操作系统课程设计

类 UNIX 的二级文件系统

1550431 王甯琪

一、需求分析

使用一个普通的大文件模拟 UNIX V6++ 的一个文件卷，即把一个大文件当成一张磁盘来使用。这张磁盘中，以块为单位来储存信息，每个块大小为 512 字节。

S	inode区	文件数据区

需要定义磁盘的文件结构、SuperBlock 结构、INode 节点结构、INode 节点分配和回收算法、物理块的分配和回收、目录和文件索引等等。

本次课程设计海完成了磁盘高速缓存模块。

在完成文件系统的基础上，需要构建一些文件操作接口，如格式化文件卷、列目录、创建新文件、读写文件等等，以供用户使用。

在最终的课程设计成果中，需要先在主程序创建一些文件和文件夹，并将电脑中的一些文件写入二级文件系统。之后需要创建 /test/Jerry 文件，并进行读写测试。最后进入命令行界面，供用户手动输入指令进行使用和测试。

二、概要设计

磁盘设定为 65536 个物理块。首先可以将磁盘划分成三大区域，第一个区域为 SuperBlock 区域，这个区域主要记录 INode 节点及物理盘块的分配和使用情况；第二个区域为 INode 节点区，磁盘一共有 1024 个 INode 节点，每个节点对应于一个文件，其中记录了文件的大小、文件的索引情况、文件的状态等等；最后一个区域由剩下的盘块构成，用于记录文件内容。

具体的磁盘读写由 DeviceDriver 这个类负责，超级块的管理（包括 INode 及物理盘块的分配与回收）由 SuperBlockManager 这个类进行管理，INode 区域的读写由 INodeManager 类进行。

向磁盘读写数据并不是直接进行的，而是借由中间层 BufferManager 进行，这个中间层提供了高速缓存功能，在使用时不需要考虑某个物理块的内容是在内存中还是在磁盘中，只需要使用这个类暴露的接口即可以完成有效的文件操作。

文件的概念在 File 类中得到抽象，这个类提供了逻辑块到物理块的映射、向某个文件读或写数据、删除文件一部分内容的功能；直接提供更为抽象的文件操作接口的是 FileManager 这个类，这个类对 File 进行了更高层的封装，能够针对文件夹和普通文件进行不同的操作（如对文件夹可以增删目录项，对普通文件可以提供读写接口等）；直接供顶层使用的接口（如 fformat、ls、fread、

fwrite 等）由 FileOperator 提供，这个类通过 File 和 FileManager 提供的各个接口进行再次的封装和组合。

简而言之，不同的类之间或通过中间层进行抽象，或作为类成员，共同形成了整个二级文件系统。

2.1 SuperBlock 设计

超级块的类定义如下。

```
class SuperBlock {
public:
    int s_isize; // inode 总个数
    char s_ibitmap[IBITMAP_SIZE]; // 记录 inode 分配情况的位图
    int s_ilock; // 临界区锁

    int s_fsize; // 盘块总数
    int s_nfree; // 直接管理的空闲盘块数
    int s_free[100]; // 直接管理的空闲盘块的索引表
    int s_flock; // 临界区锁

    int s_dirty; // 指示是否需要写回磁盘

    char padding[472]; // 填充至 1024 字节

public:
    SuperBlock();
};
```

其中最主要的是记录了 INode 节点及物理盘块的使用情况。管理 INode 节点的方法是位图示法，即用 128 个字节公 128 * 8 = 1024 位对应于 1024 个 INode 节点，通过每一位是 0 还是 1 来表示对应的 INode 是空闲还是在使用中。对于物理盘块的管理，使用的是成组链接法，即用空闲盘块本身记录空闲的情况，不需要额外的存储空间。将空闲盘块划分成组，每组内部构成一个栈，每组之间构成一串链表。

2.2 SuperBlockManager 类

这个类提供了一些接口，能够根据超级块记录的信息分配和回收 INode 节点及物理盘块。

```
/**
 * 这个类提供了对 SuperBlock 管理的各方法
 * 提供格式化磁盘、分配外存 inode 节点、分配一个盘块等方法
 */
class SuperBlockManager {
private:
    DeviceDriver* DD;
    SuperBlock* SB;
    BufferManager* BM;

private:
    /**
     * 读取磁盘中的 SuperBlock
     */
    void loadSuperBlock();

    /**
     * 保存 SuperBlock 到磁盘
     */
    void saveSuperBlock();

    /**
     * 重置 SuperBlock 中空闲盘块的相关记录信息
     * 使用成组链接法
     */
    void resetFreeBlockInfo();

    /**
     * 为清空后的磁盘进行根目录创建工作
     * 创建外存 inode，并创建好相关 DirectoryEntry 项（如 . ..）
     */
    void createRootDir();
```

```
public:
    SuperBlockManager();
    ~SuperBlockManager();

    /**
     * 格式化整个磁盘
     */
    void formatDisk();

    /**
     * 检查一个外存 inode 是否已经分配
     * 传入 inode 节点标号
     */
    bool hasAllocedDINode(int ino);

    /**
     * 分配一个 disk inode 节点
     * 返回外存 inode 编号
     */
    int allocDiskINode();

    /**
     * 释放一个 disk inode 节点
     * 传入 inode 节点编号
     */
    void freeDiskINode(int no);

    /**
     * 分配一个 block 盘块
     * 返回一个缓存块的指针
     */
    Buffer* allocBlock();

    /**
     * 释放一个盘块
```

```
     * 传入盘块编号
     */
    void freeBlock(int no);
};
```

2.3 INode 节点

INode 节点在外存（即磁盘）上保存，同时在需要时需要读入内存。所以一共有两个 INode 结构，DiskINode 及 MemINode。

```
class DiskINode {
public:
    enum INodeFlags {
        IUPD = 0x2,          /* 内存 inode 被修改过，需要更新外存 inode */
        IACC = 0x4,          /* 内存 inode 被访问过，需要修改最近一次访问时间 */
        IALLOC = 0x8000,     /* 文件被使用 */
        IFDIR = 0x4000,      /* 文件类型：目录文件 */
        ILARG = 0x1000       /* 文件长度类型：大型或巨型文件 */
    };

    unsigned int d_mode; // 状态的标志位
    int d_nlink; // 文件在目录中不同路径的数量
    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]; // 文件的盘块的混合索引表
    int d_atime; // 最后访问时间
    int d_mtime; // 最后修改时间

public:
    DiskINode();
};

class MemINode {
public:
```

```
    enum INodeFlags {
        IUPD = 0x2,          /* 内存 inode 被修改过，需要更新外存 inode */
        IACC = 0x4,          /* 内存 inode 被访问过，需要修改最近一次访问时间 */
        IALLOC = 0x8000,     /* 文件被使用 */
        IFDIR = 0x4000,      /* 文件类型：目录文件 */
        ILARG = 0x1000       /* 文件长度类型：大型或巨型文件 */
    };

    unsigned int m_mode; // 状态的标志位
    int m_nlink; // 文件在目录中不同路径的数量
    short m_uid; // 文件所有者的用户标识数
    short m_gid;  // 文件所有者的组标识数
    int m_size; // 文件大小，字节为单位
    int m_addr[10]; // 文件的盘块的混合索引表
    int m_atime; // 最后访问时间
    int m_mtime; // 最后修改时间

    int m_number; // 外存对应的 inode 编号
    int m_count; // 共享的 file 结构数

    int m_lastr; // 最后一次读的块号

public:
    MemINode();

    /**
     * 读取外存的 inode
     * 传入一个外存 inode 对象
     */
    void readDiskINode(DiskINode dinode);

    /**
     * 将内存 inode 节点转换成外存 inode 节点
     */
    DiskINode getDiskINode();
};
```

2.4 INodeManager 类

这个类主要对 INode 节点的分配、写回等功能进行了封装，能够提供给其他类更为方便的接口来获取、使用、保存和销毁 INode 节点。

```
/**
 * 这个类的作用主要是用于管理内存 INode 节点
 * 需要创建、打开、更改、删除文件时，需要通过这个类获取节点
 * 这个类记录 INode 共享情况，即可能有多个文件共享一个 INode
 * 这个类提供 打开、创建、销毁 INode 的接口，也提供将 INode 写回外存的接口、从外存读取 INode 的接口
 */
class INodeManager {
private:
    set<MemINode*> Iset;
    BufferManager* BM;
    DeviceDriver* DD;
    FileSystem* FS;

public:
    INodeManager();
    ~INodeManager();

    /**
     * 根据外存的 inode 编号，找到并读取到内存中
     */
    MemINode* readDINode(int ino);

    /**
     * 分配一个新的内存 inode
     */
    MemINode* getNewMINode();

    /**
     * 释放一个内存 inode
     * 如果 inode 只被一个 File 结构使用，则从 iset 中清除
```

```
     * 如果有 inode 共享的情况，则不释放
     */
    void freeMINode(MemINode* pinode);

    /**
     * 写回一个内存 inode 到外存上
     */
    void writeBackMINode(MemINode* pinode);

    /**
     * 检查一个外存 inode 是否已经读入内存
     * 传入外存 inode 编号
     */
    bool hasLoadedDINode(int ino);

    /**
     * 在 Iset 中查找已经读入内存的外存 inode
     * 传入外存 inode 编号
     * 返回找到的 inode 指针
     */
    MemINode* getLoadedDINode(int ino);
};
```

2.5 DeviceDriver 类

这个类直接对磁盘进行读写，是直接和磁盘打交道的类。其他类都需要直接或间接调用这个类的方法来对磁盘进行操作。

```
/**
 * 这个类封装了一些对于磁盘文件读写的操作
 */
class DeviceDriver {
private:
    static const char* DISK_FILE_NAME;
    fstream fs;
```

```
public:
    DeviceDriver();
    ~DeviceDriver();

    /**
     * 检查虚拟磁盘文件是否存在
     */
    bool isExisting();

    /**
     * 将 buf 中的内容写入磁盘
     * offset 指字节数
     */
    void write(const char* buf, const int buf_size, const int offset);

    /**
     * 将磁盘中的内容读入 buf
     * offset 指字节数
     */
    void read(char* buf, const int buf_size, const int offset);
};
```

2.6 Buffer 及 BufferManager 类

Buffer 类描述的是一个缓存控制块。

```
class Buffer {
public:
    /* flags中标志位 */
    enum BufferFlag {
        B_WRITE = 0x1,      /* 写操作。将缓存中的信息写到硬盘上去 */
        B_READ = 0x2,       /* 读操作。从盘读取信息到缓存中 */
        B_DONE = 0x4,       /* I/O操作结束 */
        B_ERROR = 0x8,      /* I/O因出错而终止 */
```

```
        B_BUSY = 0x10,      /* 相应缓存正在使用中 */
        B_WANTED = 0x20,    /* 有进程正在等待使用该buf管理的资源，清B_BUSY标志
时，要唤醒这种进程 */
        B_ASYNC = 0x40,     /* 异步I/O，不需要等待其结束 */
        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 */
    };

    unsigned int b_flags; // 缓存控制块标志位

    Buffer* b_forw; // 前一个
    Buffer* b_back; // 后一个

    int b_wcount; // 需传送的字节数
    char* b_addr; // 指向该缓存控制块所管理的缓冲区的首地址
    int b_blk_no; // 磁盘逻辑块号

    Buffer();
    ~Buffer();
};
```

BufferManager 类中使用了 Buffer 类作为其成员。这个类提供了接口，能够分配和回收各缓存块。这个类相当于一个中间层，更为顶层的类使用这个类提供的功能进行物理盘块的读、写、延迟写。但是某一个物理块上的数据可能不会立刻写回磁盘上或是立刻从内存中清楚，而是留在内存中以便未来的重复访问。其他类不需要关心数据是怎么被储存以及何时被写回磁盘的。

```
/**
 * 这个类是对缓存块进行管理和调度的类
 * 提供了方法，能够利用高速缓存读写盘块
 */
class BufferManager {
private:
    DeviceDriver* DD;
```

```
    Buffer free_list;   // 自由队列
    Buffer buf[BUF_NUM]; // 缓存控制块
    char buffer[BUF_NUM][BUF_SIZE]; // 缓冲区
    map<int, Buffer*> map; // 记录每个 buf 与所读取的盘块号之间的对应关系

private:
    /**
     * 初始化所有缓存块
     */
    void initialize();

    /**
     * 从队列中摘取某个缓存块
     */
    void detachBuf(Buffer* buf);

    /**
     * 将一个缓存块加入队尾
     */
    void insertBuf(Buffer* buf);

public:
    BufferManager();
    ~BufferManager();

    /**
     * 申请一块缓存，用于读写磁盘上的块
     */
    Buffer* getBuf(int blk_no);

    /**
     * 释放缓存控制块 buf
     */
    void freeBuf(Buffer* buf);

    /**
```

```
     * 读磁盘的 blk_no 盘块
     */
    Buffer* readBuf(int blk_no);

    /**
     * 写一个盘块
     */
    void writeBuf(Buffer* buf);

    /**
     * 延迟写一个盘块
     */
    void dwriteBuf(Buffer* buf);

    /**
     * 清空这个 buf 中的内容
     */
    void clearBuf(Buffer* buf);

    /**
     * 将延迟写的缓存块中的内容全部写入磁盘
     */
    void flushAllBuf();
};
```

2.7 File 类

这个类对文件进行了第一次抽象。它通过组合使用其他类提供的方法，提供了对文件进行基本操作的接口，如进行逻辑盘块到物理盘块的映射、新增一个盘块、对文件进行读写、删除文件内容等。

```
/**
 * 这个类主要根据传入的 MemINode 节点打开一个文件
 * 通过调用这个类提供的方法，可以读写或删除文件内容
 */
```

```
class File {
public:
    enum FileType {
        FDIR = 0x1 // 是否是文件夹
    };

    FileSystem* FS;
    BufferManager* BM;

    int f_type;
    MemINode* f_minode;
    int f_offset;

private:
public:
    /**
     * 为文件新申请一个盘块
     * 返回相应缓存块
     */
    Buffer* applyNewBlk();

    /**
     * 将文件截断
     * 根据输入，保留文件前面若干字节
     */
    void trunc(const int size);

public:
    File(MemINode* minode);
    ~File();

    /**
     * 获取文件大小
     */
    int getFileSize();
```

```
    /**
     * 获取文件使用的盘块数
     */
    int getBlkNum();

    /**
     * 得到当前文件使用的最后一块盘块剩余容量
     */
    int getLastBlkRest();

    /**
     * 进行逻辑盘块号到物理盘块号的映射
     */
    int mapBlk(int lbn);

    /**
     * 删除文件的所有内容
     */
    void deleteAll();

    /**
     * 读取文件内容
     * 以当前 f_offset 为起始字节开始读取
     * 返回读取的字节数
     */
    int read(char* content, int length);

    /**
     * 将内容写入文件
     * 以当前 f_offset 为起始字节开始写入
     * 返回写入的字节数
     */
    int write(char* content, int length);

    /**
     * 删除文件中一部分内容
```

```
     * 以当前 f_offset 为起始开始删除
     * 删除的内容保存在 content 中（如果 content 不为 nullptr）
     * content 为 nullptr 则不保存删除的内容
     * 返回删除的字节数
     */
    int remove(char* content, int length);
};
```

2.8 FileManager 类

这个类对 File 进行了二次封装。这个类提供了对文件夹进行操作的方法，如向文件夹中新增一个目录项、删除一个目录项；也提供了新建文件夹和普通文件的方法；还可以对普通文件进行读写。这个类处理了比 File 更多的繁琐的细节，提供了更加简洁的接口。

当然，在这之前需要简单介绍一下目录项结构体。

```
struct DirectoryEntry {
public:
    char fname[28] = {0}; // 初始化为全0
    int ino;
};
```

```
#define ROOT_INO 0 // 根目录的 inode 编号
typedef map<string, int> FMAP; // 文件名到外存 inode 号的映射

/**
 * 通过这个类可以管理 File 类的对象
 * 这个类提供更便捷的方法操作文件夹目录项（如增删目录项，为目录项改名等）
 * 也提供便捷的方法操纵文件
 */
class FileManager {
private:
    FileSystem* FS;
    BufferManager* BM;
    INodeManager* IM; // 用于申请新的内存快
    File* file;
```

```
private:
    /**
     * 根据 inode 号找到并打开文件
     */
    File* openFile(int ino);

    /**
     * 删除当前目录下的某个文件项
     */
    void deleteItem(string fname);

public:
    FileManager(); // 默认打开根目录
    FileManager(int ino); // 打开 ino 对应的 inode 块对应的文件
    ~FileManager();

    /**
     * 检查文件是否为文件夹
     */
    bool isFolder();

    /**
     * 得到文件大小
     */
    int getSize();

    /**
     * 当前文件夹下是否有某名称的文件
     * ！调用前需检查当前文件是否为文件夹!
     */
    bool hasItem(string fname);

    /**
     * 统计文件夹中的目录项数量
     * ！调用前需检查当前文件是否为文件夹!
```

```
     */
    int countItems();

    /**
     * 加载文件夹中的目录项名称及对应 inode 号码
     * ！调用前需检查当前文件是否为文件夹!
     */
    FMAP* loadItems();

    /**
     * 如果文件夹下有某名称的文件，则返回它的外存 inode 编号
     * ！调用前需检查当前文件是否为文件夹!
     */
    int getDiskINodeNo(string fname);

    /**
     * 在当前目录下新增文件
     * 默认生成普通文件
     * 返回外存 inode 编号
     * ！调用前需检查当前文件是否为文件夹!
     */
    int createFile(string nfname, bool is_folder=false);

    /**
     * 在当前目录下删除普通文件
     * 不能删除文件夹!
     * ！调用前需检查当前文件是否为文件夹!
     */
    void deleteNormalFile(string fname);

    /**
     * 在当前目录下删除文件夹
     * 默认文件夹为空才能删除，否则不能删除
     * 传入 true 则先递归删除文件夹中所有内容再删除本身
     * ！调用前需检查当前文件是否为文件夹!
     */
```

```
    void deleteFolder(string fname, bool force_del=false);

    /**
     * 重命名当前目录下某文件
     * oname: old name, nname: new name
     * ! 调用前需检查当前文件是否为文件夹!
     */
    void renameFile(string oname, string nname);

    /**
     * 复制一个文件到新的地址
     * 第一个参数是当前目录下某文件的名字，第二个参数是指向目的地址的 FileManager
指针
     * 本质是做了勾连，共享一个外存 inode
     * ! 调用前需检查当前文件是否为文件夹!
     */
    void copyFile(string fname, FileManager* des_f);

    /**
     * 移动一个文件到新的地址
     * 第一个参数是当前目录下某文件的名字，第二个参数是指向目的地址的 FileManager
指针
     * ! 调用前需检查当前文件是否为文件夹!
     */
    void moveFile(string fname, FileManager* des_f);

    /**
     * 传入相对地址，返回对应文件的 inode 编号
     * ! 调用前需检查当前文件是否为文件夹!
     */
    int getFileIno(string addr);

    /**
     * 得到文件读写指针地址
     */
    int getFOffset();
```

```
    /**
     * 更改文件读写指针地址
     */
    void setFOffset(int noffset);

    /**
     * 读文件
     * 以当前文件读写指针为起始
     * 返回读取的字节数
     * !调用前需要检查当前文件是否为普通文件!
     */
    int read(char* content, int length);

    /**
     * 写文件
     * 以当前文件读写指针为起始
     * 返回写入的字节数
     * !调用前需要检查当前文件是否为普通文件!
     */
    int write(char* content, int length);

    /**
     * 删除文件的一部分
     * 以当前文件读写指针为起始
     * 删除的内容保存在 content 中（如果 content 不为 nullptr）
     * content 为 nullptr 则不保存删除的内容
     * 返回删除的字节数
     * !调用前需要检查当前文件是否为普通文件!
     */
    int remove(char* content, int length);
};
```

2.9 FileOperator 类

这是对文件进行最高抽象的一个类，它直接提供了诸如 fformat、ls、fopen、fclose、flseek、fread、fwrite、mkdir、fdelete、dirdelete 等方法。

```
/**
 * 直接向用户层提供创建、删除、写入文件/文件夹的方法
 */
class FileOperator {
private:
    FileSystem* FS;
    string cur_dir; // 当前的路径
    FileManager* cdir_fm; // 管理当前路径
    FileManager* ofile_fm; // 管理当前打开的文件

private:
    /**
     * 计算执行 cd come_to 后，路径变化情况
     */
    string getNewDir(string come_to);

public:
    FileOperator();
    ~FileOperator();

    /**
     * 当前是否打开文件
     */
    bool isOpeningFile() {
        return ofile_fm != nullptr;
    }

    /**
     * 格式化整个磁盘
     */
    void format();
```

```
/**
 * 获取当前路径
 */
string pwd();

/**
 * 列出当前路径下所有文件和文件夹
 */
set<string> ls();

/**
 * 在当前路径下创建新的文件夹
 */
int mkdir(string fld_name);

/**
 * 在当前路径下创建新文件
 */
int fcreate(string f_name);

/**
 * 打开文件
 */
int fopen(string f_name);

/**
 * 关闭文件
 */
void fclose();

/**
 * 读取文件
 * 返回读取的字节数
 */
int fread(char* buf, int length);
```

```
/**
 * 写入文件
 * 返回写入的字节数
 */
int fwrite(char* buf, int length);

/**
 * 更改文件读写的 offset
 */
void flseek(int noffset);

/**
 * 返回当前 offset
 */
int curseek();

/**
 * 查看文件大小
 */
int fsize();

/**
 * 删除文件
 */
int fdelete(string f_name);

/**
 * 删除目录（递归）
 */
int dirdelete(string fld_name);

/**
 * 更换路径
 */
int cd(string fld_name="");
```

```
    /**
     * 复制
     */
    int cp(string f_name, string des_name);

    /**
     * 移动
     */
    int mv(string f_name, string des_name);

    /**
     * 回到根目录
     */
    void goroot();
};
```

2.10 UserInterface 类

最后介绍一下处理用户输入的类。这个类接收用户在命令行中的输入，进行分析并调用 FileOperator 的方法。

```
class UserInterface {
private:
    FileOperator* FO;

private:
    vector<string> splitString(const string &s, const string &seperator);

public:
    UserInterface(FileOperator* pFO);
    ~UserInterface();

    /**
     * 处理用户输入
```

```
     */
    void processInput(string input);

    /**
     * 开始工作
     */
    void start();
};
```

三、详细设计

在这个部分，将对概要设计中部分重要、关键的思想和算法进行说明。

3.1 外存 INode 分配与回收

```
/**
 * 分配一个 inode 节点
 * 返回外存 inode 编号
 */
int SuperBlockManager::allocDiskINode() {
    const int bitmap_size = IBITMAP_SIZE;
    int i, j;

    bool find = false;
    int ino = -1;
    const unsigned char one = 1;
    char line;
    for(i = 0; i < bitmap_size && !find; i++) {
        line = (SB -> s_ibitmap)[i];
        for(j = 0; j < 8; j++) {
            if(!((one << j) & line)) { // 如果某一位为 0
                find = true;
                ino = 8 * i + j;

                SB -> s_dirty = 1;
```

```
                (SB -> s_ibitmap)[i] |= (one << j);
                break;
            }
        }
    }

    if(ino == -1) {
        cout << "外存 inode 节点用尽" << endl;
    }

    return ino;
}
```

```
/**
 * 释放一个 inode 节点
 * 传入 inode 节点编号
 */
void SuperBlockManager::freeDiskINode(int no) {
    const unsigned char one = 1;
    int row, col;

    row = no / 8;
    col = no % 8;
    SB -> s_dirty = 1;
    (SB -> s_ibitmap)[row] &= ~(one << col);
}
```

申请和回收外存 INode 节点时，都是通过检查、搜索保存在超级块中的位示图来进行的。比较简单。

3.2 物理盘块的分配与回收

```
/**
 * 分配一个 block 盘块
 * 返回一个缓存块的指针
```

```
 */
Buffer* SuperBlockManager::allocBlock() {
    Buffer* buf;
    int i;

    SB -> s_dirty = 1;
    int blk_no = (SB -> s_free)[SB -> s_nfree - 1];
    SB -> s_nfree -= 1;

    if(SB -> s_nfree == 0) {
        int content[101];
        buf = BM -> readBuf(blk_no);

        // DD -> read((char*)content, sizeof(content), 512*blk_no);
        for(i = 0; i < 101; i++) {
            content[i] = ((int*)buf -> b_addr)[i];
        }

        SB -> s_nfree = content[0];
        for(int i = 0; i < 100; i++) {
            (SB -> s_free)[i] = content[i+1];
        }

        BM -> freeBuf(buf);
    }

    // cout << blk_no << endl;
    buf = BM -> getBuf(blk_no);
    return buf;
}
```

```
/**
 * 释放一个盘块
 * 传入盘块编号
 */
```

```
void SuperBlockManager::freeBlock(int no) {
    Buffer* buf;
    SB -> s_dirty = 1;

    if(SB -> s_nfree == 100) {
        buf = BM -> getBuf(no);

        int content[101];
        content[0] = SB -> s_nfree;
        for(int i = 0; i < 100; i++) {
            content[i+1] = (SB -> s_free)[i];
        }
        // DD -> write((char*)content, sizeof(content), 512*no);
        BM -> writeBuf(buf);

        SB -> s_nfree = 0;
    }

    (SB -> s_free)[SB -> s_nfree] = no;
    SB -> s_nfree += 1;
}
```

分配和回收物理块本质就是对由一个个小的栈组成的链表进行维护。超级块记录这个链表的第一项，每次增减新的空闲块都是对超级块内记录的 100 个数据进行修改。当超级块记录的数据超过

100 项时，则将前 100 项形成一个新的栈插入链表中；如果超级块记录的数据不足 0 项，则取出链表中的头结点，填入超级块中。

3.3 高速缓存的相关算法

在 BufferManager.h 文件中，规定了整个文件系统一共有 100 个缓存控制块，每个块有对应的 512 字节的空间供读写。

以下是用于获取缓存块的算法。

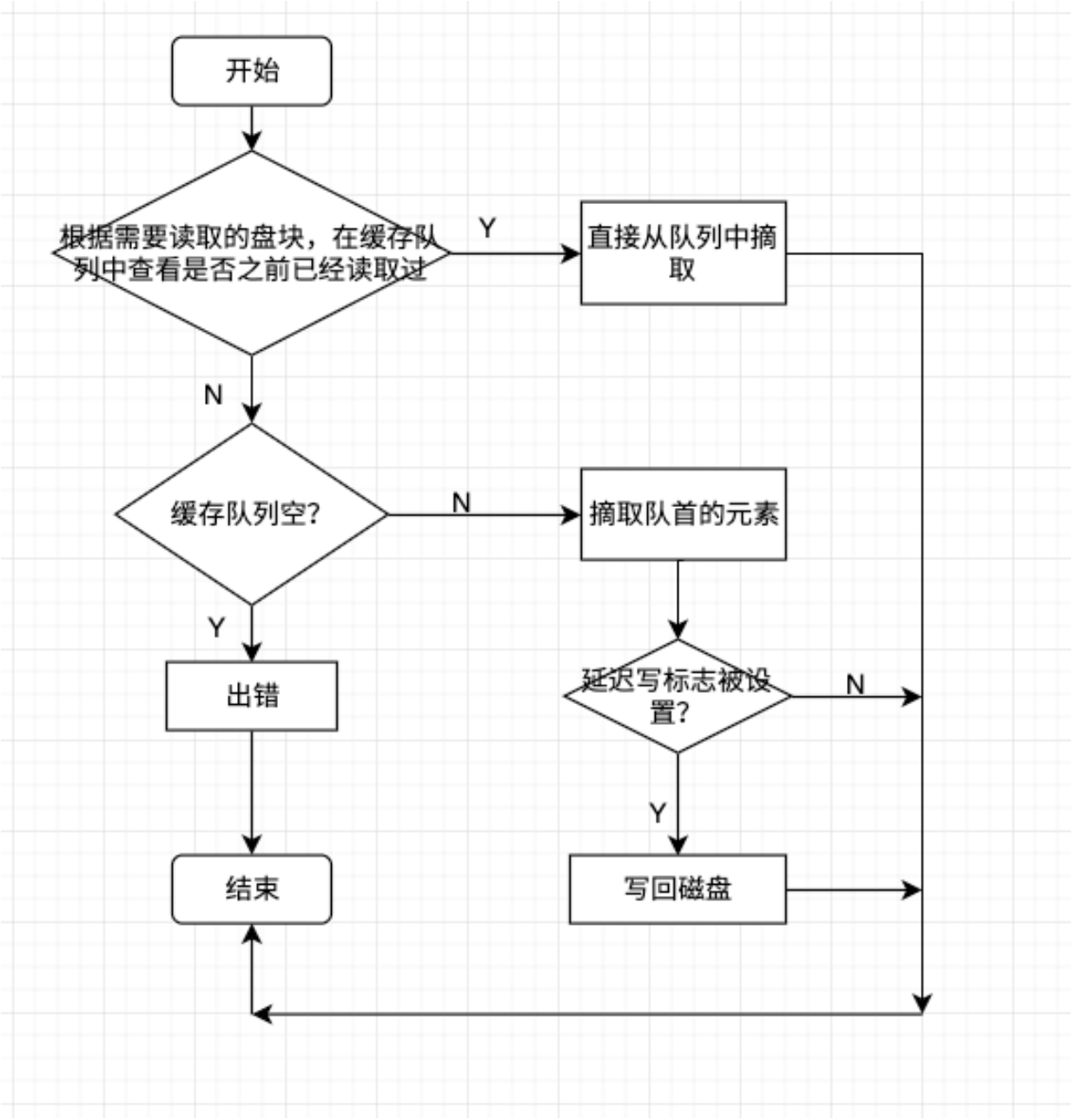
```
/**
 * 申请一块缓存，用于读写磁盘上的块
 */
Buffer* BufferManager::getBuf(int blk_no) {
    Buffer* buf;

    if(map.find(blk_no) != map.end()) { // 如果之前有某 buf 读取过这个盘块
        buf = map[blk_no];
        detachBuf(buf);
        return buf;
    }

    if(free_list.b_back == &free_list) { // 如果队列空了
        cout << "Error of buf" << endl;
        return nullptr;
    }

    buf = free_list.b_back;
    detachBuf(buf);

    if(buf -> b_flags & Buffer::B_DELWRI) { // 如果设置了延迟写
        DD -> write(buf -> b_addr, BUF_SIZE, buf -> b_blk_no * 512);
    }

    map.erase(buf -> b_blk_no); // 删除这块缓存之前读的盘块的记录
    map[blk_no] = buf; // 在记录中加上新的读取记录
    buf -> b_blk_no = blk_no;
```

```
    buf -> b_flags &= ~(Buffer::B_DELWRI | Buffer::B_DONE); // 消除之前的标志位

    return buf;
}
```

算法思想非常朴素。在访问某一块盘块时，很有可能之前已经访问过。那么在上次访问之后，如果内存中的盘块的数据仍然还完整且正确，那么可以直接从缓存队列中将之前用过的缓存控制块摘取出来，而不用再读磁盘。

在分配时，优先从队首摘取。在释放时，将缓存控制块插入队尾。这就自动完成了 LRU 算法。

同样需要注意的是，之前使用这个缓存控制块时，可能在释放前设置了延迟写标志位。那么当取用之前，需要将数据先写回磁盘，才能开始使用。



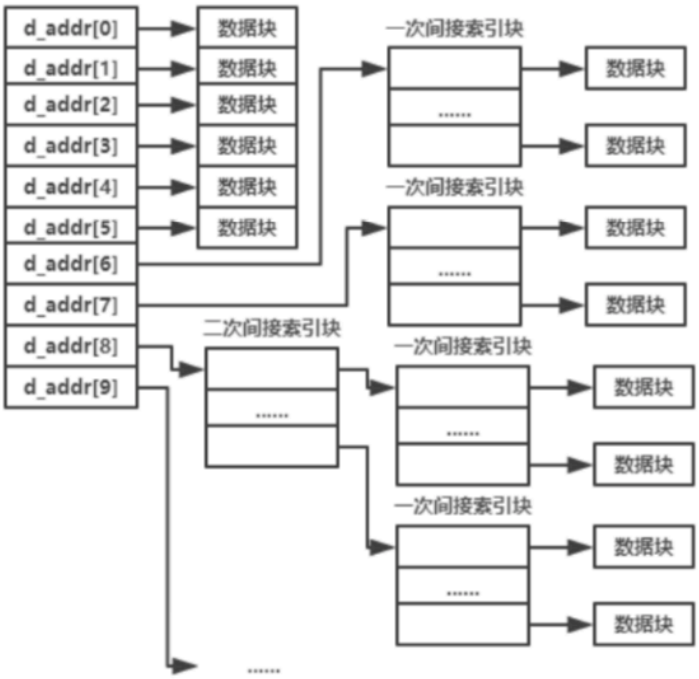
接下来是利用 Buffer 进行读写的相关代码。

```
/**
 * 读磁盘的 blk_no 盘块
 */
Buffer* BufferManager::readBuf(int blk_no) {
    Buffer* buf = getBuf(blk_no);
    if(buf -> b_flags & (Buffer::B_DELWRI | Buffer::B_DONE)) {
        // 如果某个块是之前读取过相关盘块，或是设置了延迟写，则直接返回
        return buf;
    }

    DD -> read(buf -> b_addr, BUF_SIZE, buf -> b_blk_no * 512);
    return buf;
}

/**
 * 写一个盘块
 */
void BufferManager::writeBuf(Buffer* buf) {
    buf -> b_flags &= ~Buffer::B_DELWRI; // 清除延迟写标志位
    DD -> write(buf -> b_addr, BUF_SIZE, buf -> b_blk_no * 512);
    buf -> b_flags |= Buffer::B_DONE;

    freeBuf(buf);
}

/**
 * 延迟写一个盘块
 */
void BufferManager::dwriteBuf(Buffer* buf) {
    buf -> b_flags |= Buffer::B_DELWRI;
    buf -> b_flags |= Buffer::B_DONE;
```

```
    freeBuf(buf);
}
```

3.4 逻辑盘块到物理盘块的映射算法

盘块的信息记录在 INode 节点的用于索引的数组中。其中前 6 块是直接索引，之后 2 块是一级间接索引，最后 2 块是二级间接索引。




```
/**
 * 进行逻辑盘块号到物理盘块号的映射
 */
int File::mapBlk(int lbn) {
    int bn;
    const int* addr = f_minode -> m_addr;
    const int fsize = f_minode -> m_size;
    const int blk_num = getBlkNum();

    if(lbn > blk_num || lbn < 0) {
        cout << "文件没有第 " << lbn << " 个逻辑块" << endl;
```

```
        return -1;
    }

    if(lbn < SMALL_FILE_MOST_BLK) { // 0 - 5 个盘块，不需要索引
        return addr[lbn];
    } else if(lbn < LARGE_FILE_MOST_BLK) { // (6 - 6 + 128 * 2) 个盘块，
一次间接索引
        const int indexed_blk_num = lbn + 1 - 6; // 除去前六个盘块的盘块数
        Buffer* index_blk; // 一级索引块
        if(indexed_blk_num > 128) {
            index_blk = BM -> readBuf(addr[7]);
        } else {
            index_blk = BM -> readBuf(addr[6]);
        }

        const int inner_index_blk_no = 1 + (indexed_blk_num - 1) % 128;
// lbn 对应的物理盘块号储存在一级索引块中的位置
        bn = ((int*)(index_blk -> b_addr))[inner_index_blk_no-1];
        BM -> freeBuf(index_blk);
    } else if(lbn < HUGE_FILE_MOST_BLK){ // 需要二次间接索引
        const int indexed_blk_num = lbn + 1 - 6 - 128 * 2; // 除去前面不需
要索引及一级索引的盘块，需要进行索引的盘块数
        Buffer* index1_blk;
        Buffer* index2_blk;
        if(indexed_blk_num > 128*128) {
            index1_blk = BM -> readBuf(addr[9]);
        } else {
            index1_blk = BM -> readBuf(addr[8]);
        }

        const int inner_index1_blk_no = 1 + ((indexed_blk_num - 1) / 128)
% 128; // 二级索引块在第一级索引块内的编号
        const int index2_blk_no = ((int*)(index1_blk ->
b_addr))[inner_index1_blk_no-1]; // 找出二级索引块的物理盘块编号
        index2_blk = BM -> readBuf(index2_blk_no);
```

```
        const int inner_index2_blk_no = 1 + (indexed_blk_num - 1) % 128;
// lbn 号盘块在第二级索引块内的编号
        bn = ((int*)(index2_blk -> b_addr))[inner_index2_blk_no-1];

        BM -> freeBuf(index1_blk);
        BM -> freeBuf(index2_blk);
    } else {
        cout << "文件过大, " << "检查 File.cc 中的 mapBlk 函数" << endl;
        return -1;
    }

    return bn;
}
```

先按 INode 块中储存的文件大小计算出所有块数。再根据块数大小判断某一个逻辑块要进行几次索引才能对应到物理盘块。之后通过高速缓存模块读取索引块，并找到对应物理块号。

3.5 文件读写算法

读写有些相似，这里以读举例。

```
/**
 * 读取文件内容
 * 以当前 f_offset 为起始字节开始读取
 * 返回读取的字节数
 */
int File::read(char* content, int length) {
    const int rest_byte_num = max(getFileSize() - f_offset, 0); // 文件剩余的大小
    const int toread_byte_num = min(rest_byte_num, length);

    int i;
    int read_byte_count = 0;
    int has_read_blk_num = f_offset / 512; // 已经读过的块数（逻辑块），根据 f_offset 确定
```

```
    int blk_rest_byte_num = 512 - (f_offset - has_read_blk_num * 512); // 在当前盘块内，当前文件指针后，还剩余多少字节未读取

    while(read_byte_count < toread_byte_num) {
        if(blk_rest_byte_num == 0) {
            blk_rest_byte_num = 512;
            has_read_blk_num++;
        }

        const int rest_toread_byte_num = toread_byte_num - read_byte_count; // 剩余待读取字节数
        if(blk_rest_byte_num >= rest_toread_byte_num) { // 如果当前块剩余字节数大于剩余所需读取字节数
            // 只需要读取本块，便可以完成本轮读取
            const int bn = mapBlk(has_read_blk_num);
            Buffer* buf = BM -> readBuf(bn);
            const char* addr = buf -> b_addr;

            // cout << "读取 " << has_read_blk_num << " 块盘块，" << rest_toread_byte_num << " 字节" << endl;
            for(i = 0; i < rest_toread_byte_num; i++) {
                content[read_byte_count] = addr[f_offset % 512 + i];
                read_byte_count++;
            }

            f_offset += rest_toread_byte_num;
            break;
        } else {
            // 需要读取本块剩余所有字节
            const int bn = mapBlk(has_read_blk_num);
            Buffer* buf = BM -> readBuf(bn);
            const char* addr = buf -> b_addr;

            // cout << "读取 " << has_read_blk_num << " 块盘块，" << blk_rest_byte_num << " 字节" << endl;
            for(i = 0; i < blk_rest_byte_num; i++) {
```

```
                content[read_byte_count] = addr[f_offset % 512 + i];
                read_byte_count++;
            }

            f_offset += blk_rest_byte_num;
            blk_rest_byte_num = 0;
        }
    }

    return toread_byte_num;
}
```

首先需要确定到底能读多少字节，因为在读写指针之后的长度可能比用户传入需要读取的字节数少。具体读出的字节数是两者之间小的那一个。

确定好需要读出多少字节后，先需要计算当前读写指针所在的逻辑块还有多少没有读完，现将这一部分数据读出。读完后，计算有多少完整的块需要被读出。最后，计算最后一个待读的逻辑块需要读出多少字节。

写与之类似。唯一要注意的是，写操作可能会增加文件长度。

3.6 文件夹目录项相关操作

先看看如何找到文件夹内部内所有文件。

```
/**
 * 加载文件夹中的目录项名称及对应 inode 号码
 * ! 调用前需检查当前文件是否为文件夹!
 */
FMAP* FileManager::loadItems() {
    int i;
    const int item_num = countItems();
    const int f_size = getSize();
    char* read_buf = new char[f_size];
    file -> f_offset = 0;
    file -> read(read_buf, f_size);
```

```
    FMAP* fm = new FMAP();
    DirectoryEntry* items = (DirectoryEntry*)(read_buf);
    for(i = 0; i < item_num; i++) {
        string fname = items[i].fname;
        (*fm)[fname] = items[i].ino;
    }
    return fm;
}
```

简单来说，就是利用先前 File 类暴露的读方法，读出该文件的所有内容，将其储存在一个 buf 中。之后将这些内容拆分成一个个 DirectoryEntry 对象，就可以看到该文件夹内有哪些文件，它们的名字和对应的 INode 号分别是多少。

再看看如何在文件夹内新建文件。

```
/**
 * 在当前目录下新增文件
 * 默认生成普通文件
 * 返回外存 inode
 * ! 调用前需检查当前文件是否为文件夹!
 */
int FileManager::createFile(string nfname, bool is_folder) {
    if(nfname.length() > 27) {
        cout << "文件名 " << nfname << " 过长" << endl;
        return -1;
    }

    if(hasItem(nfname)) {
        cout << "文件 " << nfname << " 已存在" << endl;
        return -1;
    }

    MemINode* new_minode = IM -> getNewMINode();
    const int dino = new_minode -> m_number; // 在外存 inode 的编号
```

```
    if(is_folder) {
        new_minode -> m_mode |= MemINode::IFDIR; // 设置成文件夹
        File f(new_minode);
        // 为新的文件夹新建 . .. 两项
        DirectoryEntry dir[2];
        memcpy(dir[0].fname, ".", 2);
        memcpy(dir[1].fname, "..", 3);
        dir[0].ino = dino;
        dir[1].ino = file -> f_minode -> m_number;
        f.f_offset = 0;
        f.write((char*)dir, sizeof(dir));
    }
    IM -> freeMINode(new_minode);

    DirectoryEntry new_item;
    new_item.ino = dino;
    int i;
    for(i = 0; i < nfname.length(); i++) {
        new_item.fname[i] = nfname[i];
    }
    new_item.fname[i] = '\0'; // 添加尾 0

    file -> f_offset = getSize();
    file -> write((char*)&new_item, sizeof(DirectoryEntry));
    return dino;
}
```

首先是分配一个新的 INode 节点给新创建的文件。之后把新文件名和分配的外存 INode 节点编号写到此文件夹对应文件内容之后，即增加一个目录项。

如果新的文件是文件夹类型的，则还需要分配 “.” 及 “..” 两个目录项，分别对应到自身及父文件夹的 INode 节点编号。

通过地址查找文件的过程，本质就是一次次的目录索引。

```
/**
 * 传入相对地址，返回对应文件的 inode 编号
 * ！调用前需检查当前文件是否为文件夹！
 */
int FileManager::getFileIno(string addr) {
    string s(addr);
    vector<string> sv;
    vector<string>::const_iterator it;

    int cur_ino = (file -> f_minode) -> m_number;

    sv = split(s, "/");
    for(it = sv.begin(); it != sv.end(); it++) {
        FileManager tFM(cur_ino);

        if(tFM.hasItem(*it) && tFM.isFolder()) {
            cur_ino = tFM.getDiskINodeNo(*it);
       } else {
            return -1;
        }
    }
   return cur_ino;
}
```

将地址按照斜线分开，形成一个数组。接下来进行一个循环的过程，即从当前目录开始，调用之前查看文件夹下所有目录项的函数，得到文件夹所有目录项，再查看其中是否有下一级文件名，如果有的话则将当前目录改为下一级文件。重复以上过程，直到遍历完整个数组。

3.7 处理用户输入

这个部分较为简单，举几个例子。

```
    auto sv = splitString(input, " ");
    const int para_amount = sv.size() - 1; // 参数个数
    const string cmd = sv[0];
```

```
    for(int i = 0; i < sv.size(); i++) {
        const string s = sv[i];
        auto pos = s.find("/");
        if(pos != string::npos) {
            cout << "目前不支持多层路径，如 cd ./folder/f1/" << endl;
            cout << "请使用其他指令代替，如 cd folder, cd f1" << endl;
            return;
        }
    }

    if(cmd == "format") {
        if(para_amount == 0) {
            FO -> format();
            cout << "格式化成功" << endl;
        } else {
            cout << "format 不能接受参数" << endl;
        }
        return;
    }

    if(cmd == "pwd") {
        if(para_amount == 0) {
            cout << FO -> pwd() << endl;
        } else {
            cout << "pwd 不能接受参数" << endl;
        }
        return;
    }


    if(cmd == "ls") {
        if(para_amount == 0) {
            auto lsr = FO -> ls();
            for(auto it = lsr.begin(); it != lsr.end(); it++) {
                cout << *it << endl;
            }
```

```
        } else {
            cout << "ls 不能接受参数" << endl;
        }
        return;
    }

    if(cmd == "mkdir") {
        if(para_amount == 1) {
            int res = FO -> mkdir(sv[1]);
            if(res == -1) {
                cout << "存在同名文件! " << endl;
            }
        } else {
            cout << "mkdir 接受 1 个参数" << endl;
        }
        return;
    }

    if(cmd == "fcreate") {
        if(para_amount == 1) {
            int res = FO -> fcreate(sv[1]);
            if(res == -1) {
                cout << "存在同名文件! " << endl;
            }
        } else {
            cout << "fcreate 接受 1 个参数" << endl;
        }
        return;
    }
```

先将用户在命令行中的输入按空格分开。之后根据命令类型分类，再判断参数个数是否正确，最后调用相关函数。

四、调试分析

整个测试逻辑是完成一个小组件后就测试小组件的功能；小组件拼凑成大的组件后再进行一次功能测试。

在程序中进行适当的输出是一个非常好的调试和检查方法。

比如在读文件的代码中插入输出提示，如下。

```
/**
 * 读取文件内容
 * 以当前 f_offset 为起始字节开始读取
 * 返回读取的字节数
 */
int File::read(char* content, int length) {
    const int rest_byte_num = max(getFileSize() - f_offset, 0); // 文件剩余的大小
    const int toread_byte_num = min(rest_byte_num, length);

    int i;
    int read_byte_count = 0;
    int has_read_blk_num = f_offset / 512; // 已经读过的块数（逻辑块），根据 f_offset 确定
    int blk_rest_byte_num = 512 - (f_offset - has_read_blk_num * 512); // 在当前盘块内，当前文件指针后，还剩余多少字节未读取

    while(read_byte_count < toread_byte_num) {
        if(blk_rest_byte_num == 0) {
            blk_rest_byte_num = 512;
            has_read_blk_num++;
        }

        const int rest_toread_byte_num = toread_byte_num - read_byte_count; // 剩余待读取字节数
        if(blk_rest_byte_num >= rest_toread_byte_num) { // 如果当前块剩余字节数大于剩余所需读取字节数
            // 只需要读取本块，便可以完成本轮读取
            const int bn = mapBlk(has_read_blk_num);
```

```
            Buffer* buf = BM -> readBuf(bn);
            const char* addr = buf -> b_addr;

            cout << "读取 " << has_read_blk_num << " 块盘块, " <<
rest_toread_byte_num << " 字节" << endl;
            for(i = 0; i < rest_toread_byte_num; i++) {
                content[read_byte_count] = addr[f_offset % 512 + i];
                read_byte_count++;
            }

            f_offset += rest_toread_byte_num;
            break;
        } else {
            // 需要读取本块剩余所有字节
            const int bn = mapBlk(has_read_blk_num);
            Buffer* buf = BM -> readBuf(bn);
            const char* addr = buf -> b_addr;

            cout << "读取 " << has_read_blk_num << " 块盘块, " <<
blk_rest_byte_num << " 字节" << endl;
            for(i = 0; i < blk_rest_byte_num; i++) {
                content[read_byte_count] = addr[f_offset % 512 + i];
                read_byte_count++;
            }

            f_offset += blk_rest_byte_num;
            blk_rest_byte_num = 0;
        }
    }

    return toread_byte_num;
}
```

那么在进行读文件时，就能看到每个块的读取情况。

```
[/] # cd home
读取 0 块盘块，224 字节
读取 0 块盘块，224 字节
[/home] # ls
读取 0 块盘块，160 字节
.
..
photos
reports
texts
[/home] # cd texts
读取 0 块盘块，160 字节
读取 0 块盘块，160 字节
[/home/texts] # ls
读取 0 块盘块，96 字节
.
..
text.txt
[/home/texts] # fopen text.txt
读取 0 块盘块，96 字节
读取 0 块盘块，96 字节
[/home/texts] $file opened$ #
```

还可以在代码中加入一些条件分支语句，这些语句是正常执行不可能执行到的，那么如果发现这些语句被执行了，就可以定位到错误，如下。

```
if(pre_blk_num < SMALL_FILE_MOST_BLK) { // 0 - 5 个盘块，不需要索引
   …
} else if(pre_blk_num < HUGE_FILE_MOST_BLK){ // 需要二次间接索引
   …
} else {
    cout << "错误，检查 File.cc applyNewBlk 函数" << endl;
    return nullptr;
}
```

在输出错误信息时，就能知道相应区块出了错误。

本次课程设计中，唯一的严重错误是重复释放分配的内存空间。

```
[/] # ls
.
..
bin
dev
etc
home
test
[/] # cd home
[/home] # mkdir a
[/home] # cp texts a
[/home] # mv tests a
要移动的文件/文件夹不存在
[/home] # cd a
文件夹不存在
[/home] # ls
libc++abi.dylib: terminating with uncaught exception of type std::bad_alloc: std::bad_alloc
[1]    50991 abort      ./a
→  src git:(master) ✗
```

经过排查，发现是如下代码出现了错误。

```
FileManager::~FileManager() {
    MemINode* minode = file -> f_minode;
    if(IM -> hasLoadedDINode(minode -> m_number)) {
        IM -> freeMINode(minode); // 释放构造函数中申请的内存 inode 节点
    }
    delete file;
}
```

内存 INode 节点在之前的函数中已经得到释放，却在析构函数中重复释放，所以出现了错误。修正后程序运行正常。

```
[/] # cd home
[/home] # mkdir a
[/home] # ls
.
..
a
photos
reports
texts
[/home] # cp texts a
[/home] # ls
.
..
a
photos
reports
texts
[/home] # cd a
[/home/a] #
```

五、用户使用说明

5.1 开发平台和编译说明

开发的平台是 Mac OS 操作系统。使用的编译器是 clang++。

进入源代码文件夹后，通过如下语句进行编译。

```
clang++ -o a main.cc DeviceDriver.cc SuperBlockManager.cc SuperBlock.cc INode.cc Buffer.cc BufferManager.cc INodeManager.cc File.cc FileManager.cc FileOperator.cc UserInterface.cc
```

```
→  src git:(master) ✗ clang++ -o a main.cc DeviceDriver.cc SuperBlockManager.cc SuperBlock.cc INode.cc Buffer.cc BufferManager.cc INodeManager
.cc File.cc FileManager.cc FileOperator.cc  UserInterface.cc
→  src git:(master) ✗
```

之后执行可执行文件就可以开始运行程序。

5.2 主程序简述

main 函数内容如下。

```
const string text_dir = "./example/text.txt";
const string report_dir = "./example/report.pdf";
const string photo_dir = "./example/photo.jpg";

void initialize(FileOperator& FO);
void JerryTest(FileOperator& FO);

int main() {

    FileOperator FO;
    FO.format();

    FO.mkdir("bin");
    FO.mkdir("etc");
    FO.mkdir("home");
    FO.mkdir("dev");
```

```
    FO.cd("home");
    FO.mkdir("texts");
    FO.mkdir("reports");
    FO.mkdir("photos");

    initialize(FO); // 读入三个文件
    cout << "已将三个文件读入二级文件系统内" << endl;
    FO.goroot(); // 回到根目录

    JerryTest(FO); // 新建 test/Jerry 并进行文件读写测试
    FO.goroot(); // 回到根目录

    cout << endl;
    UserInterface UI(&FO);
    UI.start(); // 开始接收用户输入

    return 0;
}
```

先进行了格式化，之后进行了初始化工作，即新建了文件夹，并将电脑中的三个文件拷贝进二级文件系统；之后新建了 test/Jerry 文件，并进行了读写测试，先循环写入 800 字节（abcd...xyz 循环写入），再从第 500 字节处读出 20 字节。

读写测试的代码如下。

```
void JerryTest(FileOperator& FO) {
    char buf[800];
    char read_out[20];
    int i;
    for(i = 0; i < 800; i++) {
        buf[i] = 'a' + i % 26;
    }

    FO.mkdir("test");
    FO.cd("test");
```

```
    FO.fcreate("Jerry");
    FO.fopen("Jerry");
    cout << "/test/Jerry 创建成功" << endl;

    FO.fwrite(buf, 800);
    cout << "向 Jerry 写入 800 字节" << endl;

    FO.flseek(500); // 就读写指针定位到 500 字节
    FO.fread(read_out, 20);
    cout << "从 Jerry 500 字节开始读出 20 字节" << endl;
    FO.fclose();

    cout << "这 20 字节为: " << endl;
    for(i = 0; i < 20; i++) {
        cout << read_out[i];
    }
    cout << endl;
}
```

执行结果如下。

```
➜  src git:(master) ✗ ./a
已将三个文件读入二级文件系统内
/test/Jerry 创建成功
向 Jerry 写入 800 字节
从 Jerry 500 字节开始读出 20 字节
这 20 字节为:
ghijklmnopqrstuvwxyz

[/] #
```

5.3 文件系统接口说明

在主程序进行完 5.2 中的测试后，将进入用户指令输入阶段。

用户可以输入以下指令。

format：格式化整个磁盘

pwd: 查看当前工作路径

ls: 列出当前文件夹下所有文件名

mkdir xxx: 创建文件夹 xxx

fopen f: 打开文件 f （打开文件后在命令行左边部分会出现提示 $file opened$，可以见上图）

fclose: 关闭当前打开的文件

fread i: 从当前文件读写指针开始向后读 i 字节的内容

fwrite: 从当前文前读写指针开始写入数据

flseek i: 将文件读写指针改为 i

curseek: 查看当前文件读写指针位置

fsize: 查看当前打开的文件大小

fdelete f: 删除目录下的文件 f

dirdelete dir: 删除目录下的文件夹 dir 及其内部所有文件

cd dir: 进入文件夹 dir

mv f dir: 将文件 f 移动到 dir 处

cp f dir: 将文件 f 复制到 dir 处

goroot: 回到根目录

其中蓝色部分需要先执行 fopen 后才能执行。

六、运行结果分析

6.1 文件夹建立和文件复制执行结果

先建立文件夹，指令如下。

```
FO.mkdir("bin");
FO.mkdir("etc");
FO.mkdir("home");
FO.mkdir("dev");

FO.cd("home");
FO.mkdir("texts");
FO.mkdir("reports");
FO.mkdir("photos");
```

之后，将电脑中三个文件 photo.jpg、report.jpg、text.txt 复制到指定的文件夹中（三个文件在 src/example 中）。其中 text.txt 的内容如下。

```
You, a few seconds ago | 1 author (You)
#ifndef SUPERBLOCK
#define SUPERBLOCK

#define INODE_NUM 1024
#define IBITMAP_SIZE 128
#define DISK_BLOCK_NUM 64 * 1024

class SuperBlock {
public:
    int s_isize; // inode 总个数
    char s_ibitmap[IBITMAP_SIZE]; // 记录 inode 分配情况的位图
    int s_ilock; // 临界区锁

    int s_fsize; // 盘块总数
    int s_nfree; // 直接管理的空闲盘块数
    int s_free[100]; // 直接管理的空闲盘块的索引表
    int s_flock; // 临界区锁

    int s_dirty; // 指示是否需要写回磁盘

    char padding[472]; // 填充至 1024 字节

public:
    SuperBlock();
};

#endif
```

进入文件系统进行测试。

```
[/] # ls
.
..
bin
dev
etc
home
test
[/] # cd home
[/home] # ls
.
..
photos
reports
texts
[/home] # cd photos
[/home/photos] # ls
.
..
photos.jpg
[/home/photos] # cd ..
[/home] # cd reports
[/home/reports] # ls
.
..
reports.pdf
[/home/reports] # cd ..
[/home] # cd texts
[/home/texts] # ls
.
..
text.txt
[/home/texts] #
```

可以发现文件夹被正确建立，文件也正确读入。

接下来打开 text.txt 进行查看。

```
[/home/texts] # ls
.
..
text.txt
[/home/texts] # fopen text.txt
[/home/texts] $file opened$ # fsize
文件大小为 609 字节
[/home/texts] $file opened$ # fread 10
读出 10 字节
#ifndef SU
[/home/texts] $file opened$ # fread 10
读出 10 字节
PERBLOCK
#
[/home/texts] $file opened$ # flseek 0
[/home/texts] $file opened$ # fread 20
读出 20 字节
#ifndef SUPERBLOCK
#
```

如上图，先打开了文件。分两次读取了 10 字节数据。再修改文件读写指针回到开头，再读出 20 字节数据。可以发现分两次和一次性读出 20 字节数据一致。

```
[/home/texts] $file opened$ # fread 20
读出 20 字节
#ifndef SUPERBLOCK
#
[/home/texts] $file opened$ # curseek
20
[/home/texts] $file opened$ # flseek 0
[/home/texts] $file opened$ # fwrite
abcdefghijk
写入 11 字节
[/home/texts] $file opened$ # flseek 0
[/home/texts] $file opened$ # fread 20
读出 20 字节
abcdefghijkERBLOCK
#
[/home/texts] $file opened$ # fclose
[/home/texts] #
```

如上图，接下来继续测试，通过 curseek 查看当前文件读写指针，确实为 20。将文件读写指正改回开头候，进行 fwrite 写入，写入 11 个字母。之后重新读出开头 20 字节，文件被正确修改。最后关闭文件。

6.2 Jerry 文件读写结果

```
→  src git:(master) ✗ ./a
已将三个文件读入二级文件系统内
/test/Jerry 创建成功
向 Jerry 写入 800 字节
从 Jerry 500 字节开始读出 20 字节
这 20 字节为:
ghijklmnopqrstuvwxyz
```

如上，在程序开始时，会自动测试 Jerry 读写。

之后再手动进行检查。

```
[/] # ls
.
..
bin
dev
etc
home
test
[/] # cd test
[/test] # ls
.
..
Jerry
[/test] # fopen Jerry
[/test] $file opened$ # fsize
文件大小为 800 字节
[/test] $file opened$ # flseek 200
[/test] $file opened$ # flseek 500
[/test] $file opened$ # fread 20
读出 20 字节
ghijklmnopqrstuvwxyz
[/test] $file opened$ #
```

如上，进入 Jerry 所在目录后，打开文件，并将文件指针指向 500 字节，读出 20 字节的结果和开头的自由测试一致。

6.3 自由测试结果

```
[/] # ls
.
..
bin
dev
etc
home
test
[/] # clear
不支持的指令: clear
[/] # ls a
ls 不能接受参数
[/] # cp test
cp 接受 2 个参数
[/] #
```

之后的自由测试中，如果出现了输入错误，会有温馨的提示。

```
[/] # ls
.
..
bin
dev
etc
home
test
[/] # mkdir wnq
[/] # ls
.
..
bin
dev
etc
home
test
wnq
[/] #
```

如上图，成功通过 mkdir 创建了 wnq 这个文件夹。接下来进入新简历的文件夹，并创建一个文件。

```
[/] # cd wnq
[/wnq] # fcreate a
[/wnq] # fopen a
[/wnq] $file opened$ # fwrite
asiduyfiudshfiuyr 87239874 ********
写入 35 字节
[/wnq] $file opened$ # fsize
文件大小为 35 字节
[/wnq] $file opened$ # fclose
[/wnq] #
```

如上，进入 wnq 后，创建了文件 a，并写入了 35 字节数据。

```
[/wnq] $file opened$ # fsize
文件大小为 35 字节
[/wnq] $file opened$ # fclose
[/wnq] # mkdir 1550431
[/wnq] # cp a 1550431
[/wnq] # ls
.
..
1550431
a
[/wnq] # cd 1550431
[/wnq/1550431] # ls
.
..
a
[/wnq/1550431] #
```

如上， wnq 中创建新的文件夹 1550431，并把刚刚创建的文件 a 复制到 1550431 中。

```
[/wnq/1550431] # ls
.
..
a
[/wnq/1550431] # fopen a
[/wnq/1550431] $file opened$ # fsize
文件大小为 35 字节
[/wnq/1550431] $file opened$ # fread 150
读出 35 字节
asiduyfiudshfiuyr 87239874 ********
[/wnq/1550431] $file opened$ #
```

如上，打开复制的文件 （1550431 中）的 a，发现也是 35 字节，同时读取文件发现和之前写入内容一致，说明复制成功。

```
asiduyfiudshfiuyr 87239874 ********
[/wnq/1550431] $file opened$ # fclose
[/wnq/1550431] # cd ..
[/wnq] # ls
.
..
1550431
a
[/wnq] # fdelete a
[/wnq] # ls
.
..
1550431
[/wnq] # cd 1550431
[/wnq/1550431] # ls
.
..
a
[/wnq/1550431] #
```

之后回到 wnq 文件夹，并删除 a 文件。再回到 1550431 文件夹，发现 a 仍然存在。

七、实验总结

课程设计总体来说是成功的。

通过课程设计，我对文件系统和操作系统的理解更加深刻。

从基本的磁盘读写，到高速缓存的构造，再到利用缓存进行文件的读写，我对数据在内存和磁盘间的移动和存储有了更深刻的理解；从超级块的结构，到 INode 的使用和分配，到物理盘块的使用和分配，再到文件的储存方式，我对文件的形式和结构也有了更深刻的理解。

此外，使用面向对象的 C++ 进行编程的过程中，我的面向对象编程的能力也得到了很大的训练。我一直在思索如何更有效地划分类，以求高内聚和低耦合；我也一直在思索如何设计接口，使得类的接口简洁、高效而不冗余。

八、参考文献

[1] 第八章、Linux 磁盘与文件系统管理 http://cn.linux.vbird.org/linux_basic/0230filesystem.php
[2] Linux 文件系统(一)—虚拟文件系统VFS----超级块、inode、dentry、file https://blog.csdn.net/shanshanpt/article/details/38943731
[3] 漫谈Linux标准的文件系统(Ext2/Ext3/Ext4) https://www.cnblogs.com/justmine/p/9128730.html
[4] 浅析unix文件系统 https://yisaer.github.io/2017/03/23/pwd/
[5] https://github.com/Sen666666/SecondFileSystem