



华中农业大学
HUAZHONG AGRICULTURAL UNIVERSITY

综合实训报告

Ext2 文件系统仿真

姓名： 付义

学号： 2016317200308

专业： 计算机科学与技术

指导老师： 任继平 王颖 李小霞

中国·武汉

二〇一九年七月

2019.07

目录

1 实训目的及内容.....	2
1.1 实训目的.....	2
1.2 实训内容.....	2
1.3 报告结构.....	2
2 原理分析.....	2
3 系统设计.....	4
3.1 数据结构设计.....	4
3.2 体系结构设计.....	7
3.2 系统模块设计.....	8
3.3 系统流程设计.....	8
4 系统实现与测试.....	9
4.1 开发环境.....	9
4.2 模块实现.....	9
4.2.1 文件系统初始化.....	9
4.2.2 位图操作.....	10
4.2.3 根节点和 root 初始化.....	10
4.2.4 inode 查找.....	11
4.2.5 文件创建.....	12
4.2.6 文件删除.....	13
4.2.7 文件列出.....	13
4.3 系统测试.....	14
5 结论与展望.....	16
5.1 结论.....	16
5.2 展望.....	16

1 实训目的及内容

1.1 实训目的

1. 在内存中定义 ext2 文件系统数据结构； 初始化文件系统；
2. 将文件系统写到磁盘；
3. 实现 ext2 文件系统内存操作和文件读写，创建，删除等操作；

1.2 实训内容

仿真 linux ext2 文件系统的结构和功能，理解文件系统工作原理

- 文件系统结构体设计
- 文件系统结构初始化
- 功能设计
- 系统运行

1.3 报告结构

实训目的及内容	实训目的，项目内容、结构
原理分析	各部分设计原理，实现原理
系统设计	数据结构设计，文件系统设计，工程结构设计
系统实现与测试	文件系统结构实现，文件操作功能实现
结论与展望	学到的知识，系统存在的缺陷，完善需要的工作

2 原理分析



图 1 磁盘、分区、组块组成

硬盘首先分区,然后格式化,才能使用。格式化的过程会在硬盘上建立很多块,为了管理这些块,文件系统将其分组,每个组称为块组(block group).每个块组又由六部分组成(见图 1):超级块(super block)、块组描述表(group descriptor table)、块位图(blockbitmap)、索引位图(inode bitmap)、索引表(inodetable)和数据块(data block)。

本文件系统定义大小为 **1024*1024 字节大小**,每个块 **1KB**,所以有 **1024 块**;
超级块 1KB,组块描述符 **1KB**,数据块位图 **1KB**,索引位图 **1KB**,inode 节点表 **128KB**,数据块 **892KB**。

- 1) 先用 `dd if=/dev/zero of=/home/ringfu/Linux/ext2 bs=1024 bb=1024` 创建一个 1024*1024B 大小的文件来承载文件系统;创建固定大小的文件是仿真真实文件系统磁盘大小也是固定的;
- 2) 设计文件系统数据结构:超级块, GDT, 数据块位图, 索引位图, inode 节点表, 数据块; 按照自己文件系统大小给每个结构分配一定大小空间; 为了便于分配,给每个结构分配 1KB 大小;
- 3) 文件操作部分有创建文件, 删除文件, 打开文件, 退出文件系统; 基本操作都差不多,都是先找到 inode 或分配 inode, 置位 inode 位图和数据块位图; 最后得到 inode 对应的数据块之后对数据块进行操作;
- 4) 其中比较重要的是通过文件名找到 inode 节点编号,这是作为后面创建文件, 删除文件, 读写文件等操作的基础。
- 5) 整个系统的原理是: 首先输出提示符提示用户输入, 得到用户输入功能之后调用相应功能并读取参数, 之后调用相应的功能; 将功能返回的结构写回磁盘或者打印到屏幕; 等到用户输入 `exit` 后退出系统。

3 系统设计

/home/ringfu/linux/ext2 . 1024 * 1024B 大小.

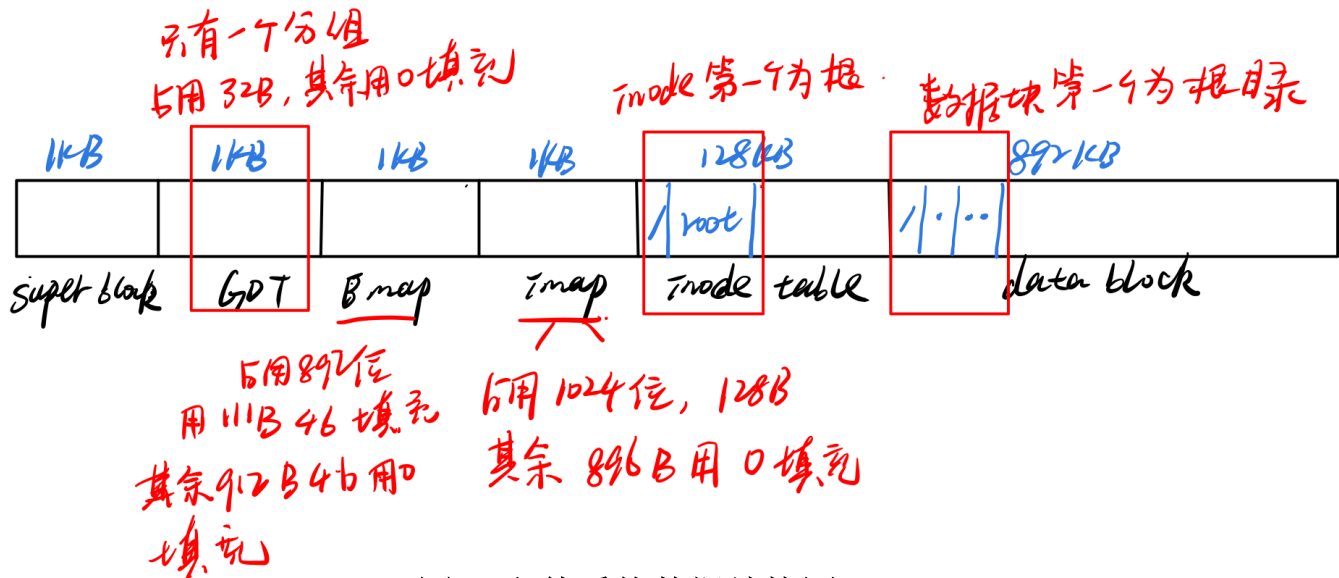


图 2 文件系统数据结构图

本文件系统定义大小为 1024*1024 字节大小, 每个块 1KB, 所以有 1024 块;
超级块 1KB, 组块描述符 1KB, 数据块位图 1KB, 索引位图 1KB, inode 节点
表 128KB, 数据块 892KB。

3.1 数据结构设计

3.1.1 超级块结构设计

super block 保存该文件系统主要信息;

其中主要的几个参数是:

```
__u32 s_inodes_count;      /* 文件系统中索引节点总数 */
__u32 s_blocks_count;     /* 文件系统中总块数 */
__u32 s_free_blocks_count; /* 文件系统中空闲块总数 */
__u32 s_free_inodes_count; /* 文件系统中空闲索引节点总数 */
__u32 s_first_data_block; /* 文件系统中第一个数据块 */
__u16 s_inode_size;       /* 索引节点的大小 */
__u16 s_block_group_nr;   /* 该超级块的块组号 */
char s_volume_name[16];   /* 卷名 */
```

```
char s_last_mounted[64]; /* 最后一个安装点的路径名 */
```

参数填写是按照文件系统大小和各部分大小来设置的；

卷名为 /home 安装点路径为 /home/ringfu/Linux/ext2

3.1.2 组块描述符表结构设计

块组中，紧跟在超级块后面的是组描述符表，其每一项称为组描述符，定义为 ext2_group_desc 的数据结构，共 32 字节。它是用来描述某个块组的整体信息的。

主要的参数：

```
struct ext2_group_desc
{
    __u32 bg_block_bitmap; /* 组中块位图所在的块号 */
    __u32 bg_inode_bitmap; /* 组中索引节点位图所在块的块号 */
    __u32 bg_inode_table; /* 组中索引节点表的首块号 */
    __u16 bg_free_blocks_count; /* 组中空闲块数 */
    __u16 bg_free_inodes_count; /* 组中空闲索引节点数 */
};
```

组块描述符表结构：

```
struct ext2_group_desc_table
{
    struct ext2_group_desc GDT[GROUP_NUMBER];
    __u8 padding[GROUP_DESC_BLOCK_LENGTH -
GROUP_NUMBER*GROUP_DESC_SIZE];
};
```

由于我的文件系统只有一个组块，所以组块描述符表里只有一个组块描述符项；但是分配给组块描述符表有 1KB，GDT 只有 32 字节，其余用 0 填充。

3.1.3 数据块位图结构设计

```
struct ext2_block_bitmap
{
    __u8 block_bitmap[1024]; // 1024 个 data block
};
```

3.1.4 索引位图结构设计

```
struct ext2_inode_bitmap{
    __u8 inode_bitmap[1024]; // 1024 个 inode
```

```
};
```

这里需要注意的是：

数据块位图和索引位图中，一个二进制位（1b）对应一个数据块或 **inode**，某位为 1 表示对应的数据块或 **inode** 可用，即可以分配；为 0 表示被占用，不能分配；

在创建文件和删除文件的时候需要申请 **inode** 和数据块以及释放 **inode** 和数据块，就涉及到置位两个位图。

3.1.5 inode 节点表设计

inode 节点表存放 inode，每一个 inode 占用 128B，共 1024 个 inode，所以会占用 128 个数据块。

具体 inode 结构是这样的：

这个比较重要，后面操作都是基于 **inode** 找到数据块，然后对数据块中数据操作；

```
struct ext2_inode {
    //在真实文件系统中文件类型和权限由 i_mode 一个参数确定
    // 这里为了方便，设置两个参数 i_type 表示文件类型，i_mode 表示文件权限

    __u16 i_mode;           /* 文件类型和访问权限 */
    __u16 i_type;           /* 文件类型*/
    __u16 i_links_count;    /* 文件的硬链接计数 */
    __u32 i_blocks;         /* 文件所占块数（每块以 256 字节计）*/
    __u32 i_flags;          /* 打开文件的方式 */
    union
    {
        __u32 i_block[EXT2_N_BLOCKS]; /* 指向数据块的指针数组；可能
        一个文件会占用多个数据块 */
    }
    // 这个 i_block 很重要，通过 inode 找到数据块就是通过这个指针找到

    __u32 i_version;        /* 文件的版本号（用于 NFS） */
    __u8 l_i_frag;          /* 每块中的片数 */
    __u32 i_faddr;          /* 片的地址 */

};

char padding[46];          // 数据填充，为了对齐
};
```

3.1.6 数据块设计

数据块每个占用 1024B，由于系统整个有 1024*1024B 大小，除去前面结构占用的，还有 892 个数据块可用，其中还有留给根目录和 root 目录一部分作为目录项。

3.1.7 目录项结构设计

目录项结构是存放在数据块中，如果当前文件是目录文件，那么这个文件存储在数据块中的就是 **dentry** 结构体，由于目录项大小固定，所以可以通过目录文件大小/目录项大小得到共有多少目录项；

// 目录项结构，256 个字节

```
struct ext2_dir_entry_2 {
    __u32 inode;    // 文件入口的 inode 号，0 表示该项未使用
    __u16 rec_len;  // 目录项长度
    __u8 name_len;  // 文件名包含的字符数
    __u8 file_type; // 文件类型
    char name[128]; // 文件名
    char padding[120]; // 字节对齐
};
```

3.2 体系结构设计

- 1) 磁盘检查，检查能否读写文件系统安装点 /home/ringfu/Linux/ext2;
- 2) 初始化： 超级块，组块描述符表，置位数据块位图，置位索引位图； 对应文件 ext2init.c。
- 3) 创建根节点 “/” 以及 root 目录 “/root”，由于是创建两个目录，所以要创建对应 ‘.’ 和 ‘..’ 表示当前目录和父目录； 还需创建目录项，写入磁盘。
- 4) 执行 funcsel()函数，进入功能选择，这是一个死循环； 首先输出提示符

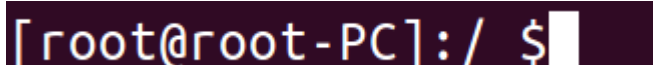


图 3 系统提示符

然后用户输入功能：ls, cd , mkdir, open , clear, exit; 除 exit 是退出函数外，其他都继续进入死循环；输入对应功能后输入参数，然后调用相应函数来处理；结

果写入磁盘或打印到屏幕。

3.2 系统模块设计

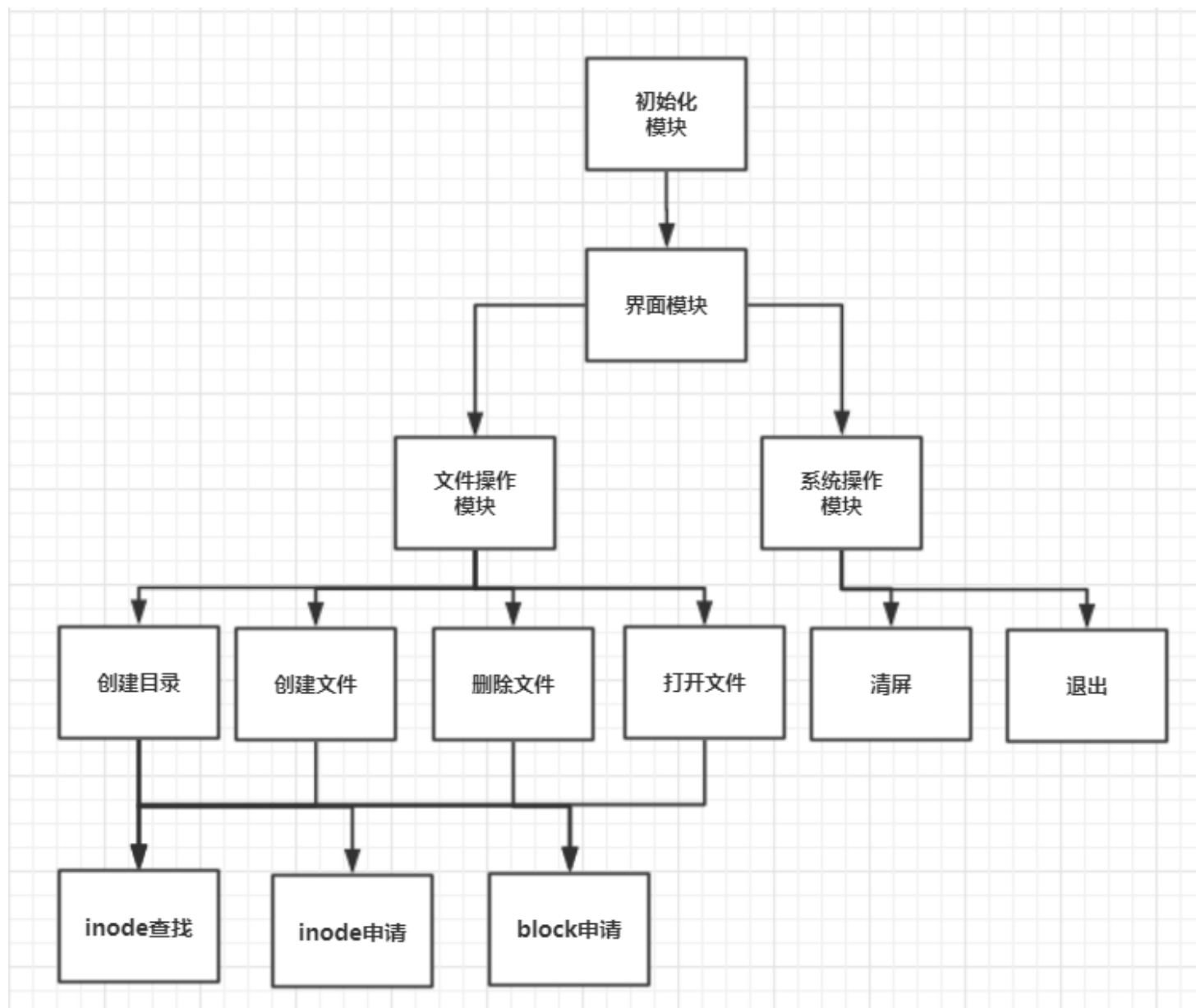


图 4 系统模块

3.3 系统流程设计

进入功能选择，这是一个死循环；首先输出提示符然后用户输入功能：ls, cd, mkdir, open, clear, exit；除 exit 是退出函数外，其他都继续进入死循环；输入对应功能后输入参数，然后调用相应函数来处理；结果写入磁盘或打印到屏幕。

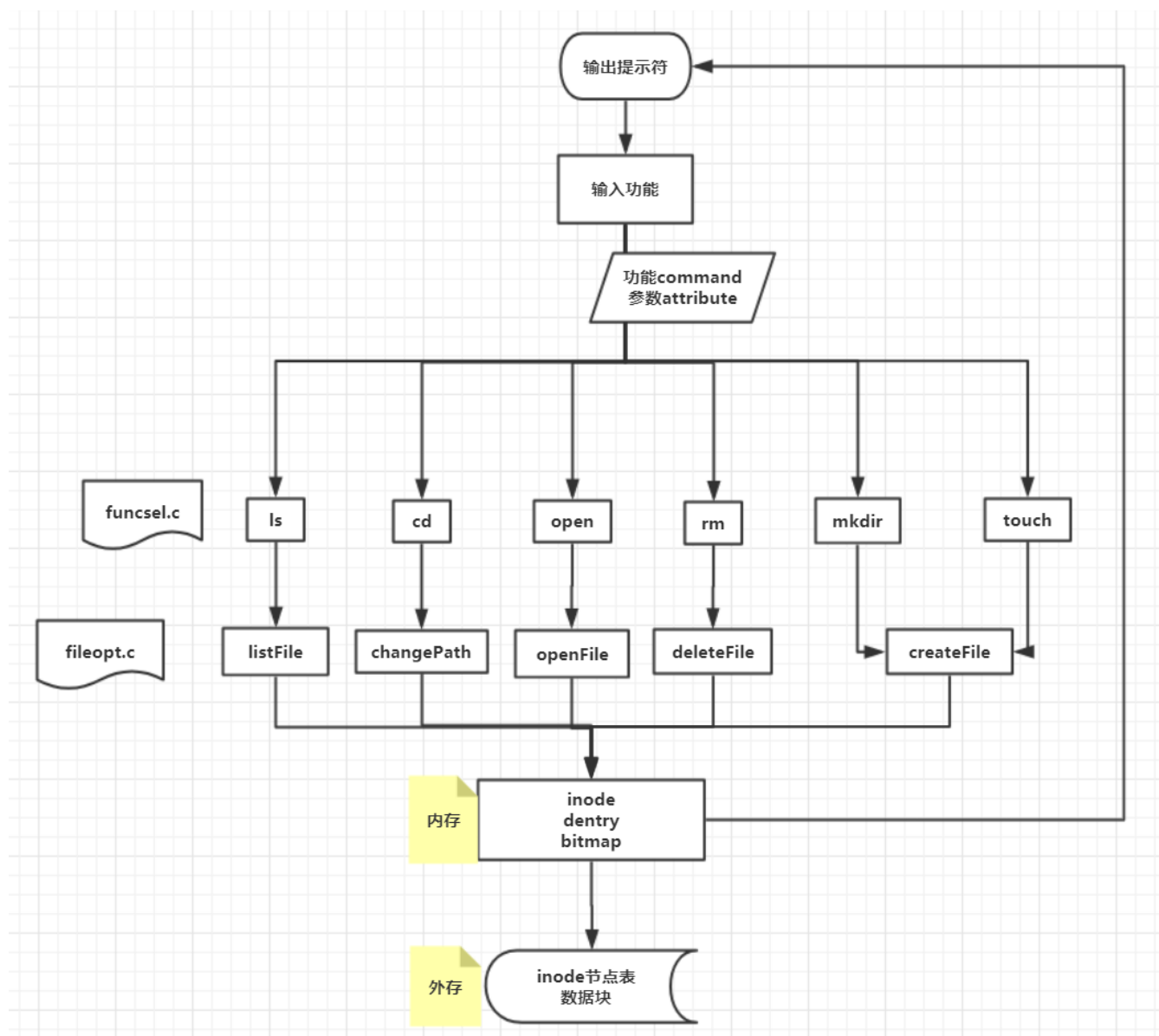


图 5 系统流程

4 系统实现与测试

4.1 开发环境

- 硬件环境：Intel Core i5 7200U 64 位操作系统 基于 x64 处理器
- 软件环境：开发系统 ubuntuLTS 16.04 开发软件 vscode 编译调试 gcc gdb

4.2 模块实现

4.2.1 文件系统初始化

基本上四个重要数据结构（超级块，组块描述符表，数据块位图，索引位图）的

初始化方式都相似：

- i. 首先是在内存申请一个结构体的空间；
- ii. 然后填写相应的参数，参数基本上都是自己设定的，根据文件系统大小和总共的数据块数来计算各个参数的数据；
- iii. 接着通过 `fseek` 函数移动读写指针；例如：
`fseek(disk,SUPER_BLOCK_LENGTH+GROUP_DESC_BLOCK_LENGTH+BLOCK_INDEX_BMP_SIZE,SEEK_CUR);`
- iv. 移动读写指针后，通过 `发 write()` 函数将内存结构体写到对应磁盘的某个位置；例如：`(fwrite(&imap,sizeof(struct ext2_inode_bitmap),1,disk)`

且常通过 `fwrite` 返回值来判断是否写入成功，来决策是继续执行还是错误返回。

4.2.2 位图操作

`BOOL set_block_bitmap(int offset, int value)`

`__u16 get_free_block()`

主要就是两种操作：第一个是获取第一个空闲的位，返回空闲位的编号作为数据块编号或 `inode` 编号。第二个是置位：`offset` 为数据块编号或 `inode` 编号，`value` 为 0 或 1，‘0’ 表示设置这位为占用，表示对应的数据块或 `inode` 被占用；‘1’ 表示设置这位对应的数据块或 `inode` 为可用，表示的是释放对应数据块或 `inode`。

4.2.3 根节点和 root 初始化

创建一个目录文件分为以下几步：

- i. 内存申请 `inode` 和目录项 `dentry`

`__u16 disk_inode_number = get_free_inode();`

在申请 `inode` 之后要设置位图对应位为占用

`set_inode_bitmap(disk_inode_number,BLOCK_INDEX_IN_USE)`

- ii. 创建目录下的 ‘.’ ‘..’ 目录，分别表示当前目录和父目录；同样要申请 `inode` 和 `dentry` 目录项；在当前目录和父目录中最主要的是两个参数：`inode` 节点号和文件名；

`disk_entry.inode = disk_inode_number;`

`strcpy(disk_entry.name, "/");`

- iii. 将在内存中创建的数据目录写入内存

`fseek(disk,FIRST_INODE_BLOCK*EVERY_BLOCK+OUT_INODE_LENGTH*(di`

```
sk_inode_number - 1),SEEK_SET);
fwrite(&disk_dir, OUT_INODE_LENGTH, 1, disk);
```

4.2.4 inode 查找

通过目录名或文件名找到对应 inode: `__u16 findInodeByName(char filename[])`

步骤:

- i. 首先读取根目录 '/' 的 inode, 从根目录开始找;
- ii. 逐层剥去路径名, 通过 `char* path = strtok(filename, delims);` 来获取路径名的首个子串; 例如 `/dir1/dir2/file1`; 依次得到 `dir1, dir2, file1`
- iii. 然后在 '/' inode 对应的数据块中逐个寻找目录项, 通过 `0 == strcmp(path, dir_entry.name)` 来判断子串和目录项中 `dir_entry.name` 是否相等;
- iv. 如果不等, 则匹配下一个目录项;
- v. 如果相等, 则剥去下一个子串, 进入匹配的目录项 inode 对应的数据块的目录项, 重复第二步到第五步, 直到满足条件: `if(flag && !path)` ; **flag** 表示当前目录项名称和剥取子串名称是否相等, **path** 表示剥取得到的子串, 当路径解析到最后, 返回 `NULL`; 这样做是防止出现欲寻找 `/root/usr/usr/file1`, 但是存在 `/root/usr/file1` 这个文件

相关核心代码:

```
for(i=0; i<root_disk.i_links_count; i++){ //i_link_count 表示该目录下还有多少子
    目录
    if(fseek(file_disk, (FIRST_DATA_BLOCK+root_disk.i_block[i/4])*EVERY_BLOCK
    + DIR_DENTRY_LENGTH*(i%4), SEEK_SET)){
        // i/4 的原因: 一个目录项占 256B, 一个数据块占 1KB, 通过 i/4 和 i%4 就可
        以找到对应目录项在数据块中的位置
        printf("find dir!\n"); // i_block[] 存放的是 inode 指向数据块的指针, 下标表
        示在那个数据块
    }else{
        printf("failed find dir\n");
    }
    // 读取目录项
    fread(&dir_entry, DIR_DENTRY_LENGTH, 1, file_disk);
    if(0 == strcmp(path, dir_entry.name)){
        flag = 1;
        printf("dir inode: %d\n", dir_entry.inode);
        break; // 只有当欲寻址路径名和当前指向的路径名相匹配时才退出
    }else{
        continue; // 如果当前指向的路径名和欲寻址的路径名不匹配, 指针
```

移向下一个目录项

```

    }
    // 根据找到的 inode 重定位数据块
    if(0 ==
fseek(file_disk,FIRST_INODE_BLOCK*EVERY_BLOCK+OUT_INODE_LENGTH
H*(dir_entry.inode - 1),SEEK_SET)){
        printf("relocate data block succeed!\n");
    }else{
        printf("relocate data block failed!\n");
    }
    // 读取 inode 对应的数据块，解析数据结构
    fread(&root_disk,OUT_INODE_LENGTH,1,file_disk);
    path = strtok(NULL,delims);
    if(flag && !path){ // flag=1 表示找到对应 inode，!path 表示给定路径名搜索结束
        fclose(file_disk); // 防止出现欲寻找 /root/usr/usr/file1，但是存在
/root/usr/file1 这个文件
        printf("file inode finded!\n");
        return dir_entry.inode;
    }else{
        flag = 0;
    }
}

```

4.2.5 文件创建

- i. 首先申请 inode 和 dentry;
- ii. 然后根据是目录文件还是普通文件;
如果是目录文件，则在新建目录下创建 ‘.’ ‘..’ 目录，同时申请两个 dentry，写到当前目录下;
- iii. 如果是普通文件，则只需申请目录项和 inode，写入磁盘，将新建 inode 写到当前目录的 block 中，表示文件指针;

// 申请数据块，如果是目录文件，则需要申请数据块存放'!'!' 两个目录;
// 普通文件不申请数据块，因为现在是创建文件，还没有写入文件内容

```

if(type == DIR_FILE){
// 如果是目录文件，需要写入两个目录'!'!'
    __u16 free_block = get_free_block(); // 如果是新建目录，则需要申请数据块，普通文件不申
        set_block_bitmap(free_block+1, BLOCK_INDEX_IN_USE);
        new_inode.i_blocks = 1;

```

```
new_inode.i_block[0] = free_block; // 填写新申请的空闲数据块标号
new_inode.i_size = 2;
new_inode.i_links_count = 2; // '.' '..' 两个目录
```

```
// 写入 '.' '..'两个目录
```

```
struct ext2_dir_entry_2 cur_dir, parent_dir;
cur_dir.inode = new_inode_number; //当前目录 inode 节点号
cur_dir.rec_len = 256; //目录项所占空间
cur_dir.file_type = DIR_FILE;
strcpy(cur_dir.name, ".");
cur_dir.name_len = 1;
```

```
parent_dir.inode = cur_inode_number; // 当前目录的父目录节点号
parent_dir.rec_len = 256;
parent_dir.file_type = DIR_FILE;
strcpy(parent_dir.name, "..");
parent_dir.name_len = 2;
```

注意!: 在创建目录文件的时候新建两个目录‘.’ ‘..’ 这里主要的参数是 **inode**, ‘.’ 当前目录的 **inode** 需要填 **new_inode_number**, ‘..’ 父目录的 **inode** 需要填 **cur_inode_number**。

4.2.6 文件删除

删除目录或文件其实没有真正删除数据块中的内容，只是将数据块位图和 **inode** 位图中对应位置位 1，表示可用，后面如果有新申请的数据，覆盖原数据块即可；

删除文件的重点是通过目录或文件名找到对应 **inode** 和数据块编号，用来置位；

```
int j;
for(j=0; j < dir_inode.i_blocks; j++){ // 考虑到一个文件可能占用多个数据块
    set_block_bitmap(dir_inode.i_block[j], BLOCK_INDEX_NOT_USE);
}
```

4.2.7 文件列出

首先找到指定目录的 **inode**，然后通过 **inode** 找到数据块，目录文件的数据块中存放的是目录项；

找到指定目录所在数据项之后，由于目录项大小固定，每次将文件读写指针移动固定大小，读取目录项，通过目录项中 inode 和 type 来判断该目录项是目录还是普通文件。

```
int i;
for(i=0; i<cur_inode.i_links_count; i++){

    fseek(cur_disk,(FIRST_INODE_BLOCK+cur_inode.i_block[i/4])*EVERY_BLOCK+(1%4)*DIR_DENTRY_LENGTH, SEEK_SET);

    fread(&cur_dentry,DIR_DENTRY_LENGTH,1,cur_disk);

    printf("%s ",cur_dentry.name); // 输出目录项文件名

}
```

如果是普通文件，则输出文件名；

如果是目录文件，则在文件名后加上 ‘/’ 标志这是目录；

4.3 系统测试

测试用例 1：测试文件系统初始化功能

输入：系统输入 ./main

预期输出：

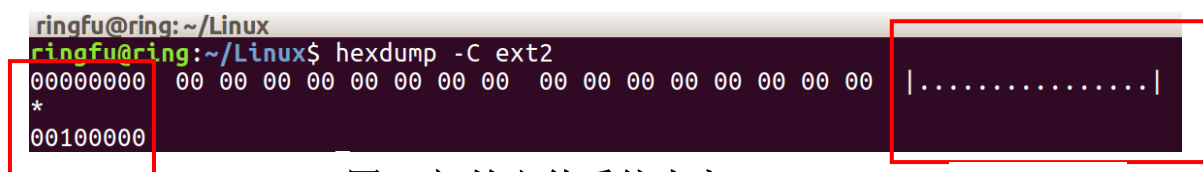
超级块初始化成功！

组块描述符表初始化成功！

数据块位图初始化成功！

索引节点表初始化成功！

i. 首先查看初始化前文件系统内容



```
ringfu@ring:~/Linux
ringfu@ring:~/Linux$ hexdump -C ext2
00000000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
*
00100000  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
```

大小为 1M

图 6 初始文件系统内容

内容为空

输出说明：00100000 表示当前文件大小为 1M， 后面全....表示文件内容为 0； 符合初始化功能设计。

ii. 控制台输出

```
ringfu@ring:~/Linux/fs-ext2$ ./main
disk open normally!
super block initialized!
超级块初始化成功!
GDT[1] initialized!
组块描述符表初始化成功!
block bitmap initialized!
数据块位图初始化成功!
inode bitmap initialized!
索引节点表初始化成功!
free inode searching...
inode search failed!
set inode bitmap!
free inode searching...
inode search failed!
set inode bitmap!
/ and root dit initialized!
根目录及root目录初始化成功!
[root@root-PC]:/ $
```

图 7 初始化输出

输出说明：出现此输出表示文件系统系统初始化成功

iii. 再次查看文件系统内容

```
ringfu@ring:~/Linux$ hexdump -C ext2
00000000  00 04 00 00 00 04 00 00 08 00 00 00 7c 03 00 00 |.....|...|
00000010  00 04 00 00 84 00 00 00 00 00 00 00 01 00 00 00 |.....|...|
00000020  00 04 00 00 00 00 00 00 00 00 00 00 01 00 00 00 |.....|...|
00000030  01 00 00 00 01 00 01 00 01 00 01 00 01 00 01 00 |.....|...|
00000040  01 00 00 00 e8 03 00 00 01 00 00 00 01 00 01 00 |.....|...|
00000050  01 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 |.....|...|
00000060  00 00 00 00 01 00 00 00 00 00 00 00 00 00 00 00 |.....|...|
00000070  00 00 00 00 01 00 00 00 01 01 00 00 01 00 00 00 |.....|...|
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|...|
*
000003b0  2f 68 6f 6d 65 00 00 00 00 00 00 00 00 00 00 00 |/home.....|
000003c0  2f 68 6f 6d 65 2f 72 69 6e 67 66 75 2f 4c 69 6e |/home/ringfu/Lin|
000003d0  75 78 2f 65 78 74 32 00 00 00 00 00 00 00 00 00 |ux/ext2.....|
000003e0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|...|
*
00000420  03 00 00 00 04 00 00 00 00 00 00 00 7c 03 00 04 |.....|...|
00000430  40 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|...|
00000440  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|...|
*
00000800  ff ff ff ff ff ff ff ff ff ff ff ff ff ff ff |.....|...|
```

超级块中卷名

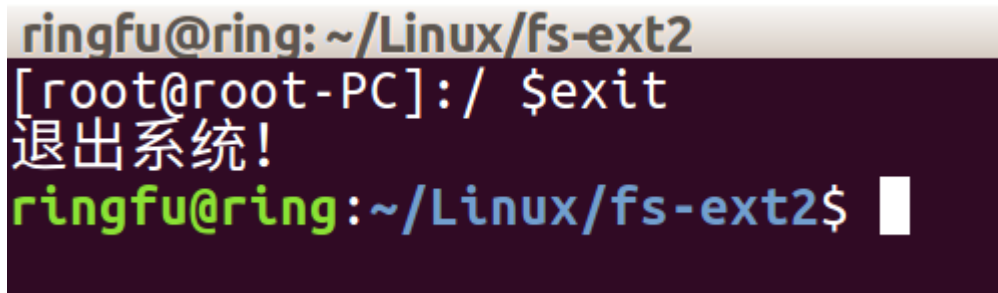
图 8 初始化后文件系统内容

输出说明：查看文件系统内容可以看见 ‘/home’ ‘/home/ringfu/Linux/ext2’ 参数即表明初始化内容成功写入磁盘。

测试用例 2：退出文件系统功能

输入：exit

预期输出：退出系统！



```
ringfu@ring: ~/Linux/fs-ext2
[root@root-PC]:/ $exit
退出系统!
ringfu@ring:~/Linux/fs-ext2$
```

图 9 exit 功能

输出说明：从仿真文件系统退出到真是文件系统，说明在仿真文件系统中退出功能可用。

5 结论与展望

5.1 结论

目前实现的功能有：

- i. 文件系统数据结构设计
- ii. 文件系统初始化；根目录和 root 目录初始化
- iii. 文件系统系统工作框架
- iv. 创建文件功能，删除文件功能，列出文件功能，退出系统
- v. 内存和磁盘间数据读写操作

存在的问题：

- i. set_inode_bitmap 对索引位图置位操作中每次只对一位（1b）操作，但是 C 中没有对一位操作的方法，只有通过循环移位和与或操作来置位，但 C 中又没有循环移位函数，所以这里有点逻辑还没理清楚；函数实现得到的效果和预期效果有差距；
- ii. 写文件的方法还没实现，打开文件较容易实现，但是写文件，怎么写，写的数据怎么输入，怎么在原有文件后面接着写，这些问题还有待考虑；
- iii. 用户打开文件表，系统打开文件表等这些多用户情况下的问题没时间考虑；

5.2 展望

- i. 文件操作其他几个函数在有充足时间情况下还要完善下；
- ii. 系统中存在的一点逻辑问题还要好好想想，对文件系统整体理解没有问题，但某些方面还是有点小问题；
- iii. 对于写文件这里，具体的实现方法还有待研究，为了更加深入了解操作系统，后面还要多花时间研究下，以后会发挥大作用；