操作系统课程设计实验报告

一.需求分析

1.1 任务描述

使用一个普通的大文件（如c:\myDisk.img，称之为一级文件）来模拟UNIX V6++的一个文件卷（把一个大文件当一张磁盘用）。

S	inode区	文件数据区

1）定义磁盘文件结构

- 定义自己的磁盘文件结构
- SuperBlock结构
- 磁盘Inode节点结构，包括：索引结构，及：逻辑块号到物理块号的映射
- 磁盘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

2）定义文件目录结构

- 目录文件结构
- 目录检索算法的设计与实现

3）定义文件打开结构

4）完成磁盘高速缓存（选作/已完成）

5）完成多用户同时访问二级文件系统（选作/未完成）

6）定义多类文件操作接口，完成各类文件功能：

- fformat：格式化文件卷
- ls：列目录
- cd：改变目录
- mkdir：创建目录
- fcreat：新建文件
- fopen：打开文件
- fclose：关闭文件
- fread：读文件
- fwrite：写文件
- fseek：定位文件读写指针
- fdelete：删除文件
- copy：拷贝文件
- copyin：从外部操作系统拷贝文件
- copyout：拷贝文件到外部操作系统
- more：以ASCII码形式显示文件内容

1.2 输入形式

通过命令行方式，进行指令输入

输入指令格式如下所示：

- 格式化：fformat
- 退出：exit
- 新建目录：mkdir <目录名>
- 改变目录：cd <目录名>
- 列出目录：ls
- 新建文件：fcreate <文件名> <选项>
- 打开文件：fopen <文件名> <选项>

- 删除文件：fdelete <文件名>
- 关闭文件：fclose <文件描述符>
- 定位文件读写指针： fseek <文件描述符> <偏移量> <起始位置>
- 写文件：fwrite <文件描述符> <写入文件> <写入大小>
- 读文件：fread <文件描述符> [-o <读出文件>] <读出大小>
- 拷贝文件：copy <源文件> <目标文件>
- 从外部操作系统拷贝文件 : copyin <源文件> <目标文件>
- 拷贝文件到外部操作系统：copyout <源文件> <目标文件>
- 显示文件：more <文件>

1.3 输出形式

通过在命令行给出提示的方式，进行指令完成的输出

1.4 程序功能

通过在命令行输入相应指令，对程序的各项功能进行展示。

可实现格式化文件卷；目录的改变、新建与列出；

文件的新建、打开、关闭与删除；

文件的读写、指针的定位；

文件的拷贝（从内到内，从内到外，从外到内）

文件内容的显示

二.概要设计

2.1 任务分解

本次课程设计的目标是实现一个基于Unix V6++的二级文件管理系统。在本系统中，使用一个二进制大文件模拟磁盘，每 512 个字节分为一个数据段，用以模拟磁盘的各个扇区。

在Unix V6++的一级文件系统中，通过文件系统向硬盘发 DMA 命令，从而在物理磁盘上读写数据；而二级文件系统则依托于宿主机的一级文件系统，可以通过操作系统提供的 fread 、fwrite 系统调用，在虚拟磁盘镜像上进行读写操作。通过改写磁盘驱动接口，再结合 Unix V6++的文件管理和高速缓冲管理模块，完成本次二级文件系统的构建。本系统分为以下几个模块：

- 磁盘驱动模块（DeviceDriver）：

  初始化磁盘镜像文件，调用系统调用对磁盘镜像文件进行读写

- 高速缓存管理模块（BufferManager）：

  管理系统中的所有缓存块，包括缓存的分配、回收、清空，调用磁盘驱动读写缓存块，在系统退出时对缓存块进行刷新；

- 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SuperBlock的分配，外存 Inode 、DiskNode节点的分配、释放，格式化磁盘文件的接口。

- 打开文件管理模块（OpenFileManager)：

  进行打开文件管理，建立用户与打开文件内核数据之间的勾连，为用户提供直接操作文件的文件描述符接口。

- 文件功能接口模块（FileManager）：

  提供各类文件操作接口，封装文件系统中对文件处理的操作过程，包括创建、打开、删除、读写、拷贝、显示文件等操作；

- 用户操作接口模块（User）：

  作为用户与系统的衔接模块，调用内核函数实现 API，直接与用户交互，同时对输入与输出的正确性进行检查。

2.2 数据结构设计

2.2.1 设备驱动模块：DeviceDriver

```
class DeviceDriver {
public:
    /* 磁盘镜像文件名 */
    static const char* DISK_FILE_NAME;

private:
    /* 磁盘文件指针 */
    FILE* fp;

public:
    DeviceDriver();
    ~DeviceDriver();

    /* 检查镜像文件是否存在 */
    bool Exists();

    /* 打开镜像文件 */
    void Construct();

    /* 写磁盘函数 */
    void write(const void* buffer, unsigned int size,
        int offset = -1, unsigned int origin = SEEK_SET);

    /* 读磁盘函数 */
    void read(void* buffer, unsigned int size,
        int offset = -1, unsigned int origin = SEEK_SET);
};
```

DeviceDriver 负责对镜像文件的初始化以及读写。

2.2.2 高速缓存管理模块：BufferManager

```
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 */
    };

public:
    unsigned int flags;     /* 缓存控制块标志位 */

    Buffer* forw;
    Buffer* back;

    int     wcount;         /* 需传送的字节数 */
    unsigned char* addr;    /* 指向该缓存控制块所管理的缓冲区的首地址 */
    int     blkno;          /* 磁盘逻辑块号 */
    int     u_error;        /* I/O出错时信息 */
    int     resid;          /* I/O出错时尚未传送的剩余字节数 */
    int no;
public:
    Buffer();
    ~Buffer();

};
```

Buffer块记录了相应缓存的使用情况等信息；同时记录该缓存相关的 I/O 请求和执行结果。

```
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:
    Buffer* bufferList;                         /* 缓存队列控制块 */
    Buffer nBuffer[NBUF];                       /* 缓存控制块数组 */
    unsigned char buffer[NBUF][BUFFER_SIZE];    /* 缓冲区数组 */
    unordered_map<int, Buffer*> map;
    DeviceDriver* deviceDriver;

public:
    BufferManager();
    ~BufferManager();

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

    /* 申请一块缓存，用于读写设备上的块blkno。*/
    Buffer* GetBlk(int blkno);

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

    /* 读一个磁盘块，blkno为目标磁盘块逻辑块号。 */
    Buffer* Bread(int blkno);

    /* 写一个磁盘块 */
    void Bwrite(Buffer* bp);
```

```
    /* 延迟写磁盘块 */
    void Bdwrite(Buffer* bp);

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

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

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

BufferManager 负责缓存块和自由缓存队列的管理，包括分配、读写、回收等操作。

2.2.3 文件系统资源管理模块 FileSystem

FileSystem类：

```
class FileSystem {
public:
    // Block块大小
    static const int BLOCK_SIZE = 512;

    // 磁盘所有扇区数量
    static const int DISK_SIZE = 16384;

    // 定义SuperBlock位于磁盘上的扇区号，占据两个扇区
    static const int SUPERBLOCK_START_SECTOR = 0;


    // 外存INode区位于磁盘上的起始扇区号
    static const int INODE_ZONE_START_SECTOR = 2;

    // 磁盘上外存INode区占据的扇区数
    static const int INODE_ZONE_SIZE = 1022;

    // 外存INode对象长度为64字节，每个磁盘块可以存放512/64 = 8个外存INode
    static const int INODE_NUMBER_PER_SECTOR = BLOCK_SIZE / sizeof(DiskINode);

    // 文件系统根目录外存INode编号
    static const int ROOT_INODE_NO = 0;

    // 外存INode的总个数
    static const int INode_NUMBERS = INODE_ZONE_SIZE* INODE_NUMBER_PER_SECTOR;

    // 数据区的起始扇区号
    static const int DATA_ZONE_START_SECTOR = INODE_ZONE_START_SECTOR +
INODE_ZONE_SIZE;

    // 数据区的最后扇区号
    static const int DATA_ZONE_END_SECTOR = DISK_SIZE - 1;

    // 数据区占据的扇区数量
```

```
    static const int DATA_ZONE_SIZE = DISK_SIZE - DATA_ZONE_START_SECTOR;

public:
    DeviceDriver* deviceDriver;
    SuperBlock* superBlock;
    BufferManager* bufferManager;

public:
    FileSystem();
    ~FileSystem();

    /* 格式化SuperBlock */
    void FormatSuperBlock();

    /* 格式化整个文件系统 */
    void FormatDevice();

    /* 系统初始化时读入SuperBlock */
    void LoadSuperBlock();

    /* 将SuperBlock对象的内存副本更新到存储设备的SuperBlock中去 */
    void Update();

    /* 在存储设备dev上分配一个空闲外存INode，一般用于创建新的文件。*/
    INode* IAlloc();

    /* 释放编号为number的外存INode，一般用于删除文件。*/
    void IFree(int number);

    /* 在存储设备上分配空闲磁盘块 */
    Buffer* Alloc();

    /* 释放存储设备dev上编号为blkno的磁盘块 */
    void Free(int blkno);

};
```

INode 类与 DiskNode 类:

```
class INode {
public:
    // INodeFlag中标志位
    enum INodeFlag {
        //ILOCK = 0x1,  // 索引节点上锁
        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
        //IMOUNT = 0x8, // 内存INode用于挂载子文件系统
        //IWANT = 0x10, // 有进程正在等待该内存INode被解锁，清ILOCK标志时，要唤醒这种进程
        //ITEXT = 0x20  // 内存INode对应进程图像的正文段
    };

    static const unsigned int IALLOC = 0x8000;      /* 文件被使用 */
    static const unsigned int IFMT = 0x6000;        /* 文件类型掩码 */
    //static const unsigned int IFCHR = 0x2000;     /* 字符设备特殊类型文件 */
    static const unsigned int IFDIR = 0x4000;       /* 文件类型：目录文件 */
    static const unsigned int IFBLK = 0x6000;       /* 块设备特殊类型文件，为0表示常
规数据文件 */
```

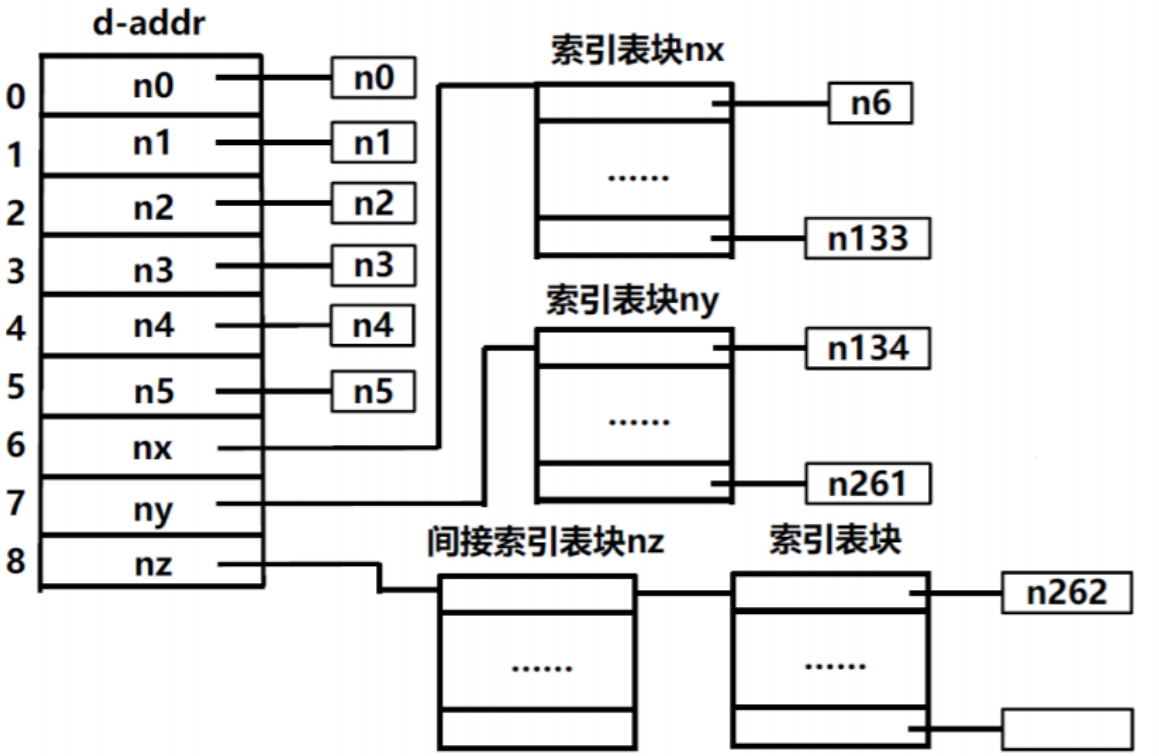
```
    static const unsigned int ILARG = 0x1000;        /* 文件长度类型：大型或巨型文件
*/
    //static const unsigned int ISUID = 0x800;       /* 执行时文件时将用户的有效用户ID
修改为文件所有者的User ID */
    //static const unsigned int ISGID = 0x400;       /* 执行时文件时将用户的有效组ID修
改为文件所有者的Group ID */
    //static const unsigned int ISVTX = 0x200;       /* 使用后仍然位于交换区上的正文段
*/
    static const unsigned int IREAD = 0x100;         /* 对文件的读权限 */
    static const unsigned int IWRITE = 0x80;         /* 对文件的写权限 */
    static const unsigned int IEXEC = 0x40;          /* 对文件的执行权限 */
    static const unsigned int IRWXU = (IREAD | IWRITE | IEXEC);      /* 文件主对文
件的读、写、执行权限 */
    static const unsigned int IRWXG = ((IRWXU) >> 3);                /* 文件主同组
用户对文件的读、写、执行权限 */
    static const unsigned int IRWXO = ((IRWXU) >> 6);                /* 其他用户对
文件的读、写、执行权限 */

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

    static const int SMALL_FILE_BLOCK = 6;  /* 小型文件：直接索引表最多可寻址的逻辑块号
*/
    static const int LARGE_FILE_BLOCK = 128 * 2 + 6;     /* 大型文件：经一次间接索引
表最多可寻址的逻辑块号 */
    static const int HUGE_FILE_BLOCK = 128 * 128 * 2 + 128 * 2 + 6; /* 巨型文件：
经二次间接索引最大可寻址文件逻辑块号 */

public:
    unsigned int i_flag;    // 状态的标志位，定义见enum INodeFlag
    unsigned int i_mode;    // 文件工作方式信息

    int     i_count;        // 引用计数
    int     i_nlink;        // 文件联结计数，即该文件在目录树中不同路径名的数量

    short   i_dev;          // 外存INode所在存储设备的设备号
    int     i_number;       // 外存INode区中的编号

    short   i_uid;          // 文件所有者的用户标识数
    short   i_gid;          // 文件所有者的组标识数

    int     i_size;         // 文件大小，字节为单位
    int     i_addr[10];     // 用于文件逻辑块好和物理块好转换的基本索引表

    int     i_lastr;        // 存放最近一次读取文件的逻辑块号，用于判断是否需要预读

public:
    INode();
    ~INode();

    /* 根据Inode对象中的物理磁盘块索引表，读取相应的文件数据 */
    void ReadI();

    /* 根据Inode对象中的物理磁盘块索引表，将数据写入文件 */
    void WriteI();

    /* 将文件的逻辑块号转换成对应的物理盘块号 */
```

```
    int Bmap(int lbn);

    /* 对特殊字符设备、块设备文件，调用该设备注册在块设备开关表
    *  中的设备初始化程序
    */
    //void OpenI(int mode);

    /* 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 */
    void Clean();

    /* 将包含外存Inode字符块中信息拷贝到内存Inode中 */
    void ICopy(Buffer* bp, int inumber);
};

class DiskINode {
public:
    unsigned int d_mode;    // 状态的标志位，定义见enum INodeFlag
    int     d_nlink;        // 文件联结计数，即该文件在目录树中不同路径名的数量

    short   d_uid;          // 文件所有者的用户标识数
    short   d_gid;          // 文件所有者的组标识数

    int     d_size;         // 文件大小，字节为单位
    int     d_addr[10];     // 用于文件逻辑块号和物理块号转换的基本索引表

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

public:
    DiskINode();
    ~DiskINode();
};
```

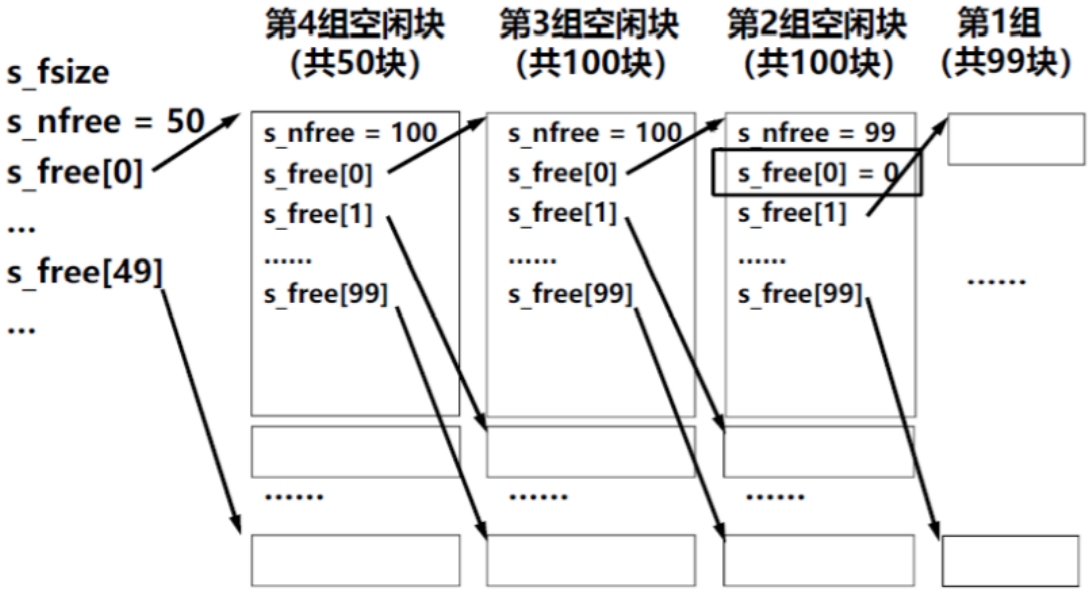
FileSystem 中定义了格式化磁盘的各个参数，DiskInode 区的大 小，数据区的长度等，对磁盘文件大小产生直接作用。

Unix v6++采用三级混合索引机制结构，在 Inode 及 DiskInode 类中有一个 d_addr[] 数组，负责映射逻辑盘块号到物理盘块号。

将逻辑盘块 n 对应的物理块号存放在 d_addr[n] 中。将文件按大小分为三级，分别为小型文件（0~6 盘块，最大 3K）、大型文件 （7~6+128 * 2 盘块） 、巨型文件（128 * 2+7~6+128 * 2+128 * 128 * 2 盘块）。d_addr[]数组中 0#~5#为直接索引，直接记录对应数据物理盘块号；6#~7#为一次间接索引，它指向一个盘 块号，再从该盘块中获取索引，一个盘块大小为 512 字节，int 型数据大小 4 字节，故可存储 128 个索引；8#~9#为二次间接索引，它指向的盘块为间接索引块，从该盘块获取对应索引表块，再从索引表块找到对应物理盘块。结构示意图如下：



SuperBlock 对于空闲 Inode 的管理，任何时候只管理 s_ninode 个空闲 Inode，按栈的方式使用，当管理的 Inode 分配完后，再重新在 Inode 区搜索 100 个空闲的 Inode 加入管理。 对于空闲盘块的管理采用成组链接法，释放一盘块时，进栈， 分配一盘块时，退栈。对所有空闲盘块分组，最后一组由 SuperBlock 直接管理，每个分组首盘块的开始 101 个字存放前一个分组的索引，第一组为 99 块，故第二组首盘块的 0#索引为 0 即为空闲盘块结束标志，基本结构示意图如下：



2.2.4 打开文件管理模块 OpenFileManager

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

```
public:
    File();
    ~File();

    unsigned int flag;      /* 对打开文件的读、写操作要求 */
    int     count;          /* 当前引用该文件控制块的进程数量 */
    INode*  inode;          /* 指向打开文件的内存INode指针 */
    int     offset;         /* 文件读写位置指针 */
};

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();

    /* 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);
};

class IOParameter {
public:
    unsigned char* base;    /* 当前读、写用户目标区域的首地址 */
    int offset;             /* 当前读、写文件的字节偏移量 */
    int count;              /* 当前还剩余的读、写字节数量 */
};
```

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

public:
    /* 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。
     */
    File m_File[MAX_FILES];

public:
    OpenFileTable();
    ~OpenFileTable();

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

```
    /* 对打开文件控制块File结构的引用计数count减1，若引用计数count为0，则释放File结构。*/
    void CloseF(File* pFile);

    void Format();
};

class INodeTable {
public:
    static const int NINODE = 100;  /* 内存INode的数量 */

private:
    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* fileSystem; /* 对全局对象gc_FileSystem的引用 */

public:
    INodeTable();
    ~INodeTable();

    /*
     * 根据外存INode编号获取对应INode。如果该INode已经在内存中，返回该内存INode；
     * 如果不在内存中，则将其读入内存后上锁并返回该内存INode
     */
    INode* IGet(int inumber);

    /*
     * 减少该内存INode的引用计数，如果此INode已经没有目录项指向它，
     * 且无进程引用该INode，则释放此文件占用的磁盘块。
     */
    void IPut(INode* pNode);

    /* 将所有被修改过的内存INode更新到对应外存INode中 */
    void UpdateINodeTable();

    /*
     * 检查编号为inumber的外存INode是否有内存拷贝，
     * 如果有则返回该内存INode在内存INode表中的索引
     */
    int IsLoaded(int inumber);

    /* 在内存INode表中寻找一个空闲的内存INode */
    INode* GetFreeINode();

    void Format();
};
```

打开文件控制块File类，记录了进程打开文件的读、写请求类型，文件读写位置等等。

进程打开文件描述符表 OpenFiles 则维护了当前进程的所有打开文件。

文件I/O的参数类 IOParameter 给出文件读、写时需用到的读、写偏移量、字节数以及目标区域首地址参数。

打开文件管理类(OpenFileManager)负责内核中对打开文件机构的管理，为进程打开文件建立内核数据结构之间的勾连关系。连关系指进程u区中打开文件描述符指向打开文件表中的File打开文件控制结构，以及从File结构指向文件对应的内存INode。

内存INode表(class INodeTable)负责内存INode的分配和释放。

2.2.5 文件功能接口模块 FileManager

```
class FileManager
{
public:
    /* 目录搜索模式，用于NameI()函数 */
    enum DirectorySearchMode
    {
        OPEN = 0,        /* 以打开文件方式搜索目录 */
        CREATE = 1,      /* 以新建文件方式搜索目录 */
        DELETE = 2       /* 以删除文件方式搜索目录 */
    };

public:
    /* 根目录内存INode */
    INode* rootDirINode;

    /* 对全局对象gc_FileSystem的引用，该对象负责管理文件系统存储资源 */
    FileSystem* fileSystem;

    /* 对全局对象gc_INodeTable的引用，该对象负责内存INode表的管理 */
    INodeTable* inodeTable;

    /* 对全局对象gc_OpenFileTable的引用，该对象负责打开文件表项的管理 */
    OpenFileTable* openFileTable;

public:
    FileManager();
    ~FileManager();

    /* Open()系统调用处理过程 */
    void Open();

    /* Creat()系统调用处理过程 */
    void Creat();

    /* Open()、Creat()系统调用的公共部分 */
    void Open1(INode* pINode, int mode, int trf);

    /* Close()系统调用处理过程 */
    void Close();

    /* Seek()系统调用处理过程 */
    void Seek();

    /* Read()系统调用处理过程 */
    void Read();

    /* Write()系统调用处理过程 */
    void Write();

    /* 读写系统调用公共部分代码 */
    void Rdwr(enum File::FileFlags mode);

    /* 目录搜索，将路径转化为相应的INode返回上锁后的INode */
    INode* NameI(enum DirectorySearchMode mode);
```

```
    /* 被Creat()系统调用使用，用于为创建新文件分配内核资源 */
    INode* MakNode(unsigned int mode);

    /* 取消文件 */
    void UnLink();

    /* 向父目录的目录文件写入一个目录项 */
    void WriteDir(INode* pINode);

    /* 改变当前工作目录 */
    void ChDir();

    /* 列出当前INode节点的文件项 */
    void Ls();
};
```

文件管理类(FileManager)，封装了文件系统的各种系统调用在核心态下处理过程，如对文件的Open()、Close()、Read()、Write()等等，封装了对文件系统访问的具体细节。

2.2.6 用户操作接口模块User

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

    enum ErrorCode {
        U_NOERROR = 0,      /* No u_error */
        U_EPERM = 1,        /* Operation not permitted */
        U_ENOENT = 2,       /* No such file or directory */
        U_ESRCH = 3,        /* No such process */
        U_EINTR = 4,        /* Interrupted system call */
        U_EIO = 5,          /* I/O u_error */
        U_ENXIO = 6,        /* No such device or address */
        U_E2BIG = 7,        /* Arg list too long */
        U_ENOEXEC = 8,      /* Exec format u_error */
        U_EBADF = 9,        /* Bad file number */
        U_ECHILD = 10,      /* No child processes */
        U_EAGAIN = 11,      /* Try again */
        U_ENOMEM = 12,      /* Out of memory */
        U_EACCES = 13,      /* Permission denied */
        U_EFAULT = 14,      /* Bad address */
        U_ENOTBLK = 15,     /* Block device required */
        U_EBUSY = 16,       /* Device or resource busy */
        U_EEXIST = 17,      /* File exists */
        U_EXDEV = 18,       /* Cross-device link */
        U_ENODEV = 19,      /* No such device */
        U_ENOTDIR = 20,     /* Not a directory */
        U_EISDIR = 21,      /* Is a directory */
        U_EINVAL = 22,      /* Invalid argument */
        U_ENFILE = 23,      /* File table overflow */
        U_EMFILE = 24,      /* Too many open files */
        U_ENOTTY = 25,      /* Not a typewriter(terminal) */
        U_ETXTBSY = 26,     /* Text file busy */
        U_EFBIG = 27,       /* File too large */
        U_ENOSPC = 28,      /* No space left on device */
```

```
        U_ESPIPE = 29,      /* Illegal seek */
        U_EROFS = 30,       /* Read-only file system */
        U_EMLINK = 31,      /* Too many links */
        U_EPIPE = 32,       /* Broken pipe */
        U_ENOSYS = 100,
        DEFAULT = 106
    };

public:
    INode* cdir;        /* 指向当前目录的Inode指针 */
    INode* pdir;        /* 指向父目录的Inode指针 */

    DirectoryEntry dent;                    /* 当前目录的目录项 */
    char dbuf[DirectoryEntry::DIRSIZ];      /* 当前路径分量 */
    string curDirPath;                      /* 当前工作目录完整路径 */

    string dirp;                /* 系统调用参数(一般用于Pathname)的指针 */
    long arg[5];                /* 存放当前系统调用参数 */
                                /* 系统调用相关成员 */
    unsigned int    ar0[5];     /* 指向核心栈现场保护区中EAX寄存器
                                   存放的栈单元，本字段存放该栈单元的地址。
                                   在V6中r0存放系统调用的返回值给用户程序，
                                   x86平台上使用EAX存放返回值，替代u.ar0[R0] */
    ErrorCode u_error;          /* 存放错误码 */

    OpenFiles ofiles;           /* 进程打开文件描述符表对象 */

    IOParameter IOParam;        /* 记录当前读、写文件的偏移量，用户目标区域和剩余字节数参
数 */

    FileManager* fileManager;

    string ls;

public:
    User();
    ~User();

    void Ls();
    void Cd(string dirName);
    void Mkdir(string dirName);
    void Fcreate(string fileName, string mode);
    void Fdelete(string fileName);
    void Fopen(string fileName, string mode);
    void Fclose(string fd);
    void Fseek(string fd, string offset, string origin);

    void Fwrite(string fd, string inFile, string size);
    void Fread(string fd, string outFile, string size);
    void Copy(string srcFile, string desfile);
    void Copyin(string srcFile, string desfile);
    void Copyout(string srcFile, string desfile);
    void More(string srcFile);

private:
    bool IsError();
    void EchoError(enum ErrorCode err);
    int INodeMode(string mode);
```

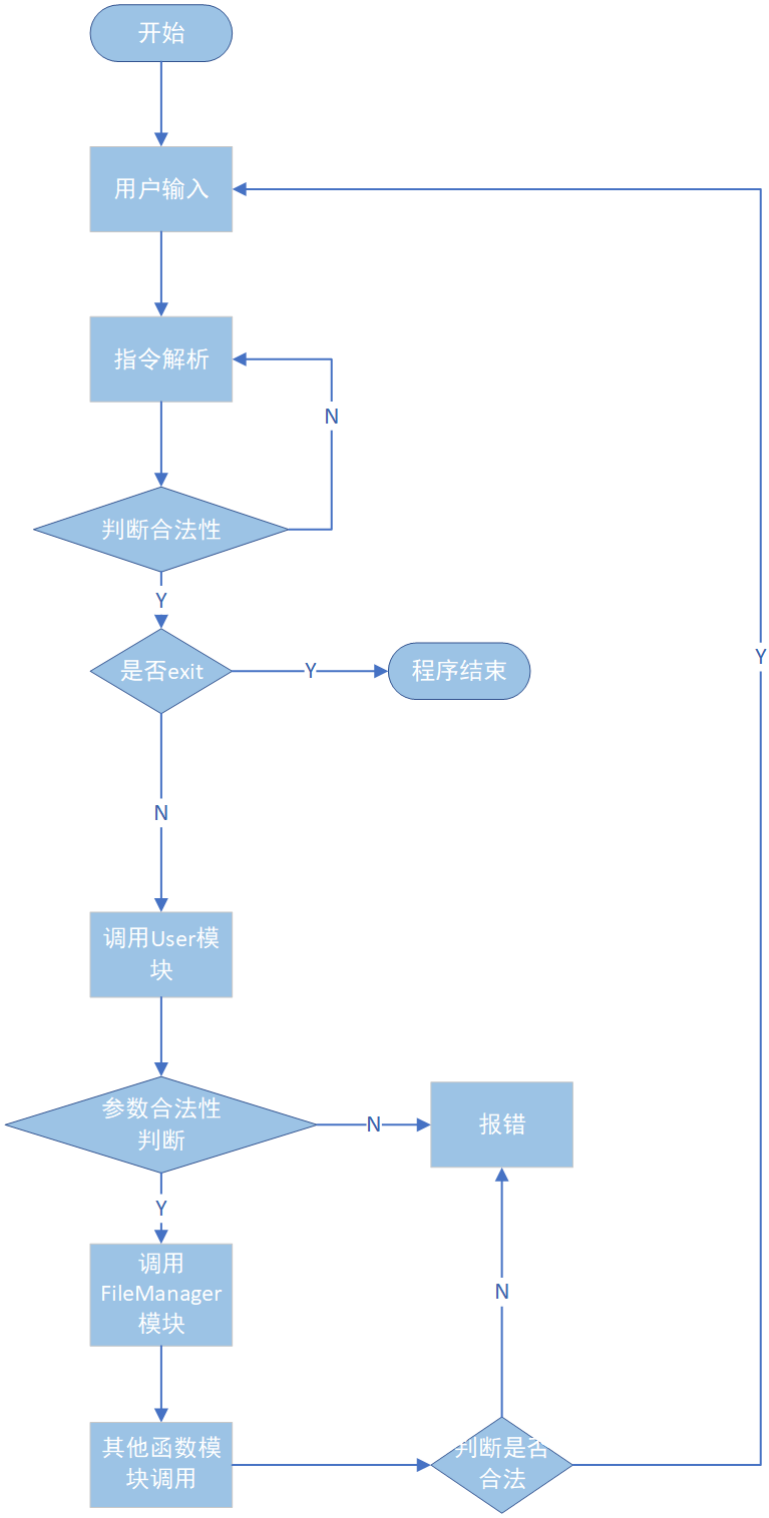
```
    int FileMode(string mode);
    bool checkPathName(string path);
};
```

User 结构主要记录一些参数，方便其他函数使用时不用大量传参。相对 Unix v6++， 本系统的 User 结构相较于Unix V6++进行了大规模的简化。首先去除了信号的一些定义，另外不涉及到用户栈、核心栈等，将 esp、ebp 指针删去，还删去了进程管理和时间相关变量。另外将 ar0 定义为unsigned int 型数据，用来存放系统调用的返回值。

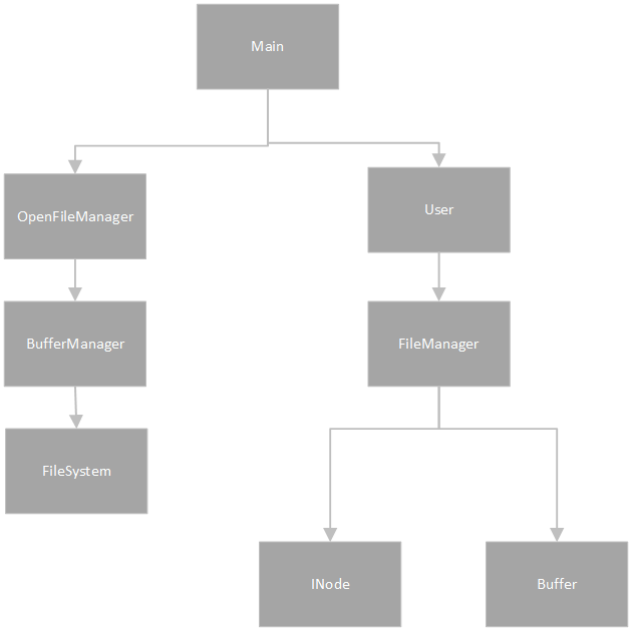
2.3 主程序流程

程序的主程序流程为：

main 函数等待用户输入相应指令，用户输入后，对指令进行简单解析， 查找 User 提供的对应操作接口，将参数交由 User 函数进行处理和判断合法性，若合法则由User调用FileManager中功能函数实现文件系统的具体功能，若不合法则报错。


开始
用户输入
指令解析
N
判断合法性
Y
是否exit
Y
程序结束
N
调用User模块
参数合法性判断
N
报错
Y
调用FileManager模块
N
其他函数模块调用
判断是否合法
Y

2.4 模块间调用关系



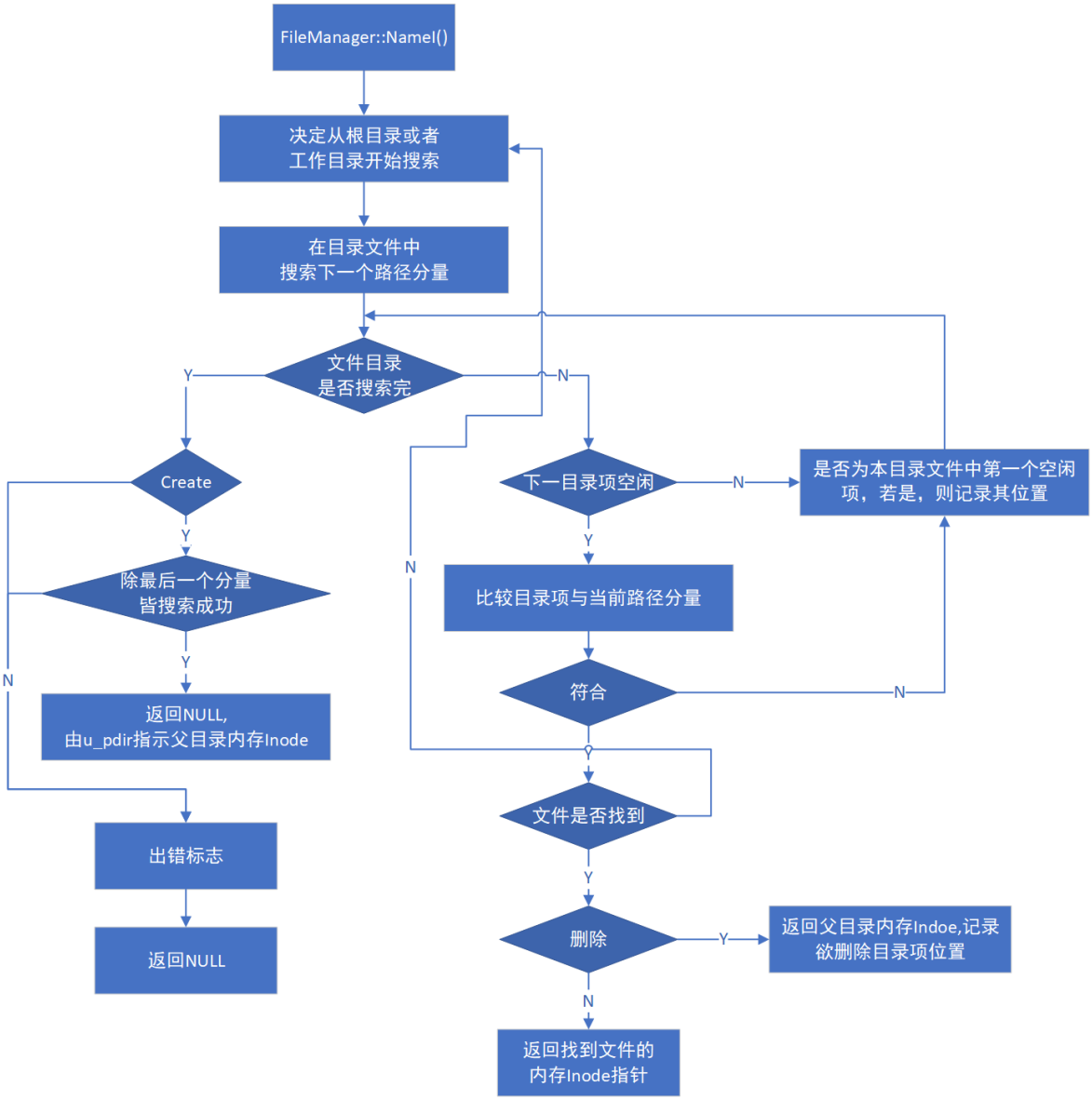
三.详细设计

3.1 目录搜索与地址变换

3.1.1目录搜索函数

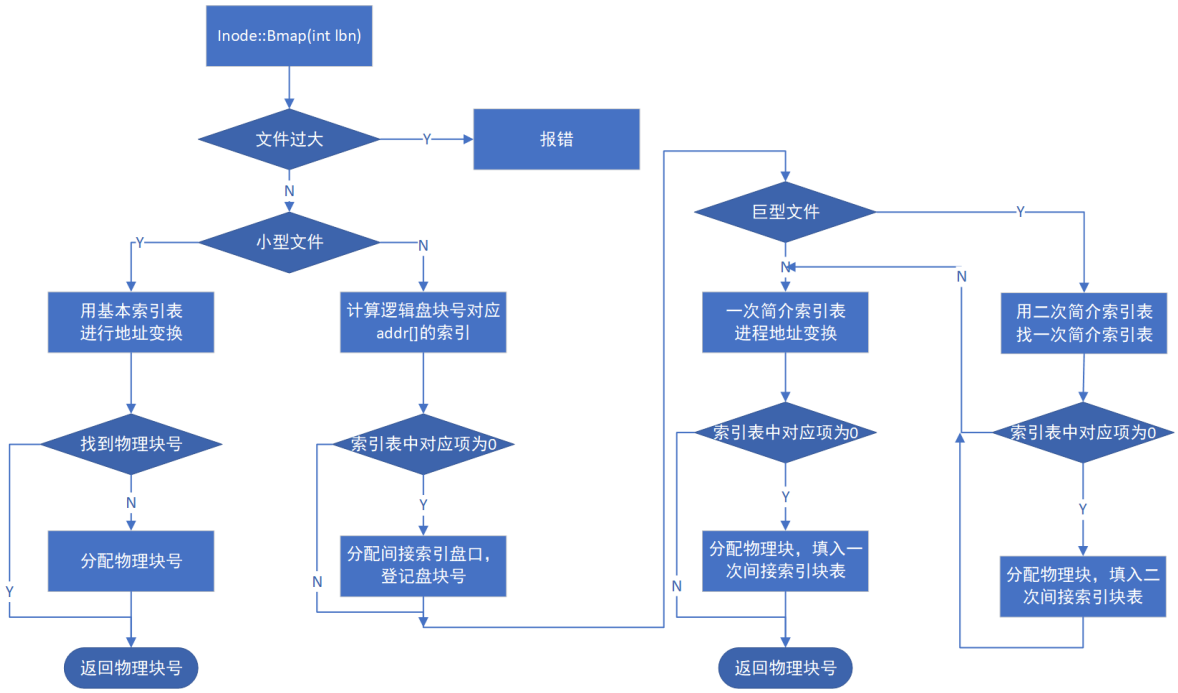
```
INode* FileManager::NameI(enum DirectorySearchMode mode)
```

将路径名到索引节点转换的过程，与 Unix V6++目录搜索函数基本类似，流程图如下：



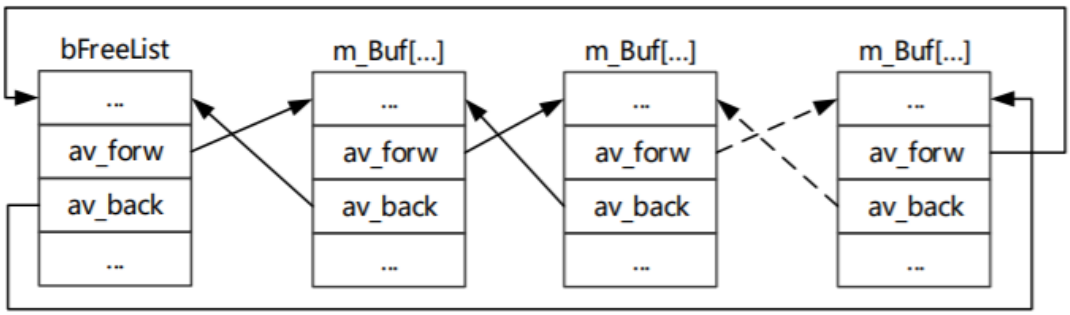
3.1.2 地址变换函数

将对应的逻辑块号转换为物理盘块号。



3.2 内存高速缓存

本次内存高速缓存的实现将自由队列和设备队列进行合并为一个缓存队列进行操作。



3.2.1 分离节点

```
void BufferManager::DetachNode(Buffer* pb)
```

采用LRU Cache 算法，每次从头部取出，使用后放到尾部.

3.2.2 插入节点

```
void BufferManager::InsertTail(Buffer* pb)
```

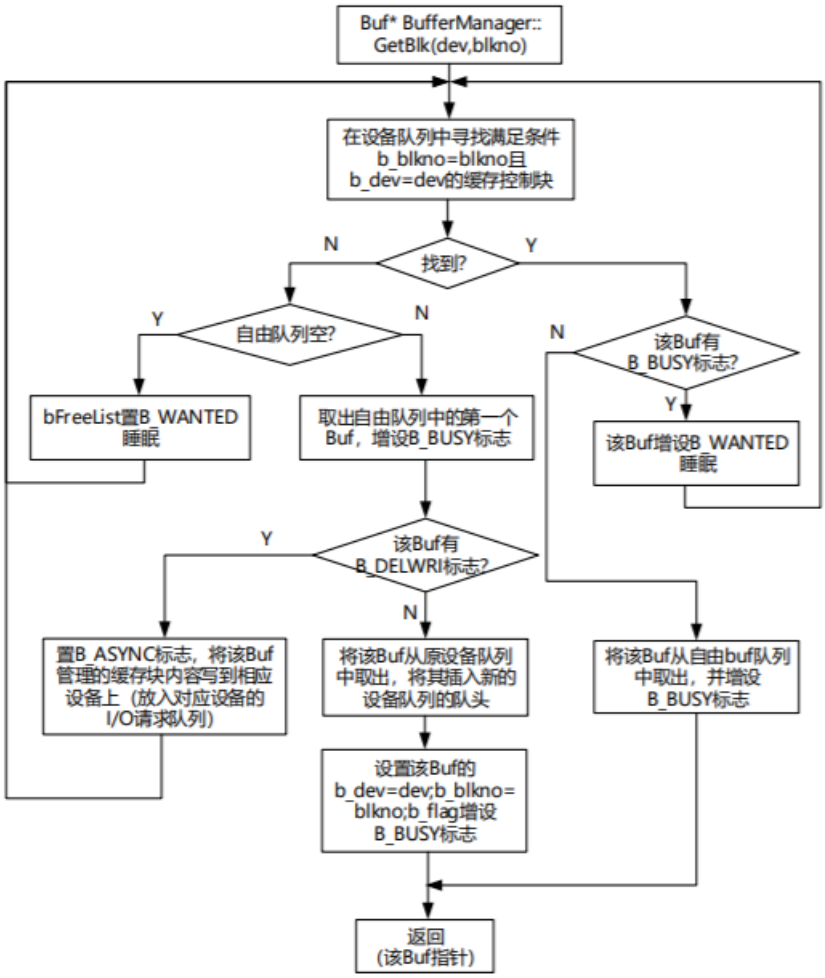
采用 LRU 算法，每次将插入的节点放置在缓存队列的最尾部，视为最近使用的缓存节点。

3.2.3 申请缓存函数

```
Buffer* BufferManager::GetBlk(int blkno)
```

申请一块缓存，将其从缓存队列中取出，用于读写设备块上的 blkno。执行过程为：

首先，从缓存队列中寻找是否有缓存块的 blkno 为目标 blkno， 如有则分离该缓存节点，并返回该节点；没有找到说明缓存队列中没有相应节点为 blkno， 需要分离第一个节点，将其从缓存队列中删除。若其带有延迟写标志，则立即写回，清空标志，将缓存 blkno 置为目标blkno，返回该缓存块。



3.3 用户操作接口

3.3.1 fopen 函数

将参数传入 User 结构中后，直接调用 FileManager 类中提供的 Open 接口。

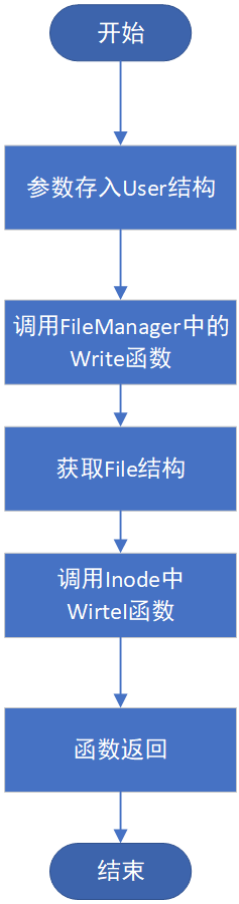
调用 Namel 函数后以传入的 mode 模式调用 Open1函数，执行 完则 Open 函数返回。

Namel 的主要功能是通过 User 结构中的 pathname 以及指向当前目录的 inode 指针，通过目录文件一层层索引按文件名找到对应打开文件的 inode 并将其返回，若是找不到则返回 NULL。

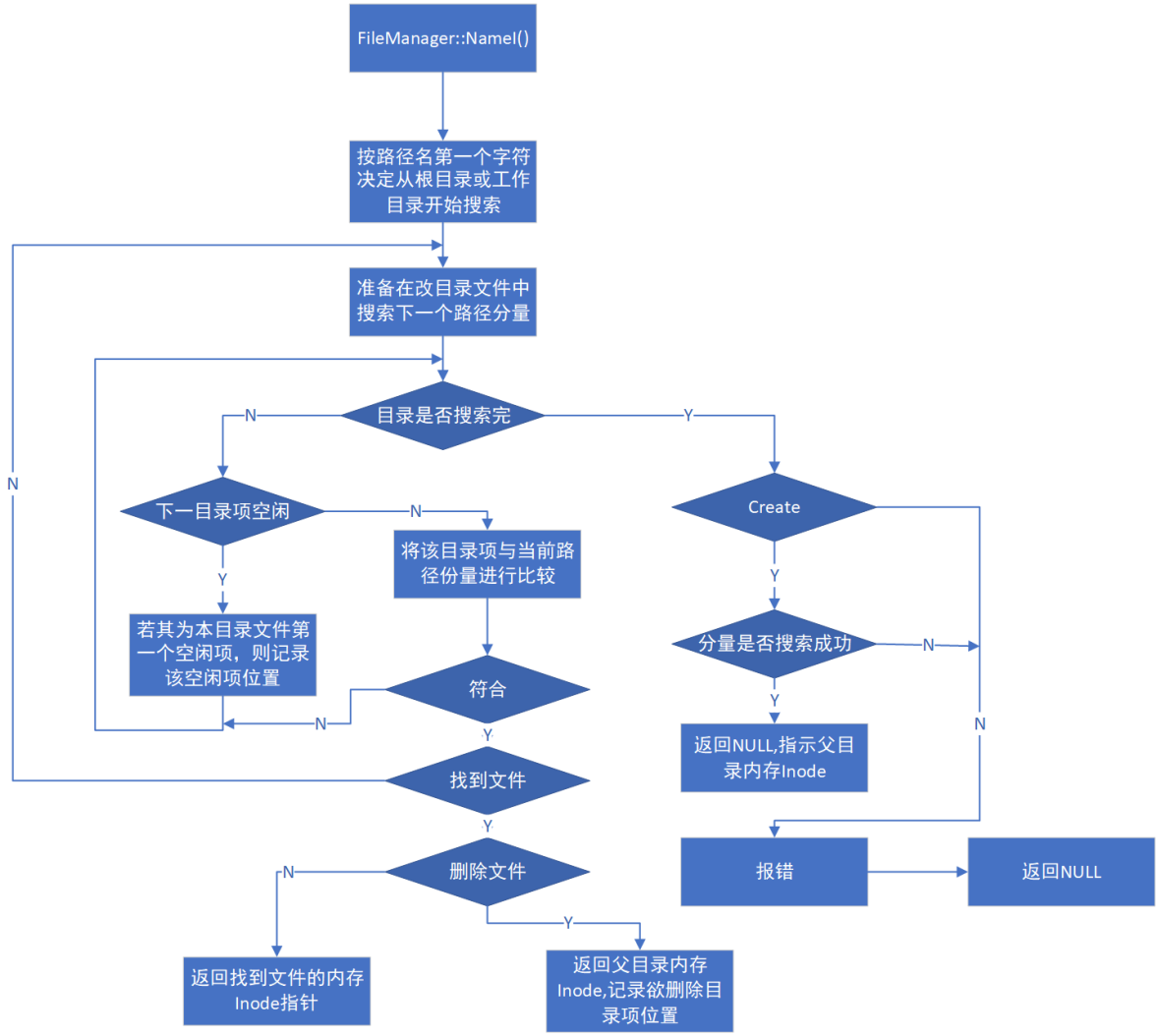
```
void User::Fopen(string fileName, string mode) {
    if (!checkPathName(fileName)) {
        return;
    }
    int md = FileMode(mode);
    if (md == 0) {
        cout << "this mode is undefined ! \n";
        return;
    }

    arg[1] = md;
    fileManager->Open();
    if (IsError())
        return;
    cout << "open success, return fd=" << ar0[EAX] << endl;
}
```

3.3.2 fwrite 函数

在User中的部分：



在Writel中的调用过程：

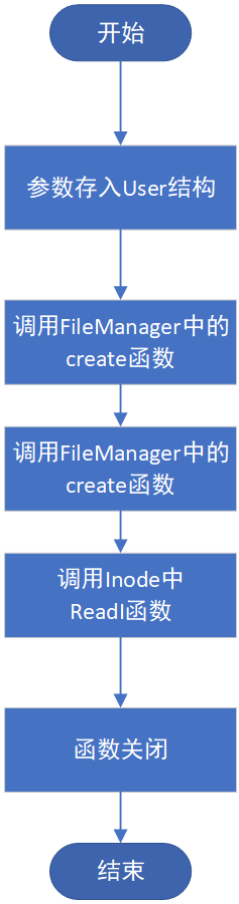


3.3.3 copy函数

通过调用create、open、read等接口函数，完成系统内部文件的拷贝。

```
void User::Copy(string srcFile, string desfile) {
    if (!checkPathName(srcFile)) {
        return;
    }

    arg[1] = File::FREAD;
    fileManager->Open();
    if (IsError())
        return;
    int fread = ar0[EAX];

    if (!checkPathName(desfile)) {
        return;
    }

    arg[1] = INode::IREAD | INode::IWRITE;
    fileManager->Creat();
    if (IsError())
        return;
    arg[1] = File::FWRITE;
    fileManager->Open();
    if (IsError())
        return;
```

```
    int fwrite = ar0[EAX];

    int usize = 1024;

    char* buffer = new char[usize];


    arg[1] = (long)buffer;
    arg[2] = usize;
    ar0[User::EAX] = usize;

    int size = 0;
    while (ar0[User::EAX] == usize) {
        arg[0] = fread;
        arg[2] = usize;
        fileManager->Read();
        if (IsError())
            return;
        size += ar0[User::EAX];



        arg[0] = fwrite;
        arg[2] = ar0[User::EAX];
        fileManager->Write();
        if (IsError())
            return;
    }
    arg[0] = fread;
    fileManager->Close();
    if (IsError())
        return;
    arg[0] = fwrite;
    fileManager->Close();
    if (IsError())
        return;
    cout << "File " << srcFile << "copy to file " << desfile << " Success, Write
" << size << " bytes done ! " << endl;
    delete[]buffer;
}
```



3.3.4 more函数

以ASCI码的形式显示文件内容，给出提示：

```
void User::More(string srcFile) {

    if (!checkPathName(srcFile)) {
        return;
    }

    arg[1] = File::FREAD;
    fileManager->Open();
    if (IsError())
        return;

    int usize = 1024;

    char* buffer = new char[usize+1];

    buffer[usize] = 0;
    arg[0] = ar0[EAX];
    arg[1] = (long)buffer;
    arg[2] = usize;
    ar0[User::EAX] = usize;

    while (ar0[User::EAX] == usize) {
        fileManager->Read();
        if (IsError())
            return;
        if (ar0[User::EAX] != usize)
            buffer[ar0[User::EAX]] =0;
```

```
        cout << buffer << endl;
        cout << "\nDisplay " << ar0[User::EAX] << " bytes," << "Press any key
continue ...";
        _getch();
    }
    cout << endl;
    fileManager->Close();
    IsError();

    delete[]buffer;
}
```

四.调试分析

4.1 测试与分析

以部分指令为例，给出指令测试

4.1.1 man 指令测试

输入正确指令，给出相应提示：

```
Usage Demo    :  man mkdir
[unix2nd-fs / ]$ man fcreate
Command       :  fcreate -新建文件
Description   :  新建指定文件。若出现错误，提示错误信息!
Usage         :  fcreate〈文件名〉〈选项〉
Parameter     :  〈文件名〉可以包含相对路径或绝对路径
                 〈选项〉-r 只读属性
                 〈选项〉-w 只写属性
                 〈选项〉-rw 读写属性
Usage Demo    :  fcreate newfile -rw
                 fcreate ../newfile -rw
                 fcreate ../../newfile -rw
                 fcreate /newfile -rw
                 fcreate /dir/newfile -rw
Error Avoided :  文件名过长，目录路径不存在，目录超出根目录等
```

输入错误指令，提示不存在：

```
[unix2nd-fs / ]$ man stu
shell : stu : don't find this commond
```

4.1.2 mkdir 指令测试

输入正确指令，给出相应提示：

```
Usage Demo    :  man mkdir
[unix2nd-fs / ]$ mkdir can
[unix2nd-fs / ]$ ls
can

[unix2nd-fs / ]$
```

4.1.3 fcreate 指令测试

在子目录下完成文件的创建：

```
[unix2nd-fs / ]$ cd can
[unix2nd-fs /can/ ]$ fcreate file1 -rw
[unix2nd-fs /can/ ]$ ls
file1

[unix2nd-fs /can/ ]$
```

缺少参数给出提示：

```
[unix2nd-fs /can/ ]$ fcreate file2
this mode is undefined !
[unix2nd-fs /can/ ]$
```

4.1.4 fopen 指令测试

打开一个已经建的文件，给出文件符：

```
[unix2nd-fs /can/ ]$ fopen file1 -rw
open success, return fd=4
```

如果文件不存在，返回错误提示：

```
[unix2nd-fs /can/ ]$ fopen file2 -w
errno = 2 No such file or directory
[unix2nd-fs /can/ ]$
```

4.1.5 fwrite、fread、fseek、more指令测试

将测试文件file2000.txt写入file1中

```
[unix2nd-fs /can/ ]$ fopen file1 -rw
open success, return fd=4
[unix2nd-fs /can/ ]$ fwrite 4 file2000.txt 100
write 100bytes success !
[unix2nd-fs /can/ ]$ more file1
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000


Display 100 bytes,Press any key continue ...
```

测试文件：

file2000.txt - 记事本

文件(F) 编辑(E) 格式(O) 查看(V) 帮助(H)

```
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444
5555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555
6666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666
7777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777
8888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
9999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
0000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
1111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
2222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
3333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
4444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444
5555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555
6666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666
7777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777
8888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888888
9999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999999
```

fread函数测试（注意读文件时，移动文件指针到文件头）

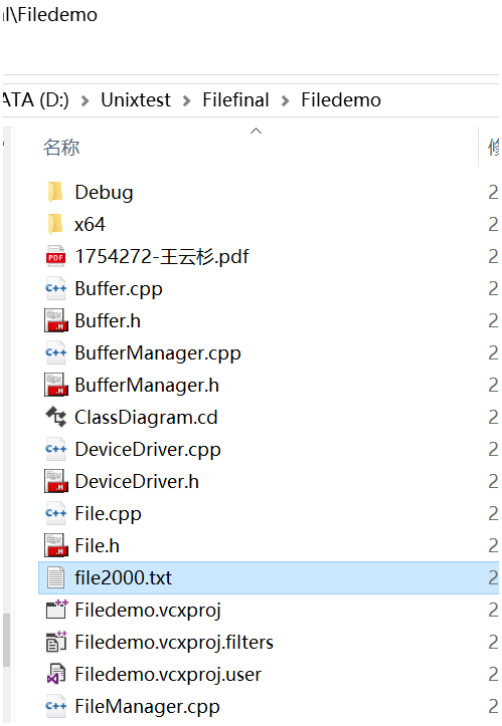
```
[unix2nd-fs /can/ ]$ fseek 2 0 0
[unix2nd-fs /can/ ]$ fread 2 3
read 3 bytes success :
000
[unix2nd-fs /can/ ]$
```

4.1.6 copy、copyin、copyout 指令测试

copy指令测试，将文件file1拷贝到新文件file2：

```
[unix2nd-fs /can/ ]$ copy file1 file2
File file1copy to file file2 Success, Write 10 bytes done !
[unix2nd-fs /can/ ]$ ls
file1
file2

[unix2nd-fs /can/ ]$ more file2
0000000000

Display 10 bytes,Press any key continue ...
```

copyin指令测试，拷入外部文件：

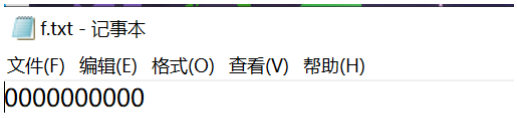



```
[unix2nd-fs /can/ ]$ copyin 1754272-王云杉.pdf file2.pdf
External file 1754272-王云杉.pdf copy to file2.pdf Success, Write 16968 bytes done !
[unix2nd-fs /can/ ]$ ls
file1
file2
file2.pdf

[unix2nd-fs /can/ ]$
```

copyout指令测试：

将内部文件拷出：

```
[unix2nd-fs /can/ ]$ copyout file1 f.txt
File file1copy to external file f.txt Success, Write 10 bytes done !
[unix2nd-fs /can/ ]$
```



可验证拷贝成功。

4.2 问题与解决方案

4.2.1 Inode分配冲突

在读入文件时，产生文件过大而报错的情况：

通过调试，发现是因为未判断inode 是否已写入磁盘导致，加入代码判断后调试成功

```
        if (gc_INodeTable.IsLoaded(ino) == -1) {
            superBlock->s_inode[superBlock->s_ninode++] = ino;
            if (superBlock->s_ninode >= SuperBlock::MAX_NINODE) {
                break;
            }
```

4.2.2 buffer 块清空

在buffer读入缓存的时候，发现程序崩溃中止，经过调试，发现是没有对flag进行判断，加入后程序调试正确。

```
/* 读一个磁盘块，blkno为目标磁盘块逻辑块号。 */
Buffer* BufferManager::Bread(int blkno) {
    Buffer* pb = GetBlk(blkno);
    if (pb->flags&(Buffer::B_DONE | Buffer::B_DELWRI)) {
        return pb;
    }
    deviceDriver->read(pb->addr, BUFFER_SIZE, pb->blkno*BUFFER_SIZE);
    pb->flags |= Buffer::B_DONE;
    return pb;
}

/* 写一个磁盘块 */
```

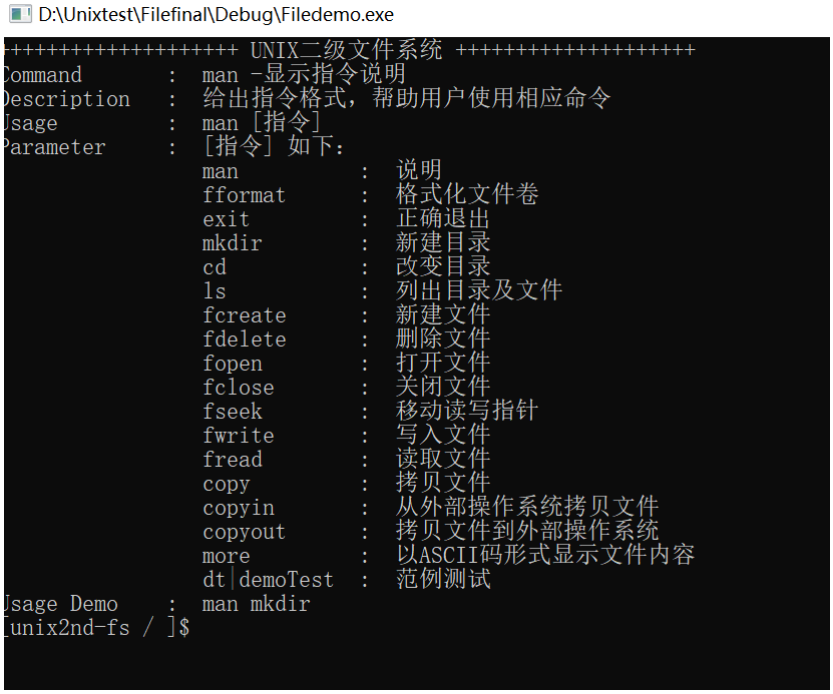
4.2.3 读函数读入字节为空

在进行fread函数测试的时候发现不论读入什么文件，得到结果都为空，

经过调试，发现是由于文件指针未移动导致，通过移动文件指针，使得读函数得以正常运行。

五.用户使用说明

Windows 环境下直接运行生成的exe文件即可



Linux环境下，先通过makefile文件进行编译，然后运行可执行文件：

```
CC = g++
CXX_FLAGS = -std=c++11 -w
# gdb调试选项 -g -rdynamic

BIN = unix2nd-fs

$(BIN): main.o Buffer.o BufferManager.o DeviceDriver.o File.o FileManager.o FileSystem.o INode.o OpenFileManager.o User.o Utility.o
    $(CC) -o $@ $^
main.o: main.cpp
    $(CC) $(CXX_FLAGS) -c -o $@ $<
Buffer.o: Buffer.cpp
    $(CC) $(CXX_FLAGS) -c -o $@ $<
BufferManager.o: BufferManager.cpp
    $(CC) $(CXX_FLAGS) -c -o $@ $<
DeviceDriver.o: DeviceDriver.cpp
    $(CC) $(CXX_FLAGS) -c -o $@ $<
File.o: File.cpp
    $(CC) $(CXX_FLAGS) -c -o $@ $<
FileManager.o: FileManager.cpp
    $(CC) $(CXX_FLAGS) -c -o $@ $<
FileSystem.o: FileSystem.cpp
    $(CC) $(CXX_FLAGS) -c -o $@ $<
INode.o: INode.cpp
    $(CC) $(CXX_FLAGS) -c -o $@ $<
OpenFileManager.o: OpenFileManager.cpp
    $(CC) $(CXX_FLAGS) -c -o $@ $<
User.o: User.cpp
    $(CC) $(CXX_FLAGS) -c -o $@ $<
Utility.o: Utility.cpp
    $(CC) $(CXX_FLAGS) -c -o $@ $<

.phony:
clean:
    rm -f *.o $(BIN)
    rm -f *.img
```

```
spruce@LAPTOP-N6M6JTQG:~/filesystem$ make
g++ -std=c++11 -w  -c -o main.o main.cpp
g++ -std=c++11 -w  -c -o Buffer.o Buffer.cpp
g++ -std=c++11 -w  -c -o BufferManager.o BufferManager.cpp
g++ -std=c++11 -w  -c -o DeviceDriver.o DeviceDriver.cpp
g++ -std=c++11 -w  -c -o File.o File.cpp
g++ -std=c++11 -w  -c -o FileManager.o FileManager.cpp
g++ -std=c++11 -w  -c -o FileSystem.o FileSystem.cpp
g++ -std=c++11 -w  -c -o INode.o INode.cpp
g++ -std=c++11 -w  -c -o OpenFileManager.o OpenFileManager.cpp
g++ -std=c++11 -w  -c -o User.o User.cpp
```

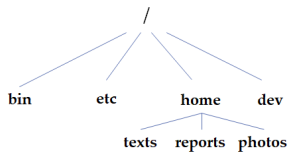
直接运行编译好的可执行文件./unix2nd-fs即可

六. 运行结果说明

> - 格式化文件卷；
> - 用mkdir命令创建子目录，建立如图所示的目录结构；
> - 把随意的一个纯文本文件，你的课设报告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，写入800个字节；
> - 将文件读写指针定位到第500字节，读出20个字节。

使用编写好的自动指令进行测试：

- 建立如图所示的目录结构：



```
[unix2nd-fs / ]$ dt
DemoTest beginning, Please press enter to continue ...
[unix2nd-fs / ]$ mkdir /bin
[unix2nd-fs / ]$ ls
test
bin

[unix2nd-fs / ]$ mkdir etc
[unix2nd-fs / ]$ cd etc
[unix2nd-fs /etc/ ]$ ls

[unix2nd-fs /etc/ ]$ mkdir /dev
[unix2nd-fs /etc/ ]$ cd /
[unix2nd-fs / ]$ mkdir home
[unix2nd-fs / ]$ mkdir /home/texts
[unix2nd-fs / ]$ copyin File.h /home/texts/file.h
External file File.h copy to /home/texts/file.h Success, Write 1449 bytes done !
[unix2nd-fs / ]$ ls
test
bin
etc
dev
home
```

调用user结构的mkdir函数，在FileManager中调用create函数：进入函数后，搜索目录的模式为1，表示创建；若父目录不可写，出错返回；没有找到相应的INode，或NameI出错；如果创建的名字不存在，使用参数trf = 2来调用open1()。如果NameI()搜索到已经存在要创建的文件，则清空该文件（用算法ITrunc()）。

- 使用copy指令拷入报告和图片：

```
[unix2nd-fs / ]$ cd /home
[unix2nd-fs /home/ ]$ mkdir photos
[unix2nd-fs /home/ ]$ cd photos
[unix2nd-fs /home/photos/ ]$ copyin tp.jpg tpnew.jpg
External file tp.jpg copy to tpnew.jpg Success, Write 7877 bytes done !
[unix2nd-fs /home/photos/ ]$ cd /home
[unix2nd-fs /home/ ]$ mkdir reports
[unix2nd-fs /home/ ]$ ls
texts
photos
reports

[unix2nd-fs /home/ ]$ cd reports
[unix2nd-fs /home/reports/ ]$ copyin 1754272-王云杉.pdf 1754272-王云杉.pdf
External file 1754272-王云杉.pdf copy to 1754272-王云杉.pdf Success, Write 16968 bytes done !
[unix2nd-fs /home/reports/ ]$ ls
1754272-王云杉.pdf
```

调用user结构的copy函数，先调用FileManager中的create接口，创建新文件，再打开新文件；调用read函数，从源文件中每次读出1024个字节直到文件尾，将读出的内容放在缓存数组中；调用write函数，向新文件中写入文件内容。

- 新建文件并完成指针的移动：

```
D:\Unixtest\Filefinal\Debug\Filedemo.exe
[unix2nd-fs /home/reports/ ]$ cd /
[unix2nd-fs / ]$ mkdir test
[unix2nd-fs / ]$ cd test
[unix2nd-fs /test/ ]$ fcreate Jerry -rw
[unix2nd-fs /test/ ]$ fopen Jerry -rw
open success, return fd=12
[unix2nd-fs /test/ ]$ fwrite 12 file2000.txt 800
write 800bytes success !
[unix2nd-fs /test/ ]$ fclose 12
[unix2nd-fs /test/ ]$ more Jerry
00000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000000
11111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111111
22222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222222
33333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333333
44444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444444
55555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555555
66666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666666
77777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777777


Display 800 bytes,Press any key continue ...
[unix2nd-fs /test/ ]$ fopen Jerry -rw
open success, return fd=12
[unix2nd-fs /test/ ]$ fseek 12 500 0
[unix2nd-fs /test/ ]$ fread 12 20
read 20 bytes success :
55555555555555555555
[unix2nd-fs /test/ ]$ fclose 12
[unix2nd-fs /test/ ]$ DemoTest finished ...
[unix2nd-fs /test/ ]$
```

调用user结构的fseek接口，调用FileManager的seek根据参数的要求（文件头、尾）进行指针偏移量的移动。

- 格式化：

```
[unix2nd-fs / ]$ ls
test
bin
etc
dev
home

[unix2nd-fs / ]$ fformat
Format finish, System must reboot, Press any key exit ...
```

```
gc_OpenFileTable.Format();
gc_INodeTable.Format();
gc_BufferManager.FormatBuffer();
gc_FileSystem.FormatDevice();
cout << "Format finish, System must reboot, Press any key exit ...";
_getch();

exit(0);
```

清空OpenFileTable、INodeTable、BufferManager、FileSystem结构

七.实验总结

通过本次操作系统课程设计，对Unix V6++系统中的缓存、文件系统管理都有了进一步的认识。在实践的过程中，对C语言的运用和理解，系统分析和设计架构的能力、调试能力、分析错误调试程序的能力都有较大的提升。

要想使项目的整体规划和模块划分具有较好的合理性，做需要首先对整个项目有一个较好的全局认识，然后在此基础上进行系统架构的设计和模块的划分。先确定好模块的大致功能，模块之间的相互调用关系，每个功能模块类的定义，函数接口的定义，才能一步一步完成编程。

在上学期的学习中，对于很多Unix V6++系统的实际运行还比较模糊，像高速缓存队列、用户接口调用、Inode、superblock的理解还很浅显，通过阅读源码、查阅资料，上网搜索，对于这些概念都有了更进一步的理解。在移植源码的过程中，由于移植一个所需的函数，往往要调用十几个更多的函数提供支持，所以需要不断来回阅读和修改。同时，也需要删除不需要的接口与函数，最初会错删或者错改一些变量，在调试的过程中，通过修正相应的错误，完成了本次二级文件系统的构造。

当然本次的课设还存在非常多的不足，还有更高层的API可以进行优化，多用户的访问可以添加，由于时间和时间精力原因止步于此。我希望在之后的学习中，可以更好地理解操作系统的不同部分，同时对本次课程设计的内容进行进一步的完善。

八.参考文献

1.莱昂氏UNIX源代码分析

2.Unix内核源码剖析.[日]青柳隆宏

3.课本 UNIX V6++块设备管理及文件系统