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

# 实验报告

开课学期：2021 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：1901105

学生学号：190110509

学生姓名：王铭

评阅教师：

报告成绩：

实验与创新实践教育中心制

2020 年 9 月

# 一、实验详细设计

## 1、 总体设计方案

实验完成了基本功能：

（1）挂载文件系统

（2）卸载文件系统

（3）创建文件/文件夹

（4）查看文件夹下的文件

总体框架为：

基于 FUSE 框架实现，在内存和磁盘使用两套数据结构，超级块用于指示文件系统的基本信息，索引节点位图指示索引节点的使用情况，数据位图指示数据块的使用情况。

总体布局：


图 1.1.1

总体思路：

以内存中超级块为全局变量，在内存中完成文件/目录的建立。通过偏移量进行磁盘内容的读写，完成更新。

inode 位图与索引节点的关系：inode 位图的每一位指示一个索引节点是否被使用，0 表示未使用，1 表示已使用。

数据块位图与数据块的关系：数据块位图的每一位指示一个数据块（两个 io 块）是否被使用，0 表示未被使用，1 表示已被使用。

索引与文件的关系为：通过索引节点号可计算偏移量，从而查找到索引节点，由索引节点的数据块指针记录的数据块号可计算偏移量，从而找到文件内容。若该索引节点的属性是目录，则其所保存的文件内容为该目录下的所有目录项。

# 2、 功能详细说明

各个数据结构如下所示，基本成员的含义可见注释。

**(1)超级块**

内存中定义如下：

```
//super block
struct nfs_super {
    int fd;         // 系统文件号
    int      max_ino; // 最大文件数,一个文件一个inode
    int     io_sz;  //io基本块大小
    int     disk_sz; // 磁盘大小
    int     map_inode_blks;// inode位图占的磁盘块大小
    int     map_data_blks;// data位图占的磁盘块大小
    int     inode_offset; // 索引的偏移量
    int     data_offset; // 数据块的偏移量
    int     use_sz; // 使用的数据块数,一个数据块 = 两个io基本块
    struct nfs_dentry*  root_dentry; //根目录
    uint8_t*    map_inode; // inode位图
    uint8_t*    map_data; // data位图
};
```

图 1.2.1

在超级块的内存表示中，保留 inode 位图和 data 位图，便于根据位图分配新的 inode 和数据块，保留了 inode 和 data 的偏移量，偏于读写磁盘的 inode 和数据块

磁盘中定义如下：

```
struct super_block_d
{
    uint32_t           magic_num;              // 幻数

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

    int                map_inode_blks;         // inode位图占用的块数
    int                map_inode_offset;       // inode位图在磁盘上的偏移

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

    int                inode_offset;           // inode区偏移量
    int                data_offset;            // 数据区偏移量
    int                 use_sz;                // 使用的数量
};
```

图 1.2.2

**(2)索引节点**

内存中定义如下：

```
// inode
struct nfs_inode {
    uint32_t ino;       //下标
    int      size;                  // 文件已占用空间
    int      dir_cnt;               // 目录项数量
    NFS_FILE_TYPE ftype;
    struct nfs_dentry*  dentry;              // 指向该inode的dentry
    struct nfs_dentry*  dentrys;             // 所有目录项
    int      block_pointer[6];      // 数据块指针
};
```

图 1.2.3

设置了成员 dentrys，若该 inode 所指向的是目录，则在内存中加载其目录下的所有目录项，用链表将其连接，方便遍历，dentrys 即为该链表的头节点。

磁盘中定义如下：

```
struct inode_d
{
    int               ino;               // 在inode位图中的下标
    int               size;              // 文件已占用空间
    // int               link;              // 链接数
    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               block_pointer[6];  // 数据块指针（可固定分配）
};
```

图 1.2.4

**(3)目录项**

内存中定义如下：

```
// 目录项
struct nfs_dentry {
    char     name[MAX_NAME_LEN];
    uint32_t ino;
    NFS_FILE_TYPE type;
    int      valid;
    struct nfs_dentry* brother;
};
```

图 1.2.5

设置成员 brother 是为了串接链表，完成对目录下的所有目录项的遍历。

磁盘中定义如下：

```
struct 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
    int               valid;                      // 该目录项是否有效
};
```

图 1.2.6

从磁盘中读取内容，涉及到许多计算偏移量的内容，为此，定义宏来方便计算 inode、data 以及位图的偏移量。其中 NFS_ROUND_DOWN 和 NFS_ROUND_UP 是将偏移量分别与 512 向下对齐和向上对齐。NFS_INO_OFF 是根据 inode 号计算 inode 的偏移量，NFS_DATA_OFF 是根据数据块号计算数据块的偏移量。

```
// 平均每个文件占用多少个io基本块
#define NFS_BLK_PER_FILE 12
#define MAX_NAME_LEN    128
#define NFS_ROUND_DOWN(value, round)    (value % round == 0 ? value : (value / round) * round)
#define NFS_ROUND_UP(value, round)      (value % round == 0 ? value : (value / round + 1) * round)
#define NFS_INO_OFF(ino)                (super.inode_offset + ino * sizeof(struct inode_d))
#define NFS_DATA_OFF(blknum)                (super.data_offset + blknum * super.io_sz * 2)
#define NFS_DATA_BLK_SZ                   2*super.io_sz
#define NFS_DEFAULT_PERM 0777
```

图 1.2.7

**实现的接口函数：**

**(1)int nfs_driver_read(int offset,uint8_t* param,int size)**

由于驱动读一次只能读 512 个字节，不能灵活的读取磁盘内容，因此实现此函数完成对读操作的封装，使得能够通过传入的偏移量以及要读取的字节数读出相应的内容。

具体实现：首先将用户传入的偏移量与 512 对齐，得到要读内容所在的起始块偏移量 offset_aligned。接着计算用户传入的偏移量距起始块的首地址的偏移量 bias。计算用户要读入文件大小的偏移量 size_aligned（也与 512 对齐）。之后根据要读的大小，每次读一个块，最后拼接选择用户要读的内容即可。

主要代码如下:

```
int      offset_aligned = NFS_ROUND_DOWN(offset, super.io_sz);
int      bias           = offset - offset_aligned;
int      size_aligned   = NFS_ROUND_UP((size + bias), super.io_sz);
uint8_t* temp = (uint8_t*)malloc(size_aligned);
uint8_t* cur = temp;
// 移动磁盘头
ddriver_seek(super.fd,offset_aligned,SEEK_SET);
while(size_aligned != 0){
    ddriver_read(super.fd,cur,super.io_sz);
    size_aligned -= super.io_sz;
    cur += super.io_sz;
}
memcpy(param,temp + bias,size);
```

图 1.2.8

**(2)nfs_driver_write**

与读操作类似，封装写操作，使得能够通过传入的偏移量以及要写的字节数写入相应的内容。

基本思路与封装读相同，但需要先将用户要写的内容所在块中未改动的内容先读出来与用户要写入的内容拼接，随后每块进行写入。

```
nfs_driver_write(int offset,uint8_t* param,int size){
    int      offset_aligned = NFS_ROUND_DOWN(offset, super.io_sz);
    int      bias           = offset - offset_aligned;
    int      size_aligned   = NFS_ROUND_UP((size + bias), super.io_sz);
    uint8_t* temp = (uint8_t*)malloc(size_aligned);
    uint8_t* cur = temp;
    // 写操作也要整块写，故先读出来没修改的部分，覆盖掉要修改的部分
    nfs_driver_read(offset_aligned,temp,size_aligned);
    memcpy(temp + bias,param,size);
    ddriver_seek(super.fd,offset_aligned,SEEK_SET);
    while(size_aligned != 0){
        ddriver_write(super.fd,cur,super.io_sz);
        size_aligned -= super.io_sz;
        cur += super.io_sz;
    }
    free(temp);
```

图 1.2.9

**(3)new_dentry**

新建目录项初始化并返回。实现代码如下：

```
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));
    memcpy(dentry->name,fname,MAX_NAME_LEN);
    dentry->type = ftype;
    dentry->ino = -1;
    dentry->brother = NULL;
    dentry->valid = 1;
    return dentry;
}
```

图 1.2.10

**(4)nfs_new_inode**

新申请一个 inode 并返回。

具体实现：根据内存的超级块中保存的 inode 位图，查找没有使用过的 inode，将 inode 号赋值给新申请的 inode 并更新超级块中 inode 位图的相应 bit 为 1。之后去数据位图中查找未使用过的数据块，分配给 inode 的数据区 6 个数据块并更新超级块中的数据位图相应 bit 为 1。若 inode 位图或数据位图不满足上述条件，则返回为空，否则返回新申请的 inode 节点。

查找 inode 位图和数据位图的实现代码如下：

```
    for(;byte_cursor < super.map_inode_blks * super.io_sz;byte_cursor++){
        uint8_t* cur =  super.map_inode + byte_cursor;
        for(bit_cursor = 0;bit_cursor < 8;bit_cursor++){
            if(((*cur) & (1 << (7 - bit_cursor))) == 0){
                (*cur) = (*cur) | (1 << (7 - bit_cursor));
                flag = 1;
                break;
            }
            ino_cursor++;
        }
        if(flag) break;
    }
```

图 1.2.11

```
        for(byte_cursor = 0;byte_cursor < super.map_data_blks * super.io_sz;byte_cursor++){
        uint8_t* cur =  super.map_data + byte_cursor;
        for(bit_cursor = 0;bit_cursor < 8;bit_cursor++){
            if(((*cur) & (1 << (7 - bit_cursor))) == 0){
                (*cur) = (*cur) | ((1 << (7 - bit_cursor)));
                inode->block_pointer[cnt++] = data_cursor;
                if(cnt == 6)
                    break;
            }
            data_cursor++;
        }
        if(cnt == 6) break;
    }
```

图 1.2.12

**(3)nfs_read_node**

通过传入的 inode 号以及指向该索引节点的目录项 dentry 从磁盘中读取索引节点，封装为内存的 inode 并返回该 inode。

具体实现：首先根据 inode 号计算偏移量，利用封装好的读函数 nfs_driver_read 从磁盘中读取并封装为内存结构的 inode，若此 inode 指向的是目录，则需要将其目录下的目录项也加载到内存，并用 inode 的 dentrys 链接起来。

读取目录项并链接的代码如下：

```
if(inode_d.ftype == NFS_DIR && inode_d.dir_cnt > 0){
    int offset;
    struct dentry_d dentry_d;
    offset = NFS_DATA_OFF(inode_d.block_pointer[0]);
    nfs_driver_read(offset,(uint8_t*)(&dentry_d),sizeof(struct dentry_d));
    struct nfs_dentry* dentry_cd = new_dentry(dentry_d.fname,dentry_d.ftype);
    dentry_cd->ino = dentry_d.ino;
    dentry_cd->valid = dentry_d.valid;
    inode_ret->dentrys = dentry_cd;
    offset += sizeof(struct dentry_d);
    for(int i = 1;i < inode_d.dir_cnt;i++){
        // int blk_num = (i) * sizeof(struct dentry_d) / (2 * super.io_sz);
        nfs_driver_read(offset,(uint8_t*)(&dentry_d),sizeof(struct dentry_d));
        struct nfs_dentry* dentry_cd1 = new_dentry(dentry_d.fname,dentry_d.ftype);
        dentry_cd1->ino = dentry_d.ino;
        dentry_cd1->valid = dentry_d.valid;
        dentry_cd->brother = dentry_cd1;
        dentry_cd = dentry_cd1;
        offset += sizeof(struct dentry_d);
    }
}
```

图 1.2.13

**(4)nfs_write_node**

将传入的索引节点 inode 写回到磁盘中。

具体实现：首先根据传入的 inode 封装为内存中存储的 inode 的数据结构，根据 inode 号计算好偏移量后即可利用封装的写函数 nfs_driver_write 写入磁盘即可。若该 inode 指向的是目录，则需要将该目录下的目录项写入到 inode 的数据块区域存储。

实现将目录项写入到数据块的代码：

```
if(inode->ftype == NFS_DIR){
    // 该inode指向的是目录项
    offset = NFS_DATA_OFF(inode->block_pointer[0]);
    dentry_cursor = inode->dentrys;
    while(dentry_cursor != NULL){
        memcpy(dentry_d.fname,dentry_cursor->name,MAX_NAME_LEN);
        dentry_d.ftype = dentry_cursor->type;
        dentry_d.ino = dentry_cursor->ino;
        dentry_d.valid = dentry_cursor->valid;
        nfs_driver_write(offset,(uint8_t*)(&dentry_d),sizeof(struct dentry_d));
        dentry_cursor = dentry_cursor->brother;
        offset += sizeof(struct dentry_d);
    }
```

图 1.2.14

**(5)nfs_look_up**

根据传入的文件路径，查找在文件系统中是否存在该路径指示的文件/目录，若存在返回该文件/目录对应的索引节点。

具体实现：首先计算函数的层级，若为 0，则表示是根目录，返回即可。否则，从根目录的 dentrys 开始，逐级向下寻找，若最终找到则返回该 inode 节点，未找到则返回为空。

```
    struct nfs_dentry* dentry_cur,*dentry_ret;
    char* path_cp = (char*)malloc(sizeof(path));
    strcpy(path_cp,path);
    char* fname = strtok(path_cp,"/");
    dentry_cur = super.root_dentry;
    while(fname != NULL){
        total_lv--;
        struct nfs_inode* inode = nfs_read_inode(dentry_cur->ino,dentry_cur);
        dentry_cur = inode->dentrys;
        while(dentry_cur != NULL){
            if(memcmp(dentry_cur->name,fname,strlen(fname)) == 0){
                dentry_ret = dentry_cur;
                if(total_lv == 0){
                    flag = 1;
                }
                else{
                    fname = strtok(NULL,"/");
                }
                break;
            }
            else{
                dentry_cur = dentry_cur->brother;
            }
        }
        if(flag == 1)break;
        if(dentry_cur == NULL)return NULL;
    }
    if(flag){
        (*find) = 1;
        return nfs_read_inode(dentry_ret->ino,dentry_ret);
    }
    else
        return NULL;
```

图 1.2.15

**(6)nfs_cal_lvl**

计算传入的文件/目录所处级别并返回，例如根目录/所在层级为 0，之后每多一级则加一。

实现代码如下：

```
int
nfs_calc_lvl(const char * path) {
    char* str = path;
    int   lvl = 0;
    if (strcmp(path, "/") == 0) {
        return lvl;
    }
    while (*str != NULL) {
        if (*str == '/') {
            lvl++;
        }
        str++;
    }
    return lvl;
}
```

图 1.2.16

在实现了上述接口函数后，便可轻松实现如下功能：

**1.挂载文件系统**

nfs_mount:

该函数完成挂载文件系统的功能，主要完成的是将超级块加载到内存当中，将根目录读入到内存里并保存在超级块中，便于查询、创建等操作。根据读出的幻数，若发现当前挂载为首次挂载，则进行对文件系统的估算，构造文件系统的超级块并写回到磁盘中。

该函数的实现流程图为图 1.2.17 所示：

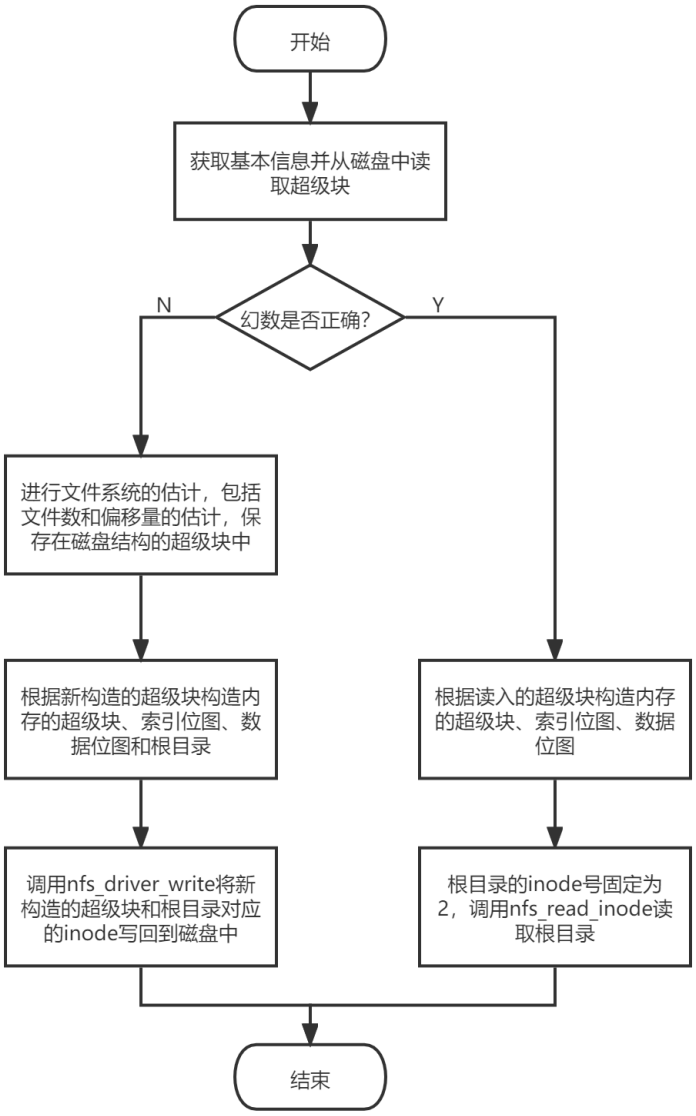


图 1.2.17

**2.卸载文件系统**

nfs_umount:

该函数较为简单，完成卸载文件系统的功能，主要完成对磁盘中的超级块和位图的更新实现代码见下图：

```
int
nfs_umount(){
    // 写回superblk
    struct super_block_d super_block_d;
    nfs_driver_read(0,(uint8_t*)(&super_block_d),sizeof(struct super_block_d));
    super_block_d.use_sz = super.use_sz;
    nfs_driver_write(0,(uint8_t*)(&super_block_d),sizeof(struct super_block_d));
    // 更新datamap和inodemap
    nfs_driver_write(super_block_d.map_inode_offset,super.map_inode,super_block_d.map_inode_blks * super.io_sz);
    nfs_driver_write(super_block_d.map_data_offset,super.map_data,super_block_d.map_data_blks * super.io_sz);
    ddriver_close(super.fd);

    return 0;
}
```

图 1.2.18

**3.创建目录**

nfs_mkdir

该函数完成创建目录的功能，主要流程如下图所示：

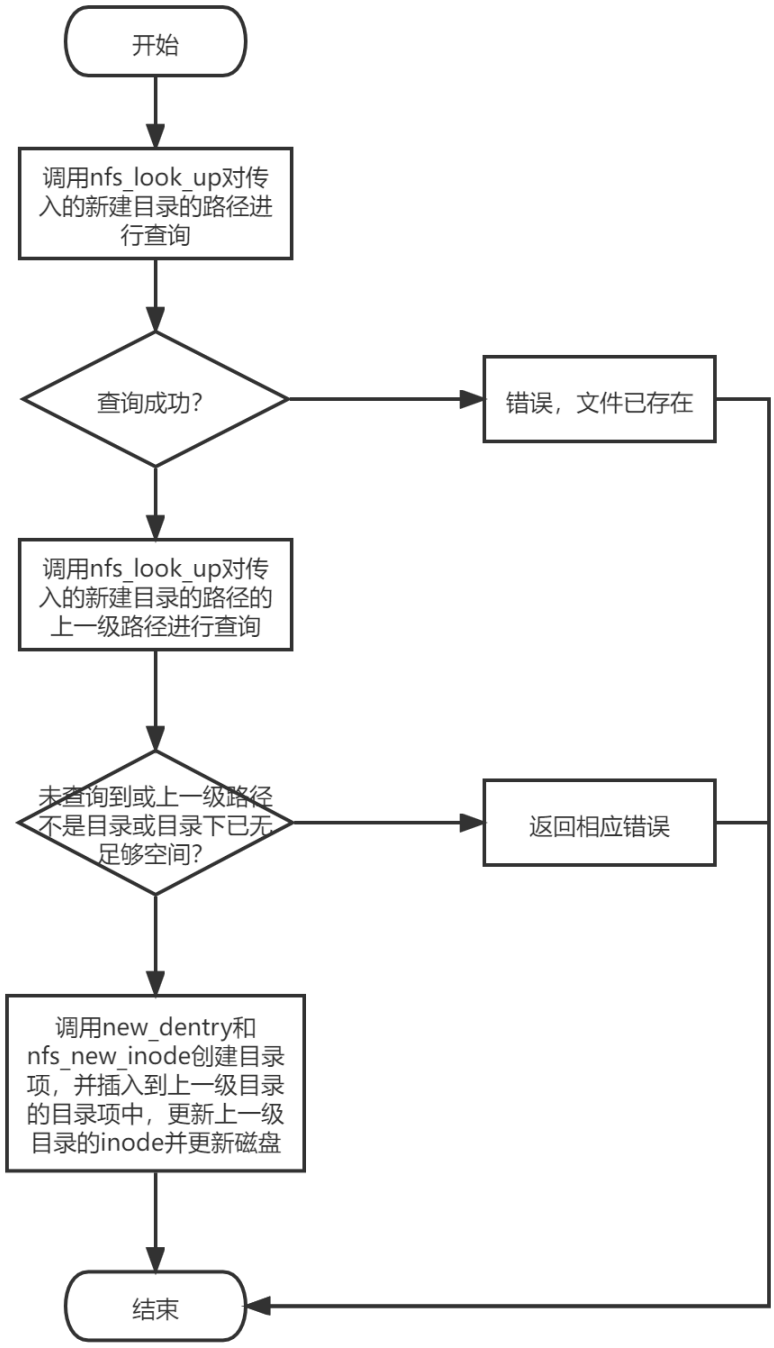


图 1.2.19

其中申请新的 inode 和 dentry 以及插入到上一级目录下的代码实现如下图：

```
    // 将新建的目录项插入到上级目录的dentry下
    struct nfs_dentry* dentry_new = new_dentry(fname,NFS_DIR);
    struct nfs_inode* inode_new = nfs_new_inode(dentry_new);
    dentry_new->brother = inode->dentrys;
    inode->dentrys = dentry_new;
    inode->dir_cnt++;
    inode->size += sizeof(struct dentry_d);
    // 写回磁盘
    nfs_write_inode(inode);
    // struct nfs_inode* test_inode = nfs_read_inode(2,super.root_dentry);
    nfs_write_inode(inode_new);
    // print();
    return 0;
}
```

图 1.2.20

**4.创建文件**

nfs_mknod

该函数完成普通文件的创建，具体流程与创建目录相同，只是新建 inode 属性为 file，可参考创建目录。

**5.获取文件属性**

nfs_getattr

具体实现参考 simplefs，首先调用封装好的 nfs_look_up 函数，获得当前要获取文件的 inode，根据 inode 的具体信息填写 nfs_stat 即可。具体代码实现如下：

```
int nfs_getattr(const char* path, struct stat * nfs_stat) {
    /* TODO: 解析路径，获取Inode，填充nfs_stat，可参考/fs/simplefs/sfs.c的sfs_getattr()函数实现 */
    printf("getattr:%s\n",path);
    int find,root;
    find = root = 0;
    struct nfs_inode* inode= nfs_look_up(path,&find,&root);
    if(inode == NULL)return -NFS_ERROR_NOTFOUND;
    if(inode->ftype == NFS_FILE){
        nfs_stat->st_mode = NFS_DEFAULT_PERM | S_IFREG;
        nfs_stat->st_size = inode->size;
    }
    else if(inode->ftype == NFS_DIR){
        nfs_stat->st_mode = NFS_DEFAULT_PERM | S_IFDIR;
        nfs_stat->st_size = inode->dir_cnt * sizeof(struct dentry_d);
    }
    nfs_stat->st_nlink = 1;
    nfs_stat->st_uid = getuid();
    nfs_stat->st_gid = getuid();
    nfs_stat->st_atime   = time(NULL);
    nfs_stat->st_mtime   = time(NULL);
    nfs_stat->st_blksize = super.io_sz * 2;

    if(root){
        nfs_stat->st_size = super.use_sz * 2 * super.io_sz;
        nfs_stat->st_nlink = 2;
        nfs_stat->st_blocks = super.disk_sz / 2 / super.io_sz;
    }

    return 0;
}
```

图 1.2.21

**6.读取目录项**

nfs_readdr

先获取要获取路径的目录的 inode，根据传入的偏移值 offset，用 filler 函数填充缓冲区即可。

```
int nfs_readdir(const char * path, void * buf, fuse_fill_dir_t filler, off_t offset,
                struct fuse_file_info * fi) {
    /* TODO: 解析路径，获取目录的Inode，并读取目录项，利用filler填充到buf，可参考/fs/simplefs/sfs.c的sfs_readdir()函数实现 */
    // printf("解析路径%s···\n",path);
    int find,root;
    find = root = 0;
    struct nfs_inode* inode = nfs_look_up(path,&find,&root);
    struct nfs_dentry* dentry;
    if(find){
        int cnt = 0;
        dentry = inode->dentrys;
        while(dentry){
            if(cnt == offset){
                filler(buf,dentry->name,NULL,++offset);
                return 0;
            }
            cnt++;
            dentry = dentry->brother;
        }
        return 0;
    }
    return -NFS_ERROR_NOTFOUND;
}
```

图 1.2.22

# 二、用户手册

1.挂载：在 CMakelist.txt 中按下 ctrl+shift+P 进行编译，在 nfs.c 中按 F5 进行挂载及后续使用。

2.卸载：在终端输入 fusermount -u “挂载点路径” 即可。注：不能在挂载路径下卸载，应该进入到挂载点的上级目录卸载

3.创建目录：与 linux 相同，在终端中输入 mkdir “上级目录路径/目录名” 即可。

4.创建文件：与 linux 相同，在终端中输入 touch “上级目录路径/文件名”即可。

5.显示当前目录下的目录/文件：在终端中通过 cd 命令进入要查看的目录，输入 ls 即可显示当前所在目录下的目录/文件

# 三、实验收获和建议

收获：通过本学期的操作系统实验课，我熟悉了 linux 的部分命令的使用，了解了进程通信中的管道通信，实现了一个简单的系统调用，熟悉了进程在具体实现中的 PCB 结构，在进程并发过程中实现同步中锁的基本使用及作用，以及通过页表实现的虚拟内存机制和其具体的翻译机制，最后实现了一个简单的文件系统。通过这些实验内容，我较好的理解了理论课上的内容，提高了自己对操作系统的理解，在实验过程中也提高了自己的代码能力。

部分建议：个人认为某些知识只是简单提及并提供使用，并未具体讲解，如 FUSE 框架等，总有种不清楚的感觉。希望能更详细的讲解，便于入手和调试程序，提高写实验的效率。

# 四、参考资料

1. Lab5：基于 FUSE 的青春版 EXT2 文件系统的实验指导书。
2. 本实验具体实现主要参考的是 simplefs 文件系统的实现。