1. 实验目的

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。

在此基础上设计并实现一个简单的文件系统。

2. 实验环境

- 编程环境：VS Code IDE
- 系统运行环境：Ubuntu 虚拟机

3. 实验内容

3.1 实验任务

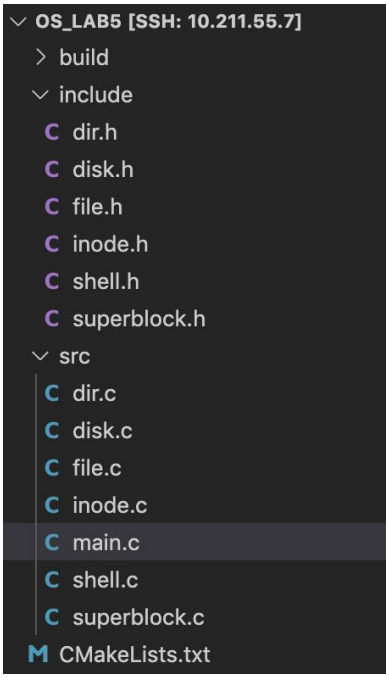
实现青春版 Ext2 文件系统，并提供简单的 shell 命令展示文件系统的基本功能。

- 创建文件，touch file
- 创建文件夹，mkdir directory
- 复制文件， cp file1 file2
- 关闭系统，shutdown
- 展示读取文件夹内容，ls
- 在系统关闭后，再次进入文件系统时可还原上次的文件部署。
- 系统会判断用户输入的命令和参数是否正确，并给出提示。
- 详细使用的用户手册将在 3.2 部分给出

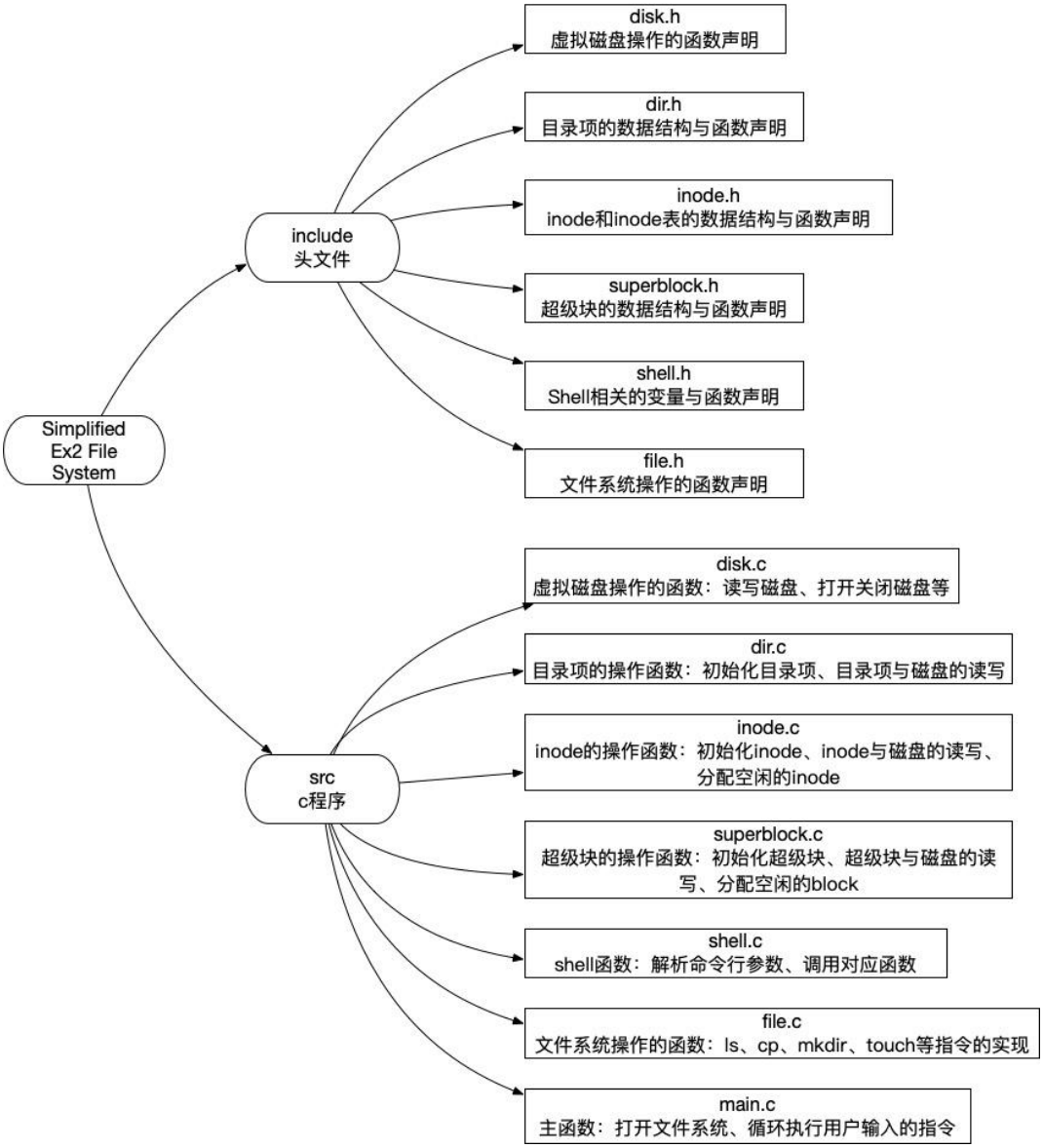
3.2 实验过程

3. 2. 1 系统架构分析

- 代码结构



- **系统架构图**



3. 2. 2 函数分析

对于 **disk.c、main.c** 中的函数不做具体分析，前者由课程提供，后者是简单的系统运行函数，将下一部分的运行中明白其逻辑。

对于 **superblock.c、dir.c、inode.c、shell.c** 中的函数较为简单，不具体分析其实现，只写明函数功能和注意点。

对于 **file.c** 中的函数，是该文件系统的核心函数，将分析其具体实现逻辑。

- **superblock**
  - 数据结构

```
/**
 * @brief 超级块
 *
 */
typedef struct super_block {
    int32_t magic_num;                  // 幻数
    int32_t free_block_count;           // 空闲数据块数
    int32_t free_inode_count;           // 空闲inode数
    int32_t dir_inode_count;            // 目录inode数
    uint32_t block_map[MAX_BLOCK_MAP];            // 数据块占用位图
    uint32_t inode_map[MAX_INODE_MAP];             // inode占用位图
}super_block;  // 656字节; 1block
```

- **void init_sp_block();**

```
/**
 * @brief 初始化超级块（文件系统第一次打开时）
 *
 */
void init_sp_block();
```

- **int write_sp_block();**

```
/**
 * @brief 将超级块写入磁盘（前两个磁盘块）
 *
 * @return int 成功返回1，否则返回0
 */
int write_sp_block();
```

注意超级块的大小是 1024 字节而一个磁盘块是 512 字节，所以读写超级块时应该是写入前两个磁盘块。

- **int read_sp_block();**

```
/**
 * @brief 从磁盘中读取超级块（前两个磁盘块）
 *
 * @return int 成功返回1，否则返回0
 */
int read_sp_block();
```

- **int alloc_block();**

```
/**
 * @brief 分配数据块
 *
 * @return int 成功返回数据组中数据块下标，失败返回-1
 */
int alloc_block();
```

注意分配空闲块时对超级块中 blockmap 的设置。

➢ **dir**

- **数据结构**

```
typedef struct dir_item {        // 目录项一个更常见的叫法是 dirent(directory entry)
    uint32_t inode_id;           // 当前目录项表示的文件/目录的对应inode
    uint16_t valid;              // 当前目录项是否有效
    uint8_t type;                // 当前目录项类型（文件/目录）
    char name[MAX_NAME_SIZE];              // 目录项表示的文件/目录的文件名/目录名
}dir_item;  // 128字节
```

- **int init_root_dir_item();**

```
/**
 * @brief 初始化根目录对应的block，即数据组的block 1
 *
 * @return int 成功返回1，否则返回0
 */
int init_root_dir_item();
```

初始化根目录的目录项时，要将根目录对应的数据块中的目录项全部初始化!

- **int write_dir_item(int index);**

```
/**
 * @brief 将block_buffer写到磁盘中
 *
 * @param index 文件系统的逻辑数据块下标（该函数需要将其转换成对应的磁盘块）
 * @return int 成功返回1，否则返回0
 */
int write_dir_item(int index);
```

关于目录项的读取：在系统中设置 block_buffer 作为数据块缓冲区，大小为

1024 字节，是长度为 8 的目录项数组（因为目录项的大小为 128 字节，所以一个数据块最多可以放 8 个目录项。且在该系统中，不需要对文件内容进行操作，所以只有读写目录项时才需要对数据块进行操作。）

注意，一个数据块对应两个磁盘块，所以这里的读取需要对两个磁盘块进行操作。

- int read_dir_item(int index);

```
/**
 * @brief 将block index读到block_buffer中
 *
 * @param index 文件系统的逻辑数据块下标（该函数需要将其换成对应的磁盘块）
 * @return int 成功返回1，否则返回0
 */
int read_dir_item(int index);
```

- inode
  - 数据结构

```
typedef struct inode {
    uint32_t size;              // 文件大小
    uint16_t file_type;         // 文件类型（文件/文件夹）
    uint16_t link;              // 连接数
    uint32_t block_point[6];    // 数据块指针
}inode; // 4+2+2+4*6=32字节
```

  - int write_inode(struct inode* node, int index);

```
/**
 * @brief 将inode写入磁盘块中
 *
 * @param node 指向inode的指针
 * @param index 该inode在inode_table中的下标
 * @return int 成功返回1，否则返回0
 */
int write_inode(struct inode* node, int index);
```

这里的 inode 与磁盘的读写函数，是只对单一的 inode 进行修改，而不是整个 inode table 的读入。所以在进行写入的时候，需要先从磁盘中读出数据到 buf 中，再修改 buf，再写入磁盘。

  - int read_inode(struct inode* node, int index);

```
/**
 * @brief 读取磁盘块中的inode
 *
 * @param node 指向读取得到的inode的指针
 * @param index 该inode在inode_table中的下标
 * @return int 成功返回1，否则返回0
 */

int read_inode(struct inode* node, int index);
```

  - int init_root_inode();

```
/**
 * @brief 初始化第一个inode，对应根目录
 *
 * @return int 成功返回1，否则返回0
 */
int init_root_inode();
```

初始化根目录对应的 inode 的时候，需要注意的是：第 0 个数据块分给了超级块，第 1 ~ 32 数据块分给了 inode table，所以根目录的数据块指针指向的是 33。

- int init_inode(struct inode* node, int size, int type, int link);

```
/**
 * @brief 初始化文件的inode
 *
 * @param node inode对应的指针，一般传入inode table[i]
 * @param size inode属性size
 * @param type inode属性type
 * @param link inode属性link
 * @return int 成功返回1，否则返回0
 */
int init_inode(struct inode* node, int size, int type, int link);
```

- int alloc_inode();

```
/**
 * @brief 分配一个空闲的inode
 *
 * @return int 分配成功，则返回inode下标，失败则返回-1
 */
int alloc_inode();
```

注意在分配空闲 inode 的时候对超级块 inodemap 的设置。

- shell
  - int get_command();

```
/**
 * @brief 通过空格分割命令行，得到指令关键字和参数
 *
 */
void get_command();
```

  - int run_command();

```
/**
 * @brief 执行命令
 *
 */
void run_command();
```

在这个函数中需要对用户输入的参数进行判断并进行反馈。

例如 touch 命令和 mkdir 命令只能有 1 个参数；ls 命令可以有 0 个或 1 个参数；cp 命令只能有两个参数；shutdown 命令不能有参数（以上的参数不包括命令名称）。

- file
  - int open_system()

程序运行时，首先执行该函数来打开文件系统。注意点是打开文件系统的时候，要先判断磁盘是否有打开。打开磁盘后读取超级块到内存中判断幻数是否正确，若正确，读取磁盘中的 inode 表到内存中；若错误，需要执行 init_system 函数来初始化文件系统。

```
int open_system()
{
    // 每次打开文件系统时，要先打开磁盘
    if (open_disk()<0) {
        printf("Open disk failed.\n");
        return 0;
    }
    // 打开磁盘后
    if(read_sp_block() && spb.magic_num == MAGIC_NUM)
    {// 如果幻数正确，说明该磁盘已经建立过文件系统
        // 将磁盘中的inode表读到内存的inode_table中
        for (int i = 0; i < INODE_NUM; i++)
        {
            if (!read_inode(&inode_table[i], i))
            {
                printf("Read inode failed.\n");
                return 0;
            }
        }
        printf("---------------- Welcome to Simlified File System! Please input instructions.------------\n");
        print_file_system_info();
    }else
    {// 如果幻数不正确，说明该磁盘还没有建立过文件系统或者是文件系统损坏
        printf("No correct file system. We are building a new file system...\n");
        init_system();
    }
    return 1;
}
```

- **int init_system()**

```
/**
 * @brief 初始化文件系统
 *
 * @return int
 */
int init_system();
```

初始化超级块并写入磁盘。初始化 inode 表并写入磁盘。初始化根目录并写入磁盘。之后用户就可以通过命令行来使用该文件系统。

- **void touch(char *path)**

① 调用 get_the_last_but_one_touch()函数来获得路径中的最后一个文件夹的 inode id 以及要创建文件的名字 tmp。

② 判断路径是否错误；文件的名字是否符合要求；文件夹下是否存在同名文件。若均符合要求，转 3

③ 调用 alloc_inode()函数来为新文件分配一个 inode，若存在空闲的 inode，转 4

④ 调用 insert_dir_item()函数对文件夹的数据块中，插入一个目录项。若可以再插入目录项，转 5。

注意：一个文件夹最多可以有 6 个数据块，每个数据块可以有 8 个目录项，所以一个文件夹下最多可以用 48 个文件（夹）。

⑤ 初始化新文件的 inode 并写入磁盘。

⑥ 调用 print_file_system_info()

```c
void touch(char *path)
{
    char tmp[MAX_NAME_SIZE];
    // 获得了路径中最后一个文件夹的inode_id
    int i_id = get_the_last_but_one_touch(path, tmp);
    // printf("i_id: %d\n", i_id);
    // printf("tmp: %s\n", tmp);
    // 路径存在各种错误
    if(i_id < 0)
    {
        return;
    }
    int i, j;
    if(tmp[0] == '\0')
    {
        printf("Wrong path!\n");
        return;
    }
    if(check_name(tmp))
    {
        return;
    }
    // 路径没有错误
    if (!check_duplicate_name(tmp, i_id, _FILE_))
    {// 如果有同名文件
        printf("There is already a file named \"%s\" in this path !\n", tmp);
        return;
    }
    int new_inode = alloc_inode();
    // printf("new_inode: %d\n", new_inode);
    if (new_inode < 0)
    {
        printf("No empty inode!\n");
        return;
    }
    if (!insert_dir_item(i_id, new_inode, tmp, _FILE_))
    {
        printf("Touch file failed. \n");
        return;
    }
    init_inode(&inode_table[new_inode], 0 , _FILE_, 1);
    write_inode(&inode_table[new_inode], new_inode);
    print_file_system_info();
    return;
}
```

■ **void mkdir(char *path)**

① 调用 get_the_last_but_one_mkdir()函数来获得路径中的倒数第二个文件夹的 inode id 以及要创建文件夹的名字 tmp。

② 判断路径是否错误；文件的名字是否符合要求；文件夹下是否存在同名文件。若均符合要求，转 3

③ 调用 alloc_inode()函数来为新文件夹分配一个 inode，若存在空闲的 inode，转 4

④ 调用 insert_dir_item()函数对 inode id 对应的文件夹的数据块中，插入一个目录项。若可以再插入目录项，转 5。

注意：一个文件夹最多可以有 6 个数据块，每个数据块可以有 8 个目录项，所以一个文件夹下最多可以用 48 个文件（夹）。

⑤ 初始化新文件的 inode 并写入磁盘。
⑥ 增加了目录节点的树木，更新超级块，写入磁盘。
⑦ 调用 print_file_system_info()

```
void mkdir(char *path)
{
    char tmp[MAX_NAME_SIZE];
    int i, j;
    // 获得了路径中倒数第二个文件夹的inode_id
    int i_id = get_the_last_but_one_mkdir(path, tmp);
    // 路径存在各种错误
    if(i_id < 0)
    {
        return;
    }
    if(tmp[0] == '\0')
    {
        printf("Wrong path!\n");
        return;
    }
    // 文件/文件夹命名错误
    if(check_name(tmp))
    {
        return;
    }
    // 检查是否有同名文件
    if (!check_duplicate_name(tmp, i_id, _FOLDER_))
    {// 如果有同名文件
        printf("There is already a folder named \"%s\" in this path !\n", tmp);
        return;
    }
    int new_inode = alloc_inode();
    if (new_inode < 0)
    {
        printf("No empty inode!\n");
        return;
    }
    // 以下开始和touch有一些不一样
    if (!insert_dir_item(i_id, new_inode, tmp, _FOLDER_))
    {
        printf("Mkdir failed. \n");
        return;
    }
    init_inode(&inode_table[new_inode], 0 , _FOLDER_, 1);
    write_inode(&inode_table[new_inode], new_inode);
    // 注意：超级块的目录inode数量要更新
    spb.dir_inode_count++;
    write_sp_block();
    print_file_system_info();
}
```

■ void ls(char *path)

① 调用 get_the_last_dir()函数来获得路径中的最后一个文件夹的 inode id 以及该文件夹的名字 tmp。
② 遍历该文件夹的 inode 信息中的六个数据块指针
③ 若对应数据块非零，说明该文件夹有被分配数据块，则读取数据块到内存 block_buffer 中。
④ 遍历该数据块中的 8 个目录项。若目录项有效则输出。

```
void ls(char *path)
{
    // printf("path: %s\n", path);
    char tmp[MAX_NAME_SIZE];
    int i_id;
    // i_id为路径中最后一个文件夹的inode id, tmp为该文件夹的名称
    i_id = get_the_last_dir(path, tmp);
    // 遍历该文件夹的6个数据块
    for (int q = 0; q < 6; q++)
    {
        int block_num = inode_table[i_id].block_point[q];
        printf("i_id: %d\n", i_id);
        printf("block_num: %d\n", block_num);
        if (block_num)
        {// 如果该文件夹有被分配数据块
            read_dir_item(block_num);  // 读取该数据块到数据块缓存区block buffer中
            // 遍历数据块的8个目录项
            for (int p = 0; p < 8; p++)
            {
                // printf("block_buffer[p].type: %d\n", block_buffer[p].type);
                // printf("block_buffer[p].valid: %d\n", block_buffer[p].valid);
                if (block_buffer[p].type == _FOLDER_ && block_buffer[p].valid)
                {// 如果该目录项为文件夹且有效
                    printf("*");
                    printf("%s\n", block_buffer[p].name);
                }
                if (block_buffer[p].type == _FILE_ && block_buffer[p].valid)
                {// 如果该目录项为文件且有效
                    printf("%s\n", block_buffer[p].name);
                }
            }
        }

    }
    return;
}
```

- **void cp(char *ori, char *dest)**

该函数分为两个部分，对源文件路径的处理以及对目标文件夹路径的处理。

① 调用 get_the_last_file()函数获得源文件对应的 inode 的 ori_id 和源文件的名字 ori_name。

② 判断路径和名字是否有错。若没错，转 3

③ 保存源文件对应的 inode 的 size 和 link

④ 将源文件的名字 ori_name 复制给目标文件 dest_name;

⑤ 调用 get_the_last_dir()函数活度目标文件夹路径的 inode 的 tmp_id 以及该文件夹的名字 tmp_folder

⑥ 判断路径和名字是否有错；检查该目录下是否有 dest_name 的同名文件。若没错，转 7

⑦ 调用 alloc_inode()函数来为新文件分配一个 inode, 若存在空闲的 inode, 转 8

⑧ 调用 insert_dir_item()函数对 tmp_id 对应的文件夹的数据块中, 插入一个目录项。若可以再插入目录项，转 9

⑨ 初始化新文件的 inode 并写入磁盘。注意这里的 link 和 size 应该跟之前保存的源文件的值相同。

⑩ 调用 print_file_system_info()

```
    char tmp_folder[MAX_NAME_SIZE];
    // tmp_id为目标文件所在文件夹的id，tmp_folder为该文件夹的名字
    int tmp_id = get_the_last_dir(dest, tmp_folder);
    // 路径存在各种错误
    if(tmp_id < 0)
    {
        return;
    }
    if(tmp_folder[0] == '\0')
    {
        printf("Wrong dest path!\n");
        return;
    }
    // 路径没有错误，检查是否有同名文件
    if (!check_duplicate_name(dest_name, tmp_id, _FILE_))
    {// 如果有同名文件
        printf("There is already a file named \"%s\" in this path !\n", dest_name);
        return;
    }
    int new_inode = alloc_inode();
    // printf("new_inode: %d\n", new_inode);
    if (new_inode < 0)
    {
        printf("No empty inode!\n");
        return;
    }
    if (!insert_dir_item(tmp_id, new_inode, dest_name, _FILE_))
    {
        printf("Touch file failed. \n");
        return;
    }
    init_inode(&inode_table[new_inode], size , _FILE_, link);  // 注意这里的size和link与源文件相同
    write_inode(&inode_table[new_inode], new_inode);
    print_file_system_info();
    return;
}
```

```
void cp(char *ori, char *dest)
{
    // 对源文件路径ori的处理
    char ori_name[MAX_NAME_SIZE];
    // ori_id为源文件对应的inode，ori_name为源文件的名字
    int ori_id = get_the_last_file(ori, ori_name);
    // 路径存在各种错误
    if(ori_id < 0)
    {
        printf("Wrong ori file path!\n");
        return;
    }
    if(ori_name[0] == '\0')
    {
        printf("Wrong ori file path!\n");
        return;
    }
    int size = inode_table[ori_id].size;
    int link = inode_table[ori_id].link;

    // 对目标文件路径dest的处理
    char dest_name[MAX_NAME_SIZE];
    memcpy(dest_name, ori_name, MAX_NAME_SIZE);
    char tmp_folder[MAX_NAME_SIZE];
    // tmp_id为目标文件所在文件夹的id，tmp_folder为该文件夹的名字
    int tmp_id = get_the_last_dir(dest, tmp_folder);
    // 路径存在各种错误
    if(tmp_id < 0)
    {
        return;
    }
    if(tmp_folder[0] == '\0')
    {
        printf("Wrong dest path!\n");
        return;
    }
```

- **void shutdown()和 close_system()**

① 调用 close_system()函数来关闭系统。

② close_system()函数的主要工作是：将超级块、inode table 写入磁盘，关闭磁盘。

- **int insert_dir_item(int i_id, int new_inode, char *tmp, int type)**

① 遍历 i_id 对应的 inode 中的数据块指针

② 若文件夹没有对应的数据块，转 3；若有数据块，转 6

③ 调用 alloc_block()分配一个空闲的数据块，然后设置 inode 中的数据块指针指向该数据块。将该数据块读到内存 block_buffer 中。

④ 初始化该数据块：将该数据块下的目录项的 valid 字段均初始化为 0。然后执行插入目录项的操作：将第一个目录项的 valid 字段设置为 1，type 和 inode_id 设置为传进来的参数。然后将该数据块写回磁盘。

⑤ 将 i_id 对应的 inode 中的 size 加一，将该 inode 写入磁盘。

```
int insert_dir_item(int i_id, int new_inode, char *tmp, int type)
{
    for (int i=0; i<6; i++)
    {
        if (!inode_table[i_id].block_point[i])
        {// 如果文件夹没有对应数据块
            int block_num = alloc_block();  // 分配一个数据块
            // printf("block_num: %d\n", block_num);
            if (block_num < 0)
            {
                printf("No empty block!\n");
                return 0;
            }
            inode_table[i_id].block_point[i] = block_num;  // 数据组的block index

            // 将block index读到block_buffer中
            read_dir_item(inode_table[i_id].block_point[i]);
            // 初始化dir item
            for (int j = 0; j < 8; j++)
            {
                block_buffer[j].valid = 0;
            }
            block_buffer[0].valid = 1;
            block_buffer[0].type = type;
            block_buffer[0].inode_id = new_inode;
            memcpy(block_buffer[0].name, tmp, MAX_NAME_SIZE);
            write_dir_item(inode_table[i_id].block_point[i]);

            inode_table[i_id].size++;
            if(!write_inode(&inode_table[i_id], i_id)){
                printf("write inode failed!\n");
            };
            return 1;
        }
```

⑥ 将对应的数据块读到内存 block_buffer 中。然后遍历该数据块的目录项。若有目录项是无效的，即 valid 字段为 0。则将新的目录项插入到该位置。valid 字段设置为 1，type 和 inode_id 设置为传进来的参数。然后将该数据块写回磁盘。

⑦ 将 i_id 对应的 inode 中的 size 加一，将该 inode 写入磁盘。

```
    // 如果文件夹有对应数据块
    // 将block index读到block_buffer中
    read_dir_item(inode_table[i_id].block_point[i]);
    // printf("block_num: %d\n", inode_table[i_id].block_point[i]);
    // 遍历该block_buffer中的dir_item
    for(int j=0; j<8; j++)
    {
        if(!block_buffer[j].valid)
        {
            // printf("block_buffer j: %d\n", j);
            block_buffer[j].valid = 1;
            block_buffer[j].type = type;
            block_buffer[j].inode_id = new_inode;
            memcpy(block_buffer[j].name, tmp, MAX_NAME_SIZE);
            write_dir_item(inode_table[i_id].block_point[i]);

            inode_table[i_id].size++;
            // printf("inode_table[i_id].size: %d\n", inode_table[i_id].size);
            write_inode(&inode_table[i_id], i_id);
            return 1;
        }
    }
}
// 该文件夹下的文件已满
printf("The number of files and dirs in this dir has met the maxmuim\n");
return 0;
```

3. 2. 3 系统运行

- 运行环境
  1. 在 Linux 上安装编译器 gcc 和调试器 gdb。
  2. 选择命令行编译方式，编写 CMake 配置文件 CMakeLists.txt 如下

```
# CMake 最低版本号要求
cmake_minimum_required(VERSION 3.0.0)
# 项目信息
project(net VERSION 0.1.0)

# 添加include目录存放.h文件
include_directories("include")
# 添加库
add_library(alllibs src/disk.c src/file.c src/dir.c src/inode.c src/shell.c src/superblock.c)
# 指定生成目标
add_executable(main src/main.c)
# 添加链接库
target_link_libraries(main alllibs)
```

  3. 在 vs code 中使用 ssh 连接 ubuntu 虚拟机，程序运行在 linux 环境下
- 运行步骤
  1. 打开终端，输入 mkdir build。
  2. 输入 cd build。
  3. 输入 cmake ..。
  4. 输入 make
  5. 输入./main，进入程序，打开文件系统，界面如下

```
root@parallels-Parallels-Virtual-Platform:/home/parallels/Desktop/os_lab5# cd build
root@parallels-Parallels-Virtual-Platform:/home/parallels/Desktop/os_lab5/build# cmake ..
-- Configuring done
-- Generating done
-- Build files have been written to: /home/parallels/Desktop/os_lab5/build
root@parallels-Parallels-Virtual-Platform:/home/parallels/Desktop/os_lab5/build# make
[ 77%] Built target alllibs
[100%] Built target main
root@parallels-Parallels-Virtual-Platform:/home/parallels/Desktop/os_lab5/build# ./main
No correct file system. We are building a new file system...
Init file system sueccessfully!
---------- Welcome to Simplified File System! Please input instructions.----------
--------------------------------------------------------------------------------
In this Symplified File System :
Supports: \ls, \mkdir, \touch, \copy, \close
It has 1 folders and 0 files in this system now ;
--------------------------------------------------------------------------------
>>>>
```

- 用户手册与事例
  - ① 创建文件：**ls filepath**
    - ✧ **filepath** 为将要创建的文件路径
    - ✧ 创建成功后，用户可以看到输出的该文件系统信息中，**file** 数量增加了 **1**
    - ✧ 若参数数量不正确，会提示 **Arguments nums error!!!**
    - ✧ 文件只允许数字、字母、下划线、小数点，命名不合法会提示 **Illegal name!! (Only numbers ,letters ,_ and . can be accepted)**。
    - ✧ 路径不正确，会提示 **Wrong path!**
    - ✧ 没有空闲的 **inode** 或 **block** 会提示 **No empty inode/block**。
    - ✧ 文件夹下有同名文件会提示 **There is already a file named xxx in this path!**
    - ✧ 若当前文件夹下文件数太多会提示 **The number of files and dirs in this dir has met the maxmuim.**
    - ✧ 过程中出现任何错误导致创建文件失败都会输出 **Touch file failed.**

```
>>>> touch /hello.c
--------------------------------------------------------------------------------
In this Symplified File System :
Supports: \ls, \mkdir, \touch, \copy, \close
It has 1 folders and 1 files in this system now ;
--------------------------------------------------------------------------------
>>>> ls /
hello.c
>>>>
```

上图：在根目录下创建了 **hello.c** 文件。

  - ② 创建文件夹：**mkdir dirctorypath**
    - ✧ **filepath** 为将要创建的文件夹路径
    - ✧ 创建成功后，用户可以看到输出的该文件系统信息中，**folder** 数量增加了 **1**
    - ✧ 若参数数量不正确，会提示 **Arguments nums error!!!**
    - ✧ 文件只允许数字、字母、下划线、小数点，命名不合法会提示 **Illegal name!! (Only numbers ,letters ,_ and . can be accepted)**。
    - ✧ 路径不正确，会提示 **Wrong path!**
    - ✧ 没有空闲的 **inode** 或 **block** 会提示 **No empty inode/block**。
    - ✧ 文件夹下有同名文件夹会提示 **There is already a file named xxx in this path!**
    - ✧ 若当前文件夹下文件数太多会提示 **The number of files and dirs in this dir has met the maxmuim.**
    - ✧ 过程中出现任何错误导致创建文件失败都会输出 **Mkdir file failed.**

```
>>>> mkdir /home
--------------------------------------------------------------------------------
In this Symplified File System :
Supports: \ls, \mkdir, \touch, \copy, \close
It has 2 folders and 1 files in this system now ;
--------------------------------------------------------------------------------
>>>> ls
hello.c
*home
>>>>
```

上图：在根目录下创建了 **home** 文件夹。

③ **输出路径下的文件：ls dirctorypath**

- ✧ **filepath** 为将要查看的文件夹路径
- ✧ 程序会输出在该文件夹下的文件和文件夹名，若为文件夹，则在左上方会有星号。
- ✧ 若参数数量不正确，会提示 **Arguments nums error!!!**
- ✧ 路径不正确，会提示 **Wrong path!**以及 **There is no directory xxx.**
- ✧ 过程中出现任何错误导致创建文件失败都会输出 **Ls failed.**

```
>>>> mkdir /home/test
--------------------------------------------------------------------------------
In this Symplified File System :
Supports: \ls, \mkdir, \touch, \copy, \close
It has 3 folders and 1 files in this system now ;
--------------------------------------------------------------------------------
>>>> touch /home/world.c
--------------------------------------------------------------------------------
In this Symplified File System :
Supports: \ls, \mkdir, \touch, \copy, \close
It has 3 folders and 2 files in this system now ;
--------------------------------------------------------------------------------
>>>> ls /home
*test
world.c
>>>>
```

上图：在**/home** 下创建了 **test** 文件夹和 **world.c** 文件，用 **ls** 命令查看**/home** 下的文件和文件夹。可看到结果正确。

④ **复制文件：cp file1path dirctorypath**

- ✧ **file1path** 为将要被复制的文件路径，**dirctorypath** 为将要新建文件的文件夹路径。
- ✧ 创建成功后，用户可以看到输出的该文件系统信息中，**file** 数量增加了 **1**
- ✧ 若参数数量不正确，会提示 **Arguments nums error!!!**
- ✧ 文件只允许数字、字母、下划线、小数点，命名不合法会提示 **Illegal name!! (Only numbers ,letters ,_ and . can be accepted)**。
- ✧ 路径不正确，会提示 **Wrong ori/dest path!**
- ✧ 没有空闲的 **inode** 或 **block** 会提示 **No empty inode/block**。
- ✧ 文件夹下有同名文件夹会提示 **There is already a file named xxx in this path!**
- ✧ 若当前文件夹下文件数太多会提示 **The number of files and dirs in this dir has met the maxmuim.**

```
>>>> ls /home
*test
world.c
>>>> cp /hello.c /home
--------------------------------------------------------------------------------
In this Symplified File System :
Supports: \ls, \mkdir, \touch, \copy, \close
It has 3 folders and 3 files in this system now ;
--------------------------------------------------------------------------------
>>>> ls /home
*test
world.c
hello.c
>>>>
```

上图：/home 下原本只有 test 文件夹和 world.c 文件，使用 cp 命令将根目录下的 hello.c 文件复制到/home 中。然后使用 ls 命令查看/home 下的情况，可以看到多出了 hello.c 文件。

⑤ 关闭该文件系统：shutdown

- ✧ 该指令没有参数，输出后会关闭文件系统并且保存已执行的文件操作
- ✧ 若参数数量不正确，会提示 Arguments nums error!!!

```
>>>> ls /home
*test
world.c
hello.c
>>>> shutdown
Save superblock successfully.
Write inode successfully.
Close disk successfully.
Shut down the Symplified File System sucessfully!
---------------------------------- Thanks for your using :) ----------------------------
root@parallels-Parallels-Virtual-Platform:/home/parallels/Desktop/os_lab5/build# ./main
-------------- Welcome to Simplified File System! Please input instructions.------------
--------------------------------------------------------------------------------------
In this Symplified File System :
Supports: \ls, \mkdir, \touch, \copy, \close
It has 3 folders and 3 files in this system now ;
--------------------------------------------------------------------------------------
>>>> ls /home
*test
world.c
hello.c
>>>>
```

上图：使用 ls 命令查看/home 下的情况。然后使用 shutdown 命令关闭系统。再次打开系统可以看到/home 下的文件和文件夹仍存在。