

同濟大學

TONGJI UNIVERSITY

《操作系统课程设计》

实验报告

实验名称

类 UNIX 二级文件系统

姓名学号

王钧涛 2050254

学院（系）

电子与信息工程学院

专 业

计算机科学与技术

任课教师

方钰

日 期

2023 年 5 月 1 日

装  
订  
线

## 目 录

实验报告 .....	I
1 设计要求 .....	1
1.1 课设设计要求 .....	1
2 系统结构 .....	3
2.1 文件系统结构说明 .....	3
2.2 目录结构说明 .....	6
2.3 文件打开结构说明 .....	9
2.4 文件系统的实现 .....	15
2.5 高速缓存结构说明 .....	18
3 系统操作 .....	23
3.1 命令行使用说明 .....	23
3.2 命令行操作结果展示 .....	23
3.3 命令行测试结果展示 .....	26
4 总结 .....	28
4.1 心得体会 .....	28

## 1 设计要求

### 1.1 课设设计要求

#### 1. 实现对该逻辑磁盘的基本读写操作

只有一个设备一个进程，缓存队列可以合理简化，但必须做到：

1. 按 LRU 的方式管理缓存；
2. 缓存可以重用尽量重用；
3. 不能重用时，通过一级文件系统的接口调用，完成 blkno 对应的 512 字节的读写；
4. 实现延迟写。

#### 2. 在该逻辑磁盘上定义二级文件系统结构

- SuperBlock 及 Inode 区所在位置及大小
- Inode 节点
- 数据结构定义：注意大小，一个盘块包含整数个 Inode 节点
- Inode 区的组织（给定一个 Inode 节点号，怎样快速定位）
- 索引结构：多级索引结构的构成，索引结构的生成与检索过程\*\*\*

#### 3. 文件系统的目录结构

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

#### 4. 文件打开结构

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

#### 5. 文件操作接口

- fformat: 格式化文件卷
- ls: 列目录
- 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

装

订

线

## 2 系统结构

### 2.1 文件系统结构说明

#### (1) SuperBlock 及 Inode 区所在位置及大小

SuperBlock 占用 1024 字节 (#0 - #2)，之后 Inode 区 #2 - #1024，总磁盘大小 16MB (32,768 个块 #0 - #32767)

#### (2) Inode 节点数据结构的定义及索引结构的说明

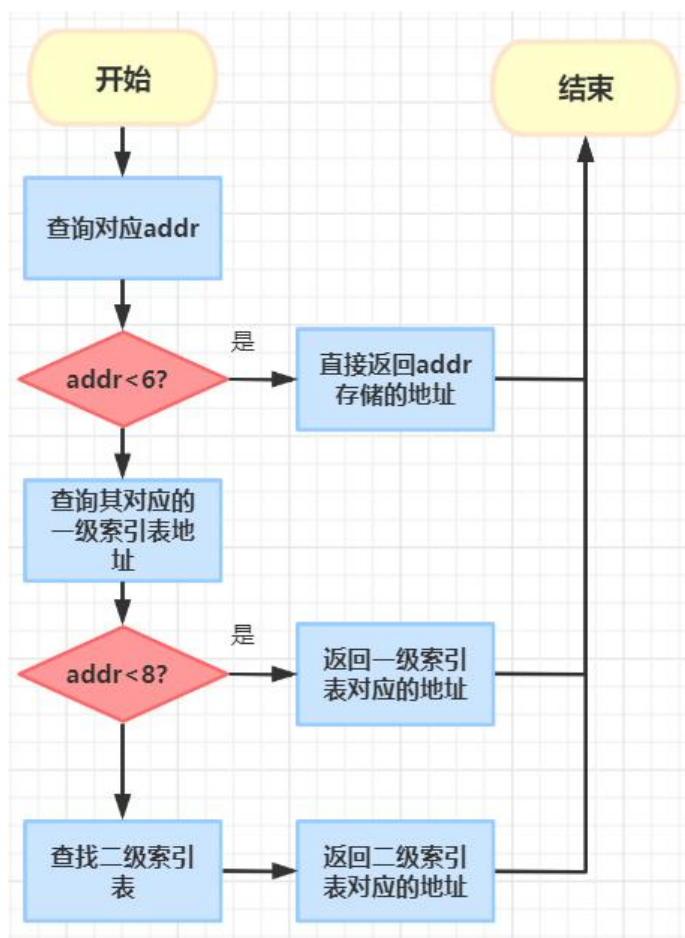
在磁盘上，Inode 节点的定义如下，一个 DiskInode 占据 64 个字节，每个 Blk 块可以存放 8 个磁盘 Inode:

```
1. class DiskInode { // 64 字节
2. public:
3.     int mode, dirLinkNum;
4.     short userMode, __none__;
5.     int fileSize, addr[10];
6.     int lastViewTime, lastEditTime;
7. public:
8.     DiskInode();
9.     ~DiskInode();
10.};
```

由于本次课程设计只有使用者一个单用户，因此本次设计中没有组用户和用户权限区分，不设置用户 id 和组 id 字段。作为替代，权限管理的内容被移动到 DiskInode 中进行实现，通过 userMode 字段保存创建时定义的用户设置权限。

索引结构上，与 V6++ 类似，同样是使用了 addr[10] 数组存放盘块号，将文件分为普通文件、大型文件和巨型文件三类，当文件大小小于 6 个块时属于普通文件，直接在 addr 数组中存取块地址；当文件大小大于 6 个块时而小于  $6+128*2$  时，属于大型文件，在 addr[6]-addr[7] 中存放一级索引表的块地址，一级索引表的每个块中都可以存储 128 个块；当文件大小大于 262 个块时，属于巨型文件，在 addr[0]~addr[7] 不变的前提下，继续在 addr[8]~addr[9] 采用二级索引表的形式，也即 addr[8] 对应的是一级索引表，一级索引表中存储的 128 个地址不是实际盘块，而是存储实际盘块的二级索引表的盘块地址。

索引方法如下：



### (3) SuperBlock 数据结构的定义及对 Inode 节点及文件数据区管理的相关算法

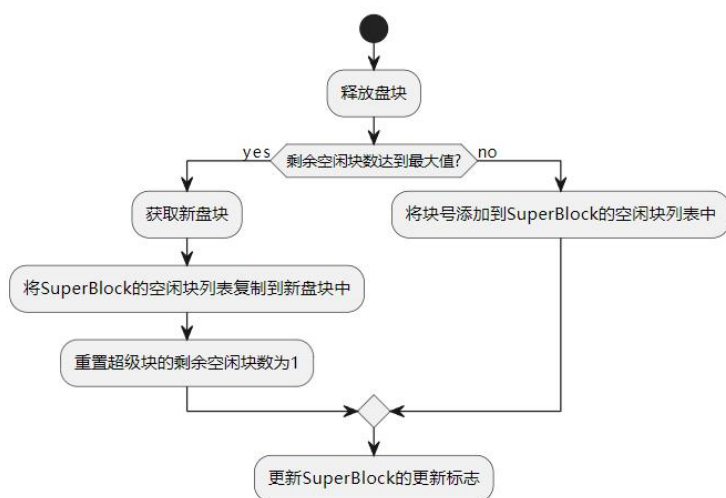
SuperBlock 数据结构的定义如下：

```

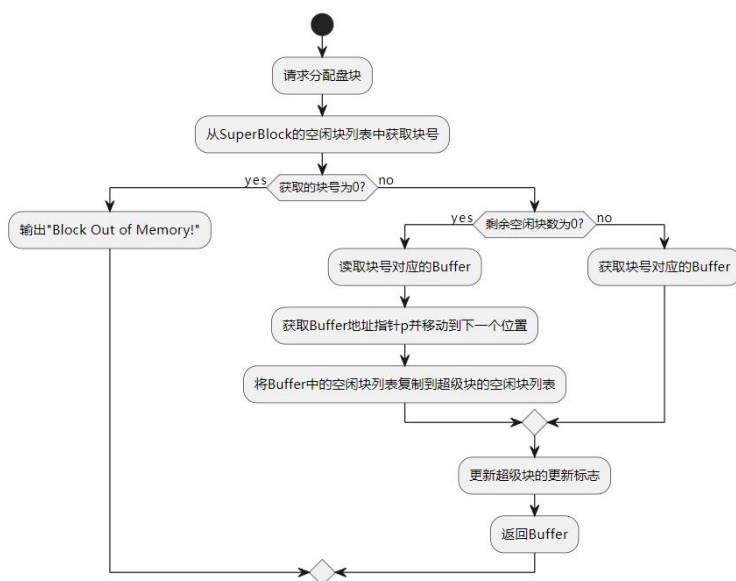
1. class SuperBlock {
2. public:
3.     const static int MAX_NBLOCK = 100;
4.     const static int MAX_NINODE = 100;
5.
6. public:
7.     int freeBlockNum;
8.     int freeBlockList[MAX_NBLOCK];
9.     int freeDiskINodeNum;
10.    int freeDiskINodeList[MAX_NINODE];
11.    int superBlockUpdatedFlag, lastUpdatedTime;
12.    int padding[256 - MAX_NBLOCK - MAX_NINODE - 4];
13.
14. public:
15.    void setUpdated(bool flag = true);
16.    bool getUpdated();
17. };
  
```

盘块管理和 Inode 管理与 V6++ 相同，盘块的分配和回收均采用成组链接法，而对 Inode 的管理只找 100 块 freeInode 放在 SuperBlock，分配时如果发现没有可分配的，就再从盘中找 100 块，如果释放时发现 100 块的空闲队列已满就直接丢弃该释放的 Inode。

其中比较重要的是成组链接法的存取流程，具体如下，  
释放：



分配：



FileSystem 和 superBlock 提供的是文件系统的最基础服务，因此不涉及函数调用关系。

## 2.2 目录结构说明

### (1) 目录文件的结构

目录文件的定义是通过继承 `Inode` 来实现目录文件的函数拓展的。

具体 `DirInode` 定义如下：

```
1. class DirInode: public Inode {
2. public:
3.     DirEntry FindNode(string name, int mode = InodeFlag::IDIR);
4.     bool AddNode(int iNodeIndex, int mode, string name);
5.     bool DeleteNode(string name, int mode = InodeFlag::IDIR);
6.     bool ChangeNode(string name, int newMode, string newName);
7.     bool DeleteAllNode();
8.     void InitDirInode(int parent = 0);
9.     vector<DirEntry> DirList();
10. };
```

其中每个目录项占据 32 个字节，包括：24 字节名字，4 字节的 `iNode` 号，4 字节的 `mode flag`（是否目录项，读写模式信息），具体定义如下：

```
1. struct DirEntry{
2.     static const int DIR_ENTRY_SIZE = 32;
3.     char name[24] = {0};
4.     int iNodeIndex = 0, mode = 0;
5.     DirEntry();
6.     DirEntry(int iNodeIndex, int mode, string name);
7. };
```

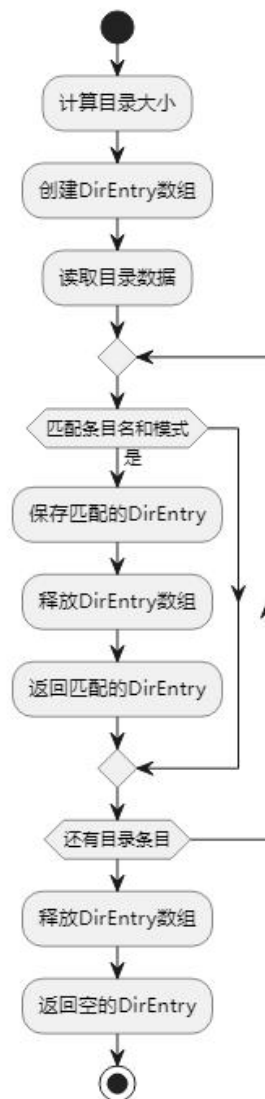
### (2) 目录检索算法的设计

目录检索通过 `NameI()` 函数实现，设计了四种匹配模式（`cd`，`open`，`create`，`delete`），通过将目录字符串以 '/' 划分，依次遍历各段进行检索匹配。

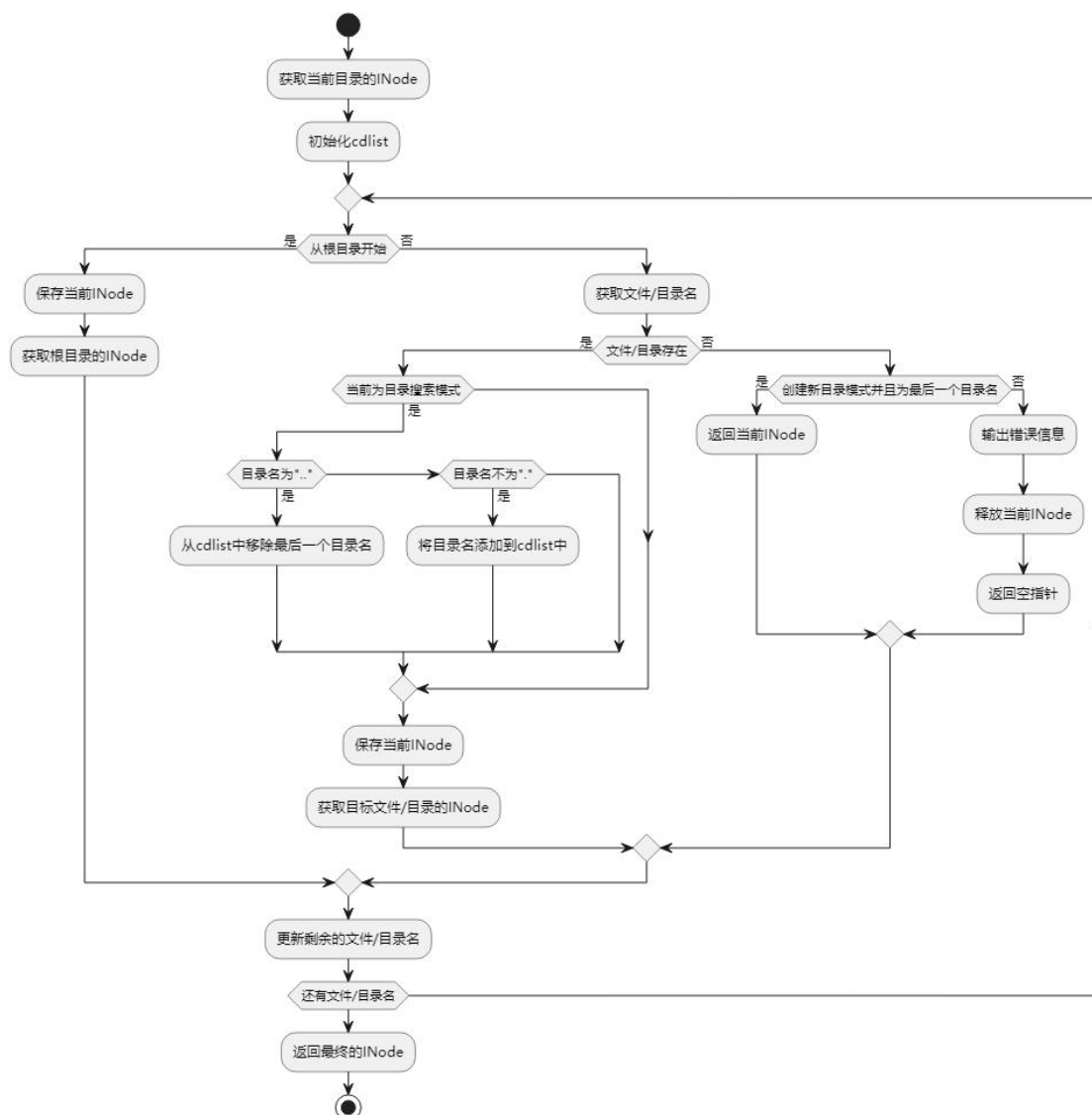
以 '/' 开头为绝对地址，从根 `Inode` 开始读取，否则为相对地址，从当前文件系统 `Inode` 开始读取，调用 `DirInode` 的 `FindNode` 函数顺序遍历目录项。

`FindNode` 的流程如下：





NameI 的流程如下：



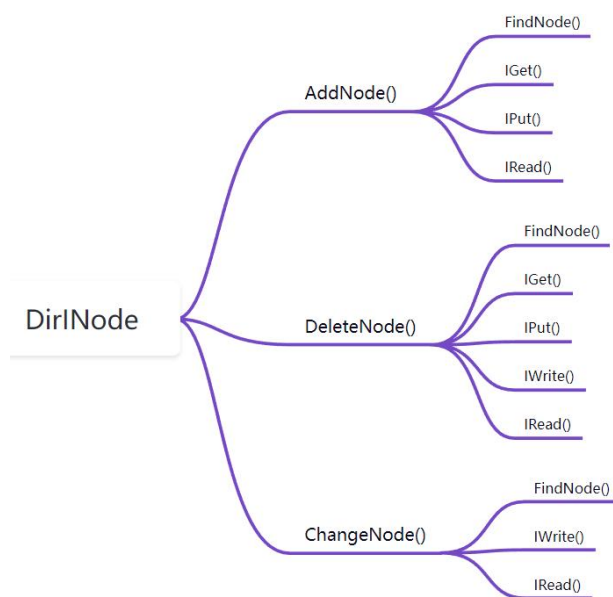
### (3) 目录结构增、删、改的设计

增：直接在文件末尾添加 32 字节的目录项（调用 Inode 的 IWrite 函数实现，取消 V6++ 系统调用设计，直接传操作要求）

删：调用 IWrite() 读取全文件，然后查找其中目录项并删除，将剩下 n-1 个目录项写入文件（速度慢，另一种方法是清空但不删除，会造成文件臃肿，相对而言对目录项的改变不频繁，速度慢更可以接受）

改：与增删搜索节点的方法相同，都是遍历，然后进行文件名或工作模式的更改（不可更改其所对应的 Inode 节点）。

函数调用图如下：



### 2.3 文件打开结构说明

#### (1) 文件打开结构的设计

文件打开结构由 File 类进行存储，由 OpenFileManager 进行管理。与 Unix V6++ 的区别是，由于本次课程设计的要求是单用户单进程，从而用户打开文件表等价于进程打开文件表，同时也等价于系统打开文件表。因此，本次课设不再重复设计用户进程打开文件表，只用系统 File 表进行管理。

文件打开结构定义如下，主要变量为打开模式 flag 和文件指针偏移量 offset:

```

1. class File {
2. public:
3.     enum FileFlags {
4.         FEMPTY = 0x0,
5.         FREAD = 0x1,
6.         FWRITE = 0x2,
7.     };
8.     int flag = 0, offset = 0;
9.     INode* inode = nullptr;
10. public:
11.     bool IsEmpty();
12.     bool HasFlag(FileFlags f);
13.     void SetFlag(int f);
14.     void UnsetFlag(int f);
15.     void Clean();
16.     void SetINode(INode* i);
17. };
    
```

值得注意的是，文件状态 flag 只设计了 FREAD 和 FWRITE，这是因为单进程下对文件的操作不需要考虑到进程安全问题，所有文件只要打开就永远准备好了为当前进程服务，因此只需要考虑打开时的权限访问问题即可。

作为管理的系统文件打开表定义如下：

```
1. class OpenFileManager {
2. public:
3.     static const int MAX_FILES = 128;
4.     File openFileTable[MAX_FILES];
5. public:
6.     OpenFileManager();
7.     ~OpenFileManager();
8.     int Alloc(INode *i);
9.     void Free(int fd);
10.    File* GetFile(int fd);
11.    void init();
12. private:
13.    File* AllocFile(INode *i);
14.    void FreeFile(File* f);
15. };
16. extern OpenFileManager openFileManager;
```

系统文件打开表定义了 128 个 File 文件，意味着二级文件系统最大可打开的文件数量不超过 128 个，由于目录检索和查询不涉及文件的打开，从而也不会占用系统文件打开表，因此这是完全够用的。

## (2) 内存 Inode 节点数据结构的定义及分配与回收

内存 Inode 负责维护对整个文件的操作，具体定义如下：

```
1. enum INodeFlag {
2.     IFREE = 0x0,
3.     IFILE = 0x1,
4.     IUPD = 0x2,    // 修改过，需要更新相应外存 INode
5.     IACC = 0x4,    // 访问过，需要修改最近一次访问时间
6.     IDIR = 0x8,
7. };
8.
9. class INode{
10. public:
11.     int mode, flag;
12.     int usingCount, dirLinkNum;
```

```

13.     int      lastViewTime, lastEditTime, iNodeNo;
14.     short    userMode, __none__;
15.     int      fileSize, addr[10];
16. public:
17.     INode();
18.     ~INode();
19.     int IRead(IOPParameter para);
20.     int IWrite(IOPParameter para);
21.     int BMap(int lbn);
22.     void IUpdate(int time);
23.     void ITrunc();//释放 Inode
24.     void Clean();
25.     void ICopy(Buffer* bp, int iNumber);
26. private:
27.     DiskINode WriteDiskINode();
28.     void readDiskINode(DiskINode diskINode);
29. public:
30.     enum UserModeFlag{READ=0x1, WRITE=0x2};
31.     void SetFileUserMode(int flag);
32.     bool HasFileUserMode(int flag);
33. };

```

BMap 在前述已经介绍过流程, INode 除了包含 DiskINode 所应当包含的信息以外, 还额外有 flag 参数, 用以存储当前 INode 是否被访问 / 更新过, 如果有则在 IUpdate 将内存 INode 写入磁盘的过程中更新 lastViewTime 和 lastEditTime。

IRead()和 IWrite()则相对比较简单, 通过将读取的逻辑块调用 BMap 函数对应到一级文件系统的物理块上, 然后将数据从中分割读取再汇总, 或者将数据分割然后进行写入。

此外, 内存 INode 由 INodeManager 统一进行管理, 进行回收和分配, 不同于之前提到的文件系统对 DiskINode, INodeManager 是进行已存在的 DiskINode 何时调入、调出内存的分配, 其定义如下:

```

1. class INodeManager{
2.     public:
3.         static const int INODE_NUM = 128;
4.
5.     private:
6.         std::map<int, INode*> iNodeMap;
7.         INode iNodeTable[INODE_NUM];
8.     public:
9.         INodeManager();
10.        ~INodeManager();

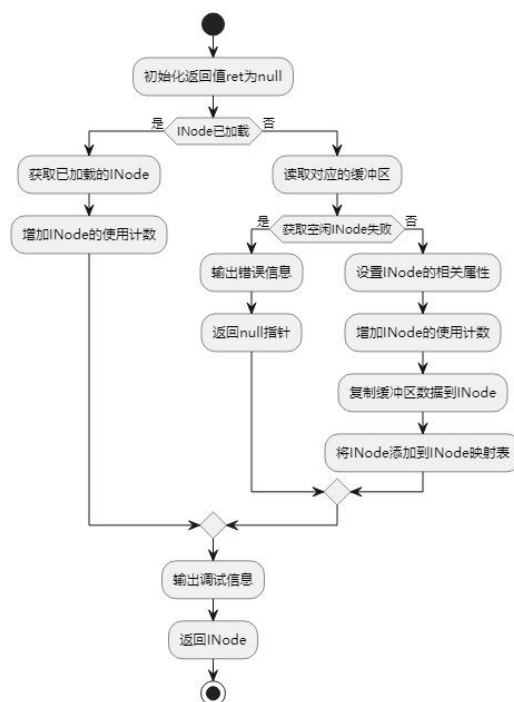
```

```

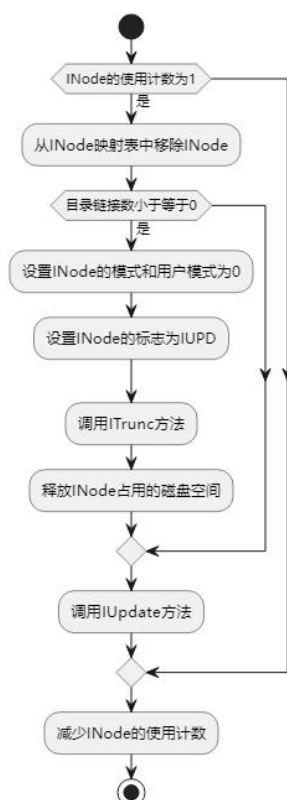
11.     INode* IGet(int no);
12.     void IPut(INode* node);
13.     void UpdateINodeTable();
14.     bool IsLoaded(int no);
15.     INode* GetFreeINode();
16.     void Format();
17. };
18. extern INodeManager iNodeManager;
    
```

总计在内存中管理 128 个 INode 节点，超过该数量则会导致系统无法分配更多的 INode，进而产生报错。此外，为了加快对 INode 的查找和复用，将 INode 信息额外保存在一张哈希表中，分配时遍历查询第一个空余项并加入哈希表，回收时直接将其从哈希表中删除，这样就可以使得每次提取变成  $O(1)$  操作。

INode 的分配由 IGet(no) 函数实现，其流程如下：



INode 的回收由 IPut(inode) 函数实现，其流程如下：



### (3) 文件打开过程

文件的打开由文件系统提出申请，在 FileManager 中被定义：

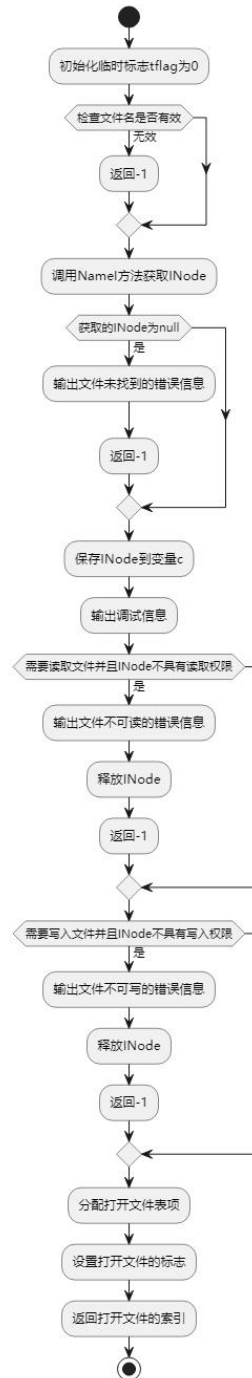
```

1. class FileManager
2. {
3. private:
4.     vector<string> curDirStringList;
5.
6. public:
7.     INode* curDirINode;
8. public:
9.     FileManager();
10.    ~FileManager();
11.    void init();
12. public:
13.    void Ls();
14.    string GetCurDir();
15.    bool Cd(string dirName);
16.    bool Mkdir(string dirName);
17.    int Create(string fileName, string mode);
18.    int Delete(string fileName);
19.    int Open(string fileName, string mode);
20.    int Close(int fd);
  
```

```

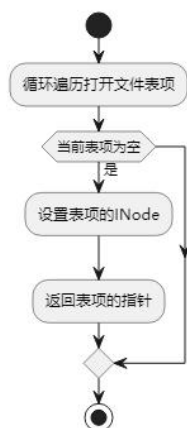
21.     int Seek(int fd, int offset, int origin = SEEK_SET);
22.     int Write(int fd, unsigned char* ptr, int size);
23.     int Read(int fd, unsigned char* ptr, int size);
24.     int Size(int fd);
25. }
    
```

其具体流程如下：





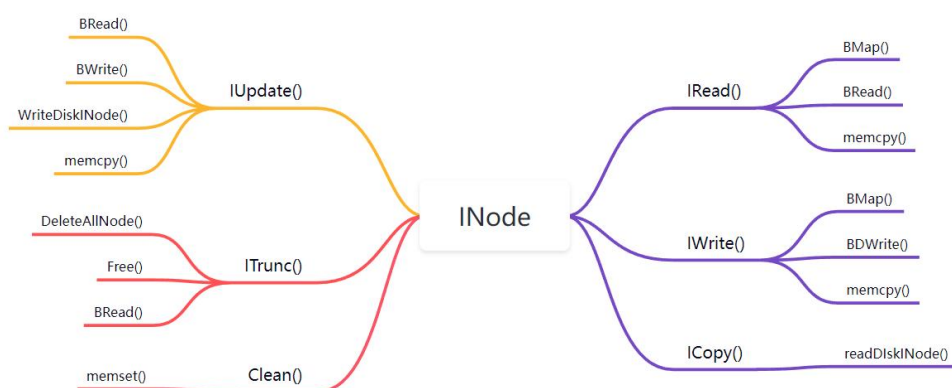
成功找到所要打开的文件的 Inode 号并完成权限判断后，文件系统会先将该文件调入内存(调用前面讲解过的 IGet 函数)，然后调用 OpenFileManager 中的 Alloc() 函数，该函数的流程如下：



函数会分配过程贪心的选择最早找到的未被使用的表项并将初始化，然后将 Inode 的指针保存在 File 文件中，也即完成了文件的打开操作。该函数会返回文件操作符 fd 作为对文件的访问句柄。

文件打开流程与 V6++ 的主要区别是会额外向 Inode 询问文件是否可以以当前方式打开，其他部分是大体相似的。

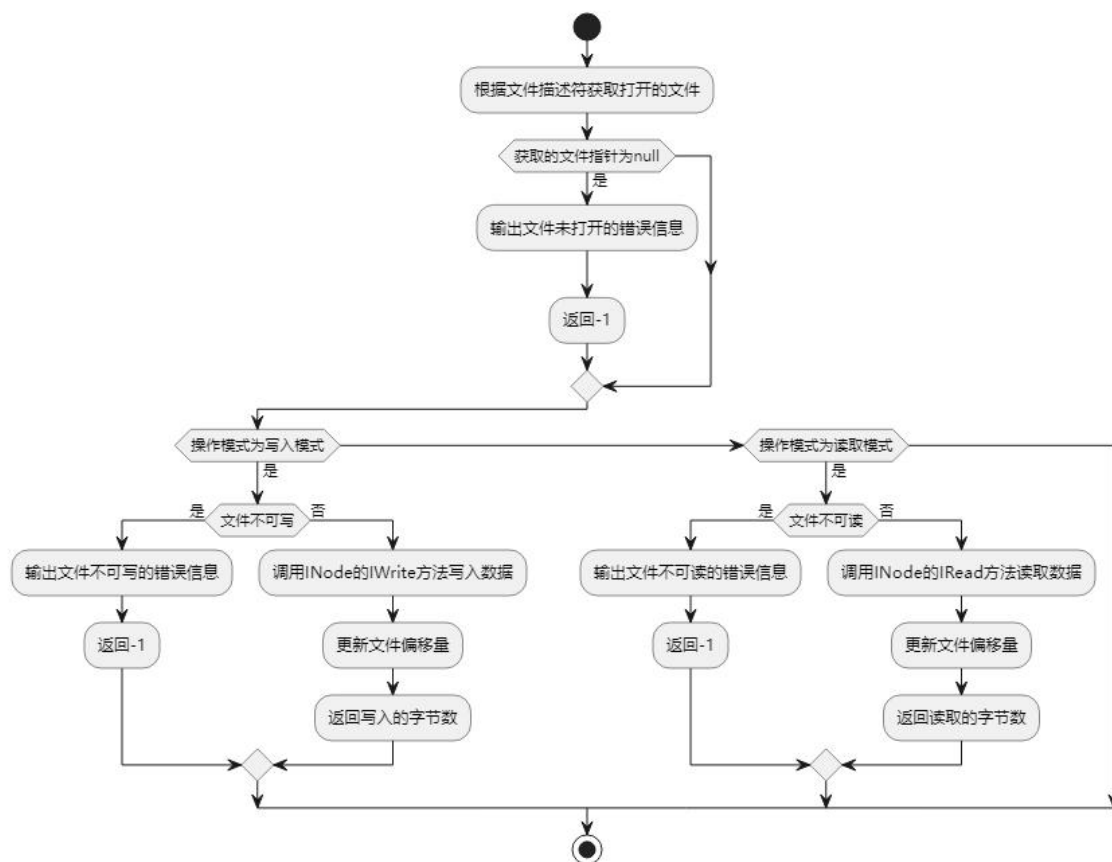
函数调用关系如下：



## 2.4 文件系统的实现

### (1) 文件读写操作的实现流程

2.3 节已经介绍过 FileManager 的定义，这里主要介绍各函数的实现流程，其中读写功能合并 Rdwr 函数中进行设计，该函数流程如下：



在此基础上，分别封装 Read 函数和 Write 函数，通过在函数的参数中指定读写方式 Mode 进行区分。通过文件描述符 fd 查找，统一调用 RdWr 函数，在其中检查权限问题，通过后调用 IRead/IWrite 写入并修改文件描述符中的 foffset

```

1. int FileManager::Write(int fd, unsigned char *ptr, int size)
2. {
3.     return Rdwr(File::FileFlags::FWRITE, fd, ptr, size);
4. }
5.
6. int FileManager::Read(int fd, unsigned char *ptr, int size)
7. {
8.     return Rdwr(File::FileFlags::FREAD, fd, ptr, size);
9. }
  
```

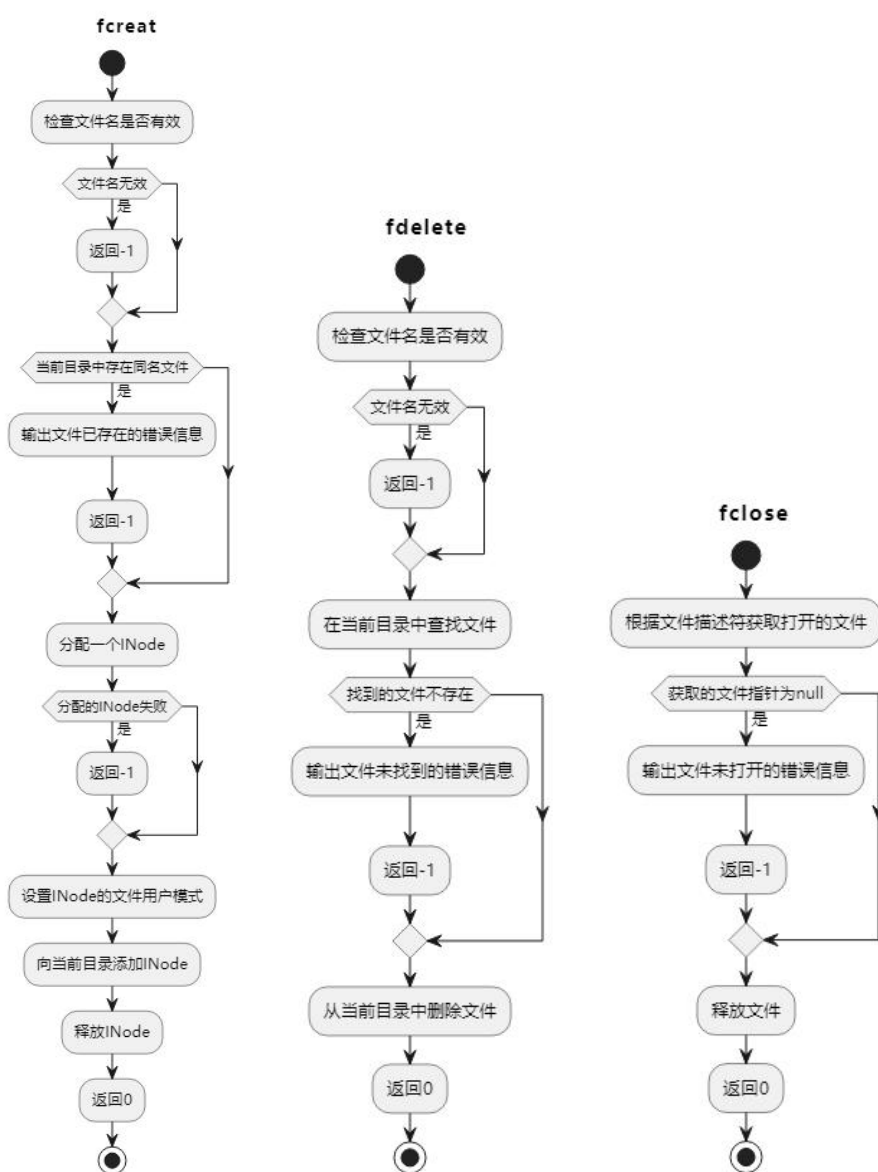
所有的功能函数与 V6++的区别在于不设计系统调用，而改成直接传参，这是因为 V6++中考虑到进程之间的切换，其对文件系统的访问不是通过直接调用的方式进行的，而是在系统函数的基础上封装出系统调用的接口，通过系统调用访问，为此在参数传递上进行了一些额外的设计，这些设计在本次课程设计中都是

无用的。

## (2) 文件其他操作的实现流程

- ✧ **Fcreate/Fdelete**: 这两个函数会检查命名规范, 然后调用#3.3 节的目录增删函数进行目录设置, **fcreate** 额外调用 **IAlloc()** 分配内存节点并设置文件权限 (**fdelete** 不额外调用函数, 这是因为在取消目录勾连后 **Inode** 的勾连数为 0, 在节点回收时会自动删除其中的文件)

流程图如下:



- ✧ Close: 该函数会尝试向 Filemanager 删除文件标识符 fd, 流程图如上:
- ✧ Seek: 该函数的实现方法是更改 File 文件的 f\_offset, 流程图如下:



## 2.5 高速缓存结构说明

### 1. 缓存控制块的设计

由于课程设计要求中只有一个进程,在当前进程发起 IO 请求时一定不会有其他进程执行 IO 请求,而当该进程 IO 请求被执行的时候进程处于阻塞状态,又不会再次发起 IO 请求。上面的论述表明在同一时刻最多只有一个缓存控制块处于忙状态,并且在该缓存块忙的时候不会有任何请求需要判断其是否处于 BUSY 状态。因此,删去缓存控制块 FLAG 中对于块 BUSY 的设计。在此基础上,对块 WANTED 的设计也毫无意义,这是因为当进程发起新一轮请求时,所有的缓存块都可以被当前进程访问,不会发生等待某一块 IO 操作完成的情况。因此,删去缓存控制块 FLAG 中对于块 WANTED 的设计。

与以上两个简化类似，在只有一个进程的情况下，对单一进程的异步写入是没有意义的，因为永远不需要等待设备队列空闲下来。此处将 B\_ANYSC 标志删去，同时由于无法实现标准的异步写（写时进程永远处于堵塞状态而不会有进程能够利用 CPU），因此当块全部重新写入时不再直接进行写入，而是永远推迟到块被缓存队列回收后才进行写入。

缓存队列的数据结构定义如下：

```
1. class Buffer {
2. public:
3.     enum BFlag {
4.         B_DONE = 0x1,
5.         B_DELWRI = 0x2
6.     };
7.     int flag, blkno, wcount;
8.     unsigned char* addr;
9.     Buffer* prev, *next;
10. public:
11.     Buffer();
12.     void Pick();
13.     void Insert(Buffer* p, Buffer* n);
14.     bool HasFlag(BFlag f);
15.     void SetFlag(BFlag f);
16.     void UnsetFlag(BFlag f);
17.     void CleanFlag();
18.     friend class BufferManager;
19. };
```

其中 flag 有两个状态，分别是 B\_DONE（用来表明块是否能重用）和 B\_DELWRI（用来区分块是需要延迟写回），wcount 用来指示块操作的字节数大小，addr 负责存放缓存块的地址。

缓存控制块的主要函数是 Pick 和 Insert：

Pick：

```
1. void Buffer::Pick()
2. {
3.     if(prev != nullptr)
4.         prev->next = next;
5.     if(next != nullptr)
6.         next->prev = prev;
7.     prev = nullptr;
8.     next = nullptr;
9. }
```

Insert:

```

1. void Buffer::Insert(Buffer *p, Buffer *n)
2. {
3.     if(p != nullptr)
4.     {
5.         prev = p;
6.         p->next = this;
7.     }
8.     if(n != nullptr)
9.     {
10.        next = n;
11.        n->prev = this;
12.    }
13. }

```

这两个函数实现了简单的链表维护功能，可以实现将一个缓存控制块从链表中摘出和插入到两个链表之间。

## 2. 缓存队列的设计及分配和回收算法

由于本次课程设计只由一个磁盘设备，因此所有的块在理论上都能被该主设备访问，从而不再设计设备管理队列，同时也将特殊的代表不从属于任何一个设备的 NODEV 队列删去，这意味着所有块在文件系统初始化完成时就默认绑定在该设备上。进一步地，由于一级文件系统在进程读取时会强制阻塞，同一时刻最多只有一个缓存控制块处于忙状态，并且在该缓存块忙的时候不会有任何新增请求，这同时意味着 IO 请求队列最多仅会有一个块被放入，因此该队列的设计不再具有意义，本次课程设计将与 Brelse 其删去。值得注意的时，这同时也意味着空闲缓存队列永远都会有可用缓存，不需要在去判断没有找到可供重用的块而使得进程睡眠在 free\_list 的 WANTED 标志上的情况。

本次课程设计提出的缓存队列设计如下：

```

1. class BufferManager{
2.
3. public:
4.     unsigned char addr[BUF_NUM][BUF_SIZE];
5.     Buffer buffer[BUF_NUM];
6.     Buffer *lru;
7.     Buffer lruitem;

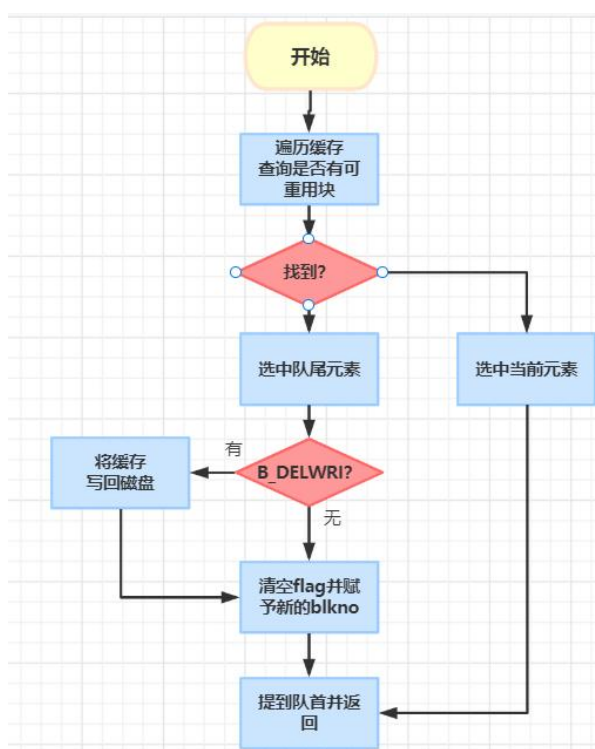
```

```

8. public:
9.     BufferManager();
10.    Buffer *GetBlk(int blkno);
11.    Buffer *BRead(int blkno);
12.    void BWrite(Buffer* bp);
13.    void BDWrite(Buffer* bp);
14.    void BClear(Buffer* bp);
15.    void BFlush();
16. };
    
```

缓存管理器将空闲缓存队列视作一个按时间排序的类 LRU 队列，越靠近队首代表访问的时间越靠近当前时间。允许该队列从中间（找到可重用块时）和末尾（未找到可重用块时）取下元素，同时仅允许将元素放入队首。

GetBlk 的流程如下：



### 3. 借助缓存实现对一级文件的读写操作的流程

BRead 和 BWrite 会调用 DeviceDriver 封装好的设备 Read 和 Write 函数，调用 fseek 定位到块位置(blkno\*512),然后写 fread 读 fwrite，其定义如下，该过程只有两行，较为简单。

```

1. class DeviceDriver{
2. public:
    
```

```

3.     FILE *fp;
4. public:
5.     DeviceDriver();
6.     ~DeviceDriver();
7.     void Read(unsigned char* addr, int size, int offset = 0);
8.     void Write(unsigned char* addr, int size, int offset = 0);
9.     void Open(bool format = false);
10.    void Format();
11. };
    
```

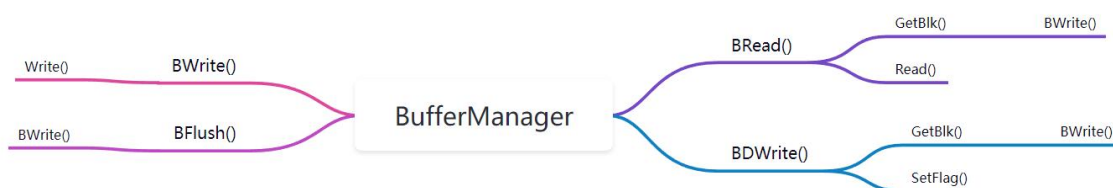
```

1. void DeviceDriver::Read(unsigned char *addr, int size, int offset)
2. {
3.     fseek(fp, offset, SEEK_SET);
4.     fread(addr, size, 1, fp);
5. }
6.
7. void DeviceDriver::Write(unsigned char* addr, int size, int offset){
8.
9.     fseek(fp, offset, SEEK_SET);
9.     fwrite(addr, size, 1, fp);
10. }
    
```

需要注意的是，如果 BRead 在调用 GetBlk 后获得的是 B\_DONE 的块，则不会进行读取而是直接返回，否则才进行读取并在 flag 上加上 B\_DONE。

另外，Format 函数负责在文件系统初始化时模拟将整个磁盘初始化（清 0）的过程。

函数调用关系如下：





## 3 系统操作

### 3.1 命令行使用说明

直接打开 exe 程序即可进入二级文件系统。

支持以下命令：

- 1) ls : 显示当前文件夹下的全部文件和文件夹
- 2) mkdir <dirname> : 在当前文件夹下创建目录<dirname>
- 3) rmdir <dirname> : 在当前文件夹下删除目录<dirname>
- 4) ffmormat : 格式化文件系统
- 5) exit : 退出系统 (必须使用该命令退出系统, 否则会导致缓存未写入磁盘而产生错误!)
- 6) fcreat <filename> <mode> : 创建名为 <filename> 的文件, 其读写权限为<mode>(mode 可以为 r, w 或 rw)
- 7) foepn <filename> <mode> : 打开名为 <filename> 的文件, 打开方式为<mode> (mode 可以为 r, w 或 rw), 其会返回该文件的文件操作符 fd
- 8) fread <fd> <size> : 从文件描述符<fd>指向的文件中读取最大<size>个字节到缓存区, 如果<size>为 0 则默认将文件读取完毕, 输出实际读取的字节数
- 9) fwrite <fd> <size> : 从缓存区写入最大<size>个字节到文件描述符<fd>指向的文件中, 如果<size>为 0 则默认将缓存区全部写入, 输出实际写入的字节数
- 10) fseek <fd> <offset> <mode>: 将文件描述符<fd>指向的文件偏移到<mode>和<offset>指向的位置, 其中 mode 可以取 0、1、2, 其含义分别为从文件头、当前位置、文件尾开始计算偏移量。

此外, 为了与一级文件系统交互的方便, 我还新增了三个命令:

- 11) fin <filename>: 将<filename>指向的一级操作系统的实际文件读取到缓存区中
- 12) fout <filename>: 将缓存区中的所有字节写入<filename>指向的一级操作系统的实际文件
- 13) bshow : 查看缓存区信息

### 代码编译方式:

安装 mingw-g++-5.7, 直接在文件夹下执行 make 命令即可。

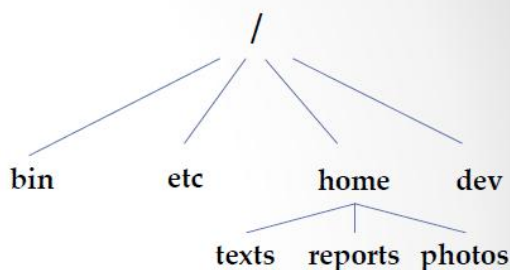
### 3.2 命令行操作结果展示

要求如下:

- ① 格式化文件卷:

② 用 `mkdir` 命令创建子目录，建立如图所示目录结构；

③ 把你的课设报告，关于课程设计报告的 `ReadMe.txt` 和一张图片存进这个文件系统，分别放在 `/home/texts`，`/home/reports` 和 `/home/photos` 文件夹；



转化为我的程序代码为：

```

1. fformat
2. mkdir bin
3. mkdir etc
4. mkdir home
5. mkdir dev
6. ls
7. cd home
8. fin 2050254-王钧涛.pdf
9. fcreat reports rw
10. fopen reports rw
11. fwrite 0 0
12. fclose 0
13. fcreat photos rw
14. fopen photos rw
15. fin 1.jpg
16. fwrite 0 0
17. fclose 0
18. fcreat texts rw
19. fopen texts rw
20. fin Readme.txt
21. fwrite 0 0
22. fclose 0
23. ls
  
```

运行，查看结果：

建立文件夹：

```
/$ffformat
/$mkdir bin
/$mkdir etc
/$mkdir home
/$mkdir dev
/$ls
bin etc home dev
```

写入报告：

```
/$cd home
/home$fin 2050254-王钧涛.pdf
Read 4208458 bytes to buffer.
/home$fcreat reports rw
/home$fopen reports rw
Alloed file descriptor: 0
/home$fwrite 0 0
Write 4208458 bytes to file.
/home$fclose 0
```

写入照片：

```
/home$fcreat photos rw
/home$fopen photos rw
Alloed file descriptor: 0
/home$fin 1.jpg
Read 42790 bytes to buffer.
/home$fwrite 0 0
Write 42790 bytes to file.
/home$fclose 0
```

写入 README：

```
/home$fcreat texts rw
/home$fopen texts rw
Alloed file descriptor: 0
/home$fin Readme.txt
Read 9630 bytes to buffer.
/home$fwrite 0 0
Write 9630 bytes to file.
/home$fclose 0
/home$ls
reports photos texts
/home$
```

创建过程符合要求，程序正确无误。

再将照片读出测试（便于直观显示是否有错误，因为能在照片上清晰体现）：

1. fopen photos r
2. fread 0 0
3. fout photo.jpg

将输出的文件与输入文件进行对比：



图片正常显示，准确无误。同时也使用 `fc` 命令验证报告读出的准确性：

```
1. fopen reports r
2. fread 0 0
3. fclose 0
4. fout 2050254-王钧涛-1.pdf
5. exit
```

```
D:\Repos\PigFileSystem>fc /b 2050254-王钧涛.pdf 2050254-王钧涛-1.pdf
正在比较文件 2050254-王钧涛.pdf 和 2050254-王钧涛-1.PDF
FC: 找不到差异
```

证明了本次课设设计的二级文件系统的准确性和可靠性。

### 3.3 命令行测试结果展示

要求如下：

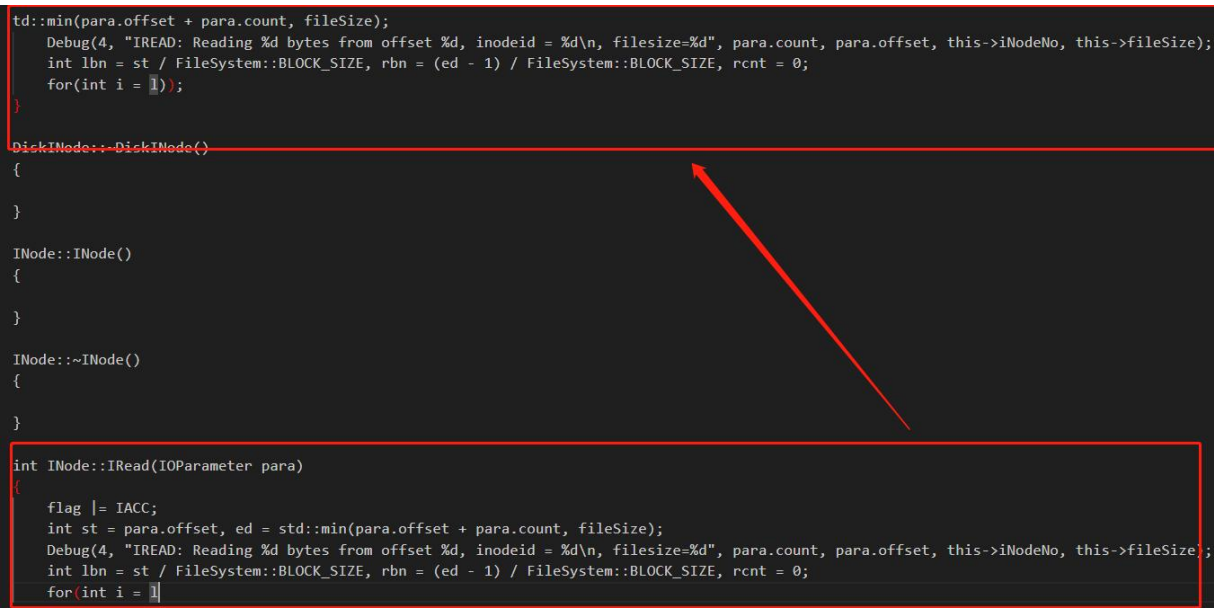
- ① 新建文件 `/test/Jerry`，打开该文件，任意写入 800 个字节；
- ② 将文件读写指针定位到第 500 字节，读出 500 个字节到字符串 `abc`。
- ③ 将 `abc` 写回文件。

转化为我的程序代码为：

```
1. fformat
2. mkdir test
3. cd test
4. fcreat Jerry rw
5. fopen Jerry rw
6. fin 1.txt
7. fwrite 0 800
8. fseek 0 500 0
9. fread 0 500
```

```
10. flseek 0 0 0
11. fwrite 0 0
12. flseek 0 0 0
13. fread 0 0
14. fout 2.txt
```

其中，1.txt 放置的是 INode.cpp 的前 800 字节，观察结果如下：



```
td::min(para.offset + para.count, fileSize);
Debug(4, "IREAD: Reading %d bytes from offset %d, inodeid = %d\n, fileSize=%d", para.count, para.offset, this->iNodeNo, this->fileSize);
int lbn = st / FileSystem::BLOCK_SIZE, rbn = (ed - 1) / FileSystem::BLOCK_SIZE, rcnt = 0;
for(int i = 1);
}

DiskInode::~DiskInode()
{
}

INode::INode()
{
}

INode::~INode()
{
}

int INode::IRead(IOParameter para)
{
    flag |= IACC;
    int st = para.offset, ed = std::min(para.offset + para.count, fileSize);
    Debug(4, "IREAD: Reading %d bytes from offset %d, inodeid = %d\n, fileSize=%d", para.count, para.offset, this->iNodeNo, this->fileSize);
    int lbn = st / FileSystem::BLOCK_SIZE, rbn = (ed - 1) / FileSystem::BLOCK_SIZE, rcnt = 0;
    for(int i = 1)
```

可以观察到后 300 字节被移动到了前 300 字节的位置并覆盖了原有代码，与期望相符。

### 4 总结

#### 4.1 心得体会

写完课程设计后，我问了自己一个问题：这次课设难吗，难在哪里？以我的观点来看，本次课设的难度是不大的，尽管他已经是我本科实现的最大代码量的工程之一。但是从缓存到 INode，再到目录和文件系统，每一个层次包含哪些功能、需要执行哪些操作都已经在上学期的原理课程中讲透彻了，每一层只要按部就班地实现即可，也没有很难需要思考的算法和数据结构——用到的最多不过是 LRU 链表而已，甚至连优先队列都不需要设计。然而，就是这样一个看似简单的任务，需要用巨大的精力去编写，去创造。我在写到 FileManager 的时候，复盘了下之前写过的代码，回头望望才感觉之前洋洋洒洒写的诸多代码根本没有理清架构的思路，杂乱无章，为此我又推翻了之前的代码，重构了整个 Buffer 和 INode 的设计框架，完善接口思路。

当项目越来越大的时候，不能盲目为了进度随意地实现底层框架——现在省下了多少时间，以后就要成倍地还回来，这是我认识的第一点。

本次课设的时间跨度也很长，重构之后隔天就受到五一新冠爆发的影响，高烧多日没有接触项目，直接导致了某些模块的一些小问题因为省略测试的缘故没有发现，而成为了埋在以后的一颗颗地雷，在后面整体测试的时候常常为了一个很小的错误调试大半天。为此，我还特地为大型测试开发了一套树形 Debug API 来实现层与层之间的故障定位，放在了 Debug.cpp 中。实现这个 API 的两小时，帮助我成功节约了一天多的 debug 时间。有时候多写代码是为了少写代码，这是我认识的第二点。

虽然课程题目叫“操作系统课程设计”，但在我看来其实称不上是操作系统。在我看来操作系统的核心是进程，是进程安全，而这是本次课设没有涉及的内容了。我不认为写完“操作系统课程设计”，我就真的完成了“操作系统课程设计”，对操作系统的认识我想还有很长的路要走。

长路漫漫其修远兮，吾将上下而求索。