FsLab实验报告

姓名：梅祎航 学号：2022201459

本次实验旨在完成一个VSFS，我通过仔细的思考，设计了一套VSFS的体系。能保证完成所有的功能要求，并通过了所有的16个测试点。同时为一些函数加上了互斥锁，以防止多线程的错误操作。

整体结构

```
| Super Block       (1 blocks)     |
|----------------------------------|
| Data Block Bitmap (2 blocks)     |
|----------------------------------|
| Inode Bitmap      (1 block)      |
|----------------------------------|
| Inode Table       (512 blocks)   |
|----------------------------------|
| Data Blocks       (65276 blocks)|
```

如上图，整体的结构设计如上，接下来我会一一讲解每个部分以及相关的辅助函数

输入输出

由于disk_read和disk_write没有报错功能，所以我对这两个函数在外面包装了一层报错，定义为safe_read和safe_write，之后的读入和写入都用我包装后的函数~~（虽然直到写完了这个报错也没派上用场）~~

```
int safe_read(int block_id, void* buffer) {
    if (disk_read(block_id, buffer) != 0) {
        fprintf(stderr, "Error reading block %d\n", block_id);
        return -1;
    }
    return 0;
}
//safe_write差不多，我就不放上来了
```

锁

添加了三种锁，分别锁住对bitmap，inode和目录的操作

```
pthread_mutex_t bitmap_mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t inode_mutex = PTHREAD_MUTEX_INITIALIZER;
```

```
pthread_mutex_t dir_mutex = PTHREAD_MUTEX_INITIALIZER;

void init_locks() {
    pthread_mutex_init(&bitmap_mutex, NULL);
    pthread_mutex_init(&inode_mutex, NULL);
    pthread_mutex_init(&dir_mutex, NULL);
}

void destroy_locks() {
    pthread_mutex_destroy(&bitmap_mutex);
    pthread_mutex_destroy(&inode_mutex);
    pthread_mutex_destroy(&dir_mutex);
}
```

Super Block

Super Block的设计

对于Super Block，我的想法是存放根目录的编号以及函数fs_statfs所需要的信息，因此有以下的结构体：

```
struct superblock {
    unsigned long  f_bsize;  // 块大小
    uint32_t       f_blocks; // 块数量
    uint32_t       f_bfree;  // 空闲块数量
    uint32_t       f_bavail; // 可用块数量
    uint32_t       f_files;  // 文件节点数
    uint32_t       f_ffree;  // 空闲节点数
    uint32_t       f_favail; // 可用节点数
    unsigned long  f_namemax; // 文件名长度上限
    uint32_t       root_inode; // 根目录的 i 节点编号
};
```

辅助函数

有初始化函数

```
void init_superblock() {
    struct superblock sb;
    sb.f_bsize = BLOCK_SIZE;
    sb.f_blocks = BLOCK_NUM;
    sb.f_bfree = BLOCK_NUM - (1 + 2 + 1 + INODE_BLOCKS); // 初始化时空闲块
数
    sb.f_bavail = sb.f_bfree;
    sb.f_files = MAX_INODE; // 总文件节点数
    sb.f_ffree = sb.f_files; // 初始化时空闲节点数
    sb.f_favail = sb.f_ffree;
    sb.f_namemax = NAME_MAX;
    sb.root_inode = ROOT_INODE; // 根目录的 i 节点编号
    safe_write(0,&sb);
}
```

最初我还定义了四个辅助函数data_plus(),data_minus(),inode_plus(),inode_minus()用来实现f_bavail，f_bfree，f_ffree，f_favail的增减，但后来发现还不如在fs_statfs函数被调用时直接统计Bitmap的0的个数，故而放弃了上面四个函数。。。

BitMap

BitMap的设计

由于有65536个block，所以Data Block Bitmap有65536个Bit，也就是65536/8/4096 = 2Block

由于需要容纳32768个inode，所以Inode Bitmap有32768个Bit，也就是32768/8/4096=1Block

故而将Block1和Block2作为Data Block Bitmap，将Block3作为Inode Bitmap。

辅助函数

对于Bitmap，主要有三种辅助函数：

- 第一种，对Bitmap直接操作的,以下三个函数分别是对指定的Bit进行赋1，赋0，检验是否为1：

```
void set_bit(uint8_t *bitmap, int index) {
    bitmap[index / 8] |= (1 << (index % 8));
}

void clear_bit(uint8_t *bitmap, int index) {
    bitmap[index / 8] &= ~(1 << (index % 8));
}

int test_bit(uint8_t *bitmap, int index) {
    return bitmap[index / 8] & (1 << (index % 8));
}
```

- 第二种，在Bitmap上分配或释放一个Bit

```
// BITMAP上分配1个新的数据块
int allocate_block_in_bitmap(){
    pthread_mutex_lock(&bitmap_mutex);
    uint8_t data_block_bitmap[BLOCK_SIZE * 2];
    safe_read(1, data_block_bitmap);
    safe_read(2, data_block_bitmap + BLOCK_SIZE);
    for (int i = 0; i < BLOCK_NUM; ++i) {
        if (!test_bit(data_block_bitmap, i)) {
            set_bit(data_block_bitmap, i);
            safe_write(1,data_block_bitmap);
            safe_write(2, data_block_bitmap + BLOCK_SIZE);
            pthread_mutex_unlock(&bitmap_mutex);
            return i;
        }
    }
    pthread_mutex_unlock(&bitmap_mutex);
    return -1;
}
// BITMAP上释放1个数据块
void free_block_in_bitmap(int block_id) {
    pthread_mutex_lock(&bitmap_mutex);
    uint8_t data_block_bitmap[BLOCK_SIZE * 2];
    safe_read(1, data_block_bitmap);
    safe_read(2, data_block_bitmap + BLOCK_SIZE);
    clear_bit(data_block_bitmap, block_id);
    safe_write(1,data_block_bitmap);
    safe_write(2, data_block_bitmap + BLOCK_SIZE);
    pthread_mutex_unlock(&bitmap_mutex);
}
// BITMAP上分配新的1个inode
int get_new_inode_bit_in_bitmap(){
    pthread_mutex_lock(&bitmap_mutex);
    uint8_t inode_bitmap[BLOCK_SIZE];
    safe_read(3, inode_bitmap);
    for (int i = 0; i < MAX_INODE; ++i) {
        if (!test_bit(inode_bitmap, i)) {
            set_bit(inode_bitmap, i);
            safe_write(3,inode_bitmap);
            pthread_mutex_unlock(&bitmap_mutex);
            return i;
        }
    }
    pthread_mutex_unlock(&bitmap_mutex);
    return -1; // 没有空闲 inode
}
// BITMAP上释放1个inode
void free_inode_in_bitmap(int inode_id) {
```

```
    pthread_mutex_lock(&bitmap_mutex);
    uint8_t inode_bitmap[BLOCK_SIZE];
    safe_read(3, inode_bitmap);
    clear_bit(inode_bitmap, inode_id);
    safe_write(3,inode_bitmap);
    pthread_mutex_unlock(&bitmap_mutex);
}
```

- 第三种，专门供函数fs_statfs用的，用来统计多少Bit是空闲的

```
//bitmap上数有多少1
int count_free_bits(uint8_t *bitmap, int size) {
    int count = 0;
    for (int i = 0; i < size * 8; ++i) {
        if (!test_bit(bitmap, i)) {
            count++;
        }
    }
    return count;
}
```

Inode

inode的设计

Inode的设计是比较重要的，特别是指针的设计。

注意到每个Block都可以用0~65535的数字表示，所以在本次实验中，可以用一个2Byte的数字做“指针”。

因此，一个indirect_pointer指向的Block可以存放4096/2 = 2048个指针，而这2048个Block可以存放2048*4096B = 8MB，正好符合最多8MB的空间要求，所以Inode中只需要一个indirect_pointer即可，不需要pointer或double_pointer。有以下示意图：

```
Inode
    |---indirect_pointer
                    |---Blocki
                            |--Blocki1
                            |--Blocki2
                            |...
                            |--Blocki2048
```

由上述论述设计的Inode如下，按照内存从小到大声明，使得对齐浪费的字节只有1Byte，总共32Byte：

```
struct inode {
    uint8_t is_file;              // 是不是普通文件                    // 1
byte
    uint16_t indirect_pointer;    // 一级间接块编号                    // 2
bytes
    uint32_t size;                // 文件大小（字节）                  // 4
bytes
    time_t st_Atime;              // 被访问的时间                      // 8
bytes
    time_t st_Mtime;              // 被修改的时间                      // 8
bytes
    time_t st_Ctime;              // 状态改变时间                      // 8
bytes
};//共32 byte
```

所以一个Block能放下4096/32 = 128个Inode，要放下32768需要32768/128=256个Block。因此将Block4~Block259作为Inode区。

目录的设计

对于目录文件，其indirect_pointer与普通文件不同。

首先我设计了如下结构体作为目录的条目。这样设计既能巧妙地对齐32字节，又使得name后面的字节一定为0，杜绝了name字符串末尾不为0的问题（这个问题不这样解决的话十分麻烦）。：

```
struct dir_entry {
    char name[NAME_MAX];
    uint8_t padding[2];      // 填充字节使其凑齐 32 字节
    uint8_t is_used;
    uint32_t inode_index;
};
```

这样的话，一个Block能存放4096/32=128个entry，根据以下的示意图，一个目录文件能记录2048*128 = 262144个文件，其总量超过了Inode的总数32768，能记录下所有的Inode。

```
Inode
    |---indirect_pointer
                    |---Blocki
                            |--Blocki1-----{entry1, ...entry128}
                            |
                            |--Blocki2-----{entry1, ...entry128}
                            |
                            |...
                            |
                            |--Blocki2048-----{entry1, ...entry128}
```

辅助函数

这部分的辅助函数，注释解释地很清楚了，我就不赘述了：

```
//创建一个新的inode并写入inode_id指定的inode区
void create_inode(int inode_id, uint8_t is_file) {
    struct inode a;
    a.is_file = is_file;
    a.size = 0;
    a.indirect_pointer = allocate_block_in_bitmap();
    clear_block(a.indirect_pointer);
    a.st_Atime = time(NULL);
    a.st_Mtime = a.st_Atime;
    a.st_Ctime = a.st_Atime;
    write_inode(inode_id, &a);
}

//把栈上的inode写入inode_index指定的inode区
int write_inode(int inode_index, struct inode *a) {
    pthread_mutex_lock(&inode_mutex);
    uint8_t inode_block[BLOCK_SIZE];
    int block_num = 4 + (inode_index / INODE_NUM);
    int offset = (inode_index % INODE_NUM) * INODE_SIZE;

    if (safe_read(block_num, inode_block) != 0) {
        pthread_mutex_unlock(&inode_mutex);
        return -1;
    }

    memcpy(inode_block + offset, a, sizeof(struct inode));


    if (safe_write(block_num, inode_block) != 0) {
        pthread_mutex_unlock(&inode_mutex);
        return -1;
    }
    pthread_mutex_unlock(&inode_mutex);
    return 0;
}

//把inode_index指定的inode写入栈上的inode
int read_inode(int inode_index, struct inode *inode) {
    pthread_mutex_lock(&inode_mutex);
    uint8_t buffer[BLOCK_SIZE];
    int block_num = 4 + (inode_index / INODE_NUM);
    int offset = (inode_index % INODE_NUM) * INODE_SIZE;

    if (safe_read(block_num, buffer) != 0) {
        pthread_mutex_unlock(&inode_mutex);

```

```
        return -1;
    }

    memcpy(inode, buffer + offset, sizeof(struct inode));
    pthread_mutex_unlock(&inode_mutex);
    return 0;
}


//通过path找inode,成功则返回inode的index
int get_inode_by_path(const char *path, struct inode *inode) {
    if (strncmp(path, "/", 1) == 0 && strlen(path) == 1) {
        return read_inode(ROOT_INODE, inode);
    }

    int inode_index = ROOT_INODE;
    if (read_inode(inode_index, inode) != 0) {
        return -ENOENT;
    }

    char *path_copy = strdup(path);
    if (!path_copy) {
        return -ENOMEM;
    }

    char *token = strtok(path_copy, "/");
    while (token != NULL) {
        if(inode->is_file)
            return -ENOENT;
        uint16_t pointers[BLOCK_SIZE / sizeof(uint16_t)];
        if (safe_read(inode->indirect_pointer, pointers) != 0) {
            free(path_copy);
            return -ENOENT;
        }

        int found = 0;
        for (int i = 0; i < BLOCK_SIZE / sizeof(uint16_t); ++i) {
            if (pointers[i] == 0) {
                continue;;
            }

            uint8_t buffer[BLOCK_SIZE];
            if (safe_read(pointers[i], buffer) != 0) {
                free(path_copy);
                return -ENOENT;
            }

            struct dir_entry *entries = (struct dir_entry *)buffer;
            for (int j = 0; j < DIR_ENTRY_PER_BLOCK; ++j) {
```

```
                if(!entries[j].is_used)
                    continue;
                if (strcmp(entries[j].name, token) == 0) {
                    inode_index = entries[j].inode_index;
                    if (read_inode(inode_index, inode) != 0) {
                        free(path_copy);
                        return -ENOENT;
                    }
                    found = 1;
                    break;
                }
            }

            if (found) {
                break;
            }
        }

        if (!found) {
            free(path_copy);
            return -ENOENT;
        }

        token = strtok(NULL, "/");
    }

    free(path_copy);
    return inode_index;
}

//在parent_inode这个父文件夹中删除文件name的相关信息并更新时间
int delete_directory_entry(struct inode *parent_inode, int
parent_inode_index, const char *name) {
    pthread_mutex_lock(&dir_mutex);
    uint8_t pointers[BLOCK_SIZE];
    if (safe_read(parent_inode->indirect_pointer, pointers) != 0) {
        pthread_mutex_unlock(&dir_mutex);
        return -EIO;
    }

    uint16_t *dir_block_pointers = (uint16_t *)pointers;
    for (int i = 0; i < BLOCK_SIZE / sizeof(uint16_t); ++i) {
        if (dir_block_pointers[i] == 0) {
            continue;
        }

        uint8_t block[BLOCK_SIZE];
        if (safe_read(dir_block_pointers[i], block) != 0) {
            pthread_mutex_unlock(&dir_mutex);
```

```
            return -EIO;
        }

        struct dir_entry *entries = (struct dir_entry *)block;
        for (int j = 0; j < DIR_ENTRY_PER_BLOCK; ++j) {
            if (entries[j].is_used && strcmp(entries[j].name, name) ==
0) {
                int inode_index = entries[j].inode_index;
                entries[j].is_used = 0;
                entries[j].inode_index = 0;
                if (safe_write(dir_block_pointers[i], block) != 0) {
                    pthread_mutex_unlock(&dir_mutex);
                    return -EIO;
                }

                parent_inode->st_Mtime = parent_inode->st_Ctime =
time(NULL);
                if (write_inode(parent_inode_index, parent_inode) != 0)
{
                    pthread_mutex_unlock(&dir_mutex);
                    return -EIO;
                }
                pthread_mutex_unlock(&dir_mutex);
                return inode_index;
            }
        }
    }
    pthread_mutex_unlock(&dir_mutex);
    return -ENOENT;
}

//往parent_inode这个父文件夹中写入文件name的相关信息并更新时间
int add_directory_entry(struct inode *parent_inode, int
parent_inode_index, const char *name, int inode_index) {
    pthread_mutex_lock(&dir_mutex);
    uint8_t pointers[BLOCK_SIZE];
    if (safe_read(parent_inode->indirect_pointer, pointers) != 0) {
        pthread_mutex_unlock(&dir_mutex);
        return -EIO;
    }

    uint16_t *dir_block_pointers = (uint16_t *)pointers;
    for (int i = 0; i < BLOCK_SIZE / sizeof(uint16_t); ++i) {
        if (dir_block_pointers[i] == 0) {
            dir_block_pointers[i] = allocate_block_in_bitmap();
            if (dir_block_pointers[i] < 0) {
                pthread_mutex_unlock(&dir_mutex);
                return -ENOSPC;
            }
```

```
            clear_block(dir_block_pointers[i]);
            safe_write(parent_inode->indirect_pointer,pointers);
        }

        uint8_t block[BLOCK_SIZE];
        if (safe_read(dir_block_pointers[i], block) != 0) {
            pthread_mutex_unlock(&dir_mutex);
            return -EIO;
        }

        struct dir_entry *entries = (struct dir_entry *)block;
        for (int j = 0; j < DIR_ENTRY_PER_BLOCK; ++j) {
            if (!entries[j].is_used) {
                strncpy(entries[j].name, name, NAME_MAX);
                entries[j].inode_index = inode_index;
                entries[j].is_used = 1;
                if (safe_write(dir_block_pointers[i], block) != 0) {
                    pthread_mutex_unlock(&dir_mutex);
                    return -EIO;
                }
                parent_inode->st_Mtime = parent_inode->st_Ctime =
time(NULL);
                if (write_inode(parent_inode_index, parent_inode) != 0)
{
                    pthread_mutex_unlock(&dir_mutex);
                    return -EIO;
                }
                pthread_mutex_unlock(&dir_mutex);
                return 0;
            }
        }
    }
    pthread_mutex_unlock(&dir_mutex);
    return -ENOSPC;
}

//修改inode的大小为new_size
int resize_file(struct inode *inode, size_t new_size) {
    size_t current_blocks = (inode->size + BLOCK_SIZE - 1) / BLOCK_SIZE;
    size_t new_blocks = (new_size + BLOCK_SIZE - 1) / BLOCK_SIZE;

    uint16_t pointers[BLOCK_SIZE / sizeof(uint16_t)];
    if (safe_read(inode->indirect_pointer, pointers) != 0) {
        return -EIO;
    }

    if (new_size > inode->size) { // 扩展文件
        for (size_t i = current_blocks; i < new_blocks; ++i) {
            int new_block = allocate_block_in_bitmap();
```

```
            if (new_block < 0) {
                return -ENOSPC;
            }
            clear_block(new_block);
            pointers[i] = new_block;
        }
    } else if (new_size < inode->size) { // 缩小文件
        for (size_t i = new_blocks; i < current_blocks; ++i) {
            clear_block(pointers[i]);
            free_block_in_bitmap(pointers[i]);
            pointers[i] = 0;
        }
    }

    if (safe_write(inode->indirect_pointer, pointers) != 0) {
        return -EIO;
    }

    inode->size = new_size;
    return 0;
}
```

Data Block

辅助函数

这一部分主要就有两个辅助函数：

一个是清空一个Block

一个是清空一个inode，包括其indirect_pointer指向的空间，indirect_pointer指向空间内存放的指针指向的空间，并在Bitmap上把相应的inode和data block置0

```
int clear_block(int block_id) {
    uint8_t buffer[BLOCK_SIZE] = {0};
    return safe_write(block_id, buffer);
}

void free_inode_and_blocks(struct inode *inode, int inode_index) {
    uint16_t pointers[BLOCK_SIZE / sizeof(uint16_t)];
    if (safe_read(inode->indirect_pointer, pointers) != 0) {
        return;
    }

    // 清空并释放所有指向的数据块
    for (int i = 0; i < BLOCK_SIZE / sizeof(uint16_t); ++i) {
        if (pointers[i] != 0) {
            clear_block(pointers[i]);
```

```
            free_block_in_bitmap(pointers[i]);
        }
    }

    clear_block(inode->indirect_pointer);
    free_block_in_bitmap(inode->indirect_pointer);
    free_inode_in_bitmap(inode_index);
}
```

主体函数

mkfs

做了五件事，初始化锁，Super Block，Bitmap，Inode ，然后创建根目录的Inode。

```
int mkfs()
{
    init_locks();
    //初始化super blcok
    struct superblock sb;
    init_superblock(&sb);

    // 初始化data Bitmap
    uint8_t data_block_bitmap[BLOCK_SIZE * 2] = {0};
    for (int i = 0; i <= 259; i++) {
        set_bit(data_block_bitmap, i);
    }
    if (safe_write(1, data_block_bitmap) != 0 || safe_write(2,
data_block_bitmap + BLOCK_SIZE) != 0) {
        return -1;
    }

    // 初始化inode Bitmap
    uint8_t inode_bitmap[BLOCK_SIZE] = {0};
    set_bit(inode_bitmap, ROOT_INODE);
    if (safe_write(3, inode_bitmap) != 0) {
        return -1;
    }

    // 初始化inode
    uint8_t inode_blocks[BLOCK_SIZE * INODE_BLOCKS] = {0};
    for (int i = 0; i < INODE_BLOCKS; i++) {
        if (safe_write(4 + i, inode_blocks + i * BLOCK_SIZE) != 0) {
            return -1;
        }
    }
```

```
    //写入根目录inode
    create_inode(0,0);
    return 0;
}
```

fs_getattr

这个函数很简单，只需要通过path找到对应的inode，然后根据inode填写attr即可。

```
int fs_getattr (const char *path, struct stat *attr)
{
    printf("fs_getattr is called:%s\n",path);
    struct inode inode;
    if (get_inode_by_path(path, &inode) < 0) {
        return -ENOENT;
    }

    attr->st_mode = inode.is_file ? REGMODE : DIRMODE;
    attr->st_nlink = 1;
    attr->st_uid = getuid();
    attr->st_gid = getgid();
    attr->st_size = inode.size;
    attr->st_atime = inode.st_Atime;
    attr->st_mtime = inode.st_Mtime;
    attr->st_ctime = inode.st_Ctime;

    return 0;
}
```

fs_readdir

主要做了以下几件事：

- 根据path找到inode，并检验其是目录
- 读取indirect_pointer指向的Block并存入pointers，遍历pointers中的2048个指针，若指针
    - 是0，则跳过
    - 非0，读取指针指向的Block到entries，遍历128个entries
        - 如果is_used==1，则filler(buffer, entries[j].name, NULL, 0)
- 最后更新inode的Atime，并调用write_inode写入磁盘

```

```

```c
int fs_readdir(const char *path, void *buffer, fuse_fill_dir_t filler,
off_t offset, struct fuse_file_info *fi) {
    printf("fs_readdir is called:%s\n",path);
    struct inode dir_inode;
    int inode_index = get_inode_by_path(path, &dir_inode);
    if (inode_index < 0) {
        return inode_index;
    }

    if (dir_inode.is_file) {
        return -ENOTDIR;
    }

    uint16_t pointers[BLOCK_SIZE / sizeof(uint16_t)];
    if (safe_read(dir_inode.indirect_pointer, pointers) != 0) {
        return -EIO;
    }

    for (int i = 0; i < BLOCK_SIZE / sizeof(uint16_t); ++i) {
        if (pointers[i] == 0) {
            continue;
        }

        uint8_t block[BLOCK_SIZE];
        if (safe_read(pointers[i], block) != 0) {
            return -EIO;
        }

        struct dir_entry *entries = (struct dir_entry *)block;
        for (int j = 0; j < DIR_ENTRY_PER_BLOCK; ++j) {
            if(!entries[j].is_used)
                continue;
            if (filler(buffer, entries[j].name, NULL, 0) != 0) {
                return -ENOMEM;
            }
        }
    }

    dir_inode.st_Atime = time(NULL);
    if (write_inode(inode_index, &dir_inode) != 0) {
        return -EIO;
    }

    return 0;
}
```

fs_read

这个函数和上个函数差不多，主要干了以下的事情：

- 根据path找inode
- 把indirect_block指向的Block存入data_block_pointers，然后计算初始读取的位置block_offset，从data_block_pointers[block_offset]指向的Block开始读，一直读完size个字节。把读入的内容全存入buffer
- 最后更新Atime

```
int fs_read(const char *path, char *buffer, size_t size, off_t offset,
struct fuse_file_info *fi) {
    printf("fs_read is called:%s\n",path);
    struct inode file_inode;
    int inode_index = get_inode_by_path(path, &file_inode);
    if (inode_index < 0) {
        return inode_index;
    }

    if (!file_inode.is_file) {
        return -EISDIR;
    }

    if (offset >= file_inode.size) {
        return 0;
    }

    size_t bytes_to_read = min(size, file_inode.size - offset);
    uint8_t indirect_block[BLOCK_SIZE];
    if (safe_read(file_inode.indirect_pointer, indirect_block) != 0) {
        return -EIO;
    }

    uint16_t *data_block_pointers = (uint16_t *)indirect_block;
    size_t bytes_read = 0;
    size_t block_offset = offset / BLOCK_SIZE;
    size_t block_start_offset = offset % BLOCK_SIZE;

    while (bytes_read < bytes_to_read) {
        if (data_block_pointers[block_offset] == 0) {
            return -EIO;
        }

        uint8_t data_block[BLOCK_SIZE];
        if (safe_read(data_block_pointers[block_offset], data_block) !=
0) {
            return -EIO;
        }

        size_t bytes_from_block = min(BLOCK_SIZE - block_start_offset,
```

fs_mknod

这个函数是创建一个常规文件的函数，主要干了以下的事情：

- 使用dirname和basename把path分成父文件夹路径dir_path和文件名字base_name，然后根据dir_path找到父文件夹的inode为dir_inode
- 调用get_new_inode_bit_in_bitmap为新文件在Bitmap上找一个空位new_inode_index
- 调用create_inode在new_inode_index创建一个常规文件
- 调用add_directory_entry在父文件夹中添加这个新文件的信息

```
int fs_mknod(const char *path, mode_t mode, dev_t dev) {
    printf("fs_mknod is called:%s\n",path);
    char *path_copy1 = strdup(path);
    char *path_copy2 = strdup(path);
    if (!path_copy1 || !path_copy2) {
        free(path_copy1);
        free(path_copy2);
        return -ENOMEM;
    }

    char *dir_path = dirname(path_copy1);
    char *base_name = basename(path_copy2);

    struct inode dir_inode;
    int parent_inode_index = get_inode_by_path(dir_path, &dir_inode);
    if (parent_inode_index < 0) {
        free(path_copy1);
        free(path_copy2);
        return parent_inode_index;
    }

    if (dir_inode.is_file) {
        free(path_copy1);
        free(path_copy2);
        return -ENOTDIR;
    }

    int new_inode_index = get_new_inode_bit_in_bitmap();
    if (new_inode_index < 0) {
        free(path_copy1);
        free(path_copy2);
        return -ENOSPC;
    }

    create_inode(new_inode_index, 1);
```

```
    int ret = add_directory_entry(&dir_inode, parent_inode_index,
base_name, new_inode_index);
    free(path_copy1);
    free(path_copy2);
    return ret;
}
```

fs_mkdir

这个函数是创建一个目录文件，和上一个函数基本一样的，除了一点：

- create_inode的第二个参数为0，代表此时创建的是目录文件

```
int fs_mkdir(const char *path, mode_t mode) {
    printf("fs_mkdir is called:%s\n",path);
    char *path_copy1 = strdup(path);
    char *path_copy2 = strdup(path);
    if (!path_copy1 || !path_copy2) {
        free(path_copy1);
        free(path_copy2);
        return -ENOMEM;
    }

    char *dir_path = dirname(path_copy1);
    char *base_name = basename(path_copy2);

    struct inode dir_inode;
    int parent_inode_index = get_inode_by_path(dir_path, &dir_inode);
    if (parent_inode_index < 0) {
        free(path_copy1);
        free(path_copy2);
        return parent_inode_index;
    }

    if (dir_inode.is_file) {
        free(path_copy1);
        free(path_copy2);
        return -ENOTDIR;
    }

    int new_inode_index = get_new_inode_bit_in_bitmap();
    if (new_inode_index < 0) {
        free(path_copy1);
        free(path_copy2);
        return -ENOSPC;
    }
```

```
    create_inode(new_inode_index, 0);

    int ret = add_directory_entry(&dir_inode, parent_inode_index,
base_name, new_inode_index);
    free(path_copy1);
    free(path_copy2);
    return ret;
}
```

fs_rmdir

这个函数是删除一个目录文件，主要流程是：

- 使用dirname和basename把path分成父文件夹路径dir_path和文件名字base_name，然后根据dir_path找到父文件夹的inode为dir_inode，根据base_name找到目标文件的inode为target_inode
- 调用delete_directory_entry，在dir_inode中删除目标文件的信息
- 调用free_inode_and_blocks清空目标文件

```
int fs_rmdir(const char *path) {
    printf("fs_rmdir is called:%s\n",path);
    char *path_copy1 = strdup(path);
    char *path_copy2 = strdup(path);
    if (!path_copy1 || !path_copy2) {
        free(path_copy1);
        free(path_copy2);
        return -ENOMEM;
    }

    char *dir_path = dirname(path_copy1);
    char *base_name = basename(path_copy2);

    struct inode dir_inode;
    int parent_inode_index = get_inode_by_path(dir_path, &dir_inode);
    if (parent_inode_index < 0) {
        free(path_copy1);
        free(path_copy2);
        return parent_inode_index;
    }

    if (dir_inode.is_file) {
        free(path_copy1);
        free(path_copy2);
        return -ENOTDIR;
    }
```

```
    struct inode target_inode;
    int target_inode_index = get_inode_by_path(path, &target_inode);
    if (target_inode_index < 0) {
        free(path_copy1);
        free(path_copy2);
        return target_inode_index;
    }

    if (target_inode.is_file) {
        free(path_copy1);
        free(path_copy2);
        return -ENOTDIR;
    }

    int ret = delete_directory_entry(&dir_inode, parent_inode_index,
base_name);
    if (ret < 0) {
        free(path_copy1);
        free(path_copy2);
        return ret;
    }

    free_inode_and_blocks(&target_inode, target_inode_index);

    free(path_copy1);
    free(path_copy2);
    return 0;
}
```

fs_unlink

这个函数就是删一个常规文件，和上一个函数一模一样

```
int fs_unlink(const char *path) {
    printf("fs_unlink is called:%s\n",path);
    char *path_copy1 = strdup(path);
    char *path_copy2 = strdup(path);
    if (!path_copy1 || !path_copy2) {
        free(path_copy1);
        free(path_copy2);
        return -ENOMEM;
    }

    char *dir_path = dirname(path_copy1);
    char *base_name = basename(path_copy2);

    struct inode dir_inode;
```

```
    int parent_inode_index = get_inode_by_path(dir_path, &dir_inode);
    if (parent_inode_index < 0) {
        free(path_copy1);
        free(path_copy2);
        return parent_inode_index;
    }

    if (dir_inode.is_file) {
        free(path_copy1);
        free(path_copy2);
        return -ENOTDIR;
    }

    struct inode target_inode;
    int target_inode_index = get_inode_by_path(path, &target_inode);
    if (target_inode_index < 0) {
        free(path_copy1);
        free(path_copy2);
        return target_inode_index;
    }

    if (!target_inode.is_file) {
        free(path_copy1);
        free(path_copy2);
        return -EISDIR;
    }

    int ret = delete_directory_entry(&dir_inode, parent_inode_index,
base_name);
    if (ret < 0) {
        free(path_copy1);
        free(path_copy2);
        return ret;
    }

    free_inode_and_blocks(&target_inode, target_inode_index);

    free(path_copy1);
    free(path_copy2);
    return 0;
}
```

fs_rename

这个函数是把oldpath的文件转移到newpath，主要流程：

- 通过oldpath和newpath分别找到四个inode，即两个路径的父文件夹和目标文件，记为old_dir_inode（老父文件夹），target_inode（老文件），new_dir_inode（新父文件夹），check_inode（新文件）
- 调用add_directory_entry，在new_dir_inode中添加新文件的名字和inode编号
- 调用delete_directory_entry，在old_dir_inode中删除target_inode

```
int fs_rename(const char *oldpath, const char *newpath) {
    printf("fs_rename is called:%s\n",oldpath);
    char *oldpath_copy1 = strdup(oldpath);
    char *oldpath_copy2 = strdup(oldpath);
    if (!oldpath_copy1 || !oldpath_copy2) {
        free(oldpath_copy1);
        free(oldpath_copy2);
        return -ENOMEM;
    }

    char *old_dir_path = dirname(oldpath_copy1);
    char *old_base_name = basename(oldpath_copy2);

    struct inode old_dir_inode;
    int old_parent_inode_index = get_inode_by_path(old_dir_path,
&old_dir_inode);
    if (old_parent_inode_index < 0) {
        free(oldpath_copy1);
        free(oldpath_copy2);
        return old_parent_inode_index;
    }

    struct inode target_inode;
    int target_inode_index = get_inode_by_path(oldpath, &target_inode);
    if (target_inode_index < 0) {
        free(oldpath_copy1);
        free(oldpath_copy2);
        return target_inode_index;
    }

    char *newpath_copy1 = strdup(newpath);
    char *newpath_copy2 = strdup(newpath);
    if (!newpath_copy1 || !newpath_copy2) {
        free(oldpath_copy1);
        free(oldpath_copy2);
        free(newpath_copy1);
        free(newpath_copy2);
        return -ENOMEM;
    }
```

```
    char *new_dir_path = dirname(newpath_copy1);
    char *new_base_name = basename(newpath_copy2);

    struct inode new_dir_inode;
    int new_parent_inode_index = get_inode_by_path(new_dir_path,
&new_dir_inode);
    if (new_parent_inode_index < 0) {
        free(oldpath_copy1);
        free(oldpath_copy2);
        free(newpath_copy1);
        free(newpath_copy2);
        return new_parent_inode_index;
    }

    struct inode check_inode;
    int check_result = get_inode_by_path(newpath, &check_inode);
    if (check_result > 0) {
        free(oldpath_copy1);
        free(oldpath_copy2);
        free(newpath_copy1);
        free(newpath_copy2);
        return -EEXIST;
    }

    int ret = add_directory_entry(&new_dir_inode,
new_parent_inode_index, new_base_name, target_inode_index);
    if (ret < 0) {
        free(oldpath_copy1);
        free(oldpath_copy2);
        free(newpath_copy1);
        free(newpath_copy2);
        return ret;
    }

    ret = delete_directory_entry(&old_dir_inode, old_parent_inode_index,
old_base_name);
    if (ret < 0) {
        free(oldpath_copy1);
        free(oldpath_copy2);
        free(newpath_copy1);
        free(newpath_copy2);
        return ret;
    }

    free(oldpath_copy1);
    free(oldpath_copy2);
    free(newpath_copy1);
    free(newpath_copy2);
    return 0;
```

```
}
```

fs_write

这个和fs_read差不多，不过是操作反过来了而已，不再赘述

```
int fs_write(const char *path, const char *buffer, size_t size, off_t
offset, struct fuse_file_info *fi) {
    printf("fs_write is called:%s\n",path);
    struct inode file_inode;
    int inode_index = get_inode_by_path(path, &file_inode);
    if (inode_index < 0) {
        return 0;
    }

    if (!file_inode.is_file) {
        return 0;
    }

    if (fi->flags & O_APPEND) {
        offset = file_inode.size;
    }

    size_t end_offset = offset + size;
    if(end_offset>file_inode.size){
        if (resize_file(&file_inode, end_offset) < 0) {
            return 0;
        }
    }
    size_t bytes_written = 0;
    uint16_t pointers[BLOCK_SIZE / sizeof(uint16_t)];
    if (safe_read(file_inode.indirect_pointer, pointers) != 0) {
        return 0;
    }
    while (bytes_written < size) {
        size_t block_index = (offset + bytes_written) / BLOCK_SIZE;
        size_t block_offset = (offset + bytes_written) % BLOCK_SIZE;
        size_t bytes_to_write = min(BLOCK_SIZE - block_offset, size -
bytes_written);

        uint8_t block[BLOCK_SIZE];
        if (safe_read(pointers[block_index], block) != 0) {
            return 0;
        }

        memcpy(block + block_offset, buffer + bytes_written,
bytes_to_write);
```

```
        if (safe_write(pointers[block_index], block) != 0) {
            return 0;
        }

        bytes_written += bytes_to_write;
    }

    file_inode.st_Mtime = file_inode.st_Ctime = time(NULL);
    if (write_inode(inode_index, &file_inode) != 0) {
        return 0;
    }


    return bytes_written;
}
```

fs_truncate

这个函数是修改文件大小的：

- 先通过path找到inode
- 再调用resize_file
- 最后更新下Ctime

```
int fs_truncate(const char *path, off_t size) {
    printf("fs_truncate is called:%s\n",path);
    struct inode file_inode;
    int inode_index = get_inode_by_path(path, &file_inode);
    if (inode_index < 0) {
        return inode_index;
    }

    if (!file_inode.is_file) {
        return -EISDIR;
    }

    int ret = resize_file(&file_inode, size);
    if (ret < 0) {
        return ret;
    }

    file_inode.st_Ctime = time(NULL);
    if (write_inode(inode_index, &file_inode) != 0) {
        return -EIO;
    }

    return 0;
```

```
}
```

fs_utime

这个函数是修改文件时间的：

- 先通过path找到inode
- 修改时间

```
int fs_utime(const char *path, struct utimbuf *buffer) {
    printf("fs_utime is called:%s\n",path);
    struct inode file_inode;
    int inode_index = get_inode_by_path(path, &file_inode);
    if (inode_index < 0) {
        return inode_index;
    }

    file_inode.st_Atime = buffer->actime;
    file_inode.st_Mtime = buffer->modtime;

    file_inode.st_Ctime = time(NULL);

    if (write_inode(inode_index, &file_inode) != 0) {
        return -EIO;
    }

    return 0;
}
```

fs_statfs

这个函数就是调用count_free_bits两次去数一数inode和data block的bitmap有多少0，并给stat赋值。

但是我既然设计了Super Block也不能完全不管它，所以我在这个函数中顺便更新了Super Block

```
int fs_statfs(const char *path, struct statvfs *stat) {
    printf("fs_statfs is called:%s\n",path);
    struct superblock sb;
    if (safe_read(0, &sb) != 0) {
        return -EIO;
    }

    memset(stat, 0, sizeof(struct statvfs));
```

```
    stat->f_bsize = sb.f_bsize;
    stat->f_blocks = sb.f_blocks;
    stat->f_files = sb.f_files;
    stat->f_namemax = sb.f_namemax;

    uint8_t block_bitmap[BLOCK_SIZE * 2];
    if (safe_read(1, block_bitmap) != 0 || safe_read(2, block_bitmap +
BLOCK_SIZE) != 0) {
        return -EIO;
    }

    uint8_t inode_bitmap[BLOCK_SIZE];
    if (safe_read(3, inode_bitmap) != 0) {
        return -EIO;
    }

    int free_blocks = count_free_bits(block_bitmap, BLOCK_SIZE * 2);
    int free_inodes = count_free_bits(inode_bitmap, BLOCK_SIZE);

    sb.f_bfree=sb.f_bavail=stat->f_bfree = stat->f_bavail = free_blocks;
    sb.f_ffree=sb.f_favail=stat->f_ffree = stat->f_favail = free_inodes;
    if (safe_write(0, &sb) != 0) {
        return -EIO;
    }

    return 0;
}
```

fs_open

这个函数没啥说的，就是根据fi->flags和O_APPEND来设置inode的size

```
int fs_open(const char *path, struct fuse_file_info *fi) {
    printf("fs_open is called:%s\n",path);
    struct inode file_inode;
    int inode_index = get_inode_by_path(path, &file_inode);
    if (inode_index < 0) {
        return inode_index;
    }

    if (!file_inode.is_file) {
        return -EISDIR;
    }

    if (fi->flags & O_APPEND) {
        fi->fh = file_inode.size;
    } else {
```

```
        fi->fh = 0;
    }

    return 0;
}
```

遇到的问题

本次实验进行的比较顺利，没遇到太多问题。也就以下三个问题：

- dir_entry起初是没有设计is_used这个参数的。导致fs_readdir函数中，把一个Block的128个dir_entry全都调用了filler函数指针，而其中大部分的dir_entry是空的，这让ls指令出现了问题
- dir_entry一开始设计的时候，name后面是is_used。而对活跃的dir_entry，is_used=1。所以当文件名字正好是24字节时（2.sh），strcmp(name,...)会出现严重的错误。
- 在add_directory_entry等函数中，有时需要修改inode的indirect_pointer指向的那个Block，为其增添一个指针。但是我修改后没有写入磁盘中，导致了一系列错误

附加思考

1.多个线程可能会同时访问和修改共享资源，如全局变量、文件系统结构等。这可能导致以下问题：

- **竞争**：多个线程同时读取和修改共享数据，导致数据不一致。
- **死锁**：两个或多个线程相互等待对方释放资源，导致程序停止运行。

特别对于文件系统来说，有一个经典的读—写问题，如果不妥善处理会使读入和写入磁盘的操作有巨大的问题

2.在文件系统中，以下几个地方需要互斥访问：

位图操作：分配和释放数据块和 inode 时，需要互斥访问位图。

inode 读写：读写 inode 信息时需要互斥访问，以防止数据不一致。

目录操作：添加和删除目录时需要互斥访问。

可以对相应的辅助函数加锁，例如：

```
pthread_mutex_t bitmap_mutex;
pthread_mutex_t inode_mutex;
pthread_mutex_t dir_mutex;

void init_locks() {
    pthread_mutex_init(&bitmap_mutex, NULL);
    pthread_mutex_init(&inode_mutex, NULL);
```

```
    pthread_mutex_init(&dir_mutex, NULL);
}

void destroy_locks() {
    pthread_mutex_destroy(&bitmap_mutex);
    pthread_mutex_destroy(&inode_mutex);
    pthread_mutex_destroy(&dir_mutex);
}

int allocate_block_in_bitmap() {
    pthread_mutex_lock(&bitmap_mutex);
    // ......
    pthread_mutex_unlock(&bitmap_mutex);
}

void free_block_in_bitmap(int block_id) {
    pthread_mutex_lock(&bitmap_mutex);
    // ......
    pthread_mutex_unlock(&bitmap_mutex);
}

int write_inode(int inode_index, struct inode *a) {
    pthread_mutex_lock(&inode_mutex);
    // ......
    pthread_mutex_unlock(&inode_mutex);
}

int read_inode(int inode_index, struct inode *inode) {
    pthread_mutex_lock(&inode_mutex);
    // ......
    pthread_mutex_unlock(&inode_mutex);
}

int add_directory_entry(struct inode *parent_inode, int
parent_inode_index, const char *name, int inode_index) {
    pthread_mutex_lock(&dir_mutex);
    // ......
    pthread_mutex_unlock(&dir_mutex);
}

int delete_directory_entry(struct inode *parent_inode, int
parent_inode_index, const char *name) {
    pthread_mutex_lock(&dir_mutex);
    // ......
    pthread_mutex_unlock(&dir_mutex);
}
```

3.已添加至代码

样例测试结果

本次实验我的代码能通过所有的16个样例点，具体输入如下：

- **0：没有输出**
- **1**

```
dir1  dir2
dir3
```

- **2**

```
file1  file2
abcdefghijklmnopqrstuvwx  file1  file2
```

- **3**

```
0
32
10000
24
```

- **4:这里时区不同，所以样例是+0005，而本地输出是+0001**

```
2012-12-21 00:00:36.000000000 +00002012-12-21 00:00:36.000000000 +0000
2012-12-21 00:00:36.000000000 +00002012-12-21 00:00:36.000000000 +0000
```

- **5**

```
dir1  dir3
dir3
dir1  dir3
```

- **6**

```
file1  file2  picture
file1  file2
file1
```

- **7**

```
dir2  file1
dir2  file2
dir3  file2
dir
dir3  file2
```

- 8

```
dir1-1  dir1-2
dir1  dir2
dir2
dir2-1
dir1-1  dir1-2
dir1-1  dir2
dir1-2
```

- 9

```
fuse
```

- 10

```
12345
23333333333
23333333333
abcdefg
```

- 11：没有任何输出，因为fs.c和fs-test.c相同，diff fs.c fs-test.c不会有输出

- 12：这里的输出和样例有些许不同，File没有带引号，uid和gid也不同，但这是因为系统不同导致的

```
  File: file1
  Size: 0               Blocks: 0          IO Block: 4096   regular
empty file
Access: (0644/-rw-r--r--)  Uid: ( 1087/u2022201459)   Gid: ( 1002/
2022ICS)
  File: file1
  Size: 6               Blocks: 0          IO Block: 4096   regular file
Access: (0644/-rw-r--r--)  Uid: ( 1087/u2022201459)   Gid: ( 1002/
2022ICS)
```

- 13：和第11题类似，没有任何输出，因为test.in和test.in.out相同，diff test.in test.in.out不会有输出
- 14

```
```

```
dir0     dir145  dir213  dir282  dir350  dir419  dir488  dir556  dir624
 dir693  dir761  dir83   dir899  dir967    file112  file181  file25
file318  file387  file455  file523  file592  file660  file729  file798
 file866  file934
dir1     dir146  dir214  dir283  dir351  dir42   dir489  dir557  dir625
 dir694  dir762  dir830  dir9    dir968    file113  file182  file250
 file319  file388  file456  file524  file593  file661  file73   file799
 file867  file935
dir10    dir147  dir215  dir284  dir352  dir420  dir49   dir558  dir626
 dir695  dir763  dir831  dir90   dir969    file114  file183  file251
 file32   file389  file457  file525  file594  file662  file730  file8
 file868  file936
dir100   dir148  dir216  dir285  dir353  dir421  dir490  dir559  dir627
 dir696  dir764  dir832  dir900  dir97     file115  file184  file252
 file320  file39   file458  file526  file595  file663  file731  file80
file869  file937
dir1000  dir149  dir217  dir286  dir354  dir422  dir491  dir56   dir628
 dir697  dir765  dir833  dir901  dir970    file116  file185  file253
 file321  file390  file459  file527  file596  file664  file732  file800
 file87   file938
dir1001  dir15   dir218  dir287  dir355  dir423  dir492  dir560  dir629
 dir698  dir766  dir834  dir902  dir971    file117  file186  file254
 file322  file391  file46   file528  file597  file665  file733  file801
 file870  file939
dir1002  dir150  dir219  dir288  dir356  dir424  dir493  dir561  dir63
dir699  dir767  dir835  dir903  dir972    file118  file187  file255
 file323  file392  file460  file529  file598  file666  file734  file802
 file871  file94
dir1003  dir151  dir22   dir289  dir357  dir425  dir494  dir562  dir630
 dir7    dir768  dir836  dir904  dir973    file119  file188  file256
 file324  file393  file461  file53   file599  file667  file735  file803
 file872  file940
dir1004  dir152  dir220  dir29   dir358  dir426  dir495  dir563  dir631
 dir70   dir769  dir837  dir905  dir974    file12   file189  file257
 file325  file394  file462  file530  file6    file668  file736  file804
 file873  file941
dir1005  dir153  dir221  dir290  dir359  dir427  dir496  dir564  dir632
 dir700  dir77   dir838  dir906  dir975    file120  file19   file258
 file326  file395  file463  file531  file60   file669  file737  file805
 file874  file942
dir1006  dir154  dir222  dir291  dir36   dir428  dir497  dir565  dir633
 dir701  dir770  dir839  dir907  dir976    file121  file190  file259
 file327  file396  file464  file532  file600  file67   file738  file806
 file875  file943
dir1007  dir155  dir223  dir292  dir360  dir429  dir498  dir566  dir634
 dir702  dir771  dir84   dir908  dir977    file122  file191  file26
file328  file397  file465  file533  file601  file670  file739  file807
 file876  file944
```

```
dir1008  dir156  dir224  dir293  dir361  dir43   dir499  dir567  dir635
 dir703  dir772  dir840  dir909  dir978    file123  file192  file260
 file329  file398  file466  file534  file602  file671  file74   file808
 file877  file945
dir1009  dir157  dir225  dir294  dir362  dir430  dir5    dir568  dir636
 dir704  dir773  dir841  dir91   dir979    file124  file193  file261
 file33   file399  file467  file535  file603  file672  file740  file809
 file878  file946
dir101   dir158  dir226  dir295  dir363  dir431  dir50   dir569  dir637
 dir705  dir774  dir842  dir910  dir98     file125  file194  file262
 file330  file4    file468  file536  file604  file673  file741  file81
file879  file947
dir1010  dir159  dir227  dir296  dir364  dir432  dir500  dir57   dir638
 dir706  dir775  dir843  dir911  dir980    file126  file195  file263
 file331  file40   file469  file537  file605  file674  file742  file810
 file88   file948
dir1011  dir16   dir228  dir297  dir365  dir433  dir501  dir570  dir639
 dir707  dir776  dir844  dir912  dir981    file127  file196  file264
 file332  file400  file47   file538  file606  file675  file743  file811
 file880  file949
dir1012  dir160  dir229  dir298  dir366  dir434  dir502  dir571  dir64
dir708  dir777  dir845  dir913  dir982    file128  file197  file265
 file333  file401  file470  file539  file607  file676  file744  file812
 file881  file95
dir1013  dir161  dir23   dir299  dir367  dir435  dir503  dir572  dir640
 dir709  dir778  dir846  dir914  dir983    file129  file198  file266
 file334  file402  file471  file54   file608  file677  file745  file813
 file882  file950
dir1014  dir162  dir230  dir3    dir368  dir436  dir504  dir573  dir641
 dir71   dir779  dir847  dir915  dir984    file13   file199  file267
 file335  file403  file472  file540  file609  file678  file746  file814
 file883  file951
dir1015  dir163  dir231  dir30   dir369  dir437  dir505  dir574  dir642
 dir710  dir78   dir848  dir916  dir985    file130  file2    file268
 file336  file404  file473  file541  file61   file679  file747  file815
 file884  file952
dir1016  dir164  dir232  dir300  dir37   dir438  dir506  dir575  dir643
 dir711  dir780  dir849  dir917  dir986    file131  file20   file269
 file337  file405  file474  file542  file610  file68   file748  file816
 file885  file953
dir1017  dir165  dir233  dir301  dir370  dir439  dir507  dir576  dir644
 dir712  dir781  dir85   dir918  dir987    file132  file200  file27
file338  file406  file475  file543  file611  file680  file749  file817
 file886  file954
dir1018  dir166  dir234  dir302  dir371  dir44   dir508  dir577  dir645
 dir713  dir782  dir850  dir919  dir988    file133  file201  file270
 file339  file407  file476  file544  file612  file681  file75   file818
 file887  file955
```

```
dir1019  dir167  dir235  dir303  dir372  dir440  dir509  dir578  dir646
 dir714  dir783  dir851  dir92   dir989    file134  file202  file271
 file34   file408  file477  file545  file613  file682  file750  file819
 file888  file956
dir102   dir168  dir236  dir304  dir373  dir441  dir51   dir579  dir647
 dir715  dir784  dir852  dir920  dir99     file135  file203  file272
 file340  file409  file478  file546  file614  file683  file751  file82
file889  file957
dir1020  dir169  dir237  dir305  dir374  dir442  dir510  dir58   dir648
 dir716  dir785  dir853  dir921  dir990    file136  file204  file273
 file341  file41   file479  file547  file615  file684  file752  file820
 file89   file958
dir1021  dir17   dir238  dir306  dir375  dir443  dir511  dir580  dir649
 dir717  dir786  dir854  dir922  dir991    file137  file205  file274
 file342  file410  file48   file548  file616  file685  file753  file821
 file890  file959
dir1022  dir170  dir239  dir307  dir376  dir444  dir512  dir581  dir65
dir718  dir787  dir855  dir923  dir992    file138  file206  file275
 file343  file411  file480  file549  file617  file686  file754  file822
 file891  file96
dir1023  dir171  dir24   dir308  dir377  dir445  dir513  dir582  dir650
 dir719  dir788  dir856  dir924  dir993    file139  file207  file276
 file344  file412  file481  file55   file618  file687  file755  file823
 file892  file960
dir103   dir172  dir240  dir309  dir378  dir446  dir514  dir583  dir651
 dir72   dir789  dir857  dir925  dir994    file14   file208  file277
 file345  file413  file482  file550  file619  file688  file756  file824
 file893  file961
dir104   dir173  dir241  dir31   dir379  dir447  dir515  dir584  dir652
 dir720  dir79   dir858  dir926  dir995    file140  file209  file278
 file346  file414  file483  file551  file62   file689  file757  file825
 file894  file962
dir105   dir174  dir242  dir310  dir38   dir448  dir516  dir585  dir653
 dir721  dir790  dir859  dir927  dir996    file141  file21   file279
 file347  file415  file484  file552  file620  file69   file758  file826
 file895  file963
dir106   dir175  dir243  dir311  dir380  dir449  dir517  dir586  dir654
 dir722  dir791  dir86   dir928  dir997    file142  file210  file28
file348  file416  file485  file553  file621  file690  file759  file827
 file896  file964
dir107   dir176  dir244  dir312  dir381  dir45   dir518  dir587  dir655
 dir723  dir792  dir860  dir929  dir998    file143  file211  file280
 file349  file417  file486  file554  file622  file691  file76   file828
 file897  file965
dir108   dir177  dir245  dir313  dir382  dir450  dir519  dir588  dir656
 dir724  dir793  dir861  dir93   dir999    file144  file212  file281
 file35   file418  file487  file555  file623  file692  file760  file829
 file898  file966
```

```
dir109   dir178  dir246  dir314  dir383  dir451  dir52   dir589  dir657
 dir725  dir794  dir862  dir930  file0     file145  file213  file282
 file350  file419  file488  file556  file624  file693  file761  file83
file899  file967
dir11    dir179  dir247  dir315  dir384  dir452  dir520  dir59   dir658
 dir726  dir795  dir863  dir931  file1     file146  file214  file283
 file351  file42   file489  file557  file625  file694  file762  file830
 file9    file968
dir110   dir18   dir248  dir316  dir385  dir453  dir521  dir590  dir659
 dir727  dir796  dir864  dir932  file10    file147  file215  file284
 file352  file420  file49   file558  file626  file695  file763  file831
 file90   file969
dir111   dir180  dir249  dir317  dir386  dir454  dir522  dir591  dir66
dir728  dir797  dir865  dir933  file100   file148  file216  file285
 file353  file421  file490  file559  file627  file696  file764  file832
 file900  file97
dir112   dir181  dir25   dir318  dir387  dir455  dir523  dir592  dir660
 dir729  dir798  dir866  dir934  file1000  file149  file217  file286
 file354  file422  file491  file56   file628  file697  file765  file833
 file901  file970
dir113   dir182  dir250  dir319  dir388  dir456  dir524  dir593  dir661
 dir73   dir799  dir867  dir935  file1001  file15   file218  file287
 file355  file423  file492  file560  file629  file698  file766  file834
 file902  file971
dir114   dir183  dir251  dir32   dir389  dir457  dir525  dir594  dir662
 dir730  dir8    dir868  dir936  file1002  file150  file219  file288
 file356  file424  file493  file561  file63   file699  file767  file835
 file903  file972
dir115   dir184  dir252  dir320  dir39   dir458  dir526  dir595  dir663
 dir731  dir80   dir869  dir937  file1003  file151  file22   file289
 file357  file425  file494  file562  file630  file7    file768  file836
 file904  file973
dir116   dir185  dir253  dir321  dir390  dir459  dir527  dir596  dir664
 dir732  dir800  dir87   dir938  file1004  file152  file220  file29
file358  file426  file495  file563  file631  file70   file769  file837
 file905  file974
dir117   dir186  dir254  dir322  dir391  dir46   dir528  dir597  dir665
 dir733  dir801  dir870  dir939  file1005  file153  file221  file290
 file359  file427  file496  file564  file632  file700  file77   file838
 file906  file975
dir118   dir187  dir255  dir323  dir392  dir460  dir529  dir598  dir666
 dir734  dir802  dir871  dir94   file1006  file154  file222  file291
 file36   file428  file497  file565  file633  file701  file770  file839
 file907  file976
dir119   dir188  dir256  dir324  dir393  dir461  dir53   dir599  dir667
 dir735  dir803  dir872  dir940  file1007  file155  file223  file292
 file360  file429  file498  file566  file634  file702  file771  file84
file908  file977
```

```
dir12    dir189  dir257  dir325  dir394  dir462  dir530  dir6    dir668
 dir736  dir804  dir873  dir941  file1008  file156  file224  file293
 file361  file43   file499  file567  file635  file703  file772  file840
 file909  file978
dir120   dir19   dir258  dir326  dir395  dir463  dir531  dir60   dir669
 dir737  dir805  dir874  dir942  file1009  file157  file225  file294
 file362  file430  file5    file568  file636  file704  file773  file841
 file91   file979
dir121   dir190  dir259  dir327  dir396  dir464  dir532  dir600  dir67
dir738  dir806  dir875  dir943  file101   file158  file226  file295
 file363  file431  file50   file569  file637  file705  file774  file842
 file910  file98
dir122   dir191  dir26   dir328  dir397  dir465  dir533  dir601  dir670
 dir739  dir807  dir876  dir944  file1010  file159  file227  file296
 file364  file432  file500  file57   file638  file706  file775  file843
 file911  file980
dir123   dir192  dir260  dir329  dir398  dir466  dir534  dir602  dir671
 dir74   dir808  dir877  dir945  file1011  file16   file228  file297
 file365  file433  file501  file570  file639  file707  file776  file844
 file912  file981
dir124   dir193  dir261  dir33   dir399  dir467  dir535  dir603  dir672
 dir740  dir809  dir878  dir946  file1012  file160  file229  file298
 file366  file434  file502  file571  file64   file708  file777  file845
 file913  file982
dir125   dir194  dir262  dir330  dir4    dir468  dir536  dir604  dir673
 dir741  dir81   dir879  dir947  file1013  file161  file23   file299
 file367  file435  file503  file572  file640  file709  file778  file846
 file914  file983
dir126   dir195  dir263  dir331  dir40   dir469  dir537  dir605  dir674
 dir742  dir810  dir88   dir948  file1014  file162  file230  file3
 file368  file436  file504  file573  file641  file71   file779  file847
 file915  file984
dir127   dir196  dir264  dir332  dir400  dir47   dir538  dir606  dir675
 dir743  dir811  dir880  dir949  file1015  file163  file231  file30
file369  file437  file505  file574  file642  file710  file78   file848
 file916  file985
dir128   dir197  dir265  dir333  dir401  dir470  dir539  dir607  dir676
 dir744  dir812  dir881  dir95   file1016  file164  file232  file300
 file37   file438  file506  file575  file643  file711  file780  file849
 file917  file986
dir129   dir198  dir266  dir334  dir402  dir471  dir54   dir608  dir677
 dir745  dir813  dir882  dir950  file1017  file165  file233  file301
 file370  file439  file507  file576  file644  file712  file781  file85
file918  file987
dir13    dir199  dir267  dir335  dir403  dir472  dir540  dir609  dir678
 dir746  dir814  dir883  dir951  file1018  file166  file234  file302
 file371  file44   file508  file577  file645  file713  file782  file850
 file919  file988
```

```
dir130   dir2    dir268  dir336  dir404  dir473  dir541  dir61   dir679
 dir747  dir815  dir884  dir952  file1019  file167  file235  file303
 file372  file440  file509  file578  file646  file714  file783  file851
 file92   file989
dir131   dir20   dir269  dir337  dir405  dir474  dir542  dir610  dir68
dir748  dir816  dir885  dir953  file102   file168  file236  file304
 file373  file441  file51   file579  file647  file715  file784  file852
 file920  file99
dir132   dir200  dir27   dir338  dir406  dir475  dir543  dir611  dir680
 dir749  dir817  dir886  dir954  file1020  file169  file237  file305
 file374  file442  file510  file58   file648  file716  file785  file853
 file921  file990
dir133   dir201  dir270  dir339  dir407  dir476  dir544  dir612  dir681
 dir75   dir818  dir887  dir955  file1021  file17   file238  file306
 file375  file443  file511  file580  file649  file717  file786  file854
 file922  file991
dir134   dir202  dir271  dir34   dir408  dir477  dir545  dir613  dir682
 dir750  dir819  dir888  dir956  file1022  file170  file239  file307
 file376  file444  file512  file581  file65   file718  file787  file855
 file923  file992
dir135   dir203  dir272  dir340  dir409  dir478  dir546  dir614  dir683
 dir751  dir82   dir889  dir957  file1023  file171  file24   file308
 file377  file445  file513  file582  file650  file719  file788  file856
 file924  file993
dir136   dir204  dir273  dir341  dir41   dir479  dir547  dir615  dir684
 dir752  dir820  dir89   dir958  file103   file172  file240  file309
 file378  file446  file514  file583  file651  file72   file789  file857
 file925  file994
dir137   dir205  dir274  dir342  dir410  dir48   dir548  dir616  dir685
 dir753  dir821  dir890  dir959  file104   file173  file241  file31
file379  file447  file515  file584  file652  file720  file79   file858
 file926  file995
dir138   dir206  dir275  dir343  dir411  dir480  dir549  dir617  dir686
 dir754  dir822  dir891  dir96   file105   file174  file242  file310
 file38   file448  file516  file585  file653  file721  file790  file859
 file927  file996
dir139   dir207  dir276  dir344  dir412  dir481  dir55   dir618  dir687
 dir755  dir823  dir892  dir960  file106   file175  file243  file311
 file380  file449  file517  file586  file654  file722  file791  file86
file928  file997
dir14    dir208  dir277  dir345  dir413  dir482  dir550  dir619  dir688
 dir756  dir824  dir893  dir961  file107   file176  file244  file312
 file381  file45   file518  file587  file655  file723  file792  file860
 file929  file998
dir140   dir209  dir278  dir346  dir414  dir483  dir551  dir62   dir689
 dir757  dir825  dir894  dir962  file108   file177  file245  file313
 file382  file450  file519  file588  file656  file724  file793  file861
 file93   file999
```

```
dir141   dir21   dir279  dir347  dir415  dir484  dir552  dir620  dir69
dir758  dir826  dir895  dir963  file109   file178  file246  file314
 file383  file451  file52   file589  file657  file725  file794  file862
 file930
dir142   dir210  dir28   dir348  dir416  dir485  dir553  dir621  dir690
 dir759  dir827  dir896  dir964  file11    file179  file247  file315
 file384  file452  file520  file59   file658  file726  file795  file863
 file931
dir143   dir211  dir280  dir349  dir417  dir486  dir554  dir622  dir691
 dir76   dir828  dir897  dir965  file110   file18   file248  file316
 file385  file453  file521  file590  file659  file727  file796  file864
 file932
dir144   dir212  dir281  dir35   dir418  dir487  dir555  dir623  dir692
 dir760  dir829  dir898  dir966  file111   file180  file249  file317
 file386  file454  file522  file591  file66   file728  file797  file865
 file933
```

- **15**：这一题没有输出，但我修改了一下样例使之有输出：

```
cd mnt
for ((i=0;i<128;++i)); do
    mkdir dir
    ls
    cd dir
    done
```

输出见下，共128个dir：

```
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
```

```
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
```

```
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
```

```
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
dir
```