同济大学

TONGJI UNIVERSITY

操作系统课程设计

课题名称	二级文件系统设计
副 标 题	操作系统课程设计
学　　院	电子与信息工程学院
专　　业	计算机科学与技术
学生姓名	张哲源
学　　号	1850772
指导老师	方钰
日　　期	2024 年 05 月 18 日

目录

装 订 线

装 订 线

1 题目分析及其要求

1.1 实验目的

UNIX 文件系统提供了层次结构的目录和文件，负责系统内文件信息的管理，是 UNNIX 系统中极为重要的一个部分。本次实验的目的是通过模拟 UNIX V6++ 编写一个 UNIX 文 件系统，实现基本文件操作，从而掌握 UNIX 文件系统结构

1.2 题目分析

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++的一张磁盘。磁盘中存储的信息以块为单位。每块 512 字节

图 1.1 文件卷结构

1.3 设计要求

1. 实现对该逻辑磁盘的基本读写操作
   - 解决文件物理地址和逻辑的转换
   - 完成文件读出和写入操作
   - 设计缓存队列，完成读写操作

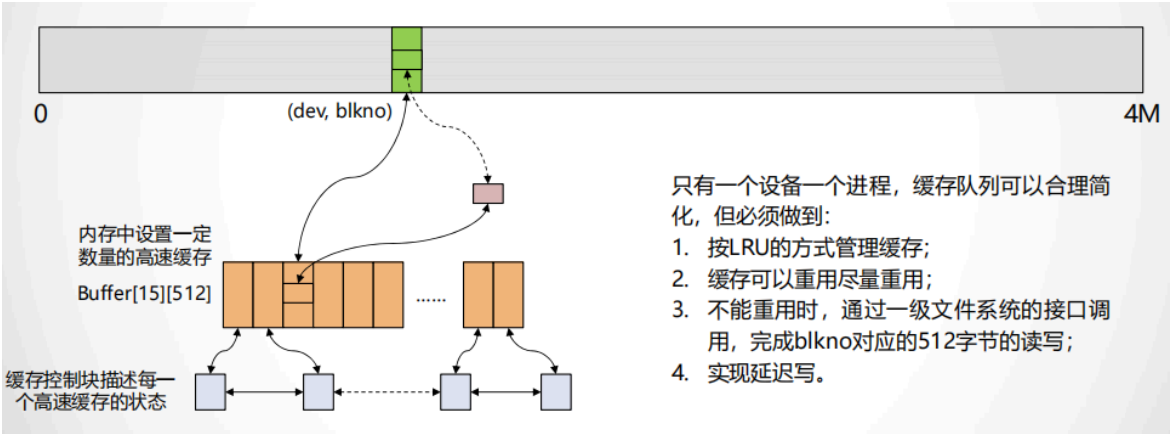


图 1.2 缓存队列

2. 在该逻辑磁盘上定义二级文件系统结构
   - SuperBlock 及 Inode 区所在位置及大小
   - Inode 节点

- 数据结构定义：注意大小，一个盘块包含整数个 Inode 节点
- Inode 区的组织（给定一个 Inode 节点号，怎样快速定位）
- 索引结构：多级索引结构的构成，**索引结构的生成与检索过程**

- SuperBlock
  - 数据结构定义
  - 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

3. 文件系统目录结构
   - 目录文件的结构
   - 目录检索算法的设计与实现
   - 目录结构增、删、改的设计与实现
4. 文件打开结构
   - 文件打开结构的设计：内存 Inode 节点，File 结构？进程打开文件表？
   - 内存 Inode 节点的分配与回收
   - 文件打开过程
   - 文件关闭过程
5. 文件操作接口
   - fformat： 格式化文件卷
   - ls： 列目录
   - mkdir： 创建目录
   - fcreat： 新建文件
   - fopen： 打开文件
   - fclose： 关闭文件
   - fread： 读文件
   - fwrite： 写文件
   - flseek： 定位文件读写指针
   - fdelete： 删除文件

1.4 测试要求

1. 通过命令行方式完成以下操作：
   - 格式化文件卷；
   - 用 mkdir 命令创建子目录，建立如图所示目录结构；
   - 把你的课设报告，关于课程设计报告的 ReadMe.txt 和一张图片存进这个文件系统，分别放在/home/texts ，/home/reports 和/home/photos 文件夹；

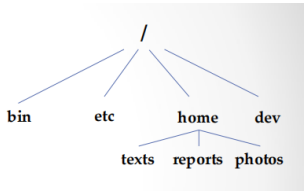


图 1.3　所需目录结构

2. 通过命令行方式测试：

- 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。

本次课设需要完成上述所有数据结构与算法的设计，并做到：

- 控制台程序， 图形界面或命令行
- 提供文件系统的基本功能
- 根据用户不同输入，返回结果

2 需求分析

2.1 用户接口分析

根据对题目要求分析和用户使用分析，文件系统采用控制台命令作为输入，按照系统给 定的命令即可完成相应的功能。本文件模拟系统使用尽可能详细的控制台引导用户输入指令。用户可以在控制台输入的指令如下：

表 2.1 help 指令

指令	**help**
描述	帮助用户使用相应命令
使用方式	help [命令]
参数	• fformat: 格式化 • exit: 正确退出 • mkdir: 新建目录 • cd: 改变目录 • ls: 列出目录及文件 • create: 新建文件 • delete: 删除文件 • open: 打开文件 • close: 关闭文件 • seek: 移动读写指针 • write: 写入文件 • read: 读取文件 • autotest: 自动测试

表 2.2 fformat 指令

指令	**fformat**
描述	将整个文件系统进行格式化，即清空所有文件及目录!
使用方式	fformat

参数	无
使用示例	fformat

表 2.3　exit 指令

指令	**exit**
描述	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。若使用 ctrl + C 终止会导致无法调用析构，再次启动时可能出现错误!
使用方式	exit
参数	无
使用示例	exit

表 2.4　mkdir 指令

指令	**mkdir**
描述	新建一个目录。若出现错误，会有相应错误提示!
使用方式	mkdir
参数	可以是相对路径，也可以是绝对路径
使用示例	• mkdir dirName • mkdir ../dirName • mkdir ../../dirName • mkdir /dirName • mkdir /dir1/dirName

表 2.5　ls 指令

指令	**ls**
描述	列出当前目录中包含的文件名或目录名。若出现错误，会有相应错误提示!

使用方式	ls
参数	无
使用示例	ls

表 2.6　cd 指令

指令	**cd**
描述	改变当前工作目录。若出现错误，会有相应错误提示!
使用方式	cd
参数	可以是相对路径，也可以是绝对路径
使用示例	• cd dirName • cd ../dirName • cd ../../dirName • cd /dirName • cd /dir1/dirName

表 2.7　create 指令

指令	**create**
描述	新建一个文件。若出现错误，会有相应错误提示!
使用方式	create
参数	可以包含相对路径或绝对路径 -r 只读属性 -w 只写属性 -rw == -r -w 读写属性
使用示例	• create newFileName -rw • create ../newFileName -rw • create ../../newFileName -rw • create /newFileName -rw • create /dir1/newFileName -rw

表 2.8　delete 指令

指令	**delete**
描述	删除一个文件。若出现错误，会有相应错误提示!
使用方式	delete
参数	可以包含相对路径或绝对路径
使用示例	• delete fileName • delete ../fileName • delete ../../fileName • delete /fileName • delete /dir1/fileName

表 2.9　open 指令

指令	**open**
描述	类 Unix/Linux 函数 open，打开一个文件。若要进行文件的读写必须先 open。若出现错误，会有相应错误提示!
使用方式	open
参数	可以包含相对路径或绝对路径 -r 只读属性 -w 只写属性 -rw == -r -w 读写属性
使用示例	• open fileName -r • open ../fileName -w • open ../../fileName -rw • open /fileName -r -w • open /dir1/fileName -rw

表 2.10　close 指令

指令	**close**
描述	类 Unix/Linux 函数 close，关闭一个文件。可以对已经打开的文件进行关闭。若出现错误，会有相应错误提示!
使用方式	close <file descriptor>

参数	<file descriptor> 文件描述符
使用示例	close 1

表 2.11 seek 指令

指令	**seek**
描述	类 Unix/Linux 函数 fseek，移动读写指针。若出现错误，会有相应错误提示!
使用方式	seek <file descriptor>
参数	<file descriptor> open 返回的文件描述符 指定从 开始的偏移量 可正可负 指定起始位置 可为 0(SEEK_SET), 1(SEEK_CUR), 2(SEEK_END)
使用示例	seek 1 500 0

表 2.12 write 指令

指令	**write**
描述	类 Unix/Linux 函数 write，写入一个已经打开的文件中。若出现错误，会有相应错误提示!
使用方式	write <file descriptor>
参数	<file descriptor> open 返回的文件描述符 指定写入内容为文件 InFileName 中的内容 指定写入字节数，大小为
使用示例	write 1 InFileName 123

表 2.13 read 指令

指令	**read**
描述	类 Unix/Linux 函数 read，从一个已经打开的文件中读取。若出现错误，会有相应错误提示!
使用方式	read <file descriptor> [-o]

参数	<file descriptor> open 返回的文件描述符 [-o] -o 指定输出方式为文件，文件名为 默认为 shell 指定读取字节数，大小为
使用示例	• read 1 -o OutFileName 123 • read 1 123

表 2.14　autotest 指令

指令	**autotest**
描述	帮助测试，在系统启动初期帮助测试。测试不一定所有命令都是正确的，但是系统具有容错性不会使系统异常。
使用方式	autotest | at
参数	无
使用示例	at

2.2　程序输出分析

程序通过控制台命令行交互的方式进行用户输入引导和显示反馈输出，具体输出将在后续进行介绍。

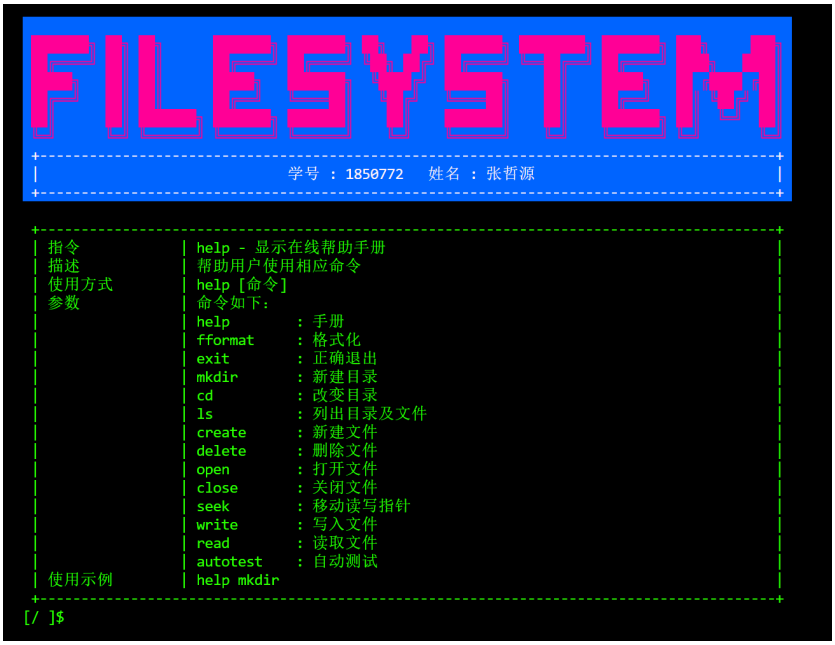


图 2.1　程序输出

装 订 线

2.3 功能分析

实验模拟 UNIX 文件操作系统实现了多用户下文件的一系列操作。具体可以实现的功能如下所示：

- 文件的创建、删除
- 文件的打开、关闭、读、写、读写指针移动
- 当前打开文件列表的显示
- 目录的创建、删除
- 工作目录的切换
- 格式化文件卷
- 退出文件系统
- 帮助文档

3 概要设计

3.1 模块设计

参照 UNIXV6++的源码，需要实现二级文件系统，需要的模块有

- DiskDriver：负责直接对磁盘文件进行读写的设备驱动模块。
- BufferManager：高速缓存管理模块，主要负责管理所有缓存块，包括缓存块的申请、释放、读写和清空功能，以及在系统退出时刷新所有缓存块。
- FileSystem：系统盘块管理模块，负责镜像文件存储空间的管理，包括 SuperBlock 空间使用、DiskINode 空间分配和数据块区空间分布的管理。提供分配和回收 DiskINode 节点、数据块节点及格式化磁盘文件的接口。
- FileManager：系统文件操作功能实现模块，封装文件系统中对文件的处理过程，具体实现打开、创建、关闭、Seek 文件指针、读取、写入和删除文件等功能。
- OpenFileManager：打开文件管理模块，负责管理打开的文件，为用户打开文件建立数据结构的连接，并提供文件描述符接口以供直接操作文件。
- User：用户操作接口模块，将用户界面执行的命令转化为相应函数的调用，同时处理输出，并检查用户输入的正确性和合法性。

3.2 数据结构设计

3.2.1 数据结构定义

表 3.1 DiskDriver 磁盘驱动

DiskDriver

```
class DiskDriver {
public:

   static const char* DISK_FILE_NAME;    /* 磁盘镜像文件名 */
private:

    FILE* fp;                            /* 磁盘文件指针 */
public:
    DiskDriver();
    ~DiskDriver();
    bool exists();
    void construct();
    void write(const void* buffer, unsigned int size, int offset = -1,int origin
```

```
= 0);

    void read(void* buffer, unsigned int size,int offset = -1, int origin = 0);
};
```

表 3.2　Buf 缓存控制块

Buf

```
class Buf {
public:
  enum BufFlag {              /* flags 中标志位 */
    B_DONE = 0x4,             /* I/O 操作结束 */
    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 */
  };
public:
  unsigned int b_flags;       /* 缓存控制块标志位 */
  int   padding;

  Buf* b_forw;                /* 缓存控制块队列勾连指针 */
  Buf* b_back;

  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_blkno;             /* 磁盘逻辑块号 */
  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 */
  int no;

public:
  Buf();
  ~Buf();
  void debugMark();
  void debugContent();
};
```

表 3.3 BufferManager

BufferManager

```
class BufferManager {
public:
    static const int NBUF = 150;       /* 缓存控制块、缓冲区的数量 */
    static const int BUFFER_SIZE = 512;    /* 缓冲区大小。 以字节为单位 */

private:
    Buf* bFreeList;              /* 缓存队列控制块 */
    Buf m_Buf[NBUF];             /* 缓存控制块数组 */
    unsigned char buffer[NBUF][BUFFER_SIZE];  /* 缓冲区数组 */
    unordered_map < int, Buf* > map;         /* 相当于设备队列 */
    DiskDriver*  m_DeviceDriver;

public:
    BufferManager();
    ~BufferManager();


    Buf* GetBlk(int blkno);      /* 申请一块缓存，用于读写设备上的块 blkno。*/
    void Brelse(Buf* bp);        /* 释放缓存控制块 buf */
    Buf* Bread(int blkno);       /* 读一个磁盘块，blkno 为目标磁盘块逻辑块号。 */
    void Bwrite(Buf* bp);        /* 写一个磁盘块 */
    void Bdwrite(Buf* bp);       /* 延迟写磁盘块 */
    void ClrBuf(Buf* bp);        /* 清空缓冲区内容 */

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

    /* 获取空闲控制块 Buf 对象引用 */
    //Buffer& GetFreeBuffer();

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

private:
    void InitList();
    void DetachNode(Buf* pb);
    void InsertTail(Buf* pb);
};
```

装 订 线

装 订 线

表 3.4 文件连接控制块

File

```
class File {
public:
    enum FileFlags {
        FREAD = 0x1,        /* 读请求类型 */
        FWRITE = 0x2,       /* 写请求类型 */
    };

public:
    File();
    ~File();

    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 */
    INode*  inode;          /* 指向打开文件的内存 INode 指针 */
    int    offset;          /* 文件读写位置指针 */
};
```

表 3.5 文件 fd 连接通路

FileManager

```
class FileManager
{
public:
    /* 目录搜索模式，用于 NameI()函数 */
    enum DirectorySearchMode
    {
        OPEN = 0,    /* 以打开文件方式搜索目录 */
        CREATE = 1,    /* 以新建文件方式搜索目录 */
        DELETE = 2     /* 以删除文件方式搜索目录 */
    };
public:
    INode* rootDirINode;
    /* 对全局对象 g_FileSystem 的引用，该对象负责管理文件系统存储资源 */
    FileSystem* fileSystem;
    /* 对全局对象 g_INodeTable 的引用，该对象负责内存 INode 表的管理 */
```

```
    INodeTable* inodeTable;
    /* 对全局对象 g_OpenFileTable 的引用，该对象负责打开文件表项的管理 */
    OpenFileTable* openFileTable;
public:
    FileManager();
    ~FileManager();
    /* Open()系统调用处理过程 */
    void Open();
    /* Creat()系统调用处理过程 */
    void Creat();
    /* Open()、Creat()系统调用的公共部分 */
    void Open1(INode* pINode, int mode, int trf);
    /* Close()系统调用处理过程 */
    void Close();
    /* Seek()系统调用处理过程 */
    void Seek();
    /* Read()系统调用处理过程 */
    void Read();
    /* Write()系统调用处理过程 */
    void Write();
    /* 读写系统调用公共部分代码 */
    void Rdwr(enum File::FileFlags mode);
    /* 目录搜索，将路径转化为相应的 INode 返回上锁后的 INode */
    INode* NameI(enum DirectorySearchMode mode);
    /* 被 Creat()系统调用使用，用于为创建新文件分配内核资源 */
    INode* MakNode(unsigned int mode);
    /* 取消文件 */
    void UnLink();
    /* 向父目录的目录文件写入一个目录项 */
    void WriteDir(INode* pINode);
    /* 改变文件访问模式 */
    //void ChMod();
    /* 改变当前工作目录 */
    void ChDir();
    /* 列出当前 INode 节点的文件项 */
    void Ls();
};
```

装 订 线

表 3.6 分配磁盘和目录结构块

FileSystem & SuperBlock & DirectoryEntry

```
class SuperBlock {
public:
    const static int MAX_NFREE = 100;
    const static int MAX_NINODE = 100;
public:
    int s_nfree;            // 直接管理的空闲盘块数量
    int s_free[MAX_NFREE];      // 直接管理的空闲盘块索引表
    int s_fsize;            // 盘块总数
    int s_flock;            // 封锁空闲盘块索引表标志
    int s_isize;            // 外存 INode 区占用的盘块数
    int s_ninode;           // 直接管理的空闲外存 INode 数量
    int s_inode[MAX_NINODE];  // 直接管理的空闲外存 INode 索引表
    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[47];          // 填充使 SuperBlock 块大小等于 1024 字节，占据 2 个扇区
};
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 */
    char name[DIRSIZ];  /* 目录项中路径名部分 */
};
class FileSystem {
public:
    // Block 块大小
    static const int BLOCK_SIZE = 512;
    // 磁盘所有扇区数量
    static const int DISK_SIZE = 16384;
    // 定义 SuperBlock 位于磁盘上的扇区号，占据两个扇区
    static const int SUPERBLOCK_START_SECTOR = 0;
    // 外存 INode 区位于磁盘上的起始扇区号
    static const int INODE_ZONE_START_SECTOR = 2;
    // 磁盘上外存 INode 区占据的扇区数
    static const int INODE_ZONE_SIZE = 1022;
```

```
    // 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 INODE_NUMBER_PER_SECTOR = BLOCK_SIZE / sizeof(DiskINode);
    // 文件系统根目录外存 INode 编号
    static const int ROOT_INODE_NO = 0;
    // 外存 INode 的总个数
    static const int INode_NUMBERS = INODE_ZONE_SIZE * INODE_NUMBER_PER_SECTOR;
    // 数据区的起始扇区号
        static const int DATA_ZONE_START_SECTOR = INODE_ZONE_START_SECTOR +
INODE_ZONE_SIZE;
    // 数据区的最后扇区号
    static const int DATA_ZONE_END_SECTOR = DISK_SIZE - 1;
    // 数据区占据的扇区数量
    static const int DATA_ZONE_SIZE = DISK_SIZE - DATA_ZONE_START_SECTOR;
public:
    DiskDriver* deviceDriver;
    SuperBlock* superBlock;
    BufferManager* bufferManager;
public:
    FileSystem();
    ~FileSystem();
    /* 格式化 SuperBlock */
    void FormatSuperBlock();
    /* 格式化整个文件系统 */
    void FormatDevice();
    /* 系统初始化时读入 SuperBlock */
    void LoadSuperBlock();
    /* 将 SuperBlock 对象的内存副本更新到存储设备的 SuperBlock 中去 */
    void Update();
    /* 在存储设备 dev 上分配一个空闲外存 INode，一般用于创建新的文件。*/
    INode* IAlloc();
    /* 释放编号为 number 的外存 INode，一般用于删除文件。*/
    void IFree(int number);
    /* 在存储设备上分配空闲磁盘块 */
    Buf* Alloc();
    /* 释放存储设备 dev 上编号为 blkno 的磁盘块 */
    void Free(int blkno);
};
```

表 3.7　文件节点控制块 INode

INode & DiskINode

```
class INode {
public:
    // INodeFlag 中标志位
    enum INodeFlag {
        IUPD = 0x2,    // 内存 INode 被修改过，需要更新相应外存 INode
        IACC = 0x4,    // 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 IFBLK = 0x6000;    /* 块设备特殊类型文件，为 0 表示常规数据文件 */
    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 unsigned int IEXEC = 0x40;      /* 对文件的执行权限 */
    static const unsigned int IRWXU = (IREAD | IWRITE | IEXEC);    /* 文件主对文件的读、写、执行权限 */
    static const unsigned int IRWXG = ((IRWXU) >> 3);          /* 文件主同组用户对文件的读、写、执行权限 */
    static const unsigned int IRWXO = ((IRWXU) >> 6);          /* 其他用户对文件的读、写、执行权限 */

    static const int BLOCK_SIZE = 512;     /* 文件逻辑块大小：512 字节 */
    static const int ADDRESS_PER_INDEX_BLOCK = BLOCK_SIZE / sizeof(int);  /* 每个间接索引表(或索引块)包含的物理盘块号 */

    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 */
    static const int LARGE_FILE_BLOCK = 128 * 2 + 6;  /* 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 HUGE_FILE_BLOCK = 128 * 128 * 2 + 128 * 2 + 6; /* 巨型文件：经二次间接索引最大可寻址文件逻辑块号 */

public:
```

```
    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
    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
    int   i_number;   // 外存 INode 区中的编号
    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_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_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

public:
    INode();
    ~INode();
    /* 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 ReadI();
    /* 根据 Inode 对象中的物理磁盘块索引表，将数据写入文件 */
    void WriteI();
    /* 将文件的逻辑块号转换成对应的物理盘块号 */
    int Bmap(int lbn);
    void IUpdate(int time);
    /* 释放 Inode 对应文件占用的磁盘块 */
    void ITrunc();
    /* 清空 Inode 对象中的数据 */
    void Clean();
    /* 将包含外存 Inode 字符块中信息拷贝到内存 Inode 中 */
    void ICopy(Buf* bp, int inumber);
};

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
    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
public:
    DiskINode();
```

```
    ~DiskINode();
};
```

表 3.8　调用接口

User

```
#pragma once
#include"FileManager.h"
#include"Sys.h"
using namespace std;

class User {
public:
    static const int EAX = 0; /* u.ar0[EAX]；访问现场保护区中 EAX 寄存器的偏移量 */

    enum ErrorCode {
        U_NOERROR = 0,     /* No u_error */
        U_ENOENT = 2,      /* No such file or directory */
        U_EBADF = 9,       /* Bad file number */
        U_EACCES = 13,       /* Permission denied */
        U_ENOTDIR = 20,      /* Not a directory */
        U_ENFILE = 23,       /* File table overflow */
        U_EMFILE = 24,       /* Too many open files */
        U_EFBIG = 27,      /* File too large */
        U_ENOSPC = 28,       /* No space left on device */
    };

public:
    INode* cdir;     /* 指向当前目录的 Inode 指针 */
    INode* pdir;     /* 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 */
    long arg[5];         /* 存放当前系统调用参数 */
    /* 系统调用相关成员 */
    unsigned int  ar0[5];      /* 指向核心栈现场保护区中 EAX 寄存器
                                  存放的栈单元，本字段存放该栈单元的地址。
                                  在 V6 中 r0 存放系统调用的返回值给用户程序，
                                  x86 平台上使用 EAX 存放返回值，替代 u.ar0[R0] */
```

```
    ErrorCode u_error;      /* 存放错误码 */
    OpenFiles 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
*/
    FileManager* fileManager;
    string ls;
public:
    User();
    ~User();
    void Ls();
    void Cd(string dirName);
    void Mkdir(string dirName);
    void Create(string fileName, string mode);
    void Delete(string fileName);
    void Open(string fileName, string mode);
    void Close(string fd);
    void Seek(string fd, string offset, string origin);
    void Write(string fd, string inFile, string size);
    void Read(string fd, string outFile, string size);
private:
    bool IsError();
    void EchoError(enum ErrorCode err);
    int INodeMode(string mode);
    int FileMode(string mode);
    bool checkPathName(string path);
};
```

3.2.2 重点数据结构图解

A. Buf 缓冲块算法

使用 LRU 算法， 每次从队列中拿出来节点（DeTachNode）,未选中则从队头拿取，用完的节点插到队尾,由于只有一个外设（磁盘镜像） 因此可以用 **unordered_map** 建立 **blk** 和 **Buf** 的模拟设备队列（方便存取）自由队列依然使用双向链表

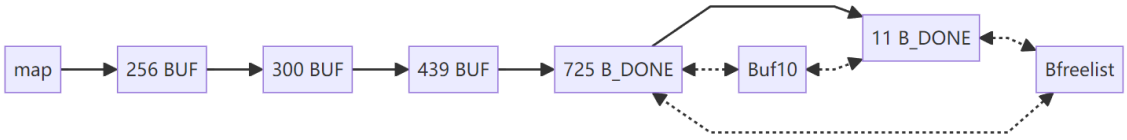


图 3.1 自由队列和设备队列

B. 磁盘存储空间分布

SuperBlock 管理磁盘块分配，Inode 对应每一个节点，SuperBlock 占 2 个盘快，Inode 区共 100 个 Inode，剩下的是文件数据区域

图 3.2　磁盘存储空间分布

C. 文件索引结构

文件索引结构放置于 Inode 的 addr 区，使用三级文件存储管理，分为小文件，大文件，巨型文件，分别使用直接索引，一级间接索引， 二级间接索引，下图给出三级索引示意图

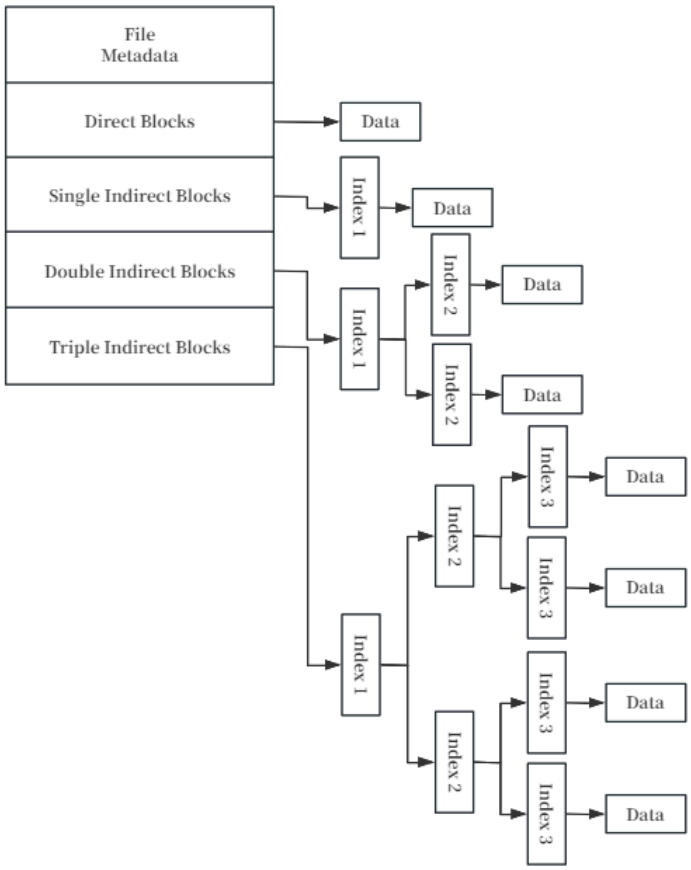


图 3.3　三级文件索引示意图

装 订 线

装 订 线

D. 文件夹打开结构

文件打开时，使用三个数据结构管理文件打开，因为是单用户，使用一个 processOpenFileTable 即可，通过 File 结构块,（存放至 OpenFileTable）,对应连接文章的 Inode,返回的 fd 未 OpenFileTable 中的索引（0 和 1 被空出来指向根目录（默认打开））

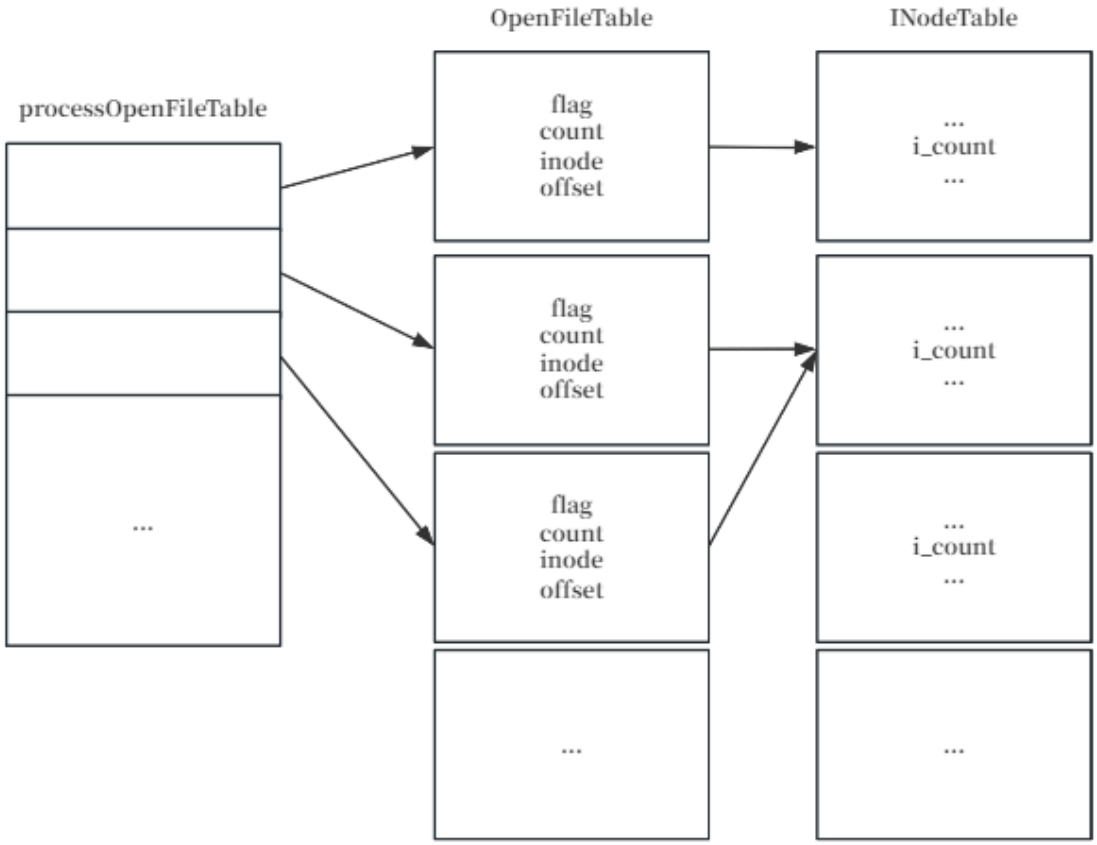


图 3.4　文件打开结构

3.3　程序流程图

全局变量 g_User 表示当前唯一用户，通过 User 中的接口，进行指令的识别和解析，并执行，通过子模块，进行文件的创建删除，目录的创建删除等操作，并更新底层数据结构

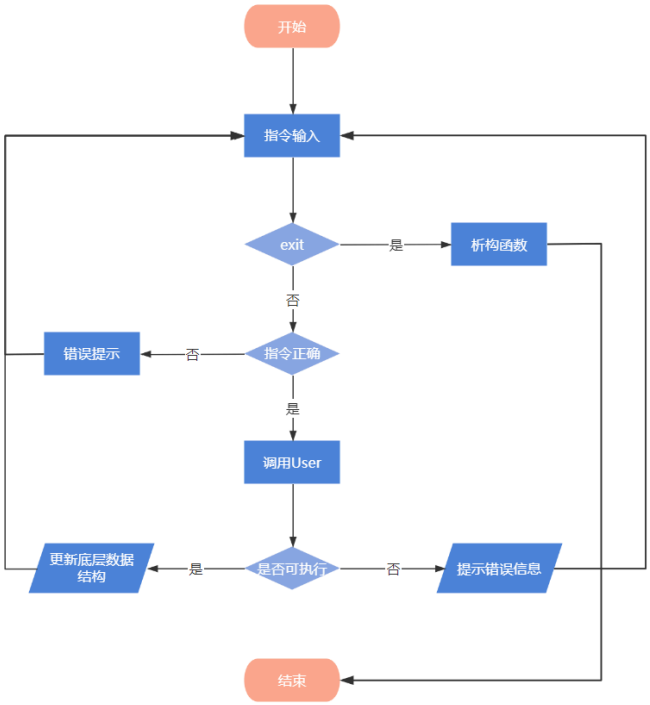


图 3.5　程序流程图

3.4　子模块调用关系

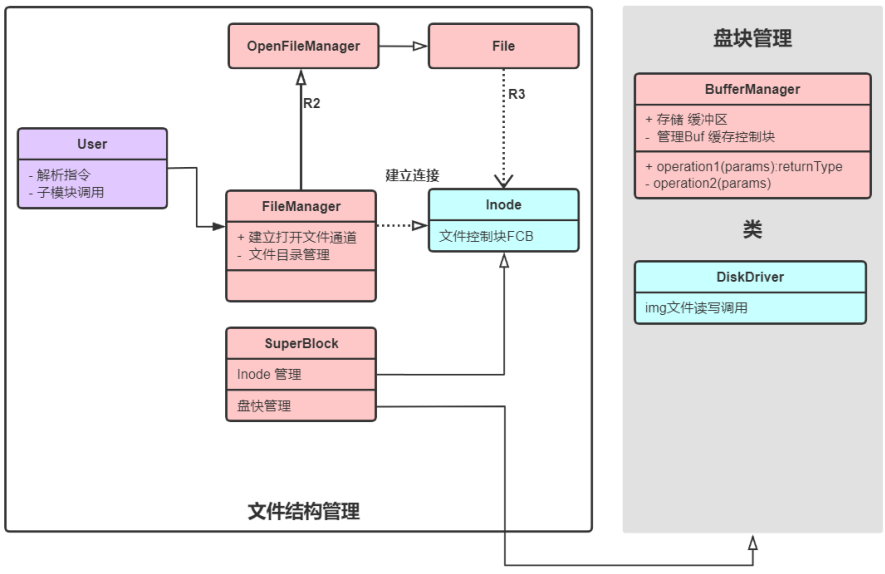


图 3.6　模块调用关系

4 详细设计

4.1 缓存设计和实现

在 UnixV6++ 系统中，缓存块的管理采用了自由队列和设备队列的方式。并支持多设备，然而，由于本次课程设计项目只对单个磁盘读写，整个模拟过程只有一个设备，无需并行操作，因此我们可以将自由队列和设备队列合并为一个缓存队列。具体实现细节如下：

A. 高速缓存初始化

初始化过程中，我们构造了一个双向循环链表，并初始化其成员变量。

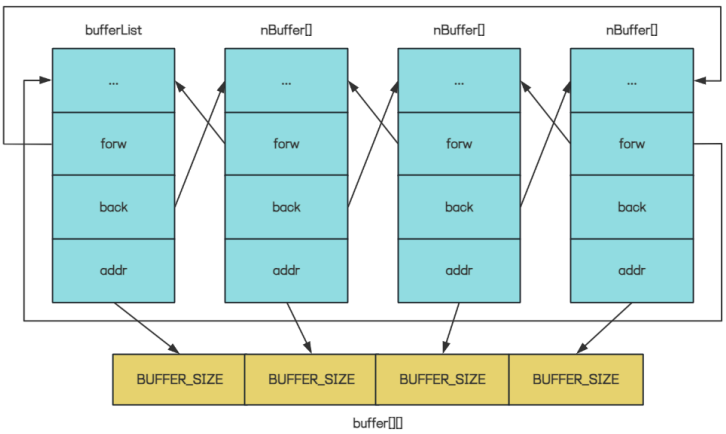


图 4.1

B. 分离插入节点函数 DetachNode/InsertTail

我们采用了 LRU（Least Recently Used，最近最少使用）算法，每次从缓存队列中分离第一个节点。由于第一个节点是过去一段时间内最少使用的节点，因此选择它进行分离。插入节点从队尾插入，保证刚使用过的块，最后被使用

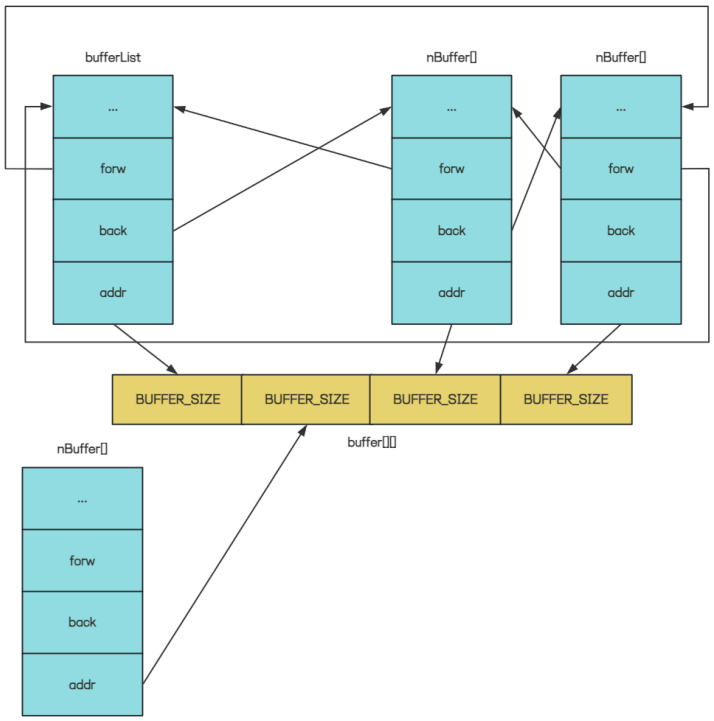


图 4.2

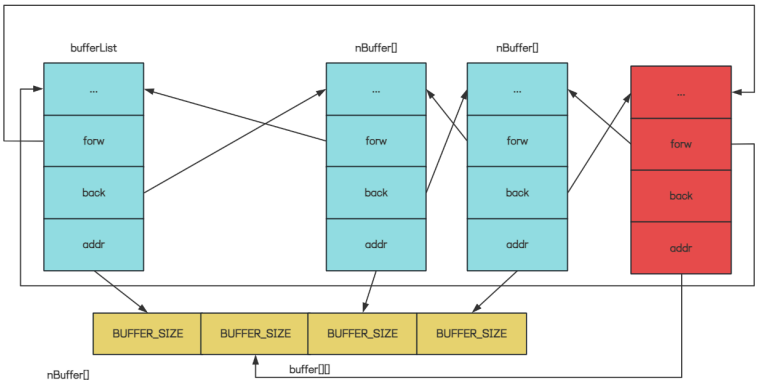


图 4.3

C. 申请缓存块函数 GetBlk

该函数用于申请一块缓存，并将其从缓存队列中取出，以便对设备块上的 blkno 进行读写操作。具体执行步骤如下：

首先在缓存队列中查找是否有缓存块的 blkno 与目标 blkno 相同。如果找到，则分离该缓存节点并返回该节点。如果未找到相应的 blkno，说明缓存队列中没有目标 blkno 的节点。这时需要分离第一个节点，并将其从缓存队列中删除。如果该节点带有延迟写标志，则立即写回数据并清空标志。将该缓存块的 blkno 设置为目标 blkno，并返回该缓存块。

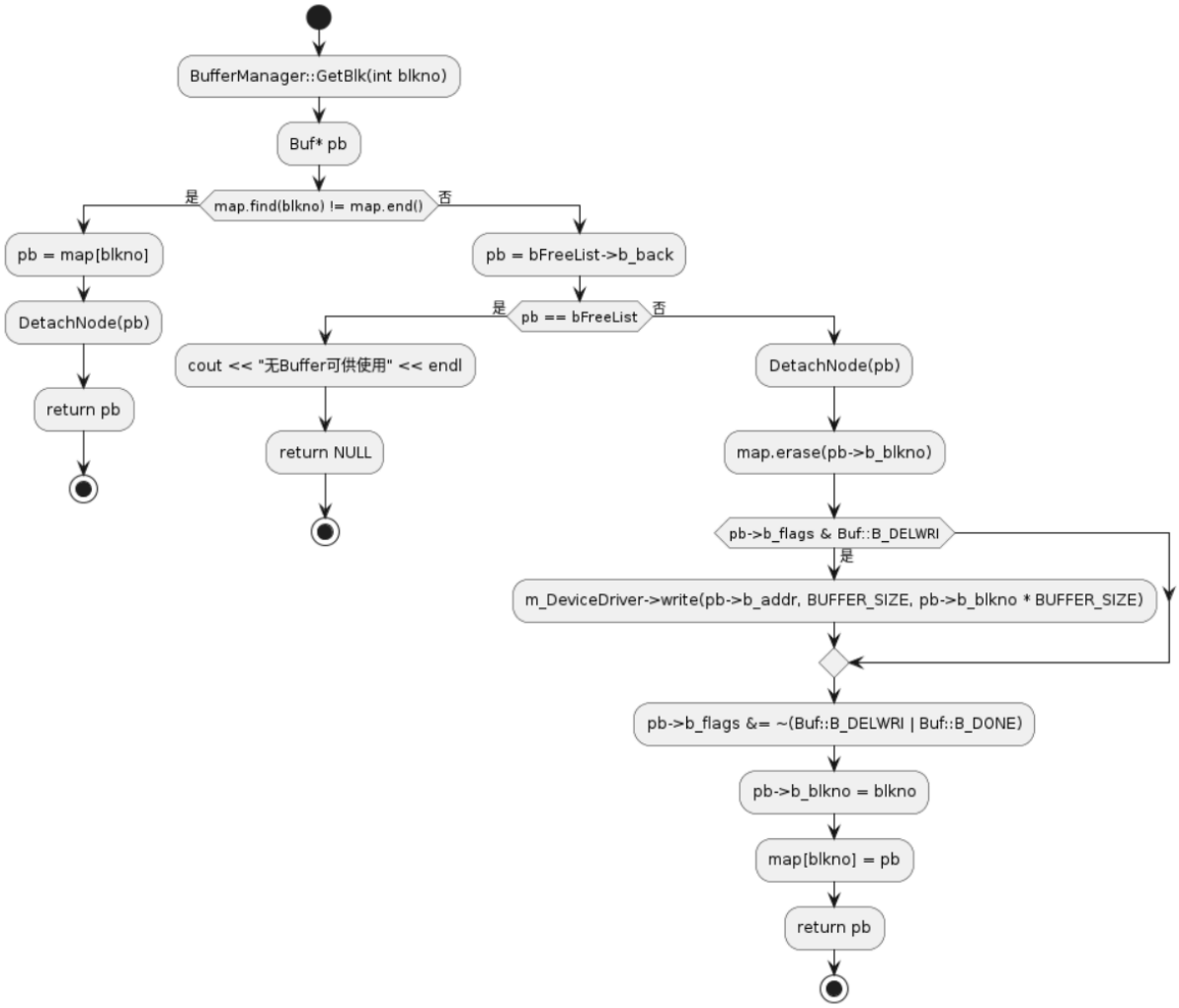


图 4.4　GetBlk 函数流程

通过上述实现，我们能够高效地模拟 Unix 文件系统中的内存高速缓存管理机制，合并了自由队列和设备队列，简化了操作步骤，提高了整体执行效率。在实际应用中，该方法可以有效减少缓存命中率低带来的性能损失。

4.2　磁盘块分配函数

同原有的 UNIXV6++ 一致，在 SuperBlock 中使用栈方式管理其中 100 个磁盘块，将所有的磁盘块使用链接分组，100 个为一组，期中每组的第一块管理下 100 块的磁盘，相关的两个函数是 ALLOC 和 FREE

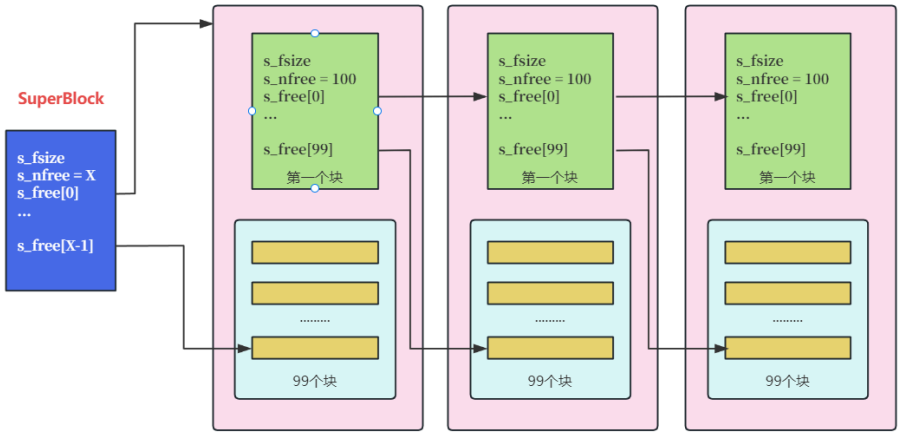


图 4.5　磁盘块分配函数示意图

A. ALLOC

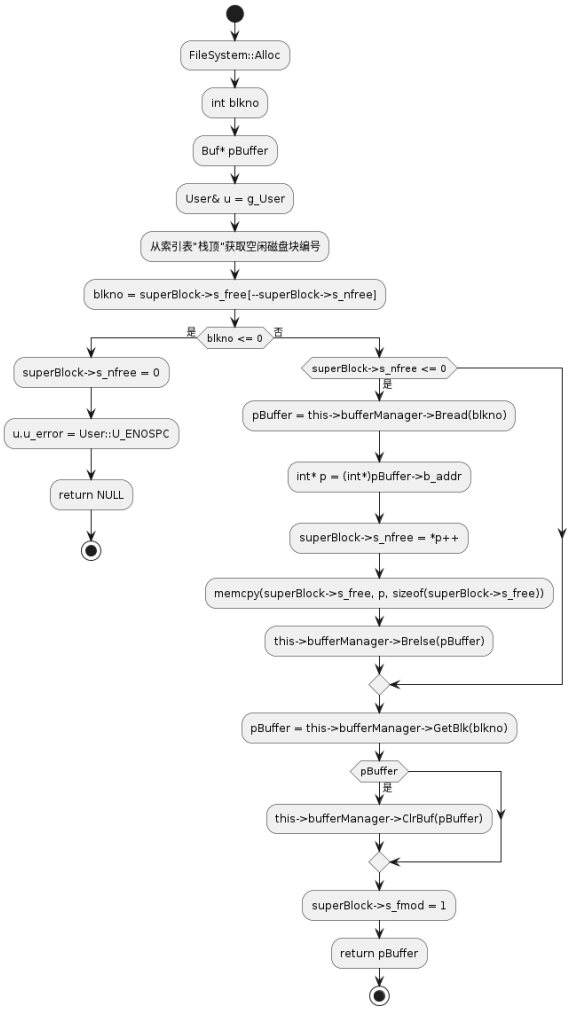


图 4.6　Alloc 函数流程

装 订 线

B. FREE

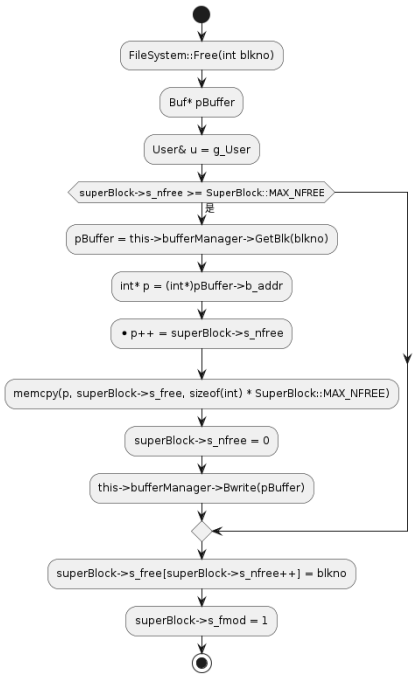


图 4.7 FREE 函数流程

4.3 Inode 节点分配

总体思想类似，由于 Inode 对应每一个单独文件， 所以 Inode 一般用于文件的创建和删除， 对应于 IALLOC 和 IFREE 两个函数

A. IALLOC

FileSystem::IAlloc 函数用于分配一个空闲的 inode，主要步骤如下：

1. 检查超级块 superBlock 中是否有可用的空闲 inode。
2. 如果没有空闲 inode，从磁盘中搜索空闲的 inode。
   - 遍历磁盘上的 inode 区域，读取每个扇区的 inode 信息。
   - 检查每个 inode 的模式，如果为 0 且未被加载到内存中，则将其标记为空闲 inode 并添加到超级块的空闲 inode 索引表中。
3. 如果找到空闲 inode，将其从超级块的空闲 inode 索引表中取出。
4. 调用 INodeTable 获取相应的 inode。
5. 如果成功获取 inode，则对其进行初始化，并更新超级块的修改标志。
6. 返回分配的 inode。

具体流程图如下：

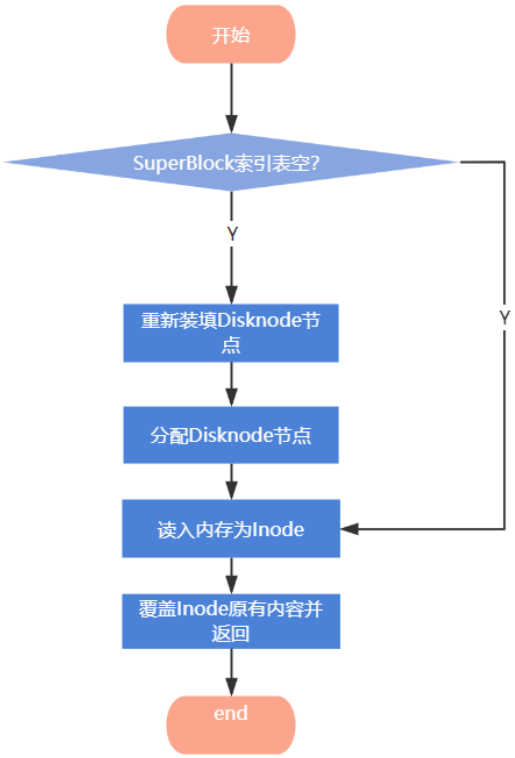


图 4.8

B. IFREE

同样 FREE,因为考虑到再被分出去时，Inode 会被覆盖写入，因此如果超出 DiskInode 索引区的容量时，可以将其散落到 DiskINode 中去，（实际操作为什么都不做）

```
void FileSystem::IFree(int number) {
    if (superBlock->s_ninode >= SuperBlock::MAX_NINODE) {
        return;
    }
    superBlock->s_inode[superBlock->s_ninode++] = number;
    superBlock->s_fmod = 1;
}
```

4.4 地址变换函数

由于文件索引最大是二级索引， 即 i_addr[0] i_addr[5] 是直接索引， i_addr[6] i_addr[7] 是 1 级索引， i_addr[8] i_addr[9] 是二级索引， 需要需要建立地址映射函数， 即把逻辑块号，对应到实际的物理块号， 此部分操作和原来的 UNIXV6++没有过多出入，实现函数为 BMap 函数

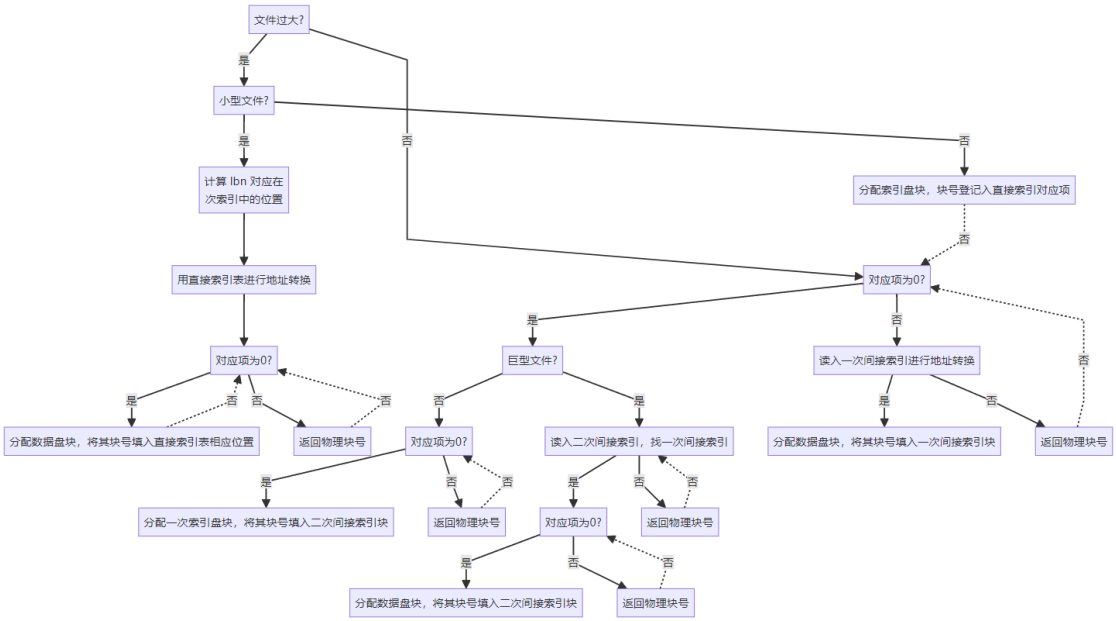


图 4.9　Bmap 函数执行流程

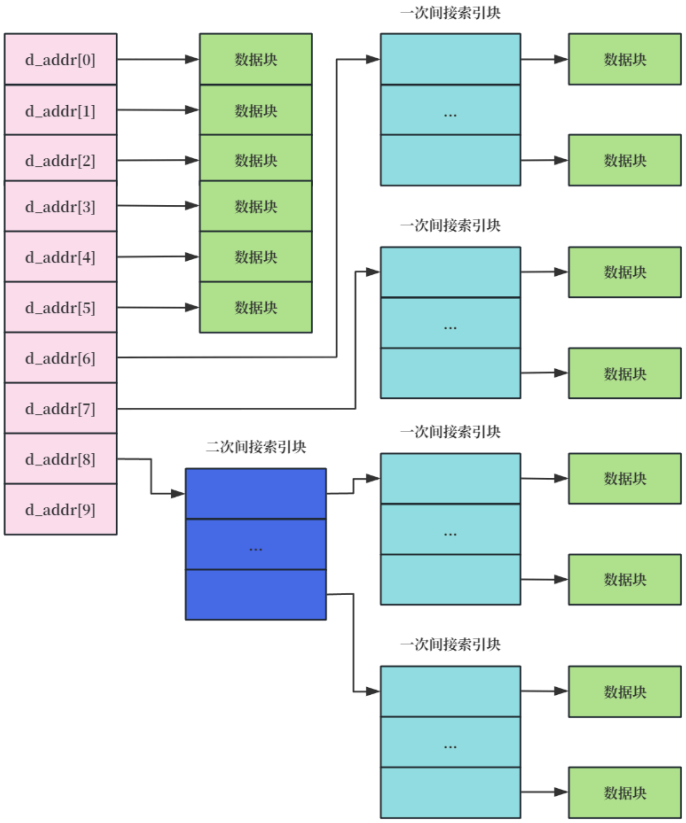


图 4.10　二级文件系统结构示意图

装 订 线

4.5 目录搜索函数

目录搜索函数，为了读出文件目录中的目录项，映射到对应的路径中去，需要有目录搜索函数，分为三个功能，创建文件，删除文件和更新文件，对应不同的模式，在 UNIXV6++ 中使用 NAMEI 函数

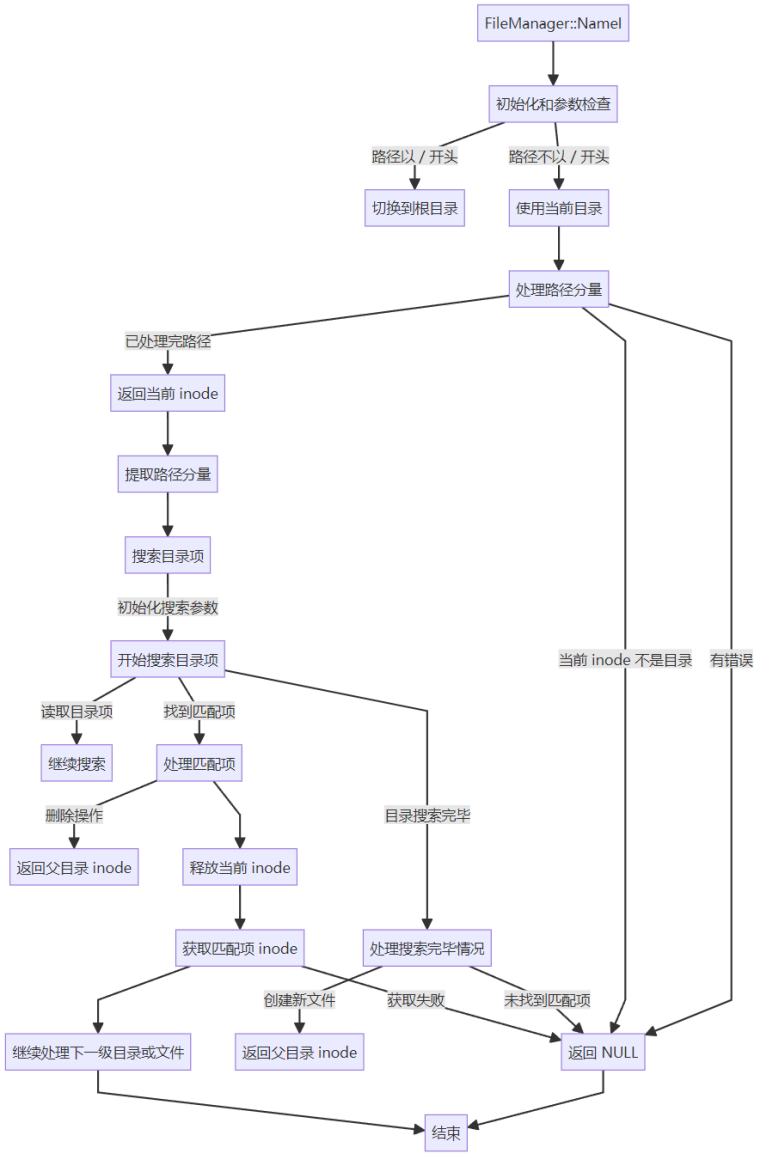


图 4.11 NameI 函数

操作流程描述如下：

1. 初始化和参数检查：
    - 如果路径以 / 开头，切换到根目录 rootDirINode。
2. 处理路径分量：
    - 如果出现错误则退出。
    - 检查是否已处理完路径名。
    - 检查当前 inode 是否为目录，不是则退出。
    - 提取路径中的下一个分量。
3. 搜索目录项：
    - 初始化搜索参数。
    - 在目录中逐项搜索：
        - 如果目录搜索完毕，根据操作模式决定是否创建新文件或报告错误。
        - 读取目录项。
        - 如果找到匹配项，跳出循环。
        - 如果是删除操作，返回父目录 inode。
4. 处理匹配项：
    - 释放当前目录 inode。
    - 获取匹配项的 inode。
    - 如果失败，返回 NULL。

5 运行与结果分析

5.1 开始运行

使用 vs_studio 2022 启动项目， 启动编译，得到可执行文件 exe,开始界面默认给出了可以支持的指令和信息

图 5.1 开始界面

程序做了错误提示，有一定的鲁棒性和错误处理能力

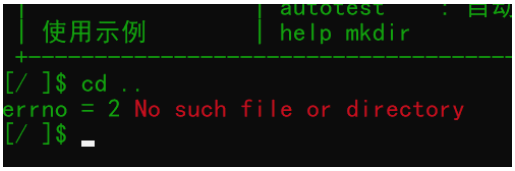


图 5.2 错误示范 1

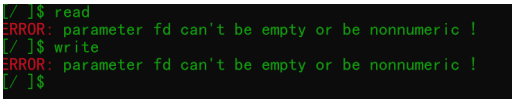


图 5.3 错误示范 2

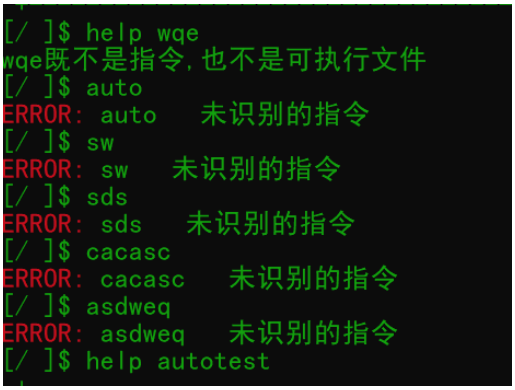


图 5.4 错误示范 3

装 订 线

5.2 用户使用说明

5.2.1 开发环境

操作系统	Window11 家庭中文版
集成开发环境	visual studio 2022 社区版
编译器	visual C++ 19
运行环境	windows cmd 窗口或 powershell

5.2.2 使用说明和注意事项

用户点击 exe 即可进入系统，本系统会在当前目录下开 img，建议使用时，单独开一个文件夹进行装入，用户保存的文件在 img 镜像中，对外不可见，只能在该文件系统中看到， 注意事项如下:

注意事项
1. 用户可以使用 fformat 格式化镜像，解决一切非法操作导致的问题
2. 非法退出（ ctrl + c ）会导致无法正确调用析构函数，可能导致错误
3. 关闭弹窗会导致无法正确调用析构函数， 可能导致错误
4. 正确退出，使用 exit 指令
5. 本系统支持指令类似 linux 操作系统，但不支持上下键读取历史指令，vim 查看文件等操作，具体支持的操作，可以使用 help 查看，具体某一指令的操作，可以使用 help [指令] 进行查看
6. 本系统给出了自动测试程序，使用 autotest1,autotest2,autotest3 即可进行相关的测试.
7. 使用自动测试前需要 fformat 文件系统（ fd 写死了，否则会对不上 ）
8. 测试时，需要保证测试读写用的文件，和可执行文件在同一目录下，具体包括该实验报告和一张测试用的图片(压缩包中的文件结构不要破坏)
9. 本次程序运行结束关闭再打开时，文件系统中的文件依然会保留
10. 测试程序中支持读写的文件上限写死固定，上限为文件的大小，(注意：不是文件占用大小)，超出这个上限可能导致无法读全文件。
11. 测试程序会有多余的输出提示， 一直回车就好。

5.3 测试用例和结果

为了测试文件系统，这里使用了三个测试，来检测文件系统的正确性，每次测试前确保 fformat 系统

5.3.1 测试用例 1

测试文件读写指针的正确性

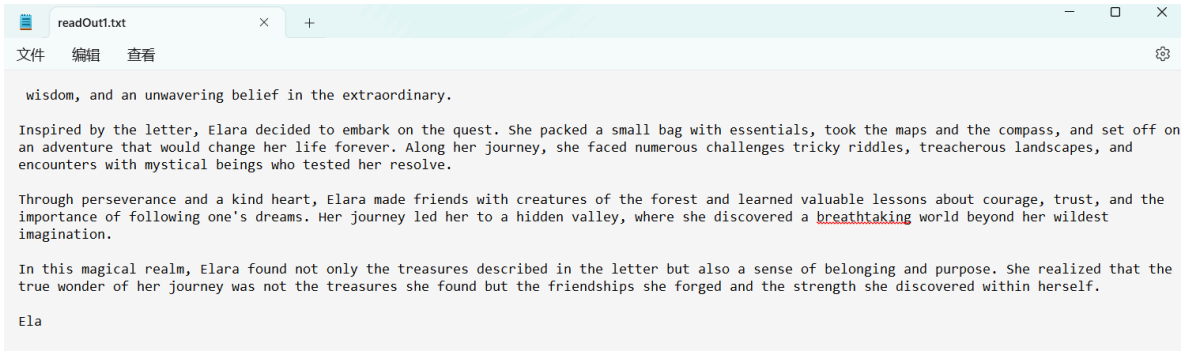
测试数据 1: 使用 autotest1 执行

```
"mkdir test1",
"cd test1",
"mkdir dir1",
"mkdir /test1/dir2",
"ls",
"cd dir1",
"ls",
"create file1 -rw",
"ls",
"open file1 -rw",
"write 4 testFile.txt 4096",
"seek 4 0 0",
"read 4 -o readOut1.txt 2000",
"seek 4 2000 0",
"read 4 -o readOut1.txt 2000",
"seek 4 0 1",
"read 4 50",
"seek 4 0 0",
"close 4",
"cd /",
"ls",
"####"
```

表 5.1　测试程序 1 运行结果

测试结果

readOut1.txt 中读出来的结果



控制台读到的 50 字节文字：

ra returned to her village as a changed person. Sh

控制台输出:

```
[/ ]$ autotest1
测试程序1文件指针测试开始执行，请确保是格式化后执行以确保正确
 按回车键开始测试
[/ ]$ mkdir test1
[/ ]$ cd test1
[/test1/ ]$ mkdir dir1
[/test1/ ]$ mkdir /test1/dir2
[/test1/ ]$ ls
dir1
dir2

[/test1/ ]$ cd dir1
[/test1/dir1/ ]$ ls

[/test1/dir1/ ]$ create file1 -rw
[/test1/dir1/ ]$ ls
file1

[/test1/dir1/ ]$ open file1 -rw
open success, return fd=4
[/test1/dir1/ ]$ write 4 testFile.txt 2519
write 2519bytes success !
[/test1/dir1/ ]$ seek 4 0 0
[/test1/dir1/ ]$ read 4 -o readOut1.txt 1000
read 1000 bytes success :
read to readOut1.txt done !
[/test1/dir1/ ]$ seek 4 1000 0
[/test1/dir1/ ]$ read 4 -o readOut1.txt 1000
read 1000 bytes success :
read to readOut1.txt done !
[/test1/dir1/ ]$ seek 4 0 1
[/test1/dir1/ ]$ read 4 50
read 50 bytes success :
ra returned to her village as a changed person. Sh
[/test1/dir1/ ]$ seek 4 0 0
[/test1/dir1/ ]$ close 4
[/test1/dir1/ ]$ cd /
[/ ]$ ls
test1

[/ ]$ 测试1执行完成，请到当前目录中的readOut1.txt中查看，再次进行测试请先fformat格式化！
```

5.3.2 测试用例 2

/test/Jerry 创建和读写

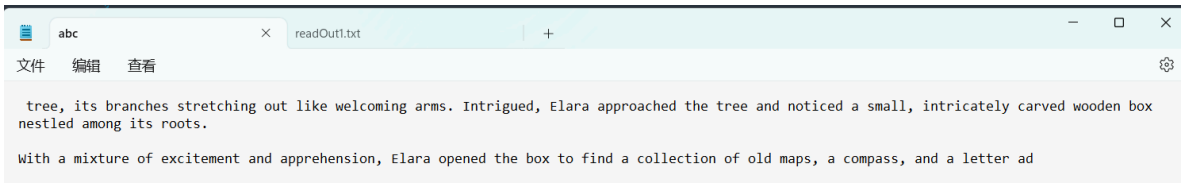
测试数据 2: 使用 autotest2 执行

```
"mkdir test2",
"cd test2",
"create Jerry -rw",
"ls",
"open Jerry -rw",
"write 2 testFile.txt 800",
"seek 2 500 0",
"read 2 -o abc 300",
"write 2 abc 300",
"seek 2 0 0",
"read 2 -o readOut2.txt 1100",
```

```
    "ls",
    "cd /",
    "ls",
    "####"
```

表 5.2　测试程序 2 运行结果

测试结果
abc 中读出来的结果



控制台输出:

```
[/ ]$ autotest2
 测试程序2/test2/Jerry读写测试开始执行，请确保是格式化后执行以确保正确
 按回车键开始测试
[/ ]$ mkdir test2
[/ ]$ cd test2
[/test2/ ]$ create Jerry -rw
[/test2/ ]$ ls
Jerry

[/test2/ ]$ open Jerry -rw
open success, return fd=2
[/test2/ ]$ write 2 testFile.txt 800
write 800bytes success !
[/test2/ ]$ seek 2 500 0
[/test2/ ]$ read 2 -o abc 300
read 300 bytes success :
read to abc done !
[/test2/ ]$ write 2 abc 300
write 300bytes success !
[/test2/ ]$ seek 2 0 0
[/test2/ ]$ read 2 -o readOut2.txt 1100
read 1100 bytes success :
read to readOut2.txt done !
[/test2/ ]$ ls
Jerry
```

5.3.3　测试用例 3

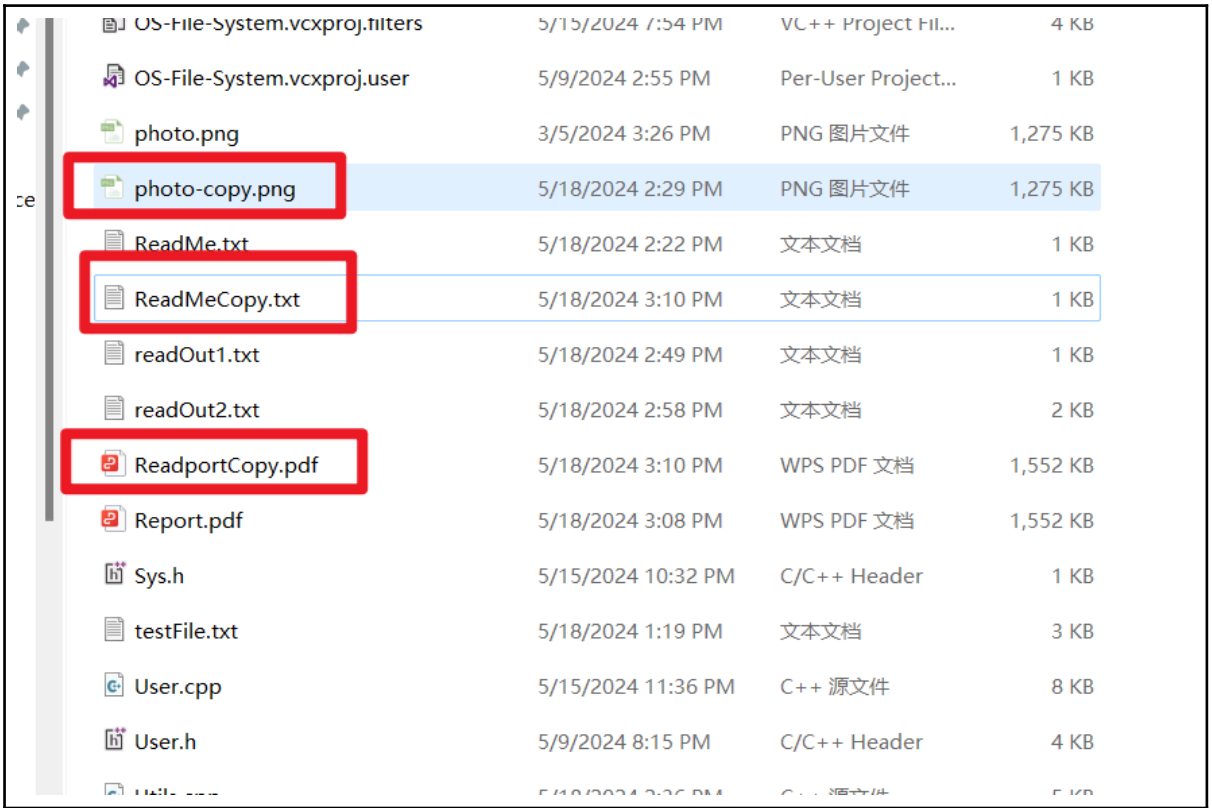
- ReadMe.txt
- 图片读写
- pdf 文档读写

测试数据 3: 使用 autotest3 执行

```
"mkdir /bin",
"mkdir /etc",
"mkdir /home",
"mkdir /dev",
"ls",
"cd home",
"ls",
"create texts -rw",
"ls",
"create photos -rw",
"ls",
"create /home/reports -rw",
"ls",

"open texts -rw",
"write 7 ReadMe.txt 319",
"seek 7 0 0",
"read 7 -o ReadMeCopy.txt 319",
"close 7",

"open photos -rw",
"write 7 photo.png 1305209",
"seek 7 0 0",
"read 7 -o ReadMeCopy.txt 319",
"close 7",

"open reports -rw",
"write 7 Report.pdf 1305209",
"seek 7 0 0",
"read 7 -o ReadportCopy.txt 1305209",
"close 7",
"ls",
```

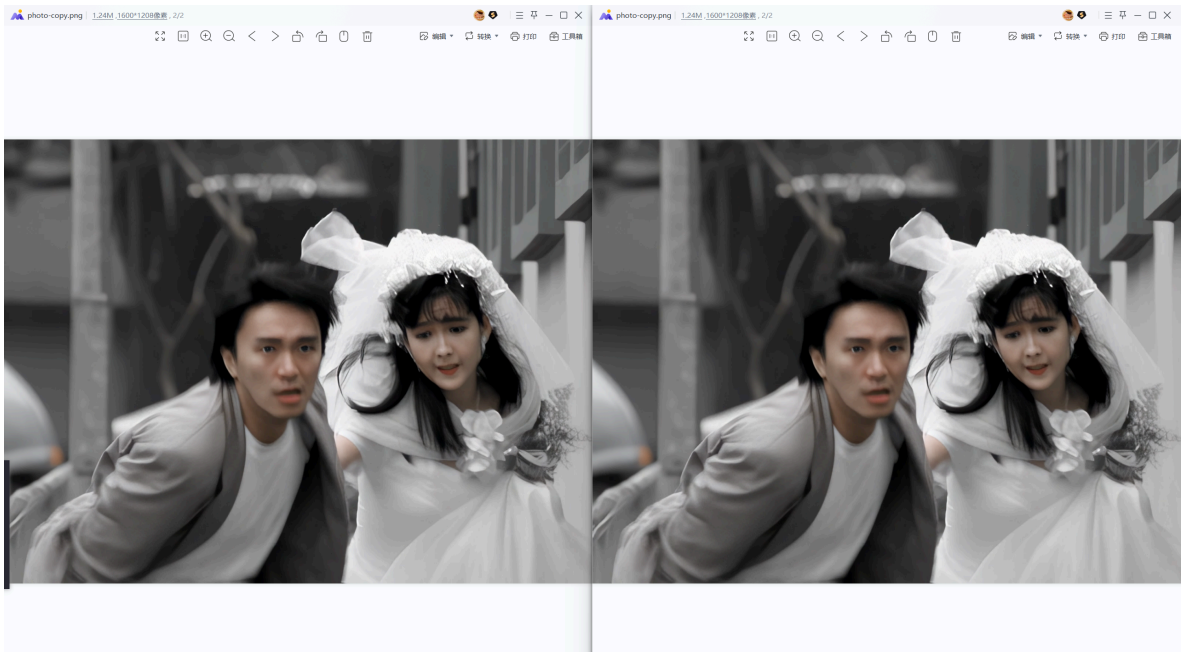
表 5.3 测试程序 3 运行结果

测试结果
生成的三个文件

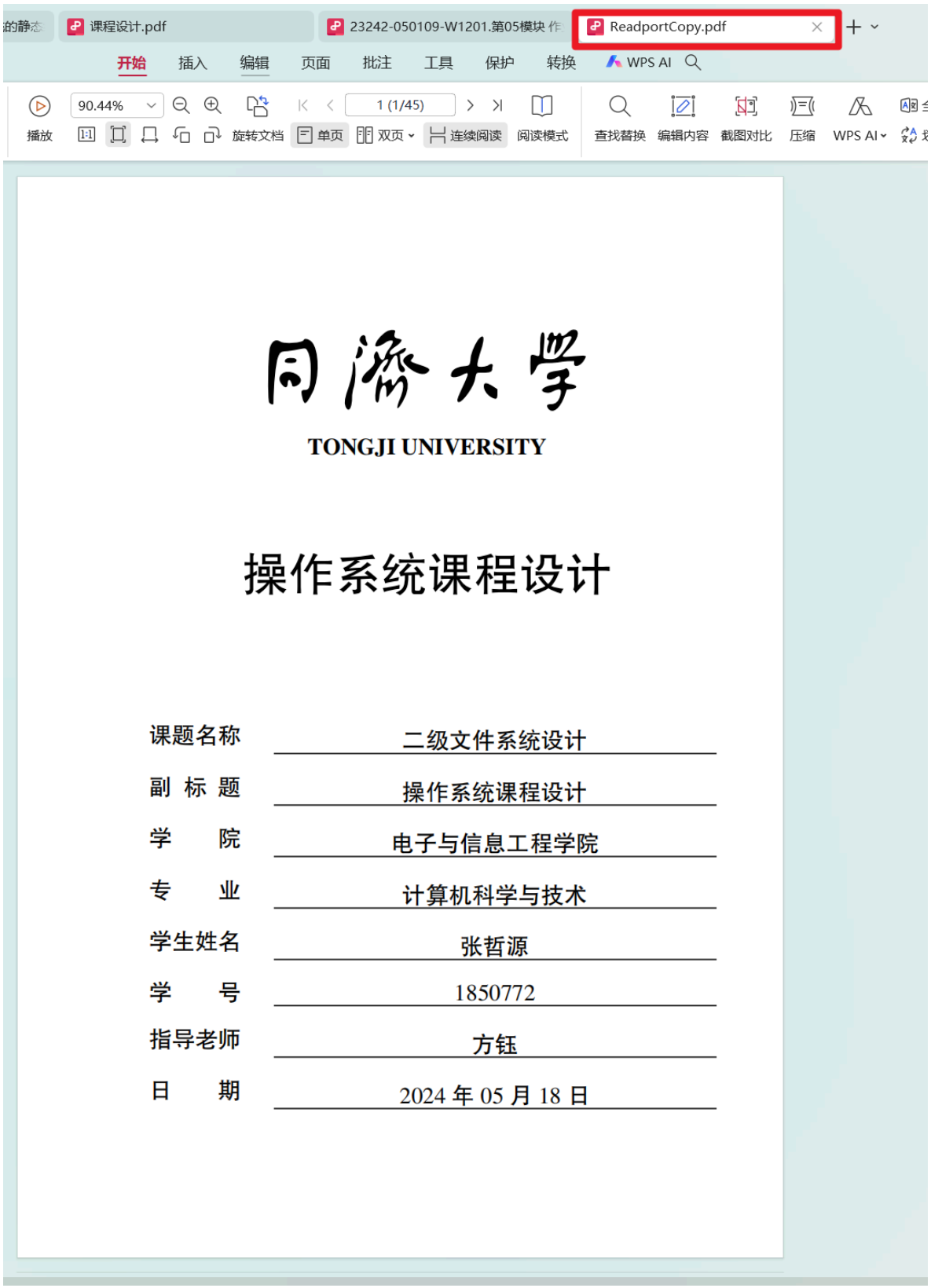


检测正确性：

两张图片



PDF 报告


课程设计.pdf
23242-050109-W1201.第05模块 作
ReadportCopy.pdf
开始 插入 编辑 页面 批注 工具 保护 转换 WPS AI
播放 90.44% 1 (1/45) 旋转文档 单页 双页 连续阅读 阅读模式 查找替换 编辑内容 截图对比 压缩 WPS AI
同济大学
TONGJI UNIVERSITY
操作系统课程设计
课题名称 二级文件系统设计
副 标 题 操作系统课程设计
学 院 电子与信息工程学院
专 业 计算机科学与技术
学生姓名 张哲源
学 号 1850772
指导老师 方钰
日 期 2024 年 05 月 18 日

控制台输出:

```
[/home/ ]$ create texts -rw
[/home/ ]$ ls
texts

[/home/ ]$ create photos -rw
[/home/ ]$ ls
texts
photos

[/home/ ]$ create /home/reports -rw
[/home/ ]$ ls
texts
photos
reports

[/home/ ]$ open texts -rw
open success, return fd=7
[/home/ ]$ write 7 ReadMe.txt 319
write 319bytes success !
[/home/ ]$ seek 7 0 0
[/home/ ]$ read 7 -o ReadMeCopy.txt 319
read 319 bytes success :
read to ReadMeCopy.txt done !
[/home/ ]$ close 7
[/home/ ]$ open photos -rw
open success, return fd=7
[/home/ ]$ write 7 photo.png 1305209
write 1305209bytes success !
[/home/ ]$ seek 7 0 0
[/home/ ]$ read 7 -o ReadMeCopy.txt 319
read 319 bytes success :
read to ReadMeCopy.txt done !
[/home/ ]$ close 7
[/home/ ]$ open reports -rw
open success, return fd=7
[/home/ ]$ write 7 Report.pdf 1588500
write 1588500bytes success !
[/home/ ]$ seek 7 0 0
[/home/ ]$ read 7 -o ReadportCopy.txt 1588500
read 1588500 bytes success :
read to ReadportCopy.txt done !
[/home/ ]$ close 7
[/home/ ]$ ls
texts
photos
reports

[/home/ ]$ 测试3执行完成，请到当前目录中的out3.pdf和out3.png中查看,再次进行测试请先fformat格式化！
```

装 订 线

6 实验总结

整个操作系统课设 + 报告耗时将近 10 天时间，从一开始打算自己写，只写要用的部分，到后面转变思路变成改 UNIXV6++ 源码，去掉不用的部分，期间花费了很多精力研究内核和 C++的一些语法特性，中间收货颇多，虽然耗费了很多时间，但是最后完成此文件系统，让我对操作系统内核又有了一个全新的认识。

首先是对项目整体的架构，我觉着很幸运是写这样一个单线程，单进程的系统，让我能够集中精力去研究文件系统的组织，而不是关心操作系统中复杂的同步和异步问题（好像隔壁班写多用户文件系统）。虽然实际操作系统中肯定不是单进程问题，但是我觉着此次课设更重要的是帮助我理解文件的组织架构过程。整个系统大体还是采用 unixv6++的源代码进行改写，和课程学习的操作系统，并没有过多的区别，只不过某些细节的实现，需要修改源码，其中有几个非常重要的函数比如说 NameI , GetBlk, Bmap 等， 这些源码读懂并正确需改，其实并不容易。

第二个就是精简源码的过程，因为是单线程，所以要从 kernel 中精简自己要写的部分，把他们变成单个的全局变量，比如说设备，此系统中就只有一个设备， 因此不需要考虑 buf 中增加两队指针，为了方便建立映射，其实使用 unordered_map 会更容易一点，再比如说此系统只有一个 User, 一个进程，kernel 中复杂的进程管理相关的内容就可以去掉， 只需要设置一个全局变量即可。

其三是一些美中不足，比如说 SuperBlock 对磁盘块的管理，分配的算法依然会使得其中很多块频繁被使用，而另外的一些块，几乎不能被用到，我个人考虑过用队列去实现，可是在最后调试过程中发现总会遇到一些无法解释的问题和Bug,时间有限,最后还是被迫改回了原来栈组织的分组链式结构。

第四是一些新的想法和期待，现代 linux 操作系统中，有一个很重要的 vim 编辑器，包括还有通过上下键可以实现指令的历史读取等操作,（想过用一个全局的 List 可以实现）不过后面确实没时间弄了，也就不了了之了。 再就是 vim 编辑器，实现这个感觉还是比较困难的。

本科期间最后几个比较有含金量的课设了，可能读研或者工作以后就不会再有机会去研究操作系统内核了， 不过还是很感谢 tj 的操作系统，让我能够如此直观的认识操作系统内核，写课设前在对分易上 1.5 倍速回听复习时有一种在 b 站听国外公开课的感觉，最后祝愿学校的课程能越做越好，让学弟学妹们未来能够学习到更好的知识。

参考文献

[1] 尤晋元. UNIX 操作系统教程[M]. 1 版. 西安: 电子科技大学出版社, 1985.

[2] LIONS J. 莱昂氏 UNIX 源码分析[M]. 1 版. 西安: 机械工业出版社, 2006.

[3] 汤小丹、梁红兵等. 计算机操作系统[D]. 2013.