TONGJI UNIVERSITY

同濟大學

《操作系统》课程设计

二级文件系统

学 号：1552258

姓 名：黄学森

专 业：计算机科学与技术

授课老师：邓蓉

2018 年 6 月

目录

目录

一、课程设计基础任务描述

为 LINUX 设计一个简单的二级文件系统。本实验用某个大文件，如 c:\myDisk.img ，存储整个文件卷中的所有信息。一个文件卷实际上就是一张逻辑磁块盘，磁盘中存储的信息以块为单位。每块 512 字节。

复习并深入领会 UNIX V6 文件管理系统的内核设计思想。

要求做到以下几点：

可以实现下列基础 API

```
void ls(); 列目录
Int fopen(char *name, int mode);
Void fclose(int fd);
Int fread(int fd, char *buffer, int length);
Int fwrite(int fd, char *buffer, int length);
Int flseek(int fd, int position);
Int fcreat(char *name, int mode);
Int fdelete(char *name)
```

同时做到创建目录，进入目录等简单的辅助功能，同样对应三个 API：

```
Void mkdir(char* dirname);
Void cd(char* dirname);
Void backDir()
```

二、设计思想说明

2.1 任务分析

一个文件系统从功能上划分程序为四个部分：

第一部分是有关高速缓冲区的管理程序，主要实现了对硬盘等块设备进行数据高速存取的函数；

第二部分代码描述了文件系统的底层通用函数，说明了文件索引节点的管理、磁盘数据块的分配和释放以及文件名与 i 节点的转换算法；

第三部分程序是有关对文件中数据进行读写操作，包括对字符设备、管道、块读写文件中数据的访问；

第四部分的程序与文件的系统调用接口的实现有关，主要涉及文件打开、关闭、创建以及有关文件目录操作等的系统调用。

二级文件系统不专门设计驱动程序，要模拟文件系统的设计、实现和功能，就不能把它直接作为操作系统实际的文件系统进行挂接。鉴于此，我在实际的硬盘上创建一个文件，把它作为我们的文件系统的磁盘进行各种对磁盘的模拟操作，这样做的好处是可以对它进行连续操作，只要在退出文件系统时，及时保存它的状态。

为了达到这样的效果，能方便该“磁盘文件”的操作，我们在实际的程序中调用 mmap 函数，将“磁盘文件”映射到内存中，将映射到的大内存空间当作整个二级文件系统的磁盘，

直接对它进行操作。在退出或手动刷新磁盘内容时，只需调用 msync 函数将该内存空间的值重新写入“磁盘文件”中，这样就保存了本次执行的一系列操作，在下一次再进入二级文件系统时能够继续操作。

2.2 设计任务剖析

在本次的课程设计中，采用了简化的 UNIX V6++的设计。不同点在于：

1. 只考虑单用户的在线操作，去除了各种类中的锁结构，不用考虑同步或是 cpu 抢占问题等；
2. UNIX 世界中一切皆文件的设计，使得它的文件系统中会包含很多特殊设备的处理函数和处理判断，如字符设备。在本次课设中只存在块设备，即我们的“磁盘文件”，故删除了关于设备号的一系列判断和使用；
3. 在 UNIX V6++中，一个缓存块至少存在于两个队列中，每个设备有自己的设备队列。而在本次课设中只会存在一个设备----“磁盘文件”，不存在设备驱动，于是我们使用 BufferManager 类直接管理所有的缓存块，在系统中只设置一个队列。缓存块的使用逻辑是：分配空闲缓存块时从队列头取出，Brelse 时只需将该缓存块移动到队列的尾部即可。这也符合了最基本的 LRU 算法的思想；
4. 磁盘文件的设计取消了磁盘最开始的 200 个引导块，SuperBlock 即为第 0 号磁盘块；在参数方面适当减小了 Diskinode 区块数和 datablock 的块数，作为实验程序够用即可。

作为二级文件系统，我们的文件系统其实与 EXT2 文件系统需要实现的功能类似，需要编写的程序要实现下列功能：

- 格式化程序。在硬盘上创建一个文件，用来模拟磁盘；对磁盘按照 EXT2 文件系统结构进行划分：

SuperBlock	DiskInode	……	DiskInode	一般数据块	……	一般数据块

DiskInode 和一般数据块的块数定义在 FileSystem 类中，在下文的类设计中将会描述。

在格式化磁盘时主要需要初始化 SuperBlock 类和一般数据块，UNIX V6++对空闲磁盘块的管理使用成组链接法，需要在 SuperBlock 块中初始化正确的磁盘序列，将空闲的数据块正确的分组，并为每一个分组中的第一个盘块写入索引数据。

同时格式化还需要初始化 0 号 DiskInode，它将在系统启动的时候作为指定的根目录 Inode 被直接读入内存，必须在格式化磁盘时就将其初始化成功。

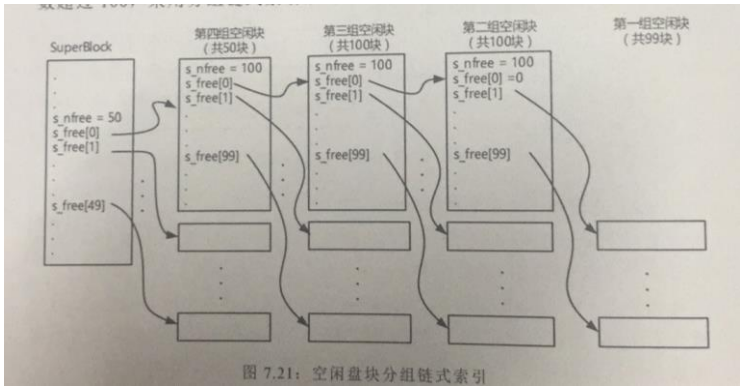


图 7.21：空闲盘块分组链式索引

- 磁盘读写函数。这部分函数我取消了 UNIX V6++中复杂而可扩展的设备驱动程序，

磁盘读写函数被定义在 BufferManager 类中，和 UNIX V6++相比这里只实现了延迟写函数并没有实现异步写函数，同时由于没有实现预读。

- 数据块分配和回收。数据块的分配和回收完整的按照成组链接法寻找空闲盘块，回收盘块。在这里不再赘述。

- 索引节点分配和回收。UNIX V6++中对空闲的外存索引并没有使用成组链接法，而是用外存索引表直接管理 100 个，当直接管理的外存 INODE 结点全部分配之后，直接搜索整个 DiskInode 区去找空闲的 i 节点，本次课程设计保持了这一设计。

- 目录操作函数。列表显示当前目录内目录项。其本质和读文件类似，它调用已经封装好的 fopen 和 fread 函数去读目录文件，并打印目录文件中的存在的所有目录项。

- 文件操作函数。组织进程和文件系统的关系，包括用户打开文件表和系统打开文件表的初始化和管理。创建文件、删除文件、打开文件、关闭文件、读文件和写文件。程序是引用文件描述符来操作文件的。
  在 UNIX V6++中系统调用的参数将被存放在 User 结构中，直接调用内核的类中函数是不会直接传入参数的，本次课设也保留这样的设计，在我编写的 API 中会将传入的参数直接放在 User 结构中的指定变量中，在类中会对 User 结构进行操作，最大程度的保留了 UNIX V6 的设计。

2.3 程序设计环境

运行平台：*@RHEL74-X64*
编 译 器：*线程模型：posix*
gcc 版本 4.8.5 20150623 (Red Hat 4.8.5-16) (GCC)

说明：因为是 64 系统在该版本的 gcc 编译下，指针将会是 8 位长度，这和 UNIX V6++的设计不同会带来一系列问题，如此时指针类型和 int 型之间的转化将会截断，C++会认为这是个不安全的操作而报错，所以需要在编译选项中加上 *-m32（详细可见 Makefile 文件）*可能会需要手动安装相应的 32 位库。特此说明。

三、详细设计

3.1 类功能和详细设计

文件系统超级块 SuperBlock

定义如下：

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

    /* Members */
public:
    int      s_isize;     /* 外存Inode区占用的盘块数 */
    int      s_fsize;     /* 盘块总数 */
    int      s_nfree;     /* 直接管理的空闲盘块数量 */
    int      s_free[100]; /* 直接管理的空闲盘块索引表 */
    int      s_ninode;         /* 直接管理的空闲外存Inode数量 */
    int      s_inode[100];/* 直接管理的空闲外存Inode索引表 */
    int      s_fmod;           /* 内存中super block副本被修改标志，意味着需要更新外
存对应的Super Block */
    int      s_ronly;     /* 本文件系统只能读出 */
    int      s_time;           /* 最近一次更新时间 */
    int      padding[49]; /* 填充使SuperBlock块大小等于1024字节，占据2个扇区 */

};
```

说明：由于是单用户模式的文件系统与 UNIX V6++相比取消了 s_flock 和 s_ilock 两个类成员，同时需要增加两个 padding，以保证 SuperBlcok 块仍为 1024 字节，占据两个盘块。

主要包含两类重要的数据：

（1） 用于外村索引节点的管理

s_isize：表示存储设备上的外存索引节点区占据的块数，这些值在格式化磁盘文件的时候确定，由 FileSystem 类中的静态变量直接定义；

s_ninode,s_inode[100]：SuperBlock 直接管理的空闲外村索引节点数量和索引表，索引表中记录的是 s_ninode 个空闲外存索引节点的编号；

（2） 用于空闲数据块的管理

S_fsize:表示文件系统的数据块总数；

S_nfree,s_free[100]:SuperBlock 直接管理的空闲块数和空闲索引表。

在操作系统初始化时，会将磁盘的 SuperBlock 读入一个内存的 SuperBlock 副本，以便于内核以更快的速度随时访问内存副本。一旦内存中的副本发生变化，会通过设置 s_fmod 标志，由内核将内存副本写入磁盘。

目录结构 DirectoryEntry

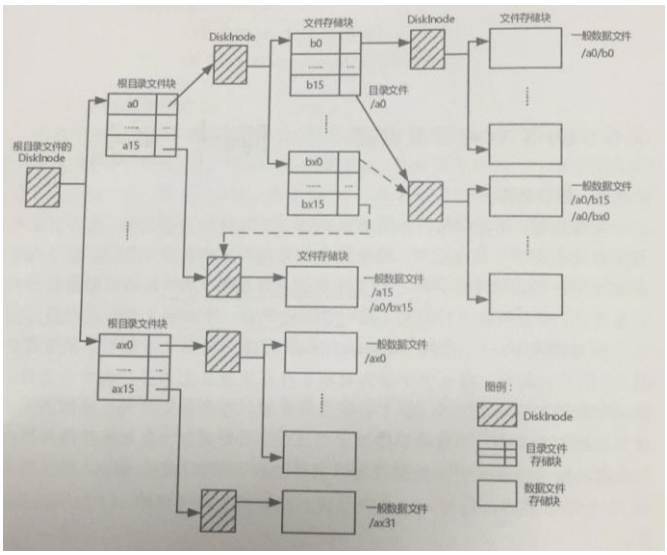
```
class myDirectoryEntry
{
    /* static members */
public:
    static const int DIRSIZ = 28;  /* 目录项中路径部分的最大字符串长度 */
public:
    /* Constructors */
    myDirectoryEntry();
    /* Destructors */
    ~myDirectoryEntry();

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

本次课设中采用和 UNIX V6++一样的树形带交叉勾连的目录结构。整个目录结构系统包含若干个目录文件，每个目录文件由一系列目录项组成。目录项是目录文件的基本构成单位，每一个文件系统中存在的文件对一定对应某一个目录文件中的一条目录项。

在 DirectoryEntry 中，前 4 个字节为对应文件在块设备上的外存索引节点号，它作为该文件的内部标识，后 28 字节为文件名，是文件的外部表示，于是文件目录项为其内外部标识建立了对照关系。一块数据盘块可容纳 512/32=16 个目录项。

值得一提的是 UNIX V6++文件系统的根目录文件，其索引节点是 DiskInode 区的 1#，指定位置在系统初始化时装载。

目录结构如下图：



内存 Inode 节点

```
class myInode
{
public:
    /* i_flag中标志位 */
    enum myInodeFlag
    {
        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 */
    };


    /* Members */
public:
    unsigned int i_flag;    /* 状态的标志位，定义见enum myInodeFlag */
    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 */
};
```

Inode 类的成员部分成员变量如上，与 DiskInode 相比增加了：

（1） 对应外存索引节点的位置信息：

被拷贝到内存的 Inode 中的索引节点数据需要知道它来自于哪个外存 DiskInode，以便

于将来内存副本被修改之后更新到外存对应的 DiskInode 中去，因此 Inode 类中包含了用于记录 DiskInode 位置信息的 i_number；

注意在 UNIX V6++中完整的内存 Inode 还记录了设备号，而由于本次课设中只有一个设备故删去了这一成员变量。

（2） Inode 状态标志位：

I_flag 用于指示该内存 Inode 当前状态，在 UNIX V6++系统中，该标志位主要用于记录该内存 Inode 是否上锁，是否被其他进程所需要，用于多进程的同步工作，而本次课设中不需要这样的操作，所以其主要功能体现在是否需要将该内存 Inode 更新到磁盘上的“脏”置位。

（3） 引用计数

I_count 指出该索引节点当前活跃的实例数目。例如有进程通过系统调用 Open()打开文件，则该内存 Inode 用的引用计数会+1，这是系统最优化内存使用空间的做法。同时如果引用计数为 0 则表示该 Inode 空闲，可以被分配它用；

（4） 预读判断

UNIX V6++中内存 Inode 会记录上次读取文件的逻辑块号以供预读判断使用，而本次课设中没有实现预读的功能，该变量未被使用。

成员函数的说明：

（1） ReadI()： 根据Inode对象中的物理磁盘块索引表，读取相应的文件数据，其中会判断调用BufferManager中的Bread函数，是read系统调用中比较底层的封装；

（2） WriteI(): 根据Inode对象中的物理磁盘块索引表，将数据写入文件。其中会调用Bmap它会将逻辑块号转变为物理块号，通过物理块号查找内存中是否存在相应的缓存块，存在则获取，不存在则新申请。同时在写入该缓存块时要检查是否能写满一个整块，若不能则需要先读后写以保护磁盘的原始数据，注意写入缓存块后并不会立刻同步到磁盘上，标志的为延迟写；

（3） Bmap(int lbn);将文件的逻辑块号转成对应的物理盘块号，这里会设计到举行文件的索引设计，不再赘述；

（4） IUpdate()：将内存Inode的值更新到对应的外存DiskInode中

（5） ITrunc()：释放Inode对应的文件占用的磁盘块

（6） Clean()：情况INode中的数据

（7） ICopy(myBuf* bp, int inumber)：将包含外存Inode字符块中的信息拷贝到内存Inode中

缓存控制块 Buf

```
class myBuf
{
public:
    enum BufFlag /* b_flags中标志位 */
    {
        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 b_flags; /* 缓存控制块标志位 */

    int       padding;      /
    myBuf*    b_forw;
    myBuf*    b_back;
    int       b_wcount;         /* 需传送的字节数 */
    unsigned char* b_addr;/* 指向该缓存控制块所管理的缓冲区的首地址 */
    int       b_blkno=-1;       /* 磁盘逻辑块号 */

    //注：不可能IO出错，不许要出错信息
};
```

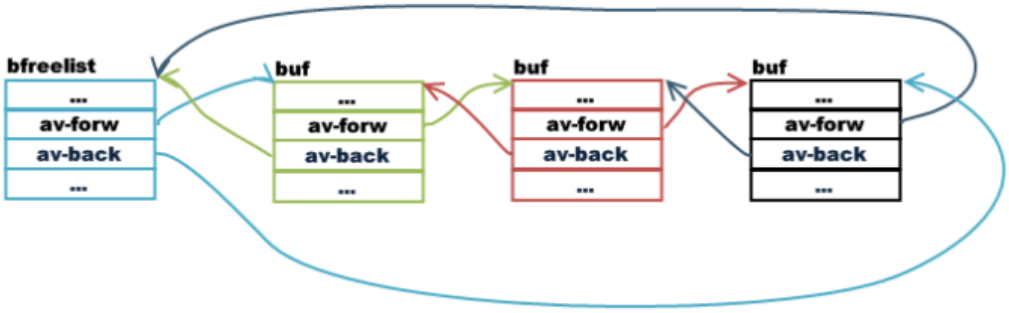
说明：Buf 控制块每一个缓存块都会对应一个缓存控制块，它会指明这个缓存块所在的队列位置。如上文所述，由于本次课程设计中不会存在多个设备，于是我取消了所有的设备队列，缓存块只会存在与 NODEV 队列中。分配和释放的操作也非常简单，分配只是简单的从队列头取第一个缓存块，释放时将该缓存块标志位置换后放在队列尾部。

需要将逻辑块号初始化为-1 否则为 0 时，系统处理是不存在该块号，因为 UNIX V6++中不可能读 0 号盘块，而本次课设中是存在的所以需要置初值以作区分。

缓存管理类 BufferManager

```
class myBufferManager
{
public:
    /* static const member */
    static const int NBUF = 15;             /* 缓存控制块、缓冲区的数量 */
    static const int BUFFER_SIZE = 512;/* 缓冲区大小。 以字节为单位 */
private:
    myBuf* InCore(int blkno); /* 检查指定字符块是否已在缓存中 */

private:
    myBuf bFreeList;                        /* 自由缓存队列控制块 */
    //一个buf对应一个存储块
    myBuf m_Buf[NBUF];                      /* 缓存控制块数组 */
    unsigned char Buffer[NBUF][BUFFER_SIZE];    /* 缓冲区数组 */

    //这是整个文件系统用mmap映射到内存后的起始地址
    char *p;
};
```

如上文所述，本次课设中没有多余的设备驱动，BufferManager 将直接管理这 15 块缓存。做初始化时，缓存块将被初始化在 NODEV 队列中，在本次课设中 NODEV 队列和自由队列是一样的。NODEV 是一个特殊的设备,表示无设备。队列头 为 bfreelist,bfreelist 同时作为自由队列的队列头。队列的形式如下图所示：



成员函数说明：

1. void Initialize(char *start)：初始化 BufferManager 在本次课设中只需初始化 NODEV 队列即可（UNIX V6++这段初始化的代码写的非常优雅）；
2. myBuf* GetBlk(int blkno)：申请一块缓存，用于读写 blkno，取消了 dev 设备号，该函数首先会在缓存队列中寻找是否该盘块已经在系统内存在缓存，若不存在将会从自由队列头开始搜索得到，缓存块，同时若该缓存块被置为延迟写也会将其台同步到对应的磁盘上；
3. void Brelse(myBuf* bp)：UNIX V6++中在该函数中对自由缓存队列进行初始化，和进行释放缓存后的自由缓存队列的调整，而由于我们在分配缓存是已经调整过缓存块的位置，故在此函数中只需对缓存块的标志位进行一些处理；
4. Bread(int blkno)：读一块磁盘，其中会调用 GetBlk 函数申请缓存块，UNIX V6++中会调用比较复杂的设备驱动等，本次课设中计算好对应的盘块位置直接用 memcpy 将其拷贝对应的缓存中；
5. Bwrite(myBuf* bp)：UNIX V6++也是需要调用更底层的设备驱动来等待 IO 完成，在本次课设中直接用 memcpy 拷贝到磁盘的指定位置；
6. Bdwrite(myBuf* bp)：在延迟写的函数中只是简单的置 buf 的标志位为延迟写，在手动 update 或该缓存被新分配时再调用 Bwrite 到磁盘文件中；
7. ClrBuf(myBuf* bp)：情况缓存的内容
8. Bflush()：遍历所有的缓存将需要同步的缓存同步到磁盘上，在本次课设中，在系统退出的时候会执行；
9. GetBFreeList()：返回缓存队列的头。

文件系统类 FileSystem

```
class myFileSystem
{
public:
    /* static consts */
    static const int SUPER_BLOCK_SECTOR_NUMBER = 0;  /* 定义SuperBlock位于磁盘上的扇区号，占据1,
2两个扇区。 */

    static const int ROOTINO = 0;           /* 文件系统根目录外存Inode编号 */

    static const int INODE_NUMBER_PER_SECTOR = 8;        /* 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_ZONE_START_SECTOR = 2;        /* 外存Inode区位于磁盘上的起始扇区号 */
    static const int INODE_ZONE_SIZE = 27 - 2/*1024 - 202*/;      /* 磁盘上外存Inode区占据的扇区
数 */ //注：有修改

    static const int DATA_ZONE_START_SECTOR = 27/*1024*/;     /* 数据区的起始扇区号 */     //注:有
修改
    static const int DATA_ZONE_END_SECTOR = 1000 - 1;     /* 数据区的结束扇区号 */
    static const int DATA_ZONE_SIZE = 1000 - DATA_ZONE_START_SECTOR;  /* 数据区占据的扇区数量 */


private:
    myBufferManager* m_BufferManager;       /* FileSystem类需要缓存管理模块(BufferManager)提供的
接口 */
};
```

说明：如上文提到的，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，这将直接影响我们的磁盘文件的大小；

成员函数说明：

- void Initialize();初始化成员变量，简单的从Kernel类中获取全局对象BufferManager的引用；
- void LoadSuperBlock();系统初始化时读入SuperBlock，注意所有的读入操作都是针对缓存的，有完整的缓存查找缓存，申请缓存和Bread的步骤，UNIX V6++是正统的面向对象的编程，各个类直接相互勾连又封装的无比完好；
- mySuperBlock* GetFS();根据文件存储设备的设备号dev获取该文件系统的SuperBlock，返回全局变量即内存副本Superblock的地址
- void Update();将SuperBlock对象的内存副本更新到存储设备的SuperBlock中去
- myInode* IAlloc();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。是通过SuperBlock中的索引查到的，若直接管理的空闲Inode已经全部分配，将会遍历整个DiskInode区去寻找空闲的Inode，于此同时重新装填直接管理的Inode节点；
- void IFree(int number); 释放存储设备dev上编号为number的外存INode，一般用于删除文件。这里只是简单将其加入SuperBlock的空闲Inode索引表中，因为在新分配的时候会重新置位的；
- myBuf* Alloc();在存储设备dev上分配空闲磁盘块，从SuperBlock的索引表的栈顶获得空闲的磁盘块编号，如果获取的编号为0，说明直接管理的索引表中没有空闲的磁盘块了，需要通过成组链接法去寻找空闲的磁盘块，对索引表做一系列的调整，同时申请到空闲磁盘块后还需要为该磁盘块申请缓存，调用GetBlk函数，又是一番复杂的操作了，申请到之后会清空缓存中的数据，最后返回该缓存对应的Buf控制块；

- void Free(int blkno)；释放存储设备dev上编号为blkno的磁盘块，主要是对SuperBlock中的索引表进行操作，当索引表管理的空闲磁盘号的栈已满，会重新申请缓存按照成组链接法的规则去做移动，并将需要释放的盘块计入索引表中即可，在它被新分配的时候会重新清零的，如上面的函数所作的那样。

打开文件控制块 File 类

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

    /* Functions */
public:
    /* Constructors */
    myFile();
    /* Destructors */
    ~myFile();


    /* Member */
    unsigned int f_flag;        /* 对打开文件的读、写操作要求 */
    int      f_count;           /* 当前引用该文件控制块的进程数量 */
    myInode* f_inode;           /* 指向打开文件的内存Inode指针 */
    int      f_offset;              /* 文件读写位置指针 */
};
```

说明：记录进程打开文件的读写请求类型，文件读写位置等动态信息。

F_flag：包含对打开文件请求类型，包括读写类型，本次课设中不考虑管道类型。

F_inode:指向一个打开文件的 Inode；

F_offset：是相对应打开文件进行读写的位置指针。文建刚打开是，读写位置指针初始值为 0，每次读写后，都将其移到已读写的下一字节；

F_count:是该 File 控制块的引用计数。若为 0 则表示该 File 空闲，可以分配作他用。

打开文件描述符表 OpenFiles 类

```
class myOpenFiles
{
    /* static members */
public:
    static const int NOFILES = 15; /* 进程允许打开的最大文件数 */

                                        /* Functions */
public:
    /* Constructors */
    myOpenFiles();
    /* Destructors */
    ~myOpenFiles();

    /*
    * @comment 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();


    /*
    * @comment 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
    */
    myFile* GetF(int fd);
    /*
    * @comment 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, myFile* pFile);

    /* Members */
private:
    myFile *ProcessOpenFileTable[NOFILES];      /* File对象的指针数组，指向系统打开
文件表中的File对象 */
};
```

说明：在本次课设中只存在一张文件打开描述符表存在于 User 结构之中，即我们只允许同时打开 15 个文件。

该类中提供的 AllocFreeSlot()h 桉树用于在打开文件描述符表中分配一个空闲的表项。该函数线性扫描打开文件描述符表，寻找 File 指针为 NULL 的空闲项分配，并将该空闲项在 OPenFIles::ProcessOpenFileTable[]数组中的索引作为打开文件描述符 fd,返回给执行 Open（）系统调用的进程，返回的 fd 即为相应被打开文件的 OPenFIles::ProcessOpenFileTable[]数组中的索引。

打开文件管理类 OpenFileTable

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

                                        /* Functions */
public:
    /* Constructors */
    myOpenFileTable();
    /* Destructors */
    ~myOpenFileTable();

    /*
    * @comment 在系统打开文件表中分配一个空闲的File结构
    */
    myFile* FAlloc();
    /*
    * @comment 对打开文件控制块File结构的引用计数f_count减1，
    * 若引用计数f_count为0，则释放File结构。
    */
    void CloseF(myFile* pFile);

    /* Members */
public:
    myFile m_File[NFILE];          /* 系统打开文件表，为所有进程共享，进程打开文件描述符
表
                                    中包含指向打开文件表中对应File结构的指针。*/
};
```

说明：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。 勾连关系指进程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。

成员函数：

FAlloc（）：在User结构中的打开文件描述符表中申请一个空闲项，即为文件句柄，然后在自己管理的系统打开文件表中寻找空闲的File结构，将申请的文件句柄与该结构勾连起来，返回File结构的地址；

CloseF（）：关闭文件控制块的操作很简单，如果File结构中的引用次数降低到0需要释放该File结构对应的内存Inode，否则只需要将File结构中的引用次数递减即可。

内存 Inode 表 InodeTable

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

                                        /* Functions */
public:
    /* Constructors */
    myInodeTable();
    /* Destructors */
    ~myInodeTable();

    /*
    * @comment 初始化对g_FileSystem对象的引用
    */
    void Initialize();
    /*
    * @comment 根据指定设备号dev，外存Inode编号获取对应
    * myInode。如果该Inode已经在内存中，对其上锁并返回该内存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
    */
    myInode* IGet( int inumber);
    /*
    * @comment 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(myInode* pNode);

    /*
    * @comment 将所有被修改过的内存Inode更新到对应外存Inode中
    */
    void UpdateInodeTable();

    /*
    * @comment 检查设备dev上编号为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);
    /*
    * @comment 在内存Inode表中寻找一个空闲的内存Inode
    */
    myInode* GetFreeInode();

    /* Members */
public:
    myInode m_Inode[NINODE];        /* 内存Inode数组，每个打开文件都会占用一个内存Inode */

    myFileSystem* m_FileSystem;     /* 对全局对象g_FileSystem的引用 */
};
```

说明：该类将会负责所有内存 Inode 的分配和释放。一组连续的内存 Inode 构成了一张内存文件索引节点表，当打开某一文件时，如果找不到其相应的内存 Inode，就在该表中分配一个空闲项，并将该文件的外存 inode 中的主要部分拷贝进去，然后填写相应的外存 DiskInode 的地址信息。当关闭文件时，如果相应的内存 Inode 已经没有其他用处，则被放弃以便移作他用，同时在释放前如果发现其被置为已修改标志，需要将其更新到 DiskInode 上。

成员函数：

- IGet（int inumber）：该函数首先调用 IsLeaded 函数遍历内存 Inode 表查看相应的

Inode 是否已经存在于内存之中，如果找到那么增加 Inode 的引用次数后直接返回该 inode，否则将会做比较复杂的申请 Inode 和读入磁盘上的 Inode 的流程；

- IPut(myInode* pNode)：减少指定的 inode 的引用计数，若计数将为 0，会判断是否仍存在目录路径指向它，以此来决定是否释放该文件占据的数据盘快，在引用计数为 0 是会直接将 inode 同步到磁盘上，即调用 IUpdate；
- UpdateInodeTable（）：对内存 Inode 表中的所有 Inode 执行 update 操作，将其刷新到磁盘文件中；
- IsLoaded（int inumber）：遍历内存 inode 表，是否存在对应可用的 inode；
- GetFreeInode（）：遍历内存 inode 表，返回空闲的 inode。

文件管理类 FileManager

```
class myFileManager
{
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 */
    myInode* 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 */
    myFileSystem* m_FileSystem;

    /* 对全局对象g_InodeTable的引用，该对象负责内存Inode表的管理 */
    myInodeTable* m_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 */
    myOpenFileTable* m_OpenFileTable;

    /*注：对全局对象g_spb的引用*/
    mySuperBlock *m_gspb;
};
```

说明：这是文件系统的各种系统调用的最高层接口，我们实现的 API 只需要调用一个对应的成员函数就能实现相应的功能。

这个类中定义了大部分的全局对象的引用，上述的所有类都在 FileManager 中存在引用。需要注意的是，不同的设备都有自己专属的 SuperBlock，故在 UNIX V6++中 SuperBlock 当然不可能定义在 File Manager 中，但在本次课设中只有一个 SuperBlock，我还是在 FileManager 类中添加了对 SuperBlock 的引用。

成员函数：

- void Initialize();初始化对全局对象的引用，同时会调用InodeTable的初始化程序
- void Open();Open()系统调用处理过程，会调用NameI函数返回对应的Inode节点，之后在以读相应的mode调用Open1函数，会经历复杂的过程打开文件，返回相应的文件句柄；

- void Creat();Creat()系统调用处理过程，会调用NameI函数，同样返回对应文件的Inode，如果已经存在这个需要创建的文件，将会让删除这个文件，如果不存在该文件则进入创建文件的正常流程，调用MakNode之后调用Open1返回文件句柄；
- void Open1(myInode* pInode, int mode, int trf); Open()、Creat()系统调用的公共部分，该函数将会以不同的trf来对应不同的过程，而相同的是在该函数中需要申请空闲的File结构，建立File结构和内存Inode的勾连关系；
- void Close();Close()系统调用处理过程，获取打开文件控制块File结构，释放打开文件描述符fd，递减File结构引用计数
- void Seek();Seek()系统调用处理过程，该函数会判断User结构中的参数，在按规则修改对应的File结构中的偏移量等参数；
- void FStat();FStat()获取文件信息
- void Stat();FStat()获取文件信息
- void Stat1(myInode* pInode, unsigned long statBuf); FStat()和Stat()系统调用的共享例程。获取文件的参数信息，本次课设中直接通过memcpy拷贝到用户提供的地址；
- void Read();Read()系统调用处理过程，直接调用Rdwr
- void Write();Write()系统调用处理过程，直接调用Rdwr
- void Rdwr(enum myFile::FileFlags mode); 读写系统调用公共部分代码，在获得fd对应的File结构之后，进行一系列的User结构中的参数搬运，设置偏移量和需要读写的字节数之后，分别调用WriteI和ReadI;
- myInode* NameI(char(*func)(), enum DirectorySearchMode mode); 目录搜索，将路径转化为相应的Inode，返回上锁后的Inode
- static char NextChar();获取路径中的下一个字符
- myInode* MakNode(unsigned int mode); 被Creat()系统调用使用，用于为创建新文件分配内核资源
- void WriteDir(myInode* pInode); 向父目录的目录文件写入一个目录项
- void SetCurDir(char* pathname); 设置当前工作路径
- int Access(myInode* pInode, unsigned int mode); 检查对文件或目录的搜索、访问权限，作为系统调用的辅助函数
- void ChDir();改变当前工作目录
- void UnLink();取消删除文件

文件的 IO 参数类 IOParameter

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

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

说明：存放文件读写时需要用到的读写偏移量，字节数以及目标区域的首地址等参数。在 User 结构中存在一个对象，其中的成员变量往往作为中间 temp，一个简单的系统调用背后将会是极其复杂的嵌套调用，各个调用之间并不会以形参传递这些必要的参数，而是将这些参数存在 User 结构的某个对象当中，就如这个 IOParmeter 一样，在某个需要使用的函数中间再拿出来使用，而不用一直将其作形参一层层的传递。这种设计思想在我们今后自己的程序中也可以使用到。

User 结构 User

```
class myUser
{
public:
        /* 系统调用相关成员 */


    int u_arg[5];              /* 存放当前系统调用参数 */
    char* u_dirp;              /* 系统调用参数(一般用于Pathname)的指针 */

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

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

    ErrorCode u_error;             /* 存放错误码 */

    int u_ar0;//注：本次课设中简化为int形

                           /* 文件系统相关成员 */
    myOpenFiles u_ofiles;      /* 进程打开文件描述符表对象 */

                           /* 文件I/O操作 */
    IOParameter u_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 */
};
```

说明：相比于 UNIX V6++本次课设中的 User 结构做了大规模的简化，删除了关于进程

管理的所有对象，改变了 u_ar0 的类型。只保留了与本次文件系统有关的成员。

int u_arg[5]：这是我们的 API 存放参数的变量，在 UNIX V6++中，某一个系统调用使用的函数往往是存放在 User 结构中的指定变量中的，例如在文件系统中的 FileManager 的一系列系统调用接口都是无形参的，在这些函数中将会使用记录在 User 结构中的这些参数，否则一层层传递形参，也是不小的工作量；

char* u_dirp：在文件系统的系统调用中经常存在 pathname 作为形参，此时作为存储参数的 Int u_arg 会比较不方便，于是在 fopen，fdelete 等 API 中 pathname 这些字符床形的参数将作为指针存储在 u_dirp 中；

myInode* u_cdir：指向当前目录目录文件的指针，在系统初始化时需要将其指向根目录；

myInode* u_pdir：指向父目录，这个变量在创建文件中十分重要，它需要需要写入目录项的目录文件；

其他参数在上文都有过介绍不再赘述，User 结构相当于一个 temp，它的参数在内核的所有类中都能够轻松访问到。

文件系统内核 Kernel 类

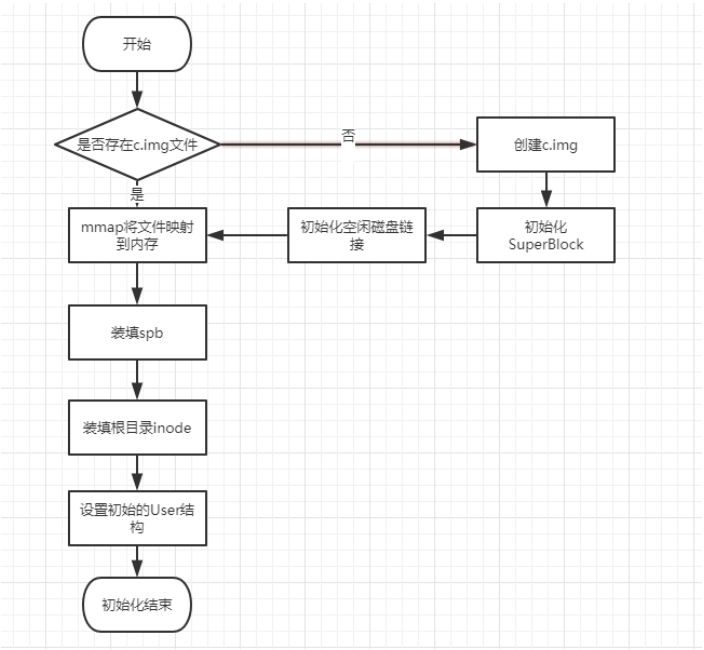
```
class myKernel
{

public:
    myKernel();
    ~myKernel();
    static myKernel& Instance();
    void Initialize(char*p);        /* 该函数完成初始化内核大部分数据结构的初始化 */

    myBufferManager& GetBufferManager();
    myFileSystem& GetFileSystem();
    myFileManager& GetFileManager();
    myUser& GetUser();          /* 获取当前进程的User结构 */

private:
    void InitBuffer(char*p);
    void InitFileSystem();

private:
    static myKernel instance;       /* Kernel单体类实例 */

    myBufferManager* m_BufferManager;
    myFileSystem* m_FileSystem;
    myFileManager* m_FileManager;
    myUser *m_u;

};
```

说明：Kernel 用于封装本次文件系统中的全局实例对象，UNIX V6++的内核使用的是单体模式，即在 Kernel 类其中定义一个静态的 Kernel 对象，保证内核中封装的各个内核模块的对象只有一个副本，initialize（）函数即获取各个对象的引用，并调用它们的初始化函数。

四、API 实现及主要函数分析

说明：文件系统拥有超级庞大的体系，函数的嵌套调用无比复杂，在这里我只会详细叙述 fcreat API 的执行流程，考虑到详细的流程图在 word 文档上难以展示，我采用了文字、箭头、缩进的方式来尽量还原整个流程。

系统初始化



系统初始化时会检测是否存在 c.img 文件，如果存在则直接装载“磁盘”即可，如上文所述，本次课程设计为了方便对“磁盘”进行操控，直接使用 mmap 函数将“磁盘”映射到用户内存空间，直接进行操作。

如果不存在 c.img 文件则会创建文件并对其进行格式化，格式化的主要内容包括：

a） 按照 FileSystem 的定义申请相应的数据结构

b） 初始化 SuperBlock，填写 inode 和空闲磁盘块的初始索引表

c） 成组链接法初始化空闲磁盘块

再将格式化的磁盘文件映射到内存空间。

然后会进行二级文件系统的初始化，主要包括：

1. 从“磁盘”读 SuperBLock 和根目录 inode 到内存

2. 设置初始的 user 结构，主要包括当前目录等

至此系统的初始化结束。

Fcreate 函数流程

→进入 fcreate（）函数，从内核中获取 User 结构和 FileManager 的引用，将参数设置到 User 结构中，调用 FileManager 中的 creat 接口

→进入 FileManager.Creat()函数，获得相应类的引用后调用 NameI 函数

→进入 FileManager.NameI,该函数将按路径搜索到对应项的内存 inode 节点，此时在根目录下 pInode 指向内存根目录 inode，执行 IGet 函数检查 inode 是否正在被使用，保证在目录搜索的过程中不会被释放

→ 进入 InodeTable.IGet,调用 IsLoaded(int inumber)函数检查内存中是否有对应 inode

→ InodeTable.IsLoaded,我们在系统初始化时读入了内存 inode，很显然是能找到的找到相应内存 inode 节点，编号为 0，增加 inode 引用计数后，直接返回 inode 地址,IsLoaded()函数返回

→ InodeTable.IGet，找到对应 inode 返回内存 inode 地址，函数返回

→ FileManager.NameI,函数继续，会进行一系列的判断操作，确保搜索的是目录文件且拥有相应权限。系统第一次执行创建文件的指令，此时还没有生成根目录文件，UNIX V6++的处理方式非常的优雅，并没有立即创建根目录文件，它的处理方法是和正常创建文件一样的，此时检查到是以 creat 方式进入函数的，没有相应检索的目录项直接返回 NULL，NameI 函数返回

→FileManager.Creat()函数继续执行，此时检查到 NameI 返回为 NULL，将为该文件创建自己的 i 节点和目录项，调用 MakNode 函数

→ 进入 FIleManager.MakNode,执行 IAlloc 用来申请磁盘上空闲的 Inode

→进入 FileSystem.IAlloc,获得 SuperBlock 的副本后存在直接管理的空闲 inode，直接分配，获得空闲 Inode 编号为 ino=99 再调用 IGet 将这个 inode 读入内存

→进入 InodeTable.IGet,调用 IsLoaded 函数检查内存中是否存在这一副本

→进入 InodeTable.IsLoaded，刚刚分配的是空闲的外存 inode 显然，内存不存在，IsLoaded 函数返回-1

→ InodeTable.IGet 函数继续执行，内存没有该外存 Inode，只有新申请一个空闲的内存 inode，执行 GetFreeInode

→进入 GetFreeInode，遍历内存 Inode 表，直接返回空闲的 inode 地址

→ InodeTable.IGet 函数继续执行，获得空闲 inode 之后，调用 Bread 函数，将该外存 inode 读入缓冲区

→进入 BufferManager.Bread，首先根据传入的盘块号申请缓存，调用 GetBlk

→进入 BufferManager.GetBlk，首先遍历缓存队列查看队列中是否已经存在相应的缓存，在这里是一个新申请的空闲 inode 盘块，显然缓存队列中不会存在，需要重新分配，于是 GetBlk 函数会从我们的自由队列的队列头摘下首个缓存，检查它没有延迟写标志，则清空它，并将它放置到自由队列的尾部，设置初始参数，最后返回它的缓存控制块，GetBlk 函数返回

→BufferManager.Bread 函数继续执行，申请到缓存块后，本次课设做的十分简单只需要找到映射到内存的“磁盘”文件的指定位置，用 memcpy 拷贝出来即可，Bread 函数成功返回

→InodeTable.IGet 函数继续执行，读入整个盘块后还不够，一个盘块中存在多个 inode，我们需要提取出其中的对应 Inode 项，于是执行 ICopy 函数

→进入 Inode.ICopy，该函数通过 inumber 定位到缓存中的指定地址，将它赋值给我们分配的这个内存 inode，ICopy 成功返回

→ InodeTable.IGet 函数继续执行,将会调用 Brelse 函数释放分配的缓存，这个函数十分简单，在这里不再列出，此后 IGet 函数成功返回

→ FileSystem.IAlloc 继续执行，经过复杂的操作后终于获得可使用的内存 inode 清理空闲 inode 的数据，IAlloc 成功返回 inode 的指针

→FIleManager.MakNode 继续执行，成功为新建的文件申请到 inode，此时将调用 WriteDir 函数写入目录项了

→进入 FileManager.WriteDir，将要写入的目录项 inode 号和文件名都存放在 User 结构中，共下层的函数直接调用，之后调用 WriteI 函数写入父目录文件

→ 进入 Inode.WriteI，注意这里的 WriteI 函数是由我们在 NameI 函数中提到的记录在 User 结构中的父目录 Inode 调用的，这里我们 User 结构中的偏移量只能得到逻辑地址，所以还要通过 Bmap 函数将逻辑盘块号转化为物理盘块号

→进入 Inode.Bmap 函数，这里我们会查看 inode 节点中的 i_addr 文件索引结构，显然的这个初始运行的系统该目录文件根本不存在，这时候就是 UNIX V6++系统的高明之处了，如之前的步骤一样虽然我们创建了这个 text 文件但其实系统并没有帮他申请磁盘，在下次需要写入的时候，在 Bmap 函数中会帮它申请盘块。在这里对这个目录文件做的工作是一样的，对 Bmap 函数来说它不知道目录文件和普通文件的区别它通过 inode 来写入文件内容，当没有属于该 inode 的磁盘块时，它会自动帮这个文件申请空闲磁盘，于是我们系统初始化时明明不存在的根目录目录文件在此时，会被 Bmap 一步步申请空间创建出来，在这里 Bmap 会调用 Alloc 函数（注意与上文的 IAlloc 函数做区分）

→进入 FileSystem.Alloc 函数，新的系统 SuperBlock 中有非常多空闲的磁盘块我们直接摘下最尾端的磁盘块，此时同样的我们需要通过高速缓存来读取和使用这个磁盘块，调用 GetBlk 函数，申请空闲缓存

→进入 BufferManager.GetBlk，该函数的使用在上文有详细提及不再赘述，成功分配到可用的缓存，getBlk 将成功返回

→FileSystem.Alloc 函数函数继续执行，申请到磁盘和缓存后需要清空缓存中的数据，执行 ClrBuf 函数，这个函数很简单，在这里掠过，然后 Alloc 函数成功返回

→Inode.Bmap 函数继续执行，终于成功获得磁盘和缓存，在目录 inode 的所以表上填上该外存磁盘号，Bmap 返回这个物理盘块号，Bmap 成功返回

→Inode.WriteI 函数继续执行，终于获得 Bmap 返回的物理磁盘号，此时判断需要写入的 32 字节不满一个盘块，于是需要先执行 Bread 函数将该磁盘的内容读出再写入缓存来避免污染原始数据，调用 Bread 函数，但其实这是有改进余地的，UNIX V6++的设计是想函数尽量的通用，我们看到 WriteI 函数简单的判断写不满一个盘块就会直接调用 Bread，但我们现在创建的其实是一个新文件，读入我们分配的空闲盘块其实是没有意义的，在我们这次课设中这个问题不显，其实在真正的操作系统中这意味着一次没有意义的磁盘 IO，是很耗时的，这里还有优化的余地

→进入 BufferManager.Bread 函数，这里的操作和上述调用 Bread 是类似的，他需要 GetBlk 获得缓存，再直接用 memcpy 拷贝到缓存，这里就直接略过了， Bread 函数成功返回

→ Inode.WriteI 函数继续执行,此时通过 User 结构中的偏移量计算出写入数据的起始位置，这是一个新文件当然是从 0 起始位置开始写，UNIX V6++的处理时将数据从 User 结构拷贝到缓存块后并不会立即更新到磁盘上，而是

置该缓存延迟写的标志，然后直接释放缓存，终于执行到这里目录文件的写入算是完成了，WriteI 函数正确返回；

→FileManager.WriteDir 函数继续执行，处理后续工作，执行 IPut 减少根目录的引用计数，这里略过，WriteDir 函数继续返回

→ FIleManager.MakNode 继续执行,接收返回，MakNode 函数整个返回

→ FileManager.Creat()继续执行此时创建 text 文件的 inode 返回，需要执行 Open1 函数，来打开文件，这里主要处理的是文件描述符和 File 结构的分配和勾连了

→进入 FileManager.Open1 函数，为该文件分配打开文件控制块 File 结构，调用 FAlloc 函数（注意与上文中的 Alloc 和 IAlloc 函数做区分）

→进入 OpenFileTable.FAlloc，这里调用 AllocFreeSlot 函数在进程打开文件描述符表中获得一个空闲项，这里的处理很简单，遍历数组接的返回一个空闲的 fd，然后继续申请分配 File 结构同样是遍历的方法，找到空闲的 File 块后需要调用 SetF 函数建立 fd 和 File 结构的勾连关系，很简单在 ProcessOpenFileTable 数组中填入即可，这里略过这两个函数的处理过程

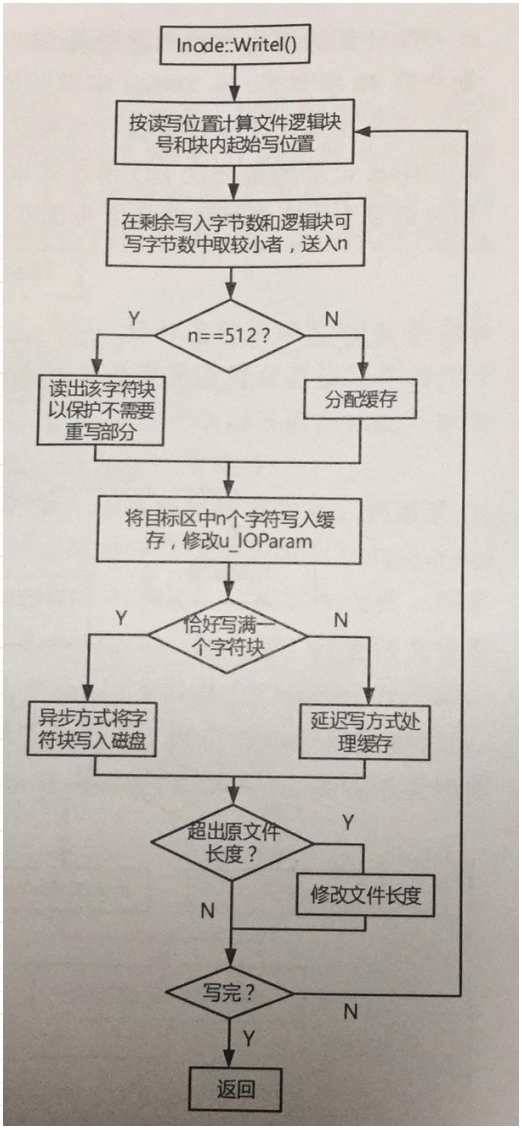
→FileManager.Open1 函数成功执行完毕，返回

→ FileManager.Creat()函数终于至此成功返回

→fcreatAPI 成功返回

Fwrite 函数

函数的主要调用关系,和书上的主要 WriteI 函数的流程图,其中写更详细的部分在 fcreat 中有提到，write 的过程和写目录项是相同的。



Fopen 函数

同样的将参数传入 User 结构中后直接的调用 FileManager 类中提供的 Open 接口。同时在 Open 函数中封装的也十分简单，调用 NameI 函数后以传入的 mode 模式调用 Open1 执行完则 Open 函数返回。

NameI 的主要功能就是通过 User 结构中的 pathname 和指向当前目录的 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 如 creat()函数一样，如上文所述返回 NULL 时 creat()函数会创建 inode。可以见得 NameI 函数也是个复用的非常好的函数，UNIX V6++中充满了这样精巧的设计，高复用的设计使得函数十分优雅，但是虽然程序员们极力使这些能够复用的函数在应对不同的情况时都能够最优而不做多余的操作，但事实上这是不可能的，就如上文中提到的系统会 Bread 为文件新分配的盘块，这就是高复用带来的一些问题----可能会有一定程度上的资源浪费。这里贴上书上的 NameI 函数的流程图，以及 UNIX V6++中的目录打开结构：

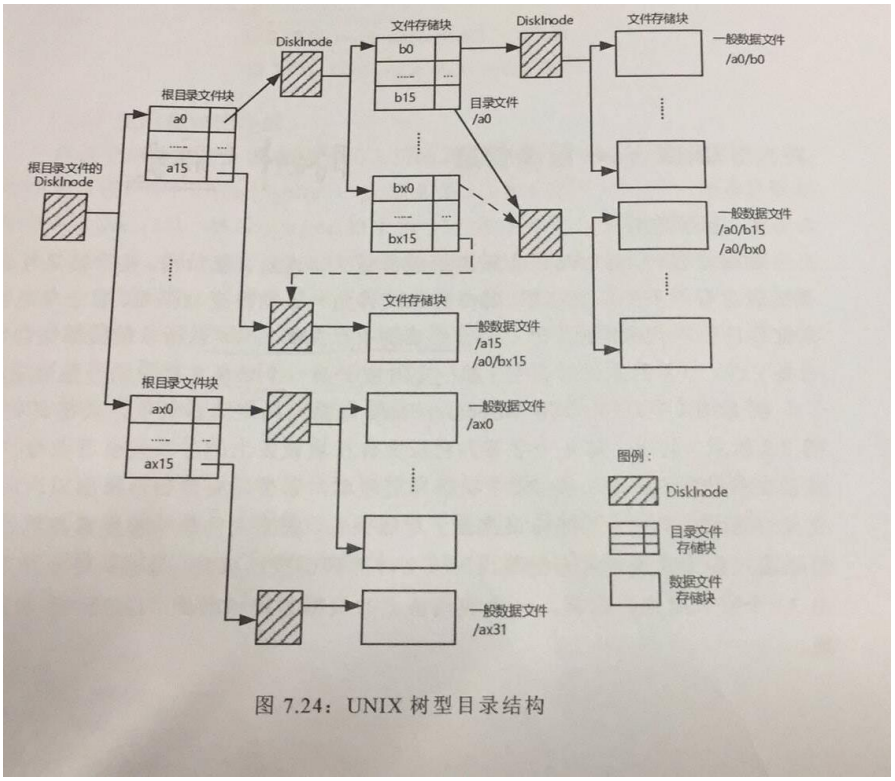
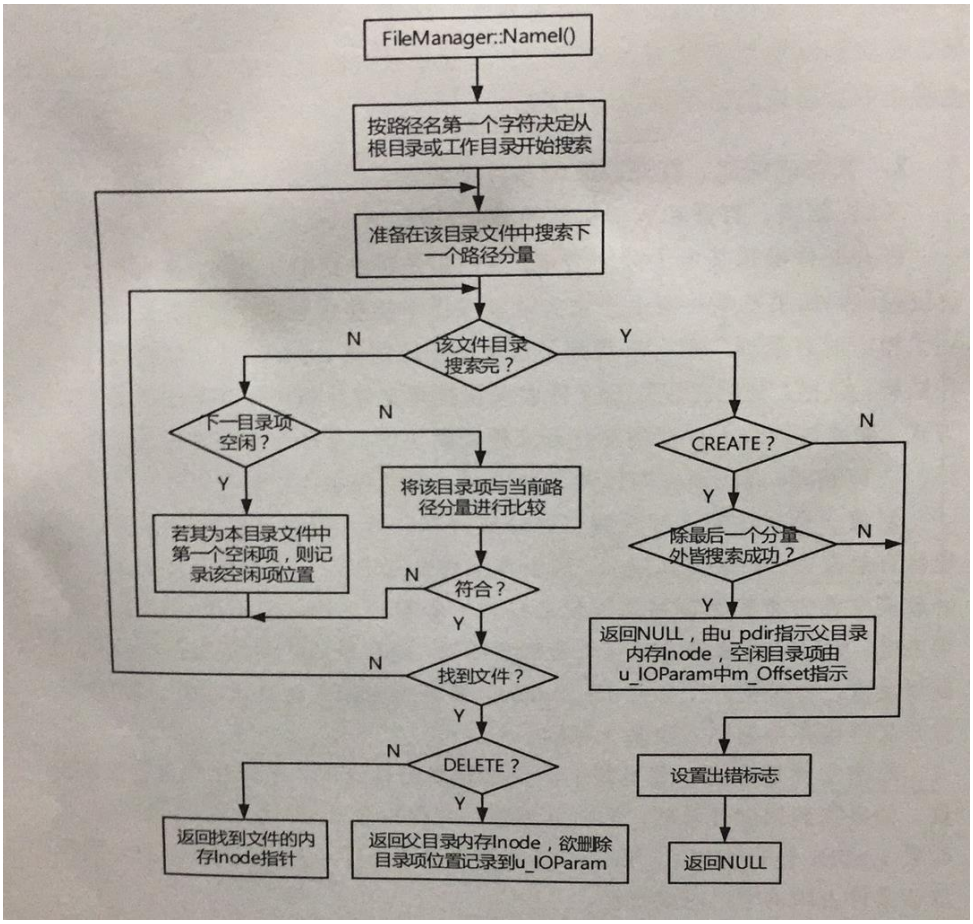


图 7.24：UNIX 树型目录结构



在 NameI 返回找到的 Inode 之后将会调用 Open1 函数，和上文提到的一样，这个函数主要是申请空闲的 File 结构和空闲的文件句柄，负责这两者的分配，处理的流程比较

简单，在上文也有过叙述，在这里就略过了。

Flseek 函数

在 UNIX V6++的基础上，API 的实现一般都比较简单，都是将 User 结构作为一个参数传递的 temp，然后直接调用最顶层的内核接口即可。Seek 函数的实现也是这样，直接调用 FileManager 中的 Seek()函数即可。

Seek()函数的实现比较简单，只需要通过传入的文件描述符找到自己对应的 File 结构修改其中的偏移量即可。具体实现代码如下：

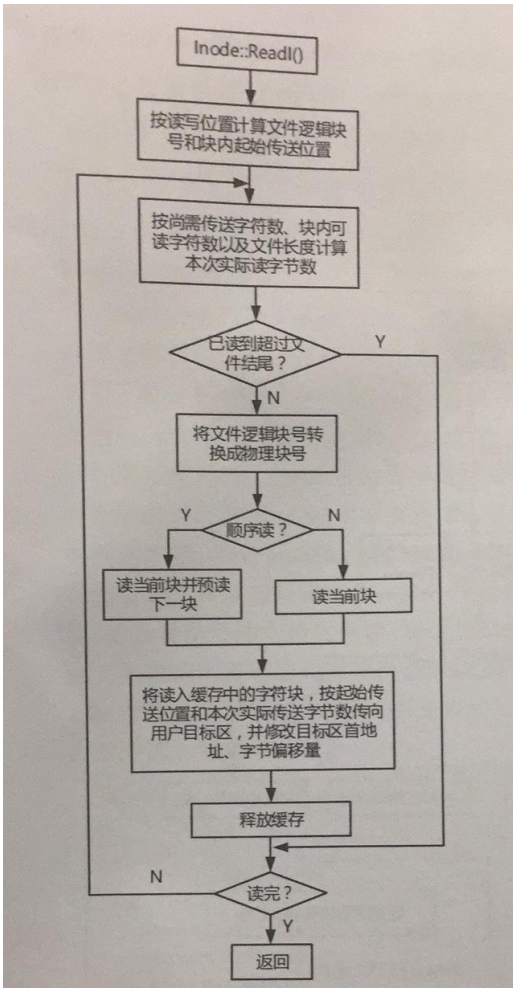
```
void myFileManager::Seek()
{
    cout << "FileManager.Seek" << endl;
    myFile* pFile;
    myUser& u = myKernel::Instance().GetUser();
    int fd = u.u_arg[0];

    pFile = u.u_ofiles.GetF(fd);
    if (NULL == pFile)
    {
        cout << "没找到对应的file结构 seek错误返回" << endl;
        return;  /* 若FILE不存在，GetF有设出错码 */
    }
    int offset = u.u_arg[1];

    /* 如果u.u_arg[2]在3 ~ 5之间，那么长度单位由字节变为512字节 */
    if (u.u_arg[2] > 2)
    {
        offset = offset << 9;
        u.u_arg[2] -= 3;
    }

    switch (u.u_arg[2])
    {
        /* 读写位置设置为offset */
    case 0:
        pFile->f_offset = offset;
        break;
        /* 读写位置加offset(可正可负) */
    case 1:
        pFile->f_offset += offset;
        break;
        /* 读写位置调整为文件长度加offset */
    case 2:
        pFile->f_offset = pFile->f_inode->i_size + offset;
        break;
    }
    cout << "Seek成功返回 pFile->f_offset=" << pFile->f_offset<< endl;
}
```

Fread 函数

Fread 函数与 fwrite 的调用方式类似。并且在 FileManager 中的 Write 和 Read 函数处理方式相同，他们将以不同的调用方式调用复用 Rwdr 函数，Rwdr 函数在进行参数的处理并

申请得到 File 结构块后会按照调用方式选择调用 WriteI 或者 ReadI。

ReadI 的处理相较于 WriteI 更加简单（毕竟 WriteI 中可能也包括读文件的操作），ReadI 处理的过程从宏观上来说，就是处理 User 结构确定需要读入的磁盘号，然后调用更底层的 Bread 来进行磁盘和缓存的操作。下面附上书上的 ReadI 的处理流程，具体的函数嵌套关系省略。



需要说明的是由于时间原因在本次课程设计中没有设置预读操作。

Ls 函数

Ls 函数是本次实现的 API 中唯一一个不能直接调用一个接口函数就能实现的 API 了。Ls 函数的本质是打开并读完所有的目录文件中的目录项，打印出文件名。所以 ls 函数是通过 fopen 和 fread 函数一起实现的。

将 User 结构中的当前目录的完整路径作为 fopen 的参数打开文件，并一条条读出所有的目录项。

```
void ls()
{
    myUser &u = myKernel::Instance().GetUser();
    int fd = fopen(u.u_curdir, (myFile::FREAD) );
    char temp_filename[32] = { 0 };
    for (;;)
    {
        if (fread(fd, temp_filename, 32) == 0) {
            cout << "ls成功返回" << endl;
            return;
        }
        else
        {
;
            myDirectoryEntry *mm = (myDirectoryEntry*)temp_filename;
            cout << "======" << mm->m_name << "======" << endl;
            memset(temp_filename, 0, 32);
        }
    }

}
```

以上是一个简单的 ls 函数的实现。

Fdelete 函数

Fdelete API 对应的是 FIleManager 中的 UnLink()函数。UnLink 函数同样是调用 NameI 函数寻找需要删除的文件所在目录的目录 inode，同时从 User 结构中可以获得当前需要删除文件的目录项的 inode 部分，通过 IGet 函数获得这个内存 Inode 的引用。现在我们得到了两个 inode，一个目录文件的 inode，一个需要删除文件的 inode。

我们需要通过目录文件的 inode 找到目录文件，在原本目录项的指定位置写入清零后的目录项（这需要调用 WriteI 函数，具体的流程在上面已经叙述过）。然后利用 IPut 函数释放掉这两个 inode，事实上真正的释放操作是由 Iput 函数完成的。

在 IPut 函数中会做判断 inode 的引用计数如果为 0，这时才会找到对应文件的数据盘快释放掉该文件的数据盘快，这个操作是由 ITrunc 函数实现的。

ITrunc 函数需要做的是将这些盘块重新装填到 SuperBlock 的空闲盘块索引表中，当然这个操作在某些时候也是很复杂的，因为成组链接法索引的空闲盘块，SuperBlock 直接索引的空闲盘块只有 100 块，如果当前要加入的直接索引表已满，我们需要找到未满 100 块的空闲磁盘组将这一个空闲盘块加进去。虽然说来简单但只要涉及到读写的操作，我们都是通过缓存和我们的磁盘打交道的，特别是这个操作可能会涉及到多个磁盘块的读写。

Mkdir 函数实现

该 API 的实现直接调用 FIleManager 类中的 MkNod()函数。 MkNod 函数的执行流程如下：

同样的首先调用 NameI 函数来查找获取将要创建的文件的 inode，在这里正常的 NameI 返回为 NULL，表示不存在这个目录可以创建，如果返回非 NULL，说明当前文件夹下已存在一个同名文件。很显然创建目录时不能像 Creat 的逻辑，直接删除这个文件，所以这里会

直接识别为错误，函数返回。

通过以上判断，允许创建，则会调用 MakNode 函数来创建该文件的 inode，完成所有的操作。MakNode 的函数详细执行流程在 fcreate 函数的分析中已有过叙述，在这里不再赘述。

这样 MkNod 函数成功返回，创建目录成功。

Cd 函数的实现

Cd API 的实现对应 FileManager 中的 Chdir 函数，ChDir 的执行流程如下：

同样首先调用 NameI 函数获取需要进入的目录的 inode 节点，很显然如果 NameI 返回 NULL 没有找到该目录 inode，cd 函数会错误返回，同时 ChDir 会判断搜索到的 inode 是否对应目录文件，操作者是否有执行权限。

获得 inode 并经过必要的检查之后，ChDir 函数会调用 SetCurDir 函数，它会将传入的路径拷贝或加到 User 结构的当前完整目录路径 u.u_curdir 上。

至此 ChDir 成功返回。

backDir 函数的实现

我对 backDir 实现的功能是返回上级目录，我对该 API 的处理比较简单，我会从 user 结构中的 u.u_curdir 中手动提取完整的上级目录，再通过 cd 函数进入那个目录。

五 界面以及检验的流程

在这里只演示下面这个非常简单的流程：

1. *创建一个名为 text.txt 的文件*
2. *向文件中写入一定的字符*
3. *打开该文件（create 函数固定是只写模式，必须重新打开）*
4. *移动文件的读写指针*
5. *在该位置读出一定的字符，并打印，观察是否正确*
6. *查看该目录下的文件*
7. *关闭该文件*
8. *创建目录*
9. *查看目录是否创建成功*
10. *Cd 进入目录*
11. *创建文件*
12. *查看文件是否创建成功*
13. *返回上级目录*
14. *删除第一次创建的文件*
15. *重新试图打开文件，查看是否会返回文件不存在*

启动程序

```
[root@RHEL74-X64 OS]# ./myos
格式化磁盘完毕
Initialize Buffer...OK.
Initialize File System...OK.
Initialize File Manager...OK.
进入系统初始化：装填spb和根目录inode,设置user结构中的必要参数
系统初始化结束
                    类UNIX V6++二级文件系统实验
                                ********
                            Welcome to Linux World
            CopyRight @ Xuesen Huang Tongji University  2018
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
================================================================
||请输入需要执行的API的对应编号，如下所示：                          ||
||a   fopen(char *name, int mode)                                ||
||b   fclose(int fd)                                            ||
||c   fread(int fd, int length)                                 ||
||d   fwrite(int fd, char *buffer, int length)                  ||
||e   flseek(int fd, int position, int ptrname)                 ||
||f   fcreat(char *name, int mode)                              ||
||g   fdelete(char *name)                                       ||
||h   ls()                                                      ||
||i   mkdir(char* dirname)                                      ||
||j   cd(char* dirname)                                         ||
||k   backDir()--返回上级目录                                    ||
||q  退出文件系统                                               ||
```

这是第一次运行程序所以并没有检测到 c.img 文件，如上图所示会显示 **格式化磁盘完毕** 当检测到存在 c.img 文件时不会存在该步操作。接下来的 Initialize 提示是初始化整个系统。

执行创建选项

```
||SecondFileSystem@ 请输入编号>>f
||SecondFileSystem@ 请输入第一个参数文件名>>test.txt
||SecondFileSystem@ 请输入第二个参数，创建方式>>1
create成功 返回fd=0
```

提示 creat 成功，返回文件句柄为 0，本次课设的文件句柄是直接从 0 开始的。

执行 fwrite 操作，写入 hello,world!十三个字符，写完毕成功返回 13.

```
create成功 返回fd=0
================================================================
||请输入需要执行的API的对应编号，如下所示：                          ||
||a   fopen(char *name, int mode)                                ||
||b   fclose(int fd)                                            ||
||c   fread(int fd, int length)                                 ||
||d   fwrite(int fd, char *buffer, int length)                  ||
||e   flseek(int fd, int position, int ptrname)                 ||
||f   fcreat(char *name, int mode)                              ||
||g   fdelete(char *name)                                       ||
||h   ls()                                                      ||
||i   mkdir(char* dirname)                                      ||
||j   cd(char* dirname)                                         ||
||k   backDir()--返回上级目录                                    ||
||q  退出文件系统                                               ||

||SecondFileSystem@ 请输入编号>>d
||SecondFileSystem@ 请输入第一个参数文件句柄>>0
||SecondFileSystem@ 请输入第二个参数，写入数据>>hello,world!
||SecondFileSystem@ 请输入第三个参数，写入数据的长度>>13
||SecondFileSystem@ 写返回为13
```

执行 fopen 操作，以读写权限，打开文件，返回文件句柄为 1。

```
||SecondFileSystem@ 请输入编号>>a
||SecondFileSystem@ 请输入第一个参数文件名>>test.txt
||SecondFileSystem@ 请输入第二个参数，打开方式>>3
||SecondFileSystem@ open 返回fd=1
```

执行 fseek 操作，将文件的偏移量设置为 6

```
||SecondFileSystem@ 请输入编号>>e
||SecondFileSystem@ 请输入第一个参数文件句柄>>1
||SecondFileSystem@ 请输入第二个参数，移动位置>>6
||SecondFileSystem@ 请输入第三个参数，移动方式>>0
```

执行 fread 操作，读出 5 个字节，可以看到读出的是 world，很显然是正确的.

```
||SecondFileSystem@ 请输入编号>>c
||SecondFileSystem@ 请输入第一个参数，文件句柄>>1
||SecondFileSystem@ 请输入第二个参数，读出的数据长度:5
||SecondFileSystem@ read返回5
||SecondFileSystem@ 读出数据为:
world
```

执行 ls()函数，可以看到输出的是我们刚刚创建的 test.txt 文件的文件名

```
||SecondFileSystem@ 请输入编号>>h
||SecondFileSystem@ 输出如下>>
======test.txt======
--------------------------------------------------------------------
```

执行 mkdir 函数创建目录 testdir

```
||SecondFileSystem@ 请输入编号>>i
||SecondFileSystem@ 请输入第一个创建目录名>>testdir
--------------------------------------------------------------------
```

再次执行 ls 函数，可看到 dir 成功创建

```
||SecondFileSystem@ 请输入编号>>h
||SecondFileSystem@ 输出如下>>
======test.txt======
======testdir======
```

执行 cd 函数进入目录

```
||SecondFileSystem@ 请输入编号>>j
||SecondFileSystem@ 请输入第一个进入目录名>>testdir
```

在这里我们新创建文件 test1

```
||SecondFileSystem@ 请输入编号>>f
||SecondFileSystem@ 请输入第一个参数文件名>>test1.txt
||SecondFileSystem@ 请输入第二个参数，创建方式>>1
create成功 返回fd=3
```

执行 ls 函数查看该目录下的文件

```
||SecondFileSystem@ 请输入编号>>h
||SecondFileSystem@ 输出如下>>
======test1.txt======
```

执行 backDir 函数返回上级目录

```
||SecondFileSystem@ 请输入编号>>k
backDir
将返回根目录
/
```

经过 fdelete 之后再打开文件，可以看到显示打开的文件不存在：

```
||SecondFileSystem@ 请输入编号>>a
||SecondFileSystem@ 请输入第一个参数文件名>>test.txt
||SecondFileSystem@ 请输入第二个参数，打开方式>>3
发生错误NameI返回NULL
找的文件不存在噢
```

演示完毕

六 文件说明

本次课设提交的文件如下

```
BufferManager.cpp
BufferManager.h
Buf.h
demo1.cpp
file.cpp
file.h
filesystem.cpp
filesystem.h
inode.cpp
inode.h
Kernel.cpp
Kernel.h
Makefile
Makfile
openfilemanager.cpp
openfilemanager.h
user.h
```

Buf.h 中定义了 myBuf 类
BufferManager.h 中定义了 myBufferManager 类
File.h 中定义了 myFile，myOpenFiles，IOParemeter 三个类
Filesystem.h 中定义了 mySuperBlock，myFileSystem，myDirectoryEntry 三个类
Inode.h 中定义了 myInode，DiskInode 两个类
Kernel.h 中主要定义了 myKernel 类
Openfilemanager 中主要定义了 myOpenFIleTable，myInodeTable 两个类
User.h 中主要定义了 myUser 类

对应的 cpp 文件是对应头文件的类的实现，而 demo.cpp 中实现了磁盘的格式化，系统的初始化，和文件操作的 API。

最后 make 出的可执行文件名为 myos

七 总结

通过本次课设更加完整深刻的理解了 UNIX V6++文件系统的设计。在阅读了 UNIX 的源码后我很难想象最初的系统是怎么由一两个人全部写完的。各种类的定义，各种函数的嵌套非常的复杂而优雅。

在最开始动手写课设的时候比想象中的要困难很多，事实上实现整个课设的要求并不困难，在我的设想中实现这些功能大概只需要 500-600 行代码即可完成。但是如果想要移植 UNIX V6++的代码则工作量大太多了。最开始我根本不知道如何入手（当然这可能是我对它的理解不够深刻），我几乎迷失在无穷无尽的类定义之中，函数一层嵌套一层。移植源码绝对比阅读理解源码难得多，我很难直接移植一个所需的函数，因为它往往要调用十几个更多的函数提供支持。最开始我苦于复杂的的设备驱动，其中的部分内联汇编更是让我不知道这东西是怎么跑起来的。

我大概花了一个星期的课余时间来理清整个系统的函数调用关系，这时我才开始对我所需要架构的文件系统有一个比较清晰的认识。我需要一个单用户单进程的文件系统，于是 UNIX 中的各种锁我不需要；我们只有一个设备，于是我不需要有设备号与十分复杂的设备缓存队列；我们的二级文件系统默认挂载，不会有新的文件挂载，于是我们不需要 UNIX 中的 mount 设计；我们的二级文件系统是通过 mmap 映射到内存的，直接调用 memcpy，于是 UNIX 中复杂的设备驱动我不需要。这最终定下了我们最后二级文件系统的概貌。

接下来是实现虽然仍然 bug 甚多，但总比开始时摸不着头脑来的顺利些。我移植所需要的代码删减掉不需要的，得到了最终的成品，编写完 API 和界面的总代码量大约在 4000 行左右，虽然大部分的代码都是移植的 UNIX V6++的代码，但完成这么大的工程确实成就感还是挺大的。

其中深刻体会到了在较大的工程中 debug 的困难之处，我不得不承认在 linux 下我对 gdbg 的使用技能几乎为 0，最终还是通过 printf 大法（执行一个函数打印很多条提示语句）的帮助完成了整个工程。

当然本次的课设还有非常多的不足，事实上我还摘取了很多可用的内核函数，但是并没有编写更高层的 API，主要还是时间精力不太够。这学期的课设和课程巨多无比，我很难静下心来继续研究操作系统的设计。我确实觉得研究操作系统还是需要一段连续的专注的研究的。本次课设收获良多。