同济大学计算机科学与技术系

操作系统课程设计报告

项目名称 类 Unix 二级文件系统

专　　业 计算机科学与技术

授课老师 方钰

姓　　名 毛朝炜

学　　号 2053459

日　　期 2023 年 5 月 25 日

目录

1 项目概要

本项目构造了一个单进程的类 Unix 二级文件系统，使用一个普通的大文件（mcwDisk.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 512 字节的数据块为单位。

该文件系统的设计思路参考 Unix V6++，主要实现了以下功能：

- 实现对该逻辑磁盘的基本读写操作，并在该逻辑磁盘上定义了二级文件系统结构。包括存储资源管理信息块 Superblock、外存索引结点 DiskInode 区和文件数据区。
- 实现了对文件存储资源的有效管理。DiskInode 结构通过“多级混合索引方式”实现了文件索引结构，Superblock 通过“栈方式”管理一定数量的空闲数据块和 DiskInode 结点，并采用“成组链接法”管理多余的数据块。
- 实现了“树形带交叉勾连”的目录结构，目录检索算法和目录的增、删、改的设计与实现。
- 实现了内存的文件打开结构，提高文件操作效率。其中，利用内存 Inode 结构和内存索引结点表管理近期使用的文件；利用打开文件控制块 File 和系统打开文件表实现文件的动态管理；为进程设置进程打开文件描述符表 OpenFiles，并完成对前述数据结构的勾连。
- 实现了高速缓存结构。由于只考虑单用户单进程单设备，故缓存队列做了合理的简化，只保留了一个自由队列，但功能上保证了：按照 LRU 方式管理缓存、缓存尽可能重用、缓存延迟写回，利用高速缓存块提高了文件读取的效率。

为用户提供基本文件操作接口，具体说明见需求分析。通过文件操作接口调用相应的系统调用函数，进而调用文件系统的各功能函数和结构实现相应功能，包括但不限于：文件的创建、打开、读写、关闭和删除，目录的创建、结构展示和跳转、文件读写指针的移动等。

本实验报告的行文结构安排如下：

- 需求分析：介绍本次课程设计的具体需求，包括程序功能需求、程序交互需求，以及用户其余需求和程序可行性说明。
- 概要分析：说明文件系统的整体设计思路，包括子模块的划分、数据结构的设计、主程序流程和模块间调用关系等。
- 详细设计：进一步说明各子模块的具体设计思路，包括对重点变量和函数的说明，对重点功能的流程和函数调用说明。
- 调试分析及正确性验证：展示运行结果，测试各条指令，并说明运行中遇到的主要问题和解决方案。
- 用户使用说明：用户使用方法的介绍，展示各指令的具体使用方法和可能产生的错误等。

2 需求分析

2.1 程序功能需求

使用一个普通的大文件来模拟 Unix V6++的一张磁盘，磁盘中存储的信息以 512 字节的数据块为单位。至少应当包含以下结构和功能的设计与实现：

1) 磁盘文件结构

- 在一级文件系统上定义二级文件系统结构
- 定义 SuperBlock 结构，完成 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。
- 定义磁盘 Inode 节点结构，实现 Inode 区的组织，定义多级索引结构，以及索引结构的生成和检索

2) 文件目录结构

- 定义目录文件的结构
- 目录结构的检索算法的设计与实现
- 目录结构增、删、改的设计与实现

3) 文件打开结构

- 文件打开结构的设计：内存 Inode 节点，File 结构，进程打开文件表等
- 内存 Inode 节点的分配与回收
- 文件打开、关闭过程

4) 磁盘高速缓存

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

5) 文件操作接口

本次实验需要为用户提供的文件操作接口包括：

fformat、ls、mkdir、fcreat、fopen、fclose、fread、fwrite、flseek、fdelete…

表 2.1 展示了各指令的具体设计和功能说明，为方便用户使用，学生加入了 help、autotest、shtree、exit 等四条指令。

指令名称	功能说明
fformat	格式化文件卷
ls	查看当前目录内容
mkdir <dirname>	创建新目录
fcreat <filename>	在当前文件路径下创建新文件
fopen <file_name>	打开某文件，返回该文件的句柄指针 fd
fclose <fd>	关闭句柄<fd>指向的文件
fread <fd_of_src> <N> <dst_file>	读取句柄<fd_of_src>所指文件的<N>字节到目标文件<dst_file>，其中<dst_file>缺省则输出到 cmd 窗口
fwrite <src_file> <N> <fd_of_dst>	从<src_file>写<N>字节到句柄<fd_of_dst>所指文件
flseek <fd> <offset> <mode>	定位文件读写指针，<mode>取 beg/cur/end，分别表示从句柄<fd>，所指文件的“起始、当前、结束位置”偏移<offset>字节
fdelete <fd>	删除句柄<fd>所指文件
cd <route_to_dir>	通过相对或绝对路径进入某目录
help <command>	指令说明，<command>项可选，表示介绍某一指令
autotest	自动测试文件系统各指令
shtree <dirname>	列出<dirname>下的文件树结构
exit	退出系统，延迟写的缓存此时写回

表 2.1 用户指令说明

6）其它需求

- 能通过命令行方式完成要求的测试，具体见第五部分。
- 程序运行结果应当清晰直观，满足用户友好型需求。

2.2 程序交互需求

- **输入形式说明**

采用命令行方式实现交互，指令格式在表 2.1 已展示。

需要读写的本地文件应当放在项目文件夹“源代码\TJOS_FileSystem\”中，即和源代码在同一目录下。

- **输出形式说明**

用户输入的命令一定会在 cmd 控制台产生输出响应，包括正确指令的处理结果、错误指令的提示和可能的原因、相应解决方法的提示等，具体见第 6 部分用户使用说明。

程序的交互界面如图 2.1 所示，仿照常见的 cmd 交互思路。

图 2.1 cmd 交互界面

3 概要设计

3.1 总体设计思路

通过自顶向下的设计思路，可以将文件系统划分为几个子功能模块，将复杂问题具体化。本次实验将类 Unix 二级文件系统划分为：缓存管理模块、文件系统模块、用户界面模块。其中缓存管理模块又分为磁盘驱动部分和高速缓存管理部分，文件系统模块分为磁盘存储空间管理部分和文件系统实施部分。各功能模块由相应的全局类对象实现。

1) 磁盘驱动部分

磁盘驱动部分负责磁盘文件的管理，在一级文件系统的层级上实现对磁盘文件的直接读写。

Unix V6++中需要维护多个设备和进程，且将设备分为了块设备和字符设备，故其设备驱动部分相对复杂。本实验简化为一个 DiskDriver 类直接完成 Unix V6++中设备驱动部分的工作。

2) 高速缓存管理部分

用户地址空间和磁盘的数据交换是通过高速缓存来完成的，通过在内存中读写缓存，配合延迟写回等机制，能有效提高文件读写的效率。这一部分交付给 Buf 类和 BufferManager 类来完成。

BufferManager 类负责完成高速缓存机制，具体包括缓存块的申请、分配、释放、读写，缓存队列的管理等。它向上层文件系统提供统一的访问缓存块的接口，间接调用设备驱动对象实现真实的磁盘读写。而对于每一个具体的缓存块的管理，则是 Buf 类的任务，它具体地记载着缓存的使用情况等信息。

3) 磁盘存储空间管理部分

磁盘存储空间管理部分负责二级文件系统的定义，实现对磁盘存储空间的维护管理。相应的类有：FileSystem、SuperBlock、DiskInode。

SuperBlock 负责管理外存索引节点和空闲数据块，DiskInode 负责管理磁盘上的 Inode 节点。而 FileSystem 则负责整个磁盘文件结构的定义，并管理上述文件系统资源的分配和回收，包括：SuperBlock 构造内存映像，并维护磁盘 SuperBlock；Inode 节点的分配和回收算法的设计与实现；数据盘块的分配和回收算法的设计与实现；对 DiskInode 节点的索引结构 DirectoryEntry 的管理等。

4) 文件系统实施部分

文件系统实施部分负责在内存中实现文件打开结构，并为进程打开文件建立内核数据结构之间的勾连关系，使系统能高效的读写磁盘中的文件。相应的类有：OpenFileTable、InodeTable、Inode、File、OpenFiles、SysCall。

Inode 对象是 DiskInode 对象的内存映像，InodeTable 对象管理所有的内存 Inode 节点，File 结构是打开文件的控制块，能记载文件打开的动态权限等信息，OpenFiles 和 OpenFileTable 是构成内存文件打开结构的重要组成部分，SysCall 类负责管理文件处理相关的系统调用，其功能相当于文件管理类，封装了所需的文件系统的各种系统调用的处理过程，使得文件系统访问的具体细节对外透明。

上述类将在后续详细说明。

5) 用户界面部分

用户界面部分为用户提供能直接可用的操作文件的 cmd 命令，并通过各指令的系统调用让文件系统执行相应动作。该部分由 User 类完成。

User 类实现顶层的用户接口模块，记录了当前路径、系统调用参数、返回值、错误码等信息，其成员函数即文件操作接口，供 main 函数调用。User 的成员函数首先会对系统调用参数的预处理和判错处理，随后调用相应的 SysCll 函数，完成系统调用。

3.2 数据结构设计

分析了文件系统各模块的划分和功能后，下面将介绍其中数据结构的设计，这部分将说明各类的作用，还会概括重要数据成员的作用，而对于重要函数的作用和具体实现将在第四部分中展开介绍。

1. 缓存管理与磁盘驱动

缓存控制块 Buf

每一个缓存块由一个 Buf 结构来描述，Buf 结构并不用于提供缓存控制方法，只是简单地记录缓存的使用情况等相关信息，其数据结构定义如下，各数据成员的含义已通过注释标注。

```
class Buf
{
public:
        enum BufFlag                    // b_flags 中标志位
        {
                B_DONE = 0x4,           // I/O 操作结束
                B_DELWRI = 0x80         // 延迟写
        };
        /* 缓存控制块队列勾连指针 */
        Buf* b_forw;
        Buf* b_back;
        unsigned int    b_flags;        // 缓存控制块标志位
        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;        // 磁盘的逻辑块号 0 开始
        int             no;             // 加入一个 number 记录 buf 对象的标号
        ......
};
```

- **缓存控制块的设计：**

• 缓存块的管理：对于单用户单进程单设备的系统而言，缓存队列可以简化为一个全局的自由队列，故学生只定义了一对队列勾连指针，通过 b_forw、b_back 维护一个双向循环的自由缓存队列。通过该队列就能实现按 LRU 算法的缓存块分配和回收算法，后续将具体说明。

• 缓存块的读写：在设计 Buf 结构时要考虑到未来如何实现对磁盘文件的读写。读文件时，对于已经在缓存块中的文件，我们不在需要进行 IO，直接读取即可；写文件时，如果当前缓存块没有写满，我们不急于将其写回，采用延迟写的策略。上述过程的实现依赖 b_flags 和 b_blkno，通过 b_blkno 可以关联缓存块和物理数据块，而缓存块是否可重用取决于自由队列中缓存块的标志位 f_flags 是否含有 B_DONE 标志位，是否需要延迟写则取决于 f_flags 是否含有 B_DELWRI 标志位。

• 其它数据成员：b_wcount 记录本次读写需要传输的字节数。b_addr 则记录当前缓存块在内存空间中的起始地址。BufferManager 中会具体定义一个缓存块的大小和缓存个数。

缓存管理类 BufferManager

BufferManager 用于定义具体的缓存池细节，并向上层提供访问高速缓存的方法，向下则勾连全局磁盘管理器完成硬件的读写，其数据结构定义如下。

```
class BufferManager
{
public:
        static const int NBUF = 100;            // 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;                          // 缓存自由队列
        Buf mBuf[NBUF];                         // 缓存控制块数组
        unsigned char Buffer[NBUF][BUFFER_SIZE];// 缓冲区数组
        DiskDriver* diskDriver;                 // 指向磁盘管理器的全局对象
        map<int, Buf*> bufMap;                  // 登记 buf 和其关联的 blkno 的
                                                // map，用于快速查找是否可重用
public:
        BufferManager();
        ~BufferManager();
        Buf* GetBlk(int blkno); // 申请一块缓存，用于读写设备上的字符块 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
        void Bflush();          // 将队列中延迟写的缓存全部输出到磁盘
        void FormatBuffer();    // 格式化所有 Buffer
private:
        void InitBufList();     // 缓存队列和缓存块初始化
        void NotAvail(Buf* bp); // 从缓存队列中摘下指定的缓存控制块 buf
```

```
        void ListAppend(Buf* bp);// 在缓存队列尾加入新 buf
};
```

BufferManager 结构说明：

• 缓存池设置：Buffer[NUBF][BUFFER_SIZE]定义了缓存池的规模和地址，本系统设置 100 个缓存控制块，缓存块的大小为 512 字节，与一个磁盘块大小一致。

• 缓存块与磁盘块的对映关系：利用 STL 容器设置一个 map 结构 bufMap，该结构采用红黑树方法高效实现了键-值的对映关系，我们可以通过物理磁盘块的逻辑号 blk_no 查询对应的 Buf 结构，若没找到，则说明该磁盘块还没有读入内存，反之则得到存储该磁盘块最新内容的缓存块的指针。

• 缓存块的管理：mBuf[NBUF]即 100 个缓存块对应的缓存控制结构，再配合一个自由队列头结点 bFreeList，构成了一条双向循环的 buf 队列，借此管理所有空闲的缓存块，相关具体算法见第四部分详细设计。

磁盘驱动类 DiskDriver

DiskDriver 主要用于对磁盘文件读写，唯一的数据成员是指向磁盘的文件指针，封装的函数用于对磁盘进行直接读写。其构造函数在程序最开始就为文件系统申请一块大小为 8MB 的磁盘文件，名称为"mcwDisk.img"，位于源文件同目录下。

```
class DiskDriver
{
public:
        FILE* diskFilePointer;  //指向磁盘的文件指针

        DiskDriver();
        ~DiskDriver();
        void CreateDiskFile();//创建镜像文件，若之前已有，则会覆盖之。
        bool isMounted();//检查磁盘镜像文件是否存在
        //实现写磁盘文件
        void DiskWrite(const void* ptr, size_t size, int offset = -1,
                       size_t whence = SEEK_SET);
        //实现读磁盘文件
        void DiskRead(void* ptr, size_t size, int offset = -1,
                       size_t whence = SEEK_SET);
};
```

2. 磁盘存储空间管理

外存文件索引类 DiskInode

DiskInode 对象是管理磁盘文件的数据结构，磁盘中每一个文件都唯一地对应一个管理该文件的 DiskInode 的对象。其数据结构声明如下。

```
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;// diskinode 为 64 字节
public:
        DiskInode();
        ~DiskInode();
};
```

- 文件状态管理：d_mode 记录若干标志位的状态，包括文件的读写权限、文件的大小类型、文件是否为目录文件等。在后续对文件的读写、查找的函数中需要利用到该变量所存储的信息。

- **文件索引结构**：参考 Unix V6++的设计，采用同样的三级混合索引方式管理三种不同大小的文件，索引结构如图 3.1 所示。该结构基于 d_addr[10]结构实现，其定义了文件逻辑块号同物理块号的基本索引表。

  - 小型文件只使用 d_addr[0]~d_addr[5]，这 6 项作为直接索引，直接指向磁盘的一个物理盘块。小型文件的大小为 0~6 个数据块。
  - 大型文件在小型文件基础上扩展使用了 d_addr[6]~d_addr[7]，这 2 项具备二级结构，它们指向“一次间接索引表”，索引表是一个完整的数据块，一共含有 512/4=128 项，故可指向 128 个物理盘块。大型文件的大小为 7~（128*2+6）个数据块。
  - 巨型文件在大型文件基础上扩展使用了 d_addr[8]~d_addr[9]，这 2 项具备三级结构，它们指向“二次间接索引表”，二次间接索引表的每一项进而指向一个一级索引表，故每一个二级索引表都可以指向 128 个一级索引表，巨型文件的大小为（128*2+7）~（128*128*2+128*2+6）个数据块。

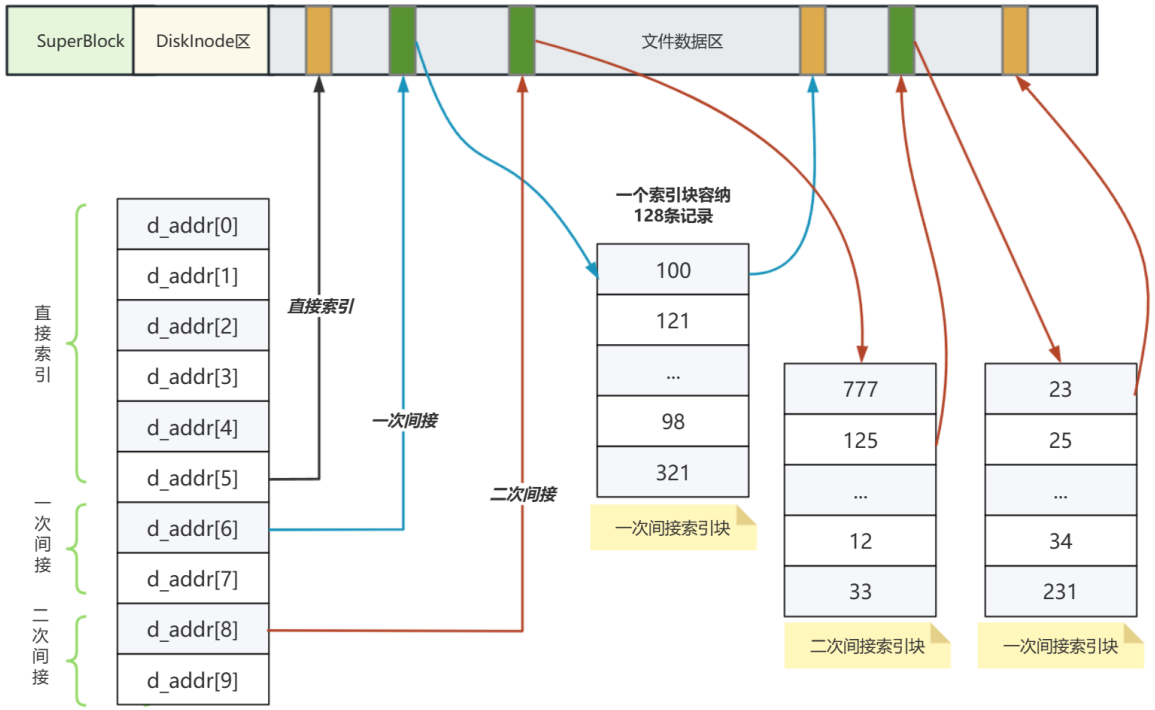


图 3.1 Inode 文件索引结构

文件系统超级块 SuperBlock

超级块主要实现对外存 Inode 区和数据区的管理。数据结构声明如下。

本系统规定超级块直接管理的空闲 Inode 节点数和空闲数据块数均为 100，并采用栈的思想进行管理。对于空闲数据块多余 100 的情形，还采用成组链接法管理，对于直接管理的 Inode 栈空时，采用一次性遍历搜集空闲 Inode 的方法，具体的分配和回收算法在后续说明。

```
class SuperBlock
{
public:
        const static int MAX_FREE = 100;
        const static int MAX_INODE = 100;
        int     s_isize;                //外存 Inode 区占用的盘块数 2~1023，1022 块
        int     s_fsize;                //数据区盘块总数 16384
        int     s_nfree;                //直接管理的空闲盘块数量
        int     s_free[MAX_FREE];       //直接管理的空闲盘块索引表，最多 100
        int     s_ninode;               //直接管理的空闲外存 Inode 数量
        int     s_inode[MAX_INODE];     //直接管理的空闲外存 Inode 索引表，最多 100
        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_time;                 //最近一次更新时间
        int     padding[50];            // 填充使 SuperBlock 块大小等于 1024 字节
        SuperBlock();
```

```
        ~SuperBlock();
};
```

文件系统类 FileSystem

FileSytem 定义了磁盘区域的结构，包括超级块、Inode 区、空闲数据块三个部分，具体结构如图 3.2 所示，8MB 的磁盘被划分为 16384 个大小为 512B 的数据块，超级块占用 0、1 两个盘块，DiskInode 占用 2~1023 共 1022 个盘块，数据区占用剩余的数据块，其中每一个 DiskInode 都是 64B，故一个盘块可以存储 8 个 DiskInode，规定第一个 Inode 区盘块的第一个 64B 存储根目录文件的 DiskInode。

该类封装了维护磁盘文件结构的主要函数，包括磁盘 SuperBlock 的内存镜像的建立、磁盘 SuperBlock 内容的更新、SuperBlock 管理 Inode 区和数据区的分配与回收算法的实现等。

```
class FileSystem
{
public:
                                /* 磁盘的空间管理 */
static const int BLOCK_SIZE = 512;              //Block 块大小，单位字节
                          ......  ......  ......
                                /* 数据成员和函数 */
public:
        DiskDriver* diskDriver;
        SuperBlock* superBlock;
        BufferManager* bufManager;

        FileSystem();
        ~FileSystem();
        void FormatSuperBlock();//格式化 SuperBlock
        void FormatDevice();    //格式化整个文件系统
        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 SuperBlock 中去
        /* 磁盘 Inode 节点的分配与回收算法设计与实现 */
        Inode* IAlloc(); //在存储设备上分配一个空闲外存 INode，一般用于创建新的文件
        void IFree(int number); //释放编号为 number 的外存 INode，一般用于删除文件
        Buf* 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
};
```

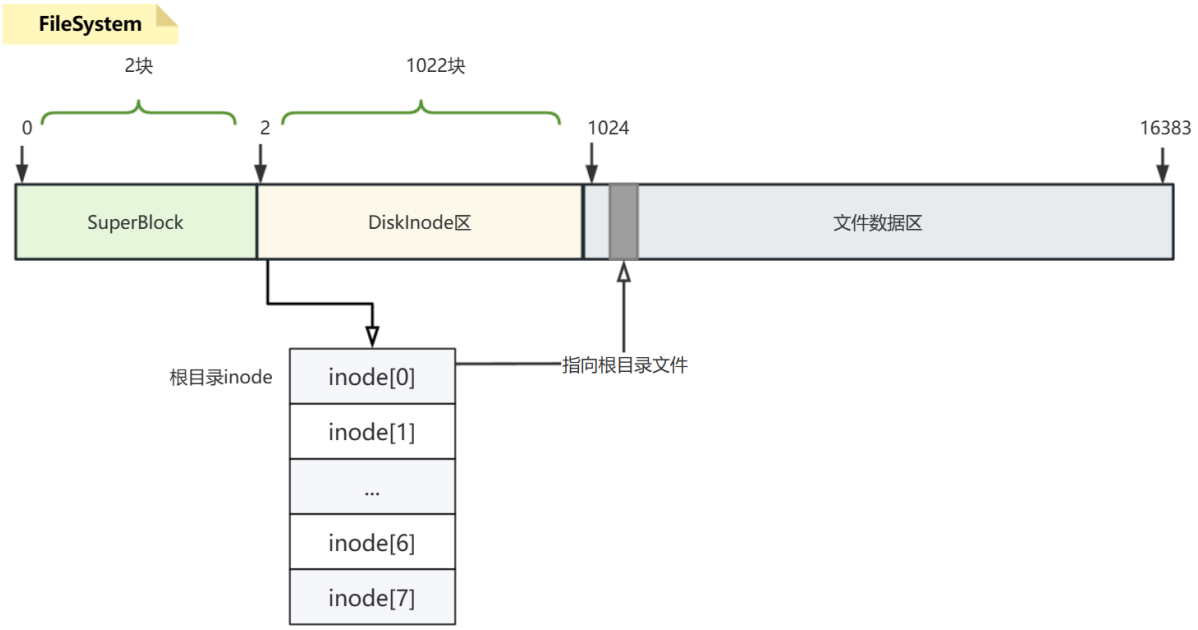


图 3.2 文件系统目录结构

3. 文件系统实施部分

内存文件索引类 Inode

内存 Inode 结构是外存 inode 在内存中的映像，目的是减少文件 IO 次数。其数据结构如下所示，与 DiskInode 基本一致故不加赘述，值得说明的是其增加了 i_count 和 i_number 成员，前者指出当前引用该 Inode 的实例数目，后者指明该 Inode 对于的外存 Inode 的编号，“-1”表示没有对应的外存 Inode，也即当前 Inode 空闲。内存 inode 需要考虑何时将变化同步到 DiskInode，以及如何通过文件逻辑块号转换成对应的物理盘块号，后续将展开说明。

```
class Inode
{
public:
        enum INodeFlag          // i_flag 中标志位
        {
                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
        };
                        /* 静态局部变量 */
        static const unsigned int IALLOC = 0x8000;      // 文件被使用

                        ......  ......  ......

                        /* 数据成员 */
        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 ps 若等于 0 表示空闲
```

```
        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;// 外存 inode 区中的编号。从 0 开始计数,-1 表示空闲
        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();
        //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);
        //更新外存 Inode 的最后的访问时间、修改时间
        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，但不清与外存 inode 连接的标志 flag\count\number:
        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);
};
```

内存文件索引节点表 InodeTable

InodeTable 主要定义了大小为 100 的内存 Inode 索引表，统一管理所有的内存 Inode 节点。其数据结构声明如下所示。

```
class InodeTable
{
public:
        static const int MAX_INODE = 100;       // 内存 Inode 的数量
        Inode m_Inode[MAX_INODE];               // 内存 Inode 数组
        FileSystem* m_FileSystem;               // 对全局对象 g_FileSystem 的引用
public:
        InodeTable();
        ~InodeTable();
        void Reset();
        Inode* IGet(int inumber);/* 根据外存 Inode 编号获取对应 Inode。
        void IPut(Inode* pNode);
        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
        Inode* GetFreeInode();  //在内存 Inode 表中寻找一个空闲的内存 Inode
```

```
};
```

文件打开类 File

File 类用于记录进程打开文件的读写权限、文件指针的偏移位置等信息，相比 Inode 存储文件的大小内容索引等静态信息，File 结构存储了打开文件后的动态信息。其数据结构声明如下。

File 通过 f_inode 勾连对应的 Inode 节点，能通过文件句柄 fd 被选中，具体在文件打开结构中说明。另外，其文件读写指针 f_offset 是在读写文件的系统调用时，通过“IO 参数类”传递给全局 User 对象的，在文件读写的实现中还会具体说明。

```
class File
{
public:
        enum FileFlags
        {
                FREAD = 0x1,                    /* 读请求类型 */
                FWRITE = 0x2,                   /* 写请求类型 */
        };
        unsigned int    f_flag;                 // 对打开文件的读、写操作要求
        int             f_count;                // 当前引用该文件控制块的进程数
        Inode*          f_inode;                // 指向打开文件的内存 Inode 指针
        int             f_offset;               // 文件读写位置指针
public:
        File();
        ~File();
        void Reset();
};
```

打开文件管理类 OpenFileTable

OpenFileTable 负责对系统中所有打开的文件进行管理，主要成员是 m_File[100]结构，表示本系统最多支持同时打开 100 份文件并生成对应的文件打开结构。可见 OpenFileTable 记载了当前系统中所有进程对文件进行读写操作的状态信息。其数据结构定义如下。

```
class OpenFileTable
{
public:
        static const int MAX_FILES = 100;// 打开文件控制块 File 结构的数量
        File m_File[MAX_FILES];          // 系统打开文件表，为所有进程共享
public:
        OpenFileTable();
        ~OpenFileTable();
        void Reset();
```

```
        File* FAlloc(); //在系统打开文件表中分配一个空闲的 File 结构
        void CloseF(File* pFile);//对打开文件控制块 File 结构的引用计数 f_count 减
1，若引用计数 f_count 为 0，则释放 File 结构。
};
```

进程打开文件类 OpenFiles

在本系统中由于只有一个进程，则实际上其定义和 OpenFileTable 存在功能上的冗余，但为了可拓展性考虑还是保留了该结构，其主要成员 ProcessOpenFileTable[MAX_FILES]记录了当前进程打开文件的 File 指针，通过文件句柄 fd 索引对应的 File 结构。其数据结构如下所示。

```
class OpenFiles
{
public:
        static const int MAX_FILES = 100;       /* 进程允许打开的最大文件数 */
private:
        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 */
public:
        OpenFiles();
        ~OpenFiles();
        void Reset();
        //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
        int AllocFreeSlot();
        //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
        File* GetF(int fd);
        //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
        void SetF(int fd, File* pFile);
};
```

目录结构类 DirectoryEntry

本系统采用树形带交叉勾连的目录结构。目录结构由若干目录文件组成，每个目录文件又由一些列目录项组成。目录结构的基本单位目录项由 DirectoryEntry 类对象描述，包括 28 字节的名字部分（文件的外部标识），以及 4 字节的对应文件在磁盘上的外存索引节点号（文件的内部标识）。这样就能利用文件名通过目录检索程序找到对应的文件数据了。

此外，由于一个目录项大小为固定的 32 字节，故一个数据块能存储 16 个目录项。且每个文件系统都应该有一个根目录文件，其它所有文件都是在该根目录下的。本系统设定根目录的索引结点是 DiskInode 区的第一个 DiskInode 结点。

```
class DirectoryEntry
{
public:
```

```
        static const int DIRSIZ = 28;
        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
public:
        DirectoryEntry() ;
        ~DirectoryEntry();
};
```

一个简单的目录索引结构如图 3.3 所示：

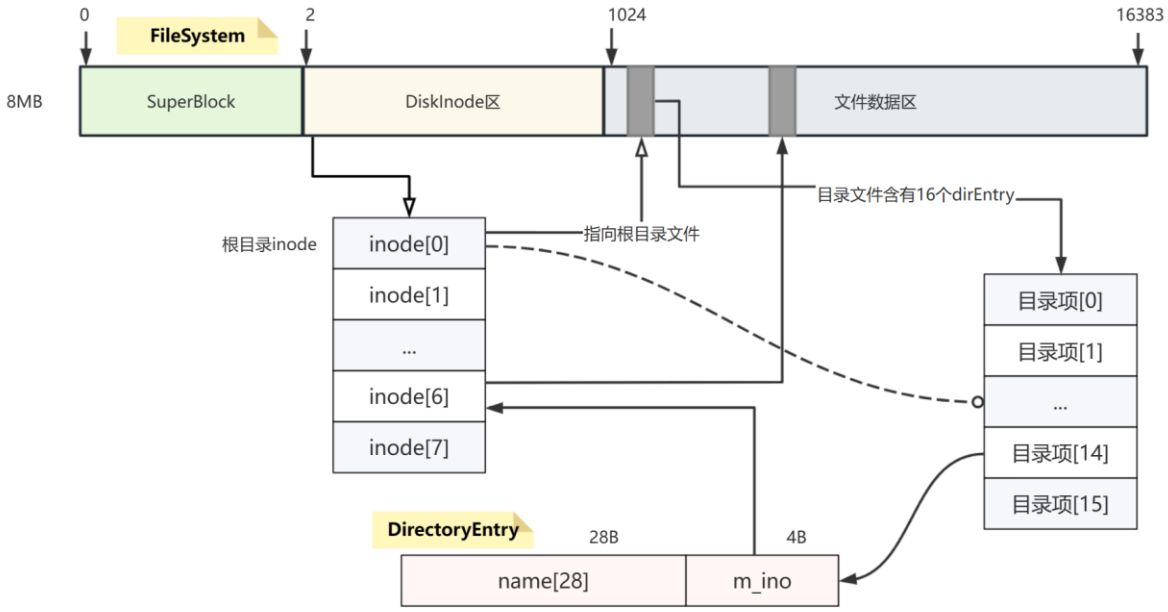


图 3.3 目录索引结构

系统调用类 SysCall

SysCall 实现系统调用模块，为各文件操作提供系统接口，并且针对每条指令执行具体的操作。实际上其作用与 Unix V6++中 FileManager 相同，因为后者在处理文件部分的系统调用时，就是调用 FileManager 的接口函数实现的。SysCall 的数据结构定义如下所示。

```
class SysCall
{
public:
        enum DirSearchMode      //目录搜索模式,NameI()中函数用到
        {
                OPEN = 0,               //以打开文件方式搜索
                CREATE = 1,             //以新建文件方式搜索
                DELETE = 2              //以删除文件方式搜索
        };
        //数据成员
```

```
        Inode* rootDirInode;    //根目录内存 INODE 结点指针
        FileSystem* fileSystem; //引用全局 g_fileSystem
        InodeTable* inodeTable; //引用全局 g_inodeTable
        OpenFileTable* openFileTable;//引用全局 g_openFileTable
        SysCall();
        ~SysCall();
        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 DirSearchMode 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
};
```

3. 用户界面部分

用户类 User

User 结构在本项目中有唯一的全局对象 g_user，通过该对象能为用户提供文件操作接口，并能利用 arg[5]和 ar0[5]与系统调用模块交互，同时 User 结构还记载了当前目录和父目录、当前用户的进程打开文件描述符表对象、读写文件的相关参数（IOParam）等。其数据结构定义如下。

其中 IOParameter 类主要用于记录文件 I/O 的参数，包括文件读写时用到的读写偏移量、字节数和目标区域首地址参数。

```
    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 {        // 错误码
                U_NOERROR = 0,
                ......  ......  ......
```

```
        };
        Inode* curDirIP;                //指向当前目录的 Inode 指针
        Inode* parDirIP;                //指向父目录的 Inode 指针
        DirectoryEntry curDirEnt;       //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
        size_t ar0[5];                  //指向核心栈现场保护区中 EAX 寄存器
        ErrorCode errorCode;            //存放错误码
        OpenFiles ofiles;               //当前用户的进程打开文件描述符表对象
        IOParameter IOParam;            //记录当前读、写文件的偏移量等参数
        SysCall* sysCall;
        string ls;                      //存放目录项
public:
        User();
        ~User();
        void u_Ls();
        ......  ......  ......

};
```

3.3 模块调用关系

各个类对象相互勾连调用形成了完整而高效的文件系统，下面将展示主要类对象之间的调用关系。

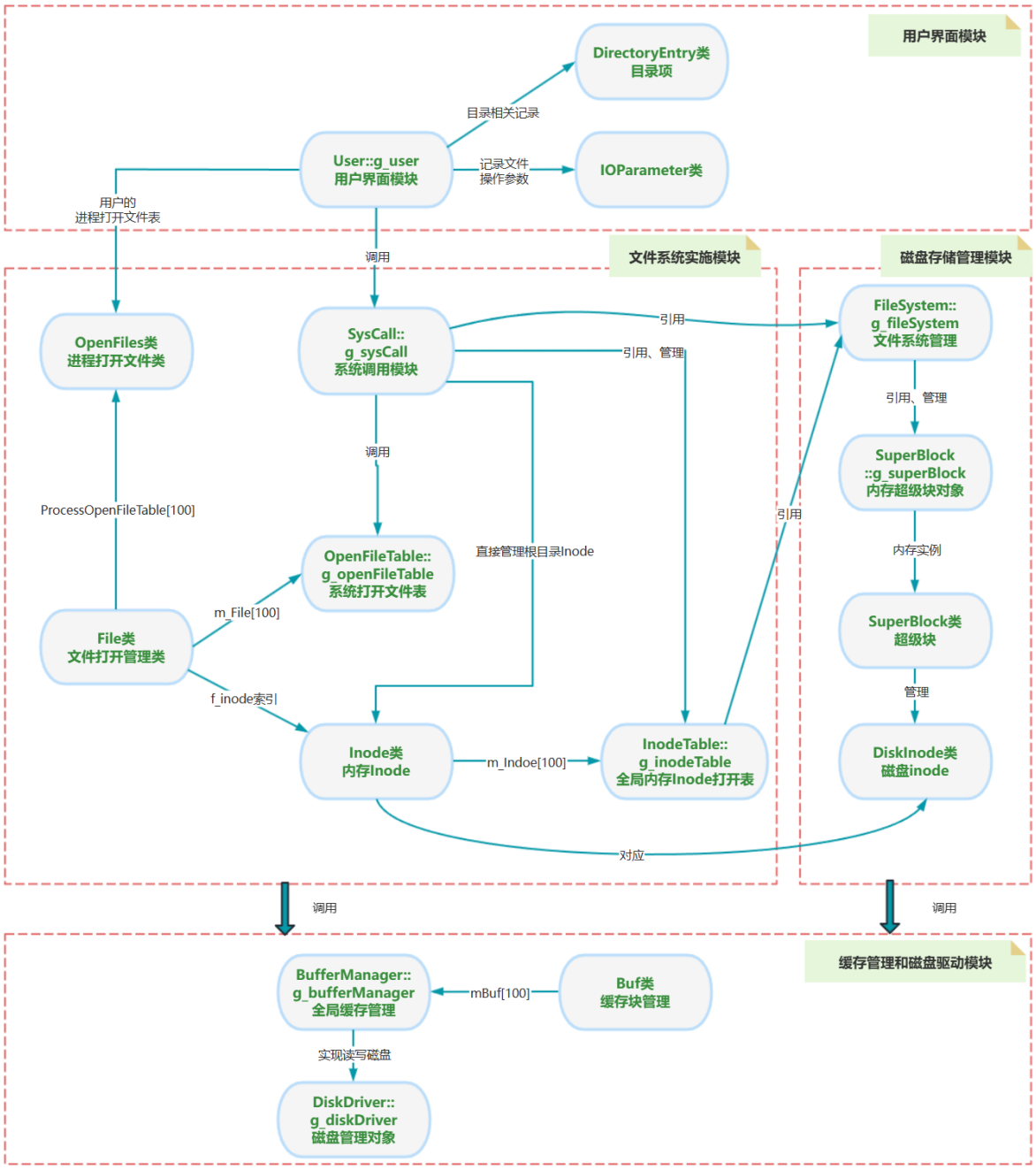


图 3.4 模块调用关系

通过图 3.4 模块调用简图，我们可以进一步看到各模块在整个系统的承担的职责和作用。

•用户的命令通过 User 对象接受，并调用其相应的系统调用函数，通过 arg[5]传递参数，SysCall 对象指向具体的系统调用，在必要时将返回值存在 ar0[5]中，使得上层的 User 对象能根据返回值指向相应的逻辑操作或在 cmd 中产生提示用户的语句。可以认为 User 对象就是为用户提供交互界面

而存在的。

• SysCall 对象根据 User 结构的提供的参数执行命令，其起到了衔接用户对文件的操作和对文件系统内部数据结构相应处理的作用，管理着本类 Unix 系统的对外接口。此外，它也是文件系统实施的核心部分，它将勾连全局的系统文件打开表、内存 Inode 结点表，同时它也实现了文件实施部分和磁盘存储管理空间的勾连。

• 磁盘存储空间主要由 FlieSystem 对象管理，FileSystem 统筹管理超级块和磁盘 INode 区的分配和回收等事务。

• OpenFileTabel、File、OpenFiles、InodeTable 共同维护着内存文件打开结构，同时 InodeTable 还通过 FileSystem 对象联系着磁盘的 Inode 区，保证磁盘内容的即使更新。

在分析 Unix 文件系统时，学生又一次感叹 Unix 系统设计的精妙，各个模块各司其职而相互关联，结构清晰且定位准确，值得我不断体会与学习！

3.4 主程序流程

主程序逻辑相对简单，不加赘述，如图 3.5 所示。

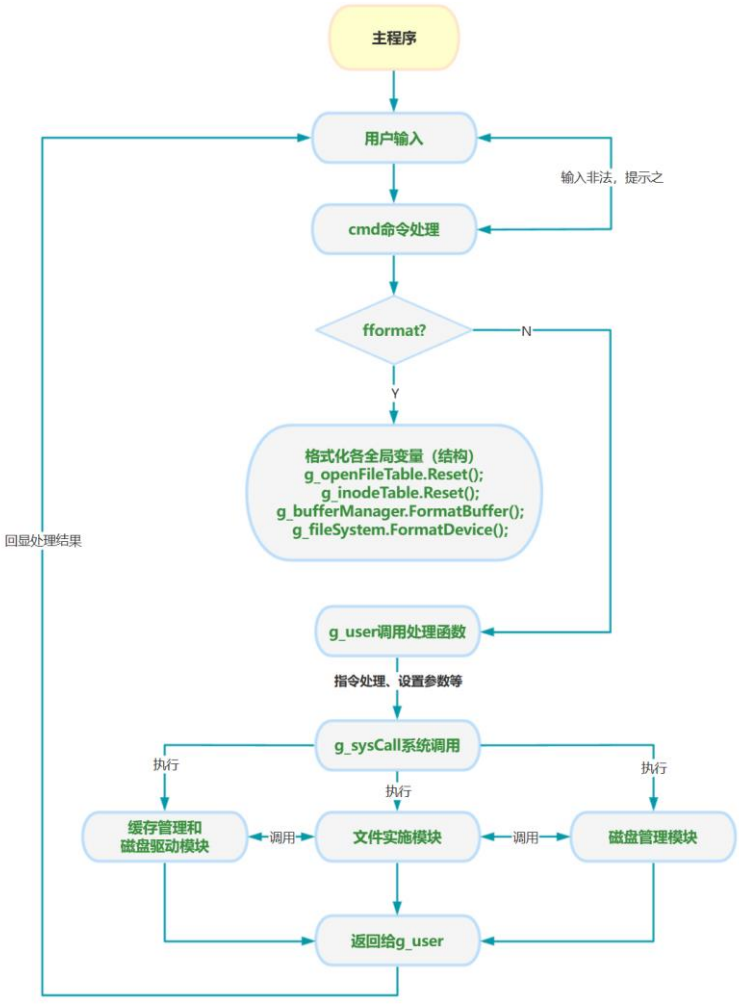


图 3.5 主程序流程图

4 详细设计

4.1 文件系统结构与算法

以下关于文件系统基本结构的说明在概要设计中已做说明，在此不做赘述。

- SuperBlco 及 Inode 区的位置大小
- Inode 节点数据的定义及三级混合索引结构的说明
- SuperBlock 数据结构的定义

下面重点介绍文件系统中对 DiskInode 节点及文件数据区管理的相关算法。

4.1.1 DiskInode 节点管理

对于外存索引节点的管理，本系统利用 SuperBlock 采用栈方式管理 100 个空闲的 DiskInode。不选择管理所有空闲节点是因为 Inode 区的节点数量很多，且空闲数量常常动态变化，这导致组织和管理所有的空闲节点的开销是很大的。所以我们选择最多管理 100 个空闲 DiskInode，重要的几个变量如下：

- s_inode[100]：记录当前超级块直接管理的空闲 inode 的编号。
- s_ninode：记录当前超级块直接管理的空闲 inode 的盘块数量。

基于这两个变量可以采用栈方式管理空闲 inode，具体逻辑如下：

- 分配 DiskInode

1、若 s_ninode！=0，即 s_inode 中还可以分配空闲 inode，则将 s_inode[--s_inode]所指向的 inode 分配出去。

2、若 s_ninode==0，即 s_inode 中没有可以分配的空闲 inode，这时我们遍历 inode 区，查找 100 个空闲的 DiskInode，登记到 s_inode 中，完成 s_inode 的填充后，就可以将栈顶的 DiskInode 分配出去。

上述分配 DiskInode 算法的具体实现由 FileSystem::IAlloc()完成，其算法流程图如图 4.1.1 所示。

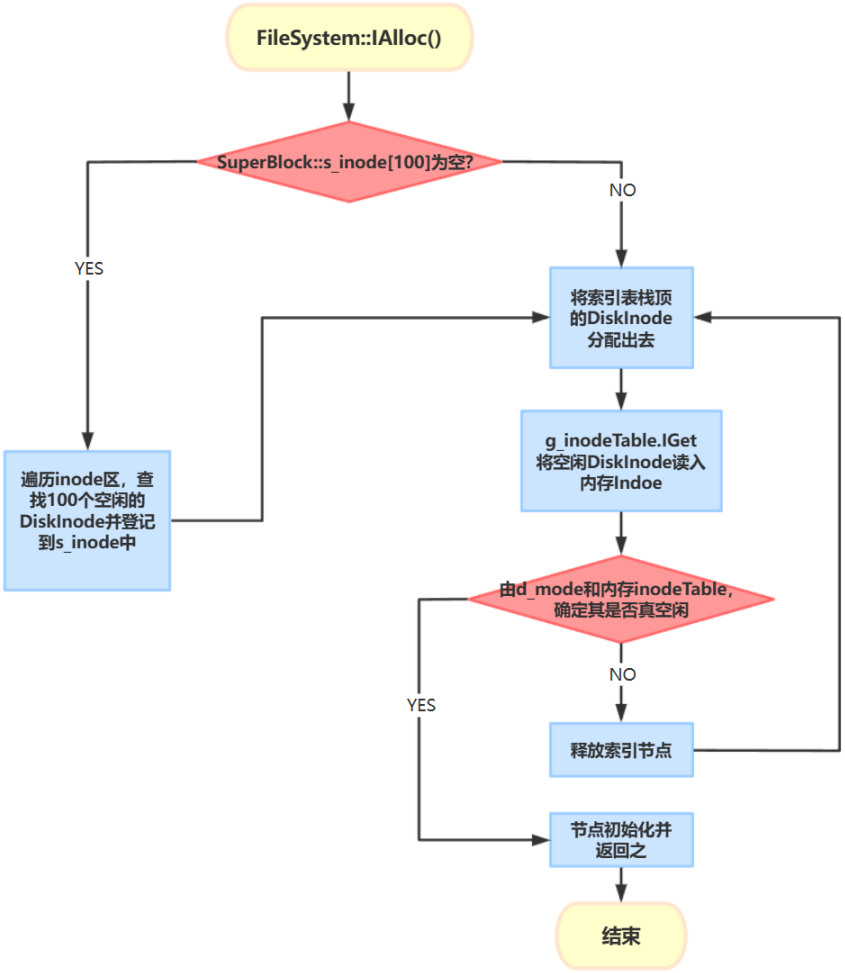


图 4.1.1 分配 DiskInode 算法流程图

当 s_inode 中没有可以分配的空闲 inode，程序会遍历 inode 区查找空闲的 inode，这里判断 inode 是否被占用采用的方法是判断 i_mode 标志位是否为 0，若不为 0 说明其被占用。但反之不足以说明其是否空闲，因为我们搜索的外存 inode 区的内容不一定是当前最新的，有可能内存 inode 已经更新而没有写回到磁盘，所以在判断 DiskInode 是否空闲时，还需要去查找当前内存的 InodeTabel，找是否由对映项，若没有登记到内存且 i_mode 为 0，才能认定为是空闲 inode。

- 释放 DiskInode

当某个 DiskInode 不再被占用时（一般在文件关闭时）我们需要回收该 DiskInode。回收流程比较简单，首先考察超级块的 s_inode 表是否已满，未满则将该 DiskInode 编号登记在 s_inode 栈顶中，否则不登记。

实现该算法的函数是 FiSystem::IFree()，其流程如图 4.1.2 所示。

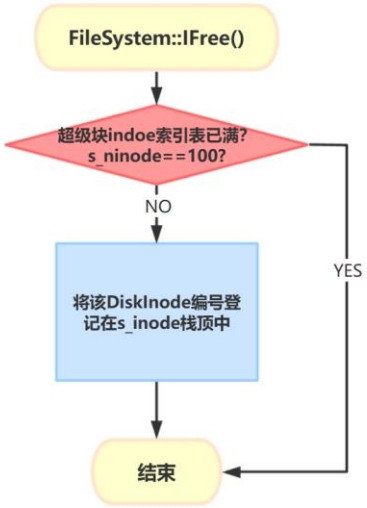


图 4.1.2 释放 DiskInode 算法流程图

4.1.2 文件数据区管理

对于空闲盘块的管理，同样利用 SuperBlock 直接管理最多 100 个空闲数据块，对于超过 100 个空闲数据块的情形，不同于磁盘 Inode 的管理方式，Unix 中引入了分组链接索引的方式对所有的数据块进行管理。

- 分组链接方式说明

首先回顾概要说明中提到的两个重要数据成员：

• s_nfree：直接管理的空闲盘块数量。

• s_free[100]：直接管理的空闲盘块索引表。

所谓分组，是将所有空闲块按照每 100 个构成一组的方式进行管理，最后一组由超级块的 s_nfree 和 s_free[100]构成的栈结构直接管理，第一组只有 99 个空闲块（即除了第一组和超级块直接管理的最后一组，中间组都是 100 个空闲块）。每一组的第一块都需要担任特殊的工作：记录上一组的 s_nfree 和 s_free[100]，即第一块的前 404 字节记载着上一分组的数据块数量和它们的物理块号。第二组的第一块的 s_free[0]等于 0，这也标志这空闲盘块链的结束。

借助该方式，我们能以分组的方式嵌套地链接庞多的数据块，这就是分组链接方式的核心思想。图 4.1.3 展示了一个分组链接的情形。

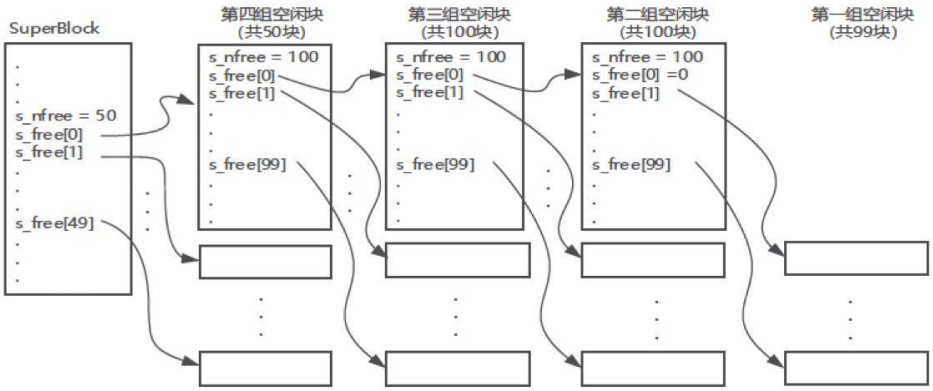


图 4.1.3 分组链接索引方式 [1]

- 空闲盘块分配算法

基于前述分组链接思想的介绍，我们很容易能理解如何分配空闲盘块。分配算法总是分配 SuperBlock 直接管理的分组的最后一块，当其直接管理的一组分配完了，即分配到该分组的第一块时，就将第一块的前 404B 内容有序读入 s_nfree、s_free[100]，再分配出第一块，这样超级块直接管理的就是下一装有 100 空闲盘块的新分组了。算法流程如图 4.1.4 所示。

相比 Inode 的管理，这样做免去了搜索的过程，对于数量级更多的数据盘块区，学生认为成组链接的优势就能在省出繁杂遍历带来的时间开销这一方面体现出来。

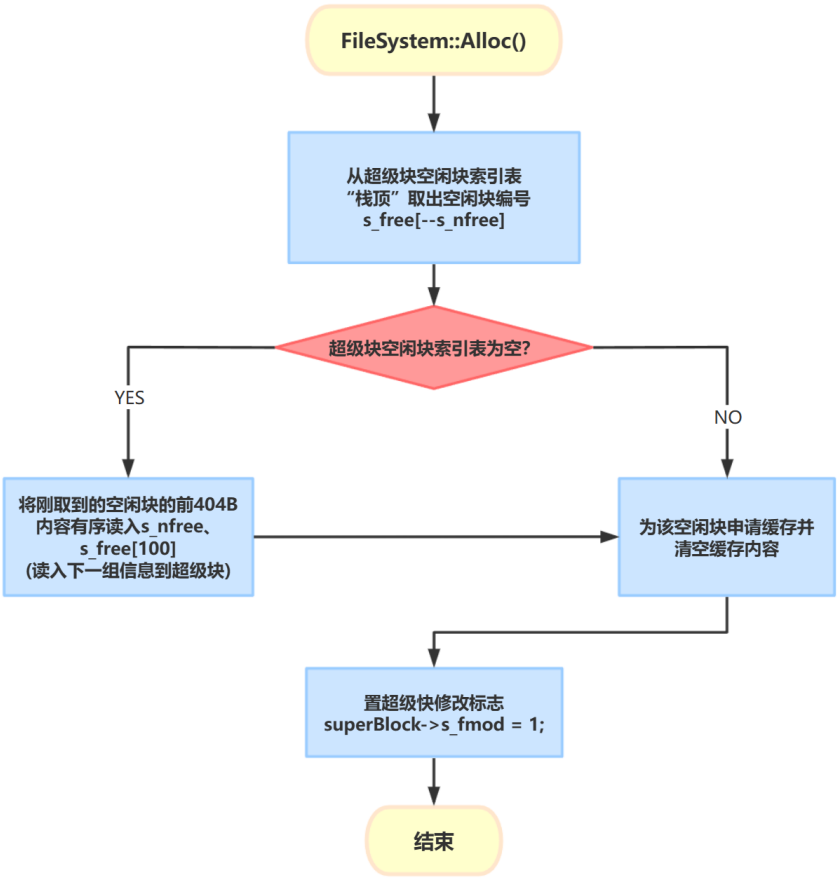


图 4.1.4 空闲盘块分配算法

- 空闲盘块的释放算法

释放空闲盘块是压栈的过程，如果栈未满则将空闲盘块索引表 1 第一个未被占用的项登记为正在释放的盘块号，反之若已经满了，则应该新建一个分组：将超级块的 s_nfree、s_free[100]写入该正在释放的盘块，将该正在释放的盘块号写入 s_free[0]，s_nfree 置为 1。

该算法由 FileSystem::Free()实现，算法流程如图 4.1.5 所示。

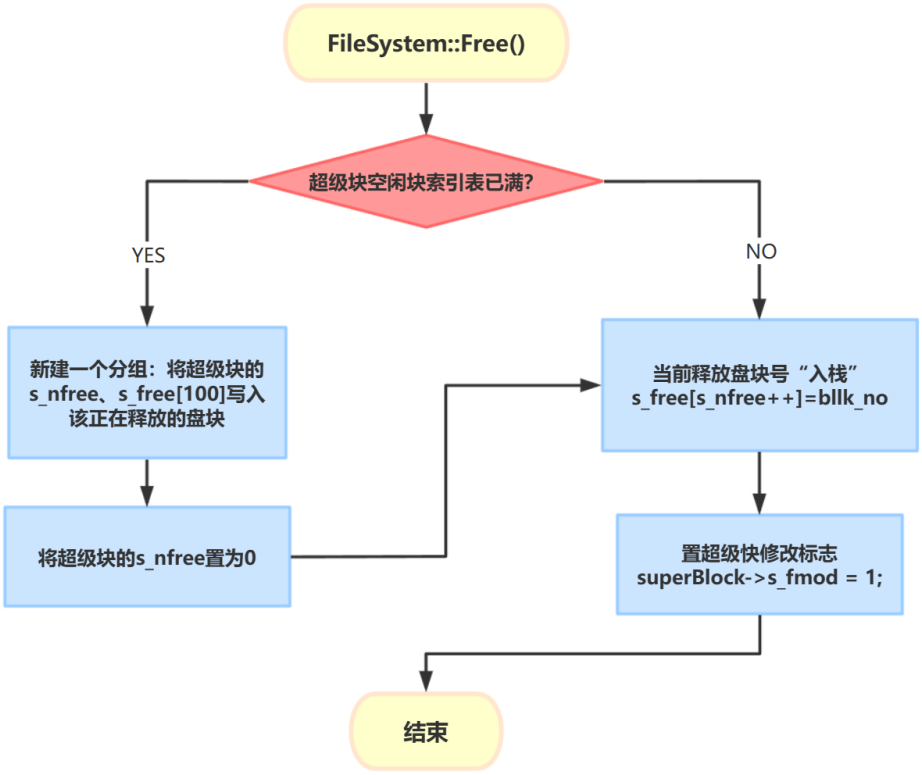


图 4.1.5 空闲盘块释放算法

4.2 目录结构和算法

本系统的目录结构声明与设计在概要设计中已做介绍，这里不再赘述。下面将重点说明目录的检索算法和其目录结构增、删、改的设计与实现。

4.2.1 目录的检索算法

目录项对外是通过文件路径名称确定的，对内部结构是通过其关联的 DiskINode 的编号确定的。目录检索算法的目的就是要实现二者的转换。该过程是由 SysCall::NameI 函数实现的，其主要的算法流程如图 4.2.1 所示。

对该算法做如下几点说明：

- 目录搜索的起始位置是当前目录，这由全局 User 对象 g_user 记录在其数据成员 DirectoryEntry curDirEnt 中，但如果输入的文件路径是从‘/’开始的，我们规定这表示从根目录开始检索。

- 检索过程入流程图中所展示的，逐级地搜索每一级路径分量对应的下一级目录的索引节点号，且对于不是最后一个分量的所有分量都都根据标志位 i_mode 判断是否为目录文件，只有不是目录文件则显然是非法的路径。对于目录文件，我们将相应目录文件中的目录项逐个顺序地匹配，如果与目标分量(记录在 g_user 的 dirp 结构中)相同，则当前分量搜索成功，接下来记录下一级路径分量的 DiskInode 编号，并获取其内存镜像为下一级目录搜索做准备。重复该搜索过程直到找到或查找失败。

- IName 函数不只是完成搜索的工作，它还能根据上层在形参中传达的需求，按需执行不同模式下的查找工作，且对于不同的查找模式，其对搜索结果的处理也不同。传入的参数 mode 是 `enum DirSearchMode` 类型，有 OPEN、CREATE、DELETE 三种模式。OPEN 表示打开文件时进行的搜索，CREATE 表示创建文件时搜索，需要在最后一级目录下创建新文件，DELETE 是删除一条文件路径时进行的搜索，找到后要取消该文件在目录中的目录项。

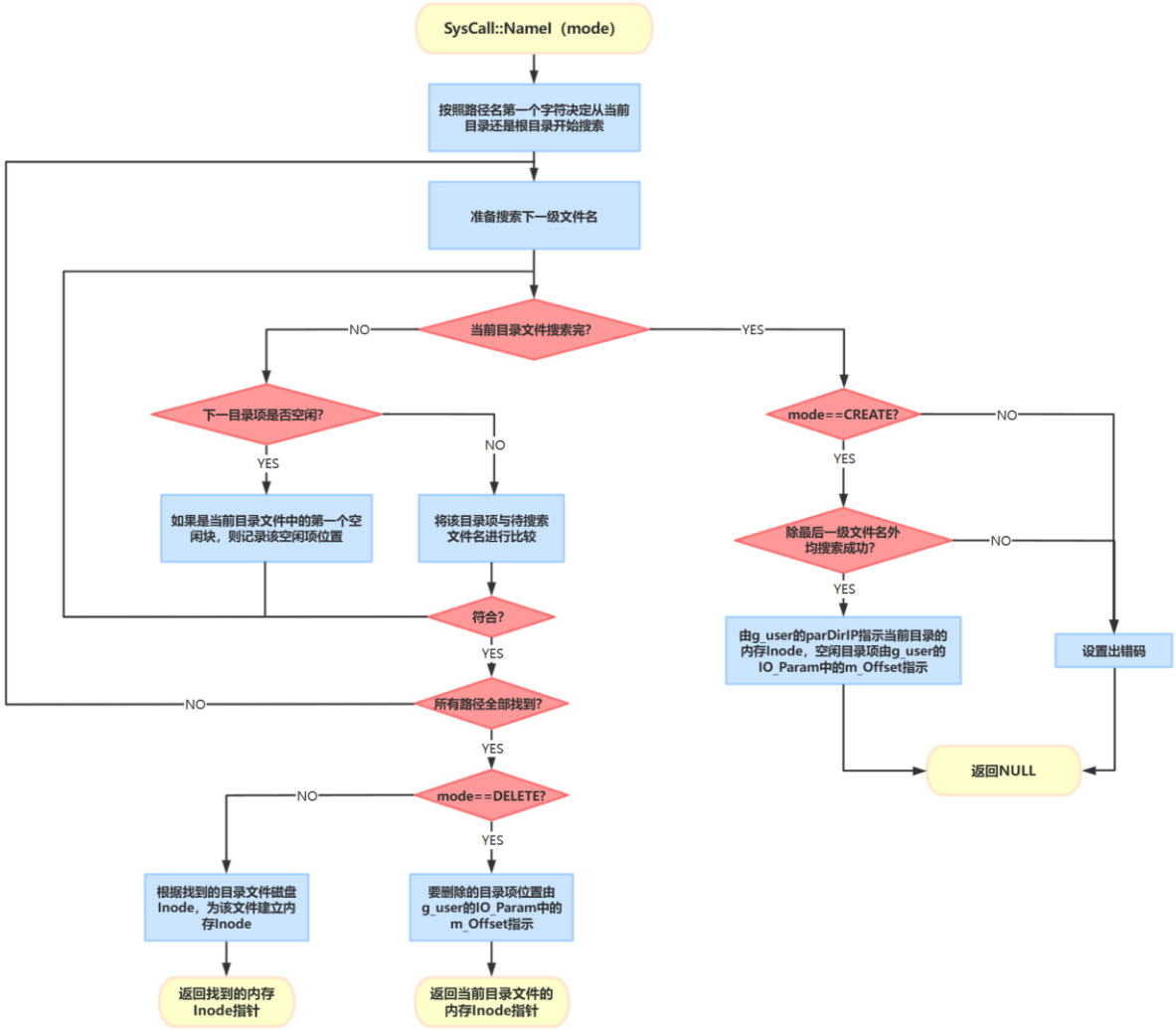


图 4.2.1 目录检索算法

4.2.2 目录修改算法

- 目录项的登记

目录项的登记发生在创建文件时，本系统中由函数 SysCall::Creat()完成，算法流程如图 4.2.2 所示。

Creat 函数首先会调用 NameI 函数，NameI 算法实现的是根据文件路径查找对应 Inode 结构的功能，在创建文件时，我们将参数 `DirSearchMode` mode 置为 CREATE，如果只是最后一级目录查找失败，这说明符合创建文件的条件。

此时 NameI 返回 NULL 表示没找到目标文件，但是不会设置出错码，同时 User::parDirIP 指向应该填入新的目录项的目录文件的内存 Inode，g_user.IOParam.m_Offset 指向该目录文件的第一个空白项目的位置。

随后 Creat 函数根据 NameI 返回的 NULL，以及没有设置出错码这两个条件，会分支到进行如下动作：为文件申请一个新的磁盘 Inode，若成功申请到，则将该创建的文件名和 DiskInode 号组成目录项，填入 g_user.IOParam.m_Offset 所指的目录项位置。至此，实现了目录项的登记。

之后 Creat 函数还会为该新创建的文件建立内存文件打开结构。

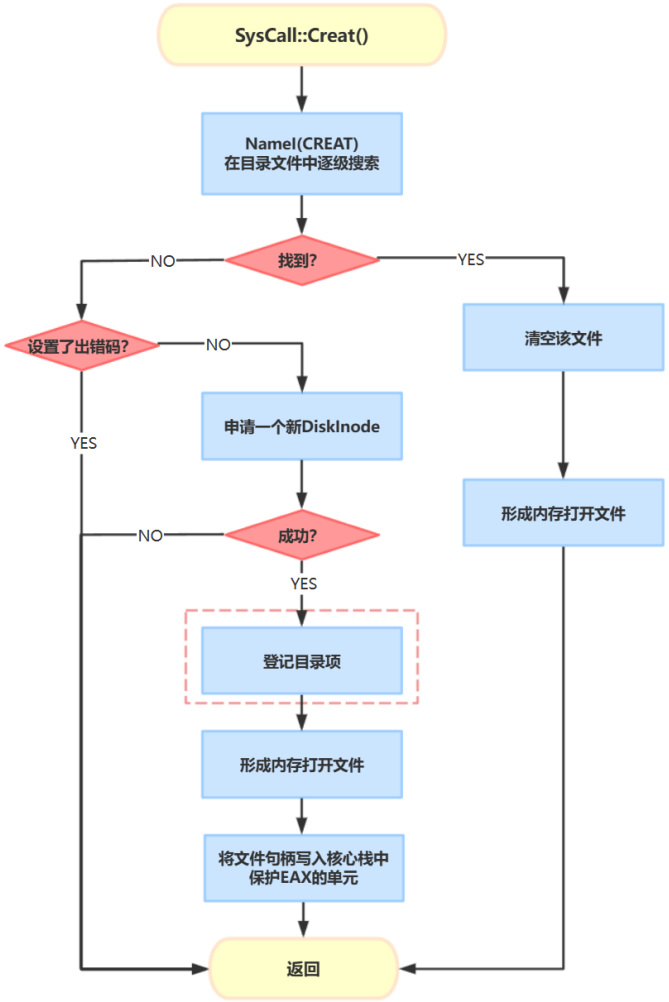


图 4.2.2 文件创建函数流程

- 目录项的清除

目录项的清除发生在文件取消的过程，本系统中由函数 SysCall::UnLink()完成，其算法流程图如图 4.2.3 所示。

首先同样是通过 NameI 函数搜索目录，但 mode 设置为 DELETE。如果顺利找到了要删除的文件，则 NameI 返回该文件的 DiskInode 的指针，且将要删除的目录项的偏移地址保存在 g_user.IOParam.m_Offset 中，其父目录存在 g_user.curDirEnt 中。

Unlink()接受到目标文件的指针后，首先根据 g_user.curDirEnt.m_ino 得到目标文件的父目录文件的内存 Inode，然后根据 g_user.IOParam.m_Offset 定位到目标文件的目录项目所在的位置，这时就可以清除目录项了。

此后，还要根据被取消的文件的 inode::i_nlink 判断是否要彻底删除其磁盘文件。

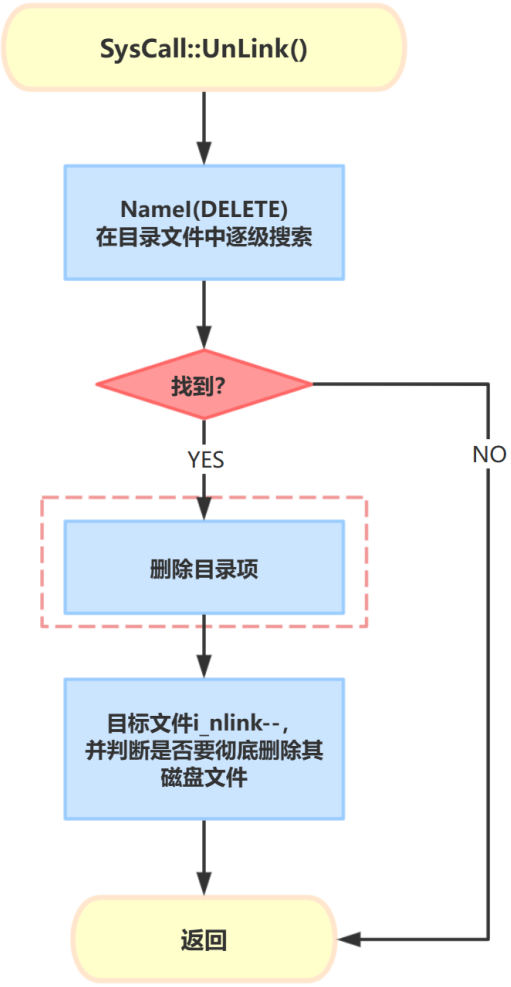


图 4.2.3 文件删除函数流程

4.3 文件打开结构说明

4.3.1 文件打开结构的设计

当进程打开一个文件时，我们要创建其内存的打开结构以减少 IO 次数。文件打开结构的构建和

维护由文件实施模块实现，主要包括以下几个结构：进程打开文件表、系统打开文件表、内存InodeTable 等。图 4.3.1 展示了本系统的文件打开结构图像。

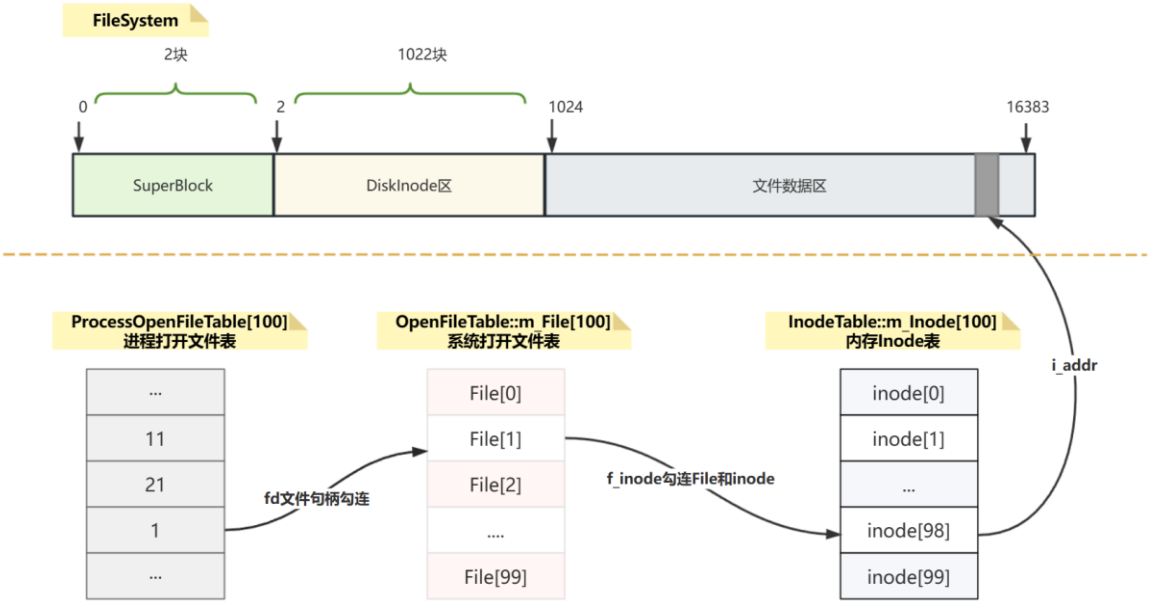


图 4.3.1 文件打开结构图像

由于本系统只有一个进程，所以进程打开文件表全局也只有一个(全局 User::g_user 中定义)，故其大小设置可以与系统打开文件表相等，均设置为 100。内存 Inode 表最多也能管理 100 个文件的 Inode。具体而言，文件打开结构做到了：

- 在内存 Inode 表中找到或分配一个内存 Inode
- 在系统打开文件表中分配一个 File 控制块
- 在进程打开文件表中分配一项作为文件句柄 fd
- 建立了三者的勾连关系，能让上层直接通过文件句柄 fd 找到对应的 File 结构，进而找到内存 Inode 结构，并最终定位到磁盘数据块。

4.3.2 内存 Inode 节点的分配与回收

所有内存 Inode 节点由全局的 InodeTable::g_inodeTable 管理，Inode 节点的分配和回收分别由其成员函数 `Inode* IGet(int inumber)`和 `void IPut(Inode* pNode)`;完成。

Inode 节点的分配算法

IGet 函数提供了通过磁盘 Inode 编号获取对应的内存 Inode 指针的方法，更准确的说，通过 IGet 函数，上层能够得到任何 DiskInode 的内存映像 Inode。

- 首先，通过 IsLoaded 函数去查找内存 Inode 表，确定目标 DiskInode 是否已经由内存映像 Inode，若有则增加 Inode 对象的引用数 i_count，返回该对象的指针。

- 反之没有找到内存映像 Inode，则通过 GetFreeInode 函数在内存 Inode 表中获取一个空闲 Inode，若获取成功则修改该 Inode 的相应标志位，将该外存 Inode 读入到某缓存块，最后将该缓存块的内容拷贝到新分配的内存 Inode 中，最后释放缓存、返回该 Inode 的指针。

Inode 节点的回收算法

在关闭文件时会通过调用IPut函数实现Inode的回收。IPut主要做的操作：内存i节点计数i_count--，若为0，释放内存Inode节点、若有改动则还需写回磁盘。

另外，文件途径的所有目录文件，搜索经过后都会Iput其内存i节点。路径名的倒数第2个路径分量一定是目录文件，如果是在其中创建新文件、或是删除一个已有文件；再如果是在其中创建删除子目录。那么必须将这个目录文件所对应的内存 i节点写回磁盘。这个目录文件无论是否经历过更改，我们都必须将它的Inode节点写回磁盘。

该算法相对简单，主要是修改标志位i_count，对于i_count==1且i_nlink==0的Inode，说明已经没有目录路径通往该文件，即用户不再需要使用之，此时采取IFree释放Inode。需要注意同步更新外存Inode的信息。

4.3.3 文件打开过程

文件的打开过程需要创建好文件的内存打开结构，该过程牵扯到多模块的调用关系，下面说明该过程的函数调用关系。

- 全局 User 对象 g_user 提供给用户的文件打开接口是 `void u_Open(string fileName)`，该函数首先会通过 `User::checkPathName(string path)`检查用户的 fileName 是否合法，如果合法，则会把文件路径存在 User::g_user.dirp 中。

然后以可读可写的方式(设置 arg[1])的方式调用 `SysCall::Open()`，若成功打开则输出打开成功和文件句柄的提示语并返回，否则在 `User::checkError()`中根据 g_user.errorCode 输出错误类型并返回。

- SysCall::Open()的功能是根据 User::g_user.dirp 中的路径打开文件，并建立文件的打开结构。首先调用 `NameI(SysCall::OPEN)`函数查看是否有该文件，若找到则，NameI 会在内存创建该文件的 Inode 结构，随后将该文件的 Inode 指针传入 `SysCall::Open1(Inode* pINode, int trf)`函数，其中 trf 设置为 0。这表示以打开文件的目的调用 open1 函数，该函数会通过调用函数 `OpenFileTable::FAlloc()`为 Inode 节点创建文件打开结构。

- FAlloc 函数为文件创建内存打开结构的算法流程为：1、在进程打开文件描述符表中获取一个空闲项；2、在打开文件表中找空闲项目，3、完成与进程打开文件表的勾连

- 如果文件打开结构顺利创建，则 Open1 正常返回，SysCall::Open()正常返回，User::u_Open()正常返回。至此，描述了文件顺利打开的过程。

上述过程的调用关系图如图 4.3.2 所示。

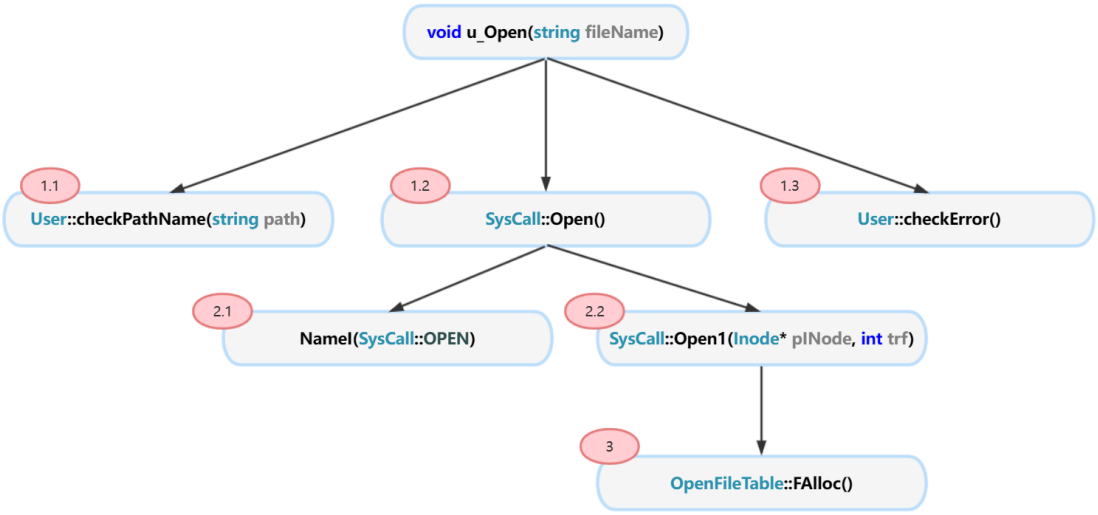


图 4.3.2 文件打开的函数调用关系

4.3.4 文件索引和数据块的映射转换

概要设计中已经详细介绍了 d_addr[10]结构的设计和定义，下面介绍如何将 inode 的文件逻辑块号 lbn 转换成磁盘上的物理块号。这个过程是由函数 int Bmap(int lbn)实现的，其算法思路如图 4.3.3 所示。

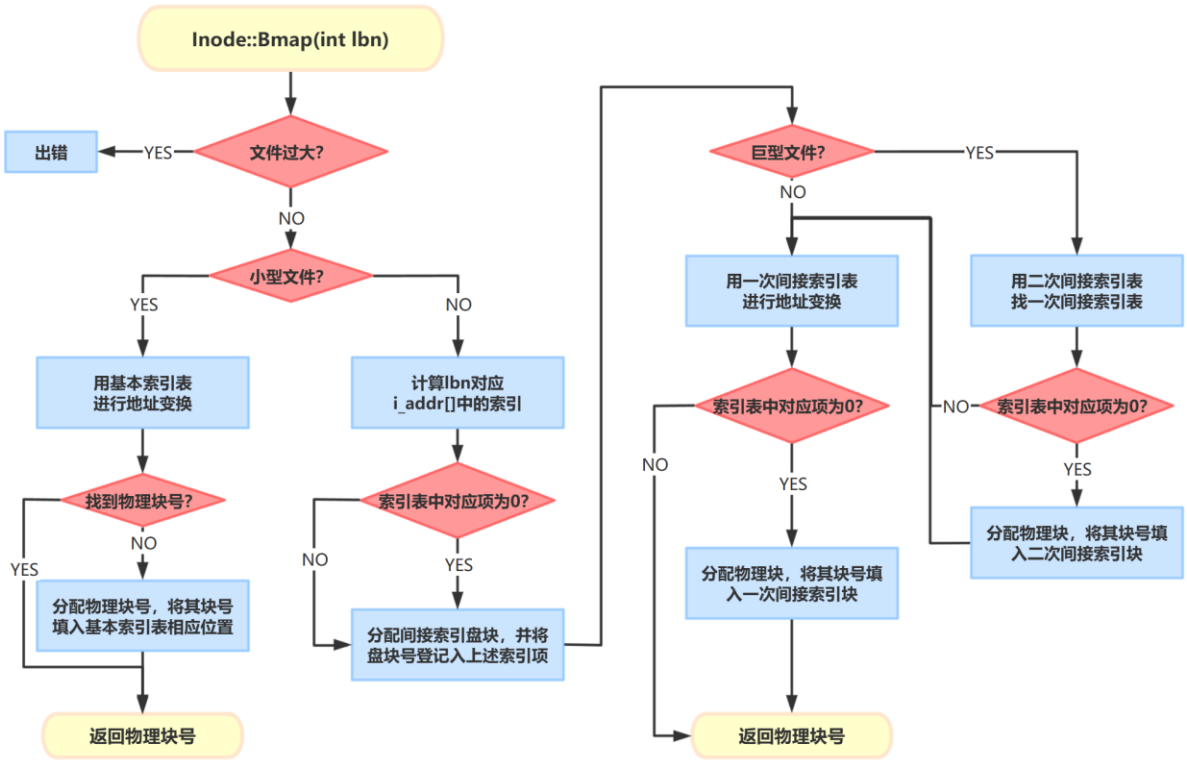


图 4.3.3 文件索引和数据块的映射转换流程图

4.4 文件系统实现

下面介绍文件系统读写功能的算法设计细节。

- 函数调用层次说明

User 类提供的文件读写接口为：

- `void u_Write(string fd, string inFile, string size)`
- `void u_Read(string fd, string outFile, string size)`

第一个参数是文件句柄，在打开文件时创建文件内存打开结构时取得。第二个参数是用户目标文件的路径名称，第三个参数是要传递的字节数。这两个函数主要通过调用 SysCall 的读写函数实现，在调用前后加上了用于提示用户输入合法性的逻辑，这里不加赘述，重点介绍真正实现读写功能的 `SysCall::Read()`和 `SysCall::Write()`。

这两个函数都是直接调用 `SysCall::Rdwr(enum File::FileFlags mode)`，但是传入的参数不同，读文件传入 `File::FREAD`，写文件传入 `File::FWRITE`。

Rdwr(mode)函数的主要流程如图 4.4.1 所示，其算法思想概括如下：

- 首先，根据 Read()/Write()的系统调用参数 fd 获取打开文件控制块结构，若不存在该打开文件结构则直接返回（GetF 已经设置过出错码，所以这里不需要再设置了）。

- 获取读写参数到 IOParam，包括目标缓冲区首地址、要求读写的字节数，以及文件读写的起始地址。

- 根据 mode 进行读或写，分别调用 `void Inode::ReadI()`和 `void Inode::WriteI()`。这两个函数具体落实读写动作，会根据 Inode 对象的物理磁盘块索引表，将数据写入文件。

- 最后根据读写字数，移动文件读写偏移指针，返回实际读写的字节数，修改存放系统调用返回值的核心栈单元。

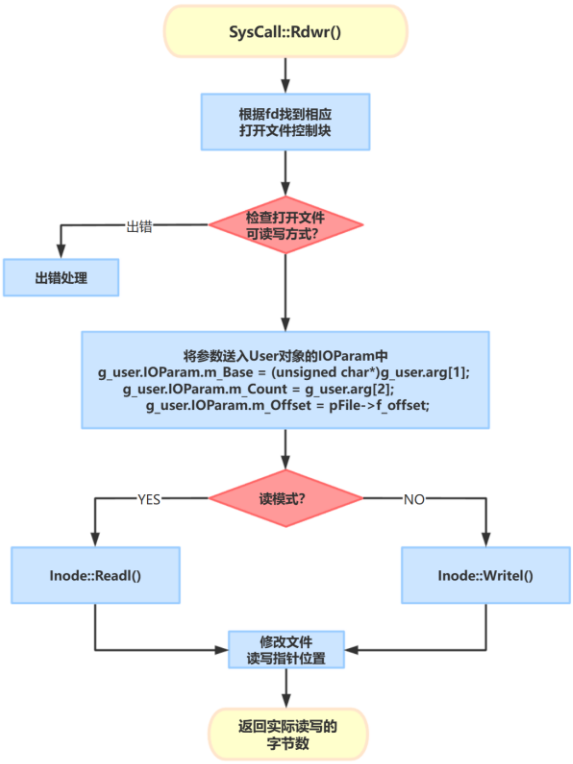


图 4.4.1 文件读写的公共逻辑 Rdwr()流程图

- 读文件

Inde::ReadI()函数根据 Rdwr 中预设好的、存在 g_user.IOParam 中的读写参数，访问需要读取的文件的内存 Inode 对象，通过逻辑文件定位文件的物理盘块号(Bmap 函数实现)，进而得到数据文件的磁盘块地址。然后，通过设备驱动模块将磁盘块按需读入缓存块。最终，送入到用户目标区中。该函数的算法流程图如 4.4.2 所示，本系统并没有实现顺序读情形下的预读算法。

值得说明的是 IOParam 参数的作用机制，它确定了以下三个值：

- 文件的逻辑块号：lbn=g_user.IOParam.m_Offset / 512
- 缓存内起始传送的位置：offset=g_user.IOParam.m_Offset% 512
- 从缓存内offset开始还要传送的字节数nbytes：

一方面，传送到用户区的字节数量，取读请求的剩余字节数与当前字符块内有效字节数较小值。

nbytes = min(Inode::BLOCK_SIZE-offset,g_user.IOParam.m_Count);

另一方面，传送的字节数量还取决于剩余文件的长度，不能超过剩余的字节数 remain。

nbytes = min(nbytes,remain);

最后，在每次读取完一个数据块后，都要更新参数的值：

g_user.IOParam.m_Base += nbytes;

g_user.IOParam.m_Offset += nbytes;

g_user.IOParam.m_Count -= nbytes;

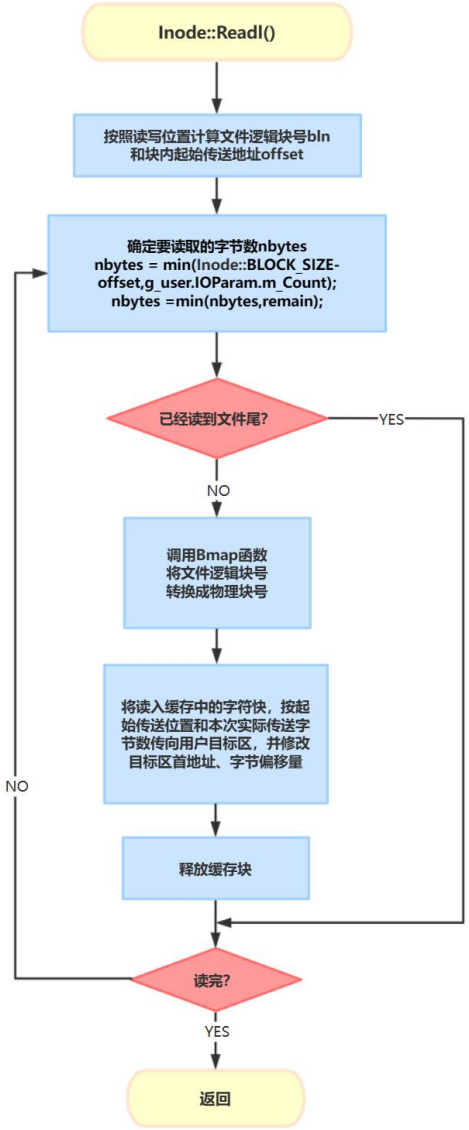


图 4.4.2 读文件流程图

- 写文件

写文件通过 Inode::WriteI 实现，本次实现对于缓存块不论是否写满都统一采用延迟写回，而没有采用异步写方式，图 4.4.3 展示了写文件的主要流程。

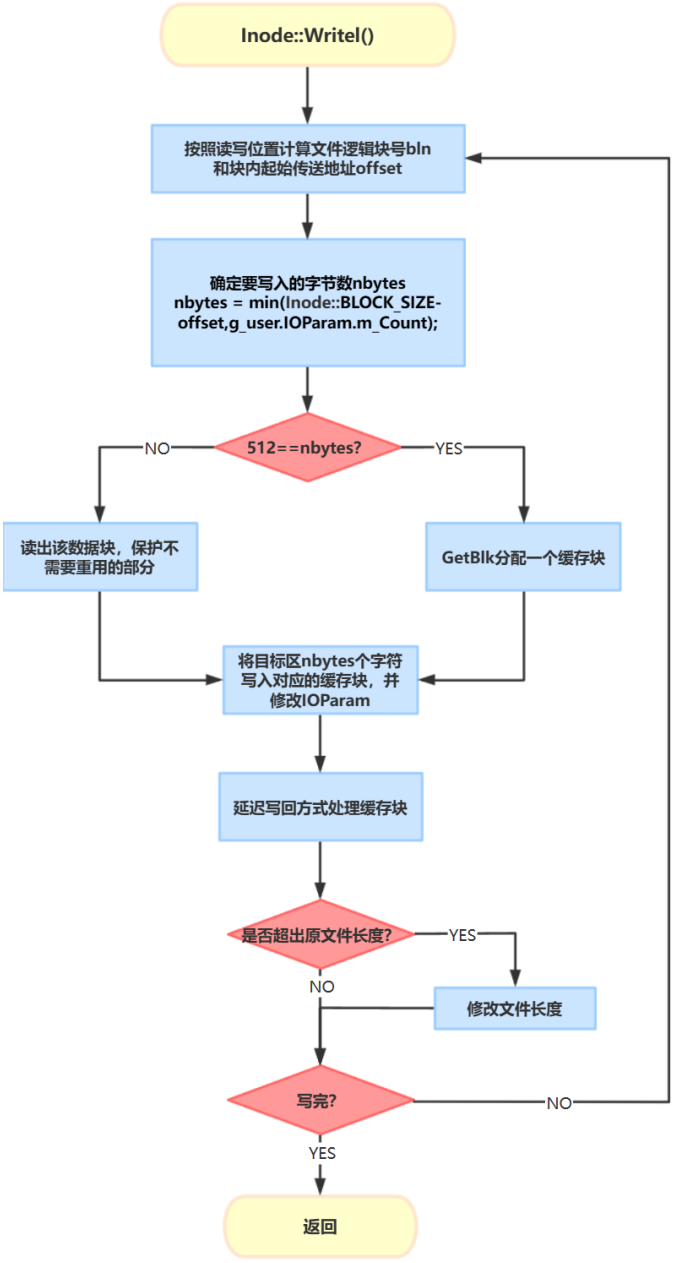


图 4.4.3 写文件的主要流程

4.5 高速缓存结构说明

基于概要设计中介绍的缓存控制块的设计与结构，下面具体说明缓存块的管理和读写的实现算法。

4.5.1 缓存队列的设计及分配和回收算法

- 缓存队列的设计

mBuf[NBUF]即 100 个缓存块对应的缓存控制结构，再配合一个自由队列头结点 bFreeList，构成了一条双向循环的 buf 队列，借此管理所有空闲的缓存块。

缓存队列的结构如图 4.5.1 所示。其中第一个是固定的头结点，初始化时，头结点 bFreeList 的

前后队列指针都指向自身。缓存队列取缓存块的操作只能取对头的缓存块，加入缓存只能加到队尾，这样就保证了缓存块能经可能久地保留在队列中，其可被重用的时间就是从队尾到队头所的时间。

另外，缓存块有 B_DONE 和 B_DELWRI 两种标志，含义和作用如下：

• 缓存队列中的所有缓存块都有 B_DONE 标志，表示它们当前都是不进行 IO 的（本系统没有设置 IO 队列，因为所有 IO 都是同步完成的）。

• 部分含有 B_DELWRI 的缓存块表示其内容相对磁盘是更新的，若要被取下作他用，需要先写回之，该标志保证了缓存块的尽量重用和延迟写回。

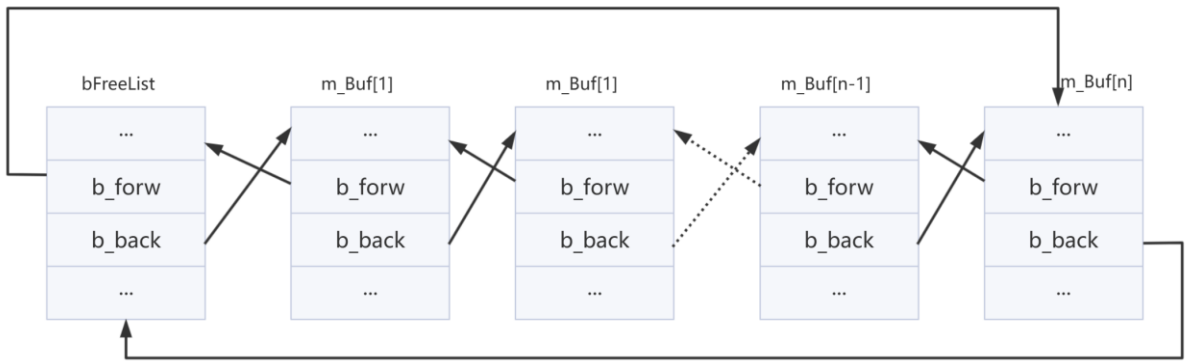


图 4.5.1 缓存块自由队列图示

- 分配缓存

通过 Buf* BufferManager::GetBlk(int blkno)实现，其大致算法思路如下：

• 若当前对列已有存储 blkno 数据块的缓存，则重用该缓存，将它从缓存队列中取下，返回该 Buf 的指针。

• 若无可重用的，申请队列的第一个缓存块。对于该取下的缓存要先判断是否需要执行延迟写回，若需要则同步写回，再清掉所有标志。

值得说明的是，根据是否为可重用情形得到的缓存块，利用 GetBlk 中对标志位的处理实现了区分，在后续 Bread 和 Rwrite 函数中起到重要作用。若得到的缓存块含有 B_DONE 标志，则说明是可重用情形下得到的缓存块，不必再进行 IO，反之则还需进行 IO。

- 回收缓存

通过函数 void BufferManager::Brelse(Buf* bp)实现缓存块的回收。该函数只是简单地将该缓存块加入到自由队列的队尾，仅设置 B_DONE，表示该缓存已无它用。

4.5.2 缓存块对一级文件系统的读写操作流程

缓存块对一级文件系统的读写操作是通过全局的磁盘驱动对象 DiskDriver g_diskDriver 提供的方法完成的。磁盘驱动类提供了直接读写磁盘文件的方法，具体是下面两个函数实现的。

• void DiskWrite(const void* ptr, size_t size, int offset = -1, size_t whence = SEEK_SET)

• `void DiskRead(void* ptr, size_t size, int offset = -1, size_t whence = SEEK_SET)`

缓存块 Buf 类中通过 `Buf* BufferManager::Bread( int blkno)`和 `void BufferManager::Bwrite(Buf* bp)`调用上述磁盘读写方法，下面介绍这两个函数的实现思路。

- 缓存块读磁盘块 Bread

• 通过 GetBlk 为盘块号 blkno 的数据块申请一个缓存块

• 若该缓存块含有 B_DONE 标志，说明是已经进行了 IO 的缓存块，直接返回指向该缓存块的指针。若不含 B_DONE 标志，说明缓存块内还没有装载磁盘块的内容，需要进行磁盘 I/O，调用 DiskRead 实现同步读取，随后为之加上 B_DONE 标志，并返回指向该缓存块的指针。

这样就得到了存有盘块号 blkno 的数据块内容的缓存块。

- 缓存块写磁盘块 Bwrite

该函数实现思路是：清除缓存块延迟写标志，调用 DiskWrite 实现同步写入，随后加上 B_DONE 标志，放回到自由队列的队尾。

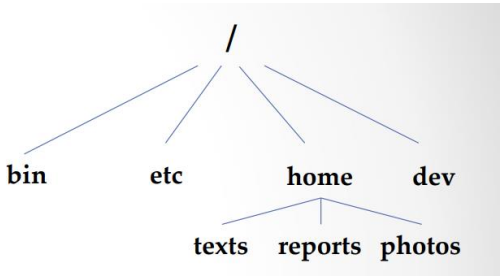
不久的未来若还需对同一磁盘数据块进行读写，只要该缓存块还在自由队列上，就能直接重用之。

5 调试分析及正确性验证

测试说明

输入指令“autotest”开始自动测试文件系统，测试内容为：

- 用 mkdir 命令创建子目录，建立如图所示目录结构；



- 将课设报告，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；
- 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

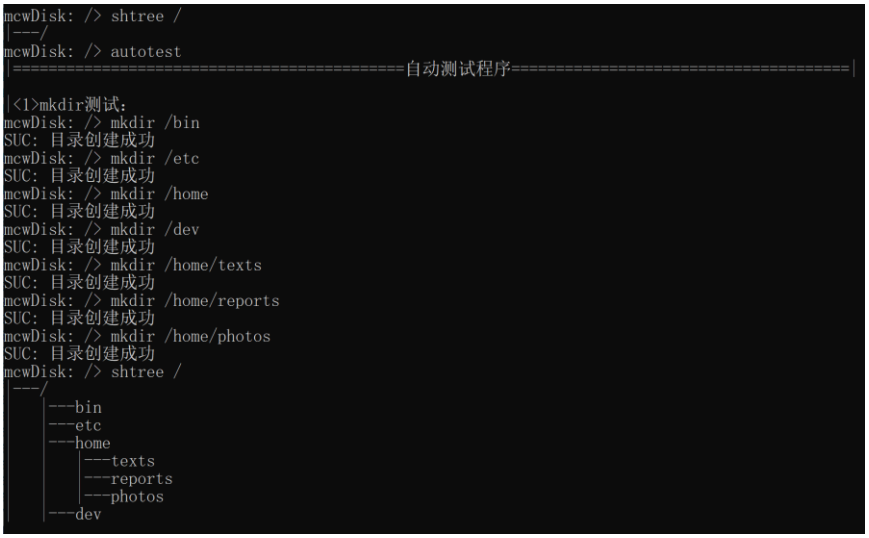
  在自动测试前，最好保证文件系统是崭新的，可以通过输入 fformat 格式化后重新运行程序。且在测试前必须将要用于辅助测试的文件放在与源文件同一工作目录下，包括以下标亮的 5 个文件。

User.cpp	2023/5/30 0:28	C++ Source	9 KB
ReportDemo.pdf	2023/5/31 20:58	Microsoft Edge PD...	203 KB
TJOS_FileSystem.vcxproj.user	2023/5/25 14:26	Per-User Project O...	1 KB
picture.png	2023/5/31 21:02	PNG 文件	590 KB
abc.txt	2023/5/31 21:39	Text Document	1 KB
ReadMe.txt	2023/5/31 21:01	Text Document	3 KB
usercin.txt	2023/5/31 21:20	Text Document	2 KB
TJOS_FileSystem.vcxproj	2023/5/29 0:38	VC++ Project	8 KB

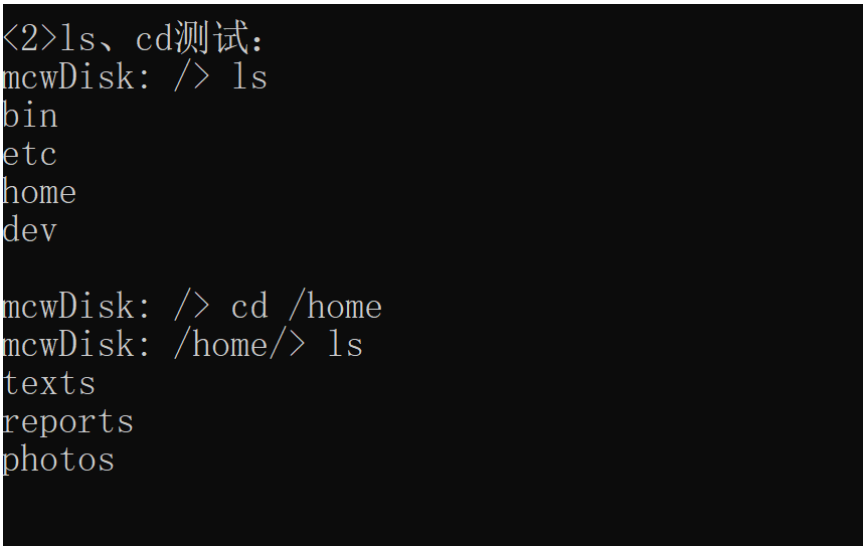
此外，事先应准备好 800+字节数据存放在 usercin.txt 中。

测试结果展示与分析

- 首先打开一个崭新的文件系统(即可)，输入指令“shtree /”可以看到当前文件系统的目录树是只有根结点的，接着输入“autotest”开始自动测试程序。
- 目录结构建立测试。测试过程和结果展示如下，可以通过目录树直观地观察到目录结构构建成功。



- 目录展示指令 ls 和路径修改指令 cd 测试。结合上面的目录树可以发现 ls 指令准确无误；通过 cd 指令，cmd 行的路径改变了，且再次输入 ls，显示的是改变后路径下的各目录项。



- 文件创建、打开测试。

  在/home/texts 中创建 ReadMe.txt 文件，然后打开它，并根据打开文件时得到的 fd 将准备好的用户 ReadMe.txt 写入磁盘的 ReadMe.txt 中。

  在/home/reports 中创建文件 Report.pdf，将然后打开它，并根据打开文件时得到的 fd 将准备好的用户 ReportDemo.pdf 写入磁盘的 Report.pdf 中。

  在/home/photos 中创建文件 myPic.png，将然后打开它，并根据打开文件时得到的 fd 将准备好的用户 picture.png 写入磁盘的 myPic.png 中。

  测试结果如下所示，各指令都得到了正确的响应结果。

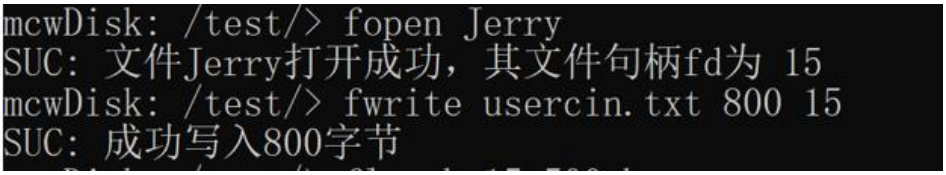
```
|<3>fcreat、fopen、fwrite测试：
mcwDisk: /home/> cd /home/texts
mcwDisk: /home/texts/> fcreat ReadMe.txt
SUC: 文件ReadMe.txt创建成功!
mcwDisk: /home/texts/> fopen ReadMe.txt
SUC: 文件ReadMe.txt打开成功，其文件句柄fd为 8
mcwDisk: /home/texts/> fwrite ReadMe.txt 2069 8
SUC: 成功写入2069字节
mcwDisk: /home/texts/> cd /home/reports
mcwDisk: /home/reports/> fcreat Report.pdf
SUC: 文件Report.pdf创建成功!
mcwDisk: /home/reports/> fopen Report.pdf
SUC: 文件Report.pdf打开成功，其文件句柄fd为 10
mcwDisk: /home/reports/> fwrite ReportDemo.pdf 206834 10
SUC: 成功写入206834字节
mcwDisk: /home/reports/> cd /home/photos
mcwDisk: /home/photos/> fcreat myPic.png
SUC: 文件myPic.png创建成功!
mcwDisk: /home/photos/> fopen myPic.png
SUC: 文件myPic.png打开成功，其文件句柄fd为 12
mcwDisk: /home/photos/> fwrite picture.png 604047 10
SUC: 成功写入604047字节
mcwDisk: /home/photos/> shtree /
|---/
    |---bin
    |---etc
    |---home
        |---texts
            |---ReadMe.txt
        |---reports
            |---Report.pdf
        |---photos
            |---myPic.png
    |---dev
```

- 文件读写相关指令的正确性测试

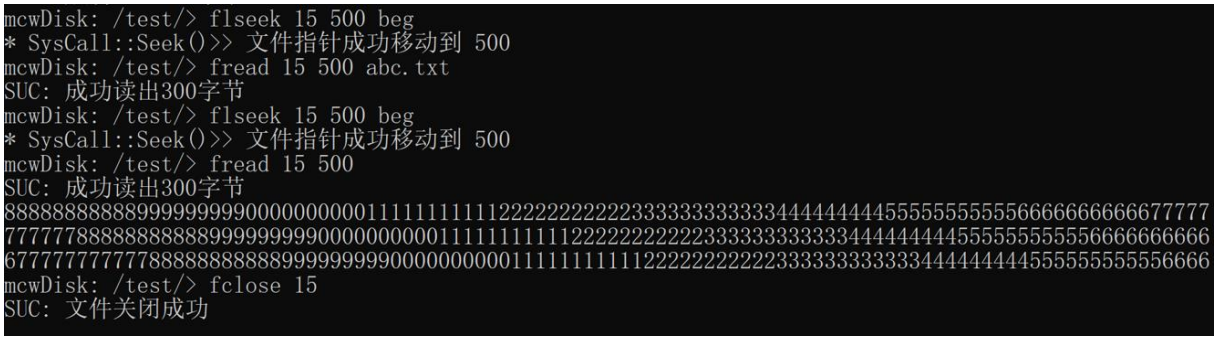
首先，按照要求创建了/test/Jerry：

```
|<4>文件读写相关指令测试：
mcwDisk: /home/photos/> mkdir /test
SUC: 目录创建成功
mcwDisk: /home/photos/> cd /test
mcwDisk: /test/> fcreat Jerry
SUC: 文件Jerry创建成功!
mcwDisk: /test/> shtree /
|---/
    |---bin
    |---etc
    |---home
        |---texts
            |---ReadMe.txt
        |---reports
            |---Report.pdf
        |---photos
            |---myPic.png
    |---dev
    |---test
        |---Jerry
```

打开 Jerry 文件，将 usercin.txt 中前 800 个字节写入其中，结果如下。



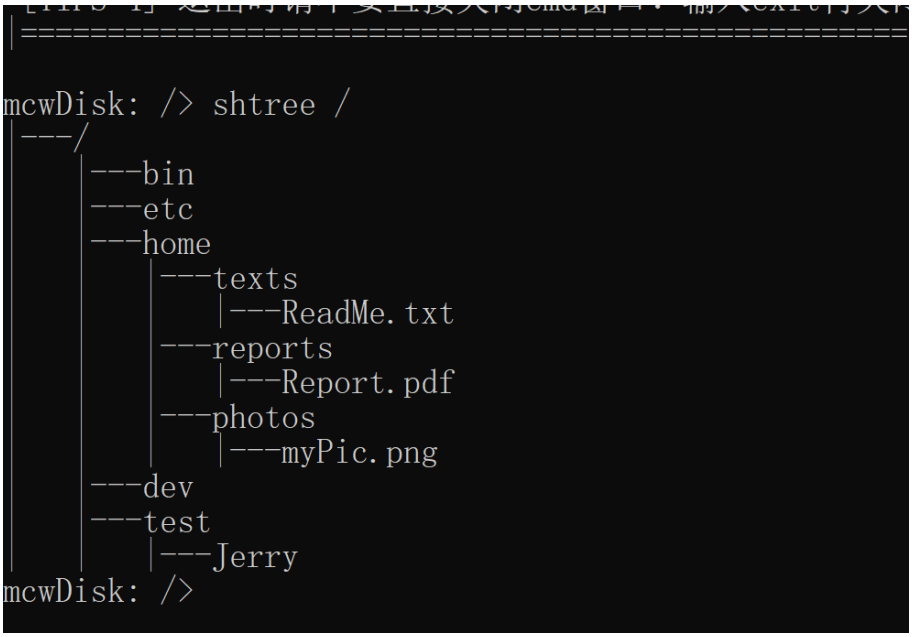
将文件读写指针定位到第 500 字节，读出 500 个字节到文件 abc 中。由于文件一共只有 800 字节，所有应当**只能读出 300 字节**，观测结果确实符合预期。再将读出的 300 字节输出到屏幕上，发现是正确的数据。



此时，打开 abc.txt 文件，发现其有 300 字节的内容，与上图中展示的输出数据一致。

最后关闭文件。

输入“exit”退出，让缓存块内容写回到磁盘。再重新打开文件系统，输入“shtree /”发现文件的目录结构没有丢失。



输入“fformat”后再打开，文件卷则完全刷新。

综上所示，本文件系统能顺利正确地实现目录的建立，文件的建立、打开关闭、读写、读写指针的移动等基本功能，能通过本次课程设计要求的测试任务。

6 用户使用说明

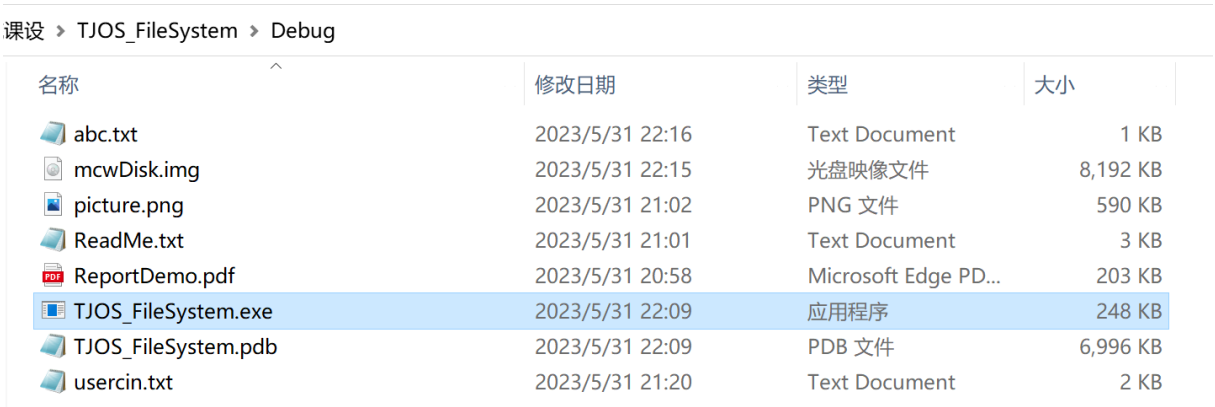
环境说明

本实验的环境部署和硬件配置说明如下表所示：

操作系统	**Windows10**
开发语言	C++
编译器	VS2019 DEBUGX86

项目运行

- 可以通过在 VS2019 集成开发环境中直接运行该项目，自动测试所需的数据已经准备好。
- 也可以直接运行生成的 exe 文件。



交互说明

通过 cmd 方式交互，运行程序就会看到用户使用指南。每一条输入都会有 cmd 输出响应，包括正确执行的结果响应、错误用法的提示、未识别语句的提示等，通过输入 help 指令能得到指令格式、功能说明和用例等信息，相对友好易用，如下所示。

```
mcwDisk: /> fdelete diary.txt
ERR: 找不到文件或者文件夹!
mcwDisk: />
mcwDisk: />
mcwDisk: /> fclose  myfile.txt
ERR: 请输入正确的文件句柄（fd号）!
mcwDisk: />
mcwDisk: /> flseek 10 0 begin
ERR: 输入有误，help flseek查看使用方法~
mcwDisk: /> help flseek
[指令]flseek <fd> <offset> <mode>
[说明]定位文件读写指针，<mode>取beg/cur/end，分别表示从句柄<fd>所指文件的“起始、当前、结束位置”偏移<offset>字
节
[用例]·flseek 1 0 beg 调整fd=1所指文件的读写指针到文件起始字节
      ·flseek 1 10 cur 调整fd=1所指文件的读写指针后移10字节
      ·flseek 1 -10 end 调整fd=1所指文件的读写指针到文件末尾倒数第10字节
mcwDisk: />
```

- 其它注意事项

由于缓存队列中所有延迟写回块在程序结束前不一定有被抢占而写回的机会，所以在程序结束时，在缓存管理对象的析构函数中统一处理了延迟写回的问题。

如果直接关闭可执行文件窗口会导致无法顺利执行析构函数，进而导致本次对文件系统的操作内容可能发生丢失。

解决方法：退出文件系统时，应当输入“exit”指令。

7 总结与收获

上学期的操作系统理论课程是我本科期间收获最大的课程之一，也是学得相对扎实的一门专业课程，通过本次课设，我加深了对 Unix 系统设计思想的理解程度，收获颇多。

在学习操作系统之前，我对底层软件结构所知甚少，理论课程让我掌握了基本的理论知识，而通过具体学习 Unix 系统，让抽象飘渺的知识落地生根，让曾经浮于表面的概念变成了脑海中复杂的思维导图和数据结构。

我记得上学期常会感叹 Unix 设计之精妙，课本上简单的概念在具体实现时其实并不简单，很可能需要多个结构的巧妙配合，比如 Unix 的分时系统、进程的抢占调度等等都需要反复斟酌才能比较完整地理解其设计思想。本次课程设计虽然没有涉及到 Unix 最复杂的进程切换调度等内容，但仅仅是单进程的文件系统已经是一座要费力才能攀登的大山。

在本次课程设计前，我重新回顾了 Unix V6++操作系统的主要设计思路，再次熟悉其主要的设计思想、并对各类有了一定的认识后，我开始研究文件系统的代码，而研究代码的过程不同于学习老师整理归纳好的设计思想，需要自己抽丝剥茧、对代码细枝末节有准确的认知。我起初则常常迷失在代码调用的迷宫中无以自拔。当好不容易理解并确定好要保留哪些结构和函数后，才真正走上正轨，其中几经波折，煞费苦心，通过反复调试一点点将代码整合了起来。

这次代码实现基本参照 Unix V6++，设计思想上也想与之一致。一方面是自己想要进一步了解 Unix 是如何设计，从中学到些设计思想；另一方面是自己目前仍功力平平，必须得多加学习，所以自身条件要求我不得不参考 Unix 源码。幸运的是，我在阅读代码的过程中的收获超过自己的预期，我的 c++代码书写的规范性、高效性得到了提升，在处理磁盘文件时对指针偏移寻址的灵活使用得到了提升等等。此外，对上学期学的不扎实的地方也起到了补充提高的作用。书写报告的过程也帮助我比较彻底地理清了思路。

总得来说，这一年学习操作系统的过程让我收获良多，感谢方老师负责细致的授课~~感谢自己一路上端正积极的学习态度。祝自己未来继续在学习与实践中收获、成长~

8 参考资料

[1] Unix V6++操作系统课程讲义

[2] Unix V6++源代码