# 同济大学计算机系

# 操作系统课程设计

****

**学 号 1652325**

**姓 名 张一鸣**

**专 业 计算机科学与技术**

**授课老师 方钰**

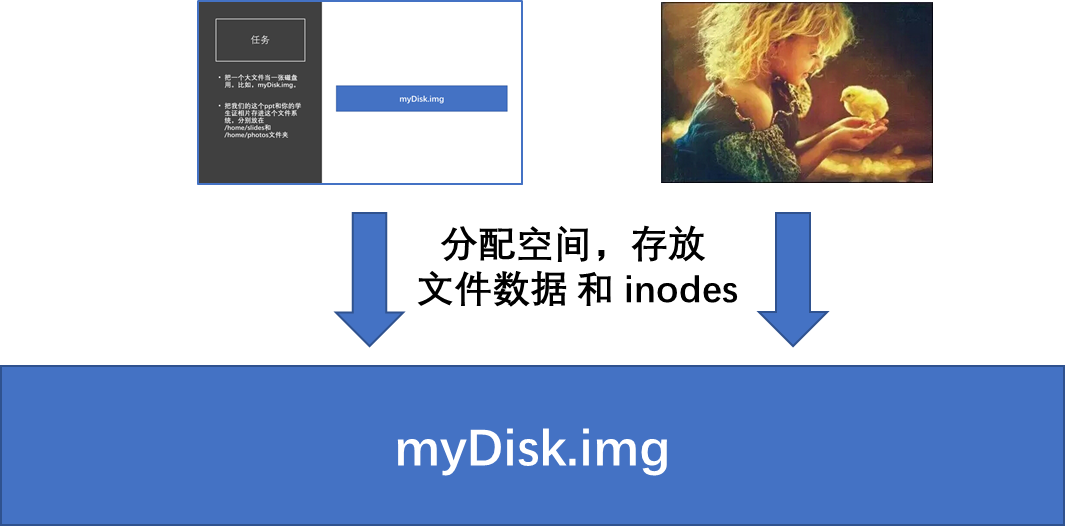
## 需求分析

### 1.1任务说明

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（把一个大文件当一张磁盘用）。



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



### 1.2需求说明

1.磁盘文件结构

（1）定义自己的磁盘文件结构

（2）SuperBlock结构

（3）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

（4）磁盘Inode节点的分配与回收算法设计与实现

（5）文件数据区的分配与回收算法设计与实现

2.文件目录结构

（1）目录文件结构

（2）目录检索算法的设计与实现

3. 文件打开结构

4. 磁盘高速缓存：选作

5. 文件操作接口：

（1）Fformat：格式化文件卷

（2）ls： 列目录

（3）mkdir： 创建目录

（4）fcreat： 新建文件

（5）fopen： 打开文件

（6）fclose： 关闭文件

（7）fread： 读文件

（8）fwrite： 写文件

（9）flseek： 定位文件读写指针

（10）fdelete：删除文件

6. 主程序：

（1）新建文件/test/Jerry，打开该文件，写入800个字节；

（2）将文件读写指针定位到第500字节，读出20个字节。

（3）图形界面或者命令行方式，等待用户输入；

（4）根据用户不同的输入，返回结果。

### 1.3指令说明

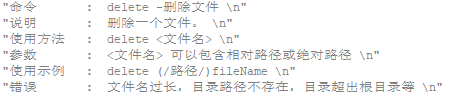
(1)fformat:



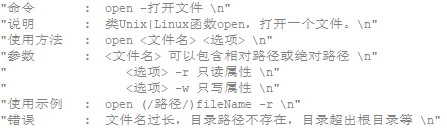
(2)exit:



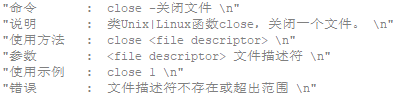
(3)delet:



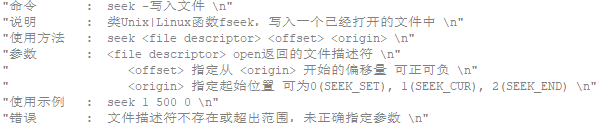
(4)open:



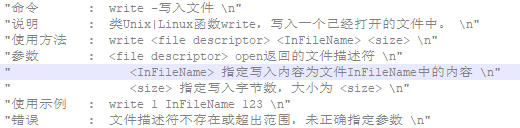
(5)close:



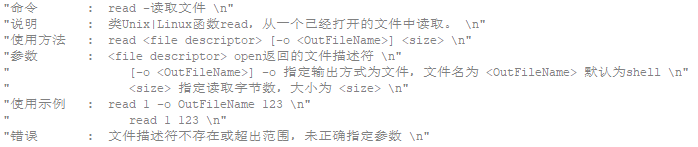
(6)seek:



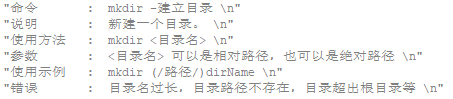
(7)write:



(8)read:



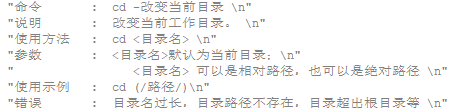
(9)mkdir:



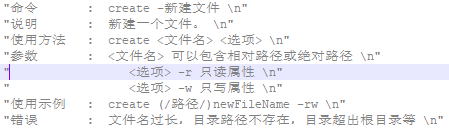
(10)ls:



(11)cd:



(12)create:



(13)test:



## 概要设计

### 2.1模块划分以及模块结构

#### 2.1.1Buf模块：高速缓存模块。

该模块主要的功能是管理文件系统中的所有高速缓存块，包括申请、读入、写出、释放一块高速缓存块的函数接口，退出时刷新所有缓存块。

class Buf {

public:

/\* flags中标志位,枚举 \*/

enum Buf\_flag {

Buf\_WRITE = 0x1,

Buf\_READ = 0x2,

Buf\_DONE = 0x4,

Buf\_ERROR = 0x8,

Buf\_BUSY = 0x10,

Buf\_WANTED = 0x20,

BBuf\_ASYNC = 0x40,

Buf\_DELWRI = 0x80

};

public:

unsigned int flags; /\* 缓存控制块标志位 \*/

Buf\* forw;

Buf\* back;

int count;

unsigned char\* Address;

int block\_num;

int u\_error;

int resid;

int no;

public:

Buf();

~Buf();

void Debug\_Mark();

void Debug\_Content();

};

class Buf\_Manager {

public:

static const int Num\_Buf = 100;

static const int Buf\_Size = 512;

private:

Buf\* Buf\_List;

Buf Buf\_Block[Num\_Buf];

unsigned char BUF[Num\_Buf][Buf\_Size];

unordered\_map<int, Buf\*> map;

Device\_Driver\* deviceDriver;

public:

Buf\_Manager();

~Buf\_Manager();

Buf\* GetBlock(int block\_num);

void Buf\_Relse(Buf\* bp);

Buf\* Buf\_Read(int blkno);

void Buf\_Write(Buf\* bp);

void Buf\_Dwrite(Buf\* bp);

void Buf\_Clear(Buf\* bp);

void Buf\_Flush();

void Format\_Buf();

private:

void InitList();

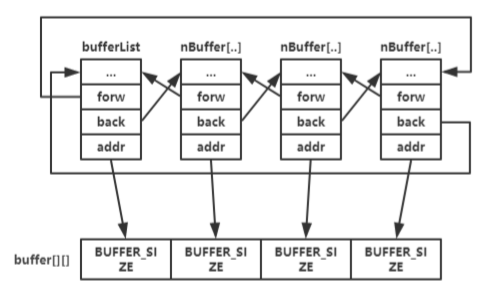
void DetachNode(Buf\* buffer);

void InsertTail(Buf\* buffer);

void debug();

};

图示：



#### 2.1.2Device\_Driver模块：设备驱动模块

该模块的功能是可以对存于磁盘的文件进行读写操作。

class Device\_Driver {

public:

static const char\* DISK\_FILE\_NAME;

private:

FILE\* fp;

public:

Device\_Driver();

~Device\_Driver();

bool Exists();

void Construct();

void write(const void\* buffer, unsigned int size,

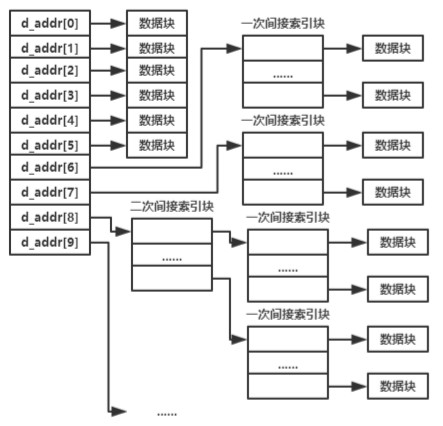
int offset = -1, unsigned int origin = SEEK\_SET);

void read(void\* buffer, unsigned int size,

int offset = -1, unsigned int origin = SEEK\_SET);

};

图示：



#### 2.1.3User模块：用户管理模块

该模块的功能是对用户的操作进行分析，检测正确性，并传递给文件系统，然后再反馈给用户。

class User {

public:

static const int EAX = 0;

enum ErrorCode {

U\_NOERROR = 0, /\* 没有错误码\*/

U\_ENOENT = 2, /\* 不存在文件或目录 \*/

U\_EBADF = 9, /\* 错误文件号 \*/

U\_EACCES = 13, /\* 没有权限 \*/

U\_ENOTDIR = 20, /\* 不是目录 \*/

U\_ENFILE = 23, /\* 文件表溢出 \*/

U\_EMFILE = 24, /\* 打开太多文件 \*/

U\_EFBIG = 27, /\* 文件太大 \*/

U\_ENOSPC = 28, /\* 设备没有空间 \*/

};

public:

INode\* cdir;

INode\* pdir;

DirectoryEntry dent;

char dbuf[DirectoryEntry::DIRSIZ];

string curDirPath;

string dirp;

long arg[5];

unsigned int ar0[5];

ErrorCode u\_error;

OpenFiles ofiles;

IOParameter IOParam;

FileManager\* fileManager;

string ls;

public:

User();

~User();

void U\_Ls();

void U\_Cd(string dirName);

void U\_Mkdir(string dirName);

void U\_Create(string fileName, string mode);

void U\_Delete(string fileName);

void U\_Open(string fileName, string mode);

void U\_Close(string fd);

void U\_Seek(string fd, string offset, string origin);

void U\_Write(string fd, string inFile, string size);

void U\_Read(string fd, string outFile, string size);

//void Pwd();

private:

bool Judge\_Error();

void EchoError(enum ErrorCode err);

int INodeMode(string mode);

int FileMode(string mode);

bool checkPathName(string path);

};

#### 2.1.4File\_System模块：文件系统盘块管理模块

该模块的功能为对镜像文件的存储空间管理，包括SuperBlock 空间占用、DiskINode 空间分布、数据块区空间分布的管理。要设置Inode操作的接口，可以格式化磁盘文件。

class SuperBlock {

public:

const static int MAX\_NFREE = 100;

const static int MAX\_NINODE = 100;

public:

int s\_isize;

int s\_fsize;

int s\_nfree;

int s\_free[MAX\_NFREE];

int s\_ninode;

int s\_inode[MAX\_NINODE];

int s\_flock;

int s\_ilock;

int s\_fmod;

int s\_ronly;

int s\_time;

int padding[47];

};

class FileSystem {

public:

static const int BLOCK\_SIZE = 512;

static const int DISK\_SIZE = 16384;

static const int SUPERBLOCK\_START\_SECTOR = 0;

static const int INODE\_ZONE\_START\_SECTOR = 2;

static const int INODE\_ZONE\_SIZE = 1022;

static const int INODE\_NUMBER\_PER\_SECTOR = BLOCK\_SIZE / sizeof(DiskINode);

static const int ROOT\_INODE\_NO = 0;

static const int INode\_NUMBERS = INODE\_ZONE\_SIZE \* INODE\_NUMBER\_PER\_SECTOR;

static const int DATA\_ZONE\_START\_SECTOR = INODE\_ZONE\_START\_SECTOR + INODE\_ZONE\_SIZE;

static const int DATA\_ZONE\_END\_SECTOR = DISK\_SIZE - 1;

static const int DATA\_ZONE\_SIZE = DISK\_SIZE - DATA\_ZONE\_START\_SECTOR;

public:

Device\_Driver\* deviceDriver;

SuperBlock\* superBlock;

Buf\_Manager\* bufferManager;

public:

FileSystem();

~FileSystem();

void FormatSuperBlock();

void FormatDevice();

void LoadSuperBlock();

void Update();

INode\* IAlloc();

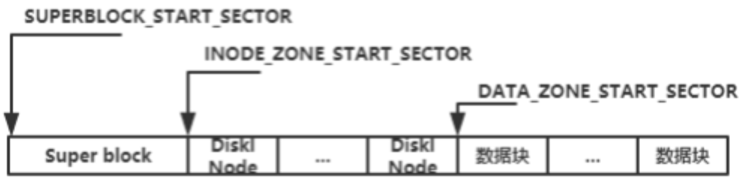
void IFree(int number);

Buf\* Alloc();

void Free(int blkno);

};

图示：



#### 2.1.5File\_Manager模块：文件功能管理模块

该模块的主要功能是对系统中的文件进行各种处理：创建、删除、打开、关闭、读入、写入等。

class INodeTable {

public:

static const int NINODE = 100;

private:

INode m\_INode[NINODE];

FileSystem\* fileSystem;

public:

INodeTable();

~INodeTable();

INode\* IGet(int inumber);

void IPut(INode\* pNode);

void UpdateINodeTable();

int IsLoaded(int inumber);

INode\* GetFreeINode();

void Format();

};

class FileManager

{

public:

enum DirectorySearchMode

{

OPEN = 0,

CREATE = 1,

DELETE = 2

};

public:

INode\* rootDirINode;

FileSystem\* fileSystem;

INodeTable\* inodeTable;

OpenFileTable\* openFileTable;

public:

FileManager();

~FileManager();

void Open();

void Creat();

void Open1(INode\* pINode, int mode, int trf);

void Close();

void Seek();

void Read();

void Write();

void Rdwr(enum File::FileFlags mode);

INode\* NameI(enum DirectorySearchMode mode);

INode\* MakNode(unsigned int mode);

void UnLink();

void WriteDir(INode\* pINode);

void ChDir();

void Ls();

};

#### 2.1.6Open\_File\_Manager模块：打开文件模块

该模块的功能为为打开文件提供接口，将文件与数据结构建立连接。

class OpenFileTable {

public:

static const int MAX\_FILES = 100; /\* 打开文件控制块File结构的数量 \*/

public:

File m\_File[MAX\_FILES];

public:

OpenFileTable();

~OpenFileTable();

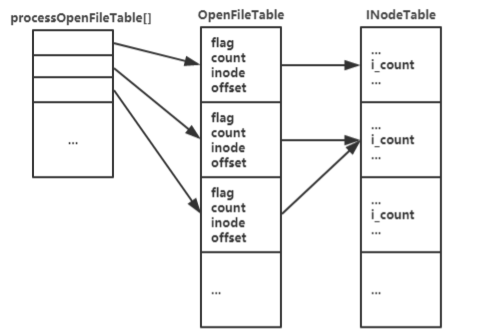
File\* FAlloc();

void CloseF(File\* pFile);

void Format();

};

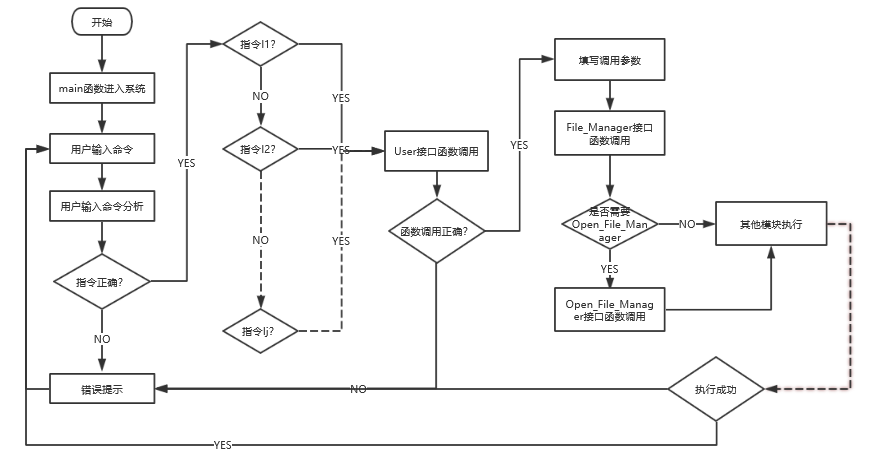
图示：



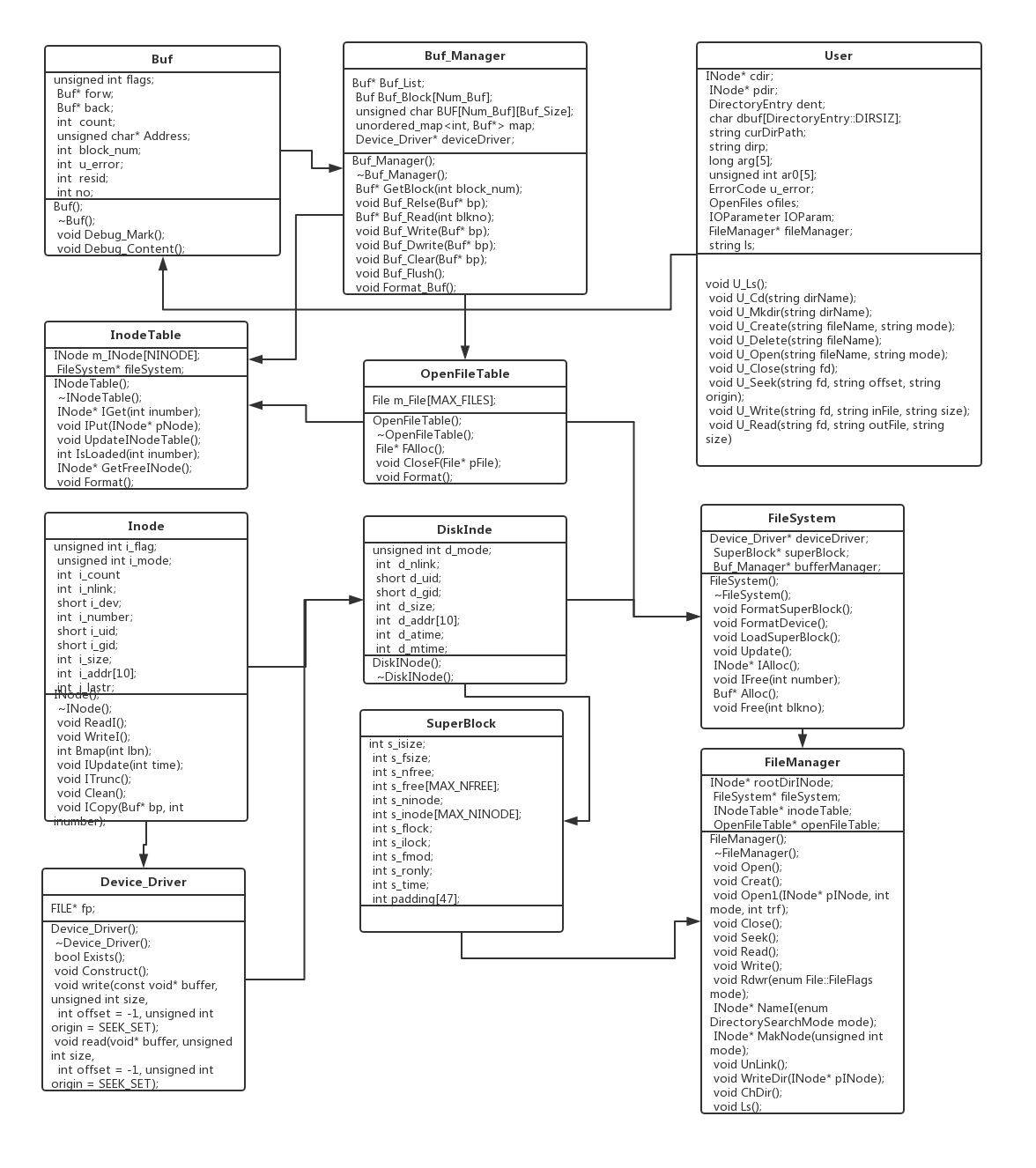
### 2.2流程分析

程序的总体流程为：main 函数进入文件系统，然后用户进行命令输入，系统对命令进行分析，正确就通过User功能函数来调用File\_Manager模块，File\_Manager根据输入的命令，调用各个模块的功能函数，进行文件操作。

整个系统流程图，如下图所示：



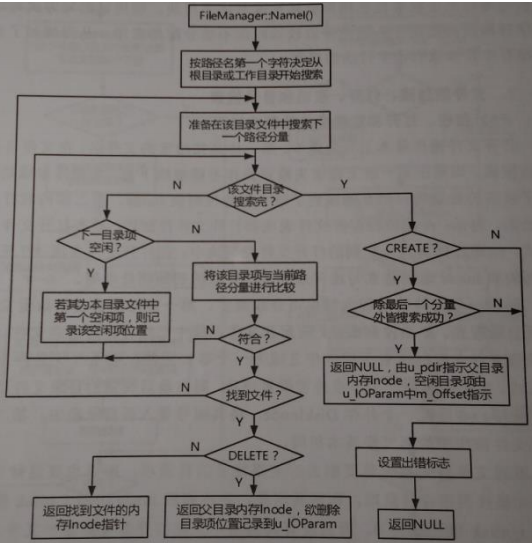
### 2.3模块调用



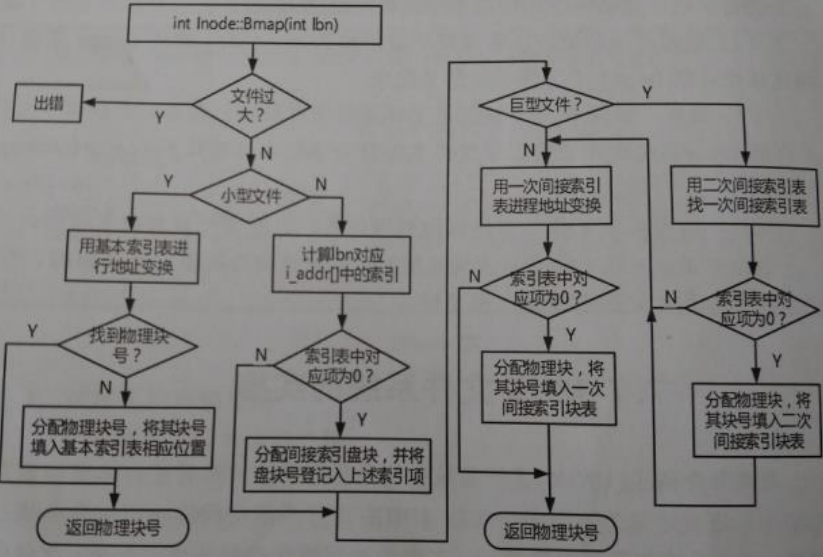
## 详细设计

### 3.1目录搜索和地址变换

目录搜索需要调用NameI 函数，流程图如下：



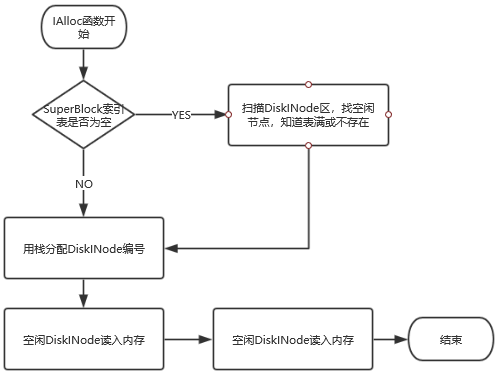
地址变换 Bmap 函数，流程图如下：



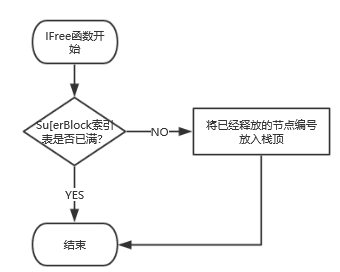
### 3.2外存磁盘文件管理

外存索引节点分配需要调用IAlloc 函数，DiskInode索引采用栈来管理DiskINode。

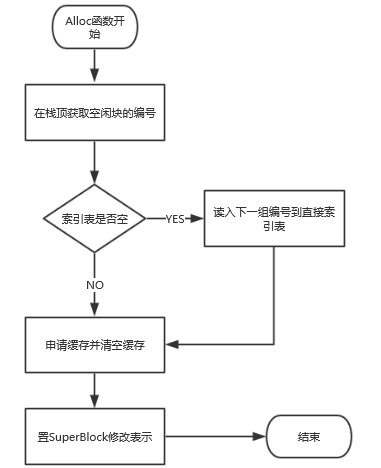
当要分配磁盘节点时，若s\_ninode!=0,则用栈来分配节点。若s\_ninode==0,则表明没有空闲节点，那要重新扫描整个磁盘，将找到的DiskINode号顺次写入s\_inode表，知道表满或搜索完毕。下面是流程图：



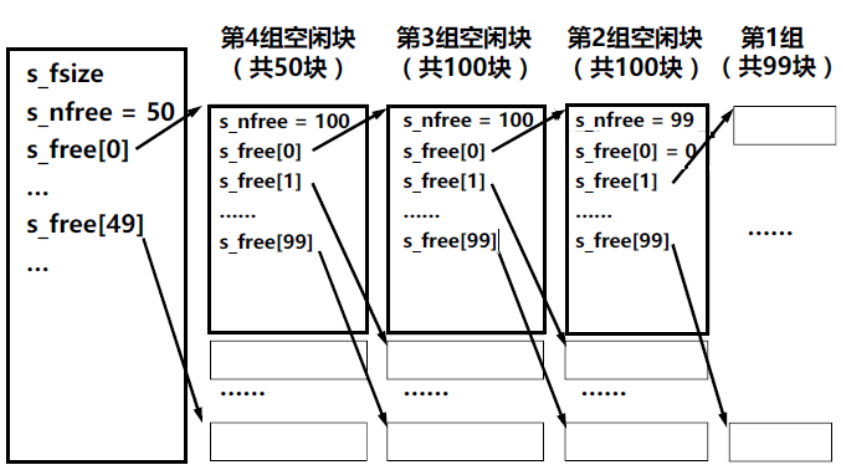
外存索引节点分释放 所需要的是IFree 函数，流程图如下：



外存空闲盘块分配调用 Alloc 函数，流程图如下图所示：



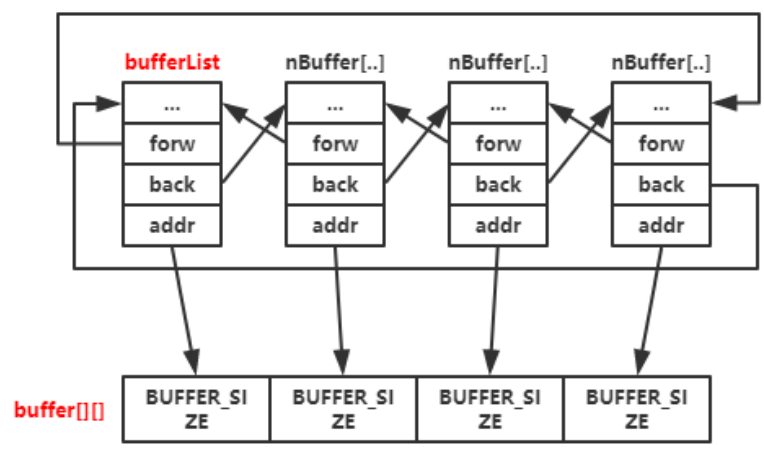
空闲盘块分组链式索引如图所示：



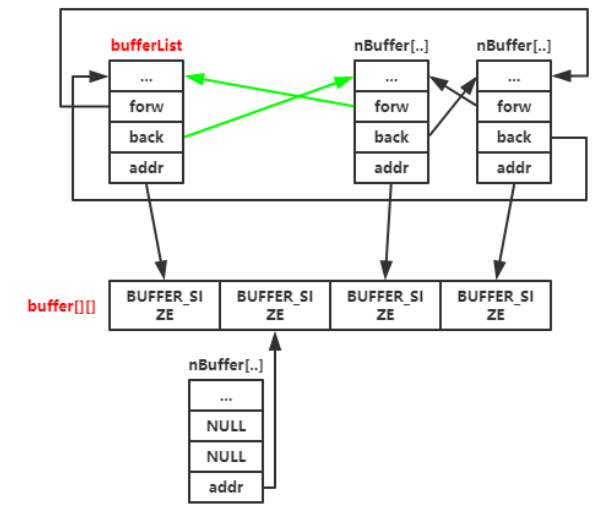
### 3.3高速缓存

本系统未实现多用户并行操作，可以将自由队列和设备队列进行合并为一个缓存队列。主要的功能为初始化高速缓存、分离节点、插入节点、申请缓存。

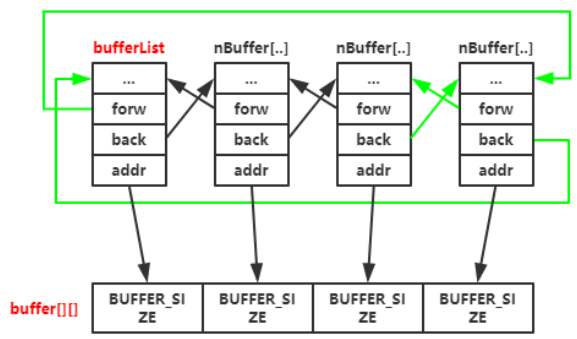
初始化高速缓存，构建双向循环链表示意图如下：



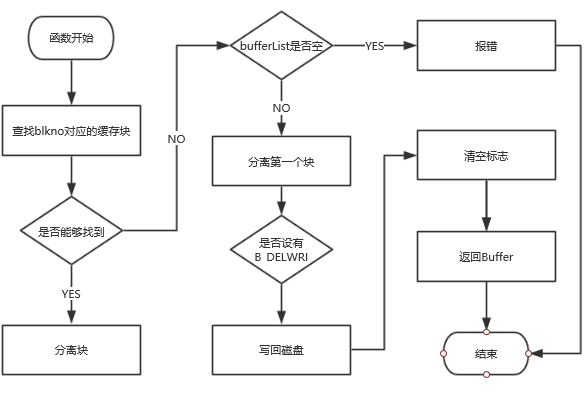
分离节点采用LRU算法，示意图如下：



插入节点采用LRU算法，示意图如下：



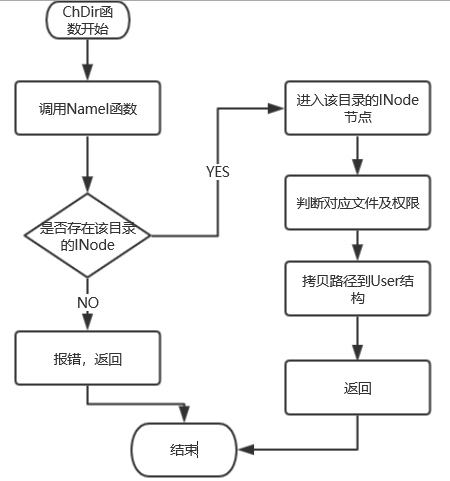
申请缓存的流程图如下：



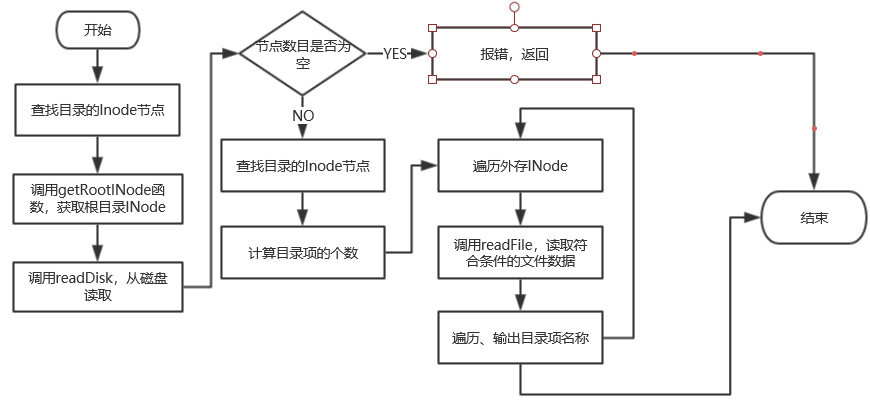
### 3.4详细功能设计

详细功能的流程调用如下：

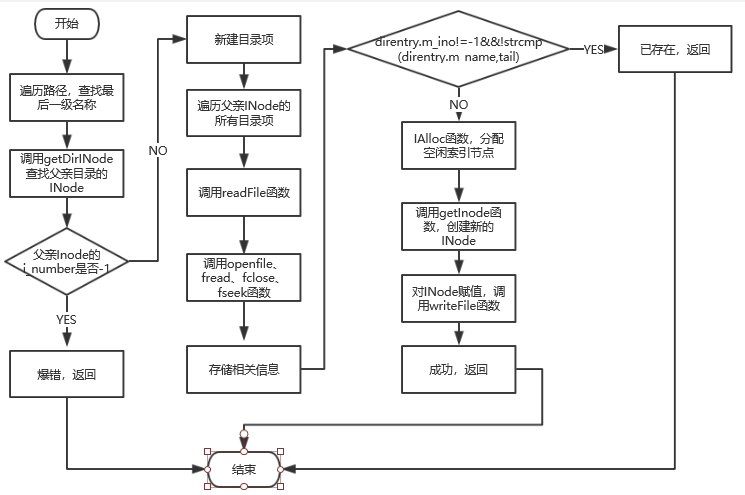
#### 3.4.1cd指令



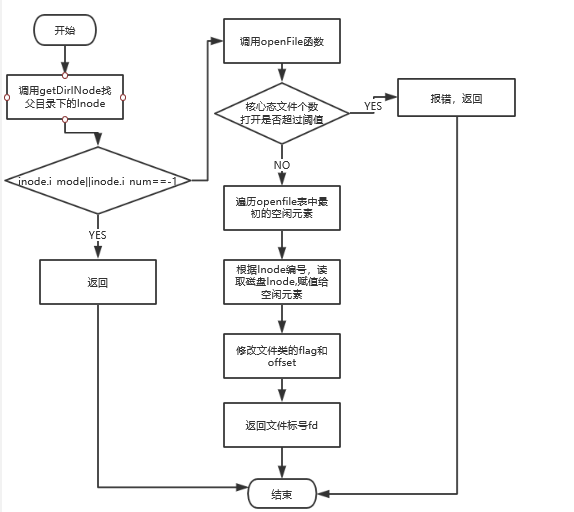
#### 3.4.2ls指令



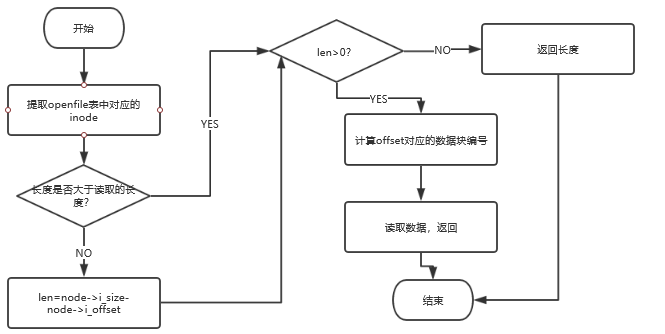
#### 3.4.3create指令



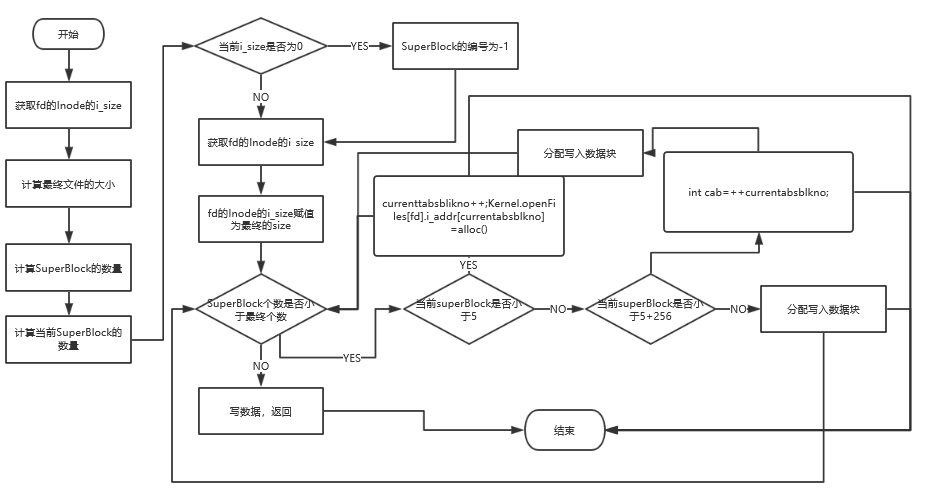
#### 3.4.4open指令



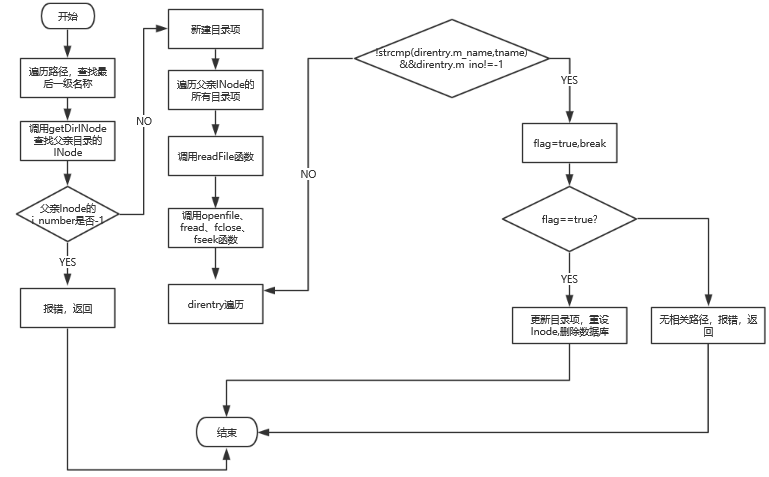
#### 3.4.5read指令



#### 3.4.6write指令



#### 3.4.7delet指令



#### 3.4.8seek指令

只需要把kernel 的对应的openFiles 类的 i\_offset 改成 offset 即可。

#### 3.4.9close指令

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

#### 3.4.10fformat指令

初始化所以类即可

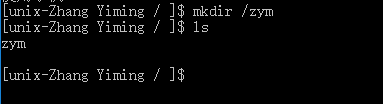
## 调试分析

### 4.1测试结果

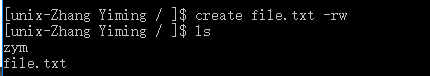
#### 4.1.1初始化界面



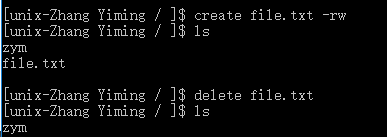
#### 4.1.2mkdir指令



#### 4.1.3create指令



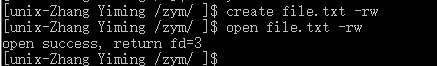
#### 4.1.4delete指令



#### 4.1.5cd指令



#### 4.1.6open指令



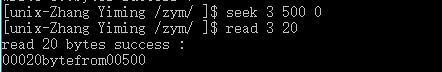
#### 4.1.7write指令



#### 4.1.8seek指令



#### 4.1.9read指令



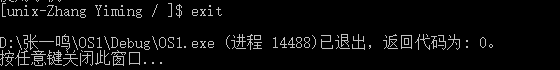
#### 4.1.10close指令



#### 4.1.11formort指令



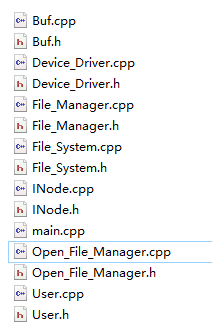
#### 4.1.12exit指令



### 4.2问题与解决

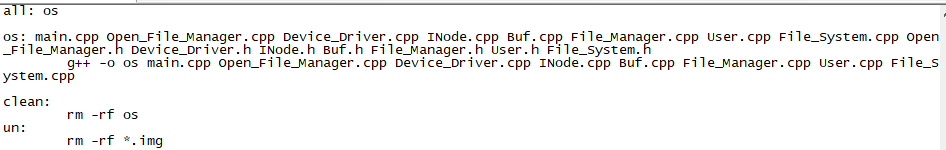
## 用户使用说明

该二级文件系统所需要的c++文件以及头文件，如下图所示：

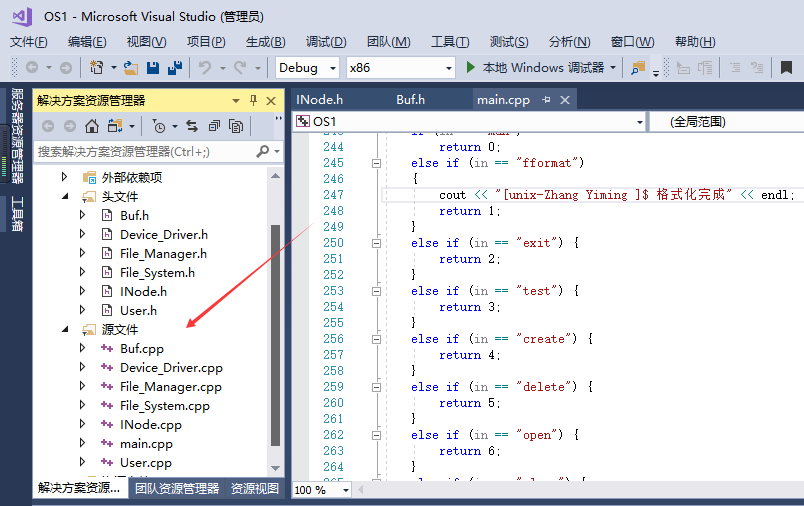


**编译方法：**

可以将源程序放在集成环境下编译，也可利用 GNU 编译工具，通过写好的 Makefile 进 行编译，make 如下

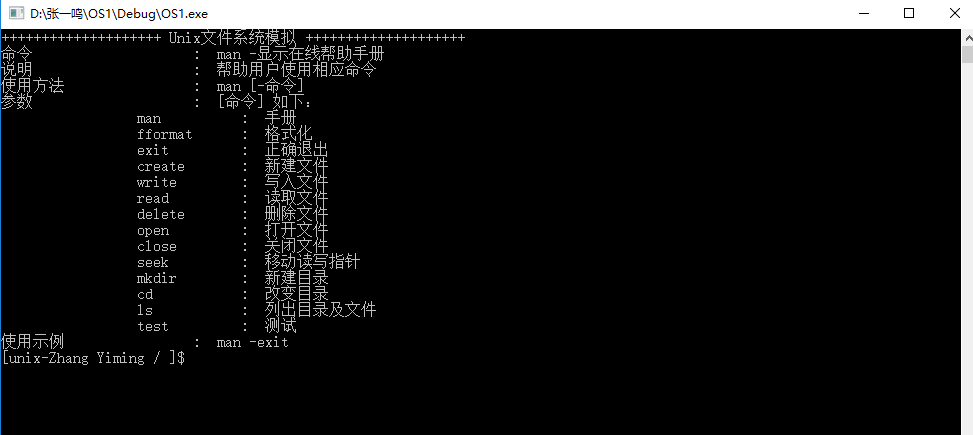


另外也可以在windows下创建VS项目，将文件添加进去，如下图所示



**运行：**

在 Windows 下直接点击生成的 exe 文件执行。



**正确退出：**

若要退出程序，最好通过 exit 命令。此时正常退出会调用析构函数。若有在内存中未 更新到磁盘上的缓存会及时更新，保证正确性。

**格式化：**

格式化命令为 fformat，运行命令后系统会进行文件系统格式化，然后正常退出，再次 进入时为初始环境。

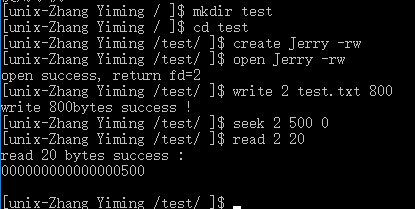
## 运行结果分析

### 6.1运行一

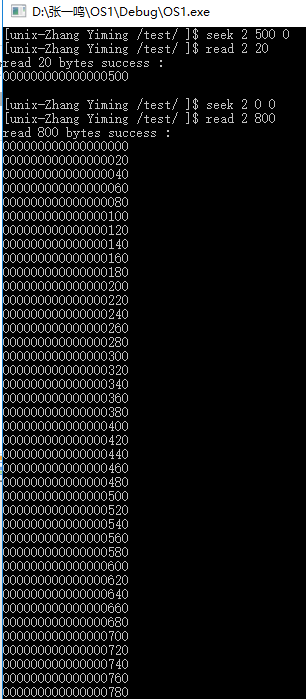
#### 6.1.1运行要求与结果

新建文件/test/Jerry，打开该文件，写入800个字节；

将文件读写指针定位到第500字节，读出20个字节。



其中读入的800字节为：



一行20字节，每一行数字代表的字符串表示该行的起始地址。

#### 6.1.2运行结果分析

**（1）mkdir test**

* 调用函数int judgement(string in)，来判断输入的指令。
* 判断输入的指令为mkdir之后，将test放入args[1]中，作为参数，调用函数void User::U\_Mkdir(string dirName)
  + 调用函数bool User::checkPathName(string path)，判断参数是否合法，不合法返回，合法继续。
  + 调用函数void FileManager::Creat()来创建目录。
    - 调用INode\* FileManager::NameI(enum DirectorySearchMode mode)，搜索目录的模式为1，表示创建；若父目录不可写，出错返回。
    - 没有找到相应的INode节点，那么就调用函数INode\* FileManager::MakNode(unsigned int mode)创建一个新的INode节点。
      * 调用INode\* FileSystem::IAlloc()来分配一个空闲DiskInode
* 将目录项写入目录文件，调用函数void FileManager::WriteDir(INode\* pINode)
* 调用Judge\_Error()判断是否创建出错。
* 创建成功，返回

**（2）cd test**

* 调用函数int judgement(string in)，来判断输入的指令。
* 将test作为参数，调用void User::U\_Cd(string dirName)函数
  + 调用函数bool User::checkPathName(string path)，判断参数是否合法，不合法返回，合法继续。
  + 调用函数void FileManager::ChDir()
    - 调用函数INode\* FileManager::MakNode(unsigned int mode)，寻找INode指针
    - 搜索到的文件不是目录文件，直接返回。如果是目录文件则取代前有目录文件，判断其父亲目录，然后加入前缀返回。
  + 调用Judge\_Error()判断是否出错。
  + 成功，返回

**（3）create Jerry –rw**

* 调用函数int judgement(string in)，来判断输入的指令。
* 将Jerry和-rw作为参数，调用函数void User::U\_Create(string fileName, string mode)
  + 调用函数bool User::checkPathName(string path)，判断参数是否合法，不合法返回，合法继续。
  + 根据mode参数的值，调用函数int User::INodeMode(string mode)，对INode的权限进行设置。
  + 调用函数void FileManager::Creat()进行文件创建
    - 调用INode\* FileManager::NameI(enum DirectorySearchMode mode)，搜索目录的模式为1，表示创建；若父目录不可写，出错返回。
    - 未找到相应的INode，创建一个新的INode，调用函数INode\* FileManager::MakNode(unsigned int mode)
    - 调用函数void FileManager::Open1(INode\* pINode, int mode, int trf)，分配打开文件控制块File结构，并与INode建立联系
* 调运函数void INodeTable::IPut(INode\* pINode)加入INode表
* 调用Judge\_Error()判断是否出错。
* 成功，返回

**（4）open Jerry –rw**

* 调用函数int judgement(string in)，来判断输入的指令。
* 将Jerry和-rw作为参数，调用函数void User::U\_Open(string fileName, string mode)
  + 调用函数bool User::checkPathName(string path)，判断参数是否合法，不合法返回，合法继续。
  + 调运函数int User::FileMode(string mode)，获取操作权限
  + 调用函数void FileManager::open()进行文件打开
    - 调用INode\* FileManager::NameI(enum DirectorySearchMode mode)，获取INode路径编号。
    - 调用函数void FileManager::Open1(INode\* pINode, int mode, int trf)，分配打开文件控制块File结构，并与INode建立联系
  + 调用Judge\_Error()判断是否出错。
  + 成功，返回，打印该文件的fd号

**（5）write 2 test.txt 800**

* 调用函数int judgement(string in)，来判断输入的指令。
* 将2， test.txt ，800作为参数，调用void User::U\_Write(string sfd, string inFile, string size)
  + 调用函数fstream fin(inFile, ios::in | ios::binary);打开文件test.txt。
  + 将字符指针buffer和800作为参数，调用fin.read(buffer, usize);
  + 将fd,buffer和8000存入arg数组作为参数，调用void FileManager::Write()
    - 调用void FileManager::Rdwr(enum File::FileFlags mode)
      * 调用pFile = u.ofiles.GetF(u.arg[0]);获取打开文件控制块结构
      * 调用pFile->inode->WriteI();对INode进行写入
  + 调用Judge\_Error()判断是否出错。
  + 成功，返回，打印读入的bytes

**（6）seek 2 500 0**

* 调用函数int judgement(string in)，来判断输入的指令。
* 将2，500，0作为参数，调用函数void User::U\_Seek(string sfd, string offset, string origin)
  + 调用函数void FileManager::Seek()
    - 调用pFile = u.ofiles.GetF(u.arg[0]);获取打开文件控制块结构
    - 对读写位置进行调整
  + 调用Judge\_Error()判断是否出错。
  + 成功，返回

**（7）read 2 20**

* 调用函数int judgement(string in)，来判断输入的指令。
* 由于不输入到文件中，则将2，“”， 20 作为参数调用void User::U\_Read(string sfd, string outFile, string size)
  + 将2，buffer(字符指针)，20放入arg数组中，调用函数void FileManager::Read()
    - 调用函数void FileManager::Rdwr(enum File::FileFlags mode)
      * 调用pFile = u.ofiles.GetF(u.arg[0]);获取打开文件控制块结构
      * 调用pFile->inode->ReadI();进行读出
  + 调用Judge\_Error()判断是否出错。
  + 成功，返回，并打印读出的内容

### 6.2运行二

#### 6.2.1运行要求与结果

## 七、实验总结

**问题一：延迟写标记的缓存块问题**

用延迟写标记的缓存快，有时未全部更新到磁盘文件。在最开始的时候几乎没有使用exit，直接关闭界面，而使用exit命令，那么退出调用析构函数，就会对NULL指针进行读写访问，从而造成死循环。

解决：采用顺序循环的方式查找有延迟写的缓存块，并判断其是否为NULL，用时间来换取正确操作。

**问题二：析构函数问题**

很多模块都需要在构造函数中，动态申请内存，然而我最开始只是写了一个空的虚构函数，这样在程序结束的时候会造成内存丢失，有时候还会报错。

解决：修改析构函数，对于有动态申请的类，将析构函数都加上手动释放内存。

## 八、参考文献

[1] UNIX 操作系统的发展[J]. 荣广颐. 微电子学与计算机. 1989(02)

[2] 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)

[3] 操作系统原理(讲义) [M]. 同济大学计算机系. .2017(09) : 293-335

[4] 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)

[5]John Lions,莱昂氏 UNIX 源代码分析[M],机械工业出版社,2006 年 8 月

[6]Unix v6++源码