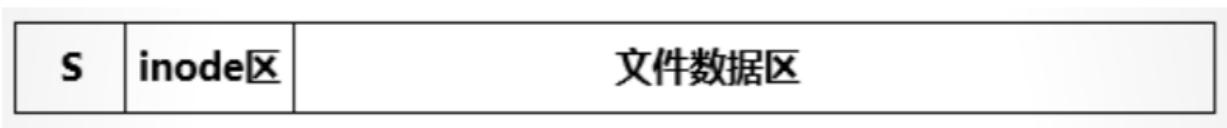


一、题目要求

使用一个普通的大文件（如 c:\myDisk.img，称之为一级文件）模拟 UNIX V6++的一个文件卷，一个文件卷实际上就是一张逻辑磁盘，磁盘中存储的信息以块为单位。每块 512字节。



1. 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. 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. 文件打开结构: 选作

4. 磁盘高速缓存: 选作

5. 文件操作接口:

- `void fformat();` 格式化文件卷
- `void ls();` 列目录
- `int fopen(char *name, int mode);` 打开文件
- `void fclose(int fd);` 关闭文件
- `int fread(int fd, char *buffer, int length);` 读文件
- `int fwrite(int fd, char *buffer, int length);` 写文件
- `int fseek(int fd, int position);` 定位文件读写指针
- `int fcreat(char *name, int mode);` 新建文件
- `int fdelete(char *name);` 删除文件

6. 主程序:

- 初始化文件卷，读入 SuperBlock;
- 图形界面或者命令行方式，等待用户输入；
- 根据用户的不同的输入，返回结果。
-

二、需求分析

根据对题目要求分析和用户使用分析，文件系统采用控制台命令作为输入，按照系统给定的命令即可完成相应功能。控制台的命令应包括如下：

```
" man : 手册 \n"
" fformat : 格式化 \n"
" exit : 正确退出 \n"
" mkdir : 新建目录 \n"
" cd : 改变目录 \n"
" ls : 列出目录及文件 \n"
" create : 新建文件 \n"
" delete : 删除文件 \n"
" open : 打开文件 \n"
" close : 关闭文件 \n"
" seek : 移动读写指针 \n"
" write : 写入文件 \n"
" read : 读取文件 \n"
" at|autoTest : 自动测试 \n"
```

每条命令具体分析如下：

命令: **fformat** =

```
"Command : fformat -进行文件系统格式化 \n"
"Description : 将整个文件系统进行格式化，即清空所有文件及目录! \n"
"Usage : fformat \n"
"Parameter : 无 \n"
"Usage Demo : fformat \n"
;"
```

命令: **exit** =

```
"Command : exit -退出文件系统 \n"
"Description : 若要退出程序，最好通过exit命令。此时正常退出会调用析构函数，\n"
" : 若有在内存中未更新到磁盘上的缓存会及时更新，保证正确性。若点 \n"
" : 击窗口关闭按钮，属于给当前程序发信号强制退出，不会调用析构函 \n"
" : 数，未写回部分信息，再次启动时可能出现错误! \n"
"Usage : exit \n"
"Parameter : 无 \n"
"Usage Demo : exit \n"
;"
```

```
命令: mkdir =
"Command : mkdir -建立目录 \n"
"Description : 新建一个目录。若出现错误，会有相应错误提示！\n"
"Usage : mkdir <目录名> \n"
"Parameter : <目录名> 可以是相对路径，也可以是绝对路径 \n"
"Usage Demo : mkdir dirName \n"
"  mkdir ../dirName \n"
"  mkdir ../../dirName \n"
"  mkdir /dirName \n"
"  mkdir /dir1/dirName \n"
"Error Avoided : 目录名过长，目录路径不存在，目录超出根目录等 \n"
;
```

```
命令: ls =
"Command : ls -列目录内容 \n"
"Description : 列出当前目录中包含的文件名或目录名。若出现错误，会有相应错误提示！\n"
"Usage : ls \n"
"Parameter : 无 \n"
"Usage Demo : ls \n"
;
```

```
命令: cd =
"Command : cd -改变当前目录 \n"
"Description : 改变当前工作目录。若出现错误，会有相应错误提示！\n"
"Usage : cd <目录名> \n"
"Parameter : <目录名>默认为当前目录；\n"
" <目录名> 可以是相对路径，也可以是绝对路径 \n"
"Usage Demo : ls \n"
"  ls ../dirName \n"
"  ls ../../dirName \n"
"  ls /dirName \n"
"  ls /dir1/dirName \n"
"Error Avoided : 目录名过长，目录路径不存在，目录超出根目录等 \n"
;
```

```
命令: create =
"Command : create -新建文件 \n"
"Description : 新建一个文件。若出现错误，会有相应错误提示！\n"
"Usage : create <文件名> <选项> \n"
"Parameter : <文件名> 可以包含相对路径或绝对路径 \n"
" <选项> -r 只读属性 \n"
" <选项> -w 只写属性 \n"
" <选项> -rw == -r -w 读写属性 \n"
"Usage Demo : create newFileName -rw \n"
" create ../newFileName -rw \n"
" create ../../newFileName -rw \n"
" create /newFileName -rw \n"
" create /dir1/newFileName -rw \n"
"Error Avoided : 文件名过长，目录路径不存在，目录超出根目录等 \n"
;
```

```
命令: delet =
"Command : delete -删除文件 \n"
"Description : 删除一个文件。若出现错误，会有相应错误提示！\n"
"Usage : delete <文件名> \n"
"Parameter : <文件名> 可以包含相对路径或绝对路径 \n"
"Usage Demo : delete fileName \n"
" delete ../fileName \n"
" delete ../../fileName \n"
" delete /fileName \n"
" delete /dir1/fileName \n"
"Error Avoided : 文件名过长，目录路径不存在，目录超出根目录等 \n"
;
```

```
命令: open =
"Command : open -打开文件 \n"
"Description : 类Unix|Linux函数open, 打开一个文件。若要进行文件的读写必须先open。 \n"
" 若出现错误, 会有相应错误提示! \n"
"Usage : open <文件名> <选项> \n"
"Parameter : <文件名> 可以包含相对路径或绝对路径 \n"
" <选项> -r 只读属性 \n"
" <选项> -w 只写属性 \n"
" <选项> -rw == -r -w 读写属性 \n"
"Usage Demo : open fileName -r \n"
" open ../fileName -w \n"
" open ../../fileName -rw \n"
" open /fileName -r -w \n"
" open /dir1/fileName -rw \n"
"Error Avoided : 文件名过长, 目录路径不存在, 目录超出根目录等 \n"
;
```

```
命令: close =
"Command : close -关闭文件 \n"
"Description : 类Unix|Linux函数close, 关闭一个文件。可以对已经打开的文件进行关闭\ n"
" 若出现错误, 会有相应错误提示! \n"
"Usage : close <file descriptor> \n"
"Parameter : <file descriptor> 文件描述符 \n"
"Usage Demo : close 1 \n"
"Error Avoided : 文件描述符不存在或超出范围 \n"
;
```

```
命令: seek =
"Command : seek -写入文件 \n"
"Description : 类Unix|Linux函数fseek, 写入一个已经打开的文件中。若出现错误, 会有相应错误提示! \n"
"Usage : seek <file descriptor> <offset> <origin> \n"
"Parameter : <file descriptor> open返回的文件描述符 \n"
" <offset> 指定从 <origin> 开始的偏移量 可正可负 \n"
" <origin> 指定起始位置 可为0 SEEK_SET), 1 SEEK_CUR), 2 SEEK_END) \n"
"Usage Demo : seek 1 500 0 \n"
"Error Avoided : 文件描述符不存在或超出范围, 未正确指定参数 \n"
;
```

```
命令: write =
"Command : write -写入文件 \n"
"Description : 类Unix|Linux函数write, 写入一个已经打开的文件中。若出现错误, 会有相应错误提示! \n"
"Usage : write <file descriptor> <InFileName> <size> \n"
"Parameter : <file descriptor> open返回的文件描述符 \n"
" <InFileName> 指定写入内容为文件InFileName中的内容 \n"
" <size> 指定写入字节数, 大小为 <size> \n"
"Usage Demo : write 1 InFileName 123 \n"
"Error Avoided : 文件描述符不存在或超出范围, 未正确指定参数 \n"
;
```

```
命令: read =
"Command : read -读取文件 \n"
"Description : 类Unix|Linux函数read, 从一个已经打开的文件中读取。若出现错误, 会有相应错误提示! \n"
"Usage : read <file descriptor> [-o <OutFileName>] <size> \n"
"Parameter : <file descriptor> open返回的文件描述符 \n"
" [-o <OutFileName>] -o 指定输出方式为文件, 文件名为 <OutFileName> 默认为shell \n"
" <size> 指定读取字节数, 大小为 <size> \n"
"Usage Demo : read 1 -o OutFileName 123 \n"
" read 1 123 \n"
"Error Avoided : 文件描述符不存在或超出范围, 未正确指定参数 \n"
;
```

```
命令: autoTest =
"Command : autoTest -自动测试 \n"
"Description : 帮助测试, 在系统启动初期帮助测试。测试不一定所有命令都是正确的, 但是系统具有容错性, \n"
" : 不会使系统异常。 \n"
"Usage : autoTest | at \n"
"Parameter : 无 \n"
"Usage Demo : at \n"
;
```

三、概要设计

3.1 模块划分

经过分析，整个文件系统可以由以下几个部分组成：

- **DeviceDriver:** 设备驱动模块，直接负责磁盘文件直接读写。
- **BufferManager:** 高速缓存管理模块，主要负责管理系统中所有的缓存块，包括申请、释放、读写、清空一块缓存的功能函数接口，以及系统退出时刷新所有缓存块。

- **FileSystem:** 系统盘块管理模块，主要负责对镜像文件的存储空间管理，包括SuperBlock 空间占用、DiskINode 空间分布、数据块区空间分布的管理。需要提供分配、回收 DiskINode 节点、数据块节点以及格式化磁盘文件的接口。
- **FileManager:** 系统文件操作功能实现模块，主要封装文件系统中对文件处理的操作过程，负责对文件系统访问的具体细节。包括打开文件、创建文件、关闭文件、Seek 文件指针、读取文件、写入文件、删除文件等系统功能实现。
- **OpenFileManager:** 打开文件管理模块，负责对打开文件的管理，为用户打开文件建立数据结构之间的勾连关系，为用户提供直接操作文件的文件描述符接口。
- **User:** 用户操作接口模块，主要将用户的界面执行命令转化为对相应函数的调用，同时对输出进行处理，也包含检查用户输入的正确性与合法性。

3.2 数据结构定义

3.2.1 DeviceDriver: 设备驱动模块

```

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();

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

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

```

3.2.2 BufferManager: 高速缓存管理模块

```
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();

    /* 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);

```

```

class Buffer {
public:

    /* flags中标志位 */
    enum BufferFlag { ... };

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();

    void debugMark();
    void debugContent();
};


```

3.2.3 FileSystem: 系统盘块管理模块

```
/*
 * 文件系统存储资源管理块(Super Block)的定义。
 */
class SuperBlock {
public:
    const static int MAX_NFREE = 100;
    const static int MAX_NINODE = 100;

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[MAX_NFREE]; // 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[MAX_NINODE]; // 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
    int s_ronly;            // 本文件系统只能读出
    int s_time;              // 最近一次更新时间
    int padding[47];         // 填充使SuperBlock块大小等于1024字节，占据2个扇区
};

class DirectoryEntry {
public:
    static const int DIRSIZ = 28; /* 目录项中路径部分的最大字符串长度 */

public:
    int m_ino;           /* 目录项中INode编号部分 */
    char name[DIRSIZ];   /* 目录项中路径名部分 */
};

class FileSystem {
public:
```

```

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();

```

```

/* 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);

};

```

3.2.4 FileManager: 系统文件操作功能实现模块

```

class FileManager
{
public:
    /* 目录搜索模式，用于NameI()函数 */
    enum DirectorySearchMode { ... };
}

```

```
public:  
    /* 根目录内存INode */  
    INode* rootDirINode;  
  
    /* 对全局对象g_FileSystem的引用，该对象负责管理文件系统存储资源 */  
    FileSystem* fileSystem;  
  
    /* 对全局对象g_INodeTable的引用，该对象负责内存INode表的管理 */  
    INodeTable* inodeTable;  
  
    /* 对全局对象g_OpenFileTable的引用，该对象负责打开文件表项的管理 */  
    OpenFileTable* openFileTable;
```

```
public:  
    FileManager();  
    ~FileManager();  
  
    /* Open()系统调用处理过程 */  
    void Open();  
  
    /* Creat()系统调用处理过程 */  
    void Creat();  
  
    /* Open()、Creat()系统调用的公共部分 */  
    void Open1(INode* pINode, int mode, int trf);  
  
    /* Close()系统调用处理过程 */  
    void Close();  
  
    /* Seek()系统调用处理过程 */  
    void Seek();  
  
    /* Read()系统调用处理过程 */  
    void Read();  
  
    /* Write()系统调用处理过程 */  
    void Write();  
  
    /* 读写系统调用公共部分代码 */  
    void Rdwr(enum File::FileFlags mode);  
  
    /* 目录搜索，将路径转化为相应的INode返回上锁后的INode */  
    INode* NameI(enum DirectorySearchMode mode);
```

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

/* 取消文件 */
void UnLink();

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

/* 改变文件访问模式 */
//void ChMod();

/* 改变当前工作目录 */
void ChDir();

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

3.2.5 OpenFileManager: 打开文件管理模块

```
/*
 * 打开文件控制块File类。
 * 该结构记录了进程打开文件的读、写请求类型，文件读写位置等等。
 */
class File {
public:
    enum FileFlags { ... };

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 */
};
```

```
/*
 * 文件I/O的参数类
 * 对文件读、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。
 */
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 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，指向系统打开的文件 */

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);
};
```

3.2.6 User: 用户操作接口模块

```
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 { ... };

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 */
    chardbuf[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空间 */

    FileManager* fileManager;
}
```

```

public:
    User();
    ~User();

    void Ls();
    void Cd(string dirName);
    void Mkdir(string dirName);
    void Create(string fileName, string mode);
    void Delete(string fileName);
    void Open(string fileName, string mode);
    void Close(string fd);
    void Seek(string fd, string offset, string origin);
    void Write(string fd, string inFile, string size);
    void Read(string fd, string outFile, string size);
    //void Pwd();

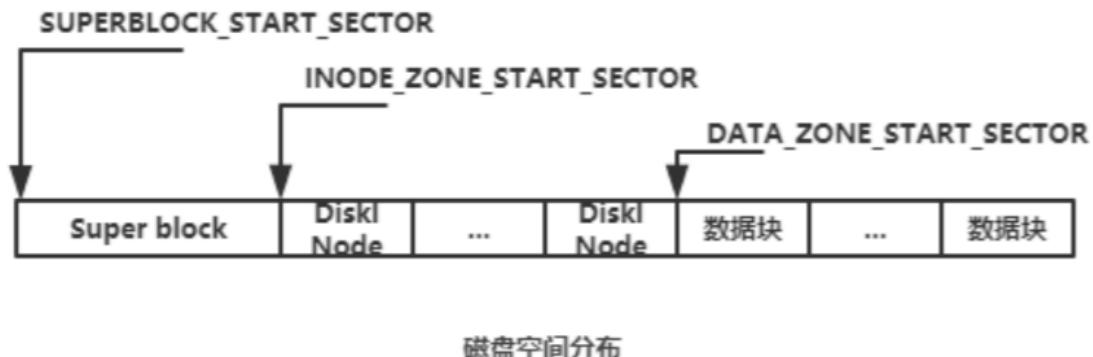
private:
    bool IsError();
    void EchoError(enum ErrorCode err);
    int INodeMode(string mode);
    int FileMode(string mode);
    bool checkPathName(string path);

};

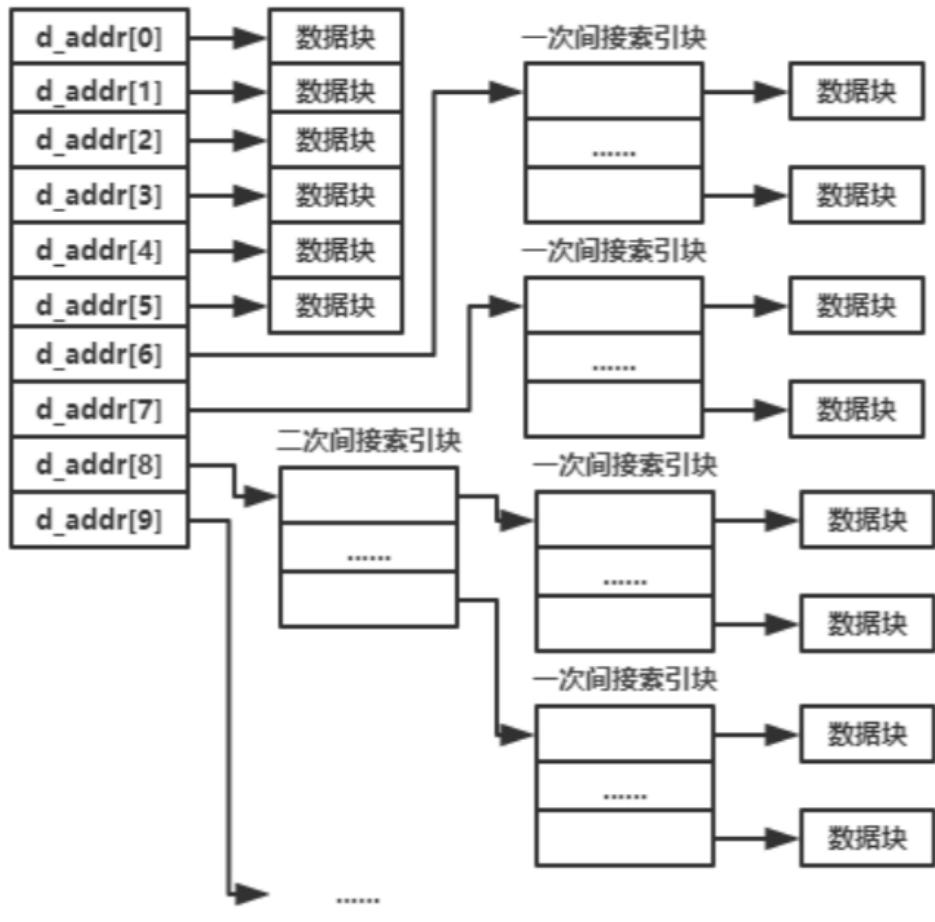
```

3.3 数据结构图示

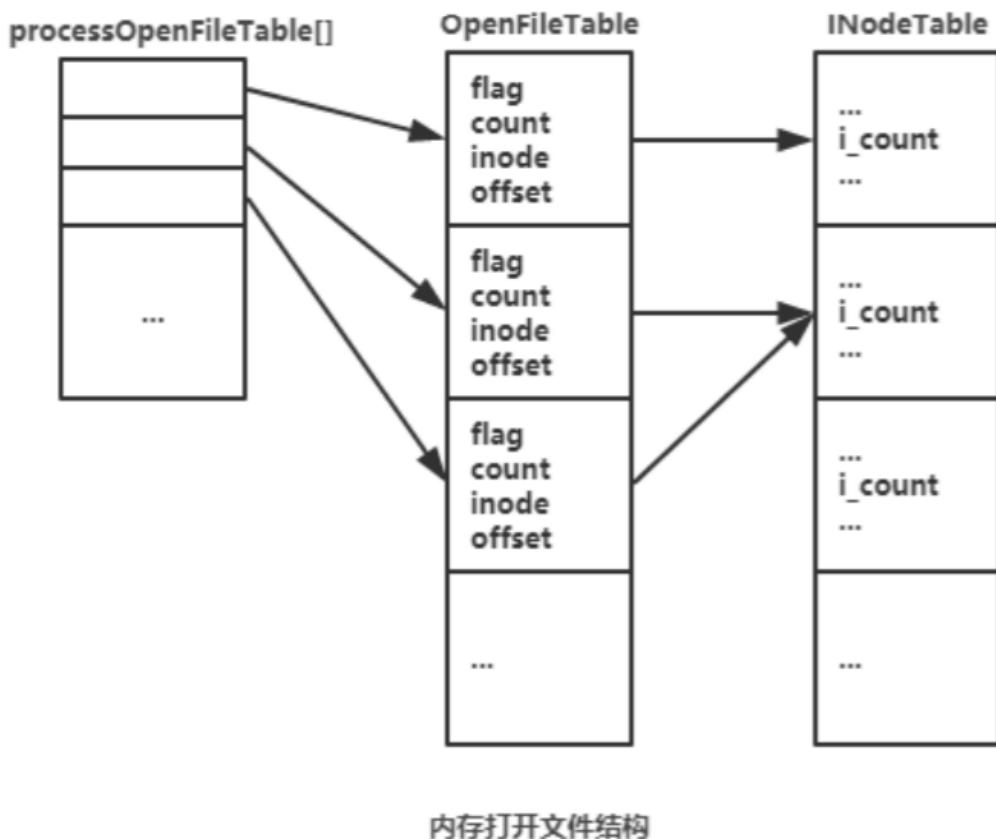
3.3.1 磁盘存储空间分布:



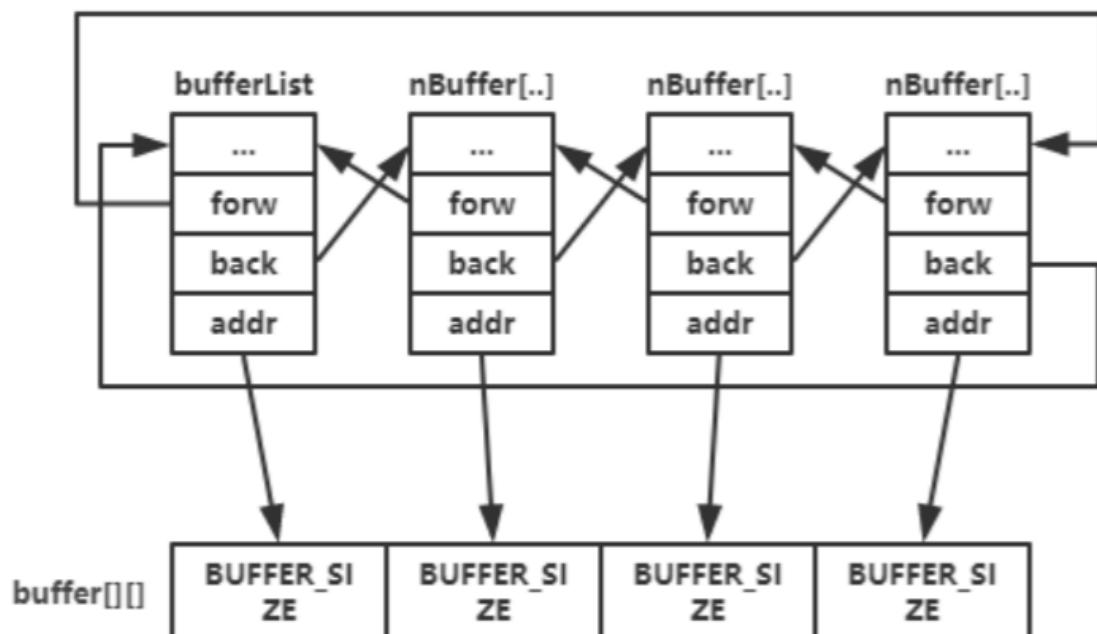
3.3.2 文件索引结构:



3.3.3 内存文件打开结构:



3.3.4 高速缓存数据结构:

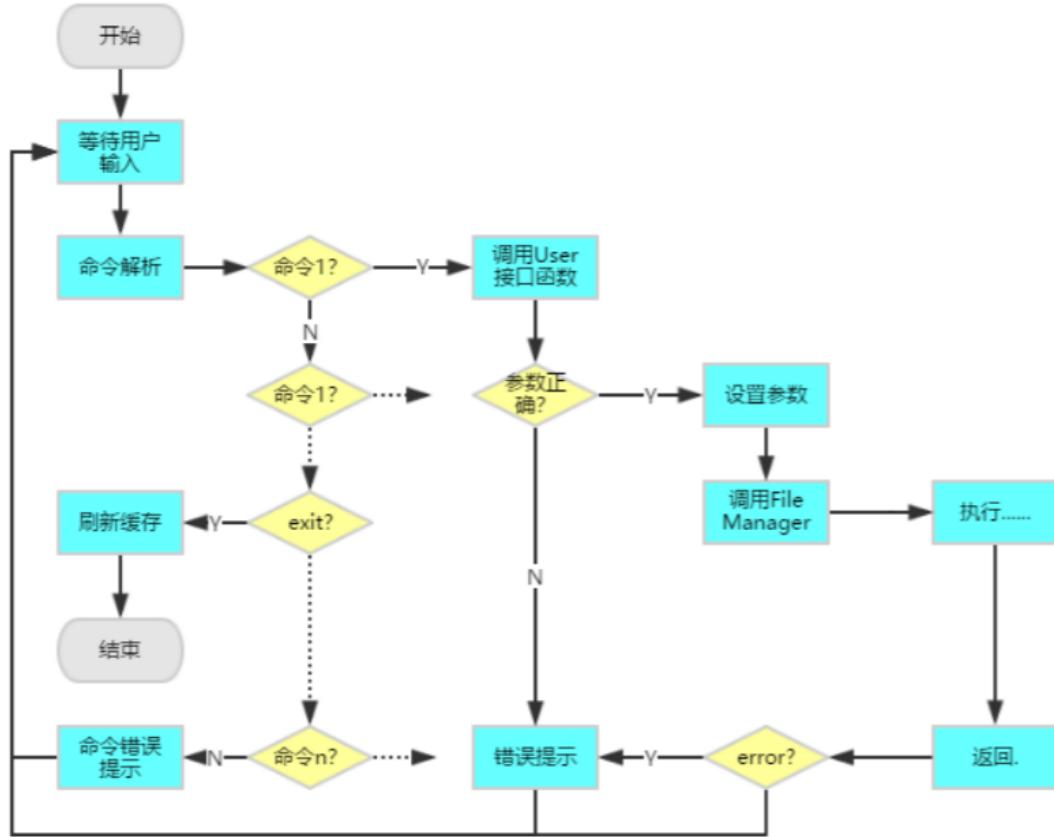


高速缓存数据结构

3.4 总体流程

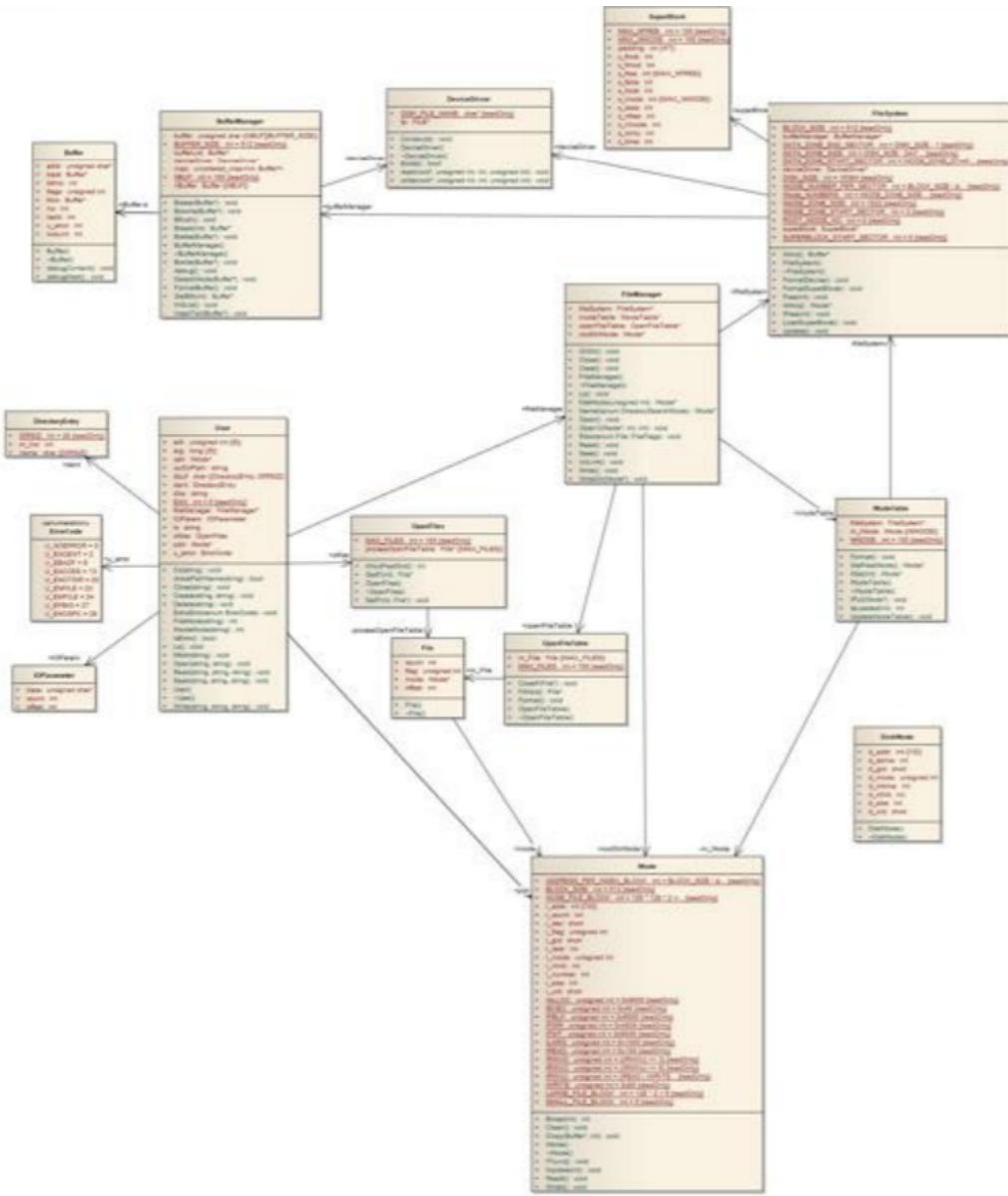
程序的总体流程为：main 函数等待用户输入，用户输入后，对其命令进行简单解析，然后由命令查找 User 提供的操作接口，将参数交由 User 函数进行处理和判断合法性，若基本符合命令的参数约定，则由User调用 FileManager中功能函数实现文件系统的具体功能。而 FileManger 中函数的具体功能则依赖于 FileSystem、BufferManager、OpenFileManager的相关函数接口调用。

总体流程图如下：



3.5 模块调用关系

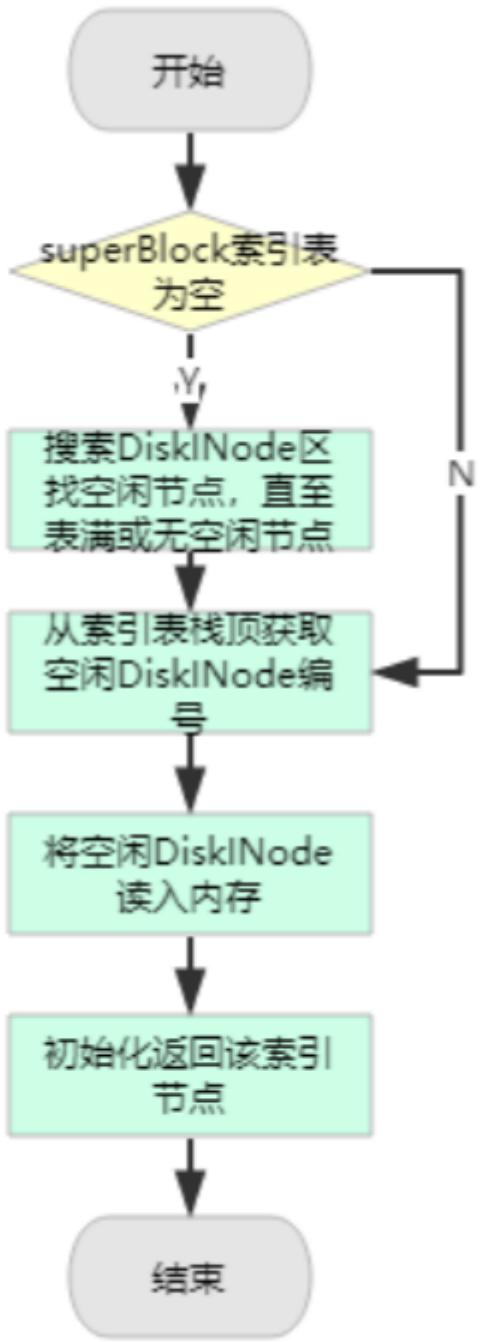
下图为模块间调用关系图示：



四、详细设计

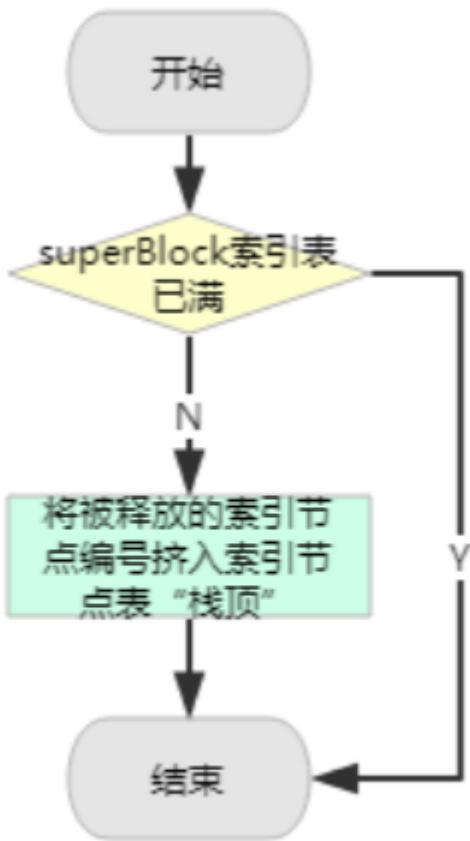
4.1 外存磁盘文件管理

- 外存索引节点分配 IAlloc 函数：外存 DiskINode 索引节点采用栈式管理 DiskINode，当需要分配 DiskINode 时，如果 s_ninode 不为 0，则将栈顶节点分配，栈指针减一；否则 s_ninode 为 0，说明外存索引节点表中已不包含任何空闲节点，就重新搜索整个 DiskINode 区，将找到的 DiskINode 号顺次登入 s_inode 表中，直到该表已满或者已搜索完整个 DiskINode 区。



FileSystem::IAlloc 函数流程

- 外存索引节点分释放 IFree 函数：当释放 DiskINode 节点时，如果超级块索引节点表中空闲 DiskINode 数小于 SuperBlock::MAX_NINODE，则将该索引节点编号记入“栈顶”，若其中记录的空闲 DiskINode 已满，则将其散落在磁盘 DiskINode 区中。



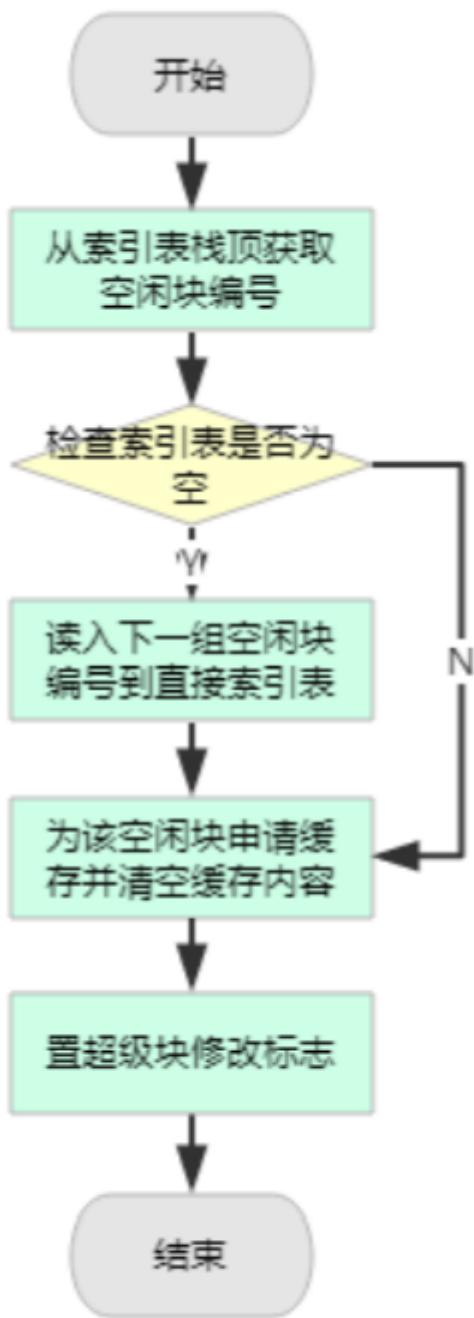
FileSystem::IFree函数流程

- 外存空闲盘块分配 Alloc 函数：外存中的空闲盘块采用分组链式索引法进行管理。超级块中的空闲块索引表用栈式管理空闲数据块，但是它最多只能直接管理 MAX_NFREE（100）个空闲块。所有空闲块按照每 MAX_NFREE（100）个进行构成一组，最后一组直接由超级块中的空闲索引表进行管理，其余各组的索引表分别存放在它们下一组第一个盘块的开头中。

空闲盘块分组链式索引示例：



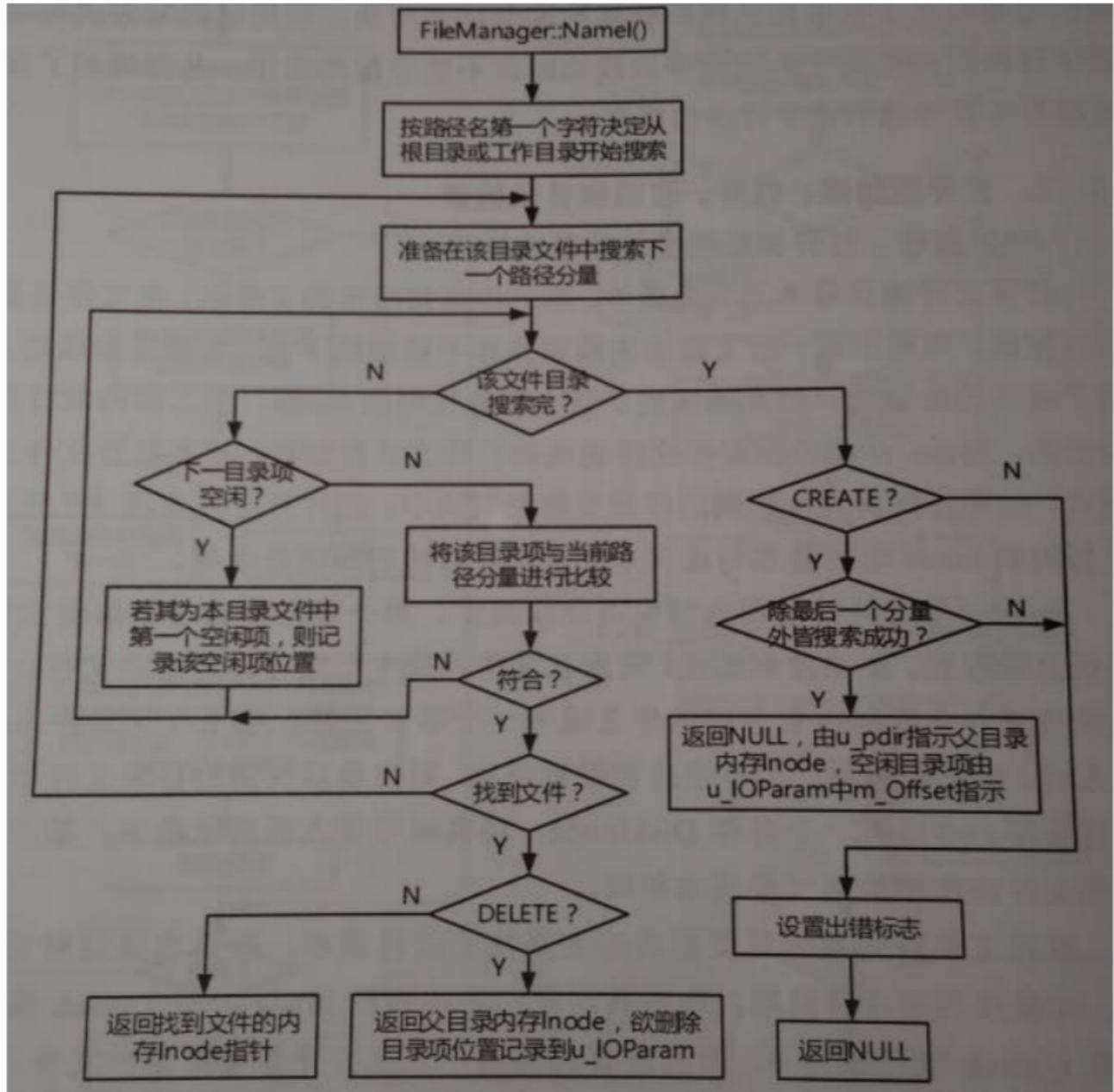
分配空闲盘块时，总是从索引表中取其最后一项的值，即 `s_free[--s_nfree]`, 相当于出栈，当及时直接管理的最后一个空闲盘块时，就将该盘块的前 404 字节读入超级块的 `s_nfree` 和索引表 `s_free` 数组中，使得用间接方式管理的下一组变为直接管理。



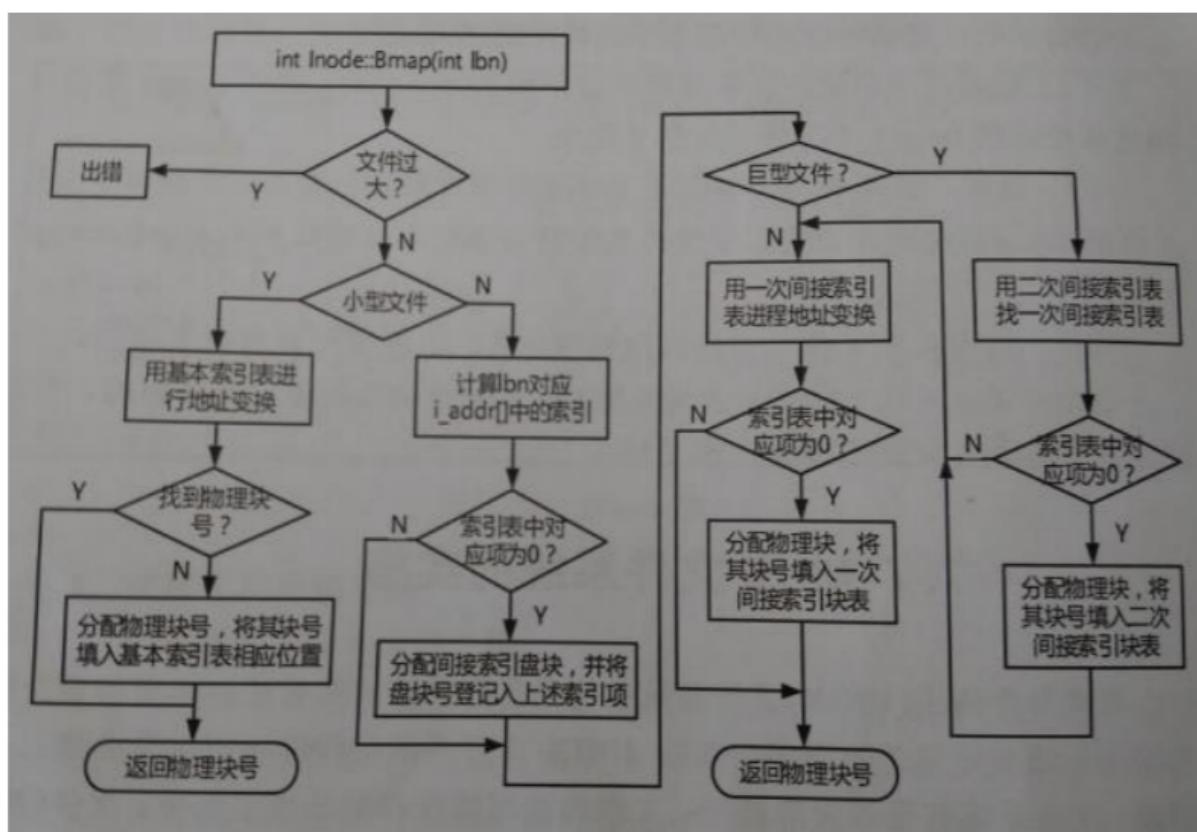
FileSystem::Alloc 函数流程

4.2 目录搜索和地址变换

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



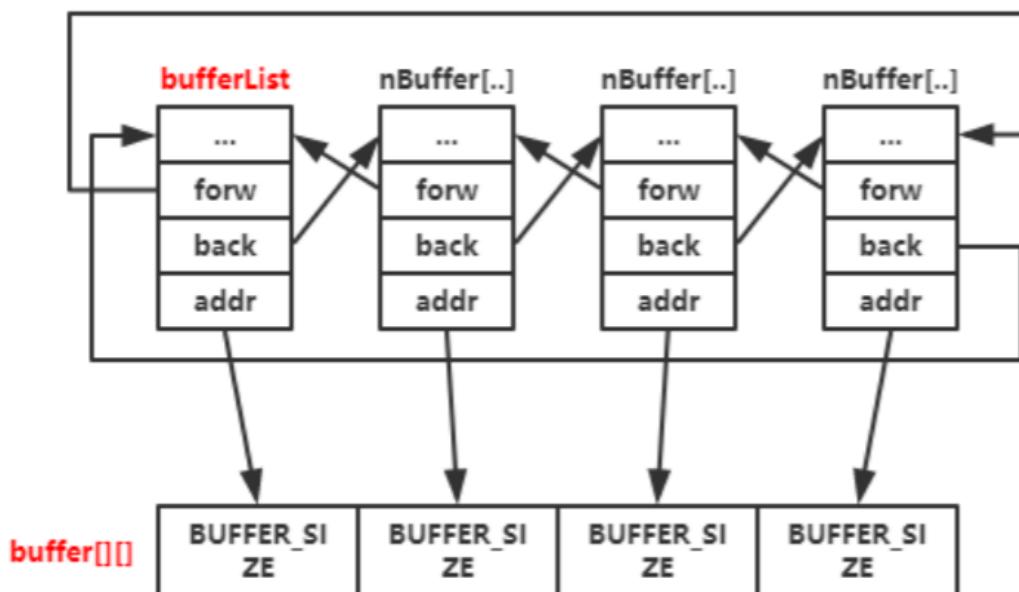
- 地址变换 Bmap 函数：将文件逻辑号变换成文件在存储设备的物理块号，与 Unix 执行的 Bmap 函数基本类似，流程图如下：



4.3 内存高速缓存模拟实现

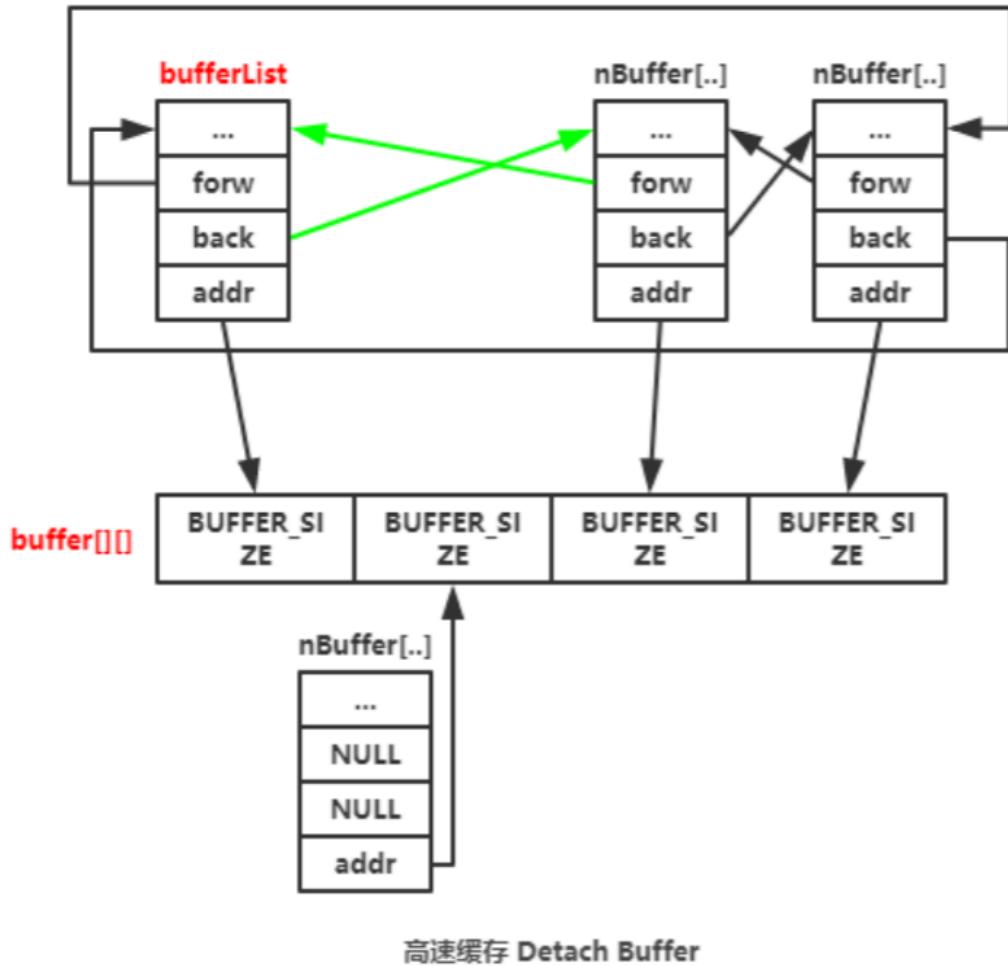
Unix 系统采用自由队列和设备队列进行对缓存块的管理，但是因为本次课程设计项目是模拟 Unix 文件系统，并且整个模拟时串行执行，无并行操作，所以经过考虑，可以将自由队列和设备队列进行合并为一个缓存队列，具体细节如下：

- 高速缓存初始化：即构造相应的双向循环链表，初始化成员变量。

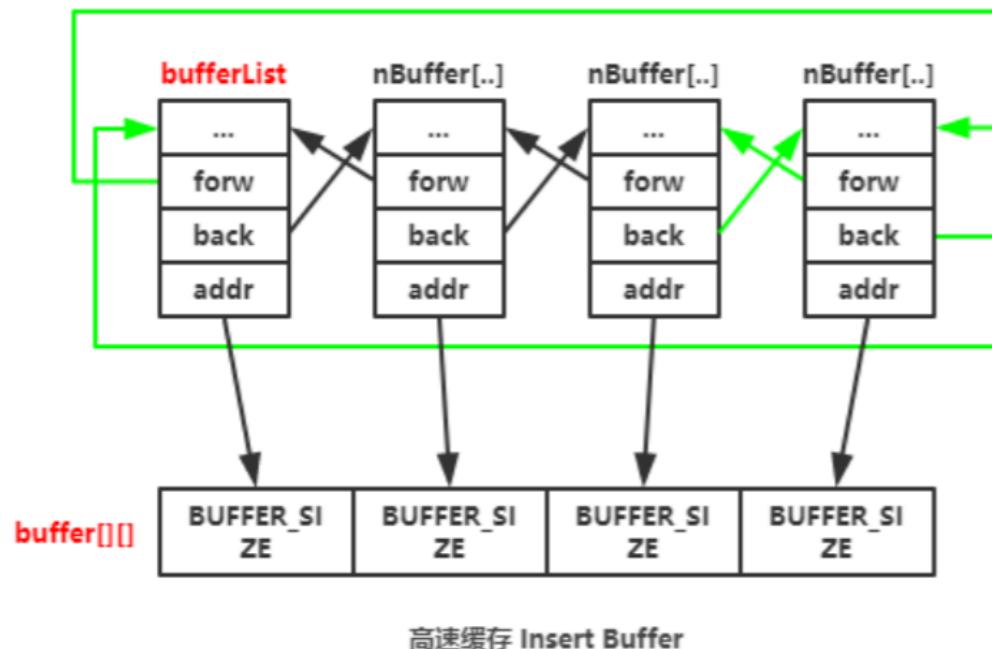


高速缓存数据结构 初始图示

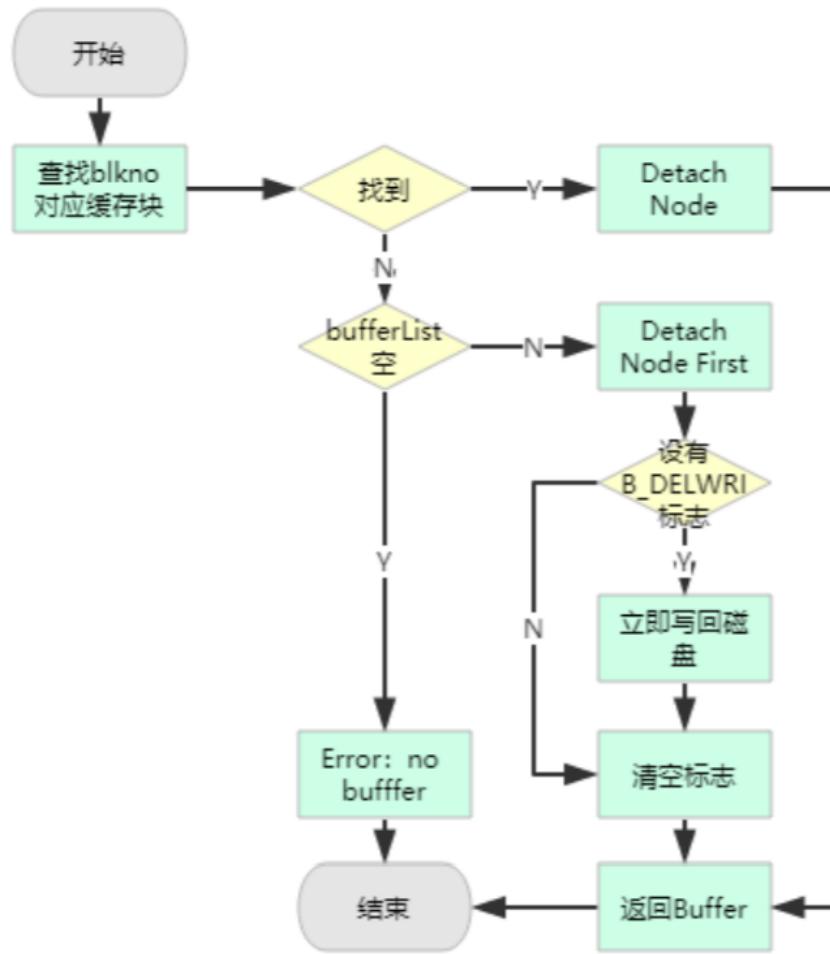
- 分离一个节点 DetashNode 函数：采用 LRU 算法，每次将第一个节点分离，第一个节点即为过去一段时间最久未使用的节点。



- 插入一个节点 InsertTail 函数：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。



- 申请一块缓存 GetBlk 函数：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 blkno，返回该缓存块。



高速缓存 GetBlk(blkno)函数流程

五、 调试分析

5.1 测试

为了方便自己程序调试和老师测试，添加了一个自动测试命令，用于测试二级文件系统容错性和正确性。自动测试命令较为完备，对各个环节均有考虑到，自动测试命令序列如下：

```
"mkdir /dir1",
"mkdir dir2",
"create file1 -rw",
"create file2 -rw",
"ls",
"delete /file2",
"ls",
"mkdir /dir1/dir11",
"create /dir1/file11-rw",
"cd dir1",
"ls",
"cd ..",
"ls",
"open file1-rw",
"write 6 file1-2000.txt 800",
"seek 6 500 0",
"read 6 20",
"seek 6 500 0",
"read 6-o readOut1.txt 20",
"seek 6-20 1",
"read 6 100",
"close 6",
"cd dir1",
"ls",
"open file11-rw",
"write 6 file11-100000.txt 100000",
"seek 6 0 0",
"read 6 100",
"read 6 100",
"seek 6 50000 0",
"read 6 100"
```

测试结果如下：

```
Command      : man -显示在线帮助手册
Description  : 帮助用户使用相应命令
Usage       : man [命令]
Parameter   : [命令] 如下:
              man          : 手册
              fformat     : 格式化
              exit        : 正确退出
              mkdir       : 新建目录
              cd          : 改变目录
              ls          : 列出目录及文件
              create      : 新建文件
              delete      : 删除文件
              open        : 打开文件
              close       : 关闭文件
              seek        : 移动读写指针
              write       : 写入文件
              read        : 读取文件
              at|autoTest : 自动测试
Usage Demo  : man mkdir
[unix-fs / ]#
[unix-fs / ]# autoTest
autoTest begin ...
just press enter to continue ...
[unix-fs / ]# mkdir /dir1
[unix-fs / ]# mkdir dir2
[unix-fs / ]# create file1 -rw
[unix-fs / ]# create file2 -rw
[unix-fs / ]# ls
dir1
dir2
file1
file2
```

```
[unix-fs / ]# delete /file2
[unix-fs / ]# ls
dir1
dir2
file1
```

```
[unix-fs / ]# mkdir /dir1/dir11
[unix-fs / ]# create /dir1/file11 -rw
[unix-fs / ]# cd dir1
[unix-fs /dir1/]# ls
dir11
file11
```

```
[unix-fs /dir1/]# cd ..  
[unix-fs /]# ls  
dir1  
dir2  
file1
```

```
[unix-fs /]# open file1 -rw  
open success, return fd=6  
[unix-fs /]# write 6 file1-2000.txt 800  
write 800bytes success !  
[unix-fs /]# seek 6 500 0  
[unix-fs /]# read 6 20  
read 20 bytes success :  
//begin000500bytes  
  
[unix-fs /]# seek 6 500 0  
[unix-fs /]# read 6 -o readOut1.txt 20  
read 20 bytes success :  
read to readOut1.txt done !  
[unix-fs /]# seek 6 -20 1  
[unix-fs /]# read 6 100  
read 100 bytes success :  
//begin000500bytes  
//begin000520bytes  
//begin000540bytes  
//begin000560bytes  
//begin000580bytes
```

```
[unix-fs /]# close 6  
[unix-fs /]# cd dir1  
[unix-fs /dir1/]# ls  
dir1  
file1
```

```
[unix-fs /dir1/]# open file11 -rw
open success, return fd=6
[unix-fs /dir1/]# write 6 file11-100000.txt 100000
write 100000bytes success !
[unix-fs /dir1/]# seek 6 0 0
[unix-fs /dir1/]# read 6 100
read 100 bytes success :
//begin000000bytes
//begin000020bytes
//begin000040bytes
//begin000060bytes
//begin000080bytes

[unix-fs /dir1/]# read 6 100
read 100 bytes success :
//begin000100bytes
//begin000120bytes
//begin000140bytes
//begin000160bytes
//begin000180bytes

[unix-fs /dir1/]# seek 6 50000 0
[unix-fs /dir1/]# read 6 100
read 100 bytes success :
//begin050000bytes
//begin050020bytes
//begin050040bytes
//begin050060bytes
//begin050080bytes

[unix-fs /dir1/]# autoTest finished ...
```

5.2 调试中的问题解决

- 问题 1：缓存刷新算法逻辑错误

经过基本的功能测试后，添加了最后的 exit 命令，由前面介绍的高速缓存模拟知道，在退出时，可能有些缓存块上带有延迟写标记，尚未更新到磁盘文件中，那么在退出时需要调用虚构函数，将 Buffer 中带有延迟写标志的数据块写入到磁盘中，没有细想将缓存队列进行线性扫描写回，错误代码如下：

```
/* 将队列中延迟写的缓存全部输出到磁盘 */
void BufferManager::Bflush() {
    Buffer* pb;
    for (pb = bufferList->forw; pb != bufferList; pb = pb->forw) {
        if ((pb->flags & Buffer::B_DELWRI)) { X
            this->Bwrite(pb);
        }
    }
}
```

已引发异常
引发了异常: 读取访问权限冲突。
pb 是 nullptr.

复制详细信息
▲ 异常设置
 引发此异常类型时中断
从以下位置引发时除外:
 FileSystem.exe
[打开异常设置](#) | [编辑条件](#)

调用 exit 命令后引发异常，进行了 NULL 指针的读写访问，经过仔细的逻辑思考，这个代码会造成死循环，并将循环队列尾节点的前后指针置为 NULL，接下来 pb 就变为了 NULL 指针，自然就错了。不应进行按照循环队列的次序写回，不应该进行调用 Bwrite 写回。纠错后代码如下：

```
/* 将队列中延迟写的缓存全部输出到磁盘 */
void BufferManager::Bflush() {
    Buffer* pb = NULL;
    for (int i = 0; i < NBUF; ++i) {
        pb = nBuffer + i;
        if ((pb->flags & Buffer::B_DELWRI)) {
            pb->flags &= ~Buffer::B_DELWRI;
            deviceDriver->write(pb->addr, BUFFER_SIZE, pb->blkno*BUFFER_SIZE);
            pb->flags |= Buffer::B_DONE;
        }
    }
}
```

- 问题 2：磁盘物理序号错写为数据分区盘块序号

在做自动测试的时候，发现小文件测试没有问题，本来想欢欢喜喜以为基本完成了，但是当把文件大小设置大 100000 字节约等于 100KB 文件大小时，就出现了下面的错误：No Sapce left on device!。显然 100KB 大小是绝对可以存储的下的，于是新的问题产生了。

```
[unix-fs /dir1/]# open file11 -rw
open success, return fd=6
[unix-fs /dir1/]# write 6 file11-100000.txt 100000
No Space On Device !
errno = 28 No space left on device
[unix-fs /dir1/]# seek 6 0 0
[unix-fs /dir1/]# read 6 100
read 100 bytes success :
//begin000000bytes
//begin000020bytes
//begin000040bytes
//begin000060bytes
//begin000080bytes

[unix-fs /dir1/]# read 6 100
read 100 bytes success :
//begin000100bytes
//begin000120bytes
//begin000140bytes
//begin000160bytes
//begin000180bytes

[unix-fs /dir1/]# seek 6 50000 0
[unix-fs /dir1/]# read 6 100
read 0 bytes success :
```

通过提示来看，不难定位错误位置，通过搜索错误关键词，在 FileSystem.cpp 文件中找到了相应位置。如下红色标记：

```

/* 在存储设备上分配空闲磁盘块 */
Buffer* FileSystem::Alloc() {
    int blkno;
    Buffer* pBuffer;
    User& u = g_User;

    /* 从索引表“栈顶”获取空闲磁盘块编号 */
    blkno = superBlock->s_free[--superBlock->s_nfree];

    /* 若获取磁盘块编号为零，则表示已分配尽所有的空闲磁盘块。 */
    if (blkno <= 0) {
        superBlock->s_nfree = 0;
        cout << "No Space On Device !\n";
        u.u_error = User::U_ENOSPC;
    }
    return NULL;
}

```

于是开始了问题分析：既然小文件没有问题，说明基本功能是没有问题的，文件大小为100KB 肯定会分配多个磁盘扇区，肯定是相应分配算法没有写对或者磁盘初始化的时候代码有问题，定位到磁盘初始化函数，经过仔细分析，发现将盘块在数据分区中序号写入 s_free表中，但是读取的时候是按照磁盘盘块的物理块号读取，结果就出现了以上问题。

```

//空闲盘块初始化
char freeBlock[BLOCK_SIZE], freeBlock1[BLOCK_SIZE];
memset(freeBlock, 0, BLOCK_SIZE);
memset(freeBlock1, 0, BLOCK_SIZE);

for (int i = 0; i < FileSystem::DATA_ZONE_SIZE; ++i) {
    if (superBlock->s_nfree >= SuperBlock::MAX_NFREE) {
        memcpy(freeBlock1, &superBlock->s_nfree, sizeof(int) + sizeof(deviceDriver->write(&freeBlock1, BLOCK_SIZE));
        superBlock->s_nfree = 0;
    }
    else {
        deviceDriver->write(freeBlock, BLOCK_SIZE);
    }
    superBlock->s_free[superBlock->s_nfree++] = i;
}

```

代码修改为：

```

superBlock->s_free[superBlock->s_nfree++] = i + DATA_ZONE_START_SECTOR;

```

- 问题 3：分配空闲盘块后清空 Buffer 内容的 Bclear 函数错写

解决上一个问题后，自以为这下总可以好好写报告了吧，天不遂人愿。测试没有问题，但是进行 exit 命令退出控制台时，控制台卡住几秒，然后磁盘镜像文件莫名变为了 GB 级大小。经过思考，我觉得文件这么大肯定是写入参数异常造成，而整个系统的读写都要通过 BufferManager 的 Bread 和 Bwrite 函数进行读写，在函数中加入 debug 函数，打印 Buffer 的相关标记信息，结果如下：

```
no = 43 blkno = 16244 flag = DONE DELWRI
no = 45 blkno = 812542311 flag = DONE DELWRI
write 100000bytes success!
```

最终的镜像文件变为了 GB 级就是因为这个 blkno 是 GB 大小的 blkno。blkno 发生错误，但是根本不知道是哪一个环节发生错误，那怎么找到错误原因呢？当然需要进一步分析何时这个编号为 45 的 Buffer 中的 blkno 变为了异常值，方法如下：

```
/* 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);
    ph->debugMark();
    if (pb->blkno == 812542311) {
        pb->debugContent();
    }
}
```

我可以在 Bread 函数中进行条件判断，当 blkno 变为 812542311 时，设置断点，查看调用堆栈，调用关系如下：

调用堆栈	
名称	
FileSystem.exe!INode::Write() 行 119	
FileSystem.exe!FileManager::Rdwr(File::FileFlags mode) 行 403	
FileSystem.exe!FileManager::Write() 行 371	
FileSystem.exe!User::Write(std::basic_string<char, std::char_traits<char>, std::allocator<char> const&) 行 355	
FileSystem.exe!main() 行 355	

发现调用函数为 INode.cpp 中的 Write() 函数，用同样的方法使函数停下来，然后查看相应局部变量。

局部变量	
名称	值
▶  this	FileSystem.exe!0x00fb2230 {i_flag=6 i_mode=33152 i_count=2 ...}
▶  bn	812542311
▶  bufferManager	{bufferList=0x00857a98 {flags=0 forw=FileSystem.exe!0x00fb46b0 ...}}
▶  lbn	195
▶  nbytes	160
▶  offset	0
▶  pBuffer	FileSystem.exe!0x00fb46f8 {flags=132 forw=0x00000000 <NULL> 0x00fb46f8}
▶  start	0x00fba8b4 "//begin097280bytes\r\n//begin097300bytes\r\\Q
▶  u	{u_cdir=FileSystem.exe!0x00fb20b4 {i_flag=6 i_mode=49152 i_count=2 ...}}

在此函数中进行单步执行，发现是在 Bmap() 函数返回时错误，继续在 Bmap 函数中添加代码使之在指定条件停住。

```
if (lbn == 195) {
    for (int i = 0; i < 128; ) {
        cout << "iTable[" << i << "]=" << iTable[i] << " ";
        if (++i % 5 == 0) {
            cout << endl;
        }
    }
}
```

发现 Buffer 中数据与预期不和，而数据的改变只能是两个地方，一个是初始化写入时不正确，另一个是读入 Buffer 中不正确，仔细看了一下初始化的代码，并没有错误。然后看了 `FileSystem.cpp` 中 `Alloc` 分配空闲盘块时会进行清空，看了一下清空的函数，结果大吃一惊，明明要清空整个 Buffer 的内容，偏偏不知道怎么就多了一个 `sizeof`。

```
/* 清空缓冲区内容 */
void BufferManager::Bclear(Buffer *bp) {
    Utility::memset(bp->addr, 0, sizeof(BufferManager::BUFFER_SIZE));
    return;
}
```

- 问题 4: vs 下编译运行正确而 Linux 下出现编译及运行错误

在 window 下 vs 集成环境下编译正确并且测试也正确之后，觉得整个工程文件有点庞大，不够简洁，决定自己写一个 Makefile 编译，简洁舒服。但是写好 Makefile 后在 Linux下使用 G++编译就报了错误，错误如下：

```
g++ -std=c++11 -w -g -rdynamic -c -o User.o User.cpp
User.cpp: 在成员函数 'void User::write(std::string, std::string, std::string'
' 中:
User.cpp:143:19: 错误: 从 'char*' 到 'int' 的转换损失精度 [-fpermissive]
    arg[1] = (int)buffer;
                  ^
make: *** [User.o] 错误 1
```

因为在 vs 下指针占用空间为四字节，int 也为四字节，没有问题。但是在 Linux 下的G++编译器中指针占用八字节空间，这样就会报精度损失错误。解决方法：将 arg 定义为 long 型数组，这样在 Linux 下大小为 8 字节，在 Window 下long 占用 4 字节。解决编译问题后，运行竟然报错了，错误为段错误。

```
[1453381@sdn os]$ ./unix-fs
+++++ Unix文件系统模拟 ++++++
段错误(吐核)
-----
```

在 window 下编译运行正确，但是在 Linux 下发生错误，肯定是两者编译器或系统有细微区别，要想找出问题，必须在 Linux 下进行调试，选用工具为 GDB 调试。在编译命令选项中加入-g -rdynamic 可以进行 gdb 调试，调试过程如下：进入 GDB 调试，运行命令 r

```
[1453381@sdn os]$ gdb ./unix-fs
GNU gdb (GDB) Red Hat Enterprise Linux 7.6.1-100.el7
Copyright (c) 2013 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/1453381/os/unix-fs...done.
(gdb)
(gdb) r
starting program: /home/1453381/os./unix-fs
```

运行中会在相应地方报段错误的信号 SIGSEGV，找到是什么函数导致段错误。

```
[unix-fs /]$ at
autoTest begin ...
just press enter to continue ...
[unix-fs /]$ mkdir /dir1

Program received signal SIGSEGV, segmentation fault.
0x000007ffff733ae40 in __memcpy_ssse3 () from /lib64/libc.so.6
Missing separate debuginfos, use: debuginfo-install glibc-2.17-196.el7.x86_64
-16.el7.x86_64
```

输入GDB命令bt打印当前函数调用栈，得知是在NameI函数中的memcpy调用出现问题。

```
(gdb) bt
#0 0x000007ffff733ae40 in __memcpy_ssse3 () from /lib64/libc.so.6
#1 0x000000000040dfd7 in utility::memcpy (dest=0x627030 <g_User+48>, src=
0x628329, n=4294967294) at utility.cpp:4
#2 0x0000000000409e58 in FileManager::NameI (this=0x626fe0 <g_FileManager
>, mode=FileManager::CREATE)
    at FileManager.cpp:117
#3 0x0000000000409c92 in FileManager::creat (this=0x626fe0 <g_FileManager
>) at FileManager.cpp:46
#4 0x000000000040cb41 in User::mkdir (this=0x627000 <g_User>, dirName="/d
ir1") at user.cpp:29
#5 0x0000000000403a6c in main () at main.cpp:324
```

再次运行，设置断点为 FileManager.cpp:115 行处，准备进行单步调试查看变量。

```
(gdb) break FileManager.cpp:115
Breakpoint 1 at 0x409de7: file FileManager.cpp, line 115.
(gdb) r
Starting program: /home/1453381/os./unix-fs
```

上面设置好断点以后，调试如下：

```

Breakpoint 1, FileManager::NameI (this=0x626fe0 <g_FileManager>,
    mode=FileManager::CREATE) at FileManager.cpp:115
115             nindex = u.u_dirp.find_first_of('/', index);
Missing separate debuginfos, use: debuginfo-install glibc-2.17-196.el7.x86_64
4 libgcc-4.8.5-16.el7.x86_64 libstdc++-4.8.5-16.el7.x86_64
(gdb) print nindex
$1 = 2
(gdb) step
116         utility::memset(u.u_dbuf, 0, sizeof(u.u_dbuf));
(gdb)
Utility::memset (s=0x627030 <g_User+48>, ch=0, n=28) at Utility.cpp:8
8             ::memset(s, ch, n);
(gdb)
9         }
(gdb)
FileManager::NameI (this=0x626fe0 <g_FileManager>,
    mode=FileManager::CREATE) at FileManager.cpp:117
117             utility::memcpy(u.u_dbuf, u.u_dirp.data() + index, (nindex =
= string::npos ? u.u_dirp.length() : nindex) - index);
(gdb) print nindex
$2 = 4294967295
(gdb) step
Utility::memcpy (dest=0x627030 <g_User+48>, src=0x628329, n=4294967294)
    at Utility.cpp:4
4             ::memcpy(dest, src, n);
(gdb) print n
$3 = 4294967294
(gdb) █

```

发现 unsigned int nindex 这个变量不等于 string::npos 值，大概知道是因为类型转换的问题，于是写了一个测试程序如下：

```

#include <iostream>
#include <string>
using namespace std;

int main() {
    string str="123456";
    int i=str.find('/');
    unsigned int ui=str.find('/');
    cout<<string::npos<<endl;
    cout<<"int i="<<i<<" == string::npos ? "
    |     <<(i==string::npos?"true":"false")<<endl;
    cout<<"unsigned int ui="<<ui<<" == string::npos ?
    |     <<(ui==string::npos?"true":"false")<<endl;

    return 0;
}

```

在 windows 下编译结果如下：

```
4294967295
int i=-1 == string::npos ? true
unsigned int ui=4294967295 == string::npos ? true
```

同样的代码放在 Linux 下结果却不一样，原因就在这。

```
[1453381@sdn ~]$ ./test
18446744073709551615
int i=-1 == string::npos ? true
unsigned int ui=4294967295 == string::npos ? false
```

由测试可以知道 Linux 64 位系统下 string::npos 的值为 unsigned long long 型，但是 windows 确是 unsigned int 型。程序错误就是 unsigned long long 截断为 unsigned int型再和 unsigned long long 型比较显然不会是 true，导致后续访问内存越界，程序段错误。更改代码如下，进行强制转换比较即可正确。

```
Utility::memcpy(u.u_dbuf, u.u_dirp.data() + index,
(nindex == (unsigned int)string::npos ? u.u_dirp.length() : nindex) - index);
index = nindex + 1;
```

结果验证：

```
open success, return fd=6
[unix-fs /dir1/]$ write 6 file11-100000.txt 100000
write 100000bytes success !
[unix-fs /dir1/]$ seek 6 0 0
[unix-fs /dir1/]$ read 6 100
read 100 bytes success :
//begin000000bytes
//begin000020bytes
//begin000040bytes
//begin000060bytes
//begin000080bytes

[unix-fs /dir1/]$ read 6 100
read 100 bytes success :
//begin000100bytes
//begin000120bytes
//begin000140bytes
//begin000160bytes
//begin000180bytes

[unix-fs /dir1/]$ seek 6 50000 0
[unix-fs /dir1/]$ read 6 100
read 100 bytes success :
//begin050000bytes
//begin050020bytes
//begin050040bytes
//begin050060bytes
//begin050080bytes

[unix-fs /dir1/]$ autoTest finished ...
[unix-fs /dir1/]$
[unix-fs /dir1/]$
[unix-fs /dir1/]$ █
```

六、 用户使用说明

- 项目结构:

```
[1453381@sdn os]$ tree
.
|-- Buffer.cpp
|-- Buffer.h
|-- BufferManager.cpp
|-- BufferManager.h
|-- DeviceDriver.cpp
|-- DeviceDriver.h
|-- file11-100000.txt
|-- file1-2000.txt
|-- File.cpp
|-- File.h
|-- FileManager.cpp
|-- FileManager.h
|-- FileSystem.cpp
|-- FileSystem.h
|-- INode.cpp
|-- INode.h
|-- main.cpp
|-- Makefile
|-- OpenFileManager.cpp
|-- OpenFileManager.h
|-- readOut1.txt
|-- User.cpp
|-- User.h
|-- Utility.cpp
`-- Utility.h
0 directories, 25 files
```

- 编译方法：
可以将源程序放在集成环境下编译，也可利用 GNU 编译工具，通过写好的 Makefile 进行编译，make 如下：

```
[1453381@sdn os]$ 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
g++ -std=c++11 -w -c -o Utility.o Utility.cpp
g++ -o unix-fs main.o Buffer.o BufferManager.o DeviceDriver.o File.o
FileManager.o FileSystem.o INode.o OpenFileManager.o User.o Utility.o
```

- 运行说明：

在 Windows 下直接点击生成的 exe 文件执行，在 Linux 平台直接./unix-fs 即可运行。运行界面为控制台的命令行方式，命令较为简单，通俗易懂，初始界面如下：

```
+++++ Unix 文件系统模拟 ++++++
Command : man - 显示在线帮助手册
Description : 帮助用户使用相应命令
Usage : man [命令]
Parameter : [命令] 如下:
man : 手册
fformat : 格式化
exit : 正确退出
mkdir : 新建目录
cd : 改变目录
ls : 列出目录及文件
create : 新建文件
delete : 删除文件
open : 打开文件
close : 关闭文件
seek : 移动读写指针
write : 写入文件
read : 读取文件
at|autoTest : 自动测试
Usage Demo : man mkdir
```

若对任何命令有不太清楚的地方，可直接 man [Command] 查看详细说明，以 read 命令为例：

```
[unix-fs / ]$ man read
Command : read -读取文件
Description : 类 Unix|Linux 函数 read, 从一个已经打开的文件中读取。若出现错误, 会有相应错误提示!
Usage : read <file descriptor> [-o <OutFileName>] <size>
Parameter : <file descriptor> open 返回的文件描述符
[-o <OutFileName>] -o 指定输出方式为文件, 文件名为
<OutFileName> 默认为 shell
<size> 指定读取字节数, 大小为 <size>
Usage Demo : read 1 -o OutFileName 123
read 1 123
Error Avoided : 文件描述符不存在或超出范围, 未正确指定参数
```

- 正确退出:
若要退出程序, 最好通过 exit 命令。此时正常退出会调用析构函数。若有内存中未更新到磁盘上的缓存会及时更新, 保证正确性。若点击窗口关闭按钮, 属于给当前程序发信号强制退出, 不会调用析构函数, 未写回部分信息, 再次启动时可能出现错误!
- 格式化:
格式化命令为 fformat, 运行命令后系统会进行文件系统格式化, 然后正常退出, 再次进入时为初始环境。

七、实验总结

这学期课程设计很多, 操作系统可能是自己花费时间精力最多的课程设计了。付出就有回报, 通过本次操作系统课程设计, 我觉得无论是系统分析和设计架构的能力, 编程语言C++的运用和理解, 还是寻找项目代码 bug 的能力, 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, 做好需求分析是非常重要的。如果对系统功能要求不够明确, 那么便不能很好的开展后续的工作。需求分析脚踏实地了, 才能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。具体就是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, 做好这些准备工作, 才能一步一步稳扎稳打的实际编程。

C++语言是建立在 C 语言之上的一门面向对象的语言, 虽然从计算机入门到现在个人技术栈一直都是以 C/C++语言为主, 语言的语法知识应该比较熟悉了, 但是仍从本次课程设计学到不少知识。例如类的静态成员数据是不占用类的大小的, 每一个类的占用空间大小是有规则定义, 可能同样的成员, 定义顺序不一样占用空间的大小是不一样的; 不同编译器对C++语法编译规则是有细微差别的, 不同操作系统运行一样的标准 C++语法代码也是有着一定区别。

本次课程设计时间跨度比较大, 本来准备先写好操作系统课程设计, 写到一半又被其它课程设计和作业无情中断, 导致前前后后费时较多。某些模块的小问题在这种来回切换的模式下被遗留忘记, 导致后面整体测试时大费周章的寻找错误。不过这样也有好处, 不知不觉中提高了自己调试代码的能力, 进一步的提升了如何分析错误的原因, 快速准确定位 bug所在。

Unix 系统是现代各类 Unix 操作系统的源头, Unix v6 是非常优秀的系统程序, 结构清晰, 短小精悍, Unix v6++又用面向对象的思想对其重新进行了设计, 并用 C++加以实现。本次课程设计就是仿照 Unix v6++中的文件系统进行改造而来, 在这过程中, 不免需要通透了解 Unix v6++的设计思想, 详细阅读 Unix v6++中的源码, 到现在可以说对其文件系统这一块的代码较为熟悉。但是 Unix v6++中的文件系统仅仅是其中的一个子部分, 还有进程管理、内存管理、设备管理等模块, 对整个 Unix 操作系统的进一步认识还有很长的路程要走。

八、 参考文献

[1] 尤晋元,UNIX 操作系统教程[M],西安:电子科技大学出版社,1985 年 6 月; [2] John Lions, 莱昂氏 UNIX 源代码分析 [M], 机械工业出版社,2006 年 8 月; [3] 汤小丹、梁红兵等,计算机操作系统[M], 西安:电子科技大学出版社,2013 年 11 月; [4] Unix v6++源码