同濟大學
TONGJI UNIVERSITY

Tongji University

Department of Computer Science and Technology

类 Unix 文件系统设计实验报告

Author:
2051454 王逸菲

An Assignment submitted for the Tongji University:

CS10043601 操作系统实验

2024 年 5 月 19 日

目录

1 需求分析

本章节详细说明了类 Unix 文件系统设计的程序任务、输入输出形式以及程序的具体功能。

1.1 程序任务

UNIX 操作系统采用层次结构的文件系统来管理文件、目录和设备，这是其核心组件之一。本课程设计旨在通过模拟 UNIX V6++ 操作系统的文件系统，实现一个类 UNIX 的文件系统。此项目不仅旨在实现操作系统的基本文件操作功能，而且还致力于深化对文件系统结构和操作的基本及全面理解。

本实验使用一个大型文件（如 `D:/myDisk.img`，以下称为" 一级文件"）来模拟 UNIX V6++ 的磁盘环境。该虚拟磁盘的数据存储以 512 字节为单位的块进行组织。结构如下:

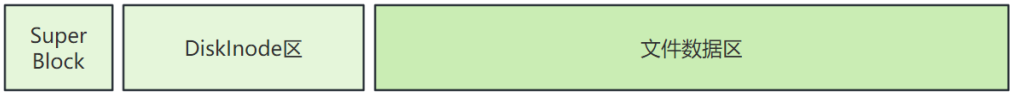

图 1: UNIX 文件系统的静态结构

1.2 程序输入和输出

1.2.1 Cmd 方式

用户通过类 Linux 指令进行交互，如 `fformat`、`mkdir`、`cd`、`ls` 、`fcreate`、`fopen`、`fseek`、`fclose` 等。

- **输入**：通过命令行输入指令。
- **输出**：返回操作结果，包括句柄输出、提示信息、错误信息以及文件内容。

1.3 程序功能需求

1.3.1 文件系统功能

下表列出了文件系统的各种用户接口命令及其功能。

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> <i>`	以 i(begin/cur/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
`exit`	退出系统，并将缓存内容存至磁盘

表 1: 文件系统用户接口命令表

1.3.2 系统调用

通过 UserCall 类中的函数来整理，检查以及传递参数，SystemCall 类的接口处理系统调用，使用参数和返回值来传递信息，确保系统的稳定性和安全性。

1.4 性能需求

- **数据精度**：确保数据的准确存储和完整性，防止任何数据损失。
- **时间特性**：读写操作的响应时间应不超过 100ms，利用高速缓存提高效率。

1.5 用户界面需求

提供简洁、直观的用户界面，支持直接在屏幕上输出结果，也可将结果保存至文件，便于用户操作和数据访问。

1.6 系统可行性

- **开发和测试环境**：使用 C++ 语言和 Visual Studio Community 2022 进行开发，Windows 11 作为操作系统进行测试。

2 概要设计

2.1 程序分析与模块划分

本节详细描述了程序的各个组件及其功能，这些组件共同构成了模拟的类 UNIX 文件系统。

- **UserCall 模块**：实现用户界面层，提供命令接口。管理用户的当前目录、父目录及进程打开的文件描述表。负责错误处理，打印错误输出如 “File or directory not found”，“File handle not found” 的错误提示，并与系统调用模块交互，使用 `ar0[5]` `arg[5]` 传递返回值和参数。
- **SystemCall 模块**：处理文件操作相关的系统调用，如 `fformat`, `mkdir`, `fopen`,`fcreate` 等功能的细节实现。用户输入指令，通过 UserCall 进行指令的转化和检查，再通过 SystemCall 模块对文件和内核结构进行相应的读写和编辑操作。
- **OpenFileManager 模块**：
  - **OpenFileTable**：负责文件打开结构，进程打开文件表 (ProcessOpenFile)，系统打开文件表 (OpenFileTable)，内存 Inode 表 (INodeTable) 之间的相互勾连实现。
  - **INodeTable**：内存 Inode 表，负责管理内存 INode 结构的分配与释放。
- **FileSystem 模块**：定义和管理磁盘文件结构，包括 SuperBlock 和 INode 的定义和相关操作。FileSystem 相当于一个总部，定义了各种盘块回收，盘块分配，磁盘和内存 INode 之间交互的函数等等。
- **CacheManager 模块**：负责高速缓存管理，包括缓存块的申请、释放、延迟写入及刷新磁盘数据。
- **DiskDriver 模块**：底层磁盘操作模块，直接负责硬盘文件 `myDisk.img` 的创建、检测、初始化及读写操作。

2.2 数据结构定义

2.2.1 磁盘存储空间管理

- **SuperBlock**：主要负责两个功能：对空闲 inode 的管理，对空闲数据块的管理. 对空闲 inode 的管理，采用了后进先出的栈式管理。对于空闲数据块，采用成组链接法。数据结构定义如下：

```
1   class SuperBlock
2   {
3   public:
4     const static int MAX_NUMBER_FREE = 100;  //最多管理100个空闲数据块
5     const static int MAX_NUMBER_INODE = 100;  //管理空闲inode
6
7
8     /* 成组链接 管理空闲数据块*/
9     int s_fsize;                //盘块总数  16384 - 1024 = 15360
10    int s_nfree;                //直接管理的空闲盘块数量
11    int s_free[MAX_NUMBER_FREE];  //最后一组直接由superblock管理
```

```
12
13      /*栈式管理 inode*/
14      int s_isize;                    //DiskINode区占用的盘块数 1022
15      int s_ninode;                   //直接管理的空闲外存INode数量
16      int s_inode[MAX_NUMBER_INODE]; //直接管理的空闲外存INode索引表
17
18      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
19
20      int s_time;                 //最近一次更新时间
21      int padding[50];              //填充使SuperBlock块大小等于1024字节，占据2个扇区
22
23  };
24
```

- **DiskInode**：是文件系统的外存索引结点。每个文件在 Inode 区有一个外存文件控制块 DiskInode（外存索引结点，64 个字节）是内存 Inode 的磁盘映像，我们系统对索引结点的处理主要是通过内存 Inode 来进行的。所以这里并没有大量存储结点的性质。

```
1   class DiskINode
2   {
3   public:
4     unsigned int d_mode;//状态的标志位
5     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
6     int d_size;       //文件大小，字节为单位
7     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
8     int d_atime;      //最后访问时间
9     int d_mtime;      //最后修改时间
10
11    int padding;          //这是填充的字节不用管它
12
13  public:
14    DiskINode();
15    ~DiskINode();
16  };
17
```

- **索引结构**：我们沿用 UnixV6++ 的设计，采用混合索引结构。对于中型，大型，句型文件都有不同的索引方法。对于中性文件，即采用直接索引的形式，d_addr 中就直接存着数据盘块的物理块号。对于大型文件，采用了一级索引结构，

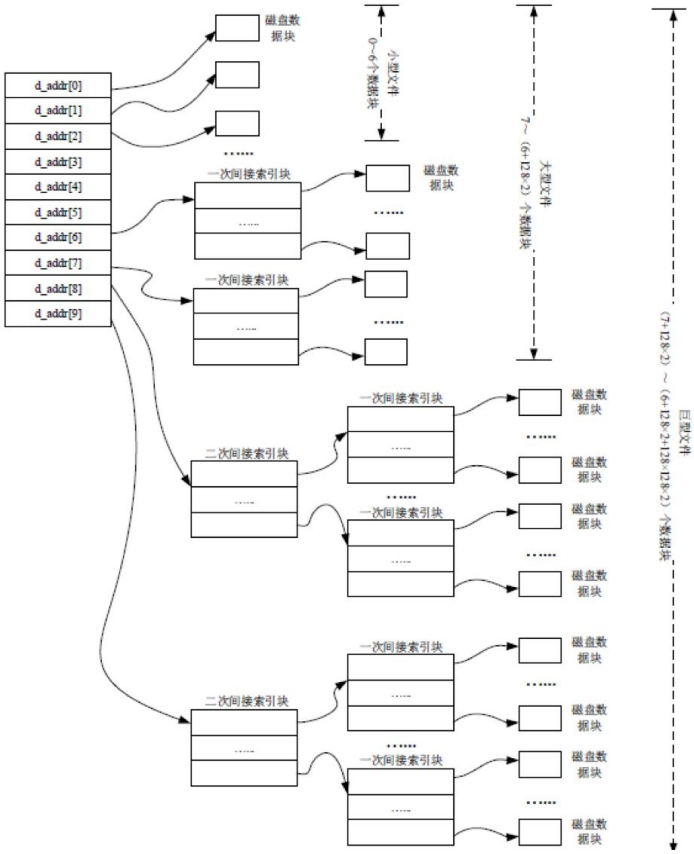

图 2: 混合索引示意图

2.2.2 目录结构

- **DirectoryEntry 目录结构**：类型为文件夹的 Inode 中的 daddr 指向的磁盘数据区，是每 32 字节为一组的目录结构，一个盘块可以储存 512/32=16 个目录区。某个文件夹的数据盘块示意图如下：

m_no Inode号　目录项路径名部分

15	Grants
64	Books
92	mbox
...	...
80	temp

图 3: 目录数据块结构

```
1    /* 目录结构 32字节  4+28 */
2    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
};

```

2.2.3 高速缓存结构

- **CacheManager**：总体管理了所有缓存控制块和缓存块。缓存块我定义了 100 个，每个 512 字节，直接使用了了一个二维数组来存储。而缓存控制块也对应的有 100 个。

  CahceManager 所管理的函数，负责申请/释放一块缓存块，读/写一个缓存块。

```

//关于缓存块的一些函数
class CacheManager
{
public:
    static const int NBUF = 100;            //缓冲区的数量
    static const int BUFFER_SIZE = 512;     //定义了一个buffer的大小，即一个盘块的大小

private:
    CacheBlock* bufferList;  //缓存控制块的头

    CacheBlock nBuffer[NBUF];     //缓存控制块数组

    unsigned char buffer[NBUF][BUFFER_SIZE]; //缓冲区数组 存数据的

        //CacheBlock和物理盘块号的映射关系
    unordered_map<int, CacheBlock*> map;
    DiskDriver* diskDriver;

public:
    CacheManager();
    ~CacheManager();

    CacheBlock* GetBlk(int blkno);  //申请一块缓存，用于磁盘盘块号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

    //清空缓冲区内容
    void Bclear(CacheBlock* bp);

```

```
    //将队列中延迟写的缓存全部输出到磁盘
    void Bflush();

    void FormatBuffer();                    //格式化所有Buffer

  private:

    //处理缓存块队列的函数
    void InitList();                    //缓存控制块队列的初始化
    void DetachNode(CacheBlock* pb);
    void InsertTail(CacheBlock* pb);
  };

```

- **CacheBlock**：缓存控制块，每一个缓存块都对应一个缓存控制块。由于本课程设计的要求是一个设备一个进程，所以仅采用一个缓存队列。要使用之前遍历整个队列，看看有没有能重用的，如果没有能重用的就从队头取出一个。本质上是一个带链表头的双向链表，链表头的目的是让删除和插入能够统一操作。高速缓存的结构图如下所示：

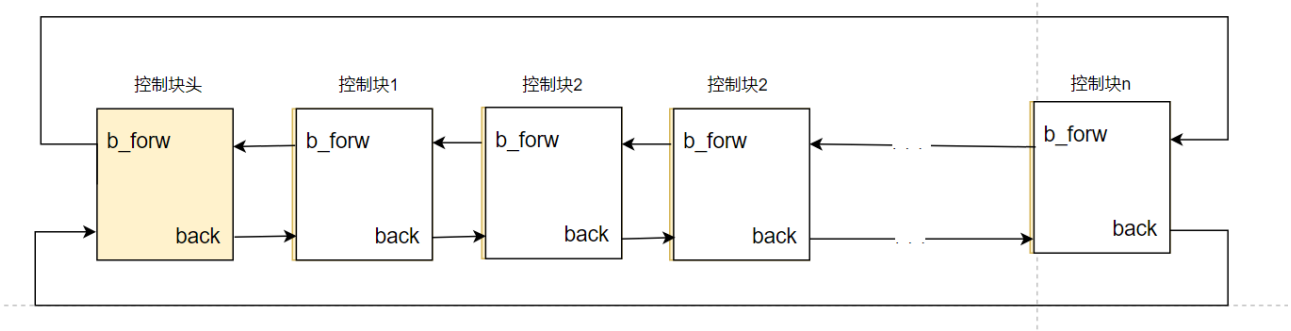

图 4: 高速缓存队列结构

```
//缓存控制块
class CacheBlock
{
public:

  enum CacheBlockFlag
  {
    CB_DONE = 0X1,     //IO操作结束
    CB_DELWRI=0X2      //延迟写标志
  };

  unsigned int flags;

  //磁盘逻辑块号,从而判断可不可以重用
  int b_blkno;
```

```
16
17      //指向该缓存控制块管理的缓冲区首地址
18      unsigned char* b_addr;
19
20      //需传送的字节数
21      int wcount;
22
23      int no;    //在数组中的序号
24
25      //缓存块之间的勾连关系
26      CacheBlock* b_forw;
27      CacheBlock* b_back;
28
29      CacheBlock()
30      {
31        flags = 0;
32        b_forw = NULL;
33        b_back = NULL;
34        wcount = 0;
35        b_addr = NULL;
36        b_blkno = -1;
37        no = 0;
38      }
39
40      void Reset()
41      {
42        flags = 0;
43        b_forw = NULL;
44        b_back = NULL;
45        wcount = 0;
46        b_addr = NULL;
47        b_blkno = -1;
48        no = 0;
49      }
50    };
51
```

2.2.4 文件打开结构

- **内存 Inode**：内存 Inode 和 DiskInode 对应，额外记录了文件的类型，访问标志位，和 DiskInode 的对应关系等等。此处我们没有记录 i_lastr, 和 UNIXV6++，我们只考虑了同步读。Unix V6++ 预读字符块的原则：对某个文件，考察相邻 2 次读操作所在的逻辑块，如果前一次的逻辑块号 + 1 = 当前逻辑块号，判定为顺序读。同步读入当前逻辑块时候，异步读入下一个逻辑块。我们舍弃了这个异步预读字符快的原则。

  打开结构的勾连关系和存储的数据结构如下:

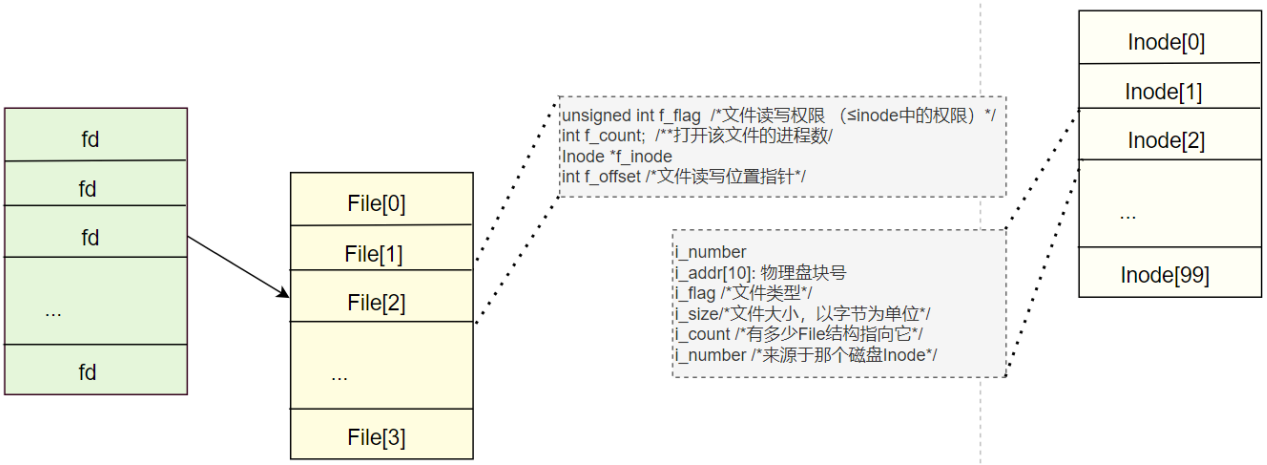

图 5: 文件打开结构

```
class INode
{
public:

/*i_flag标志位*/
enum INodeFlag
{
  IUPD = 0x1,//内存INode被修改过，需要更新对应外存INode
  IACC = 0x2 //内存INode被访问过，需要修改最近一次访问时间

};

static const unsigned int IALLOC = 0x8000;     //文件被使用
static const unsigned int IFMT = 0x6000;       //文件类型掩码
static const unsigned int IFDIR = 0x4000;      //文件类型：目录文件
static const unsigned int ILARG = 0x1000;      //文件长度类型：大型或巨型文件
static const unsigned int IREAD = 0x100;       //对文件的读权限
static const unsigned int IWRITE = 0x80;       //对文件的写权限

static const int BLOCK_SIZE = 512;

static const int ADDRESS_PER_INDEX_BLOCK = BLOCK_SIZE / sizeof(int);

static const int SMALL_FILE_BLOCK = 6;    //小型文件：直接索引表最多可寻址的逻辑块号
//大型文件：经一次间接索引表最多可寻址的逻辑块号
      static const int LARGE_FILE_BLOCK = 128 * 2 + 6;
static const int HUGE_FILE_BLOCK = 128 * 128 * 2 + 128 * 2 + 6;

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_count;     //引用计数 我们这里都可能有
 //所谓一个文件对应一个Inode，但是进程可能以不同的读写权限读取同一个文件，所以一个Inode可能有多个
   File结构（不同的offset，不同的mode）指向他

```

```
    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
    int     i_number;
      //被拷贝到内存的 Inode 中的索引节点数据需要知道它来自于哪个外存 DiskInode，
                            //以便于将来内存副本被修改之后更新到外存对应的 DiskInode 中去
    int     i_size;      //文件大小，字节为单位
    int     i_addr[10];    //用于文件逻辑块号和物理块号转换的基本索引表

      public:

    INode();
    ~INode();

    void Reset()
    {
      i_mode = 0;
      i_count = 0;
      i_number = -1;
      i_size = 0;
      memset(i_addr, 0, sizeof(i_addr));
    }
    void ReadI();                    //根据Inode对象中的物理磁盘块索引表，读取相应的文件数据
    void WriteI();                     //根据Inode对象中的物理磁盘块索引表，将数据写入文件
    int Bmap(int lbn);                        //将文件的逻辑块号转换成对应的物理盘块号
    void IUpdate(int time);                   //更新外存Inode的最后的访问时间、修改时间
    void ITrunc();                            //释放Inode对应文件占用的磁盘块
    void Clean();                             //清空Inode对象中的数据
    void ICopy(CacheBlock* bp, int inumber);
         //将包含外存Inode字符块中信息拷贝到内存Inode中

      };
```

- **File 结构**：File 结构负责管理文件 Inode，一个文件 Inode 可能对应多个 File 结构，因为进程可能多次打开同一个文件，而文件可能有不同的 offset 和读写权限。

```
    class File
    {
    public:
      enum FileFlags
      {
        FREAD = 0x1,   //读
        FWRITE = 0x2, //写
      };

    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
  };

```

- **管理 File 结构的系统打开文件表 OpenFileTable**：用于管理所有 File 结构，本质上是一个 FIle 类型的数组。同时还管理着分配空闲 File 结构，释放 File 结构（关闭文件时会用到）的函数。

```
  class OpenFileTable{
  public:
static const int MAX_FILES = 100;    //打开文件控制块FILE结构的数量

  public:
File sysFileTable[MAX_FILES];        //系统打开文件表，为所有进程共享，进程打开
                    //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

  public:
OpenFileTable();
~OpenFileTable();
void Reset();

//在系统打开文件表中分配一个空闲的File结构
File* FAlloc();

//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
void CloseF(File* pFile);
  };

```

- **进程打开文件表 processOpenFileTable**：主要管理了 File 对象的指针数组，一个指向系统打开文件表中的 File 对象。同时也管理了建立文件打开结构之间勾连关系的函数。

```

  class ProcessOpenFile {
  public:
    static const int MAX_FILES = 100;    //进程允许打开的最大文件数

  private:
        /*File对象的指针数组，指向系统打开文件表中的File对象*/
        File* processOpenFileTable[MAX_FILES];

  public:

    ProcessOpenFile();
    ~ProcessOpenFile();
```

```
14      void Reset()    //重置
15      {
16        memset(processOpenFileTable, 0, sizeof(processOpenFileTable));
17      };
18            /*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  应该会返回一个int fd*/
19      int AllocFreeSlot();
20             /*根据用户系统调用提供的文件描述符参数fd，找到对应的打开文件控制块File结构 */
21      File* GetF(int fd);
22            /*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*/
23      void SetF(int fd, File* pFile);
24    };
25
```

2.3 模块间的调用关系

文件系统的设计体现了清晰的层级调用结构，各模块之间的依赖和调用关系如下所述：

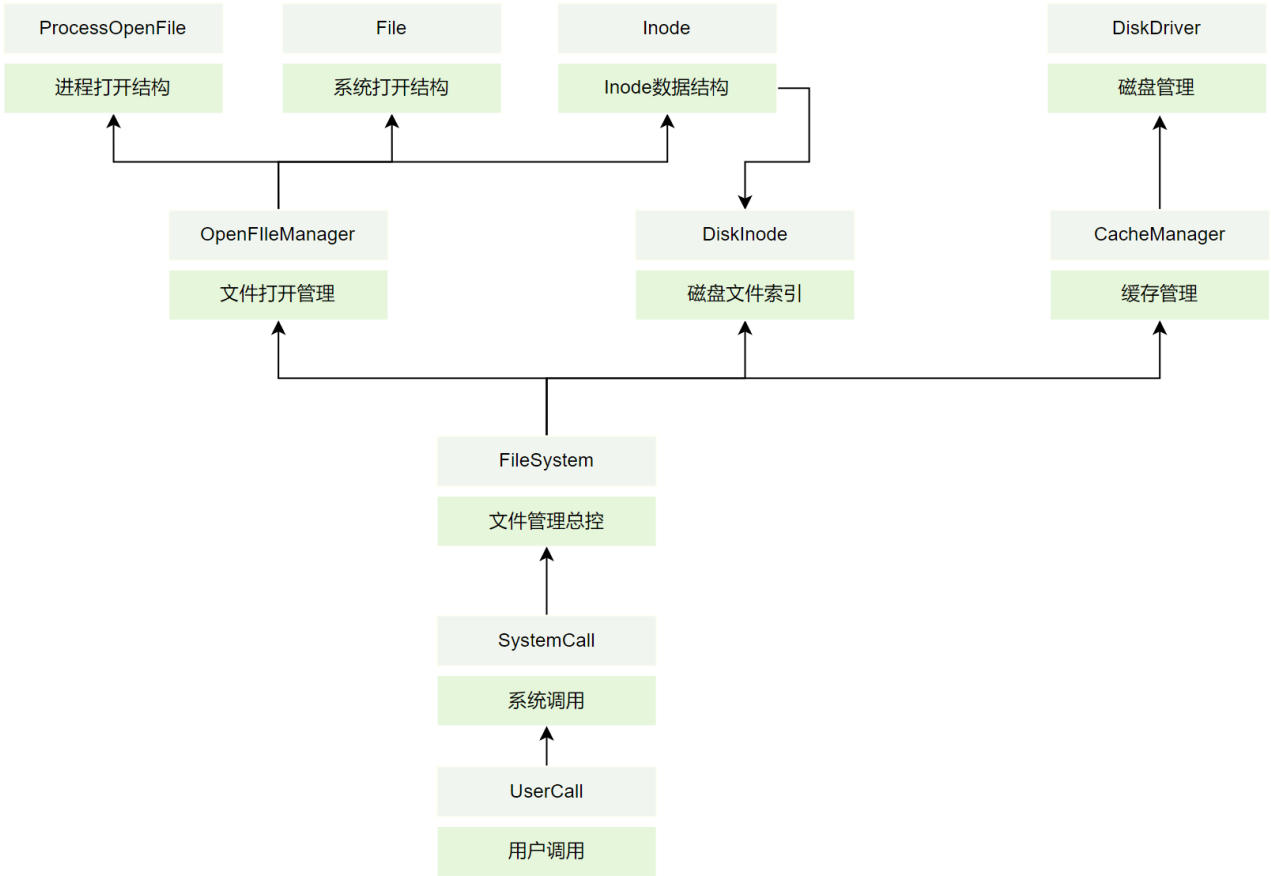

图 6: Enter Caption

- **UserCall 模块**：作为用户界面层，直接与用户互动，接收来自用户的命令并传递到 SystemCall 模块。

此模块是用户指令的第一接收点，负责初步处理并转发到后续的处理流程。

- **SystemCall 模块**：充当用户命令与文件系统操作之间的中介。接收 UserCall 模块传递的指令，并调用 FileSystem 模块以执行具体的文件系统操作。同时，此模块也负责处理与文件系统交互的系统级调用。
- **FileSystem 模块**：是文件系统的核心逻辑处理模块，直接管理所有文件和目录的操作。它通过 OpenFileManager 和 DiskInode 等子模块，执行如文件打开、创建、删除、读写等基本文件操作。
- **OpenFileManager 模块**：管理所有打开的文件，负责文件描述符的分配与回收。该模块为 ProcessOpenFile 和 File 模块提供底层支持，确保文件的正确打开和关闭。
- **DiskInode 模块**：处理磁盘上的索引节点（inode），它从 Inode 模块接收信息，确保磁盘上的数据与系统中的索引信息一致。
- **CacheManager 模块**：提供数据缓存功能，与 DiskDriver 模块协作，优化数据的读取和写入过程，减少直接对磁盘的操作，从而提高系统性能。
- **DiskDriver 模块**：作为与实际磁盘交互的最底层模块，负责所有的物理磁盘读写操作。它是实现文件数据持久化的关键组件。
- **Inode 模块**：在内存中管理 inode 信息，为文件系统中的文件元数据提供中心化管理。
- **ProcessOpenFile 和 File 模块**：分别管理与进程相关的文件打开表和具体的文件对象，维护文件操作过程中的状态信息和数据流。

这些模块的设计确保了文件系统的高效运行，通过明确的模块职责和相互之间的调用关系，实现了高效的数据处理和优化的资源管理。

2.4 算法说明

主函数等待用户输入。用户输入后，解析命令，然后查找 UserCall 提供的操作接口。接着，将参数交给 User 对象处理和判断合法性。如果参数基本符合命令的约定，UserCall 将调用 SystemCall 中的功能函数实现文件系统的具体功能。

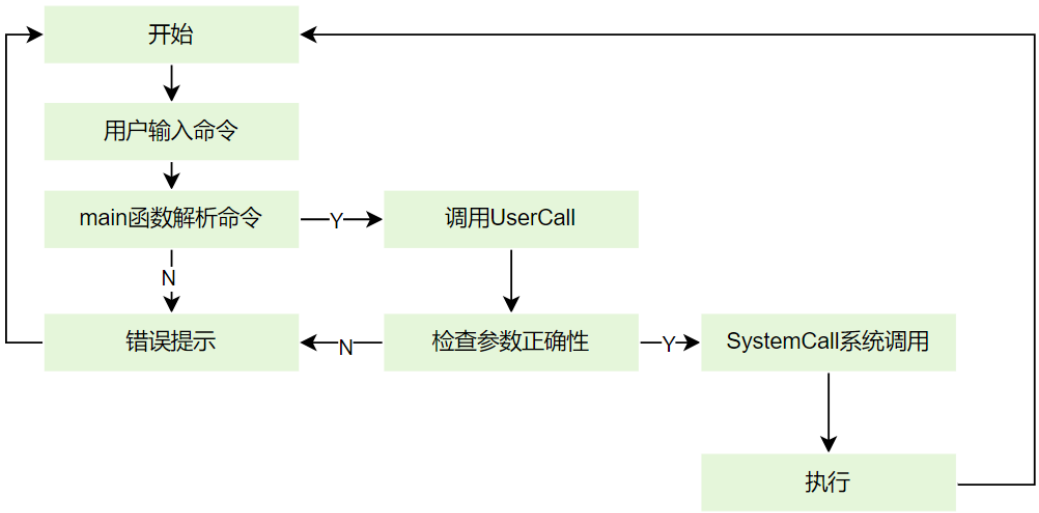

图 7: 算法说明

3 详细设计

3.1 重点函数与重点变量

表 2: 系统关键类和函数表

重点函数或变量	功能与意义
INodeTable 类	
`INode m_INodeTable[NINODE]`	内存中存储的 INode 数组。每个打开的文件会在此数组中占用一个内存 INode。
`INode* IGet(int inumber)`	根据外存 INode 编号获取对应的内存 INode。若该 INode 已在内存中，则直接返回内存中的副本；若不在内存中，则读取该 INode 至内存，并返回其地址。
`void IPut(INode* pNode)`	减少指定内存 INode 的引用计数。如果该 INode 已无目录项指向它且无进程引用该 INode，则释放该文件占用的磁盘块。
`void UpdateINodeTable()`	将所有被修改过的内存 INode 同步至外存 INode。
`int IsLoaded(int inumber)`	检查编号为 inumber 的外存 INode 是否在内存中有副本，若有则返回该内存 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，并返回其地址。
OpenFileTable 类	
`File sysFileTable[MAX_FILES]`	系统级打开文件表，用于所有进程共享。文件描述符表中包含指向此表中对应 File 结构的指针。
`File* FAlloc()`	在系统级打开文件表中分配一个空闲的 File 结构，并返回其指针。
`void CloseF(File* pFile)`	减少指定打开文件控制块 File 结构的引用计数 count。若引用计数 count 降至 0，则释放该 File 结构。
ProcessOpenTable 类	
`File* processOpenFileTable[MAX_FILES]`	指向系统级打开文件表中 File 对象的指针数组。
`int AllocFreeSlot()`	当进程请求打开文件时，用于在打开文件描述符表中分配一个空闲的表项。
`void SetF(int fd, File* pFile)`	建立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 对象之间的关联。
CacheManager 类	
`CacheBlock* GetBlk(int blkno)`	申请一块缓存，用于读写设备上的块 blkno
`CacheBlock* Bread(int blkno)`	读磁盘块
`void Bwrite(CacheBlock* bp)`	写磁盘块

续下页

表 2 – 续前页

重点函数或变量	功能与意义
void Bdwrite(CacheBlock* bp)	磁盘块延迟写
void Bflush()	将队列中延迟写的缓存刷回磁盘
FileSystem 类	
void LoadSuperBlock()	读入 SuperBlock，用于系统初始化时
INode* IAlloc()	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
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	
void Open()	处理 Open() 系统调用的函数。
void Creat()	处理 Creat() 系统调用的函数。
void Open1(INode* pINode, int trf)	Open() 和 Creat() 系统调用的共享部分处理函数。
void Close()	处理 Close() 系统调用的函数。
void Seek()	处理 Seek() 系统调用的函数。
void Read()	处理 Read() 系统调用的函数。
void Write()	处理 Write() 系统调用的函数。
void Rdwr(enum File::FileFlags mode)	处理读写系统调用的函数。
INode* NameI(enum DirectorySearchMode mode)	执行目录搜索，将路径转化为相应的 INode 并返回其上锁后的副本。
INode* MakNode(int mode)	用于 Creat() 系统调用，为创建新文件分配内核资源的函数。
void UnLink()	处理取消文件的系统调用的函数。
void WriteDir(INode* pINode)	向父目录的目录文件写入一个目录项的函数。
void ChDir()	处理改变当前工作目录的系统调用的函数。
void Ls()	列出当前 INode 节点的文件项的函数。
UserCall 类	
string dirp	指向系统调用参数的指针，通常用于路径名。
INode* paDirINodePointer	指向父目录的 Inode 指针。
INode* nowDirINodePointer	指向当前目录的 Inode 指针。
DirectoryEntry dent	当前目录的目录项。
char dbuf[DirectoryEntry::DIRSIZ]	当前路径分量的字符数组。
string curDirPath	当前工作目录的完整路径。
int arg[5]	存放当前系统调用参数的整型数组。
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等参数的结构体。
string ls	当调用 Ls() 函数时返回的值。

3.2 重点功能实现

3.2.1 空白盘块管理-成组链接法

实现了文件系统中的磁盘块分配算法。首先，它从超级块（superBlock）的空闲块列表中获取一个未分配的磁盘块（blkno）。如果没有可用的空闲块，则将错误代码设置为 U_ENOSPC，并返回空指针。

如果成功获取到空闲块，接下来会将该磁盘块读入缓存（CacheBlock* pCache）。然后，代码检查是否需要重新分配空闲块列表的组长。这是因为磁盘块的空闲块列表可能被组织成一个链表，其中每个块的第一个整数存储着下一个块的块号，而超级块的 s_free 数组存储着前几个块的块号。

如果当前的空闲块列表已用尽，需要从当前磁盘块中读取更多的空闲块号并重新分配到超级块的 s_free 数组中。为此，首先需要将当前磁盘块中的数据读取出来。这一步通过调用 cacheManager->Bread(blkno) 实现，其中 blkno 是当前磁盘块的块号。然后，将数据块内容中的第一个整数读取为新的空闲块数量（superBlock->s_nfree），接着将剩余的整数存储到超级块的 s_free 数组中。最后，释放掉之前分配的缓存块。

无论是否重新分配了空闲块列表的组长，都会对当前分配的磁盘块进行清除，即清除缓存中的数据。最后，标记超级块已修改（superBlock->s_fmod = 1），以便在必要时更新外存中的超级块信息。

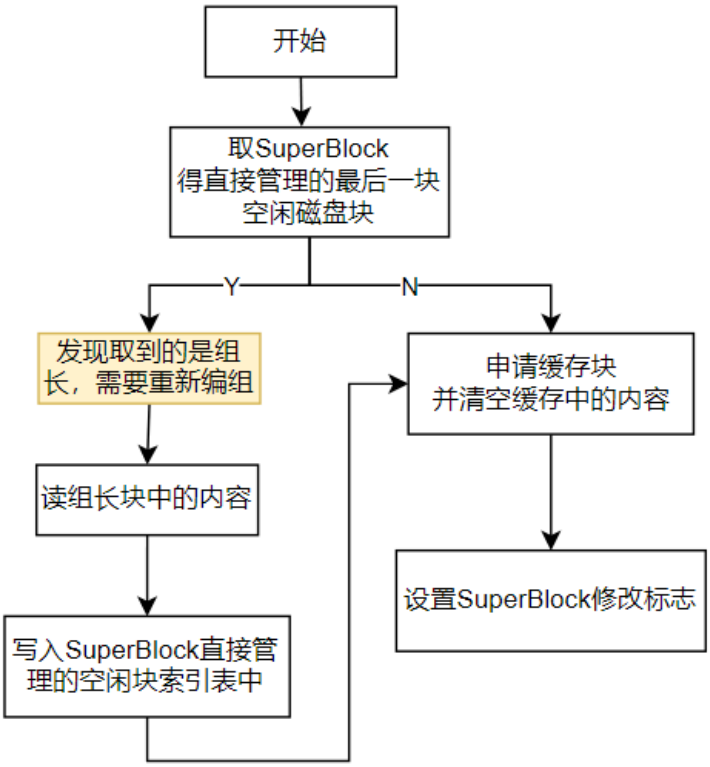

图 8: 成组链接法-分配空白盘块

```
//在存储设备上分配空闲磁盘块
//成组链接法 到内存superblocj中的s_nfree和s_free[]中查询一个新的盘块号

CacheBlock* FileSystem::Alloc()
{
  int blkno;
  CacheBlock* pCache;

  //取得直接管理的最后一块 空闲磁盘块
  blkno = superBlock->s_free[--superBlock->s_nfree];

  //若获取磁盘块编号为零，则表示已分配尽所有的空闲磁盘块
  /*****************************************************************************/
  if (blkno <= 0) {
    superBlock->s_nfree = 0;
    myUserCall.userErrorCode = UserCall::U_ENOSPC;
```

```
17      return NULL;
18    }
19
20    pCache = this->cacheManager->GetBlk(blkno);
21
22
23    //如果release的是组长，我们要做一些操作:
24    //
25    //
26    // s_free[0]  [1]   [2] 里面存的是blkno号
27    if(superBlock->s_nfree <= 0){   //直接管理的盘块数量没了  因为前面--superBlock->s_nfree了
28      //组长被分出去了 //我需要知道组长blkno对应的数据块里面写了哪些blkno号
29      //我想要将组长blkno对应数据块里面的东西挪到 直接管理的s_free中
30
31      //step1: 读blkno（刚刚读过的）中的所有数据 应该是101个int  第一个是100，后面100个是前一组的blkno
32      pCache = this->cacheManager->Bread(blkno);  //读到了整个缓存控制块
33
34      int* p = (int*)pCache->b_addr;    //p就是数据块内容起始地址
35
36      //step2: 将组长blkno对应数据块里面的东西挪到 直接管理的s_free中
37      superBlock->s_nfree = *p++;
38      memcpy(superBlock->s_free, p, sizeof(superBlock->s_free));
39      this->cacheManager->Brelse(pCache);
40    }
41
42    if (pCache)
43      this->cacheManager->Bclear(pCache);
44
45    superBlock->s_fmod = 1;    //表示内存的SB被修改过，要更新外存中的SB
46    return pCache;
47  }
```

3.2.2 空白 DiskInode 管理-栈式管理

实现了文件系统中的磁盘 Inode 分配算法。首先，它检查 superBlock 中直接管理的空闲 Inode 索引表是否为空。如果是空的，则需要到磁盘上搜索空闲 Inode。在搜索过程中，它会逐个检查磁盘上的 Inode 块，如果找到一个未被使用的 Inode，则将其索引加入到超级块的空闲 Inode 索引表中。

若成功分配到一个 Inode 索引，接下来就从内存中获取对应的 Inode 对象（pINode）。如果获取失败，则表示内存中没有足够的空闲 Inode，此时会返回空指针并输出相应的错误信息。若成功获取到 Inode 对象，则会清除其内容，并标记超级块已修改，以便在必要时更新外存中的超级块信息。最后，返回分配到的 Inode 对象。

```
1  //在存储设备dev上分配一个空闲DiskINode，一般用于创建新的文件
2  //栈式存储
3  INode* FileSystem::IAlloc()
4  {
```

```
  CacheBlock* pCache;
  INode* pINode;
  int ino;

  //SuperBlock直接管理的空闲Inode索引表已空，必须到磁盘上搜索空闲Inode 搜索100个inode

  if (superBlock->s_ninode <= 0) {
    ino = -1;
    for (int i = 0; i < superBlock->s_isize; ++i) { //搜索所有的inode
      pCache = this->cacheManager->Bread(FileSystem::INODE_START_SECTOR + i);
      int* p = (int*)pCache->b_addr;
      for (int j = 0; j < FileSystem::INODE_NUMBER_PER_SECTOR; ++j) {
        ++ino;
        int mode = *(p + j * FileSystem::INODE_SIZE / sizeof(int));
        if (mode)
          continue;
        //如果外存inode的i_mode == 0，此时并不能确定该inode是空闲的，
        //因为有可能是内存inode没有写到磁盘上，所以要继续搜索内存inode中是否有相应的项
        if (myINodeTable.IsLoaded(ino) == -1) {
          superBlock->s_inode[superBlock->s_ninode++] = ino;
          if (superBlock->s_ninode >= SuperBlock::MAX_NUMBER_INODE)
            break;
        }
      }

      this->cacheManager->Brelse(pCache);
      if (superBlock->s_ninode >= SuperBlock::MAX_NUMBER_INODE)
        break;
    }
    if (superBlock->s_ninode <= 0) {
      myUserCall.userErrorCode = UserCall::U_ENOSPC;
      return NULL;
    }
  }

  //分配一个inode，退栈
  ino = superBlock->s_inode[--superBlock->s_ninode];

  pINode = myINodeTable.IGet(ino);

  if (NULL == pINode) {
    cout << "无空闲内存存储INode" << endl;
    return NULL;
  }

  pINode->Clean();
  superBlock->s_fmod = 1;
  return pINode;
}
```

3.2.3 目录搜索 NameI

目录搜索的算法。首先从根目录开始，逐级搜索目录路径中的每个分量，直到找到匹配的目录项或遇到错误。搜索过程中，会根据路径分量逐步读取目录文件的各个盘块，并在每个盘块中查找目标字符串。根据不同的操作模式（删除、打开、创建），算法会返回相应的结果，如删除操作会返回父目录的 Inode 指针，而创建操作会在成功时返回 NULL，表示可以在目标目录中创建新文件。

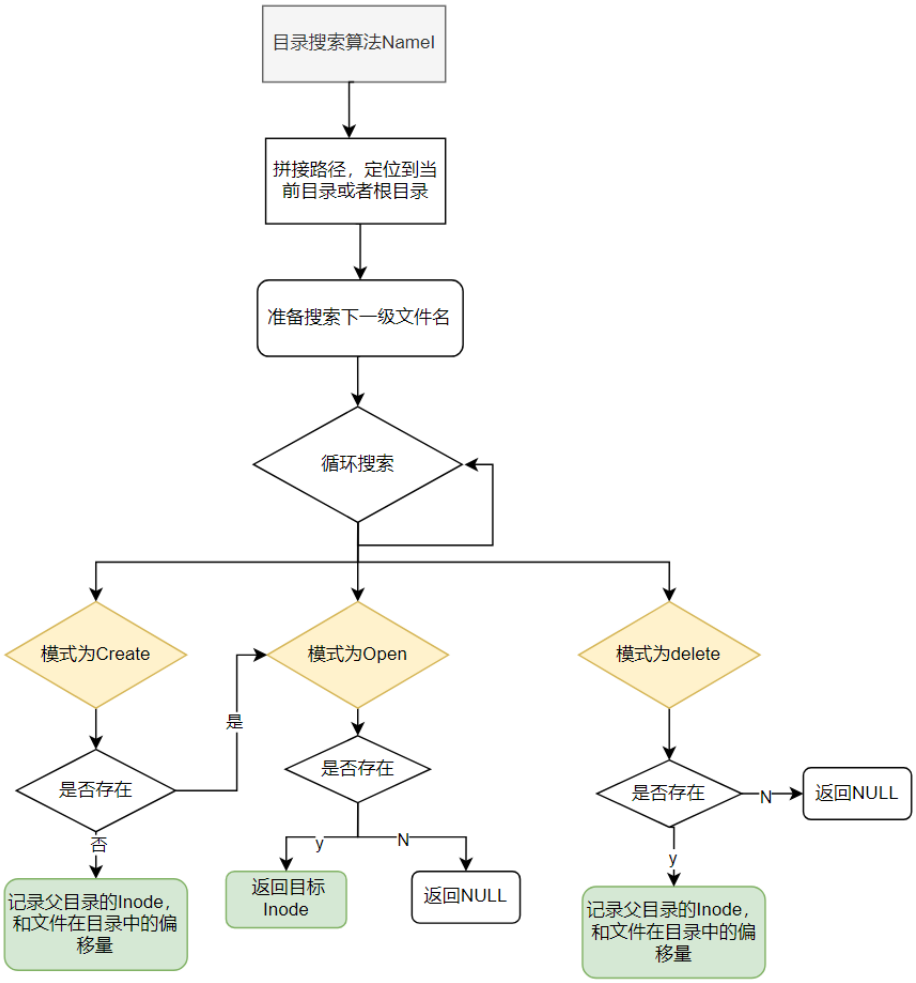

图 9: 目录搜索算法

3.2.4 目录搜索结构的增、删、改

新建目录的主要流程是，首先进行目录检索，若存在该目录项，则返回该目录的 Inode 号；若不存在该目录，如果是只有最后一个分量不成功，新申请一个磁盘 Indoe，并增加目录项，形成内存打开结构，如果有其他位置搜索不成功，则设置出错码。

删除目录与新建目录流程基本类似，首先进行目录检索，若存在该目录项，则进行删除并修改目录、Inode 与 SuperBlock；若不存在该目录，则设置出错码。更改同理。

3.2.5 逻辑-物理块映射 Bmap

实现了将文件的逻辑块号（Logical Block Number，lbn）转换成对应的物理盘块号（Physical Block Number，phyBlkno）。它采用了 Unix V6++ 的文件索引结构，支持小型、大型和巨型文件。

首先，如果 lbn 小于 6，则直接从文件的直接索引表 i_addr[0-5] 中获取物理盘块号 phyBlkno。如果对应的物理盘块号尚未分配，则分配一个新的物理块，并将其索引添加到直接索引表中。

如果 lbn 大于等于 6，则表示文件是大型或巨型文件，需要进一步进行索引转换。对于大型文件，根据 lbn 计算一次间接索引表所在的索引，然后读取一次间接索引表块，并根据索引从中获取物理盘块号。对于巨型文件，需要首先读取二次间接索引表块，然后根据 lbn 计算出在二次间接索引表中的索引，再从中获取一次间接索引表块的物理盘块号，最终再根据索引从一次间接索引表中获取物理盘块号。

无论是一次间接索引还是二次间接索引，获取物理盘块号的过程都是相似的：首先检查对应的物理盘块是否已经分配，如果没有，则分配一个新的物理块，并将其索引添加到索引表中；如果已经分配，则直接返回对应的物理盘块号。最后，返回获取到的物理盘块号 phyBlkno。

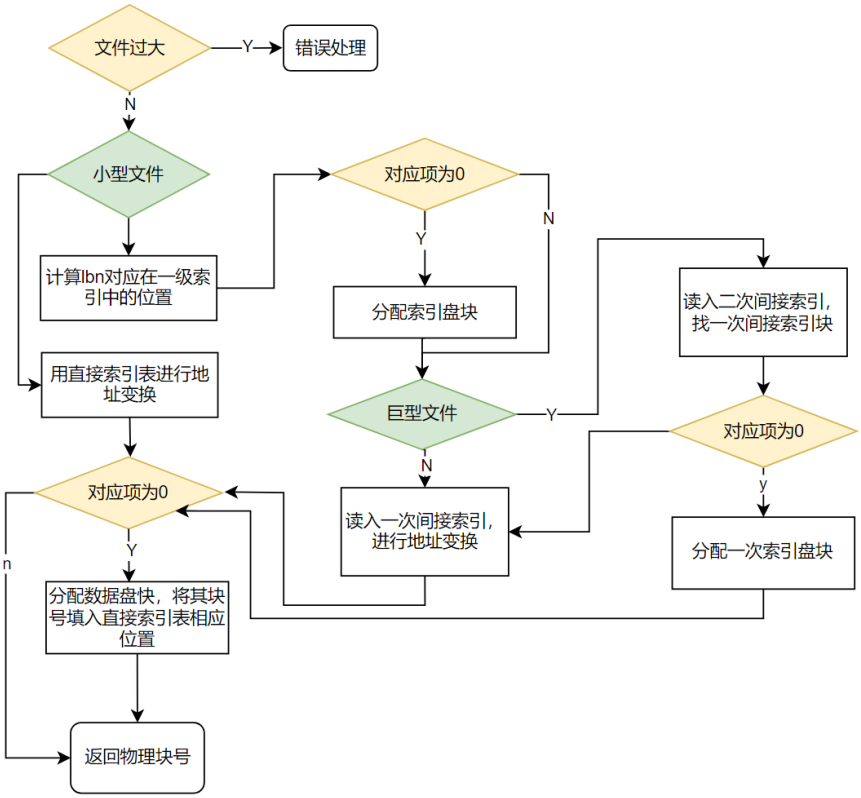

图 10: Bmap 映射函数

```
//由offset得到lbn
//将文件的逻辑块号转换成对应的物理盘块号 lbn是0-几百，它返回的就是最终索引块的物理扇区号，而不是二次/一次间接，他在中间已经处理好了
//如果是小型文件，从基本索引表i_addr[0-5]中获得物理盘块号即可
//一次间接索引：index0 = (lbn - 6) / 128 + 6;  就是d_addr的序号     (0-5, 67, 8)
//二次间接索引：  index0= (lbn-6-128*2) / (128*128) +8
//index2 = ( (lbn-(128*2+6)) /128) %128
```

```
//index3 = (lbn-(128*2+6)) %128
//S29 P20

//d_addr里面到底放了什么, int 直接的: 磁盘数据块
//一级: 512/4=128个int 也是磁盘数据块了
//返回的是 物理扇区号
int INode::Bmap(int lbn)
{
  //Unix V6++的文件索引结构: (小型、大型和巨型文件)
  //(1) i_addr[0] - i_addr[5]为直接索引表, 文件长度范围是0 - 6个盘块;
  //(2) i_addr[6] - i_addr[7]存放一次间接索引表所在磁盘块号, 每磁盘块
  //上存放128个文件数据盘块号, 此类文件长度范围是7 - (128 * 2 + 6)个盘块;
  //(3) i_addr[8] - i_addr[9]存放二次间接索引表所在磁盘块号, 每个二次间接
  //索引表记录128个一次间接索引表所在磁盘块号, 此类文件长度范围是
  //(128 * 2 + 6 ) < size <= (128 * 128 * 2 + 128 * 2 + 6)
  CacheManager& CacheManager = myCacheManager;
  FileSystem& fileSystem = myFileSystem;
  CacheBlock* pFirstCache, * pSecondCache;
  int phyBlkno, index;
  int* iTable;

  if (lbn >= INode::HUGE_FILE_BLOCK) {
    myUserCall.userErrorCode = UserCall::U_EFBIG;
    return 0;
  }


  //如果是小型文件, 从基本索引表i_addr[0-5]中获得物理盘块号即可
  if (lbn < 6) {
    phyBlkno = this->i_addr[lbn];
    //如果该逻辑块号还没有相应的物理盘块号与之对应, 则分配一个物理块
    if (phyBlkno == 0 && (pFirstCache = fileSystem.Alloc()) != NULL) {
      phyBlkno = pFirstCache->b_blkno;
      CacheManager.Bdwrite(pFirstCache);
      this->i_addr[lbn] = phyBlkno;
      this->i_flag |= INode::IUPD;
    }
    return phyBlkno;
  }

  //lbn >= 6 大型、巨型文件      index是addr中的索引, 0级索引, 小型文件就是直接读的这个, 而我们现在要去找二
    级索引
  if (lbn < INode::LARGE_FILE_BLOCK)
    index = (lbn - INode::SMALL_FILE_BLOCK) / INode::ADDRESS_PER_INDEX_BLOCK + 6;
  else //巨型文件: 长度介于263 - (128 * 128 * 2 + 128 * 2 + 6)个盘块之间
    index = (lbn - INode::LARGE_FILE_BLOCK) / (INode::ADDRESS_PER_INDEX_BLOCK * INode::
    ADDRESS_PER_INDEX_BLOCK) + 8;


  //读到了addr这个块pFirstCache
  phyBlkno = this->i_addr[index];
```

```
56  if (phyBlkno)
57    pFirstCache = CacheManager.Bread(phyBlkno);
58  else { //若该项为零，则表示不存在相应的间接索引表块
59    this->i_flag |= INode::IUPD;
60    if ((pFirstCache = fileSystem.Alloc()) == 0)
61      return 0;
62    this->i_addr[index] = pFirstCache->b_blkno;
63  }
64
65  //iTable很重要，存储的是读的块本质上是文件数据块buffer读的，这128个4位int数据，对应了物理块号，所以iTable
      [index]取到的就是想要的物理块号
66  iTable = (int*)pFirstCache->b_addr;
67  if (index >= 8) {
68    //对于巨型文件的情况，pFirstBuf中是二次间接索引表，
69    //还需根据逻辑块号，经由二次间接索引表找到一次间接索引表
70    index = ((lbn - INode::LARGE_FILE_BLOCK) / INode::ADDRESS_PER_INDEX_BLOCK) % INode::
      ADDRESS_PER_INDEX_BLOCK;
71    phyBlkno = iTable[index];
72
73    if (phyBlkno) {
74      CacheManager.Brelse(pFirstCache);
75      pSecondCache = CacheManager.Bread(phyBlkno);    //巨型文件的最后一步，接下来只需要读它，即可获得条目取
      得物理块号
76    }
77    else {
78      if ((pSecondCache = fileSystem.Alloc()) == NULL) {
79        CacheManager.Brelse(pFirstCache);
80        return 0;
81      }
82      iTable[index] = pSecondCache->b_blkno;
83      CacheManager.Bdwrite(pFirstCache);
84    }
85
86    pFirstCache = pSecondCache; //这一步要了干嘛
87    iTable = (int*)pSecondCache->b_addr;
88  }
89
90  if (lbn < INode::LARGE_FILE_BLOCK)//262
91    index = (lbn - INode::SMALL_FILE_BLOCK) % INode::ADDRESS_PER_INDEX_BLOCK;    //计算最后一步读要用的序号
92  else    //巨形
93    index = (lbn - INode::LARGE_FILE_BLOCK) % INode::ADDRESS_PER_INDEX_BLOCK;
94
95  if ((phyBlkno = iTable[index]) == 0 && (pSecondCache = fileSystem.Alloc()) != NULL) {    //通过iTable读，
      二级索引有二级索引的iTable，大文件有自己的iTable
96    phyBlkno = pSecondCache->b_blkno;
97    iTable[index] = phyBlkno;    //相当于 *iTable+index*4 是对的
98    CacheManager.Bdwrite(pSecondCache);
99    CacheManager.Bdwrite(pFirstCache);
100 }
101 else
102   CacheManager.Brelse(pFirstCache);
```

```
103     return phyBlkno;
104
105 }
```

3.2.6 缓存块管理-缓存分配/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。

-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。

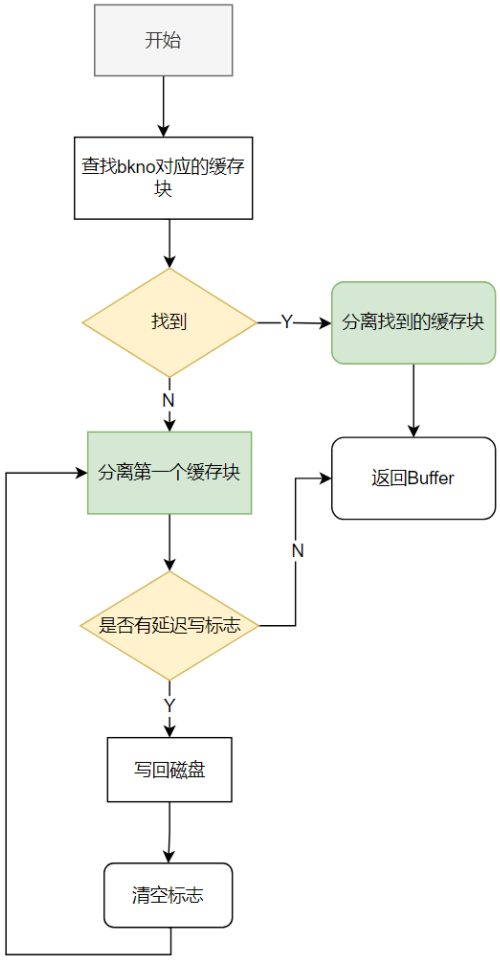

图 11: 缓存管理

3.2.7 读文件

当用户希望读取文件系统中的一个文件内容时，系统首先获取该文件的文件打开结构（File 句柄），并进一步打开该文件控制块。接着，系统会检查文件控制块中的权限，判断是否有权限进行读取操作。如果权限没有问题，系统会根据要读取的文件大小，逐步计算要读取的逻辑块号和块内偏移地址，并通过 Bmap 函数获得对应的物理磁盘号。随后，系统进行缓存的分配，将物理块内容读入缓存区，再将缓存区内容拷贝到目标区。在全部读取操作完成后，释放缓存。

本文件系统读文件的实现流程如下图所示：

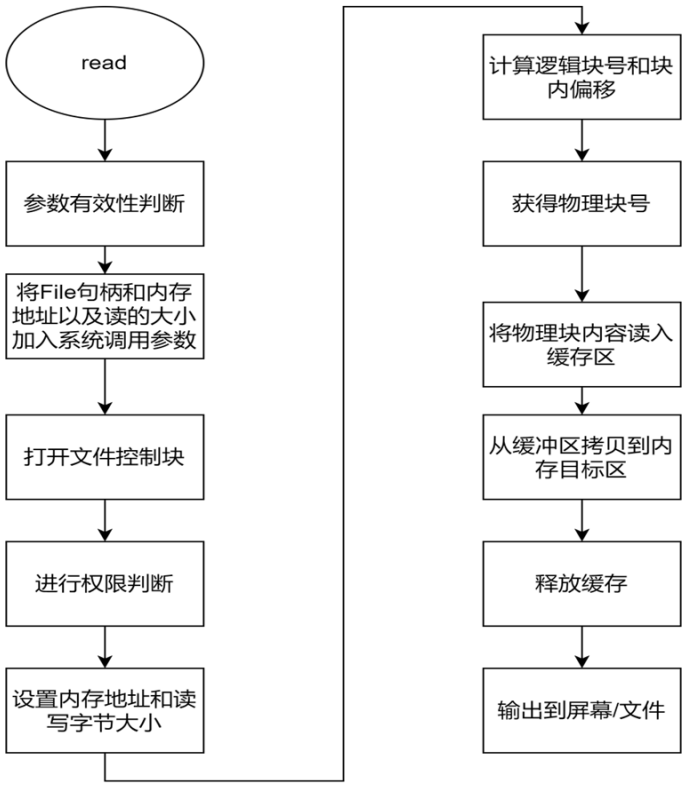

图 12: 读文件

3.2.8 写文件

写文件的流程与读文件基本类似。当用户希望将其他文件中的内容写入文件系统中的一个文件时，系统首先获取该文件的文件打开结构（File 句柄），并进一步打开该文件控制块。接着，系统会检查文件控制块中的权限，判断是否有权限进行写入操作。如果权限没有问题，系统会根据要写入的文件大小，逐步计算要写入的逻辑块号和块内偏移地址，并通过 Bmap 函数获得对应的物理磁盘号。对于写入操作，系统会首先判断写入数据是否刚好填满一个数据块。如果是，则直接进行缓存的分配；如果不是，则分配缓存器并首先将源文件的内容读入缓存区。然后，将要写入的内容拷贝到缓存区，并更新读写位置，最后将缓存区内容写回文件。

本文件系统写文件的实现流程如下图所示：

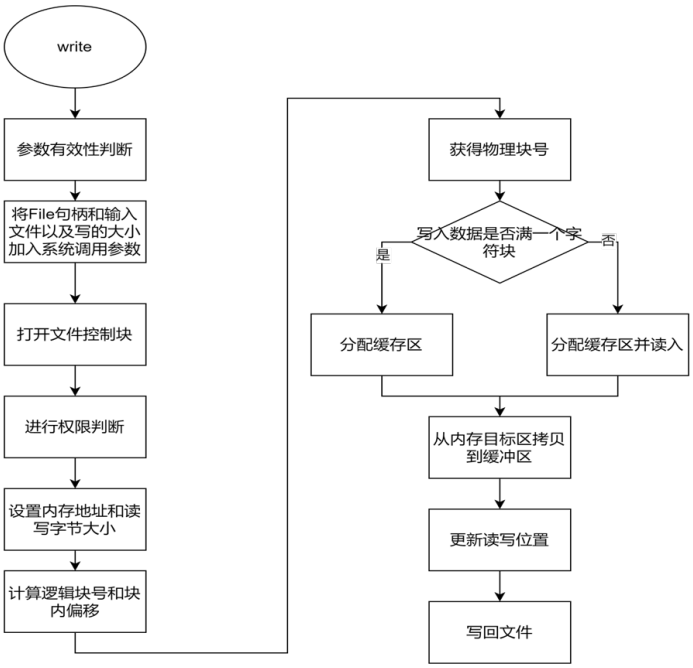

图 13: 写文件

3.3 函数调用关系

主要函数的基本调用关系如下:

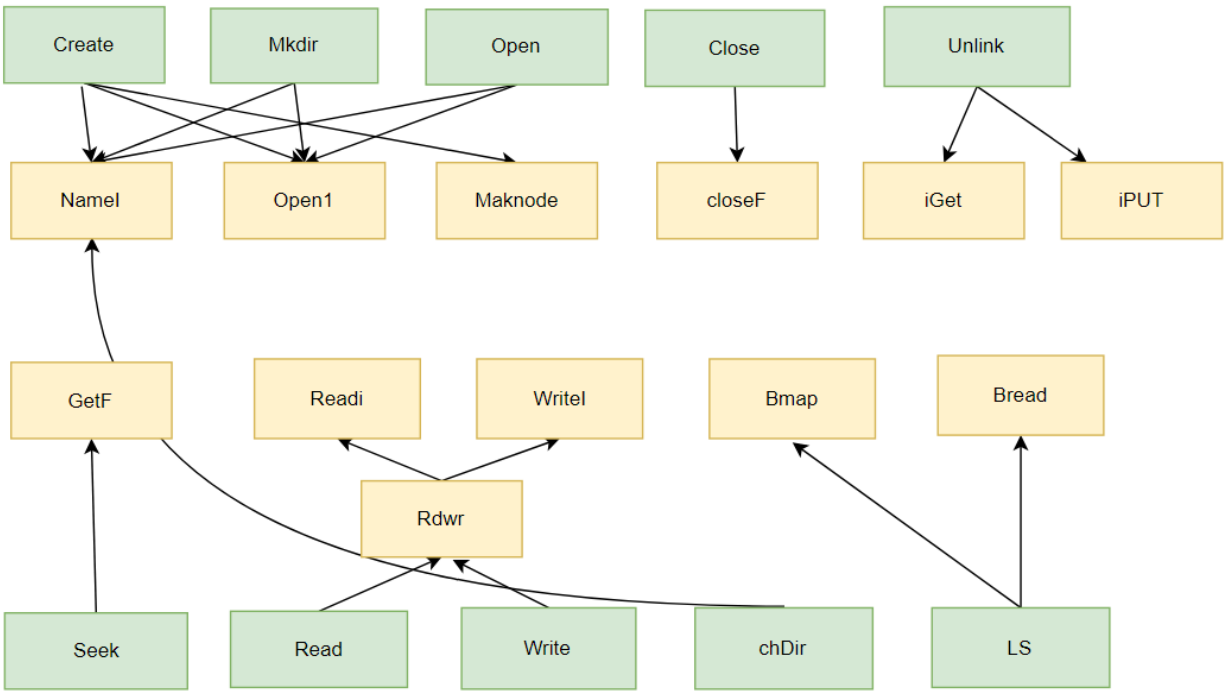

图 14: 函数调用关系

4 运行结果分析

在这一章节，我会逐个测试命令，展示包括正确运行和错误提示在内的多种情况。

4.1 格式化文件卷

在这一指令中，初始化了磁盘，定义了磁盘的基本数据结构，将磁盘中的数据结构都写入。在文件夹下也出现了 myDisk.img 的光盘盘文件。

```
*********************************************************************************************
*                                                                                           *
*                                  Unix-style File System                                   *
*                                                                                           *
* [Instructions]:                                                                           *
* [Command]:fformat                    [Description]:Format the file system                 *
* [Command]:ls                         [Description]:List contents of the current directory *
* [Command]:mkdir <dirname>            [Description]:Create a new directory                 *
* [Command]:cd <dirname>               [Description]:Change directory                       *
* [Command]:fcreate <filename>         [Description]:Create a new file named filename       *
* [Command]:fopen <filename>           [Description]:Open a file named filename             *
* [Command]:fwrite <fd> <infile> <size>  [Description]:Write size bytes from infile to fd   *
* [Command]:fread <fd> <outfile> <size>  [Description]:Read size bytes from fd to outfile   *
* [Command]:fread <fd> std <size>      [Description]:Read size bytes from fd to screen      *
* [Command]:fseek <fd> <step> begin    [Description]:Move fd file pointer by step in begin mode    *
* [Command]:fseek <fd> <step> cur      [Description]:Move fd file pointer by step in current mode  *
* [Command]:fseek <fd> <step> end      [Description]:Move fd file pointer by step in end mode      *
* [Command]:fclose <fd>                [Description]:Close the file with handle fd          *
* [Command]:fdelete <filename>         [Description]:Delete the file or folder named filename      *
* [Command]:exit                       [Description]:Exit system and save cache to disk     *
*********************************************************************************************
[20514540WYF@root/ ]$ fformat
File system formatted. Please restart!

D:\TJ_Data\操作系统课设\Project\OS_FileSystem\Debug\OS_YIFEI.exe (进程 58044)已退出，代码为 0。
```

图 15: format 结果

4.2 新建文件夹

这一指令和新建文件类似，都是搜索到目录之后，在其中创建新的 Inode，并在目录中写入。给出指令 mkdir home，mkdir etc，mkdir bin，mkdir dev。

4.3 列出当前目录下的所有内容

在上衣指令新建文件夹的基础上，我们打印目录。

结果如下：

```
[20514540WYF@root/ ]$ mkdir bin
[20514540WYF@root/ ]$ mkdir etc
[20514540WYF@root/ ]$ mkdir home
[20514540WYF@root/ ]$ mkdir dev
[20514540WYF@root/ ]$ ls
bin etc home dev
[20514540WYF@root/ ]$
```

图 16: 目录结果

4.4 切换目录

我们支持 cd.. cd / cd 这些切换目录的表达方式

```
[20514540WYF@root/ ]$ mkdir bin
[20514540WYF@root/ ]$ mkdir etc
[20514540WYF@root/ ]$ mkdir home
[20514540WYF@root/ ]$ mkdir dev
[20514540WYF@root/ ]$ ls
bin etc home dev
[20514540WYF@root/ ]$ cd home
[20514540WYF@root/home/ ]$ cd ..
[20514540WYF@root/ ]$ ls
bin etc home dev
[20514540WYF@root/ ]$ cd home
[20514540WYF@root/home/ ]$
```

图 17: 切换目录

4.5 创建文件

在目录为/home/texts 处创建文件 Readme.txt, 再用 ls 查看/home/texts 之中的所有文件，打印得到了正确结果。

```
[20514540WYF@root/home/ ]$ mkdir texts
[20514540WYF@root/home/ ]$ mkdir reports
[20514540WYF@root/home/ ]$ mkdir photos
[20514540WYF@root/home/ ]$ cd texts
[20514540WYF@root/home/texts/ ]$ fcreate Readme.txt
[20514540WYF@root/home/texts/ ]$ ls
Readme.txt
```

图 18: 创建文件

4.6 打开文件，读出写入文件

输出命令想要打开 Readne.txt, 但事实上并没有 Readne.txt 这个文件，那么程序就会给出错误处理。如下图所示。

```
[20514540WYF@root/home/texts/ ]$ fopen Readne.txt
File or directory not found
```

图 19: 找不到文件的错误输出

输入正确的命令，打开 Readme.txt，程序会返回正确的句柄 fd。我们将外部的 Readme.txt 存入系统之内，再将系统内的 Readme.txt 用 fopen 命令读出，输入到 windows 文件的 Readmeout.txt 中。其中也会用到 fseek 将文件指针移动到 begin 处，因为将外部文件读入 Readme.txt 之后，文件的 offset 是在文件末尾，而下一步用 fread 时，需要将 offset 移动到 begin。运行结果如下图所示：

```
[20514540WYF@root/home/texts/ ]$ fread 8 Readmeout.txt 2141
Successfully read 0 bytes
[20514540WYF@root/home/texts/ ]$ fopen Readme.txt
File opened successfully, the returned file handle fd is 9
[20514540WYF@root/home/texts/ ]$ fwrite 9 Readme.txt 2141
Successfully wrote 2141 bytes
[20514540WYF@root/home/texts/ ]$ fseek 9 0 begin
File pointer successfully moved to 0
[20514540WYF@root/home/texts/ ]$ fread 9 Readmeout.txt 2141
Successfully read 2141 bytes
[20514540WYF@root/home/texts/ ]$
```

图 20: fopen，fread，fwrite 结果图

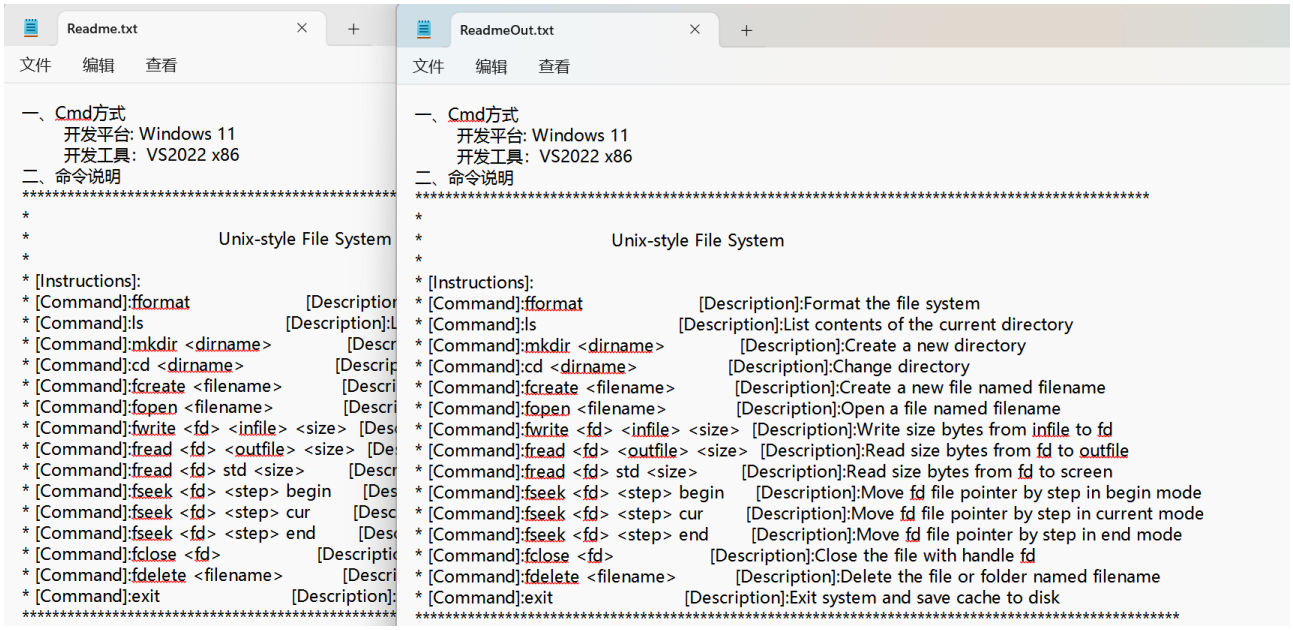

图 21: 原始文件与读出文件比对正确

4.7 图片读写

对于图片和 docx 文件都和上述的 txt 文件的读写相同。我在测试中遗忘了移动 fseek，从结果上看到，之后在读 fread 并没有读到任何字节，这和我在上一个小节的解释相同，在 fseek 之后，我们就成功地读出了 png 中的所有字节。

```
[20514540WYF@root/home/photos/ ]$ fcreate lenna.png
[20514540WYF@root/home/photos/ ]$ fopen lenna.png
File opened successfully, the returned file handle fd is 14
[20514540WYF@root/home/photos/ ]$ fwrite 14 lenna.png 473831
Successfully wrote 473831 bytes
[20514540WYF@root/home/photos/ ]$ fread 14 lennaOUT.png 473831
Successfully read 0 bytes
[20514540WYF@root/home/photos/ ]$ fseek 14 0 begin
File pointer successfully moved to 0
[20514540WYF@root/home/photos/ ]$ fread 14 lennaOUT.png 473831
Successfully read 473831 bytes
```

图 22: 图片读写

比对两个图片，可以看到图片文件的读写非常正确。

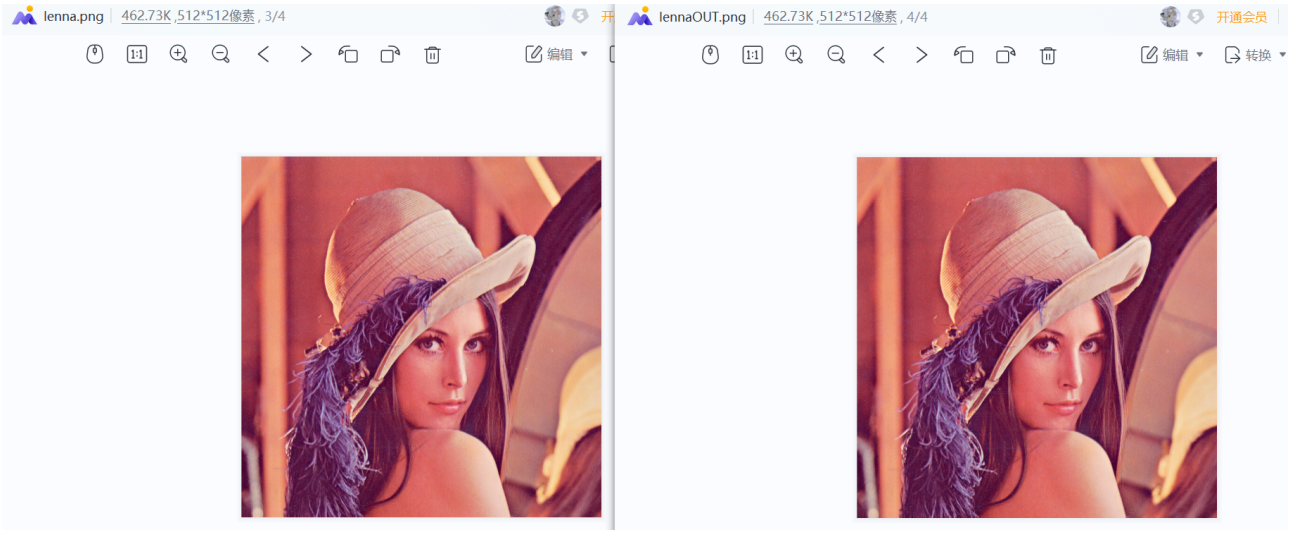

图 23: 图片的读写

4.8 要求 ppt 中的测试结果展示

ppt 要求：新建文件/test/Jerry，打开该文件，任意写入 800 个字节；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。将 abc 写回文件。可以看到 abc 字符串实际只读到了 300 个字节

```
[20514540WYF@root/test/ ]$ fcreate Jerry
[20514540WYF@root/test/ ]$ fopen Jerry
File opened successfully, the returned file handle fd is 17
[20514540WYF@root/test/ ]$ fwrite 17 input.txt 800
Successfully wrote 800 bytes
[20514540WYF@root/test/ ]$ fseek 17 500 begin
File pointer successfully moved to 500
[20514540WYF@root/test/ ]$ fread 17 abc.txt 500
Successfully read 300 bytes
[20514540WYF@root/test/ ]$ fclose 17
[20514540WYF@root/test/ ]$
```

图 24: 测试结果

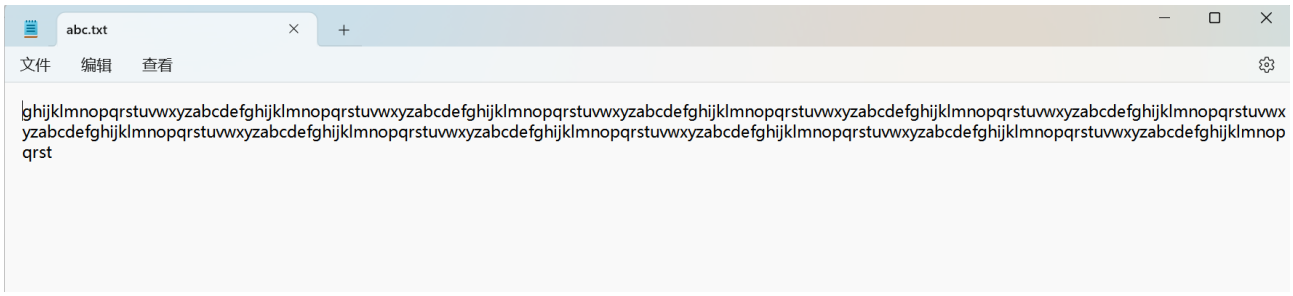

图 25: abc.txt 文件结果

5 用户使用说明

1. 运行说明

Windows Cmd 方式

开发平台: Windows 11

开发工具：VS22 x86

在 Windows 下双击生成的 OS_YIFEI.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。文件夹中 input.txt、Readme.txt、Report.docx、lenna.jpg 是输入源文件，请不要删除。

2. 注意事项

格式化：

格式化命令为 fformat，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：

建议通过 exit 命令正确退出程序，这样退出系统会在退出前将延迟写的缓存块刷回磁盘，不易出错。

6 个人感悟

在这次的文件系统设计与实现中，我创建了一个类 UNIX V6++ 文件系统，包括了基本功能如创建、读取、写入、删除文件，以及文件的权限控制和打开关闭操作。这个过程让我更深入地了解了文件系统的构建和运行机制。

首先，我深入学习了 UNIX V6++ 文件系统的核心结构，包括 SuperBlock、Inode 节点和数据块的概念。然后，我利用这些概念，结合对 UNIX V6++ 源码的参考，设计了文件系统的核心数据结构和算法。这涉及到 DiskInode、SuperBlock、目录管理、文件打开结构等方面，以及文件系统的实现和高速缓存管理。

在实验过程中，我遇到了一些挑战。其中包括对文件系统各个模块的相互关系理解不深，导致了数据结构的设计存在一些瑕疵，以及算法的优化不足。特别是在文件权限控制和打开操作上，一开始我遇到了一些逻辑上的困惑，需要多次思考和调试才能理清楚。

然而，通过不断地学习和实践，我逐渐解决了这些困难。我重新审视了文件系统的整体架构，优化了数据结构和算法设计，同时加强了对文件权限和打开操作的理解。最终，我成功地实现了文件系统的基本功能，并且对 UNIX V6++ 文件系统的原理和设计有了更深入的认识。

这次实验对我来说是一次非常宝贵的学习经历。通过挑战和克服困难，我不仅掌握了文件系统的设计与实现方法，还提升了自己的问题解决能力和逻辑思维水平。这将对我未来的计算机科学学习和职业发展产生积极的影响。

7 参考资料

1. UNIXV6++ 源码

2. https://github.com/zzhuncle/TJCS-Undergraduate-Courses