

同济大学计算机系

操作系统课程设计实验报告



题 目 Unix 文件系统

学 号 2251745

姓 名 张宇

指导老师 方钰

完成日期 2025 年 6 月 20 日

目 录

1 需求分析.....	3
1.1 程序输入和输出.....	3
1.2 程序功能需求.....	3
1.3 程序性能需求.....	3
1.4 用户其他需求.....	4
1.5 系统可行性.....	4
2 概要设计.....	5
2.1 任务分解.....	5
2.2 数据结构定义.....	5
2.3 模块间的调用关系.....	11
2.4 主程序流程与算法说明.....	11
3 详细设计.....	13
3.1 重点函数与重点变量.....	13
3.2 DiskINode 节点的分配与回收.....	15
3.3 文件数据区的分配与回收.....	16
3.4 文件索引和数据块的映射转换.....	17
3.5 目录搜索算法的实现.....	18
3.6 高速缓存的模拟和实现.....	19
4 运行结果分析.....	20
4.1 程序运行展示说明.....	20
4.2 测试结果分析.....	22
5 用户使用说明.....	24
5.1 运行说明	24
5.2 注意事项.....	24
6 实验总结.....	25

1 需求分析

1.1 程序输入和输出

使用 Windows Cmd 方式进行程序的输入和输出。

程序输入：类 linux 指令，如 fformat、mkdir、cd、ls、fcreate、fdelete、fcreate、fopen、fread、fwrite、fseek、fclose、exit、frename、ftree 等。

程序输出：返回句柄输出、提示信息输出、出错信息输出、文件内容输出等。

1.2 程序功能需求

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。



1.2.1 用户接口

命令	功能
fformat	格式化文件系统
ls	查看当前目录内容
mkdir <dirname>	生成文件夹
cd <dirname>	进入目录
fcreate <filename>	创建文件名为 filename 的文件
fopen <fd>	打开文件句柄为 fd 的文件
fwrite <fd> <infile> <size>	从 infile 输入，写入 fd 文件 size 字节
fread <fd> <outfile> <size>	从 fd 文件读取 size 字节，输出到 outfile
fread <fd> std <size>	从 fd 文件读取 size 字节，输出到屏幕
fseek <fd> <step> begin	以 begin 模式把 fd 文件指针偏移 step
fseek <fd> <step> cur	以 cur 模式把 fd 文件指针偏移 step
fseek <fd> <step> end	以 end 模式把 fd 文件指针偏移 step
fclose <fd>	关闭文件句柄为 fd 的文件
fdelete <filename>	删除文件文件名为 filename 的文件或者文件夹
frename <filename> <filename1>	将文件 filename 重命名为 filename1
ftree <dir>	列出 dir 文件夹的文件目录树
eixt	退出系统，并将缓存内容存至磁盘

1.2.2 系统调用

对相应的用户接口实现系统调用接口，使用 ar0 和 arg 来传递返回值和参数。

1.2.3 磁盘文件结构

- a. 实现 SuperBlock 结构;
- b. 磁盘 INode 节点结构，内有索引结构;
- c. 内存 INode 结构和磁盘 INode 结构的同步;
- d. 磁盘 Inode 节点的分配与回收算法设计与实现;
- e. 文件数据区的分配与回收算法设计与实现。

1. 2. 4 文件目录结构

- a. 目录文件结构;
- b. 目录检索算法的设计与实现。

1. 2. 5 文件打开结构

- a. 对进程打开的文件进行统一管理;
- b. 实现对 INode 的分配和管理。

1. 2. 6 高速缓存结构

- a. 模拟缓存，实现与硬盘数据的交互;
- b. 实现延迟写入的功能;
- c. 程序退出时将缓存内容更新到磁盘，防止数据丢失。

1. 3 程序性能需求

数据精度：文件系统用于存储管理文字、图片、视频、音频等，必须保证存储过程不能有数据损失，保证原始数据精度。

时间特性：利用高速缓存提高文件系统管理效率，读写磁盘并返回文件内容的时间不高于 100ms。

1. 4 用户其他需求

界面友好：有简洁大方的界面，便于用户进行操作。

结果输出：将结果直接在屏幕上输出，也可保存在对应的文件中，便于用户操作。

1. 5 系统可行性

编程语言：采用 C++ 实现，简洁高效。

测试平台：使用 Visual Studio 2022 实现。

2 概要设计

2.1 任务分解

对程序进行分析，将任务模块分解为以下几个部分：

- (1) UserCall 实现用户接口模块，包括用户的当前目录，父目录，进程打开文件描述表，读写文件参数。并且包含用户的命令接口。具有错误处理的功能，定义用户错误关键字，当内核态出现错误时，判断错误原因并输出。同时和系统调用模块进行交互，使用 ar0[5] 和 arg[5] 分别传递返回值和参数，实现系统调用。
- (2) SystemCall 实现系统调用模块，实现了各文件操作的系统接口，并且针对每条命令都有具体的实现。负责 fformat、mkdir、cd、ls、fcreate、fdelete、fopen、fread、fwrite、fseek、fclose、exit、frename、ftree 等系统功能的实现细节以及与用户模块的交互。同时包含根目录内存 INode 指针，便于初始化时将根目录 DiskINode 读入内存 INode 结构。SystemCall 模块起着衔接用户对文件的操作以及对内核数据结构相应处理的作用。
- (3) OpenFileManager 实现文件打开结构与内存 INode 表的管理。OpenFileTable 负责对打开文件实现文件描述符的分配，以及在文件关闭时对文件描述符进行回收，和 fcreate 以及 fclose 命令相关。INodeTable 负责内存 INode 结构的分配和释放。
- (4) FileSystem 实现磁盘文件结构的定义与整个文件系统的管理。负责磁盘文件结构的管理，包括 SuperBlock 和 INode 在磁盘的存储，以及 DiskINode 节点的索引结构 DirectoryEntry 的管理，以及内存中 SuperBlock 和 INode 与硬盘中 SuperBlock 和 INode 的更新和交互。
- (5) CacheManager 实现高速缓存管理模块，负责缓存块 CacheBlock 的申请、释放、延迟写、正常读写、刷新磁盘等操作，同时也是 DiskDriver 与上层进行交互的一个接口。
- (6) DiskDriver 是最底层的磁盘模块，直接负责硬盘文件 myDisk.img 的创建，判断是否存在，初始化，读写交互等。

2.2 数据结构定义

2.2.1 磁盘文件结构

包括 SuperBlock 区、INode 区和文件数据区：



```
class FileSystem
{
public:
    static const int BLOCK_SIZE = 512;
        //Block块大小, 单位字节
    static const int DISK_SECTOR_NUMBER = 16384;
        //磁盘所有扇区数量 8MB / 512B = 16384
    static const int SUPERBLOCK_START_SECTOR = 0;
        //定义SuperBlock位于磁盘上的扇区号, 占据两个扇区
```

```

static const int INODE_START_SECTOR = 2;
    //外存INode区位于磁盘上的起始扇区号
static const int INODE_SECTOR_NUMBER = 1022;
    //磁盘上外存INode区占据的扇区数
static const int INODE_SIZE = sizeof(DiskINode);
static const int INODE_NUMBER_PER_SECTOR = BLOCK_SIZE / INODE_SIZE;
    //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
static const int ROOT_INODE_NO = 0;
    //文件系统根目录外存INode编号
static const int INODE_NUMBER_ALL = INODE_SECTOR_NUMBER *
INODE_NUMBER_PER_SECTOR;
    //外存INode的总个数
static const int DATA_START_SECTOR = INODE_START_SECTOR + INODE_SECTOR_NUMBER;
    //数据区的起始扇区号
static const int DATA_END_SECTOR = DISK_SECTOR_NUMBER - 1;
    //数据区的最后扇区号
static const int DATA_SECTOR_NUMBER = DISK_SECTOR_NUMBER - DATA_START_SECTOR;
    //数据区占据的扇区数量
};

```

2. 2. 2 SuperBlock 结构

主要包含两类重要的数据：

(1) 外存索引节点的管理：

- a. s_isize：表示存储设备上的 DiskINode 区占据的块数，这些值在格式化磁盘文件的时候确定，由 FileSystem 类中的静态变量直接定义；
- b. s_ninode、s_inode[100]：SuperBlock 直接管理的空闲 DiskINode 节点数量和索引表，索引表中记录的是 s_inode 个空闲外存索引节点的编号。

(2) 空闲数据块的管理：

- a. s_fsize：表示文件系统的数据块总数；
- b. s_nfree、s_free[100]：SuperBlock 直接管理的空闲块数和空闲索引表。在操作系统初始化时，会将磁盘的 SuperBlock 读入一个内存的 SuperBlock 副本，从而便于内核以更快的速度随时访问内存副本。一旦内存中的副本发生变化，会通过设置 s_fmod 标志，由内核将内存副本写入磁盘。

```

class SuperBlock
{
public:
    const static int MAX_NUMBER_FREE = 100;
    const static int MAX_NUMBER_INODE = 100;
public:
    int s_isize;           //外存INode区占用的盘块数 1022
    int s_fsize;           //文件系统的数据盘块总数 16384 - 1024 = 15360
    int s_nfree;           //直接管理的空闲盘块数量
    int s_free[MAX_NUMBER_FREE]; //直接管理的空闲盘块索引表
    int s_ninode;          //直接管理的空闲外存INode数量

```

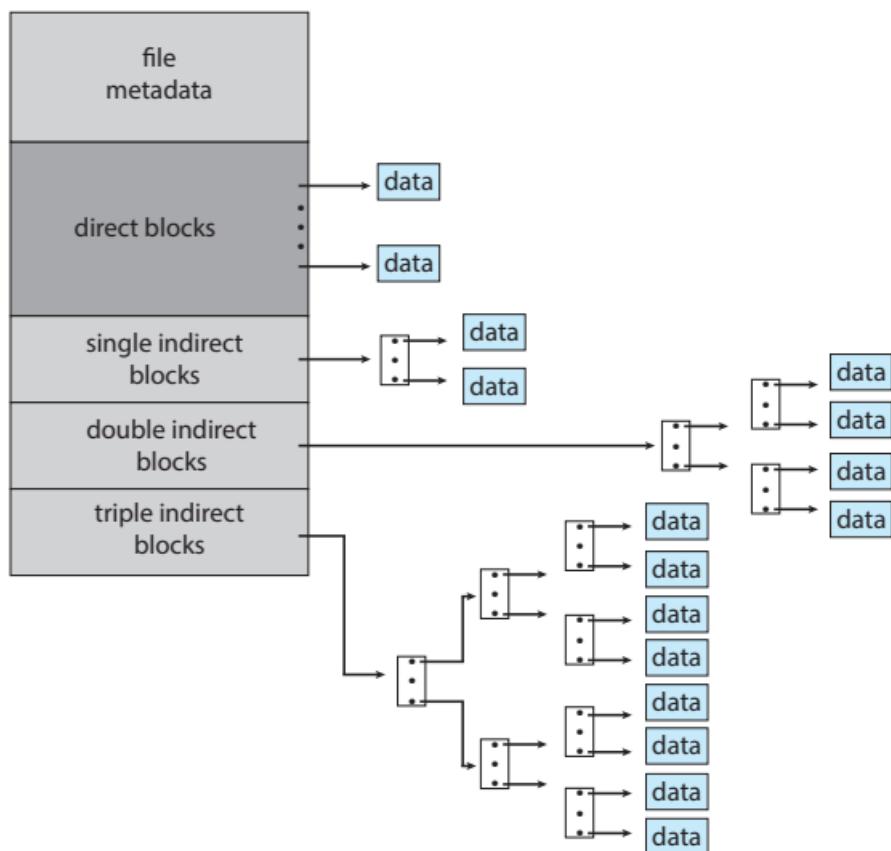
```

int s_inode[MAX_NUMBER_INODE]; //直接管理的空闲外存INode索引表
int s_fmod; //内存中super block副本被修改标志，意味着需要更新外
存
int s_time; //最近一次更新时间
int padding[50]; //填充使SuperBlock块大小等于1024字节，占据2个扇区
};

```

2.2.3 DiskINode 结构

规定了一个 DiskINode 在磁盘上的布局，即当内核将存储 DiskINode 的磁盘块读入之后，必须按照什么格式去解析该盘块中的数据，才能获得 DiskINode 中的文件控制信息。该结构还包括将文件的逻辑块号转换成对应的物理盘块号等功能。



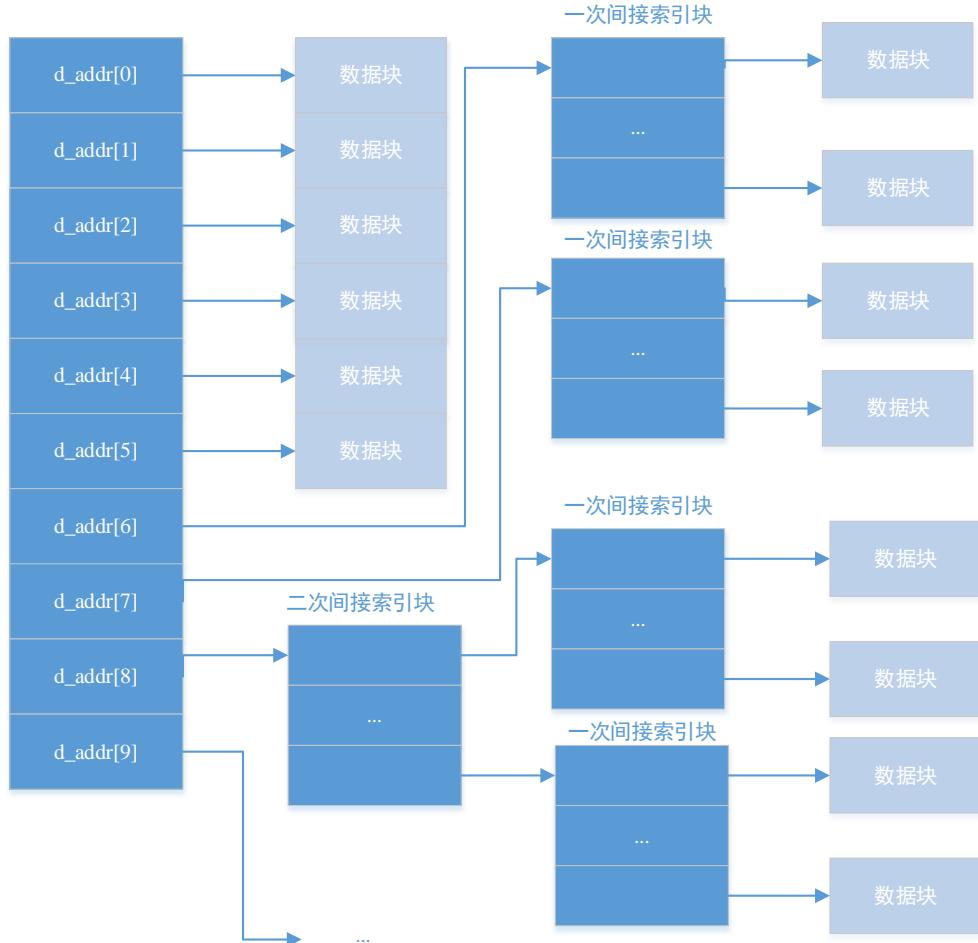
```

class DiskINode
{
public:
    unsigned int d_mode; //状态的标志位，定义见enum INodeFlag
    int d_nlink; //文件联结计数，即该文件在目录树中不同路径名的数量
    int d_size; //文件大小，字节为单位
    int d_addr[10]; //用于文件逻辑块号和物理块号转换的基本索引表
    int d_atime; //最后访问时间
    int d_mtime; //最后修改时间
    int padding;
};

```

2.2.4 索引结构

采用混合索引结构，文件数据保存在磁盘设备上的位置信息包含在 DiskINode 节点中的文件索引表 `d_addr[10]` 数组中，是一系列物理块号的序列，对于大中小文件采用不同的索引方法，分为直接索引、间接索引、二次间接索引。



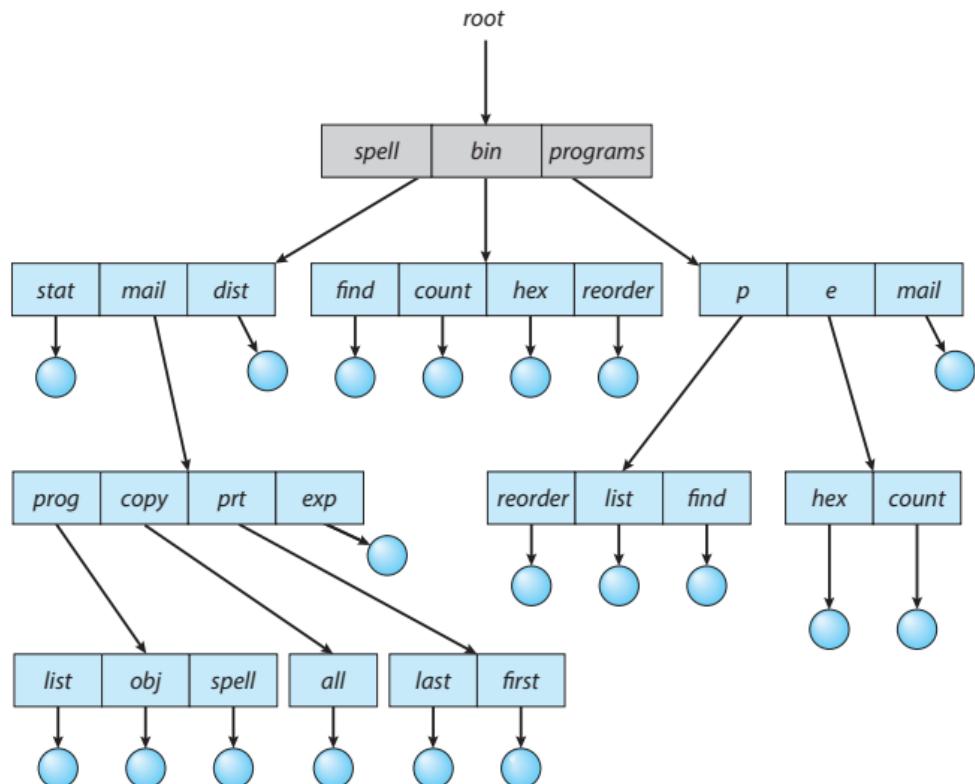
2.2.5 文件目录结构

采用树形带交叉勾连的目录结构。整个目录结构包含有若干个目录文件，每个目录文件由一系列目录项组成。使用 `DirectoryEntry` 数据结构定义目录项。

```
class DirectoryEntry
{
public:
    static const int DIRSIZ = 28; // 目录项中路径部分的最大字符串长度
public:
    int m_ino; // 目录项中 INode 编号部分，即对应文件在块设备上的外存索引节点号
    char name[DIRSIZ]; // 目录项中路径名部分
};
```

`m_ino` 为对应文件在存储设备上的 `DiskINode` 节点号，是文件的内部标识

`name` 为文件名，是文件的外部标识，因而文件目录项为内、外部标识建立了对照关系。



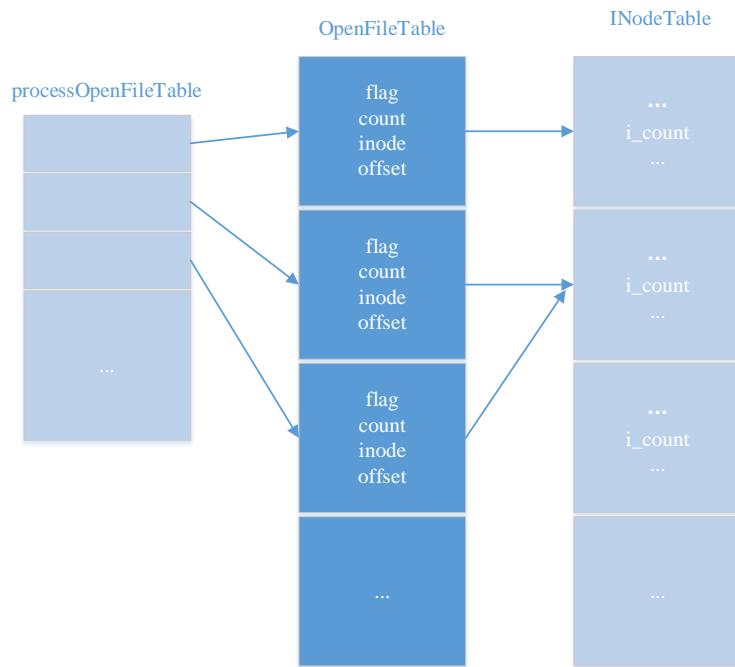
2.2.6 文件打开结构

```

class File
{
public:
    enum FileFlags
    {
        FREAD = 0x1, //读请求类型
        FWRITE = 0x2, //写请求类型
    };
public:
    File();
    ~File();
    void Reset();

    unsigned int flag;//对打开文件的读、写操作要求
    int count; //当前引用该文件控制块的进程数量，若为0则表示该File空闲，可以分配作他用
    INode* inode; //指向打开文件的内存INode指针
    int offset; //文件读写位置指针
};

```



flag 包含了对打开文件请求类型，包括读、写以及管道类型，使用枚举类型 FileFlags 定义。

inode 指向一个打开文件的内存 INode，不同进程打开同一文件会分配各自独立的 File 控制块对象，这些 File 控制块对象的 inode 都会指向同一个内存 INode。

offset 是相对应打开文件进行读、写的位置指针。文件刚打开时，读、写指针位置初始值为 0，每次读、写后，都将其移到已读、写部分的下一个字节。

2.2.7 高速缓存结构

每一个缓存块都会对应一个缓存控制块，它会指明这个缓存块所在的队列位置。由于本次课程设计中不会存在多个设备，于是取消了所有的设备队列，缓存块只存在于 NODEV 队列中。分配和释放的操作也非常简单，分配只是简单的从队列头取第一个缓存块，释放时将该缓存块标志位置换后放在队列尾部。

```

class CacheBlock
{
public:
    //flags中的标志位
    enum CacheBlockFlag
    {
        CB_DONE = 0x1,      //I/O操作结束
        CB_DELWRI = 0x2    //延迟写，对应的缓存有其他用途时，将其内容写到对应的块设备上
    };
    unsigned int flags;   //缓存控制块标志位

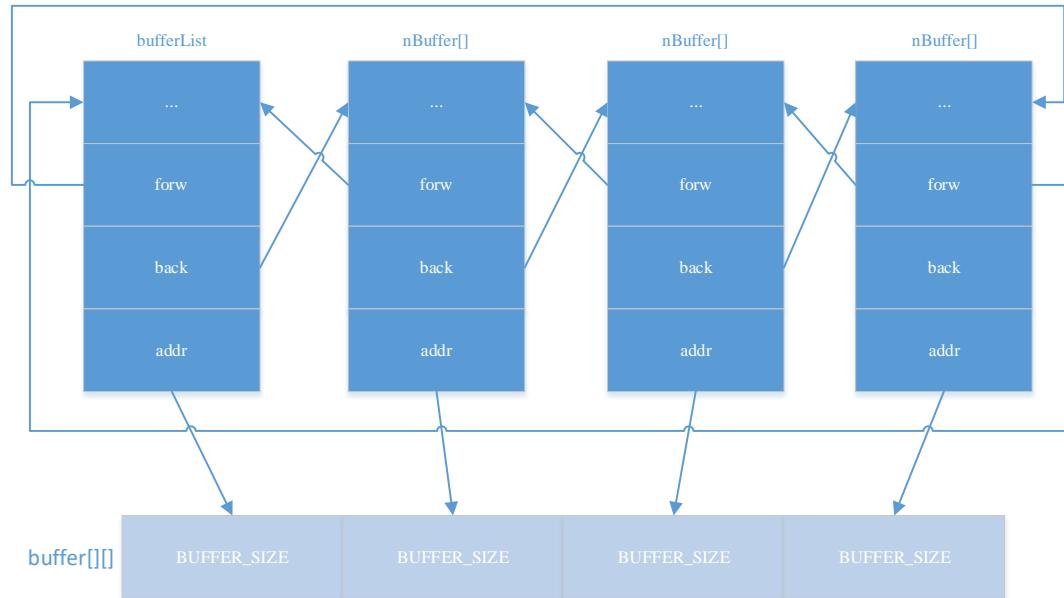
    CacheBlock* forw;
    CacheBlock* back;
}

```

```

int wcount;           //需传送的字节数
unsigned char* addr; //指向该缓存控制块所管理的缓冲区的首地址
int blkno;           //磁盘逻辑块号
int no;
};

```

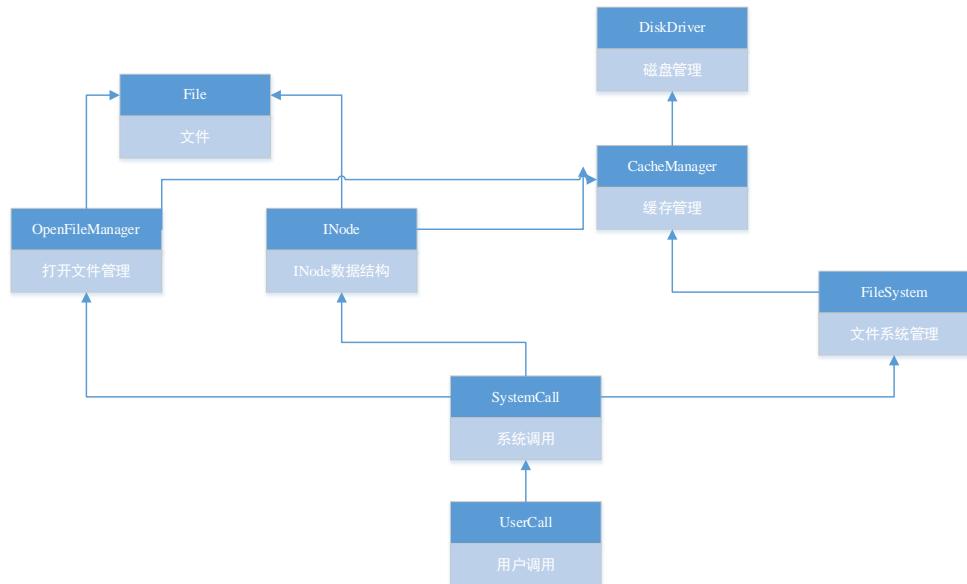


b_wcount: 记录本次 I/O 请求需要传送的字节数，通常该值等于一个缓冲区的大小，即 512 字节

addr: 指向该缓存控制块所管理的缓冲存储区的首地址，即令一个缓存控制块与一个缓冲存储区建立对应关系

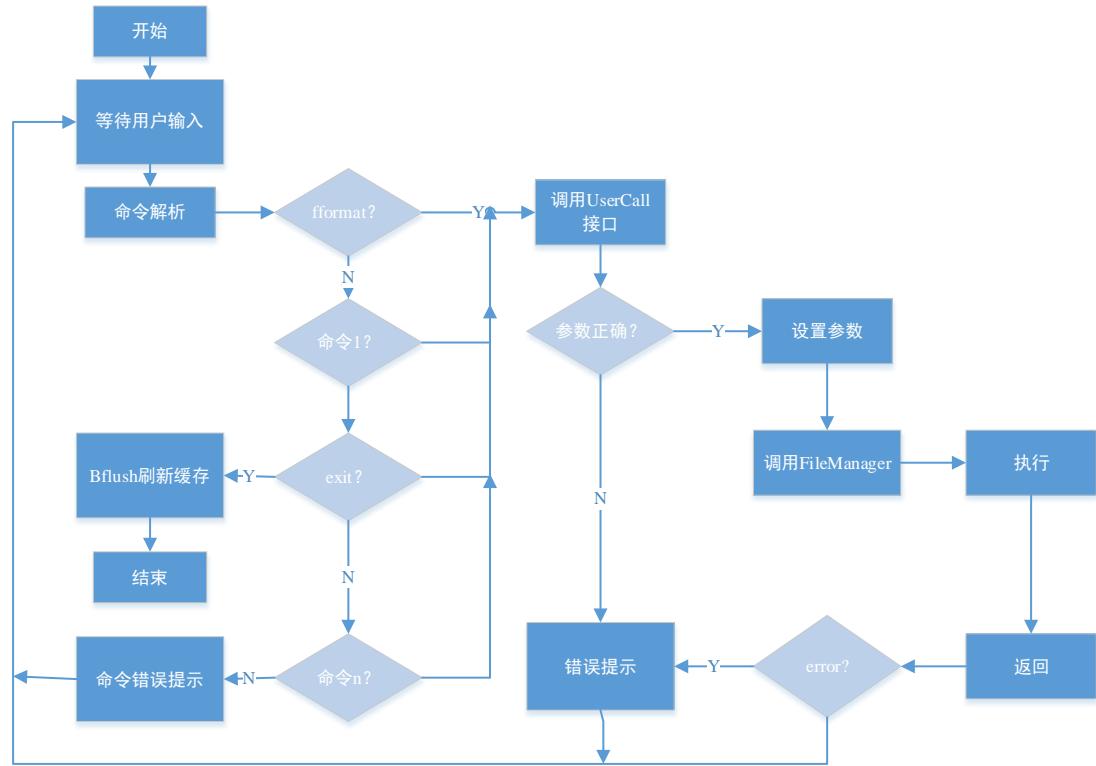
blkno: 磁盘逻辑块号，该变量指定缓冲区中的数据对应磁盘中哪一个扇区的数据

2.3 模块间的调用关系



2.4 主程序流程与算法说明

程序的总体流程为：main 函数等待用户输入，用户输入后，对其命令进行简单解析，然后由命令查找 UserCall 提供的操作接口，将参数交由 User 对象进行处理和判断合法性，若基本符合命令的参数约定，则由 UserCall 调用 SystemCall 中功能函数实现文件系统的具体功能。



3 详细设计

3.1 重点函数与重点变量

INodeTable 类

重点函数或变量	功能与意义
INode m_INodeTable[NINODE]	内存 INode 数组, 每个打开文件都会占用一个内存 INode
INode* IGet(int inumber)	根据外存 INode 编号获取对应 INode。如果该 INode 已经在内存中, 返回该内存 INode; 如果不在内存中, 则将其读入内存后上锁并返回该内存 INode
void IPut(INode* pNode)	减少该内存 INode 的引用计数, 如果此 INode 已经没有目录项指向它, 且无进程引用该 INode, 则释放此文件占用的磁盘块
void UpdateINodeTable()	将所有被修改过的内存 INode 更新到对应外存 INode 中
int IsLoaded(int inumber)	检查编号为 inumber 的外存 INode 是否有内存拷贝, 如果有则返回该内存 INode 在内存 INode 表中的索引
INode* GetFreeINode()	在内存 INode 表中寻找一个空闲的内存 INode

OpenFileTable 类

重点函数或变量	功能与意义
File sysFileTable[MAX_FILES]	系统打开文件表, 为所有进程共享
File* FAlloc()	文件描述符表中包含指向打开文件表中对应 File 结构的指针
void CloseF(File* pFile)	在系统打开文件表中分配一个空闲的 File 结构 对打开文件控制块 File 结构的引用计数 count 减 1, 若引用计数 count 为 0, 则释放 File 结构

ProcessOpenTable 类

重点函数或变量	功能与意义
File* processOpenFileTable[MAX_FILES]	File 对象的指针数组, 指向系统打开文件表中的 File 对象
int AllocFreeSlot()	进程请求打开文件时, 在打开文件描述符表中分配一个空闲表项
void SetF(int fd, File* pFile)	为已分配到的空闲描述符 fd 和已分配的打开文件表中空闲 File 对象建立勾连关系

CacheManager 类

重点函数或变量	功能与意义
CacheBlock* bufferList	Block 块大小, 单位字节
CacheBlock nBuffer[NBUF]	缓存控制块数组
buffer[NBUF][BUFFER_SIZE]	缓冲区数组
CacheBlock* GetBlk(int blkno)	申请一块缓存, 用于读写设备上的块 blkno

void Brelse(CacheBlock* bp)	释放缓存控制块 buf
CacheBlock* Bread(int blkno)	读一个磁盘块, blkno 为目标磁盘块逻辑块号
void Bwrite(CacheBlock* bp)	写一个磁盘块
void Bdwrite(CacheBlock* bp)	延迟写磁盘块
void Bflush()	将队列中延迟写的缓存全部输出到磁盘

FileSystem 类

重点函数或变量	功能与意义
BLOCK_SIZE = 512	Block 块大小, 单位字节
DISK_SECTOR_NUMBER = 16384	磁盘所有扇区数量 $8\text{MB} / 512\text{B} = 16384$
SUPERBLOCK_START_SECTOR = 0	定义 SuperBlock 位于磁盘上的扇区号
INODE_START_SECTOR = 2	外存 INode 区位于磁盘上的起始扇区号
INODE_SECTOR_NUMBER = 1022	磁盘上外存 INode 区占据的扇区数
INODE_SIZE = 64	INode 块大小
INODE_NUMBER_PER_SECTOR = 8	每个磁盘块可以存放 $512/64 = 8$ 个外存 INode
ROOT_INODE_NO = 0	文件系统根目录外存 INode 编号
INODE_NUMBER_ALL = INODE_SECTOR_NUMBER * INODE_NUMBER_PER_SECTOR	外存 INode 的总个数
DATA_START_SECTOR = INODE_START_SECTOR + INODE_SECTOR_NUMBER	数据区的起始扇区号
DATA_END_SECTOR = DISK_SECTOR_NUMBER - 1	数据区的最后扇区号
DATA_SECTOR_NUMBER = DISK_SECTOR_NUMBER - DATA_START_SECTOR	数据区占据的扇区数量
void LoadSuperBlock()	系统初始化时读入 SuperBlock
void Update()	将 SuperBlock 对象的内存副本更新到存储设备的 SuperBlock 中
INode* IAlloc()	在存储设备上分配一个空闲外存 INode, 用于创建新的文件
void IFree(int number)	释放编号为 number 的外存 INode, 一般用于删除文件
CacheBlock* Alloc()	在存储设备上分配空闲磁盘块
void Free(int blkno)	释放存储设备上编号为 blkno 的磁盘块

SystemCall 类

重点函数或变量	功能与意义
void Open()	Open() 系统调用处理过程
void Creat()	Creat() 系统调用处理过程
void Open1(INode* pINode, int trf)	Open()、Creat() 系统调用的公共部分
void Close()	Close() 系统调用处理过程

void Seek()	Seek()系统调用处理过程
void Read()	Read()系统调用处理过程
void Write()	Write()系统调用处理过程
void Rdwr(enum File::FileFlags mode)	读写系统调用公共部分代码
INode* NameI(enum DirectorySearchMode mode)	目录搜索，将路径转化为相应的 INode 返回上锁后的 INode
INode* MakNode(int mode)	被 Creat() 系统调用使用，用于为创建新文件分配内核资源
void UnLink()	取消文件
void WriteDir(INode* pINode)	向父目录的目录文件写入一个目录项
void ChDir()	改变当前工作目录
void Ls()	列出当前 INode 节点的文件项
void Rename(string ori, string cur)	重命名文件、文件夹

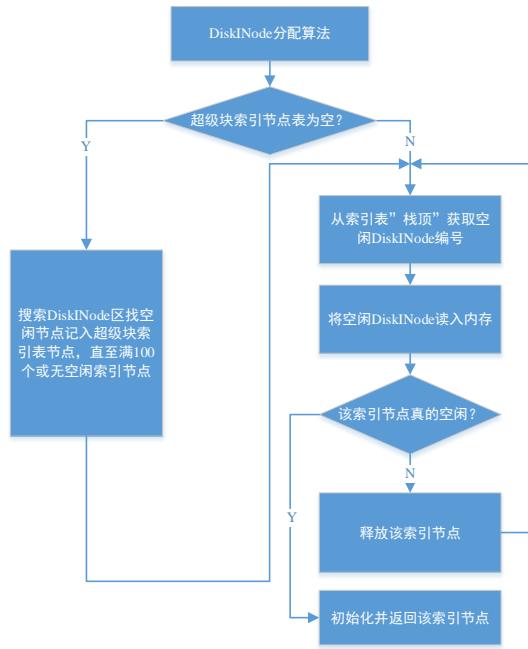
UserCall 类

重点函数或变量	功能与意义
INode* nowDirINodePointer	指向当前目录的 Inode 指针
INode* paDirINodePointer	指向父目录的 Inode 指针
DirectoryEntry dent	当前目录的目录项
char dbuf[DirectoryEntry::DIRSIZ]	当前路径分量
string curDirPath	当前工作目录完整路径
string dirp	系统调用参数(一般用于 Pathname)的指针
int arg[5]	存放当前系统调用参数
uint32 ar0[5]	指向核心栈现场保护区中 EAX 寄存器
ErrorCode userErrorCode	存放错误码
ProcessOpenFile ofiles	进程打开文件描述符表对象
IOParameter IOParam	记录当前读写文件的偏移量，用户目标区域和剩余字节数参数
string ls	调用 Ls() 函数时返回值

3. 2 DiskINode 节点的分配与回收

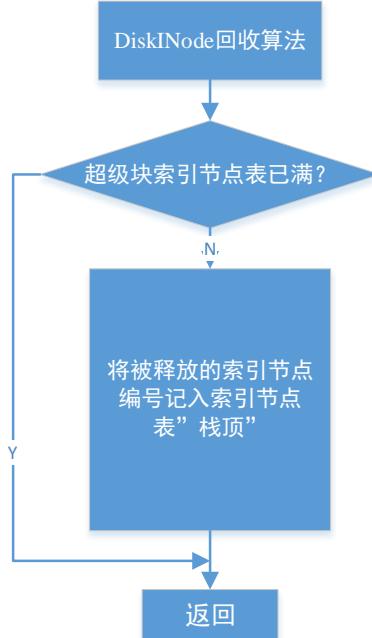
3. 2. 1 DiskINode 节点分配算法

采用栈式管理，如果栈不为空，就直接分配。如果栈顶指针为空，遍历 DiskINode 区，将找到的空闲节点压入栈中。



3.2.2 DiskINode 节点回收算法

当释放一个 DiskINode 节点时,如果超级块索引节点表中空闲 DiskINode 看的个数小于 100,则将该索引节点编号记入“栈顶”;否则直接返回, 不做操作。



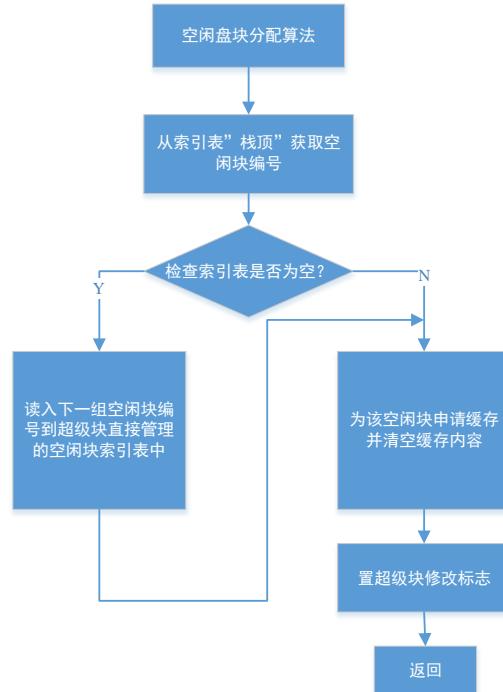
3.3 文件数据区的分配与回收

超级块用栈方式管理空闲数据块, 若空闲块数超过 100, 采用分组链式索引法对它们进行管理。

3.3.1 空闲盘块分配算法

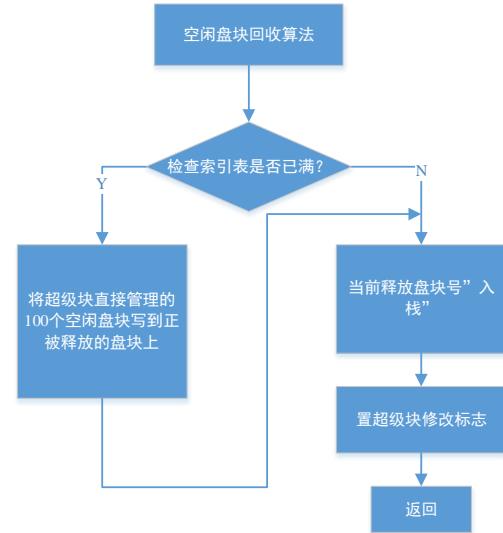
采用栈式管理, 并且采用分组链式索引法。SuperBlock 块最多只能直接管理 100 个空闲块。

因此将空闲块按照每 100 个分为一组，进行管理。SuperBlock 块管理第一组空闲块，各组的索引表存放在它们上一组第一个盘块的开头中。当分配空闲盘块的时候，将当前栈顶内容弹出，如果当前栈只剩下一个索引，则换入下一组进行管理。



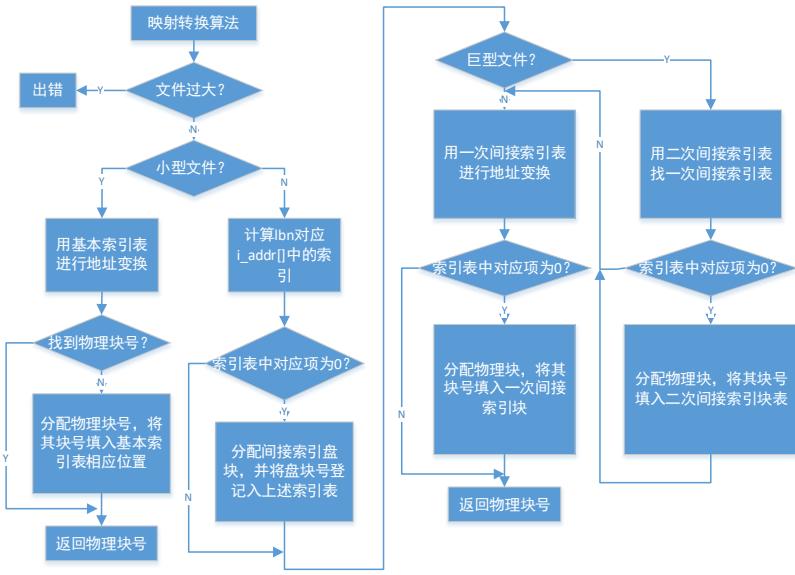
3.3.2 空闲盘块回收算法

进行回收时，如果当前的栈不满，则直接弹入栈中。否则，新增一组空闲块，并加在上一级结尾。



3.4 文件索引和数据块的映射转换

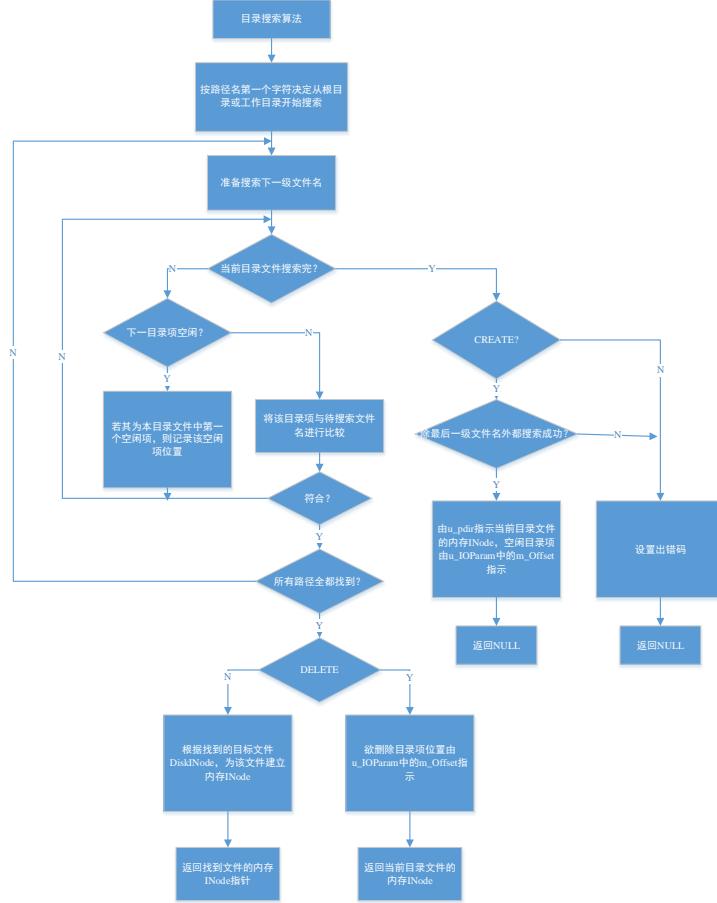
通过判断文件大小，如果是小文件，直接在索引表得到数据块索引。如果是中型文件，首先根据索引表得到一级间接索引表，然后得到数据块索引。如果是大型文件，首先根据索引表得到二级间接索引表，再通过二级间接索引表得到一级间接索引表，从而得到数据块索引。



3.5 目录搜索算法的实现

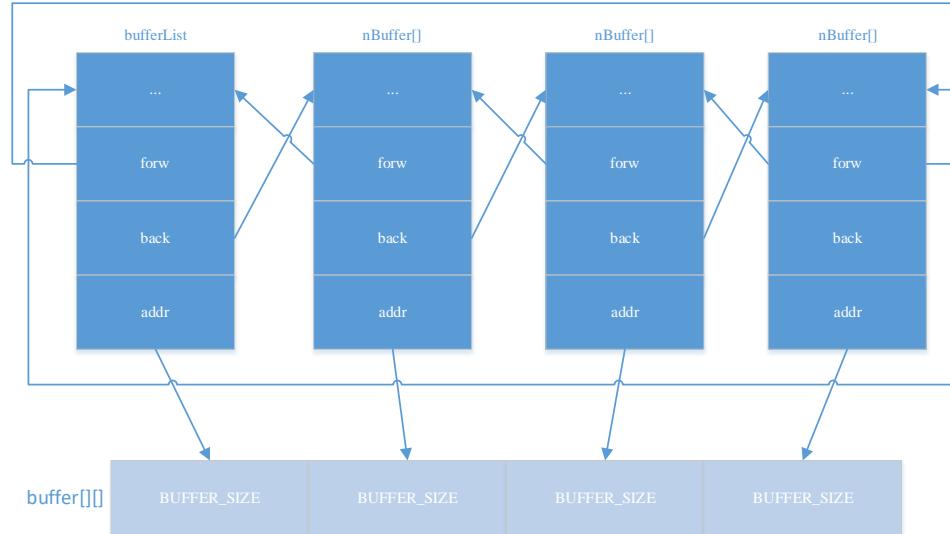
用户程序对文件的存取通过路径名字符串进行，而内核中对文件进行操作使用的是文件的索引节点号，目录搜索的目的就是完成二者的转换。

本次文件系统设计中执行目录搜索的程序是 FileManager::NameI() 函数，它负责将用户给出的路径名字符串转化为对应文件的索引节点，大致流程如图所示。



3.6 高速缓存的模拟和实现

使用双向循环链表模拟缓存队列，实现缓存的分配和释放等操作，第一个缓存是头节点保持不变。本次课设中没有多余的设备驱动，所以经过考虑，可以将自由队列和设备队列进行合并为一个缓存队列，CacheManager 将直接管理 100 块缓存。做初始化时，缓存块将被初始化在缓存队列中，队列头为 bfreelist，队列的形式如下图所示：



3.6.1 缓存淘汰算法

本次文件系统设计采用的缓存淘汰算法是较为精确的 LRU 算法，为了使得一个已被释放的缓存尽可能长地保持原来的使用状态，将它送入自由缓存队列的尾部，而分配缓存时又从自由缓存队列首部取出。

当一个缓存在自由队列内移动时，只要有按原状使用它的需求，就立即将他从自由队列中抽出。当它再次被释放时又进入自由缓存队列末尾。

这就保证了在所有自由缓存中，淘汰最后一次使用时间离现在时刻最远的一个缓存内容。

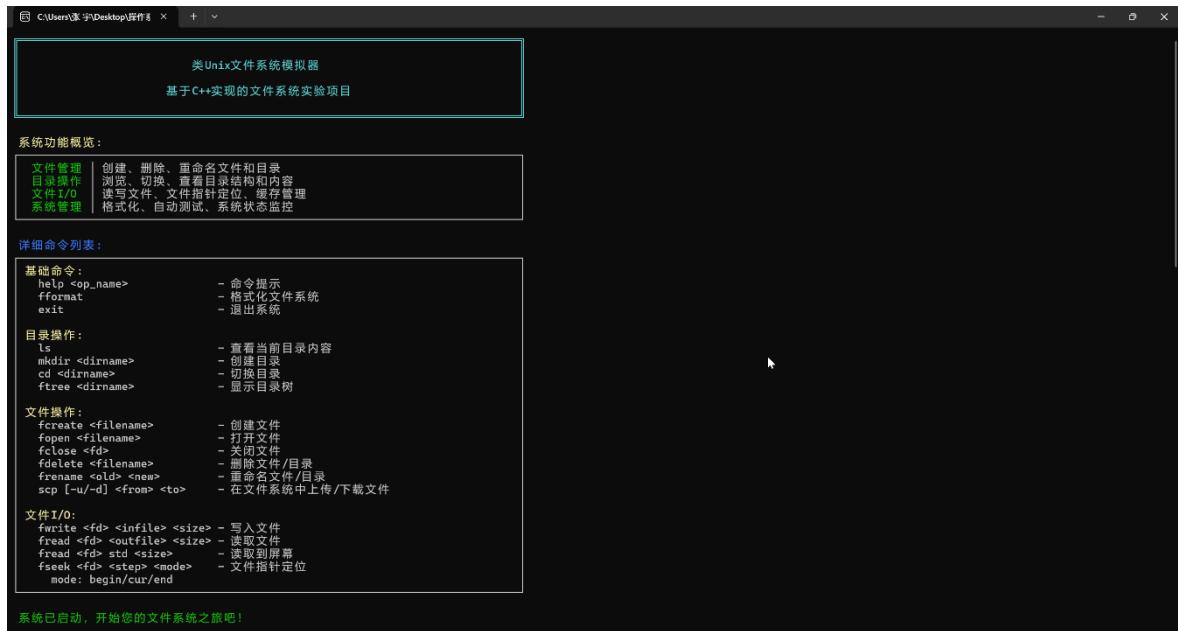
3.6.2 缓存分配算法

申请一块缓存，将其从缓存队列中取出，用于读写设备块上的 blkno。执行过程为：首先从缓存队列中寻找是否有缓存块的 blkno 为目标 blkno，如有则分离该缓存节点，并返回该节点；没有找到说明缓存队列中没有相应节点为 blkno，需要分离第一个节点，将其从缓存队列中删除。若其带有延迟写标志，则立即写回，清空标志，将缓存 blkno 置为 blkno，返回该缓存块。

4 运行结果分析

4.1 程序运行展示说明

4.1.1 程序主页面



4.1.2 程序帮助页面

该板块包括科研、教学、办公和研究生四个类别的计算机：

```
[FileSystem /] $ help
命令帮助手册：

基础命令：
  help <op_name>          - 命令提示
  fformat                  - 格式化文件系统
  exit                     - 退出系统

目录操作：
  ls                       - 查看当前目录内容
  mkdir <dirname>           - 创建目录
  cd <dirname>              - 切换目录
  ftree <dirname>            - 显示目录树

文件操作：
  fcreate <filename>        - 创建文件
  fopen <filename>           - 打开文件
  fclose <fd>                - 关闭文件
  fdelete <filename>         - 删除文件/目录
  frename <old> <new>        - 重命名文件/目录
  scp [-u/-d] <from> <to>    - 在文件系统中上传/下载文件

文件 I/O:
  fwrite <fd> <infile> <size> - 写入文件
  fread <fd> <outfile> <size> - 读取文件
  fread <fd> std <size>       - 读取到屏幕
  fseek <fd> <step> <mode>    - 文件指针定位
  mode: begin/cur/end

[FileSystem /] $ |
```

4.1.3 测试命令及输出结果

(1) 上传课设报告、ReadMe.txt 和一张图片

```
[FileSystem /home/texts] $ scp -u test_files/ReadMe.txt ReadMe.txt
FileBytes:1586
ReadMe.txt: No such a file or directory
1590 1586
File pointer:1586
[FileSystem /home/texts] $ ls
Readme.txt      2025-06-20 21:12:53      0B
ReadMe.txt      2025-06-20 21:12:53      1586B

[FileSystem /home/texts] $ cd ..
[FileSystem /home] $ cd reports
[FileSystem /home/reports] $ scp -u test_files/report.docx report.docx
FileBytes:10102
report.docx: No such a file or directory
5 10102
[WARNING] nbytes can not be greater than the length of the string
File pointer:10102
[FileSystem /home/reports] $ ls
report.docx      2025-06-20 21:12:54      10102B

[FileSystem /home/reports] $ cd ..
[FileSystem /home] $ cd photos
[FileSystem /home/photos] $ scp -u test_files/img.jpg img.jpg
FileBytes:113843
img.jpg: No such a file or directory
4 113843
[WARNING] nbytes can not be greater than the length of the string
File pointer:113843
[FileSystem /home/photos] $ ls
img.jpg 2025-06-20 21:12:58      113843B
```

(2) 新建文件/test/Jerry, 打开该文件, 任意写入 800 个字节

(3) 将文件读写指针定位到第 500 字节, 读出 500 个字节到字符串 abc

```
[FileSystem /test] $ fopen Jerry
File Handle fd = 0
[FileSystem /test] $ flseek 0 500 0
[FileSystem /test] $ cp Jerry abc
File Handle fd = 3
File pointer:500
abc: No such a file or directory
File Handle fd = 3
File pointer:0
Successfully write 0 bytes
Successfully copied 0 Byte
[FileSystem /test] $ ls
Jerry    2025-06-20 21:17:51      0B
abc      2025-06-20 21:17:51      0B
```

(3) 将 abc 写回文件

4.2 测试结果分析

4.2.1 新建文件/test/Jerry，打开该文件，任意写入 800 个字节

分析：该写操作将分两次盘块写操作完成

(1) 第一次:

1. 通过当前文件读写指针的位置 0 计算得到需要写的逻辑块号为 0, 块内偏移地址为 0 本次需写入的字节数为 512

2. 通过 `i_addr` 查询 Jerry 文件 0 号逻辑块所在的物理盘块号 0, 即: 一个新块, 所以首先申请一个新的盘块(干净的盘块), 假设分配的盘块号为 124, 将该盘块号登记入 Jerry 文件的 `i_addr` 数组;

3. 第一次是写完整的一块，因此不需要将 124 号盘块读入；
 4. 将前 512 个字节写入该缓存的 0 到 511 字节；
 5. 修改读写指针为 512，因为刚好写满一块，所以执行异步写操作将缓存内容写入磁盘；

(2) 第二次:

1. 通过当前文件读写指针的位置 512 计算得到需写的逻辑块号为 1，块内偏移地址为 0，本次需写入的字节数为 288；

2. 通过 `i_addr` 查询 Jerry 文件 1 号逻辑块所在的物理盘块号 0，即：一个新块，所以首先申请一个新的盘块（干净的盘块），假设分配的盘块号为 125，将该盘块号登记入 Jerry 文件的 `i_addr` 数组；

3. 因为 $288 < 512$, 即不是写完整的一块, 所以需将 125 号盘块读入;
 4. 对该物理盘块申请一个缓存
 5. 将后 288 个字节写入该缓存的前 288 个字节;
 6. 修改读写指针为 800, 因为一块未写满, 所以将该缓存添加 B_DELWRI 标志后直接释放;
 7. 修改文件长度为 800。

4.2.2 将文件读写指针定位到第 500 字节，读出 500 个字节到字符串 abc

分析：该读操作将分两次盘块读操作完成

(1) `seek (fd, 500, 0)`; 文件的读写指针被定位在第 500 个字节

(2) int count = read (fd, abc, 500); 该读操作将分两次盘块读操作完成。

(3) 第一次:

1. 通过当前文件读写指针的位置 500 计算得到需读取的逻辑块号为 0;
 2. 通过 `i_addr` 查询 Jerry 文件 0 号逻辑块所在物理盘块号为 124。
 3. 对该物理盘块申请一个缓存

4. 将该缓冲区中 500 到 511 字节的内容复制到 abc 数组;

5. 释放缓存

6. 修改文件读写指针为 512。

(4) 第二次:

1. 通过当前文件读写指针的位置 512 计算得到需读取的逻辑块号为 1

2. 通过 i_addr 查询 Jerry 文件 1 号逻辑块所在的物理盘块号 125;

3. 对该物理盘块申请一个缓存, 发现 125 号盘块的缓存在设备队列中, 且有 B_DONE 标志, 即该缓存中的内容可以直接使用。

4. 因为文件到该缓存中的第 288 字节就结束了, 所以将该缓冲区中 0 到 288 字节的内容复制到 abc 数组。

5. 释放缓存

6. 修改文件读写指针为 800。

4.2.3 将 abc 写回文件

分析: 该写操作将分两次盘块写操作完成

(1) 第一次:

1. 通过当前文件读写指针的位置 800 计算得到需要写的逻辑块号为 1, 块内偏移地址为 288 本次需写入的字节数为 224

2. 通过 i_addr 查询 Jerry 文件 1 号逻辑块所在的物理盘块号 125

3. 因为 $224 < 512$, 即不是写完整的一块, 所以需将 125 号盘块读入;

4. 对该物理盘块申请一个缓存

5. 将前 224 个字节写入该缓存的 288 到 511 字节;

6. 修改读写指针为 1024, 因为刚好写满一块, 所以执行异步写操作将缓存内容写入磁盘;

(2) 第二次:

1. 通过当前文件读写指针的位置 1024 计算得到需写的逻辑块号为 2, 块内偏移地址为 0, 本次需写入的字节数为 76;

2. 通过 i_addr 查询 Jerry 文件 2 号逻辑块所在的物理盘块号 0, 即: 一个新块, 所以首先申请一个新的盘块(干净的盘块), 假设分配的盘块号为 126, 将该盘块号登记入 Jerry 文件的 i_addr 数组;

3. 因为 $76 < 512$, 即不是写完整的一块, 所以需将 126 号盘块读入;

4. 对该物理盘块申请一个缓存

5. 将后 76 个字节写入该缓存的前 76 个字节;

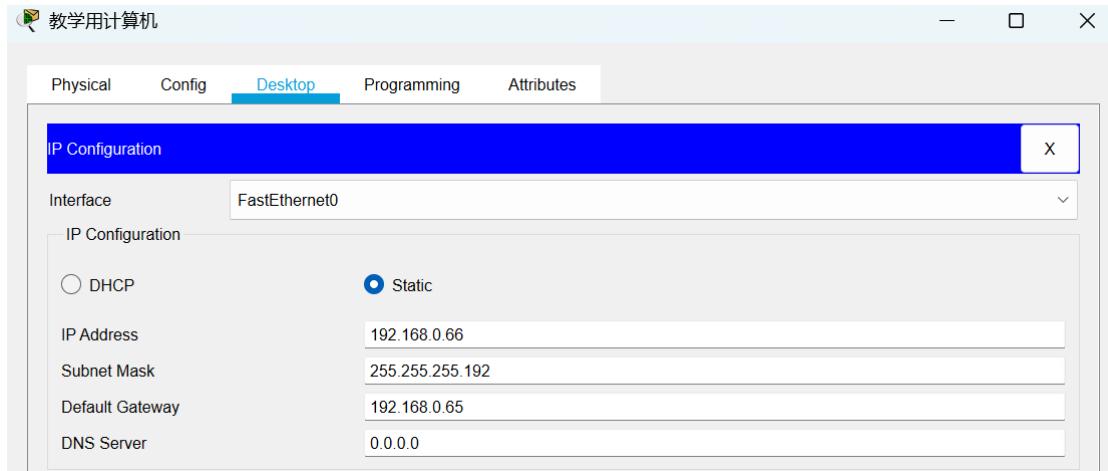
6. 修改读写指针为 1100, 因为一块未写满, 所以将该缓存添加 B_DELWRI 标志后直接释放;

7. 修改文件长度为 1100。

5 用户使用说明

5.1 运行说明

在 Windows 下双击生成的 FileSystem.exe 文件执行，运行界面为控制台的命令行方式，命令较为简单，通俗易懂，初始界面如下。



5.2 注意事项

格式化：格式化命令为 `format`，运行命令后程序会进行文件系统格式化，然后正常退出，此时需要再次进入，才能得到初始环境。

正确退出：最好通过 `exit` 命令正确退出程序，这样退出会调用 CacheManager 类中的 `Bflush` 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。

6 实验总结

本次操作系统课程设计的重点是实现一个类 UNIX 的二级文件系统，模拟 UNIX V6++文件系统的操作。通过这个实验，我深入理解了操作系统中文件管理、磁盘管理、缓存管理等方面的核心原理，并通过实际编程加深了对这些概念的掌握。

首先，文件系统的基本设计包括了磁盘结构、SuperBlock 结构、INode 节点的管理、文件数据区的分配与回收等内容。在设计过程中，我通过对磁盘块（512 字节）进行分配和管理，确保了文件的存储效率和稳定性。磁盘管理是操作系统中非常重要的部分，在这个过程中，我实现了超级块的管理，确保了对 INode 区和文件数据区的正确分配与回收。

在实现文件管理的过程中，我根据文件系统的要求设计了目录结构。文件系统使用树形结构来管理目录，并且为每个目录项分配一个 INode 节点。目录项的管理和检索是文件系统中不可或缺的部分，通过设计和实现高效的目录检索算法，使得文件系统能够快速定位文件，实现对文件的增、删、改、查等操作。

其次，缓存管理系统的工作是本次实验的一个重要部分。为了提高文件系统的读写效率，我实现了一个基于 LRU（最少使用）算法的缓存管理模块。该模块通过高速缓存的使用，减少了直接与磁盘交互的次数，从而大大提高了系统的响应速度。在缓存管理中，我特别注意了缓存的分配与回收机制，确保了缓存块的合理使用，并通过延迟写入策略优化了磁盘写入的性能。

在文件系统操作的实现上，我完成了文件的创建、打开、读写、关闭等基础功能，同时还设计了文件指针定位和文件内容修改的功能。在文件的读写操作中，我通过计算文件块号和偏移量，实现了文件内容的高效存取。在开发过程中，我特别注意了内存与磁盘之间的数据同步问题，通过合理的策略确保了数据的一致性和可靠性。

实验过程中，我也遇到了一些技术上的难题。例如，在磁盘块的分配与回收过程中，如何合理管理空闲盘块，避免浪费空间并提高效率，成为我需要解决的难题。此外，目录项的管理和检索算法的优化也是我在实现过程中反复思考的问题。通过查阅资料和不断调试，我逐步克服了这些挑战，最终实现了功能完整、稳定的文件系统。

通过这次实验，我不仅学习到了如何设计和实现一个文件系统，还加深了对操作系统内部机制的理解。尤其是在系统性能优化方面，通过高速缓存的设计和文件块管理算法的改进，我体会到了操作系统中资源管理的重要性和复杂性。