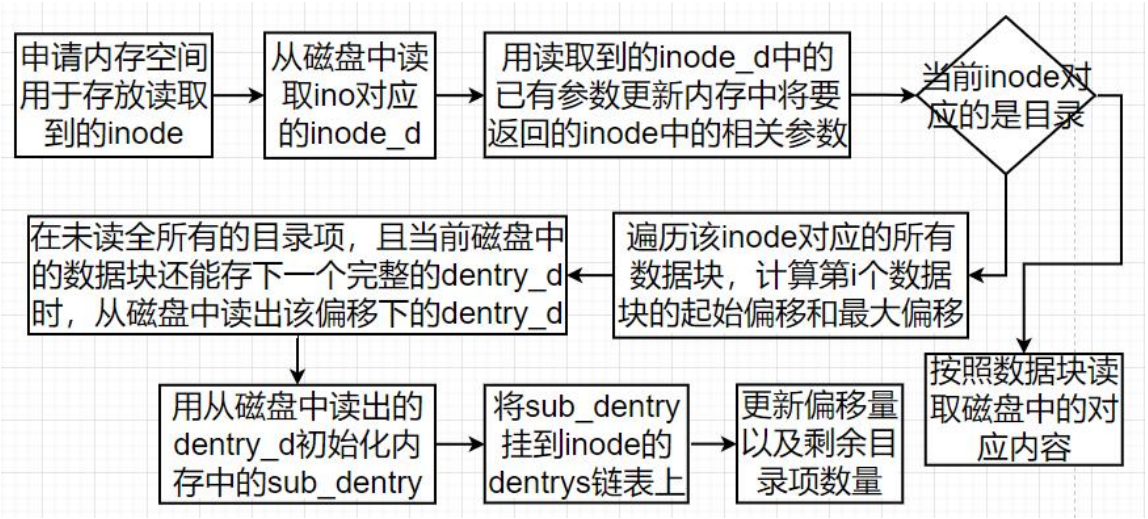
主要流程如下：



（8）struct nfs_inode* nfs_read_inode(struct nfs_dentry * dentry, int ino)：dentry 指向 ino，读取该 inode

该函数的作用是读取 dentry 所指向的 ino 对应的 inode 中的内容。需注意，若 inode 对应的是目录，则需要将其所有目录项保存到该 inode 的 dentrys 链表中，由于目录项按照 block 大小存储在磁盘上，所以需要按照 block 大小读取每一个 block 中的目录项，若成功则返回值该分配的空闲 inode，否则返回错误号。

主要流程如下：



（9）struct nfs_dentry* nfs_get_dentry(struct nfs_inode * inode, int dir)：返回 inode 中的第 dir 个 dentry

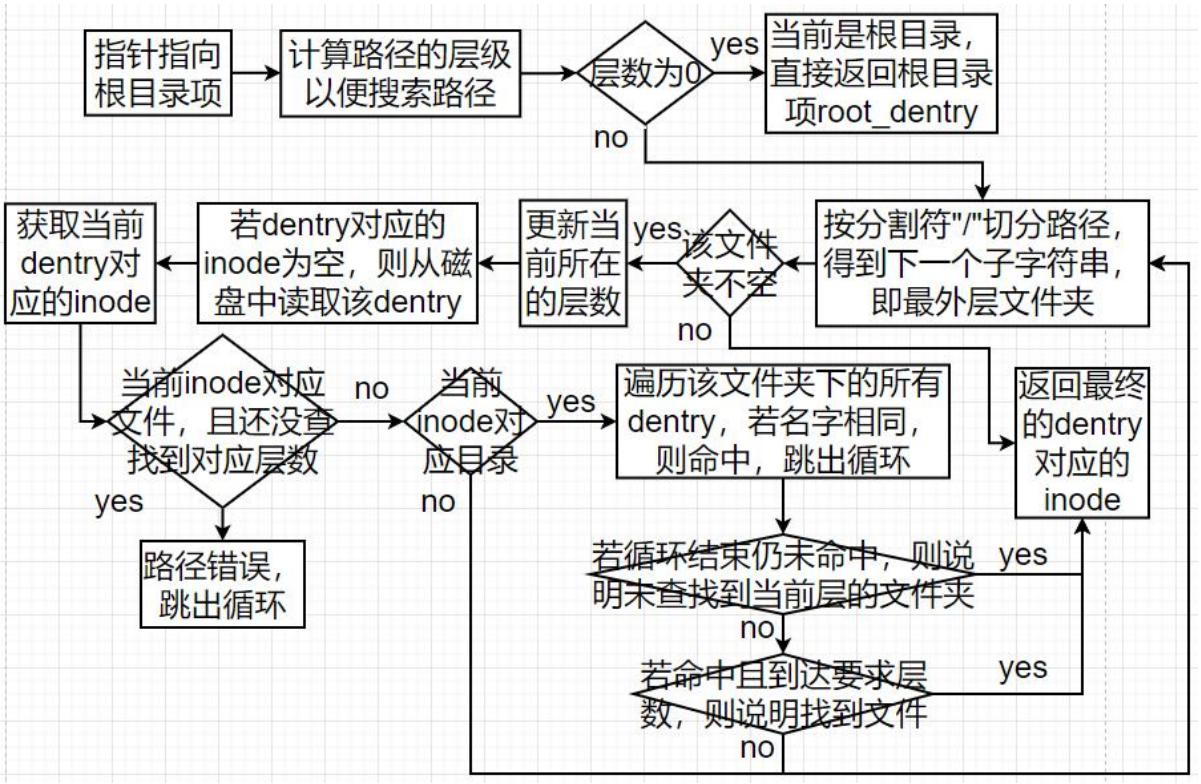
该函数的逻辑较为简单，遍历 inode 的 dentrys 链表，直到计数器的值等于 dir，返回当前指向的 dentry 即可，若失败则返回空指针。

（10）struct nfs_dentry* nfs_lookup(const char * path, boolean* is_find, boolean* is_root)：找到路径所对应的目录项，或者返回上一级目录项

该函数的作用是解析参数 path，查找该路径对应的文件是否存在，若存在则返回

对应文件的目录项，若不存在则返回其最近的外层目录项。

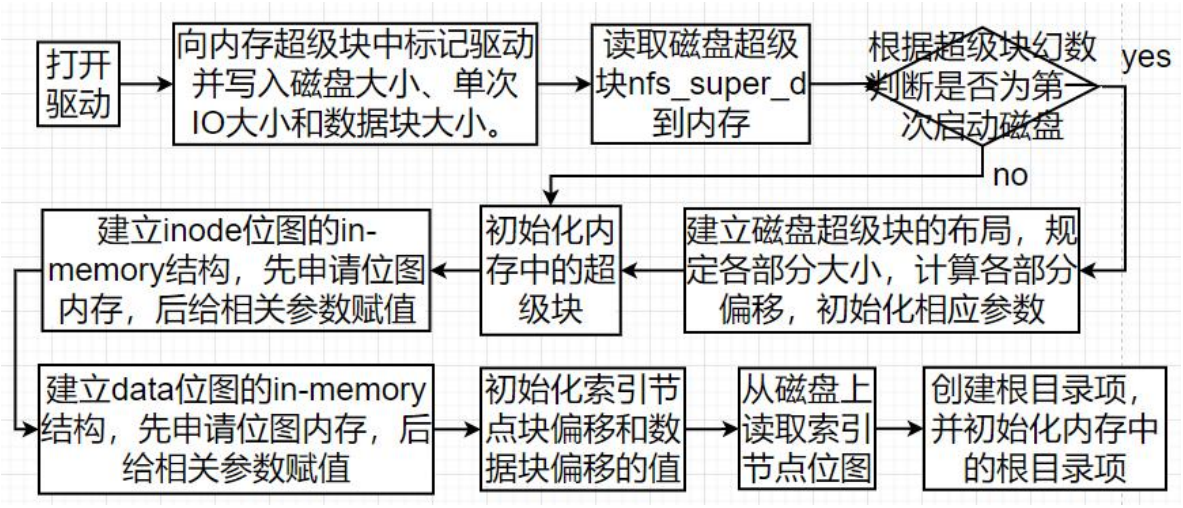
主要流程如下：



（11）int nfs_mount(struct custom_options options)：挂载 nfs

该函数的作用为挂载文件系统，完成超级块的读取、位图的建立、驱动的初始化等操作，若成功则返回 0，否则返回相关错误信息。

具体流程如下：



其中各部分的偏移量的计算公式如下图所示：

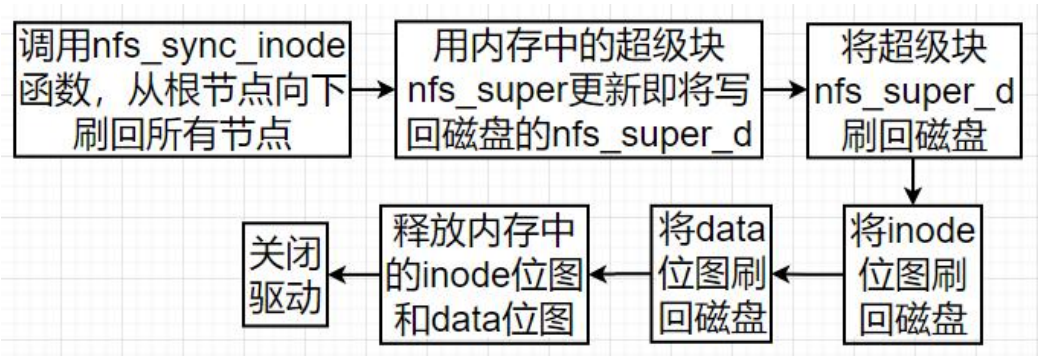
```
// inode位图偏移
nfs_super_d.map_inode_offset = NFS_SUPER_OFS + NFS_BLKS_SZ(super_blks);
// data位图偏移
nfs_super_d.map_data_offset = nfs_super_d.map_inode_offset + NFS_BLKS_SZ(nfs_super_d.map_inode_blks);
// 索引节点最大个数，700
nfs_super.max_ino = NFS_MAX_INO;
// 第一个索引节点的偏移
nfs_super_d.inode_offset = nfs_super_d.map_data_offset + NFS_BLKS_SZ(nfs_super_d.map_data_blks);
// 第一个数据块的偏移
nfs_super_d.data_offset = nfs_super_d.inode_offset + NFS_ROUND_UP(NFS_INO_OFS(nfs_super.max_ino), NFS_BLK_SZ());
// 数据块最大个数
nfs_super.max_data = (nfs_super.sz_disk - nfs_super_d.data_offset) / NFS_BLK_SZ();
```

比较困难的是第一个数据块的偏移的计算，由于在磁盘中 inode 是连续存储的，只在最后将所有 inode 的整体大小和数据块大小进行了对齐，所以应该先根据 inode 的最大个数算出所有 inode 的整体大小，然后使用 NFS_ROUND_UP 函数将该大小和数据块大小 NFS_BLK_SZ()向上对齐，最后加上第一个 inode 的偏移即可算出第一个数据块的偏移。

（12）int nfs_umount()：卸载 nfs

该函数的作用是卸载文件系统，完成超级块回写设备、驱动的关闭、更多必要结构的回写等操作，以保证下一次挂载能够恢复 ddriver 设备中的数据，若成功则返回 0，否则返回相关错误信息。

具体流程如下：



2.3. 钩子函数

该文件系统中使用到的所有钩子函数的对应关系如下：

```c
static struct fuse_operations operations = {
    .init = nfs_init,                   /* mount文件系统 */
    .destroy = nfs_destroy,             /* umount文件系统 */
    .mkdir = nfs_mkdir,                 /* 建目录，mkdir */
    .getattr = nfs_getattr,             /* 获取文件属性，类似stat，必须完成 */
    .readdir = nfs_readdir,             /* 填充dentrys */
    .mknod = nfs_mknod,                 /* 创建文件，touch相关 */
    .write = NULL,                      /* 写入文件 */
    .read = NULL,                       /* 读文件 */
    .utimens = nfs_utimens,             /* 修改时间，忽略，避免touch报错 */
    .truncate = NULL,                   /* 改变文件大小 */
    .unlink = NULL,                     /* 删除文件 */
    .rmdir  = NULL,                     /* 删除目录， rm -r */
    .rename = NULL,                     /* 重命名，mv */

    .open = NULL,
    .opendir = NULL,
    .access = NULL
};
```

（1）mount：挂载文件系统

当挂载 FUSE 文件系统时，会执行的钩子是.init，我们可以在.init 钩子中完成超级块的读取、位图的建立、驱动的初始化等操作。对应的函数为 void* nfs_init(struct fuse_conn_info * conn_info)。

该函数主要调用了上述工具函数 int nfs_mount(struct custom_options options)来实现挂载，仅在主要函数中实现了错误处理功能：

```c
if (nfs_mount(nfs_options) != NFS_ERROR_NONE) {
    fuse_exit(fuse_get_context()->fuse);
    return NULL;
}
return NULL;
```

因此便不再赘述。

（2）umount：卸载文件系统

当卸载 FUSE 文件系统时，会执行的钩子是.destroy，我们可以在.destroy 钩子中完成超级块回写设备、驱动的关闭、更多必要结构的回写等操作，以保证下一次挂载能够恢复 ddriver 设备中的数据。对应的函数为 void nfs_destroy(void* p)。

该函数主要调用了上述工具函数 int nfs_umount()来实现卸载，仅在主要函数中实现了错误处理功能：

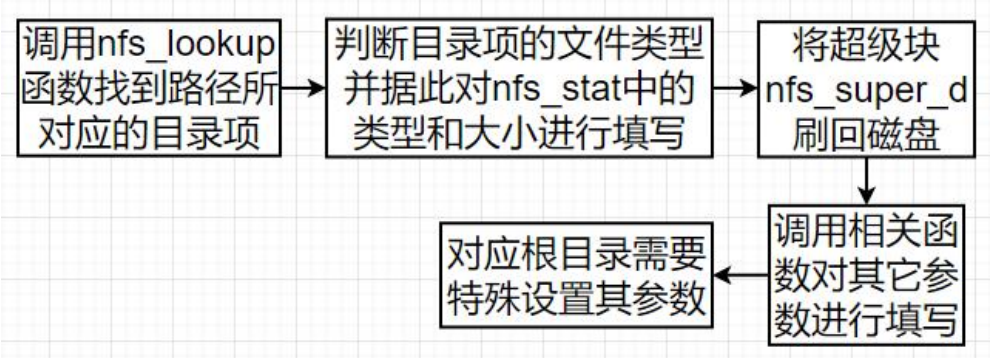
```c
if (nfs_umount() != NFS_ERROR_NONE) {
    fuse_exit(fuse_get_context()->fuse);
    return;
}
return;
```

因此便不再赘述。

（3）mkdir：创建目录

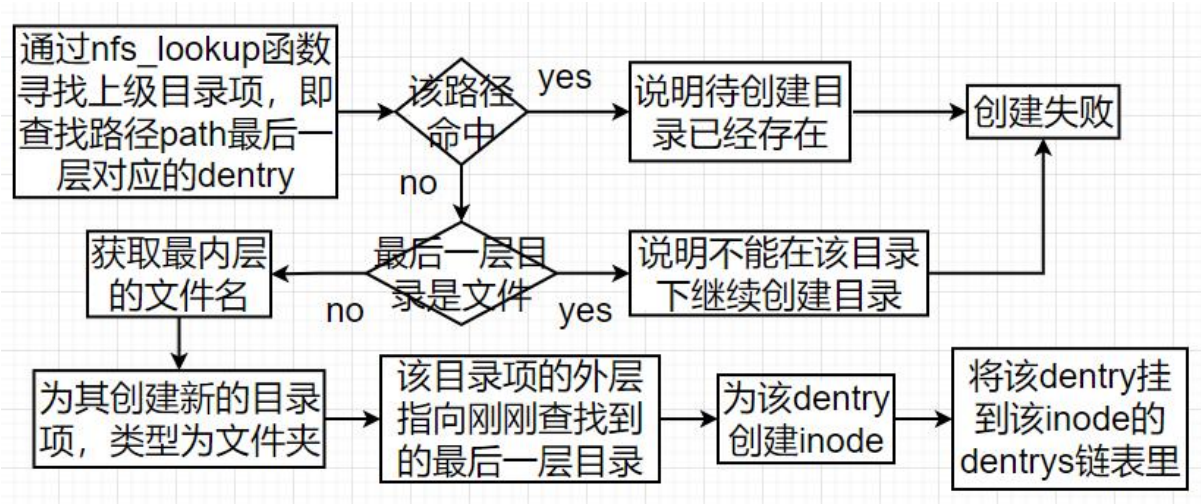
首先，FUSE 文件系统为了获得每个文件的状态，要不断调用 getattr 钩子，这个钩子函数类似于 xv6 里的 fstat。实现 ls、mkdir、touch 等操作的前提就是完成 getattr 钩子的编写。对应的函数为 int nfs_getattr(const char* path, struct stat * nfs_stat)，它用于获取文件或目录的属性，并将相关信息填写在参数 nfs_stat 指针所指向的 stat 数据结构中。

具体流程如下：



然后，我们实现创建目录的相关函数。当为 FUSE 文件系统创建目录时，会执行的钩子是.mkdir。对应的函数为 int nfs_mkdir(const char* path, mode_t mode)，它根据输入参数 path 创建对应的目录，若成功则返回 0，否则返回相关错误值。

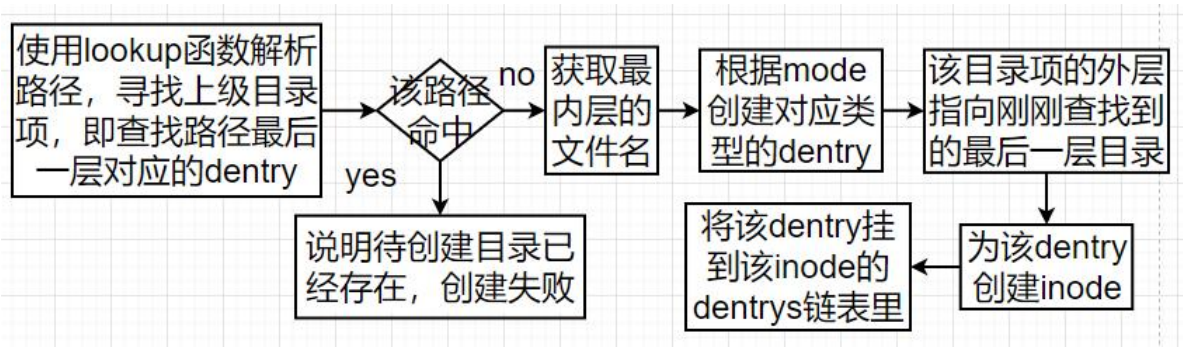
具体流程如下：



（4）touch：创建文件

当为 FUSE 文件系统创建文件时，会执行的钩子是.mknod。对应的函数为 int nfs_mknod(const char* path, mode_t mode, dev_t dev)，它根据输入参数 path 创建对应的文件，若成功则返回 0，否则返回相关错误值。

具体流程如下：

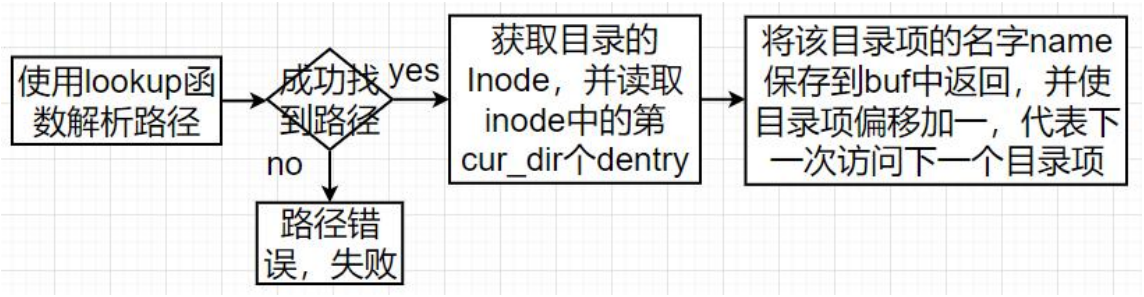


但由于 touch 要求不仅仅是创建文件，还要求可以修改文件的访问时间，因此我们还要实现 utimens 钩子，对应 int nfs_utimens(const char* path, const struct timespec tv[2])函数。这个钩子用于修改文件的访问时间（其实只需要返回 0 就好）。

（5）ls：遍历目录项

当在 FUSE 文件系统下调用 ls 时，就会触发 readdir 钩子，readdir 在 ls 的过程中每次仅会返回一个目录项，对应的函数为 int nfs_readdir(const char * path, void * buf, fuse_fill_dir_t filler, off_t offset, struct fuse_file_info * fi)，其中 offset 参数记录着当前应该返回的目录项，buf 用来保存输出的目录项，若成功则返回 0，否则返回相关错误值。

具体流程如下：



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、对于内存中和磁盘中的数据结构采用了比较不同的设计。在设计**内存中**的 inode 时，对于分别对应目录和文件的 inode 有不同的限制。其中对应文件的 inode，文件原始内容存储在 data_addr_pointer 数组指向的数据块内存地址中，而 dentrys 变量并未使用；对应目录的 inode，全部目录项仅以链表的形式保存在 dentrys 变量中，而没有另外存储在 data_addr_pointer 数组指向的地址中，因此对于**目录对应的 inode，事实上不用给其 data_addr_pointer 数组分配固定的 block，这样可以减小对于内存的使用量，同时便于内存中的访问**。而对于**磁盘中**的 inode，则没有 dentrys 变量，文件原始内容和目录对应的所有目录项 dentry 均存储在 data_addr_pointer 数组指向的数据块中。

2、在**规定每个文件最多使用 6 个数据块的基础上**，新增设计，**规定每个文件夹最多使用 2 个数据块**，尽管 inode 中的数据块号数组仍有 6 项，但若对应的是文件夹，则只使用前 2 项。经计算一个目录项 dentry_d 在磁盘上约占 136B，因此一个数据块大约可存 7 个目录项，所以分配两个数据块，使得每个文件夹大约可存下 14 个目录，这样设计有效地避免了磁盘的浪费：

```
#define NFS_DATA_PER_DIR        2    // 此处为新增设计，一个文件夹最多使用2个数据块，
                                     // 尽管inode中的数据块号数组仍有6项，
                                     // 但若对应的是文件夹，则只使用前2项。
```

3、在 simplefs 文件系统中，每个 inode 单独占用一整块数据块，造成了磁盘的严重浪费，我改为了 **inode 在磁盘中连续存储**，这样可以大大提高磁盘的利用率。同时我**修改了读写磁盘的工具函数**，使得偏移不再按照磁盘 IO 对齐，而只保证每次读写的大小与磁盘 IO 对齐。为保证索引节点后的数据块仍需对齐，我限制了**所有索引节点的总体大小与块大小对齐**，这样**只会在最后一个存储索引节点的块中存在碎片**。

4、考虑到了 inode 与数据分开存储，按照所设计的 nfs_inode_d 数据结构，以及每个对应文件的 inode 分配 6 个数据块，每个对应文件夹的 inode 分配 2 个数据块，较为**精确地设置了 inode 的最大个数，减小了磁盘空间的浪费同时简化了代码。**（此处的 inode 均为磁盘中对应的数据结构）如下图：

```
/**
 * 规定索引节点inode最大个数:
 * 由上，磁盘中最多有4096个block，一个文件不超过6个block，
 * 采用直接索引方式，每个文件对应一个inode，
 * 超级块、inode位图、data位图各占1个block，
 * 则inode个数约为(4096-3)/(6+1)=585 (这是sfs中的计算方式)
 * 由于事实上，一个nfs_inode_d约占40B，即40/1024=0.04个block，
 * (1个block大约可存25个nfs_inode_d)
 * 因此inode个数最大可达约(4096-3)/(6+0.04)=678
 * 又因为规定了一个文件夹最多使用两个数据块，
 * 所以inode个数的最大值可设置的略大于此值，
 * 向上取最接近的25的倍数700为inode的最大个数。
 */
#define NFS_MAX_INO             700
```

二、用户手册

实现的文件系统中的所有命令使用方式

该文件系统包括 mount，mkdir，touch，ls，umount 功能。

1、mount

该功能为挂载文件系统，但 FUSE 文件系统的挂载与传统文件系统的挂载略有不同，它不需要使用 mount 命令，而是直接运行编译出来的二进制文件。

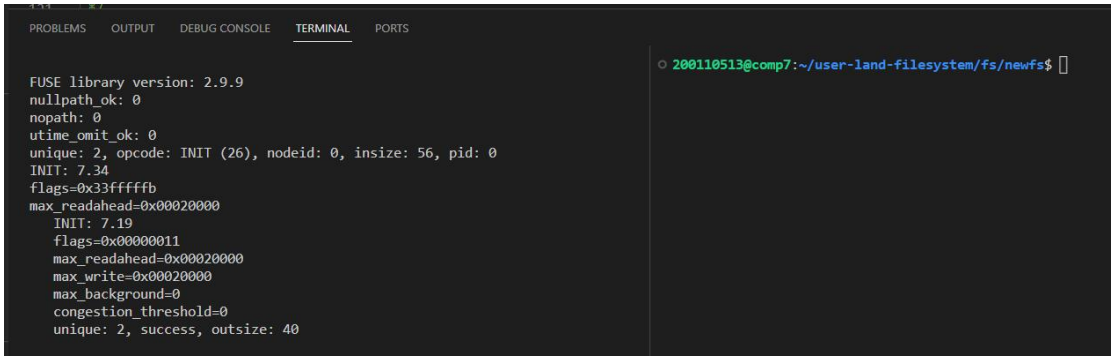
可以直接在 VS Code 中，在 newfs 目录下按 F5 进行编译并运行，也可以使用如下

命令：

```
./build/nfs --device=/home/students/200110513/ddriver -f -d -s ./tests/mnt/
```

上述命令的意义为：将设备/home/students/200110513/ddriver 以 nfs 文件系统形式，挂载到./tests/mnt 目录下。

挂载成功后会显示如下界面（右侧为呼出的另一个终端界面）：



2、mkdir

该功能为创建文件夹，指令为 `mkdir [OPTION]... DIRECTORY...`

其中 DIRECTORY 表示该文件夹的路径，只有当该文件夹不存在时，才会进行创建。

-m, --mode=模式　　设定权限<模式>

-p, --parents　　若该目录已存在，不会显示错误信息,同时将自动建立好路径中那些尚不存在的目录。

-v, --verbose　　每次创建新目录都显示信息

--help　　显示此帮助信息并退出

--version　　输出版本信息并退出

演示如下（首先使用 mkdir 创建各级文件夹，然后使用 ls 命令查看，ls 命令稍后介绍）：

3、touch

该功能为创建文件，指令为 `touch [OPTION]... FILE...`

其中 FILE 表示该文件的路径，只有当该文件夹不存在时，才会进行创建。

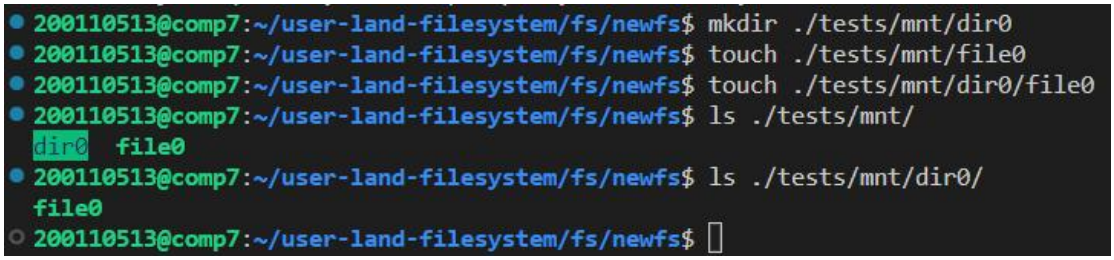
-a　　只改变访问时间

-c, --no-create　　仅修改文件时间，若文件不存在则不创建新文件

-d, --date=STRING　　解析 STRING 并使用它代替当前时间

-f　　(忽略)
-h, --no-dereference　　影响每个符号链接，而不是任何被引用的文件(仅在可以更改符号链接时间戳的系统上有用)
-m　　只改变修改时间
-r, --reference=FILE　　使用该文件的时间而不是当前时间
-t STAMP　　使用[[CC]YY]MMDDhhmm[.ss]格式的时间而不是当前时间
--time=WORD　　更改指定时间:
　　WORD 为 access、atime 或 use 时，相当于-a
　　WORD 为 modify 或 mtime 时，相当于-m
--help　　显示帮助并退出
--version　　输出版本信息并退出

演示如下（首先使用 touch 创建文件，然后使用 ls 命令查看，ls 命令稍后介绍）：



4、ls

该功能为查看文件夹下的所有文件，指令为 `ls [OPTION]... [FILE]...`

其中 FILE 为查看的文件名，若该参数省去，则查看当前所在目录。

-a　　all， 显示所有文件及目录 (. 开头的隐藏文件也会列出)
-A　　同-a ，但不列出 “.”（目前目录）及 “…”（父目录）
-l　　以长格式显示目录下的内容列表，包括文件的权限、链接数、所有者名称和组所有者、文件大小、最后修改日期时间和文件/目录名称
-r　　reverse，将排序结果反向输出，例如：原本文件名由小到大，反向则为由大到小
-R　　recursive，连同子目录内容一起列出来，等于该目录下的所有文件都会显示出来
-S　　sort by file size。根据文件大小排序，而不是文件名
-t　　sort by modification time，以文件修改时间排序（从最新开始排）
-d　　仅列出目录本身，而不是列出目录内的文件数据（常用）
-f　　直接列出结果，而不进行排序 （ls 默认以文件名排序）
-F　　根据文件、目录等信息，给予附加数据结构，例如：*:代表可可执行文件； /:代表目录； =:代表 socket 文件； |:代表 FIFO 文件
-g　　像-l，但是不列出所有者
-G　　no-group，不列出任何有关于组的信息
 - author　　打印出每一个文件的作者
-n　　类似-l，用数字 UID 和 GID 代替名称

-h 将文件大小以人类较易读的方式（例如 GB KB 等等）列
-c 输出文件的 ctime（文件状态最后更改的时间），并根据 ctime 排序
-C 由上至下的列出项目
--full-time 显示完整时间格式
--time 输出 access 时间或改变权限属性时间 （ctime）而非内容变更时间（modification time）

演示如下：

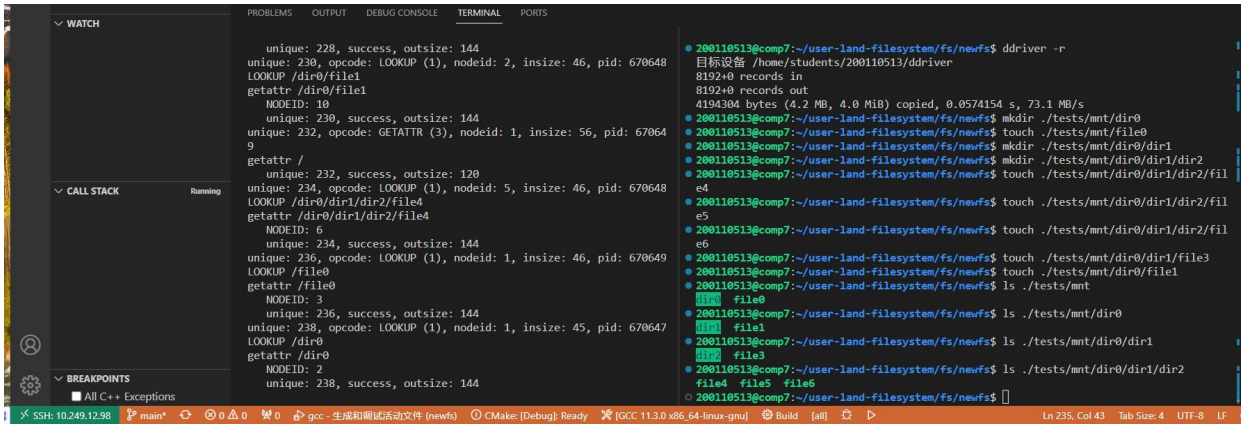
```
200110513@comp7:~/user-land-filesystem/fs/newfs$ mkdir ./tests/mnt/dir0
200110513@comp7:~/user-land-filesystem/fs/newfs$ touch ./tests/mnt/file0
200110513@comp7:~/user-land-filesystem/fs/newfs$ mkdir ./tests/mnt/dir0/dir1
200110513@comp7:~/user-land-filesystem/fs/newfs$ mkdir ./tests/mnt/dir0/dir1/dir2
200110513@comp7:~/user-land-filesystem/fs/newfs$ touch ./tests/mnt/dir0/dir1/dir2/file4
200110513@comp7:~/user-land-filesystem/fs/newfs$ touch ./tests/mnt/dir0/dir1/dir2/file5
200110513@comp7:~/user-land-filesystem/fs/newfs$ touch ./tests/mnt/dir0/dir1/dir2/file6
200110513@comp7:~/user-land-filesystem/fs/newfs$ touch ./tests/mnt/dir0/dir1/file3
200110513@comp7:~/user-land-filesystem/fs/newfs$ touch ./tests/mnt/dir0/file1
200110513@comp7:~/user-land-filesystem/fs/newfs$ ls ./tests/mnt
dir0  file0
200110513@comp7:~/user-land-filesystem/fs/newfs$ ls ./tests/mnt/dir0
dir1  file1
200110513@comp7:~/user-land-filesystem/fs/newfs$ ls ./tests/mnt/dir0/dir1
dir2  file3
200110513@comp7:~/user-land-filesystem/fs/newfs$ ls ./tests/mnt/dir0/dir1/dir2
file4  file5  file6
200110513@comp7:~/user-land-filesystem/fs/newfs$
```

5、unmount

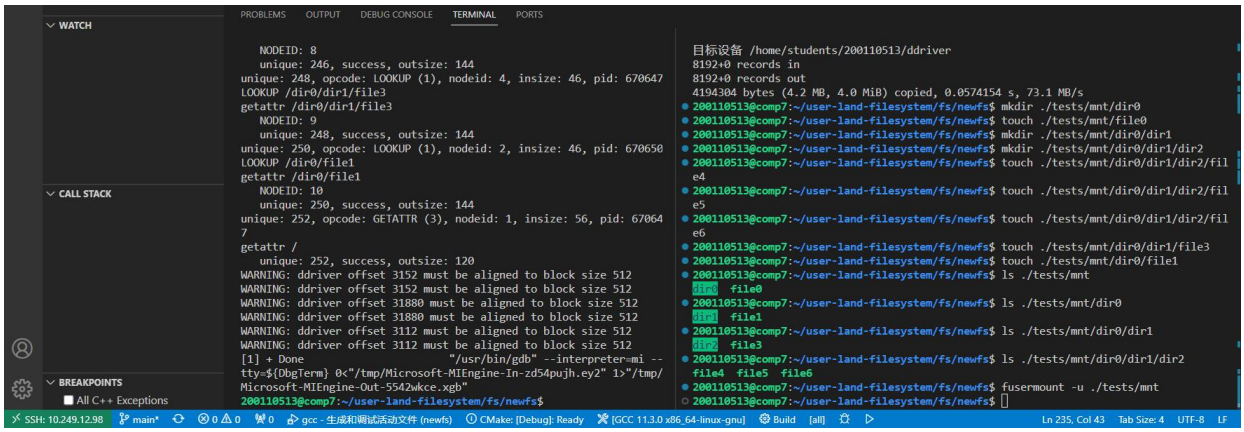
该功能为卸载文件系统，同样，FUSE 文件系统与传统内核文件系统卸载不同，它的卸载命令为 `fusermount -u ./tests/mnt`

演示。

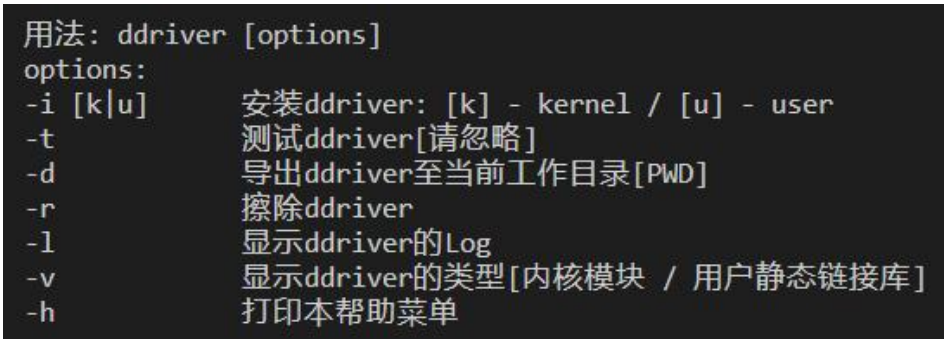
卸载前的界面如下：



卸载完成后：

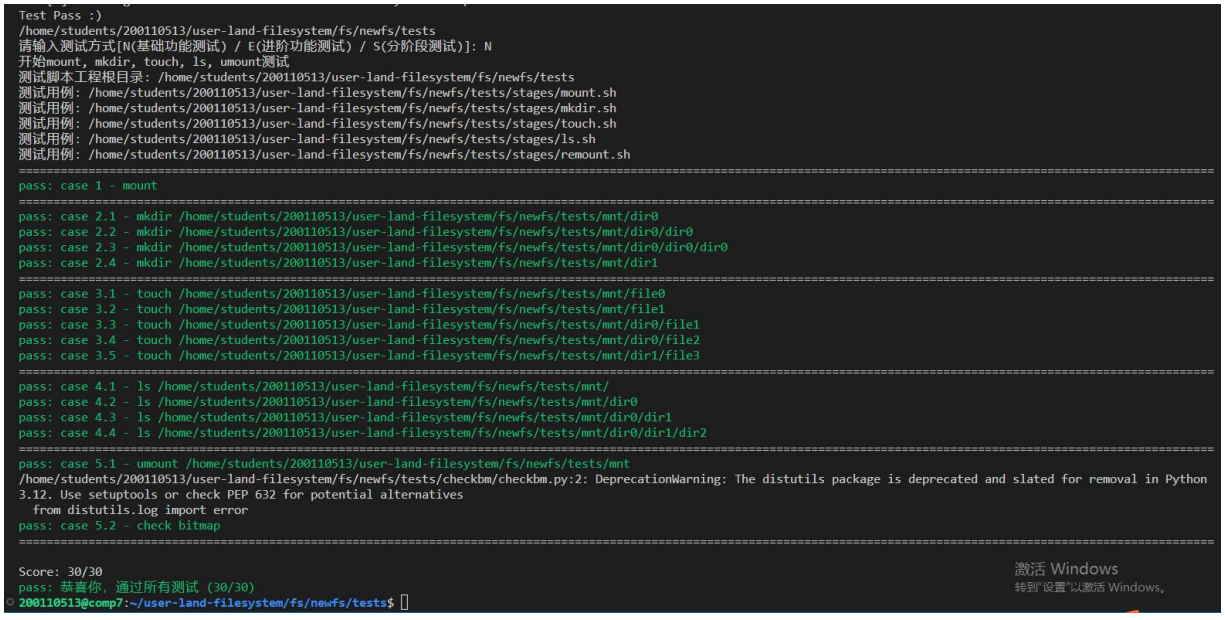


此外，与驱动 ddriver 有关的命令如下：



常用的是 ddriver -r：擦除 ddriver，以及 ddriver -d：导出 ddriver 至当前工作目录。

最后，可以直接使用 tests 文件夹下的 test.sh 脚本进行测试，采用基础功能测试，结果如下：



三、实验收获和建议

实验中的收获、感受、问题、建议等。

收获：

完成这次实验对我来说收获很大，让我学到了很多东西，也搞懂了很多理论课学的一知半解的知识，更是对理论课所学到的知识进行了实际的操作，对于各部分的功能以及它们的联系也有了比较透彻的理解。在完成这次实验的过程中，我不断发现理论课所学的知识与实际实验时所用到的仍存在较大偏差。例如，在理论课中，我们学习了 EXT2 文件系统的布局以及各个模块之间的作用，我本以为拥有这些知识就已经能够很好地完成实验了。但当我真正开始实验时，却发现事实并非如此。在具体实验时，我们还需要考虑到这些模块在内存中是怎么存储的，以及数据是如何在内存和磁盘之间进行交换的。同时，超级块、索引节点、目录项等数据结构在内存和磁盘中都有着不同的作用，如何设计它们在内存和磁盘中的数据结构也是我耗时较多的部分。起初我对于内存和磁盘之间数据交换的过程了解不够深入，因此数据结构设计的较为简略与粗糙。但后来我理解到磁盘数据读入内存和写回磁盘的这段时间中，会被多次修改，因此这些数据结构中的属性值很多都需要拷贝一份到内存中，在写回磁盘时，再用这些内存中的数据更新磁盘块中的数据。

感受：

这次实验给我的感受是总体来说比前几次都困难很多，我也花了不少时间在理解原有代码以及编写自己的代码中。尤其是开始时，对于整个文件系统的理解也没有很深入，所以面对 simplefs 文件系统中的大量代码感到非常头大。实验上手比较困难，需要基本理解 simplefs 文件系统的具体设计以及各函数的作用才能开始实现自己的 EXT2 文件系统，从开始实验直到完成第一个 mount 函数几乎花费了我这次实验总体时间的 70%，所以一开始也因为进展缓慢而感到比较焦虑，不过在完成 mount 函数之后，其它的几个函数就相对来说比较容易了。在完成整个实验再回头看时，发现这些数据结构的设计和函数功能的实现其实好像并没有那么复杂，虽然在写代码时确实是很痛苦()

问题：

实验中遇到的问题主要是花了很多时间理解 simplefs 文件系统的代码含义，以及设计自己的数据结构，还有分配各个块的大小尤其是索引节点的数量。同时由于存在内存和磁盘两种不同的环境，因此各个模块需要设计两种不同的数据结构，它们的功能也较为不同，在实现函数功能时我反复对设计的数据结构进行了多次修改。尤其是在内存中的数据结构，既要存储内存中的信息，又要存储磁盘中的信息，所以比较容易混淆，比如内存中索引节点数据结构的设计，既要包含对应数据块在内存中的地址，又要包含对应数据块在磁盘中的块号，在一开始设计时很难面面俱到。不过由于 simplefs 文件系统的代码十分规范，同时变量名称也易于理解，并且有“_d”用于区分是在内存还是在磁盘，因此花费一定的时间还是能够读懂的。

建议：

希望指导书和测试脚本的 bug 能尽快完善起来，同时由于每位同学的设计不同，所以测试时的数据位图的有效位也不太一样，指导书中可以指出这一点。同时可以把 pizza 平台上同学们遇到的问题整理到指导书中，有些是比较共性的问题，这样可以方便同学们 debug。

四、参考资料

实验过程中查找的信息和资料

实验过程中主要参考了 fs/simplefs 文件夹下的 simplefs 文件系统和实验指导书，以及与同学讨论交流。