

# 同济大学计算机系

## 操作系统课程设计报告



学 号 1553534

姓 名 李帅

专 业 计科

授课老师 方钰

## 一. 需求分析

- (1) **题目：**构建二级文件模拟系统
- (2) **题目说明：**使用一个普通的大文件（如 c:\myDisk.img ，称之为一级文件）来模拟 UNIX V6++ 的一个文件卷。

S	Inode 区	文件数据区
---	---------	-------

一个文件卷实际上就是一张逻辑磁盘，磁盘存储的信息以块为单位。每块 512 字节。

- (3) **磁盘文件结构：**定义自己的磁盘文件结构，SuperBlock 结构，磁盘 Inode 节点结构，包括：索引结构，及：逻辑块号到物理块号的映射，磁盘 Inode 节点的分配与回收算法设计与实现，文件数据区的分配与回收算法设计与实现。
- (4) **文件目录结构：**文件目录结构，目录检索算法的设计与实现。
- (5) **文件打开结构：**选做
- (6) **磁盘高速缓存：**选做
- (7) **文件操作接口：**

```
void fformat();      格式化文件卷
void ls();          列目录
int  fopen(char *name, int mode);  打开文件
void fclose(int fd);      关闭文件
int  fread(int fd, char *buffer, int length);  读文件
int  fwrite(int fd, char *buffer, int length);  写文件
int  fseek(int fd, int position);      定位文件读写指针
int  fcreat(char *name, int mode);  新建文件
int  fdelete(char *name);      删除文件
```

- (8) **主程序：**  
初始化文件卷，读入 SuperBlock;  
图形界面或者命令行方式，等待用户输入;  
根据用户不同的输入，返回结果。
- (9) **主程序输入及其范围：**  
主程序采用**命令行输入**，输入相关指令，可以操作文件系统  
指令范围是：

指令	简介
exit	退出
help	指令介绍
test	实例分析
write	向文件写内容
read	读取文件内容
ls	显示目录内容
mk	创建文件
mkdir	创建目录

rm	删除文件或者目录
fin	从外部系统输入
fout	输出到外部系统
cd	更改路径

### (10) 输出形式:

```
=====文件系统 (by-1553534-李帅)=====
exit      退出
help      指令简介
test      实例分析
write     [文件名] [内容字符串] 向文件写内容  示例: write /A/B Hello World
read      [文件名]              读文件的内容  示例: write B Hello World
ls        [目录名]              显示目录的内容 示例: read /A/B
mk        [文件名]              创建文件      示例: read B
mkdir     [目录名]              创建目录      示例: ls /A
rm        [文件或目录名]        删除文件或目录 示例: mk /A/B
fin       [外部文件名] [内部文件名] 从外部系统输入 示例: mk B
fout      [内部文件名] [外部文件名] 输出到外部系统 示例: mkdir /A
cd        [目录名或绝对路径或'..'] 更改工作路径 示例: mkdir A
          示例: rm /A/B
          示例: rm B
          示例: fin input.txt /A/B
          示例: fin input.txt B
          示例: fout /A/B output.txt
          示例: fout B output.txt
          示例: cd /A
          示例: cd A
          示例: cd ..
=====文件系统 (by-1553534-李帅)=====
filesystem(1553534-李帅) />_
```

输出形式按照如上设计

### (11) 程序功能

初始化文件卷，读入 SuperBlock;

命令行方式，等待用户输入;

根据用户不同的输入，返回结果。

## 二. 概要设计

### (1) 主要数据类型的定义

数据类型基本根据 UnixV6++源码设计成结构体，考虑到系统比较小，所以方法直接采用外部函数，保留其命名与参数设定。

#### 1) 外存索引节点(DiskInode)的定义

```
/* 外存索引节点(DiskInode)的定义
 * 外存Inode位于文件存储设备上的
 * 外存Inode区中，每个文件有唯一对应
 * 的外存Inode，其作用是记录了该文件
 * 对应的控制信息。
 * 外存Inode中许多字段和内存Inode中字段
 * 相对应，外存Inode对象长度为64字节，
 * 每个磁盘块可以存放512/64 = 8个外存Inode
 */

struct inode_d_type{
    unsigned int d_mode; /* 状态的标志位，定义见enum INodeFlag */
    int d_nlink; /* 文件链接计数，即该文件在目录树中不同路径名的数量 */
    short d_uid; /* 文件所有者的用户标识数 */
    short d_gid; /* 文件所有者的组标识数 */
    int d_size; /* 文件大小，字节为单位 */
    int d_addr[10]; /* 用于文件逻辑块好和物理块好转换的基本索引表 */
    int d_atime; /* 最后访问时间 */
    int d_mtime; /* 最后修改时间 */
};
```

#### 2) SuperBlock 的定义

```
//superblock的结构体
struct sb_type{
    int s_isize; /* 外存Inode区占用的盘块数 */
    int s_fsize; /* 盘块总数 */

    fbl_type freebl; /* free block list */
    fil_type freeil; /* free inode list */

    int s_flock; /* 封锁空闲盘块索引表标志 */
    int s_ilock; /* 封锁空闲Inode表标志 */

    int s_fmod; /* 内存中super block副本被修改标志, 意味着需要更新外存对应的Super Block */
    int s_ronly; /* 本文件系统只能读出 */
    int s_time; /* 最近一次更新时间 */
    int padding[27]; /* 填充使SuperBlock块大小等于1024字节, 占据2个扇区 */
};
```

### 3) DirectoryEntry 的定义

```
struct de_type{
    int m_ino;
    char m_name[DIRENTRYSIZE - sizeof(int)];
};
```

### 4) 内存索引节点(INode)的定义

```
/*
 * 内存索引节点(INode)的定义
 * 系统中每一个打开的文件、当前访问目录、
 * 挂载的子文件系统都对应唯一的内存inode。
 * 每个内存inode通过外存inode所在存储设备的设备号(i_dev)
 * 以及该设备外存inode区中的编号(i_number)来确定
 * 其对应的外存inode。
 */
struct inode_type{
    unsigned int i_flag; /* 状态的标志位, 定义见enum INodeFlag */
    unsigned int i_mode; /* 文件工作方式信息 */
    int i_size; /* 文件大小, 字节为单位 */
    int i_addr[10]; /* 用于文件逻辑块好和物理块好转换的基本索引表 */
    int i_number; /* 外存inode区中的编号 */
    int i_count; /* 引用计数 */
    int i_offset;
    int i_nlink; /* 文件联结计数, 即该文件在目录树中不同路径名的数量 */
    short i_dev; /* 外存inode所在存储设备的设备号 */
    short i_uid; /* 文件所有者的用户标识数 */
    short i_gid; /* 文件所有者的组标识数 */
    int i_lastr; /* 存放最近一次读取文件的逻辑块号, 用于判断是否需要预读 */
};
```

### 5) 缓存控制块 buf 定义

```

/*
 * 缓存控制块buf定义
 * 记录了相应缓存的使用情况等信息;
 * 同时兼任I/O请求块, 记录该缓存
 * 相关的I/O请求和执行结果。
 */

struct buf_type{
    unsigned int b_flags; /* 缓存控制块标志位 */
    int b_blkno; /* 磁盘逻辑块号 */
    /* 缓存控制块队列勾连指针 */
    buf_type *b_forw;
    buf_type *b_back;
    buf_type *av_forw;
    buf_type *av_back;
    char data[BLOCKSIZE];
    short b_dev; /* 主、次设备号, 其中高8位是主设备号, 低8位是次设备号 */
    int b_wcount; /* 需传送的字节数 */
    unsigned char* b_addr; /* 指向该缓存控制块所管理的缓冲区的首地址 */
    int b_error; /* I/O出错时信息 */
    int b_resid; /* I/O出错时尚未传送的剩余字节数 */
};

enum BuffFlag /* b_flags中标志位 */
{
    B_WRITE = 0x1, /* 写操作, 将缓存中的信息写到硬盘上去 */
    B_READ = 0x2, /* 读操作, 从盘读取信息到缓存中 */
    B_DONE = 0x4, /* I/O操作结束 */
    B_ERROR = 0x8, /* I/O因出错而终止 */
    B_BUSY = 0x10, /* 相应缓存正在使用中 */
    B_WANTED = 0x20, /* 有进程正在等待使用该buf管理的资源, 清除_BUSY标志时, 要唤醒这种进程 */
    B_ASYNC = 0x40, /* 异步I/O, 不需要等待其结束 */
    B_DELWRI = 0x80 /* 延迟写, 在相应缓存要移做他用时, 再将其内容写到相应块设备上 */
};

```

## 6) Kernel 类的定义

```

/* Kernel类用于封装所有内核相关的全局类实例对象,
 * 例如PageManager, ProcessManager等。
 */
/* Kernel类在内存中为单体模式, 保证内核中封装各内核
 * 模块的对象都只有一个副本。
 */

struct kernel_type{
    FILE *diskfile; /* disk文件*/
    sb_type superBlock; /*superblock*/
    buf_type buffers[BUFCOUNT]; /*缓存控制*/

    buf_type *freeBufHead; /*空闲缓存链表的头指针*/
    buf_type *freeBufTail; /*空闲缓存链表的尾指针*/
    int freeBufCount; /*空闲缓存控制块的个数*/
    inode_type openFiles[FILECOUNT]; /*打开的inode文件*/
    int openFileCount; /*计数信号*/
};

```

## 7) 参数设定

```

enum BuffFlag /* b_flags中标志位 */
{
    B_WRITE = 0x1, /* 写操作, 将缓存中的信息写到硬盘上去 */
    B_READ = 0x2, /* 读操作, 从盘读取信息到缓存中 */
    B_DONE = 0x4, /* I/O操作结束 */
    B_ERROR = 0x8, /* I/O因出错而终止 */
    B_BUSY = 0x10, /* 相应缓存正在使用中 */
    B_WANTED = 0x20, /* 有进程正在等待使用该buf管理的资源, 清除_BUSY标志时, 要唤醒这种进程 */
    B_ASYNC = 0x40, /* 异步I/O, 不需要等待其结束 */
    B_DELWRI = 0x80 /* 延迟写, 在相应缓存要移做他用时, 再将其内容写到相应块设备上 */
};

```

```
9
0 static const unsigned int IALLOC = 0x8000; /* 文件被使用 */
1 static const unsigned int IFMT = 0x6000;
2 static const unsigned int IFDIR = 0x4000;
3 static const unsigned int IFCHR = 0x2000;
4 static const unsigned int IFBLK = 0x6000;
5 static const unsigned int ILARG = 0x1000;
6 static const unsigned int ISUID = 0x800;
7 static const unsigned int ISGID = 0x400;
8 static const unsigned int ISVTX = 0x200;
9 static const unsigned int IREAD = 0x100;
0 static const unsigned int IWRITE = 0x80;
1 static const unsigned int IEXEC = 0x40;
2 static const unsigned int ITEXT = 0x20;
3 static const unsigned int IWANT = 0x10;
4 static const unsigned int IMOUNT = 0x8;
5 static const unsigned int IACC = 0x4;
6 static const unsigned int IUPD = 0x2;
7 static const unsigned int ILOCK = 0x1;
8
```

一级文件：‘myDisk.img’

磁盘 inode 节点最多 2048 个

磁盘中存储的信息以块为单位。每块 512 字节。

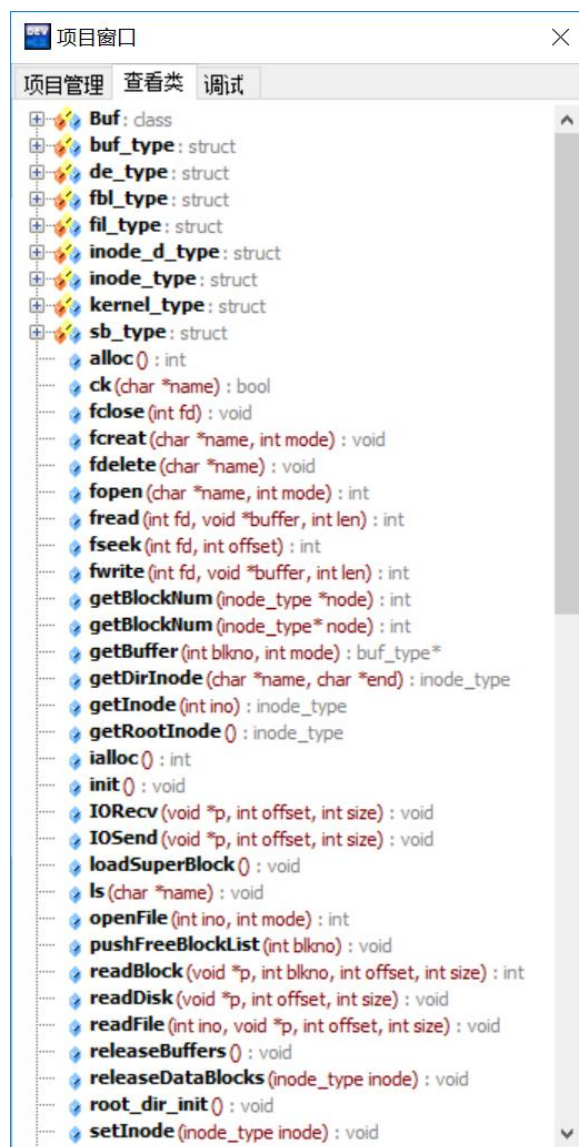
最多同时打开的文件个数设置为 20

缓存控制块的个数为 23

```
static const char *DISKFILENAME = "myDisk.img";
static const int DISKINODECAPACITY = 2048;
static const int BLOCKSIZE = 512;
static const int DATABLOCKCOUNT = 32768;
const int FILECOUNT = 20;
const int BUFCOUNT = 23;
```

## (2) 主程序流程

### (1) 类图

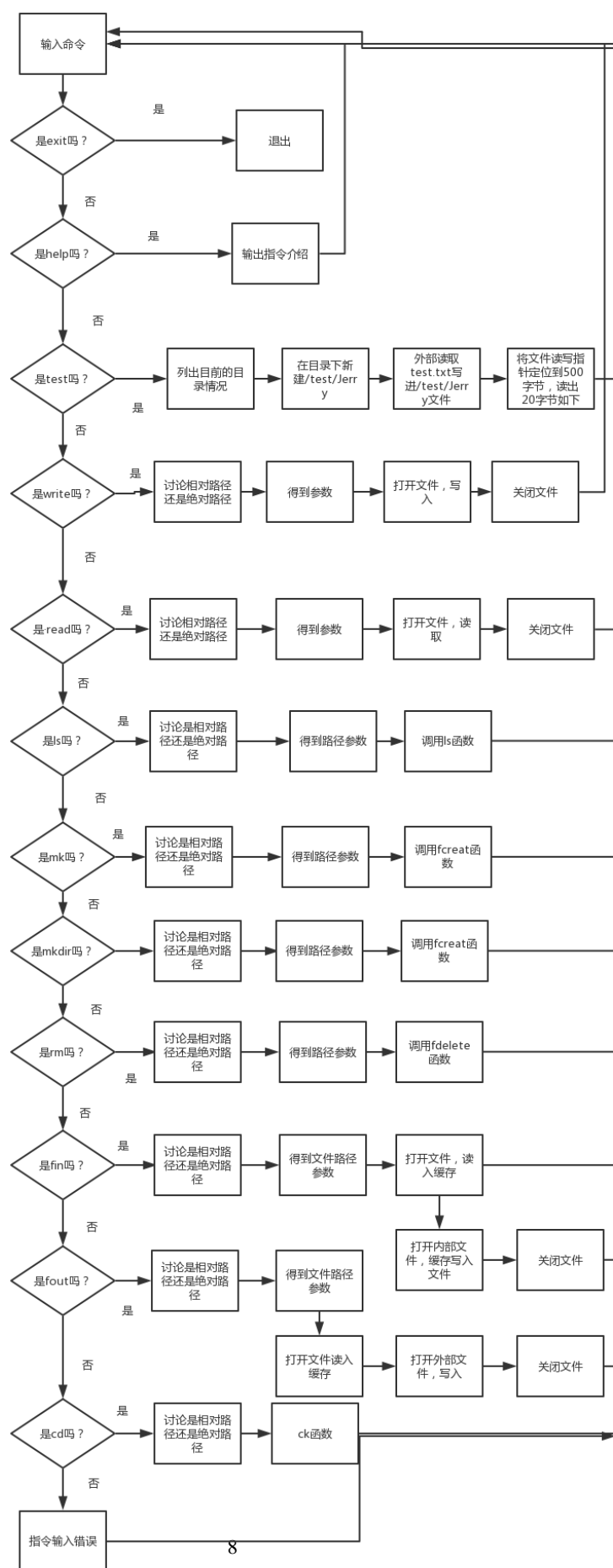


(2) 主程序流程图

main.cpp 是用户输入控制源文件

filesystem.h 是文件系统的结构体，参数，函数头文件

主程序流程图如下：

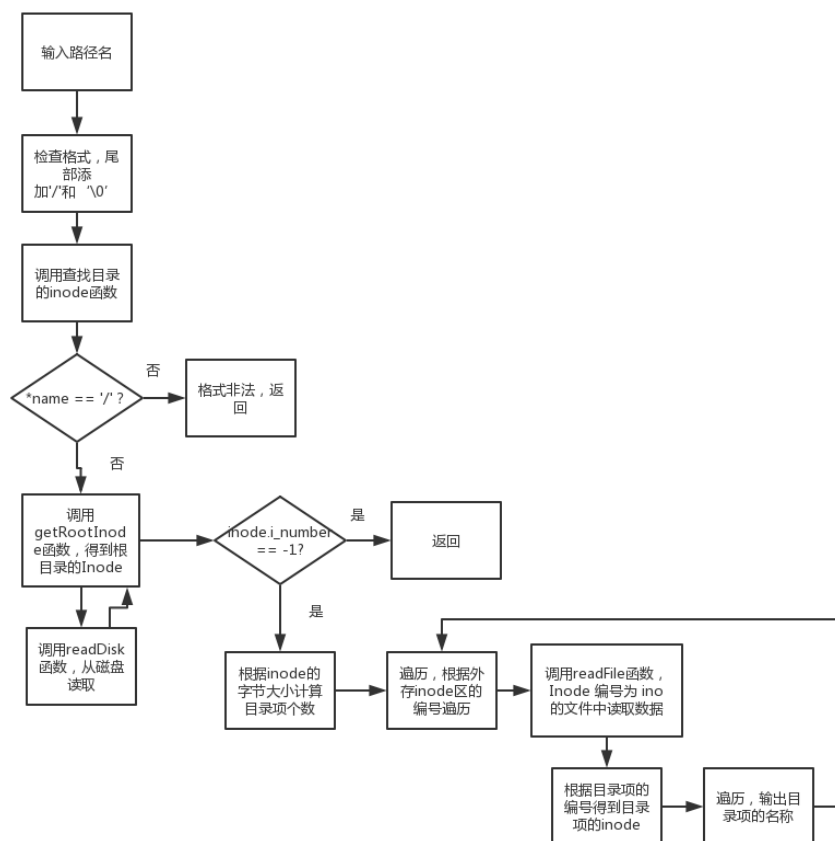




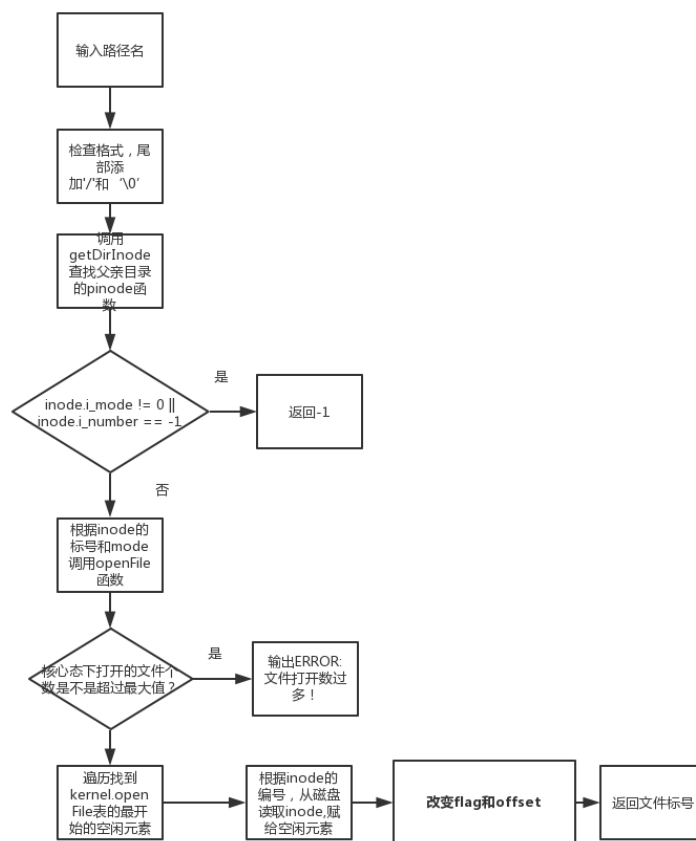
### 三. 详细设计

主要函数的流程调用关系图如下:

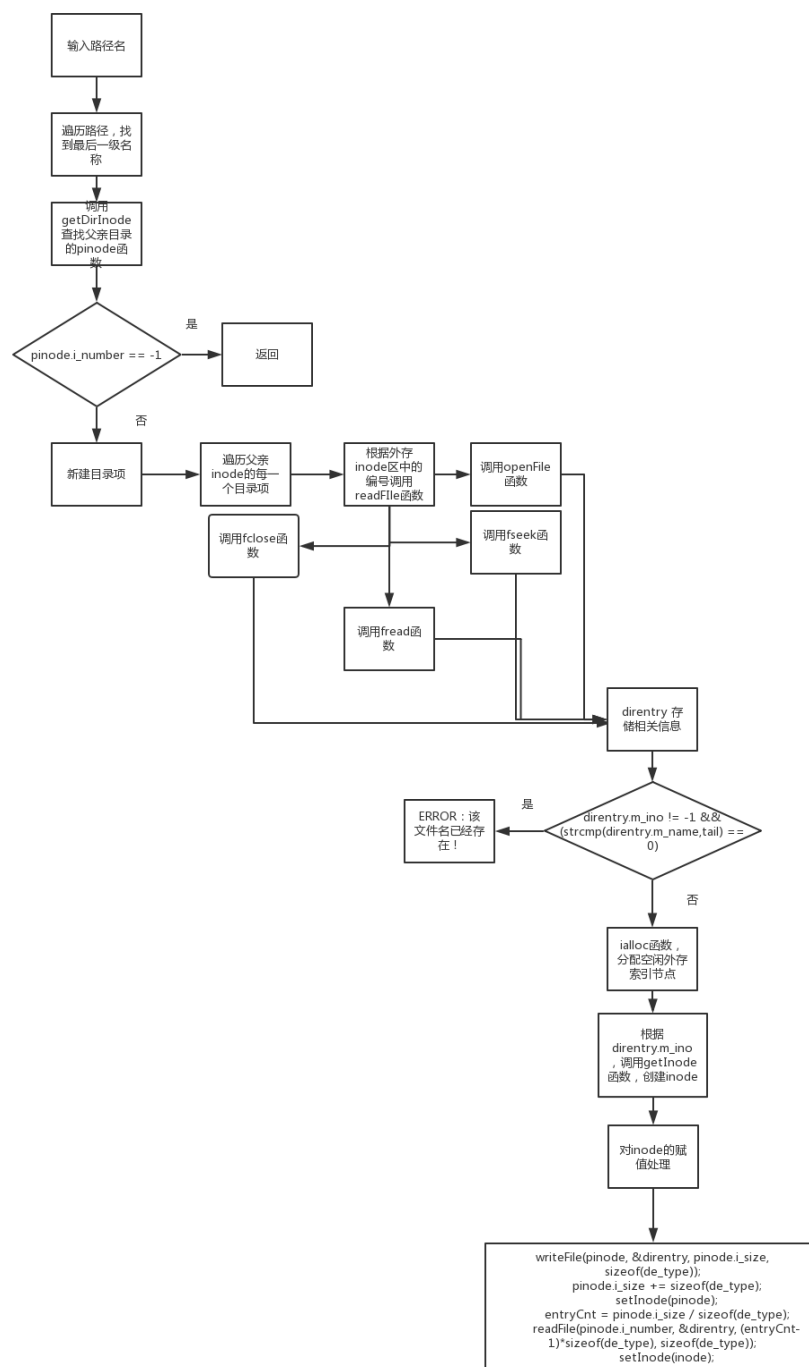
(1) `void ls(char *name);` //列出 `name` 路径下的所有文件



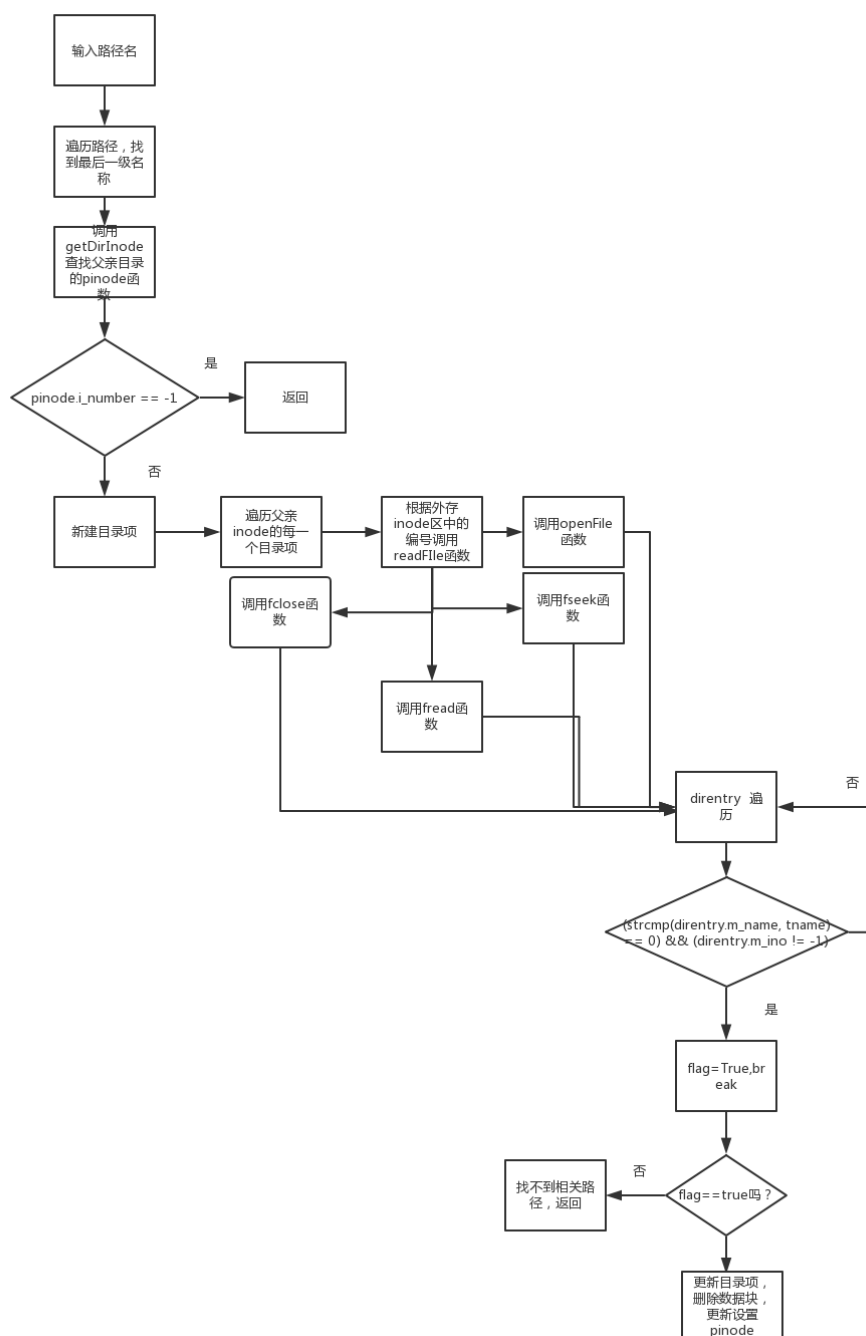
(2) `int fopen(char *name, int mode);` //以 `mode` 模式打开 `name` 文件



(3) void fcreat(char \*name, int mode); //以 mode 模式创建 name 文件



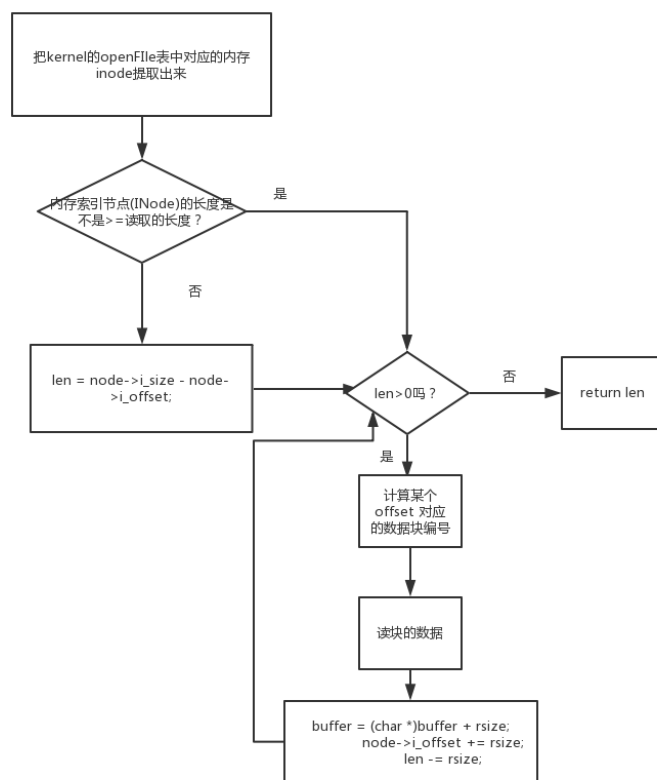
(4) void fdelete(char \*name); //删除 name 文件



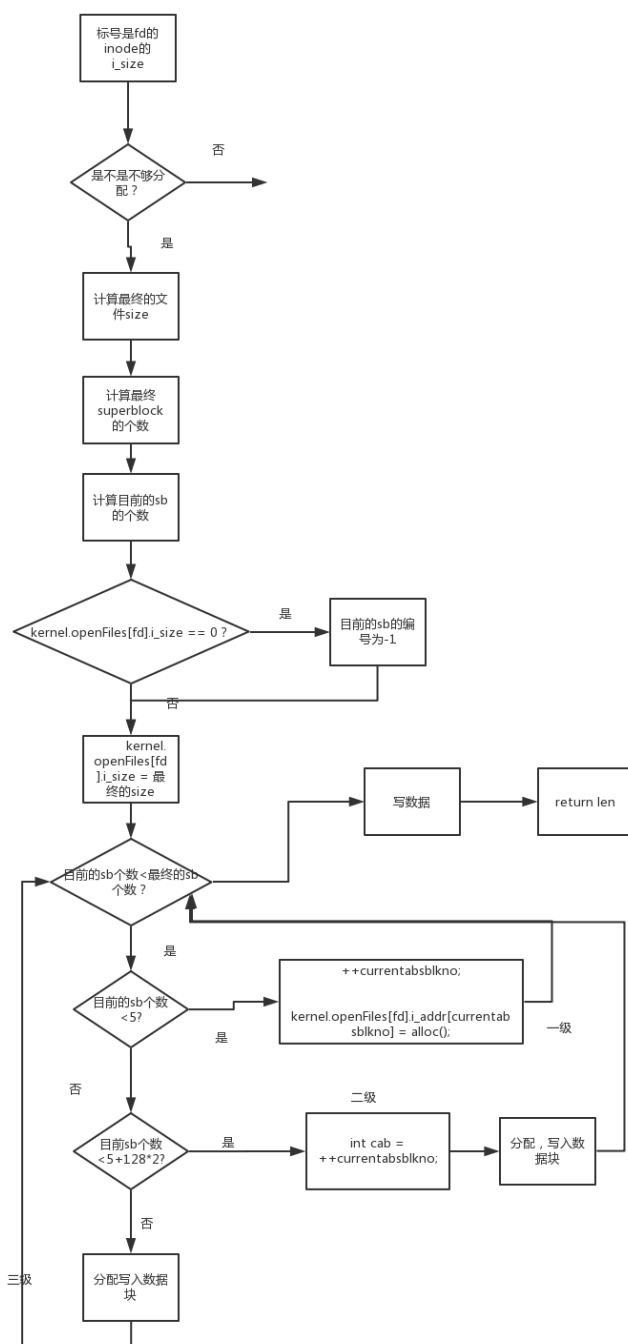
(5) `int fseek(int fd, int offset);`//移动文件指针

把 kernel 的对应的 openFiles 类的 `i_offset` 改成 `offset` 即可。

(6) `int fread(int fd, void *buffer, int len);`//从文件中读取



(7) `int fwrite(int fd, void *buffer, int len);`//向文件中写入



(8) void fclose(int fd); //关闭文件

更改 kernel.openFiles[fd].i\_flag, 以及更新 inode 即可。

## 四. 调试分析

(1) 最后的测试

```
S:\DEVCPP-projects\osfilesystem\os.exe
=====文件系统 (by-1553534-李帅) =====
filesystem(1553534-李帅) />ls
/home
/user
/DirA
/DirB
/DirC
filesystem(1553534-李帅) />mkdir test
filesystem(1553534-李帅) />ls
/home
/user
/DirA
/DirB
/DirC
/test
filesystem(1553534-李帅) />mk test/h
filesystem(1553534-李帅) />write /test/h lishuaihahahaha
成功写入 14字节
filesystem(1553534-李帅) />cd test
filesystem(1553534-李帅) /test/>read h
lishuaihahahaha
filesystem(1553534-李帅) /test/>rm /test/h
filesystem(1553534-李帅) /test/>ls
filesystem(1553534-李帅) /test/>cd /DirA
filesystem(1553534-李帅) /DirA/>cd
ERROR: 找不到指定路径!
filesystem(1553534-李帅) /DirA/>cd /
filesystem(1553534-李帅) />ls
/home
/user
/DirA
/DirB
/DirC
/test
filesystem(1553534-李帅) />rm /test
filesystem(1553534-李帅) />ls
/home
/user
/DirA
/DirB
/DirC
```

测试截图如上，功能正确无误。

## (2) 问题与解决

在具体的模块设计与代码编写的时候，我出现过以下问题：

- a. String 与 char [] 使用混乱，导致指针出现错误，后来统一了设计才解决这个 bug.
- b. 一开始的设计值设计了指令的绝对路径参数，没有考虑指令参数中的相对位置，考虑到实际情况有相对路径的参数，所以我在 main.cpp 的 main 函数中每个模块都加入了路径讨论：是相对路径还是绝对路径？
- c.

## 五. 实例分析

在你的 volumeFile 文件模拟的文件系统中，依次执行下列操作：

- (1) 新建文件/test/Jerry;
- (2) 打开该文件;
- (3) 写入 800 个字节;
- (4) 将文件读写指针定位到第 500 字节;
- (3) 读出 20 个字节。

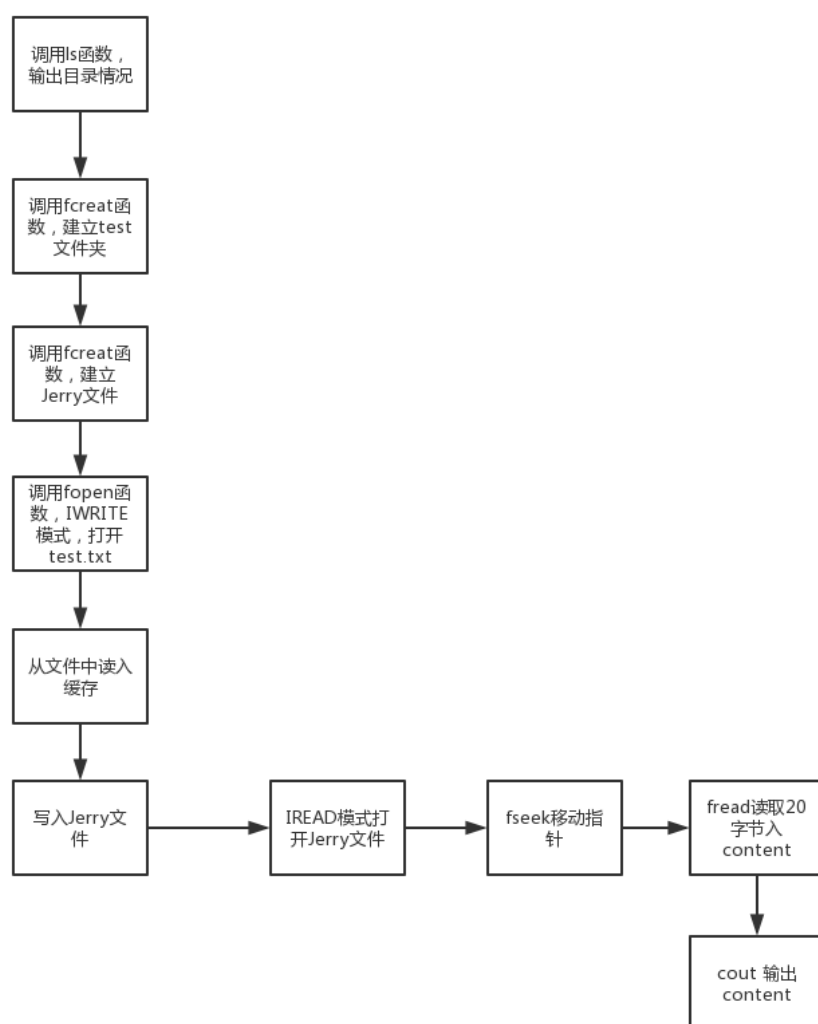
实例分析：进入用户输入界面，输入 test 指令进行实例分析。实例分析如下图：

[illegible]

- (1) 首先调用 `ls`，输出目前的目录情况。
- (2) 然后创建 `test` 文件夹以及 `Jerry` 文件。
- (3) 外部读取 `test.txt` 文件（800 字节）入缓存，缓存数据写入 `test/Jerry`
- (4) 将指针移动到文件的 500 字节位置，读出 20 个字节。
- (5) 因为我的 800 字节的 `txt` 文件是 0123456789 十个字符循环 80 次得到的，所以移动 500 字节后取 20 字节的结果就是 01234567890123456789，与输出结果一致吻合，实例分析正确。

流程图如下：



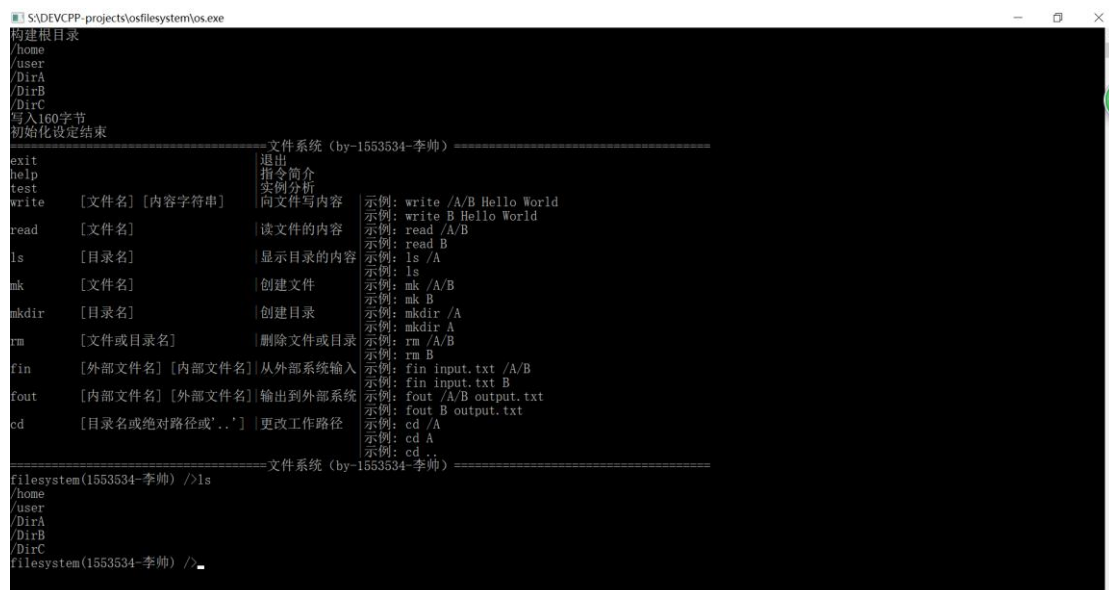


## 六. 用户使用说明

本项目的开发环境是 **win10,DevCpp**,其中包括以下文件:

- (1) `filesystem.h` 头文件
- (2) `main.cpp` 源文件
- (3) `makefile.win` makefile 文件
- (4) `myDisk.img` 模拟磁盘, 一级文件
- (5) `os.dev` Dev-cpp project 文件
- (6) `os.exe` 可执行文件
- (7) `test.txt` 预设的 800 字节文件
- (8) `test1.txt` 测试文件

用户只要打开 `os.exe` 文件就会进入文件系统, 界面如下:



```
S:\DEV\CPP-projects\osfilesystem\os.exe
构建根目录
/home
/user
/DirA
/DirB
/DirC
写入160字节
初始化设定结束

=====文件系统 (by-1553534-李帅)=====
exit      退出
help      指令简介
test      实例分析
write [文件名] [内容字符串]  向文件与内容  示例: write /A/B Hello World
read [文件名]  读文件的内容  示例: read /A/B
ls [目录名]  显示目录的内容  示例: ls /A
mk [文件名]  创建文件  示例: mk /A/B
mkdir [目录名]  创建目录  示例: mkdir /A
rm [文件或目录名]  删除文件或目录  示例: rm /A/B
fin [外部文件名] [内部文件名] 从外部系统输入  示例: fin input.txt /A/B
fout [内部文件名] [外部文件名] 输出到外部系统  示例: fout /A/B output.txt
cd [目录名或绝对路径或'..'] 更改工作路径  示例: cd /A

=====文件系统 (by-1553534-李帅)=====
filesystem(1553534-李帅) />ls
/home
/user
/DirA
/DirB
/DirC
filesystem(1553534-李帅) />.
```

- 首先进行初始化设定，创建 superblock，写入 1024 字节，创建 inode，构建如图所示的 5 个根目录。
- 命令行形式，等待用户输入指令，初始界面是指令介绍。
- 输入指令，开始你的文件系统之旅吧。。。。。。

## 七. 实验总结

本次实验在理论上，实现虚拟二级文件系统，在实现过程中，梳理了一遍课本知识，总的来说，文件系统出现是为了解决多用户存储、管理信息时出现的问题。用户所有的操作都是基于逻辑文件的，文件系统最终需要将用户对逻辑文件的操作转换成对物理文件的操作。物理文件可以是在存储设备上的存储区域，也可以使一个设备、管道、套接字，文件系统将用户对文件的操作转换成用户对设备的操作、用户间的通信操作和网络操作。在实现过程中，我对目录管理，文件存储空间管理，unix v6++的文件系统模块类结构有了更深入的了解。

遇到的问题主要还是设计思路与编程的细节问题。一开始我是按照（或者照抄）unix 源码的思路，但是随着代码解读的深入，很多源码的功能与成员函数其实在本实验中并不需要，也过于复杂，为此我走了很多弯路，最后放弃照抄源码，改成在自己理解的基础上实现了系统。编程的细节问题就是自己不好的代码习惯导致的。比如对于字符数组的不理解，对于 string 与 char\* 的混乱，还有就是目前长期 python 语言的编程导致自己对于 c++/c 语言编程的不熟悉，比如 C 语言中的函数，有着严格的顺序限制，如果要调用函数，该函数需要在本次调用之前就需要被实现，或者在程序开头事先声明，而 Python 中则没有这个限制，Python 中还有高阶函数这一概念，即函数名也可当作函数参数，函数名也是一种变量，指向内存中的某个函数，这种写法可以大大减少代码长度。我在编程的时候，由于没注意这个函数的位置，导致一直报错，最后看到了 python 与 c 语言的区别才改正了这个错误。

最后，对于本次课设与上学期理论课程的认知，我觉得上学期最后期末学的文件系统其实还不太够（至少我自己学的很不透彻），然后做课设的时候处处碰壁，不得不重新读教材，所以我觉得可以在大三上学期理论课的实验中提前加加入一点文件系统的内容，方便学生大三上做课设的时候及时上手。

最后，还是感觉自己对于 unix 源码的理解不够，所以导致最后做的课程设计其

实比较坎坷，同时在最近找工作面试中也经常被问操作系统的问题，所以暑假还是要继续读操作系统课本以及源码。

谢谢老师。

## 八. 参考文献

- [1] 操作系统原理(讲义) [M]. 同济大学计算机系. .2017(09) : 293-335
- [2] 构造嵌入式 Linux 的文件系统[J]. 郑桦,刘清,邢航,徐智穹. 微计算机信息. 2004(08)
- [3] UNIX 操作系统的发展[J]. 荣广颐. 微电子学与计算机. 1989(02)
- [4] UNIX 操作系统分析报告[J]. 刘日升 孙玉方. 计算机研究与发展:1982(09)