

# 深圳大学实验报告

课程名称： 操作系统

实验项目名称： 综合实验 2

学院： 计算机与软件学院

专业： 软件工程

指导教师： 张 滇

报告人： 郑彦薇 学号： 2020151022 班级： 软件工程 01 班

实验时间： 2023 年 06 月 08 日

实验报告提交时间： 2023/6/19

教务处制

一、实验目的与要求：

综合利用文件管理的相关知识，结合对文件系统的认知，编写简易文件系统，加深操作系统中文件系统和资源共享的认识。

二、方法、步骤：

实验基本按照以下步骤进行：

1. 分配 100MB 的共享内存，并映射到进程中。
2. 模拟 FAT 文件系统，对自己的文件管理系统结构进行设置。
3. 设计文件逻辑，确定数据结构。
4. 设置基本接口并实现。
5. 添加信号量机制，实现读写互斥。

数据结构设计、方法的实现和调用在下一部分进行具体的说明。

三、实验过程及内容：

1. 创建一个 100M 的文件或创建一个 100M 的共享内存

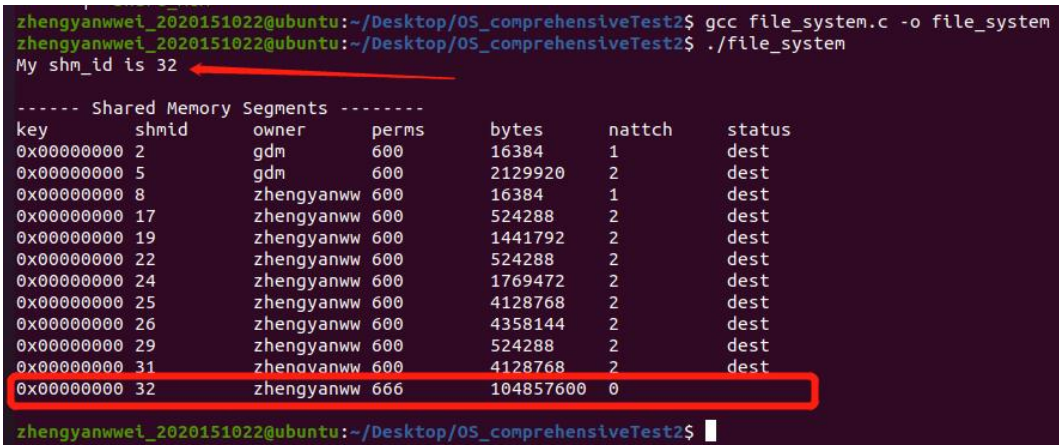
创建 100M 的共享内存，并将其映射到进程空间

```
const size_t MEM_SIZE = 100 << 20 ; //100MB memories
int sem_id;

int main(){
    void *share_mem = NULL;
    int shm_id;

    shm_id = shmget(IPC_PRIVATE, MEM_SIZE, 0666); //获得共享内存 id
    printf("My shm_id is %d\n",shm_id);
    system("ipcs -m");
    share_mem = shmat(shm_id, 0, 0); //映射到进程空间
    return 0;
}
```

编译并运行一下上述程序，可以看到已经创建了一块大小为 104857600bytes（100MB）的空间。



```
zhengyanww@ubuntu:~/Desktop/OS_comprehensiveTest2$ gcc file_system.c -o file_system
zhengyanww@ubuntu:~/Desktop/OS_comprehensiveTest2$ ./file_system
My shm_id is 32
----- Shared Memory Segments -----
key      shmid    owner    perms    bytes    nattch   status
0x00000000 2        gdm      600      16384    1        dest
0x00000000 5        gdm      600      2129920 2        dest
0x00000000 8        zhengyanww 600      16384    1        dest
0x00000000 17       zhengyanww 600      524288   2        dest
0x00000000 19       zhengyanww 600      1441792 2        dest
0x00000000 22       zhengyanww 600      524288   2        dest
0x00000000 24       zhengyanww 600      1769472 2        dest
0x00000000 25       zhengyanww 600      4128768 2        dest
0x00000000 26       zhengyanww 600      4358144 2        dest
0x00000000 29       zhengyanww 600      524288   2        dest
0x00000000 31       zhenovannww 600      4128768 2        dest
0x00000000 32       zhengyanww 666      104857600 0
```

2. 尝试自行设计一个 C 语言小程序，使用步骤 1 分配的 100M 空间（mmap or 共享内存），然后假设这 100M 空间为一个空白磁盘，设计一个简单的文件系统管理这个空白磁盘，给出

文件和目录管理的基本数据结构，并画出文件系统基本结构图，以及基本操作接口。

(1) 对于步骤 1 中申请的 100M 共享内存，模拟 FAT 文件系统，首先从 100M 中拿出 4MB 作为文件管理系统的核心，存放位图、Inode 数组等。



用于存放文件数据的 96MB，按照 4KB/页为单位划分，使用 1bit 对每一页是否被占用进行表示，这些信息会存放在 4MB 的文件系统管理中。

用于文件系统管理的 4MB，其结构可以如下图所示：



其中，FAT 表用于显示表达文件占用的数据页的跳转情况；Inode 记录当前文件类型（包含目录和可读写文件）以及父节点、左右兄弟节点、子节点的 inode 编号以及对应的 FCB 编号；FCB 记录文件名，每个文件和文件夹都占用一个 inode，与 inode 一一对应。

(2) 实验过程使用的常量声明：

```
//一个数据块的大小
#define BLOCK_SIZE 4096
//96MB对应的24576个数据块
#define BLOCK_NUM 24576
//每块对应1bit用于表示是否被占用，需要3072bytes
#define BLOCK_MAP_LENGTH 3072
//inode最多每块一个，对应数据块数量
#define INODE_MAP_LENGTH 3072
//考虑可能会有软、硬链接，可以多开一倍
#define FCB_MAP_LENGTH 6144
//系统管理核心占4MB，剩余96MB
#define CORE_SIZE (1<<22)
```

(3) 用到的基本数据结构：

```
struct Inode{
    int file_type; //文件类型，0-文件夹，1-文件
    int iblock; //文件数据块起始下标，在FAT表中进行跳转
    int size; //文件大小，可以在使用touch创建文件时指定大小
    int parent_inode_number; //父节点inode下标
    int child_inode_number; //第一个子节点下标
    int brother_inode_left_number; //左兄弟下标
    int brother_inode_right_number; //右兄弟下标
    int FCB_number; //文件对应FCB下标
};

struct FCB{
    int inode; //该FCB对应inode下标
    char FileName[12]; //文件名
};

union BlockMap{
    char block_map[BLOCK_MAP_LENGTH];
    char temp[BLOCK_SIZE];
};

struct InodeArray{
    char inode_map[INODE_MAP_LENGTH];
    struct Inode inode_array[BLOCK_NUM];
};
```

```

struct FCBArray{
    char FCB_map[FCB_MAP_LENGTH];
    struct FCB FCB_array[BLOCK_NUM << 1];
};

union InodeArrayBlock{
    struct InodeArray inode_array;
    char temp[790528];
};

union FCBArrayBlock{
    struct FCBArray fcb_array;
    char temp[794624];
};

struct SystemCore{
    int block_used; // 已使用的数据块,
    int block_rest; // 剩余空闲的数据块,
    union BlockMap bit_map; // 4KB
    union InodeArrayBlock inode_array; // 193 * 4KB
    union FCBArrayBlock FCB_array; // 194 * 4KB
    int FAT[24576]; // 96KB
};

union SystemCore_union{
    struct SystemCore core;
    char temp[1 << 22];
};

// 管理信号量的数据结构
union semun{
    int val;
    struct semid_ds *buf;
    unsigned short *array;
};

```

#### (4) 系统设置的一些基础功能接口：

```

//查看1byte即8bit中哪一位为0 (0表示可用)
int searchBit(char byte){}
//查找一个位图中第一位为0的bit的下标 (0表示可用)
int searchFirstBit(char *bytes, int length){}
//设置一个位图中第pos位的bit
void setBit(char *bytes, int pos, int val){}
//循环释放以iblock开始的数据块
void freeMemBlock(int iblock, union SystemCore_union *system_core){}
//申请block_num个数据块, 自动连接FAT表, 返回起始数据块下标
int allocMemBlock(int block_num, union SystemCore_union *system_core){}
//删除文件
void remove_file(struct Inode *inode, union SystemCore_union *system_core){}
//根据文件名、文件大小创建文件
void touch(struct FCB *dir_now, char file_name[], union SystemCore_union *system_core, int file_size){}
//在当前文件夹下创建名为dir_name的文件夹
void mkdir(struct FCB *dir_now, char dir_name[], union SystemCore_union *system_core){}
//删除文件夹
void remove_dir(struct Inode *inode, union SystemCore_union *system_core){}
//cd进入指定子目录
struct FCB* cd(struct FCB *dir_now, char file_name[], union SystemCore_union *system_core){}
//列出当前文件夹下子目录和文件的信息 (包括文件inode编号, 文件类型, 文件大小和文件名)
void ls(struct FCB *dir_now, union SystemCore_union *system_core){}
//读取文件内容, 嵌入data中
void readFile(struct Inode *inode, union SystemCore_union *system_core, char data[]){}
//把data中的内容写入文件对应的数据块
void writeFile(struct Inode *inode, union SystemCore_union *system_core, char data[]){}

// 重命名文件夹(文件同理, 改文件类型参数即可)
void rename_file(struct FCB *dir_now, char old_name[], char new_name[], union SystemCore_union *system_core){
    struct Inode *inode = searchInodeByName(dir_now, old_name, 1, system_core);
    //...
}
void rename_dir(struct FCB *dir_now, char old_name[], char new_name[], union SystemCore_union *system_core){
    struct Inode *inode = searchInodeByName(dir_now, old_name, 0, system_core);
    //...
}

```

3. 在步骤 1 的基础上实现部分文件操作接口操作, 创建目录 `mkdir`, 删除目录 `rmdir`, 修改名称, 创建文件 `open`, 修改文件。删除文件 `rm`, 查看文件系统目录结构 `ls`。

这里分成两部分进行说明：

#### 3.1. shell 程序实现文件操作接口

##### (1) 创建目录 `mkdir` 的实现

根据文件在文件管理系统中的存储特点, 创建一个新的目录就需要分配对应的 `inode` 和 `FCB`。首先设置循环获得当前文件夹下最后一个文件的 `inode` 编号。



```

// 获取当前文件夹下最后一个文件的inode编号
int last_child_inode_number = system_core->core.inode_array.inode_array.inode_array[dir_now->inode].child_inode_number;
if(last_child_inode_number != -1){ // 可能当前文件夹为空
while(system_core->core.inode_array.inode_array.inode_array[last_child_inode_number].brother_inode_right_number != -1){
last_child_inode_number = system_core->core.inode_array.inode_array.inode_array[last_child_inode_number].brother_inode_right_number;
}
}

```

然后为新文件夹获取可用的 inode, 根据上面得到的最后一个文件的 inode 编号对 inode 信息进行初始化。

```

// 获取新文件夹inode
int new_dir_inode_number = searchFirstBit(system_core->core.inode_array.inode_array.inode_map, INODE_MAP_LENGTH);
if(new_dir_inode_number == -1){
printf("FAIL: All inode have been used!\n");
return;
}
setBit(system_core->core.inode_array.inode_array.inode_map, new_dir_inode_number, 1);

// 初始化inode信息
struct Inode *new_dir_inode = &system_core->core.inode_array.inode_array.inode_array[new_dir_inode_number];
new_dir_inode->file_type = 0; // 文件夹类型为0, 文件类型为1
new_dir_inode->parent_inode_number = dir_now->inode;
new_dir_inode->brother_inode_left_number = -1;
new_dir_inode->brother_inode_right_number = -1;
new_dir_inode->child_inode_number = -1;
if(last_child_inode_number != -1){
new_dir_inode->brother_inode_left_number = last_child_inode_number;
system_core->core.inode_array.inode_array.inode_array[last_child_inode_number].brother_inode_right_number = new_dir_inode_number;
} else {
system_core->core.inode_array.inode_array.inode_array[dir_now->inode].child_inode_number = new_dir_inode_number;
}
}

```

获取可用的 FCB 并初始化。

```

// 获取FCB
int new_dir_fcb_number = searchFirstBit(system_core->core.FCB_array.fcb_array.FCB_map, FCB_MAP_LENGTH);
if(new_dir_fcb_number == -1){
setBit(system_core->core.inode_array.inode_array.inode_map, new_dir_inode_number, 0);
printf("FAIL: All FCB have been used!\n");
return;
}
setBit(system_core->core.FCB_array.fcb_array.FCB_map, new_dir_fcb_number, 1);
new_dir_inode->FCB_number = new_dir_fcb_number;

struct FCB *new_dir_fcb = &system_core->core.FCB_array.fcb_array.FCB_array[new_dir_fcb_number];
new_dir_fcb->inode = new_dir_inode_number;
strncpy(new_dir_fcb->FileName, dir_name, 12);
}

```

(2) 根据文件名、文件大小创建文件 touch 的实现

与创建目录 mkdir 方法实现思路相同, 主要是注意修改参数为文件对应的参数, 下面仅展示不同的部分:

开始创建之前, 判断当前剩余块是否够用。

```

void touch(struct FCB *dir_now, char file_name[], union SystemCore_union *system_core, int file_size){
int block_to_use = file_size / BLOCK_SIZE + 1;
if(block_to_use > system_core->core.block_rest){
printf("FATL: the file is too large.\n");
return;
}
system_core->core.block_used += block_to_use;
system_core->core.block_rest -= block_to_use;
}

```

初始化时, 文件类型为 1。

```

// 初始化inode信息
struct Inode *new_file_inode = &system_core->core.inode_array.inode_array.inode_array[new_file_inode_number];
new_file_inode->file_type = 1;
new_file_inode->parent_inode_number = dir_now->inode;
new_file_inode->brother_inode_left_number = -1;
new_file_inode->brother_inode_right_number = -1;
new_file_inode->child_inode_number = -1;
new_file_inode->size = file_size;
if(last_child_inode_number != -1){
setBit(system_core->core.inode_array.inode_array.inode_map, new_file_inode_number, 0);
}

```

新文件装入当前文件夹后, 分配数据块并清空。

```

// 分配数据块, 默认分配一块
new_file_inode->iblock = allocMemBlock(block_to_use, system_core);
writeFile(new_file_inode, system_core, ""); // 使用writeFile清空文件内容

```

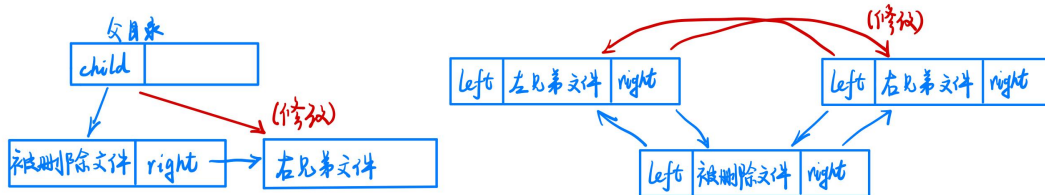
### (3) 删除文件 rm 的实现

借助位图和数据块的控制函数。首先从 inode 中获取 FCB 的编号和 inode 的编号，然后释放对应的数据块。

```
int FCB_number = inode->FCB_number;
struct FCB *fcb = &system_core->core.FCB_array.fcb_array.FCB_array[FCB_number];
int inode_number = fcb->inode;

freeMemBlock(inode->iblock, system_core); // 释放对应的数据块
```

接着，根据被删除的文件的左右兄弟节点是否存在来更新父目录的信息：如果被删除的是父目录的第一个子目录，则修改父目录的第一个子节点为被删除的右兄弟的 inode 号；如果不是，则需要更新被删除文件左兄弟的右兄弟，把节点号改成被删除文件右兄弟的 inode 号，同时更新这个右兄弟的左兄弟为被删除文件的左兄弟。



```
// 更新父目录的信息，将被删除的文件从父目录的子目录列表中删除
if(inode->brother_inode_left_number == -1){
    int father_inode_number = inode->parent_inode_number;
    struct Inode *father_inode = &system_core-
>core.inode_array.inode_array.inode_array[father_inode_number];
    father_inode->child_inode_number = inode->brother_inode_right_number;

    int brother_right_inode_number = inode->brother_inode_right_number;
    struct Inode *brother_right_inode = &system_core-
>core.inode_array.inode_array.inode_array[brother_right_inode_number];
    brother_right_inode->brother_inode_left_number = -1;
}
else
{
    int brother_inode_number = inode->brother_inode_left_number;
    struct Inode *brother_inode = &system_core-
>core.inode_array.inode_array.inode_array[brother_inode_number];
    brother_inode->brother_inode_right_number = inode->brother_inode_right_number;

    int brother_right_inode_number = inode->brother_inode_right_number;
    struct Inode *brother_right_inode = &system_core-
>core.inode_array.inode_array.inode_array[brother_right_inode_number];
    brother_right_inode->brother_inode_left_number = brother_inode_number;
}
```

最后使用 setBit 函数将被删除文件对应的 inode 和 FCB 标记为可用状态。

```
// 标记被删除的文件对应的inode和fcb为可用状态
setBit(system_core->core.inode_array.inode_array.inode_map, inode_number, 0);
setBit(system_core->core.FCB_array.fcb_array.FCB_map, FCB_number, 0);
```

### (4) 删除目录 rmdir 的实现

目录的删除与文件的删除的区别在于目录不一定为空。如果文件夹为空，直接删除即可。如果文件夹非空，则需要对其孩子进行遍历，如果孩子中存在空文件或空文件夹，同样可以直接删除，如果非空，则递归调用删除方法。

```
int FCB_number = inode->FCB_number;
struct FCB *fcb = &system_core->core.FCB_array.fcb_array.FCB_array[FCB_number];
int inode_number = fcb->inode;

while(inode->child_inode_number != -1){
    struct Inode *child_inode = &system_core->core.inode_array.inode_array[inode-
>child_inode_number];
    if(child_inode->file_type == 1){ // 孩子是文件直接删除
        remove_file(child_inode, system_core);
    } else {
        remove_dir(child_inode, system_core);
    }
}
```

文件夹清空后，接下来的操作与上述删除文件的操作相同。

(5) 修改文件名称（修改文件夹名操作方法相同，只需要修改文件类型的参数即可。其中，参数 0 表示文件夹，1 表示文件）

首先根据获取的文件名，使用 searchInodeByName 查找文件的 inode 号，如果查找不到则打

印错误信息。

```
struct Inode *inode = searchInodeByName(dir_now, old_name, 1, system_core);
//...
if(inode == NULL){
    printf("No such file!\n");
    return;
}
```

对于查找到的文件，获取其 FCB 号和 FCB 结构体指针，然后使用 `strncpy` 方法将新文件名复制到 FCB 结构体的 `FileName` 中，实现文件的重命名。

```
int FCB_number = inode->FCB_number;
struct FCB *fcb = &system_core->core.FCB_array.fcb_array.FCB_array[FCB_number];
strncpy(fcb->FileName, new_name, 12);
```

#### (6) 查看文件系统目录结构 `ls` 的实现

`ls` 命令的实现比较简单，获得当前目录的 `inode` 号，然后遍历当前目录的子节点即可。

```
//列出当前文件夹下子目录和文件的信息（包括文件inode编号，文件类型，文件大小和文件名）
void ls(struct FCB *dir_now, union SystemCore_union *system_core){
    struct Inode *dir_now_inode = &system_core->core.inode_array.inode_array[dir_now->inode];
    int child_inode_number = dir_now_inode->child_inode_number;
    while(child_inode_number != -1){
        struct Inode *child_inode = &system_core->core.inode_array.inode_array[child_inode_number];
        printf("inode:%d file_type:%d file_size:%d %s\n", child_inode_number, child_inode->file_type,
            child_inode->size, system_core->core.FCB_array.fcb_array.FCB_array[child_inode->FCB_number].FileName);
        child_inode_number = child_inode->brother_inode_right_number;
    }
}
```

#### (7) 进入某个指定目录 `cd` 的实现

首先从当前目录的 `inode` 中获得子目录的 `inode` 号，并将其进行存储。

```
struct Inode *dir_now_inode = &system_core->core.inode_array.inode_array[dir_now->inode];
int child_inode_number = dir_now_inode->child_inode_number;
```

然后设置循环，在循环体中，首先获得当前子目录的 `inode` 号和 FCB 号，然后判断当前子目录是否是普通文件并判断是否与指定进入的目录名相同。如果是则返回当前的 FCB 指针，否则继续对下一个子目录进行判断。

```
while(child_inode_number != -1){
    struct Inode *child_inode = &system_core->core.inode_array.inode_array[child_inode_number];
    int child_FCB_number = child_inode->FCB_number;
    struct FCB *child_FCB = &system_core->core.FCB_array.fcb_array.FCB_array[child_FCB_number];
    if(child_inode->file_type == 0 && strcmp(child_FCB->FileName, file_name, 12) == 0){
        return child_FCB;
    }
    child_inode_number = child_inode->brother_inode_right_number;
}
return NULL;
```

#### (8) 读/写方法的实现

**读方法：**

定义指针 `data_start` 指向数据块的起始位置（即跳过系统核心结构体的大小），初始化变量并获取系统核心结构体中的 `FAT` 数组指针。

```
char *data_start = (void *)system_core;
data_start += CORE_SIZE; // 定位到数据块开始位置
int count = 1, block_number = inode->iblock, write_pos = 0;
int *FAT = system_core->core.FAT;
char *read_p = data_start + block_number * BLOCK_SIZE;
```

通过 `read_p` 指针指向当前数据块起始位置，然后设置循环读取数据块内容直到读到结束符。最后将读取到的结束符添加到 `data` 数组中并返回读取到的数据。

```
while (*read_p != 0) { // 读取到结束符退出
    data[write_pos++] = *read_p;
    read_p++;
    count++;

    if (count == BLOCK_SIZE) { // 读取完一个数据块，跳转到下一个
        count = 1;
        block_number = FAT[block_number];
        read_p = data_start + block_number * BLOCK_SIZE;
    }
}
data[write_pos] = *read_p;
```

**写方法：**



首先通过 inode 结构体中的 size 变量计算需要的数据块数量，然后判断当前文件剩余的空间是否足够写入 data 数组中的内容。如果不够则重新分配数据块并更新文件大小为 data 数组的长度。

```
int block_num = inode->size / 4096+1;
// 如果这个文件的大小不足以装下data, 重新分配
if(block_num * 4096 <= strlen(data)){
    block_num = strlen(data) / 4096 + 1;
    int temp = allocMemBlock(block_num, system_core);
    if(temp == -1)
        return;
    freeMemBlock(inode->iblock, system_core);
    inode->iblock = temp;
    inode->size = strlen(data);
}
```

然后同样进行读方法中的初始化工作。

```
char *data_start = (void *)system_core;
data_start += CORE_SIZE;
int read_pos = 0, count = 1, block_number = inode->iblock;
int *FAT = system_core->core.FAT;
char *write_p = data_start + block_number * BLOCK_SIZE;
```

最后设置循环将 data 数组中的内容写入数据块，直到读到结束符。

```
while(data[read_pos] != 0){
    *write_p = data[read_pos];
    write_p++;
    read_pos++;
    count++;

    if(count == BLOCK_SIZE){
        count = 1;
        block_number = FAT[block_number];
        write_p = data_start + block_number * BLOCK_SIZE;
    }
}
*write_p = 0;
if(inode->size < strlen(data)){
    inode->size = strlen(data);
}
```

### 3.2. 编写程序，实现在文件系统中使用上述操作

#### (1) 预处理

编写一个 showHello 方法打印一些使用提示语

```
void showHello(){
    printf("=====\n");
    printf("Welcome to my file system.\n");
    printf("You can use these orders: help, exit, ls, touch, mkdir, cd, open, rm, rn.\n");
    printf("Now you can use 'help' to see the usage of all the orders.\n");
    printf("=====\n");
}
```

编写 help 方法对可操作指令的功能进行打印

```
void do_help() {
    printf("\n\n");
    printf("ls ----- show folders and files in this folder.\n");
    printf("cd [../ or child folder name] -- get to target.\n");
    printf("mkdir [child folder name] ----- create folder.\n");
    printf("touch [filename] (filesize) ----- create file by size, default 0 bytes.\n");
    printf("open [r / w] [filename] ----- read / write the file.\n");
    printf("exit ----- exit the file system.\n");
    printf("rm [-r] name ----- use -r to remove a folder.\n");
    printf("rn [f / d] old_name new_name --- f means file, d means dir, rename the target.\n");
    printf("\n\n");
}
```

编写循环对输入的指令进行判断，如果是 exit 则打印相应的信息并退出循环，否则执行相应的操作



```

char order[100];
while(1){
    printPath(dir_now, system_core);
    fgets(order, 100, stdin);
    order[strlen(order)-1] = 0;
    if(strncmp(order, "exit", 4) == 0){
        printf("bye~\n");
        break;
    }
    else if(strncmp(order, "ls", 2) == 0){
        ls(dir_now, system_core);
    }
    else if(strncmp(order, "mkdir", 5) == 0 )
        do_mkdir(dir_now, order, system_core);
    else if (strncmp(order, "cd", 2) == 0)
        dir_now = do_cd(dir_now, order, system_core);
    else if (strncmp(order, "touch", 5) == 0)
        do_touch(dir_now, order, system_core);
    else if (strncmp(order, "open", 4) == 0)
        do_open(dir_now, order, system_core);
    else if (strncmp(order, "rm", 2) == 0)
        do_remove(dir_now, order, system_core);
    else if (strncmp(order, "rn", 2) == 0)
        do_rename(dir_now, order, system_core);
    else
        do_help();
}
shmdt(share_mem); // 卸载共享内存
return 0;

```

上述方法中，除了 ls 操作之外，其他操作后面包含其他内容，因此需要另外设置方法对这一部分进行判断。

(2) 首先判断在指令后，是否存在表示其他内容的字符串。如果在遍历到空格之前就已经遍历到字符串的结尾，则输入不合法，输出 Param error 并返回。

```

char *p = order;
while (*p != '\0') {
    if (*p == 0) {
        printf("Param error.\n");
        return;
    }
    p++;
}

```

接下来的判断具体到每个方法：

①do\_mkdir: 判断命令后的字符串所代表的文件名长度是否超过限制长度、所创建的文件夹是否已存在。通过不合理输入的筛选后调用 mkdir 方法实现操作。

```

if (strlen(p) >= 12) {
    printf("Folder name too long. Please reset it less than 12 characters.\n");
    return;
}
if (searchInodeByName(dir_now, p, 0, system_core) != NULL) {
    printf("Folder duplicate name!\n");
    return;
}
mkdir(dir_now, p, system_core);

```

②do\_cd: 判断要进入的文件夹名称是否是 “../”，如果是，则通过当前目录的 inode 号找到父目录的 inode 号和对应的 FCB 号，如果不是，则调用 cd 进入指定的文件夹并获取返回值。

```

if (strncmp(p, "../", 3) == 0) {
    int father_inode_number = system_core->core.inode_array.inode_array.inode_array[dir_now->inode].parent_inode_number;
    int father_FCB_number = system_core->core.inode_array.inode_array[father_inode_number].FCB_number;
    return &system_core->core.FCB_array.fcb_array.FCB_array[father_FCB_number];
}
else {
    struct FCB* res = cd(dir_now, p, system_core);
    if (res == NULL) {
        printf("No such dir!\n");
        return dir_now;
    }
    return res;
}
}

```

③do\_touch: 判断文件名是否超过限定长度，不超过的话接着查找是否存在同名文件，如果不存在则使用 touch 方法进行创建。这里和上面 do\_mkdir 一样，不做展示。

④do\_open: 包含两种参数——r: 读操作，w: 写操作。所以首先需要在这个方法中对输入参数进行判断。

```

if (*p == 'r')
    do_read(dir_now, p, system_core);
else if (*p == 'w')
    do_write(dir_now, p, system_core);

```

在写方法中:

```

struct Inode *inode = searchInodeByName(dir_now, p, 1, system_core); // 获取该文件名对应inode
if (inode == NULL) {
    printf("No such file!\n");
    return;
}
struct FCB *fcb = &system_core->core.FCB_array.fcb_array.FCB_array[inode->FCB_number];
int inode_number = fcb->inode;
wait(inode_number); // 等待信号量

printf("Please write the file. Write a blank line means finish:\n"); // 输入文件新内容
char line[4096], data[1 << 20] = { 0 };
while (1) {
    fgets(line, 4096, stdin);
    if (strcmp(line, "\n", 1) == 0)
        break;
    strncat(data, line, 4096);
}
printf("Writing finish.\n");
writeFile(inode, system_core, data);
notify(inode_number); // 释放信号量

```

在读方法中:

```

struct Inode *inode = searchInodeByName(dir_now, p, 1, system_core); // 获取该文件名对应inode
if (inode == NULL) {
    printf("No such file!\n");
    return;
}
else {
    struct FCB *fcb = &system_core->core.FCB_array.fcb_array.FCB_array[inode->FCB_number];
    int inode_number = fcb->inode;
    wait(inode_number); // 等待信号量
    notify(inode_number); // 马上释放, 确保多个进程可以同时读取

    char* data = open_file_read(inode, system_core); // 读取内容并打印
    printf("-----\n");
    printf("%s", data);
    printf("-----\n");
    free(data);
}

```

⑤do\_remove: 这里同样包含两种参数——无参时: 删除文件; -r: 删除目录。  
对输入指令参数的判断。

```

char *p1 = p;
while (*p1 != ' ') {
    if (*p1 == 0) {
        do_remove_file(dir_now, p, system_core);
        return;
    }
    p1++;
}
*p1 = 0;
p1++;
if (strcmp(p, "-r", 2) == 0)
    do_remove_dir(dir_now, p1, system_core);
else
    printf("Param error.\n");

```

删除文件:

```

struct Inode *inode = searchInodeByName(dir_now, name, 1, system_core);
if (inode == NULL) {
    printf("No such file!\n");
    return;
}
remove_file(inode, system_core);

```

删除目录:

```

struct Inode *inode = searchInodeByName(dir_now, name, 0, system_core);
if (inode == NULL) {
    printf("No such folder!\n");
    return;
}
remove_dir(inode, system_core);

```

⑥do\_rename: 这里同样包含两种参数——f: 重命名文件; d: 重命名目录。  
对输入指令参数的判断, 然后直接调用 shell 程序中的方法。

```

if (*p == 'f')
    rename_file(dir_now, p1, p2, system_core);
else if (*p == 'd')
    rename_dir(dir_now, p1, p2, system_core);
else
    printf("Param error.\n");

```

4. 参考进程同步的相关章节，通过信号量机制实现多个终端对上述文件系统的互斥访问，系统中的一个文件允许多个进程读，不允许写操作；或者只允许一个写操作，不允许读。在上述功能的基础上添加对文件的读写操作。

（1）在步骤 1 申请内存的程序（这个程序类似一个生产者进程，那么使用系统的程序就可以看成一个消费者进程）中创建信号量集，在步骤 2 中我们提到文件和文件夹与 inode 一一对应，那么在创建信号量集的时候，使其数量和 inode 信号量相同即可。

```

sem_id = semget(0x0000, BLOCK_NUM, 0660 | IPC_CREAT); // 创建信号量集
union semun sem_union;
sem_union.val = 1;
for(int i=0;i<BLOCK_NUM;i++){ // 初始化信号量
    semctl(sem_id, i, SETVAL, sem_union);
}

```

（2）在实现操作的 shell 文件中封装 P/V 操作的函数

```

// inode_number即信号量集中该文件对应的信号量的下标，这里等待信号量为1，获取到后上锁，信号量置0
void wait(int inode_number) {
    struct sembuf sem_b;
    sem_b.sem_num = inode_number;
    sem_b.sem_op = -1;
    sem_b.sem_flg = SEM_UNDO;
    semop(sem_id, &sem_b, 1);
}

// 信号量置1
void notify(int inode_number) {
    struct sembuf sem_b;
    sem_b.sem_num = inode_number;
    sem_b.sem_op = 1;
    sem_b.sem_flg = SEM_UNDO;
    semop(sem_id, &sem_b, 1);
}

```

四、实验结论：（提供运行结果，对结果进行探讨、分析、评价，并提出结论性意见和改进想法）

1. 打开终端，尝试运行基本操作：

```
zhengyanwei_2020151022@ubuntu: ~/Desktop/OS_comprehensiveTest2$ ./file_system
startup finish!

zhengyanwei_2020151022@ubuntu: ~/Desktop/OS_compre...
zhengyanwei_2020151022@ubuntu:~$ cd Desktop/OS_omprehensiveTest2
bash: cd: Desktop/OS_omprehensiveTest2: No such file or directory
zhengyanwei_2020151022@ubuntu:~$ cd Desktop/OS_comprehensiveTest2
zhengyanwei_2020151022@ubuntu:~/Desktop/OS_comprehensiveTest2$ ./use_system
Loading share memory...
Finish!
=====
Welcome to my file system.
You can use these orders: help, exit, ls, touch, mkdir, cd, open, rm, rn.
Now you can use 'help' to see the usage of all the orders.
=====
[root/]:
```

(1) 输入 help 打印信息：

```
[root/]: help

ls ----- show folders and files in this folder.
cd [./ or child folder name] -- get to target.
mkdir [child folder name] ----- create folder.
touch [filename] (filesize) ---- create file by size, default 0 bytes.
open [r / w] [filename] ----- read / write the file.
exit ----- exit the file system.
rm [-r] name ----- use -r to remove a folder.
rn [f / d] old_name new_name --- f means file, d means dir, rename the target.

[root/]:
```

(2) 创建目录 mkdir:

```
[root/]: ls
[root/]: mkdir dir1
[root/]: mkdir dir2
[root/]: ls
inode:1 file_type:0 file_size:0 dir1
inode:2 file_type:0 file_size:0 dir2
[root/]:
```

(3) 修改目录名 rn d old\_folder\_name new\_folder\_name:

```
[root/]: rn d_dir1 new_dir1
[root/]: ls
inode:1 file_type:0 file_size:0 new_dir1
inode:2 file_type:0 file_size:0 dir2
[root/]:
```

(4) 删除目录 rm -r folder\_name:

```
[root/]: rm -r dir2
[root/]: ls
inode:1 file_type:0 file_size:0 new_dir1
[root/]:
```

(5) 进入目录 new\_dir1:

```
[root/]: cd new_dir1
[root/new_dir1/]:
```

(6) 创建文件 touch:

```
[root/new_dir1/]: touch file1
[root/new_dir1/]: touch file2
[root/new_dir1/]: ls
inode:2 file_type:1 file_size:0 file1
inode:3 file_type:1 file_size:0 file2
[root/new_dir1/]:
```

(7) 修改文件名:

```
[root/new_dir1/]: rn f file2 new_file2
[root/new_dir1/]: ls
inode:2 file_type:1 file_size:0 file1
inode:3 file_type:1 file_size:0 new_file2
[root/new_dir1/]:
```

(8) 读写操作:



写文件:

```
[root/new_dir1/]: open w file1
Please write the file. Write a blank line means finish:
Hello from file1 in folder dir1!

Writing finish.
[root/new_dir1/]:
```

读文件:

```
[root/new_dir1/]: open r file1
-----
Hello from file1 in folder dir1!
-----
[root/new_dir1/]:
```

(9) 删除文件:

```
[root/new_dir1/]: ls
inode:2 file_type:1 file_size:33 file1
inode:3 file_type:1 file_size:0 new_file2
[root/new_dir1/]: rm new_file2
[root/new_dir1/]: ls
inode:2 file_type:1 file_size:33 file1
[root/new_dir1/]:
```

(10) 查看文件系统目录结构 ls:

通过上面的截图可以看到该操作已经实现,打印信息为:

“inode:inode 号 file\_type:文件类型(0-目录, 1-文件) file\_size:文件大小 文件名”

(11) 输入 exit 退出:

```
[root/new_dir1/]: exit
bye~
zhengyanwei_2020151022@ubuntu: ~/Desktop/OS_comprehensiveTest2$
```

2. 打开新的窗口, 尝试同时读或写:

(1) 尝试同时读:

```
12345 zhengyanwei_2020151022@ubuntu: ~/Desktop/OS_comprehensiveTest2$
Writing finish.
[root/new_dir1/]: open r file1 [root/new_dir1/]: open r file1
-----
12345 12345
-----
[root/new_dir1/]: [root/new_dir1/]:
```

两个进程同时读。

(2) 尝试写的同时进行读操作:

```
[root/new_dir1/]: open w file1
Please write the file. Write a blank line means finish:
12345
-----
1234
-----
[root/new_dir1/]: open r file1
-----
Error: file 3 = 0, 3 = BLOCK MIN: 3, 3 = 11 318542825inode
-----
```

当上面的进程仍在执行写操作时, 下面的进程执行读操作失败。

**五、实验体会:** (根据自己情况填写)

1. 本次实验较难, 实验前查询了很多资料。首先是针对文件管理的分配, 对 FAT 文件系统进行了学习。然后深入理解 linux 文件操作的基本方法, 确定数据结构以及指令执行的逻辑, 以实现在自己的文件管理系统中能够具有相同的执行效果。

2. 在实验中, 通过实验二对生产者消费者进程、进程之间的同步机制, 通过综合实验一对 shell 程序, 还有共享内存的申请、分配方法都进行了复习, 学会结合各方面知识实现一个文件系统。

注：“指导教师批阅意见”栏请单独放置一页

指导教师批阅意见:

成绩评定：

指导教师签字:

年 月 日

备注: