同济大学计算机系

操作系统课程设计报告

学　　号 2151769

姓　　名 吕博文

专　　业 计算机科学与技术

授课老师 方钰

时　　间 2024 年 5 月 16 日

一、实验概述

1.1 实验目的

UNIX V6++文件系统提供了层次结构的目录和文件，负责系统内部的文件信息管理，文件系统是整个 UNIX V6++操作系统中的重要组成部分。本实验通过设计一个类 UNIX 文件系统，实现文件系统的一些基本功能，帮助同学们熟悉 UNIX V6++的文件系统结构及系统实现。

1.2 实验要求

要求使用本地的一个普通大文件（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。

数据结构定义上，要求：

（1）给出文件系统的结构说明，至少包括：

- Superblock 及 Inode 区所在位置及大小
- Inode 节点数据结构的定义及索引结构的说明
- Superblock 数据结构定义及对 Inode 节点及文件数据区管理算法

（2）目录结构，至少包括：

- 目录文件的结构
- 目录检索算法的设计
- 目录结构的增、删、改的设计

（3）文件打开结构说明，至少包括：

- 文件打开结构设计
- 内存 Inode 节点数据结构定义及分配与回收
- 文件打开过程

（4）文件系统实现说明，至少包括：

- 文件读写操作实现流程
- 文件其他操作实现流程

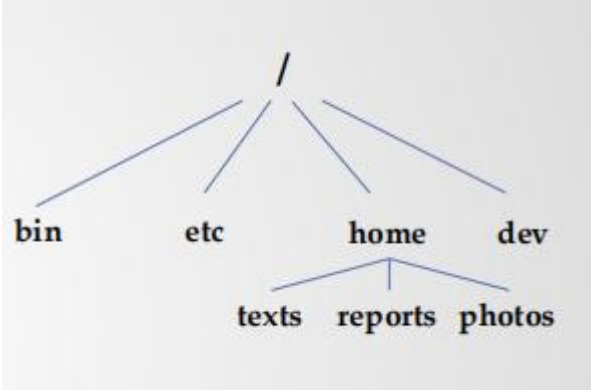
（5）高速缓存结构说明，至少包括：

- 缓存控制块的设计
- 缓存队列的设计及分配与回收算法
- 借助缓存实现对一级文件的读写操作

代码实现上，要求：

（1）通过命令行方式完成下列操作：

- 格式化文件卷
- 通过 mkdir 命令创建子目录，建立如图所示目录结构：



-
- 将课设报告，ReadMe.txt 和一张图片存入改文件系统，分别放在 /home/texts, /home/reports 和/home/photots 文件夹；

（2）通过命令行测试：

- 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

文件操作接口上，至少实现：

• fformat:	格式化文件卷	• fread:	读文件
• ls:	列目录	• fwrite:	写文件
• mkdir:	创建目录	• flseek:	定位文件读写指针
• fcreat:	新建文件	• fdelete:	删除文件
• fopen:	打开文件	• 。。。	
• fclose:	关闭文件		

1.3 实验环境

- 操作系统：Windows 11
- 编译器选项：g++ (GCC) 3.4.5 (mingw-vista special r3)（同 UNIX V6++）
- 编辑器：VsCode
- 项目管理：makefile
- 调试工具：gdb
- 代码管理：github

二、需求分析

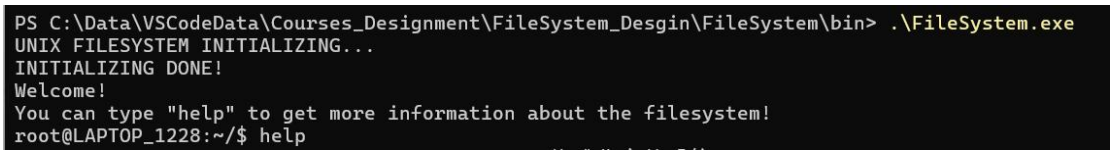
2.1 程序输入

要求实现一个控制台程序，图形界面或命令行方式，等待用户输入，提供文

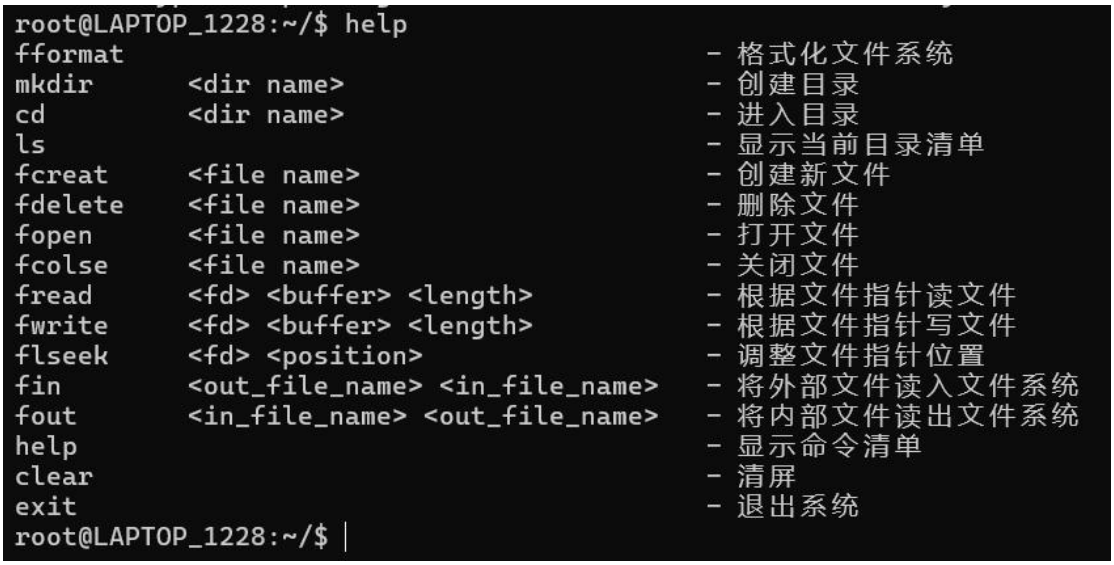
件系统的基本功能，并根据用户不同的输入，返回正确的结果。本文件系统在完成基本要求的情况下尽可能做到健壮性，对出错的命令给与提示。

2.2 程序输出

程序通过控制台命令交互方式引导用户输入并反馈输出，进入文件系统：



使用 help 命令查看指令内容如下：



2.3 程序功能

本文件系统实现的用户指令如下：

- ✧ fformat——格式化文件系统
- ✧ mkdir <dir name>——创建目录
- ✧ cd <dir name>——进入目录
- ✧ ls——显示当前文件夹内容
- ✧ fcreat <file name> ——创建新文件
- ✧ fdelete <file name>——删除文件
- ✧ fopen <file name>——打开文件
- ✧ fclose <file name>——关闭文件
- ✧ fread <fd> <buffer> <length>——根据文件指针读文件
- ✧ fwrite <fd> <buffer> <length> ——根据文件指针写文件
- ✧ flseek <fd> <position> ——调整文件指针位置
- ✧ fin <out_file_name> <in_file_name> ——将外部文件读入系统
- ✧ fout <in_file_name> <out_file_name>——将内部文件读出系统
- ✧ help——显示所有命令

- ✧ clear——清屏
- ✧ exit——退出系统

三、概要设计

3.1 任务分解

本文件系统实现过程中参考了 UNIX V6++的源代码实现，在理解 UNIX V6++源代码中有关文件系统实现的部分后，对代码进行裁剪补充，最后完成本项目文件系统，故系统模块与 UNIX V6++类似，模块主要分为：

- 磁盘驱动模块（DeviceManager）:该文件在 UNIX V6++中负责管理所有设备，这里文件系统只用到了磁盘，所以该模块负责初始化磁盘镜像文件,通过 C++系统调用对磁盘镜像文件进行读写。
- 高速缓存管理模块（BufferManager）:负责管理文件系统中的缓存块，实现了缓存块的分配、回收、读写等等。
- 文件系统资源管理模块（FileSystem）:负责管理文件存储设备中的各类存储资源，包括外存 Inode 的分配释放等等。
- 打开文件管理模块(OpenFileManager):负责用户对打开文件机构的管理，建立用户与文件内核数据的关系。
- 文件管理模块（FileManager）:提供文件系统的接口，包括文件的打开、关闭、删除、读写、创建、文件指针的移动等等。
- 内核模块（Kernel）：组合以上几个文件系统模块，提供与外部的接口
- 顶层 API 模块（Command）：系统初始化，命令行解析，实现与用户的交互。

3.2 数据结构定义

◆ Superblock 结构体

这里我们遵从 UNIX V6++中对于 Superblock 结构体的定义，共 512 字节，分别包括外存 Inode 占用盘块数，盘块总数，空闲盘块数及索引表，空闲外存数及索引表，其余与进程有关的变量本文件系统并未使用到，但为保证总字节数不变没有删除。

```
class SuperBlock
{
    /* Functions */
public:
    /* Constructors */
    SuperBlock();
    /* Destructors */
```

```
    ~SuperBlock();

    /* Members */
public:
    int     s_isize;    /* 外存 Inode 区占用的盘块数 */
    int     s_fsize;    /* 盘块总数 */
    int     s_nfree;    /* 直接管理的空闲盘块数量 */
    int     s_free[100]; /* 直接管理的空闲盘块索引表 */
    int     s_ninode;       /* 直接管理的空闲外存 Inode 数量 */
    int     s_inode[100];   /* 直接管理的空闲外存 Inode 索引表 */
    int     s_flock;    /* 封锁空闲盘块索引表标志 */
    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 */
};
```

◆ Inode 结构体：

遵循 UNIX V6++定义，大小为 64 个字节，每个 Block 块中存放 8 个 Inode 块，内包括引用计数 i_count,文件联结计数 i_nlink,文件大小 i_size,逻辑物理地址转换数组 i_addr[10],具体定义：

```
    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];
        int      i_lastr;
    };
```

◆ DirectoryEnty 结构体

目录结构，包括目录项 Inode 编号 m_ino 和目录项名称 m_name[]

```
    public:
        int m_ino;       /*  目录项中 Inode 编号部分  */
        char m_name[DIRSIZ];     /*  目录项中路径名部分  *
```

◆ User 结构体

这里的 User 主要用来实现 main 函数中的接口调用，参数传递和返回值传递，并未实现多用户功能。

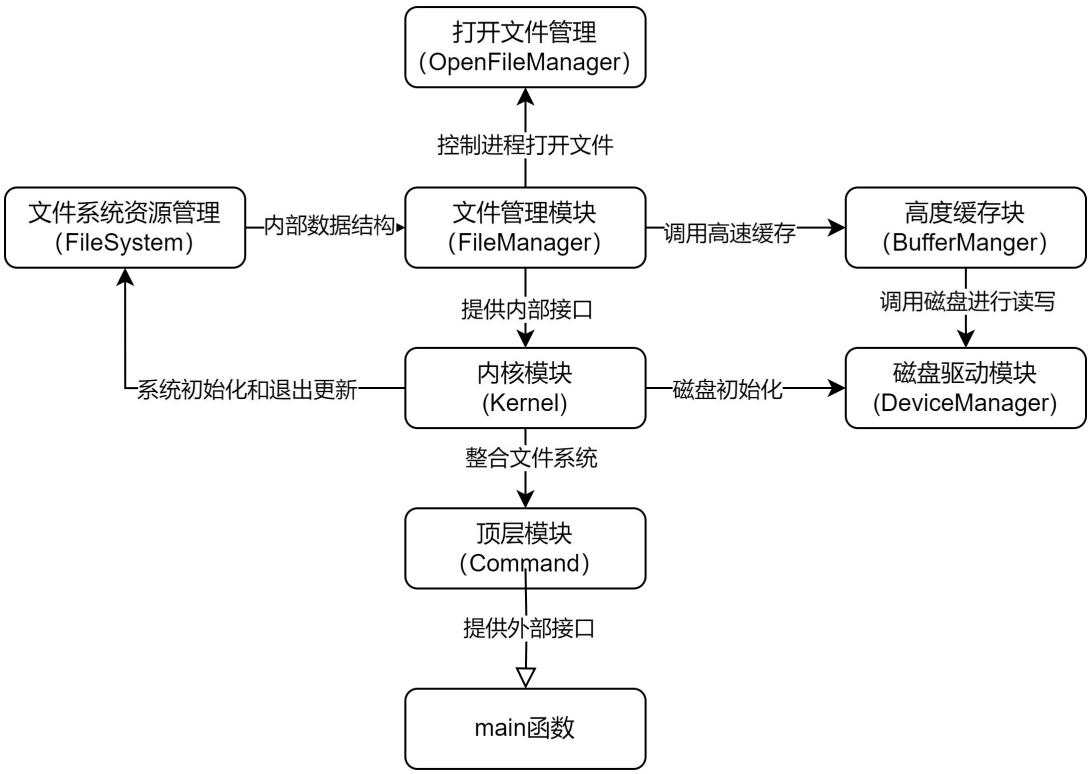
```
    public:
        //系统调用返回值
        int system_ret;
        int u_arg[5];                /*  存放当前系统调用参数  */
        char* u_dirp;                 /*  系统调用参数(一般用于 Pathname)的指针
*/

        /*  文件系统相关成员  */
        Inode* u_cdir;       /*  指向当前目录的 Inode 指针  */
        Inode* u_pdir;       /*  指向父目录的 Inode 指针  */

        DirectoryEntry u_dent;                    /*  当前目录的目录项  */
        char u_dbuf[DirectoryEntry::DIRSIZ];     /*  当前路径分量  */
        char u_curdir[128];                      /*  当前工作目录完整路径  */
```

3.3 模块调用关系

本文件系统 FileSystem.img 空间分配如图所示：

0	1	2		1023			17999
Superblock1	Superblock2	Inode(0~8)	.....	Inode	Block0	.....	Block

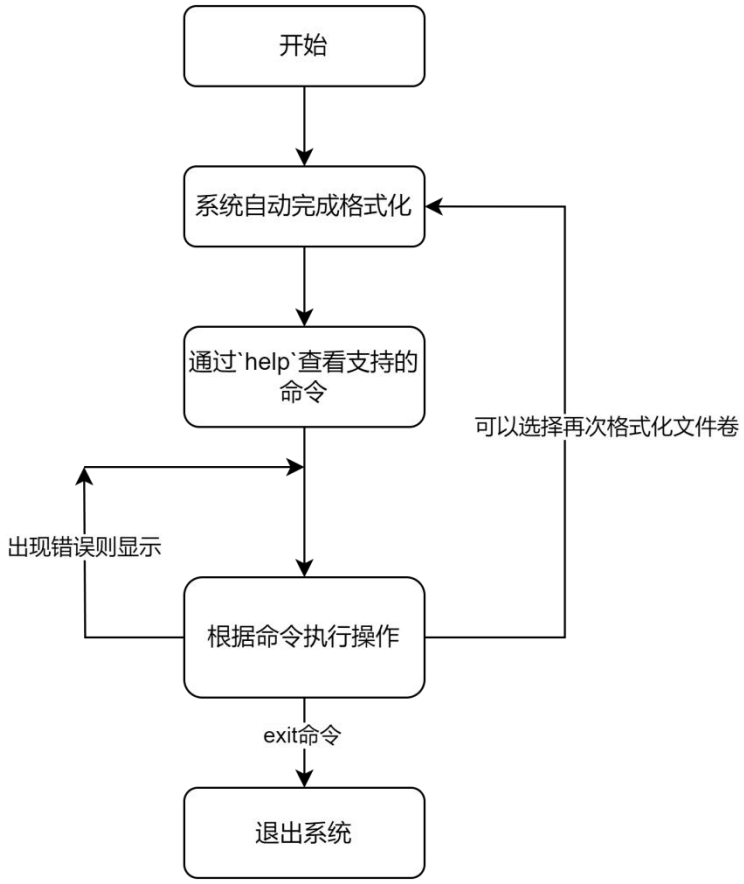
模块之间调用关系如图所示：



3.4 算法说明

整个程序执行流程为：

命令行 FileSystem.exe 启动文件系统，系统会自动格式化文件卷并提示命令，通过`help`命令可以查看文件系统支持的命令，输入对应的命令操作文件系统，操作正确或错误均有相应反馈，完成相关操作后，可以通过`exit`命令退出文件系统，再次进入时，系统保留了上次的工作记录。



四、详细设计

4.1 内核数据结构设计

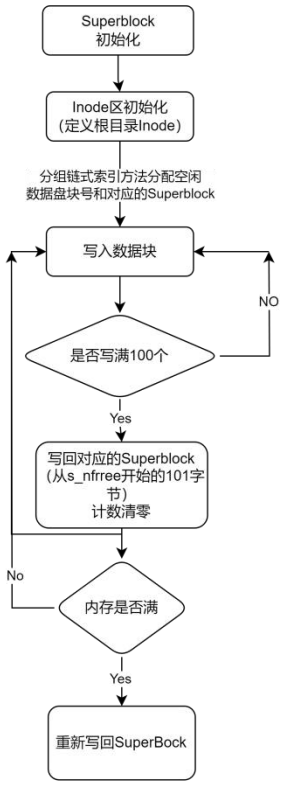
4.1.1 DeviceManger 类

```
class DeviceManager
{
private:
    static const char* DISK_FILE_NAME;   /*  磁盘文件名  */
    static const int BLOCK_SIZE = 512;

    FILE * m_DiskFile;        /*  磁盘文件指针  */
```

```
public:
    DeviceManager();
    ~DeviceManager();

    void Initialize();

    void FormatDisk();        /* 格式化磁盘 */

    void IO_read(Buf* bp);   /* 从磁盘读取数据 */

    void IO_write(Buf* bp); /* 向磁盘写入数据 */
};
```

DeviceManger 类负责对磁盘镜像文件的初始化以及读写操作，Initialize（）函数在磁盘镜像文件不存在时会创建磁盘文件并进行格式化，按照 UNIX V6++的磁盘格式分别划分为超级块、Inode 区和数据区；FormatDisk（）函数负责格式化磁盘，用户进入文件系统之后也可以选择手动二次格式化此时调用 FormatDisk 函数；IO_read()和 IO_write（）分别根据缓冲块的参数进行具体的读写磁盘操作。

FormatDisk()函数的具体流程如下：



4.1.2 Buf 和 BufferManager（高速缓存块）

```
    public:
        unsigned int b_flags;    /* 缓存控制块标志位 */
        int       padding;
        /* 缓存控制块队列勾连指针 */
        Buf*     b_forw;
        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 */
```

```
    class BufferManager
    {
    public:
        /* static const member */
        static const int NBUF = 15;            /* 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 */

    public:
        BufferManager();
        ~BufferManager();

        void Initialize();
        Buf* GetBlk(int blkno);
        void Brelse(Buf* bp);          /* 释放缓存控制块 buf */
        Buf* Bread(int blkno);   /* 读一个磁盘块。dev 为主、次设备号，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 Bawrite(Buf* bp);                /* 异步写磁盘块 */
```

```
    void ClrBuf(Buf* bp);                    /* 清空缓冲区内容 */
        void Bflush();                  /* 将 dev 指定设备队列中延迟写的缓存全部
输出到磁盘 */
        Buf& GetBFreeList();                 /* 获取自由缓存队列控制块 Buf 对
象引用 */
    private:
        void GetError(Buf* bp);              /* 获取 I/O 操作中发生的错误信息
*/
        void NotAvail(Buf* bp);              /* 从自由队列中摘下指定的缓存控
制块 buf */
    private:
        Buf bFreeList;                      /* 自由缓存队列控制块 */
        Buf m_Buf[NBUF];                     /* 缓存控制块数组 */
        unsigned char Buffer[NBUF][BUFFER_SIZE];    /* 缓冲区数组 */
        DeviceManager* m_DeviceManager;      /* 指向设备管理模块全局对
象,这里实际上只有磁盘 */
    };
```

Buf 类定义记录了相应缓存的使用情况信息，由于本文件系统不涉及进程的概念，也不存在多个设备，所以在控制指针上，只保留了自由缓存队列控制自由缓存块并对相应函数做出处理。

BufferManager 类负责缓存块和自由缓存队列的管理，包括缓存块的分配、回收、读写等操作，做到了按照 LRU 的方式管理缓存，尽量保证缓存的重用以提高文件读写效率，同时实现磁盘的延迟写操作。

4.1.3 Inode 类与 DiskInode 类

```
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 */
        int     i_number;       /* 外存 inode 区中的编号 */
```

```
    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 */
```

```
class DiskInode
{
    /* Functions */
public:
    /* Constructors */
    DiskInode();
    /* Destructors */
    ~DiskInode();

    /* Members */
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];     /* 用于文件逻辑块好和物理块好转换的基本索引表 */

    int     d_atime;        /* 最后访问时间 */
    int     d_mtime;        /* 最后修改时间 */
};
```

Inode 对应系统每个打开的文件、当前访问的目录，内存 Inode 通过 i_number 确定其唯一外存 DiskInode；

DiskInode 是外存（磁盘）中的 Inode 区中，记录每个文件对应控制信息。

每个文件根据其大小不同可以分为小型文件、大型文件、巨型文件，文件索引结构通过 d_addr[10]的多重索引结构来实现，具体叙述如下：

- 小型文件索引结构

小型文件使用文件索引表中的d_addr[0]-d_addr[5]这6项作为直接索引表对应文件的大小为 0~6 个数据块。

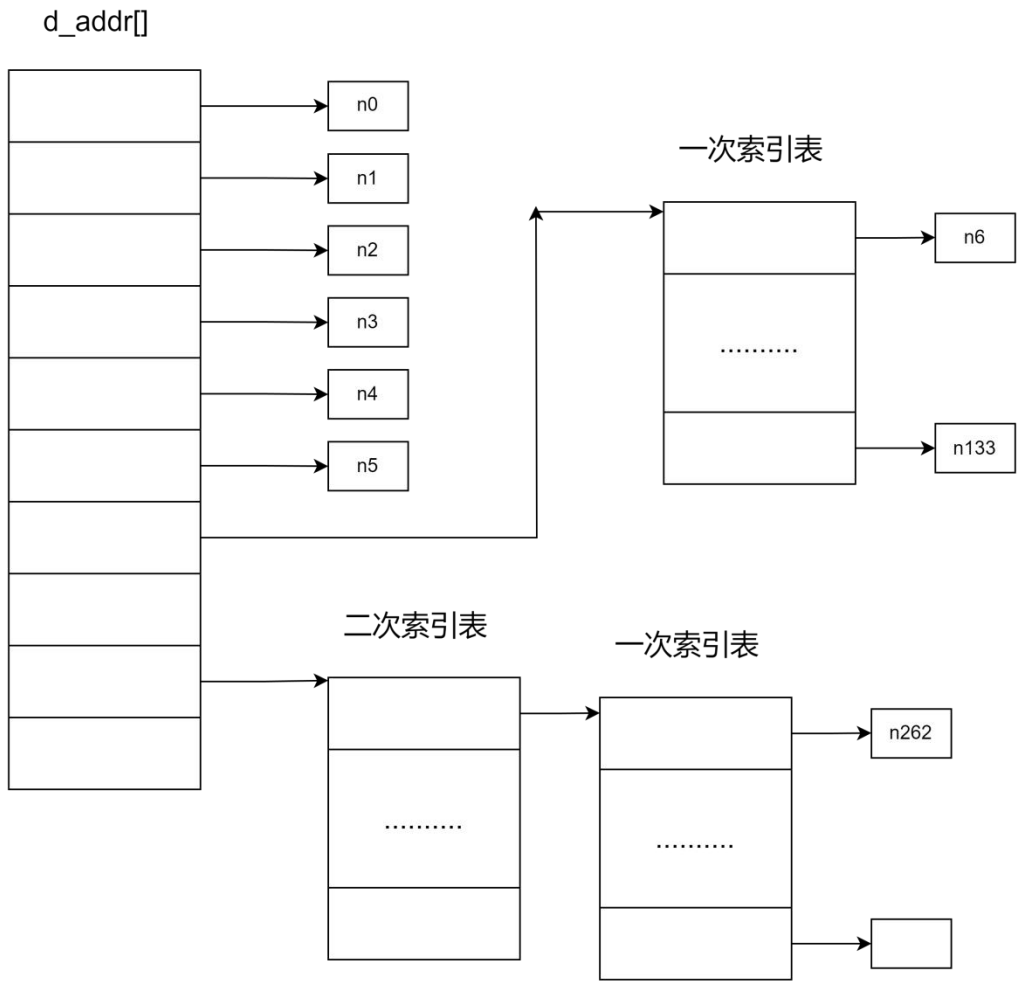
- 大型文件索引结构

大型文件除了使用前 6 个直接索引地址，还用到了 d_addr[6],d_addr[7]两项间接索引地址，每个间接索引表可容纳 128 个物理盘块号，对应文件大小为：(7~(6+128*2)个数据块。

- 巨型文件索引结构

巨型文件除了使用大型文件的索引结构还用到了 d_addr[8],d_addr[9]两个二次间接索引表，二次间接索引表中每一项执行一个一次间接索引表，再有一次间接索引表中的每一项指向一个具体的数据块号，文件大小为:(7 + 128*2) ~ (128*128*2 + 128*2 +6)个数据块。

结构示意图如下：



4.1.4 SuperBlock 类与 FileSystem 类

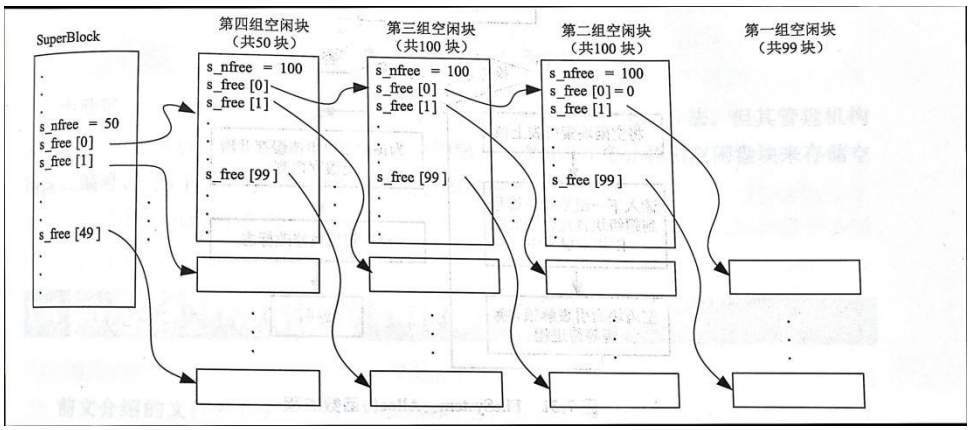
SuperBlock 的代码在概要介绍里已经给出。

```
public:
    /* Constructors */
    FileSystem();
    /* Destructors */
    ~FileSystem();

    void Initialize();
    void LoadSuperBlock();
    SuperBlock* GetFS();
    void Update();
    Inode* IAlloc();
    void IFree(int number);
    Buf* Alloc();
    void Free(int blkno);
```

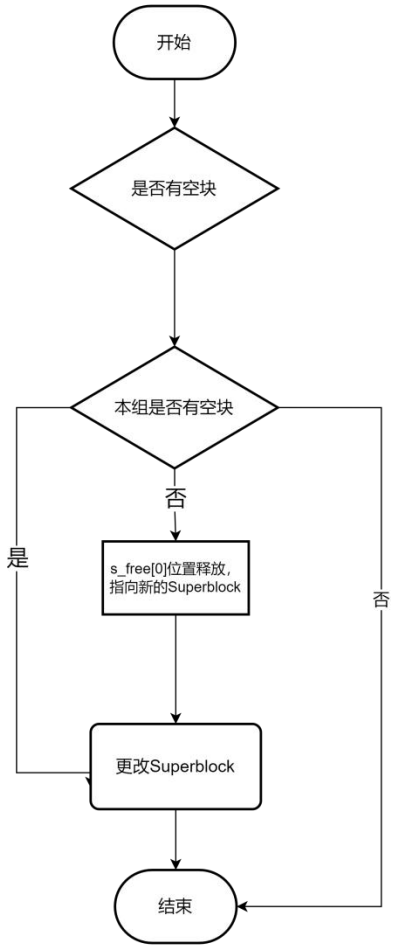
Superblock 类定义了文件系统存储资源管理块，这里直接采用了 UNIX V6++的 Superblock 类定义，因为要确保整个类大小为 1024 字节，恰好占满两个扇区，故没有删除无关变量。

FileSystem 类负责管理文件存储设备中的各类存储资源，以及磁盘块、外存 Inode 的分配与释放。

Superblock 对于空闲 Inode 的管理我们采用成组链发进行管理，所有的空闲块每 100 个构成一组（第一组只有 99 个），余下部分构成一组，最后一组直接由超级块中的空闲索引表 s_free[100]管理，其余各组的索引表则分别存放在它们下一组的第一个盘块的开头的 4*101 个字节中。基本结构图如下：



所以分配 Block 块的流程图如下：



4.1.5 File 类、OpenFIles 类、IOParameter 类

```
class File
{
public:
    /* Enumerate */
    enum FileFlags
    {
        FREAD = 0x1,            /*  读请求类型  */
        FWRITE = 0x2,           /*  写请求类型  */
        FPIPE = 0x4             /*  管道类型  */
    };
    public:
    File();
```

```
    /* Destructors */
        ~File();

        /* Member */
        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量  */
        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  */
    };
```

```
class OpenFiles
{
    /* static members */
public:
    static const int NOFILES = 15;   /*  进程允许打开的最大文件数  */
    /* Functions */
public:
    /* Constructors */
    OpenFiles();
    /* Destructors */
    ~OpenFiles();
    int AllocFreeSlot();
    File* GetF(int fd);
    void SetF(int fd, File* pFile);

    /* Members */
private:
    File *ProcessOpenFileTable[NOFILES];
    };
```

```
class IOParameter
{
    /* Functions */
public:
    /* Constructors */
    IOParameter();
    /* Destructors */
    ~IOParameter();
    /* Members */
public:
    unsigned char* m_Base;  /*  当前读、写用户目标区域的首地址  */
    int m_Offset;    /*  当前读、写文件的字节偏移量  */
    int m_Count;     /*  当前还剩余的读、写字节数量  */
};
```

File 类记录了进程打开文件的读写、请求类型、文件读写位置

OpenFiles 类维护当前进程所有打开文件，本文件系统为单用户单进程，所以一共只支持打开 15 个文件。

IOParameter 记录了对文件读写时的偏移量、字节数以及目标区域首地址。

4.1.6 OpenFileTable 类与 InodeTable 类

```
class OpenFileTable
{
public:
    /* static consts */
    //static const int NINODE    = 100;  /*  内存 Inode 的数量  */
    static const int NFILE   = 100;  /*  打开文件控制块 File 结构的数量  */
    /* Functions */
public:
    /* Constructors */
    OpenFileTable();
    /* Destructors */
```

```
    ~OpenFileTable();
        File* FAlloc();
        void CloseF(File* pFile);

        /* Members */
    public:
        File m_File[NFILE];          /* 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。*/
    };
```

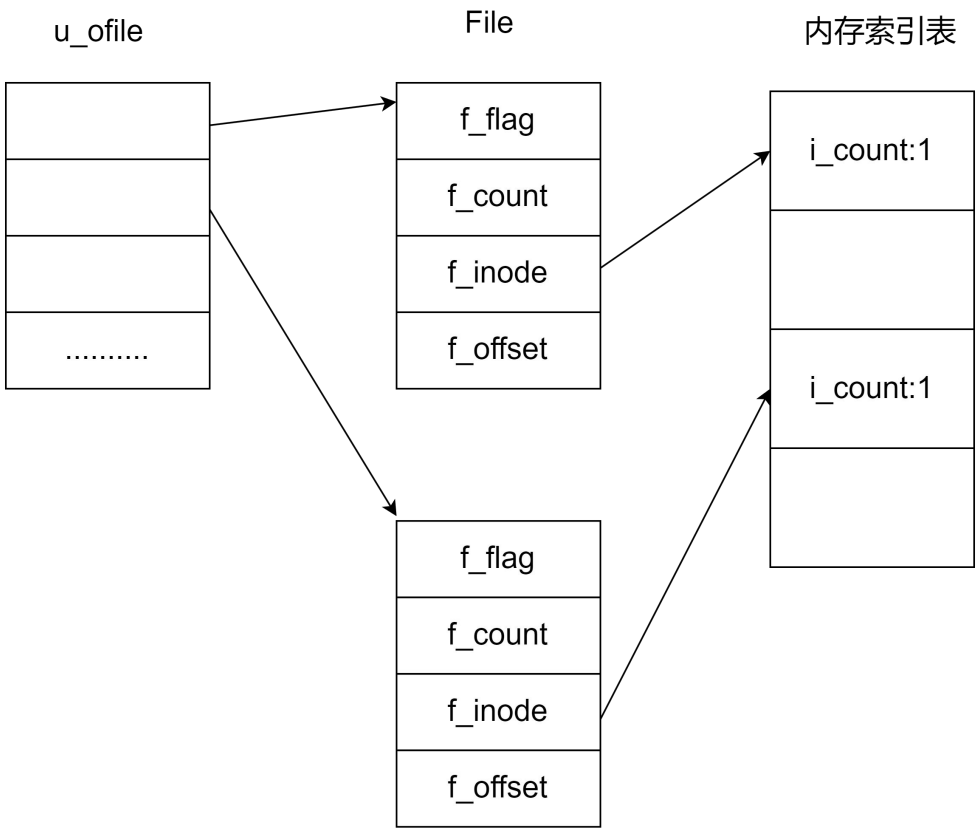
```
    class InodeTable
    {
        /* static consts */
    public:
        static const int NINODE = 100;   /* 内存 Inode 的数量 */

        /* Functions */
    public:
        /* Constructors */
        InodeTable();
        /* Destructors */
        ~InodeTable();
        void Initialize();
        Inode* IGet(int inumber);
        void IPut(Inode* pNode);
        void UpdateInodeTable();

        int IsLoaded(int inumber);
        Inode* GetFreeInode();
```

```
    /* Members */
public:
    Inode m_Inode[NINODE];      /* 内存 Inode 数组，每个打开文件都会占用一个内存 Inode */
    FileSystem* m_FileSystem;   /* 对全局对象 g_FileSystem 的引用 */
};
```

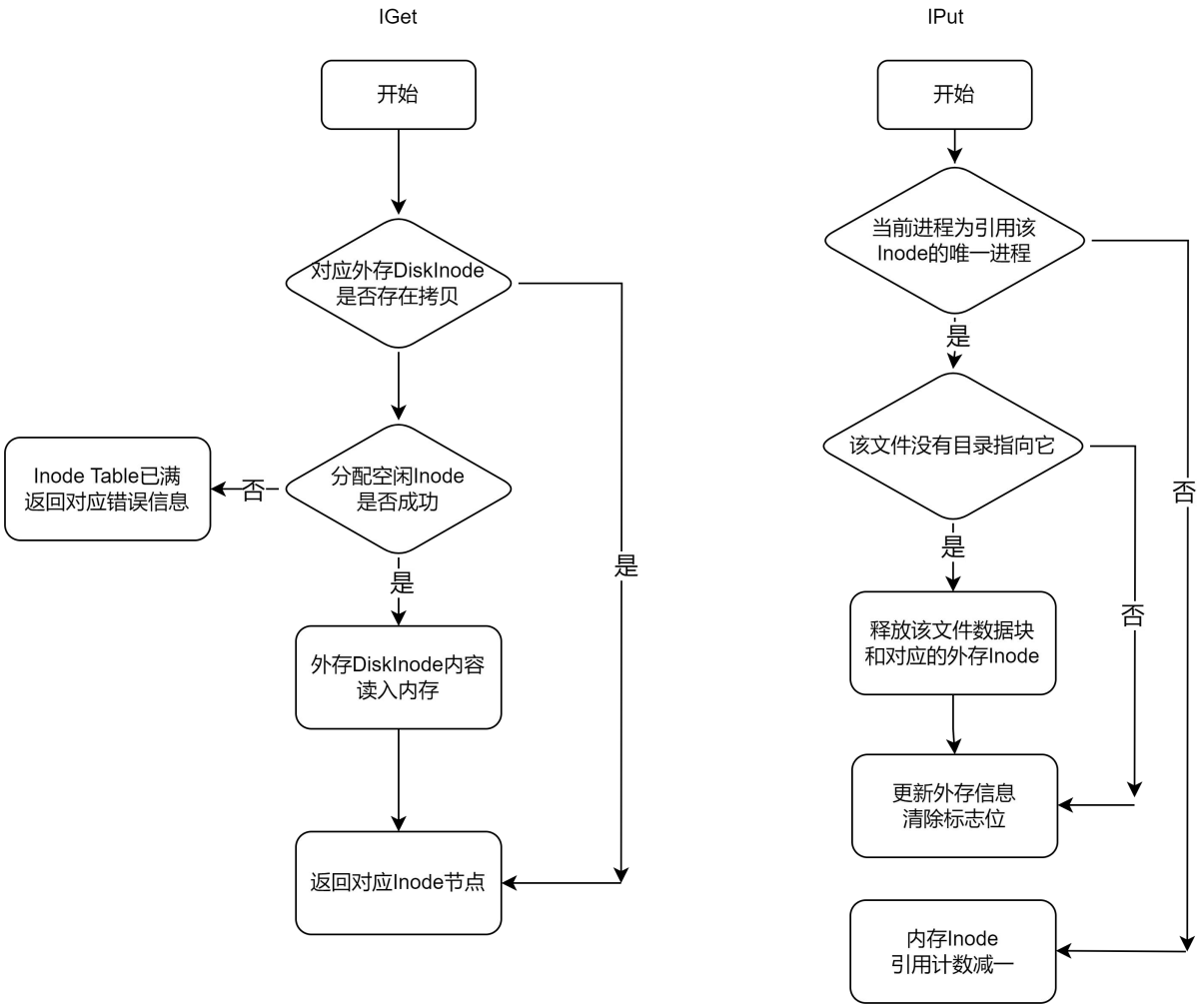
OpenFileTable 类负责内核中对打开文件机构的管理，为进程打开的文件建立内核数据结构之间的勾连关系。进程 User 区中的文件描述符指向打开文件表，由打开文件表指向内存 Inode 的位置，进而得到整个文件的信息。

InodeTable 类负责 Inode 的分配和释放。

内存打开结构如下：



Inode 数据块的分配与回收流程：



4.1.7 FileManger 类及 DirectoryEnty 类

```
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  */
    };
```

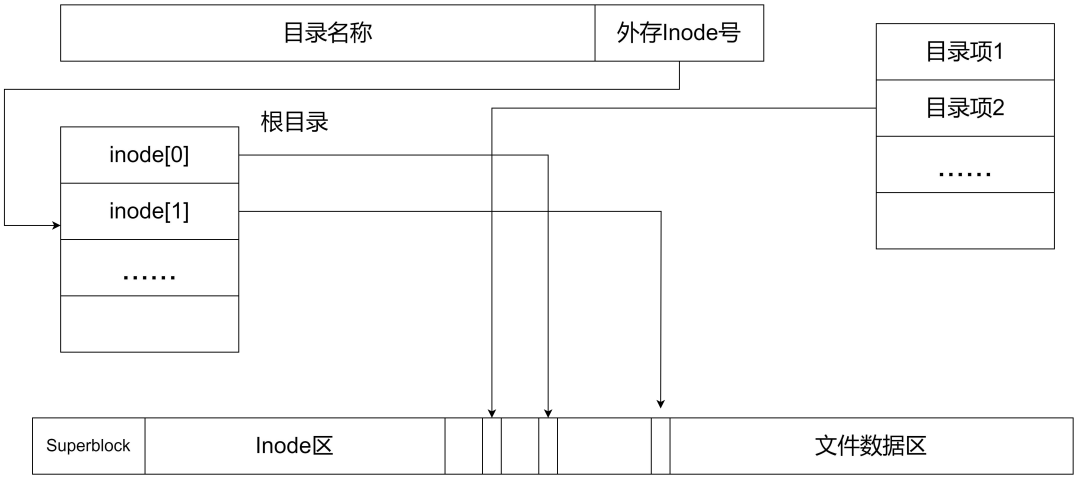
```
  /* Functions */
public:
    /* Constructors */
    FileManager();
    /* Destructors */
    ~FileManager();

    void Initialize();
    void Open();
    void Creat();
    void Open1(Inode* pInode, int mode, int trf);
    void Close();
    void Seek();
    void FStat();
    void Stat();
    /* FStat()和 Stat()系统调用的共享例程 */
    void Stat1(Inode* pInode, unsigned long statBuf);
    void Read();
    void Write();
    void Rdwr(enum File::FileFlags mode);
    Inode* NameI(char (*func)(), enum DirectorySearchMode mode);

    static char NextChar();

    Inode* MakNode(unsigned int mode);

    void WriteDir(Inode* pInode);

    void SetCurDir(char* pathname);

    int Access(Inode* pInode, unsigned int mode);
    /* 改变当前工作目录 */
    void ChDir();
    // /* 创建文件的异名引用 */
    // void Link();
```

```
    /*  取消文件  */
        void UnLink();
        /*  用于建立特殊设备文件的系统调用  */
        void MkNod();

    public:
        Inode* rootDirInode;
        FileSystem* m_FileSystem;
         InodeTable* m_InodeTable;
        OpenFileTable* m_OpenFileTable;
    };
```

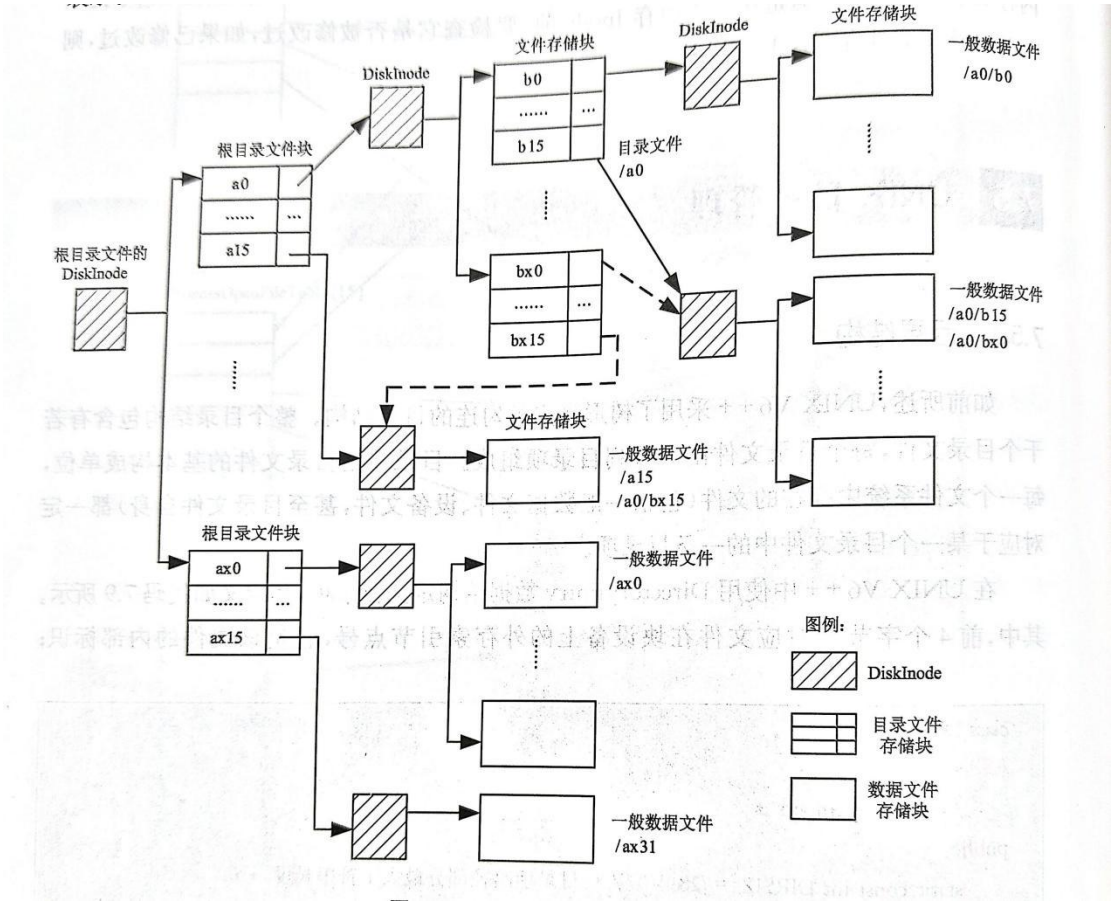
FileManger 类封装了对文件系统调用的核心操作包括文件打开、关闭、读写、文件指针移动等等，供顶层 API 模块调用。

DirectoryEnty 类定义了目录项的结构，由 4 个字节的 Inode 编号和 28 个字节的目录名称构成，一个盘块可保存 16 个目录项，文件系统中目录的存储结构如下：

```
    class DirectoryEntry
    {
        /* static members */
    public:
        static const int DIRSIZ = 28;    /*  目录项中路径部分的最大字符串长度  */
        /* Functions */
    public:
        /* Constructors */
        DirectoryEntry();
        /* Destructors */
        ~DirectoryEntry();
        /* Members */
    public:
        int m_ino;        /*  目录项中 Inode 编号部分  */
        char m_name[DIRSIZ];    /*  目录项中路径名部分  */
    };
```



UNIX V6++目录管理采用树形结构管理。首先根据根目录文件 Inode 获取目录文件块，再获取下级文件的 Inode，若 Inode 对应普通文件，则指向数据块；若为目录文件，则指向目录文件块：



打开目录流程：FileManager 类通过 NameI（）函数对目录路径进行分割递归查找，找到对应的目录位置，返回对应的 Inode；

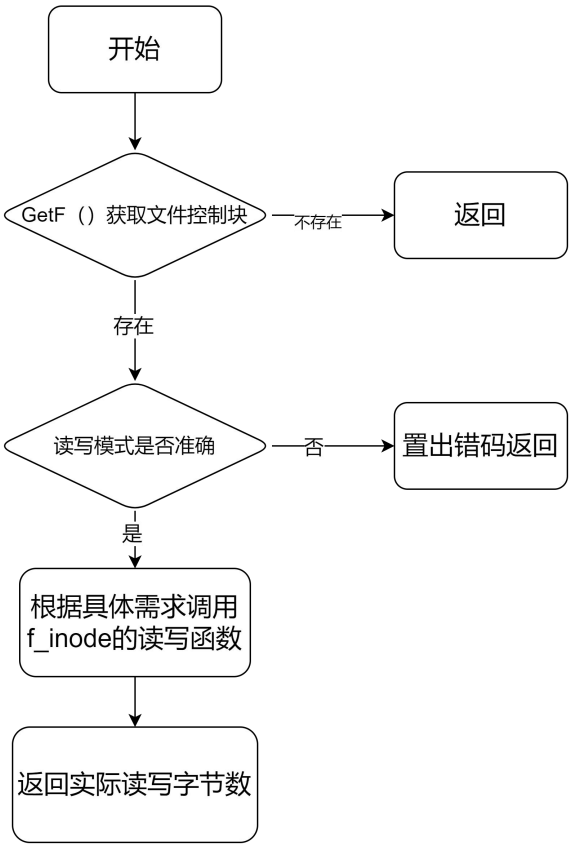
删除目录：首先确保目录名合法，其次确保目录存在，之后删除对应内存 Inode，更新位示图，当前目录。

文件读操作:外层 API 调用 Read()函数，根据用户空间中传来的相应参数进行文件读操作；

文件写操作：外层 API 调用 Write()函数，根据用户空间中传来的相应参数进行文件读操作；

文件指针移动操作：外层 API 调用 Seek()函数，根据用户空间中传来的相应参数进行文件读操作；

基本流程图：



4.1.8 User 类

User 类主要实现传参功能，作为外部 API 和文件系统内部数据的桥梁。

4.1.9 Kernel 类

Kernel 类封装所有内核想概念的全局实例副本，在内存中为单体模式，保证内核中封装各个内核模块对象只有一个副本，另外外层 API 调用文件系统内部数据结构也是以 Kernel 类为媒介。

4.1.10 Utility 类

公用函数类。

4.2 顶层接口设计

本文件系统将顶层结构封装成了 Command 类用于解析命令行命令并调用各种接口进行实现。

```
#define EXIT_CMD -1
#define OK 1

class Command{

    public:
    //分析处理命令
    int analyze(char*buf);

    //文件系统支持的命令

    void ls();
    void cd();
    void FFormat();
    void mkdir(char*dir_name);
    int Fcreat(char*file_name);
    int Fopen(char*file_name,int mode);
    void Fclose(int fd);
    int Fread(int fd,char*data,int len);
    int Fwrite(int fd,char* data,int len);
    int Flseek(int fd,int pos);
    int Fdelete(char*file_name);
    void Fin(char*out_file_name,char*in_file_name);
    void Fout(char*in_file_name,char*out_file_name);
    int   clear();
    void help();
    void exit();
};
```

五、运行分析

5.1 实验环境

- 操作系统：Windows 11
- 编译器选项：g++ (GCC) 3.4.5 (mingw-vista special r3)（同 UNIX V6++）
- 项目管理：makefile

5.2 使用说明

命令行中使用 make 命令重新编译各个源程序文件生成新的可执行文件 FileSystem.exe 或直接点击已有的 FileSystem.exe 进入文件系统，提示如下：

```
PS C:\Data\VSCodeData\Courses_Designment\FileSystem_Desgin\FileSystem\bin> .\FileSystem.exe
UNIX FILESYSTEM INITIALIZING...
INITIALIZING DONE!
Welcome!
You can type "help" to get more information about the filesystem!
root@LAPTOP_1228:~/$
```

我们可以输入`help`命令查看支持的命令：

```
root@LAPTOP_1228:~/$ help
fformat                                         - 格式化文件系统
mkdir       <dir name>                          - 创建目录
cd          <dir name>                          - 进入目录
ls                                              - 显示当前目录清单
fcreat      <file name>                         - 创建新文件
fdelete     <file name>                         - 删除文件
fopen       <file name>                         - 打开文件
fcolse      <file name>                         - 关闭文件
fread       <fd> <buffer> <length>              - 根据文件指针读文件
fwrite      <fd> <buffer> <length>              - 根据文件指针写文件
flseek      <fd> <position>                     - 调整文件指针位置
fin         <out_file_name> <in_file_name>      - 将外部文件读入文件系统
fout        <in_file_name> <out_file_name>      - 将内部文件读出文件系统
help                                            - 显示命令清单
clear                                           - 清屏
exit                                            - 退出系统
```

使用`exit`命令退出系统，`fformat`命令可以格式化文件系统。另外，可以使用`make clean`命令删除编译中间文件、img 文件即可执行程序。

5.3 功能测试

（1）创建 bin、etc、home、dev 目录：

```
root@LAPTOP_1228:~/$ mkdir bin
root@LAPTOP_1228:~/$ mkdir etc
root@LAPTOP_1228:~/$ mkdir home
root@LAPTOP_1228:~/$ mkdir dev
root@LAPTOP_1228:~/$ ls
bin
etc
home
dev
```

（3）在/home 目录中创建 texts、reports、photos 目录：

```
root@LAPTOP_1228:~/$ cd home
root@LAPTOP_1228:~/home$ mkdir texts
root@LAPTOP_1228:~/home$ mkdir reports
root@LAPTOP_1228:~/home$ mkdir photos
root@LAPTOP_1228:~/home$ ls
texts
reports
photos
```

(4) 创建临时文件 temp 并写入“hello world”

```
root@LAPTOP_1228:~/home$ fcreat temp
file temp create correct, return fd = 2
root@LAPTOP_1228:~/home$ fwrite 2 hello,world! 12
Write succesfully,write data length is 12 bytes
root@LAPTOP_1228:~/home$ fclose 2
root@LAPTOP_1228:~/home$ fopen temp
file temp open correct, return fd = 2
root@LAPTOP_1228:~/home$ fread 2 10
Read succesfully,read data length is 10 bytes
read data:
hello,worl
```

(5) 读写指针定位在文件开头再次读文件。

```
root@LAPTOP_1228:~/home$ flseek 2 0
File seek succesfully
root@LAPTOP_1228:~/home$ fread 2 100
Read succesfully,read data length is 12 bytes
read data:
hello,world!
```

（6）删除文件

```
root@LAPTOP_1228:~/home$ ls
texts
reports
photos
temp
root@LAPTOP_1228:~/home$ fdelete temp
root@LAPTOP_1228:~/home$ ls
texts
reports
photos
root@LAPTOP_1228:~/home$
```

（7）按照要求将课设报告，ReadMe.txt 和一张图片存入改文件系统，分别放在/home/texts, /home/reports 和/home/photots 文件夹

```
root@LAPTOP_1228:~/home$ cd texts
root@LAPTOP_1228:~/home/texts$ fin ../data/ReadMe.txt f_ReadMe.txt
file ../data/ReadMe.txt read succesfully to f_ReadMe.txt
```

```
root@LAPTOP_1228:~/home/reports$ fin ../data/report.pdf f_report.pdf
file ../data/report.pdf read succesfully to f_report.pdf
root@LAPTOP_1228:~/home/reports$ cd ..
root@LAPTOP_1228:~/home$ cd photos
root@LAPTOP_1228:~/home/photos$ fin ../data/test.jpg f_test.jpg
file ../data/test.jpg read succesfully to f_test.jpg
```

(8) 将这三种文件从文件系统读出并比较

```
root@LAPTOP_1228:~/home$ cd texts
root@LAPTOP_1228:~/home/texts$ ls
f_ReadMe.txt
root@LAPTOP_1228:~/home/texts$ fout f_ReadMe.txt ../data/f_ReadMe.txt
file f_ReadMe.txt write succesfully to ../data/f_ReadMe.txt
root@LAPTOP_1228:~/home/texts$ cd ..
root@LAPTOP_1228:~/home$ cd reports
root@LAPTOP_1228:~/home/reports$ ls
f_report.pdf
root@LAPTOP_1228:~/home/reports$ fout f_report.pdf ../data/f_report.pdf
file f_report.pdf write succesfully to ../data/f_report.pdf
root@LAPTOP_1228:~/home/reports$ cd ..
root@LAPTOP_1228:~/home$ cd photos
root@LAPTOP_1228:~/home/photos$ ls
f_test.jpg
root@LAPTOP_1228:~/home/photos$ fout f_test.jpg ../data/f_test.jpg
file f_test.jpg write succesfully to ../data/f_test.jpg
```

```
C:\Data\VSCodeData\Courses_Designment\FileSystem_Desgin\FileSystem\data>fc ReadMe.txt f_ReadMe.txt
正在比较文件 ReadMe.txt 和 F_README.TXT
FC: 找不到差异


C:\Data\VSCodeData\Courses_Designment\FileSystem_Desgin\FileSystem\data>fc test.jpg f_test.jpg
正在比较文件 test.jpg 和 F_TEST.JPG
FC: 找不到差异


C:\Data\VSCodeData\Courses_Designment\FileSystem_Desgin\FileSystem\data>fc report.pdf f_report.pdf
正在比较文件 report.pdf 和 F_REPORT.PDF
FC: 找不到差异
```

注意这里我在通过 fin,fout 读写文件时，因为我的本地文件存储在了./data 文件夹，而可执行程序存放在了./bin 文件夹，所以相应的添加了相对路径的参数，如果数据和 exe 文件在同一文件夹中则不需要额外添加。

5.4 错误提示测试

（1）命令大小写敏感，输入不正确的命令会提示

```
root@LAPTOP_1228:~/$ Cd bin
Cd: command not found
```

（2）尝试打开不存在的文件或目录会报错：

```
root@LAPTOP_1228:~/$ cd data
cd: data: No such file or directory
root@LAPTOP_1228:~/$ fopen a
file a open error,no such file or dirctory
```

（3）重复创建同名的目录或文件会报错：

```
root@LAPTOP_1228:~/$ mkdir bin
mkdir: cannot create directory 'bin': File exists
root@LAPTOP_1228:~/$ fcreat a
file a exist!
```

（4）读取不存在的 fd 会报错：

```
root@LAPTOP_1228:~/$ fread 100 10
fd value is out of limit
root@LAPTOP_1228:~/$ fread -1 10
fd value is out of limit
root@LAPTOP_1228:~/$ fread 5 10
No such or directory
```

（5）格式化时会多次提示

```
root@LAPTOP_1228:~/$ fformat
Are you sure to fromat the filesystem?(All data will be cleared in the disk)[y/n]
n
root@LAPTOP_1228:~/$
```

六、用户使用说明

6.1 依赖环境

- 编译器选项：g++ (GCC) 3.4.5 (mingw-vista special r3)（同 UNIX V6++）
- 项目管理：makefile

注意，这里的编译器必须是 32 位编译器，如果使用最新的 64 位编译器，那么可能会出现难以预料的错误。

6.2 使用说明

（1）之间点击已有的 exe 文件进入运行

（2）make 命令重新生成 exe 文件进入运行

已有的 exe 文件中目录结构已经创建完成，如果通过 make 重新生成或进行手动 format 命令，那么文件目录结构则为空。

（3）正常退出程序使用`exit`命令

七、总结

本次课程设计主要实现了一个类 UNIX V6++文件系统，深入探索了文件系统的关键组成部分，包括文件的创建、删除、打开、关闭以及读写等基本操作，实现了一个健壮的文件系统模型。通过这个项目，我们不仅理解了文件系统的内部工作机制，还实践了如何在现代操作系统中模拟传统 UNIX 文件系统的行为。

在实验中，通过使用 C++语言和面向对象的方法，我们设计了多个关键的数据结构和类，例如 SuperBlock、Inode、DirectoryEntry 等，每个类都承担了文件系统中的特定功能。特别是通过实现高速缓存管理和磁盘 I/O 操作，有效地提高了文件系统的读写效率。

此外，本课程设计还包括了文件系统的错误处理机制，能够对一些常见的错误进行诊断和处理，增强了系统的健壮性。通过命令行界面，用户可以直观地执行各种文件操作，并获取操作反馈，这使得文件系统的交互更为友好。

总的来说，这次课程设计不仅加深了对文件系统架构的理解，还提升了编程技能和问题解决能力。未来可以进一步探索文件系统的优化方案，例如实现更复杂的文件分配策略或者支持网络文件系统等功能，以适应更广泛的应用场景。