****

**数据库系统原理课程设计报告**

Operation System Course Design Report

**Unix二级文件系统**

|  |  |
| --- | --- |
| **学 号：** |  |
| **姓 名：** |  |
| **专 业：** | **计算机科学与技术** |
| **授课教师：** |  |
| **完成日期：** | **2021年6月6日** |

目录

[**1 需求分析** 2](#_Toc74066883)

[1.1 课程设计任务重述 2](#_Toc74066884)

[1.2 程序功能 3](#_Toc74066885)

[1.3 程序的输入输出 4](#_Toc74066886)

[1.4 可行性分析 5](#_Toc74066887)

[**2 概要设计** 5](#_Toc74066888)

[2.1 开发环境说明 5](#_Toc74066889)

[2.2 任务分解 5](#_Toc74066890)

[2.3 数据结构定义 6](#_Toc74066891)

[2.3.1 磁盘文件结构的数据结构定义 6](#_Toc74066892)

[2.3.2文件目录结构的数据结构定义 10](#_Toc74066893)

[2.3.3文件打开结构的数据结构定义 10](#_Toc74066894)

[2.3.4高速缓存结构的数据结构定义 14](#_Toc74066895)

[2.4 模块间的调用关系 16](#_Toc74066896)

[2.5 主程序流程与算法说明 16](#_Toc74066897)

[**3 详细设计** 17](#_Toc74066898)

[3.1 重点函数与重点变量 17](#_Toc74066899)

[3.2 重点功能的算法流程图及代码 22](#_Toc74066900)

[3.2.1 DiskINode分配算法 22](#_Toc74066901)

[3.2.2 DiskINode回收算法 23](#_Toc74066902)

[3.2.3 空闲盘块分配算法 23](#_Toc74066903)

[3.2.4 空闲盘块回收算法 24](#_Toc74066904)

[3.2.5 索引节点获取算法 25](#_Toc74066905)

[3.2.6 索引节点释放算法 25](#_Toc74066906)

[3.2.7 目录搜索算法 26](#_Toc74066907)

[3.2.8 文件索引与数据块的转换算法 27](#_Toc74066908)

[3.2.9 高速缓存的相关算法 27](#_Toc74066909)

[3.3 函数调用关系 28](#_Toc74066910)

[**4 运行结果分析** 29](#_Toc74066911)

[4.1 文件系统初始化 29](#_Toc74066912)

[4.2 部分命令的测试 30](#_Toc74066913)

[4.3 读写与文件指针综合测试及分析 33](#_Toc74066914)

[4.3.1 测试要求 33](#_Toc74066915)

[4.3.2 测试截图 33](#_Toc74066916)

[4.3.2 测试分析 34](#_Toc74066917)

[**5 用户使用说明** 36](#_Toc74066918)

[**6 实验总结** 37](#_Toc74066919)

[6.1 收获与感悟 37](#_Toc74066920)

[6.2 遇到的问题和解决方案 38](#_Toc74066921)

[6.3 后续工作 38](#_Toc74066922)

[6.4 对课设的认识 39](#_Toc74066923)

[**7 参考文献** 39](#_Toc74066924)

# **1 需求分析**

## 1.1 课程设计任务重述

本次“操作系统课程设计”的任务为模拟实现Unix二级文件系统，包括数据结构的定义和算法的实现，以及配套的人机交互界面。

该项目的任务具体需要实现以下部分：

**（1）Unix V6++文件卷**

使用一个普通的大文件（如 c:\myDisk.img ，称之为一级文件）来模拟 UNIX

V6++的一个文件卷（把一个大文件当一张磁盘用）。

**（2）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
* 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

**（3）文件目录结构**

* 目录文件结构
* 目录检索算法的设计与实现

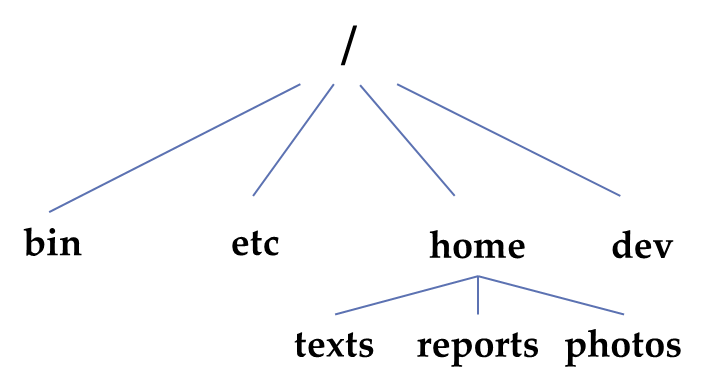
**（4）文件打开结构**

**（5）磁盘高速缓存**

**（6）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
* flseek： 定位文件读写指针
* fdelete： 删除文件
* ……

**（7）主程序**

* 格式化文件卷
* 用mkdir命令创建子目录，建立如图所示的目录结构
* 
* 把课设报告，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；
* 图形界面或者命令行方式，等待用户输入；
* 根据用户不同的输入，返回结果。

**（8）通过命令行方式测试**

* 新建文件/test/Jerry，打开该文件，任意写入800个字节
* 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
* 将abc写回文件
* 观察结果是否正确，并详细解释每一步的工作过程

## 1.2 程序功能

操作系统中与管理文件有关的软件和数据统称为文件管理系统（简称文件系统）。从系统角度，文件系统是对文件的存储空间进行组织/分配，负责文件的存储并对存储的文件进行保护、检索的系统；从用户角度，文件系统主要实现了对文件的按名存取。

本程序需要模拟Unix操作系统进行文件操作，这些文件操作包括格式化文件卷、创建文件目录、进入/退出文件目录、创建/删除文件、列出当前目录的所有文件、读写文件、调整文件指针等。

程序需要完成的功能分析如下：

|  |  |
| --- | --- |
| 命令及调用格式 | 说明 |
| fformat | 格式化文件系统 |
| ls | 查看当前目录内容 |
| mkdir <dirname> | 生成一个文件夹 |
| cd <dirname> | 进入一个文件夹下 |
| fcreate <filename> | 创建一个名为filename的打开文件 |
| fdelete <filename> | 删除名为filename的文件 |
| fopen <filename> | 打开文件名为filename的文件 |
| fclose <filename> | 关闭文件名为filename的文件 |
| fread <filename> <nbytes> | 读出文件filename从当前文件指针开始的nbytes个字符 |
| fwrite <filename> <nbytes>  <string> | 将文件string的前nbytes个字符写入当前文件指针指向的地方 |
| fseek <filename> <offset>  <ptrname> | 将filename的文件指针向指定方向ptrname移动offset个字节 |
| fmount <file1> <file2> | 将文件file1拷贝到文件file2中 |
| exit | 退出当前的文件系统 |
| init | 按要求初始化文件系统，使之具有以下目录格式：    在/home/texts放入课程报告Report.pdf  在/home/reports放入说明文件ReadMe.txt  在/home/photos放入一张图片Photo.png |

## 1.3 程序的输入输出

采用类似于Windows的cmd界面进行程序的输入输出。

程序的输入仿照linux操作系统的指令输入方式，可以键入的指令包括：fformat, ls, cd, mkdir, fcreat, fopen, fclose, fread, fwrite, flseek, fdelete。具体的使用方法在上一部分已经指出。

程序的输出则是在执行相应的操作命令后，返回句柄输出、提示信息输出、出错信息输出、文件内容输出等信息。

## 1.4 可行性分析

在《操作系统原理》课程上，我们系统的学习过一般操作系统的设计原理和Unix V6++的设计原理，其中文件系统是学习的一个重点。

课程详细讲述了Unix V6++文件系统的数据结构定义，文件的打开结构，各种与文件相关的算法执行流程等，并且还下发了一份完整的、可执行的源代码。

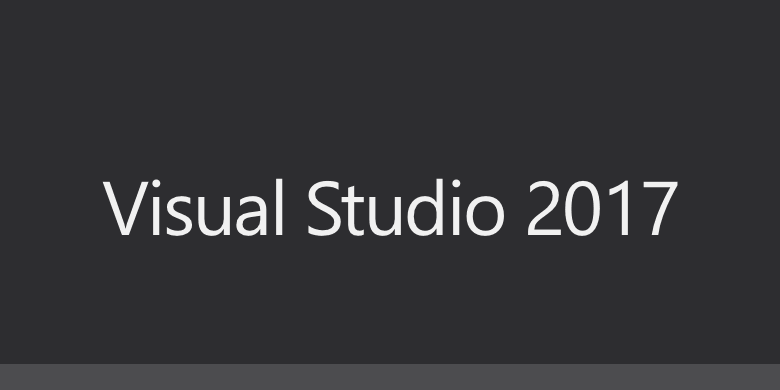
在这样的基础上，按照Unix V6++的设计思想，自己动手完成一个简易的二级文件管理系统是具有可行性的。

# **2 概要设计**

## 2.1 开发环境说明

根据需求分析的结果，本程序采用C++程序设计语言进行开发。使用的集成开发环境（IDE）为 Micorsoft Visual Studio 2017。

VS是一个基本完整的开发工具集，它包括了整个软件生命周期中所需要的大部分工具，如UML工具、代码管控工具、集成开发环境(IDE)等等。所写的目标代码适用于微软支持的所有平台，可移植性较好。



## 2.2 任务分解

根据理论课《操作系统原理》中学习到的关于Unix V6++文件系统的组织方式，不难对任务进行划分。总的来说，该项目应该分解为以下独立的模块去完成：

* 磁盘文件结构
* 文件目录结构
* 文件打开结构
* 文件操作接口
* 主程序部分

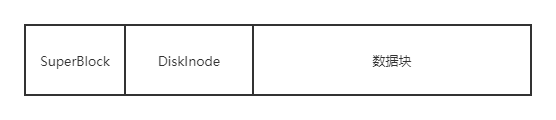
具体的设计如下：

* DiskDriver：最底层磁盘模块驱动器，它直接负责硬盘文件myDisk.img的创建、读写和初始化。
* BufferManager：高速缓存管理器，负责缓存块的申请、释放、读写等操作，直接与DiskDriver进行交互。
* FileManager：磁盘文件管理器，负责磁盘文件结构的管理，包括管理SuperBlock、INode和DiskINode等数据结构。
* OpenFileManager：打开文件管理器，负责管理文件的打开结构与内存INode 表。实现对打开文件的文件描述符的分配，实现在文件关闭时对文件描述符的回收等。
* SystemCall：系统调用模块，实现了各文件操作的系统接口，衔接用户对文件的操作以及对内核数据结构进行相应处理。
* UserCall：用户接口模块，实现用户输入指令与系统调用的联系，其中需要对一些可能出现的错误情况进行错误处理和错误提示等。

## 2.3 数据结构定义

### 2.3.1 磁盘文件结构的数据结构定义

Unix V6++的磁盘文件结构如下：



包括了SuperBlock、DiskInode块和一般的数据块。

（1）SuperBlock类

Unix V6++对整个块设备存储空间的不同区域采用不同的管理算法，而所有所需的信息都集中在SuperBlock这一数据结构中。通过这一数据结构可以知道还有多少空闲数据块、多少空闲外存索引节点等。

类的定义如下：

|  |
| --- |
| class SuperBlock  {  public:  SuperBlock() {};  ~SuperBlock() {};  public:  const static int MAX\_NUMBER\_FREE = 100;  const static int MAX\_NUMBER\_INODE = 100;  public:  int s\_isize;  int s\_fsize;  int s\_nfree;  int s\_free[MAX\_NUMBER\_FREE];  int s\_ninode;  int s\_inode[MAX\_NUMBER\_INODE];  int s\_fmod;  int padding[51];  }; |

成员变量及成员函数的含义如下：

|  |  |
| --- | --- |
| 成员变量 | 含义 |
| s\_isize | 外存Inode区占用的盘块数 |
| s\_fsize | 盘块总数 |
| s\_nfree | 直接管理的空闲盘块数量 |
| s\_free | 直接管理的空闲盘块索引表 |
| s\_ninode | 直接管理的空闲外存Inode数量 |
| s\_inode | 直接管理的空闲外存Inode索引表 |
| s\_fmod | 内存中super block副本被修改标志 |
| padding | SuperBlock块的填充 |

与Unix V6++相比，此处省略了用于对盘块进行封锁和与更新时间相关的成员变量。

（2）DiskInode类

DiskInode又叫外存索引节点，是存放文件控制信息的物理块，它是以静态形式存储在磁盘上存放文件索引信息的区域，包含了文件的全部控制信息。

类的定义如下：

|  |
| --- |
| class DiskInode  {  public:  DiskInode() {};  ~DiskInode() {};  public:  unsigned int d\_mode;  int d\_nlink;  int d\_size;  int d\_addr[10];  int padding[3];  }; |

成员变量及成员函数的含义如下：

|  |  |
| --- | --- |
| 成员变量 | 含义 |
| d\_mode | 状态的标志位 |
| d\_nlink | 文件联结计数器 |
| d\_size | 文件大小 |
| d\_addr | 文件逻辑块号和物理块号转换的基本索引表 |
| padding | 填充，使DiskInode类占64个字节 |

与Unix V6++相比，此处省略了用于标号用户进程、最后访问时间、最后修改时间的成员变量。

（3）FileSystem类

FileSystem类对整个磁盘文件组织的结构作出了详细的规定。

类的部分定义如下：

|  |
| --- |
| class FileSystem  {  public:  /\* 系统中用于挂载子文件系统的装配块数量 \*/  static const int NMOUNT = 5;  /\* 定义SuperBlock位于磁盘上的扇区号，占据200，201两个扇区。 \*/  static const int SUPER\_BLOCK\_SECTOR\_NUMBER = 200;  /\* 文件系统根目录外存Inode编号 \*/  static const int ROOTINO = 0;  /\* 外存INode对象长度为64字节，每个磁盘块可以存放512/64 = 8个外存Inode \*/  static const int INODE\_NUMBER\_PER\_SECTOR = 8;  /\* 外存Inode区位于磁盘上的起始扇区号 \*/  static const int INODE\_ZONE\_START\_SECTOR = 202;  /\* 磁盘上外存Inode区占据的扇区数 \*/  static const int INODE\_ZONE\_SIZE = 1024 - 202;  /\* 数据区的起始扇区号 \*/  static const int DATA\_ZONE\_START\_SECTOR = 1024;  /\* 数据区的结束扇区号 \*/  static const int DATA\_ZONE\_END\_SECTOR = 18000 - 1;  /\* 数据区占据的扇区数量 \*/  static const int DATA\_ZONE\_SIZE = 18000 - DATA\_ZONE\_START\_SECTOR;  ……  }; |

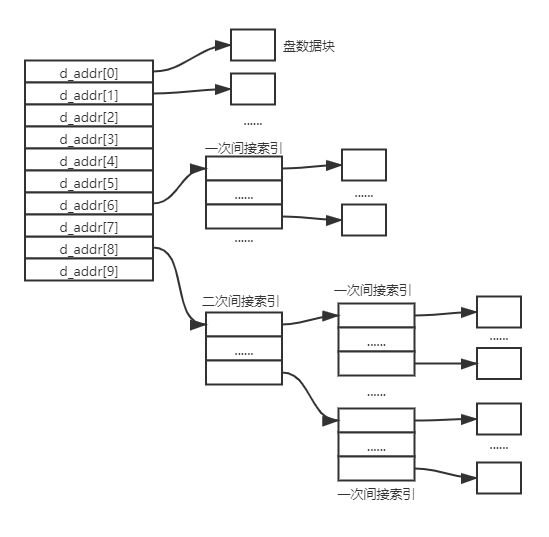
这些常变量给出了磁盘文件结构的具体组织形式。

（4）文件索引结构

Unix V6++采用了混合索引结构，文件按大小分为小型文件、大型文件、巨型文件。小型文件采用直接索引方式，占用最多6个数据块；大型文件采用一次间接索引方式，最多占用6+2\*128个数据块；巨型文件采用二次间接索引方式，最多占用6+2\*128+2\*128\*128个数据盘块。

|  |
| --- |
| /\* 文件逻辑块大小: 512字节 \*/  static const int BLOCK\_SIZE = 512;  /\* 每个间接索引表(或索引块)包含的物理盘块号 \*/  static const int ADDRESS\_PER\_INDEX\_BLOCK = BLOCK\_SIZE / sizeof(int);  /\* 小型文件：直接索引表最多可寻址的逻辑块号 \*/  static const int SMALL\_FILE\_BLOCK = 6;  /\* 大型文件：经一次间接索引表最多可寻址的逻辑块号 \*/  static const int LARGE\_FILE\_BLOCK = 128 \* 2 + 6;  /\* 巨型文件：经二次间接索引最大可寻址文件逻辑块号 \*/  static const int HUGE\_FILE\_BLOCK = 128 \* 128 \* 2 + 128 \* 2 + 6; |

文件数据保存在磁盘设备上的位置信息包含在DiskINode节点中的文件索引表d\_addr[10]数组中，DiskINode的定义在上一部分，其索引结构示例图如下：



### 2.3.2文件目录结构的数据结构定义

Unix V6++采用树形带交叉勾连的目录结构。整个目录结构包含有若干个目录文件，每个目录文件由一系列目录项组成。使用 DirectoryEntry 数据结构定义目录项。

DirectoryEntry的定义如下：

|  |
| --- |
| 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:  DirectoryEntry() {};  ~DirectoryEntry() {};  public:  int inode;  char name[DIRSIZ];  }; |

成员变量及成员函数的含义如下：

|  |  |
| --- | --- |
| 成员变量 | 含义 |
| inode | 目录项中Inode编号部分 |
| name | 目录项中路径名部分 |

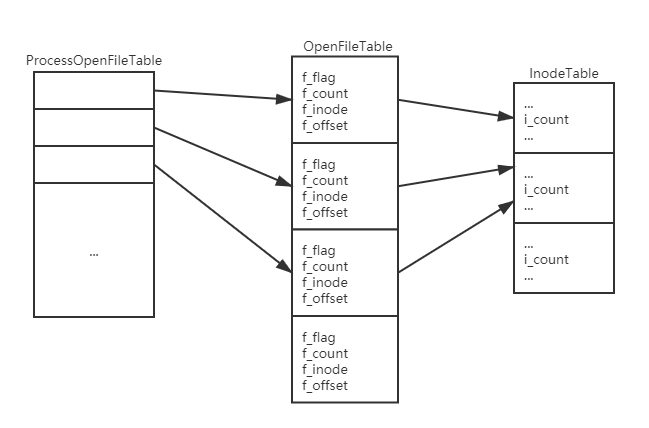
通过这样的目录结构，可以实现从文件名到文件本身的一一映射，从而从磁盘中找到我们的数据文件。同时，也规定了文件名的最大长度为28个字符。

### 2.3.3文件打开结构的数据结构定义

由于Unix V6++的文件系统相当庞大，若每次对文件的访问都需要按照查找文件名—>获得文件路径—>根据索引结构将数据读入内存—>进行读写—>写回外存区的操作流程进行，将会使得文件系统的工作效率极为低下。

为了提高文件系统的效率，需要在内存设置一个高效的文件管理结构，以方便对文件进行高效的访问。因此，Unix V6++有文件打开结构这一数据结构。

Unix V6++文件打开结构包括了以下五个数据结构：INode，InodeTable，File，OpenFileTable，OpenFiles，其逻辑关系如下图所示：



（1）内存INode类

Unix V6++设置了内存INode，与DiskInode类相似，但成员变量略有增加。

INode类的定义如下：

|  |
| --- |
| class Inode  {  public:  unsigned int i\_flag;  unsigned int i\_mode;  int i\_count;  int i\_nlink;  int i\_number;  int i\_size;  int i\_addr[10];  }; |

成员变量及成员函数的含义如下：

|  |  |
| --- | --- |
| 成员变量 | 含义 |
| i\_flag | 状态的标志位 |
| i\_mode | 文件工作方式信息 |
| i\_count | 引用计数 |
| i\_nlink | 文件联结数，即文件在目录树中不同路径名的数量 |
| i\_number | 外存inode区中的编号 |
| i\_size | 文件大小 |
| i\_addr | 用于文件逻辑块号和物理块号转换的基本索引表 |

与Unix V6++相比，此处省略了用于标志设备、用户标识、最近一次读文件的逻辑块号的成员变量。

（2）InodeTable类

Unix V6++设置了一张内存文件索引节点表去管理内存INode，它是用来封装内存索引节点的数据结构。用于管理内存INode，建立内存INode与DiskINode的关联。

类的定义如下：

|  |
| --- |
| class InodeTable  {  public:  static const int NINODE = 100; /\* 内存Inode的数量 \*/  public:  InodeTable();  ~InodeTable();  public:  Inode m\_Inode[NINODE]; /\* 内存Inode数组，每个打开文件都会占用一个内存Inode \*/  }; |

成员变量及成员函数的含义如下：

|  |  |
| --- | --- |
| 成员变量 | 含义 |
| m\_Inode | 内存Inode数组，每个打开文件将占用一个Inode |

（3）打开文件控制块File类

在Unix V6++中，一个文件可以同时被一个进程或者多个进程同时打开，可以用同一或者不同的路径名打开，可以用不同的操作要求打开。但内存INode并没有记录这些信息，因此需要打开文件控制块File类，以便记录和文件打开有关的信息。

类的定义如下：

|  |
| --- |
| class File  {  public:  enum FileFlags  {  FREAD = 0x1, /\* 读请求类型 \*/  FWRITE = 0x2, /\* 写请求类型 \*/  FPIPE = 0x4 /\* 管道类型 \*/  };  public:  File();  ~File();  public:  unsigned int f\_flag;  int f\_count;  Inode\* f\_inode;  int f\_offset;  }; |

成员变量及成员函数的含义如下：

|  |  |
| --- | --- |
| 成员变量 | 含义 |
| f\_flag | 对打开文件的读、写操作要求 |
| f\_count | 当前引用该文件控制块的进程数量 |
| f\_inode | 指向打开文件的内存Inode指针 |
| f\_offset | 文件读写位置指针 |

这一个类的定义与Unix V6++的该类的定义完全一致。

（4）OpenFileTable类

在Unix V6++中，需要用OpenFileTable类来封装系统文件打开表，从本质上来说，其就是一个类型为File的数组，数组每一项都是一个File文件控制块。

类的定义如下：

|  |
| --- |
| class OpenFileTable  {  public:  static const int NFILE = 100; /\* 打开文件控制块File结构的数量 \*/  public:  OpenFileTable() {};  ~OpenFileTable() {};  public:  File m\_File[NFILE];  }; |

成员变量及成员函数的含义如下：

|  |  |
| --- | --- |
| 成员变量 | 含义 |
| m\_File | 系统打开文件表 |

这一个类的定义与Unix V6++的该类的定义完全一致。

（5）OpenFiles类

OpenFiles类管理每个进程的打开文件，其本质上也是一个File类的数组。

类的定义如下：

|  |
| --- |
| class OpenFiles  {  public:  static const int NOFILES = 15; /\* 允许打开的最大文件数 \*/  public:  OpenFiles();  ~OpenFiles() {};  public:  File \* k\_OpenFileTable[NOFILES]; /\* File对象的指针数组，指向系统打开文件表中的File对象 \*/  }; |

成员变量及成员函数的含义如下：

|  |  |
| --- | --- |
| 成员变量 | 含义 |
| k\_OpenFileTable | File对象的指针数组，指向系统打开文件表 |

这一个类的定义与Unix V6++的该类的定义完全一致，NOFILES的值为15，表明一个进程最多可以打开15个文件。

### 2.3.4高速缓存结构的数据结构定义

对于Unix中的块设备来说，所有的读写操作都需要经由内核中的一块名为高速缓冲存储区的物理内存区域来实现。通过高速缓存结构，可以提高CPU效率和并行程度，减少对存储设备的读写次数。

Unix为每个缓存都设置了一个缓存控制块——Buf类，它包含了对缓存的控制信息以及IO请求信息。

而高速缓存机制则是由BufferManager类完成，其中包含了缓存控制块数组、自由队列首部等。

（1）Buf类

Buf类是缓存控制块类，起到对缓存的控制信息以及IO请求信息的控制。

类的定义如下：

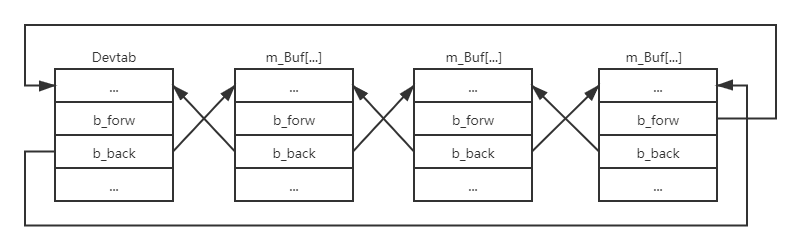
|  |
| --- |
| class Buf  {  public:  /\* b\_flags中标志位 \*/  enum BufFlag  {  B\_WRITE = 0x1,  B\_READ = 0x2,  B\_DONE = 0x4,  B\_ERROR = 0x8,  B\_BUSY = 0x10,  B\_WANTED = 0x20,  B\_ASYNC = 0x40,  B\_DELWRI = 0x80  };  public:  unsigned int b\_flags;  Buf\* b\_forw;  Buf\* b\_back;  Buf\* av\_forw;  Buf\* av\_back;  int b\_wcount;  char\* b\_addr;  int b\_blkno;  }; |

成员变量及成员函数的含义如下：

|  |  |
| --- | --- |
| 成员变量 | 含义 |
| b\_flags | 缓存控制块标志位 |
| b\_forw | 缓存控制块队列勾连指针 |
| b\_back | 缓存控制块队列勾连指针 |
| av\_forw | 缓存控制块队列勾连指针 |
| av\_back | 缓存控制块队列勾连指针 |
| b\_wcount | 需传送的字节数 |
| b\_addr | 指向该缓存控制块所管理的缓冲区的首地址 |
| b\_blkno | 磁盘逻辑块号 |

与Unix V6++相比，此处省略了用于标志设备和IO出错时的成员变量。

类定义中的四个指针，每两个为一组，使得高速缓冲控制块构成如下的结构：



（2）BufferManager类

高速缓存管理类主要负责对内核中的高速缓存块进行管理，其中设置了高速缓存块的大小、高速缓存块的个数，给出了自由队列的队首、设备队列的队首等。

类的定义如下：

|  |
| --- |
| class BufferManager  {  public:  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 \*/  public:  BufferManager();  ~BufferManager();  private:  Buf bFreeList;  Buf m\_Buf[NBUF];  Buf bDevtab;  char Buffer[NBUF][BUFFER\_SIZE];  }; |

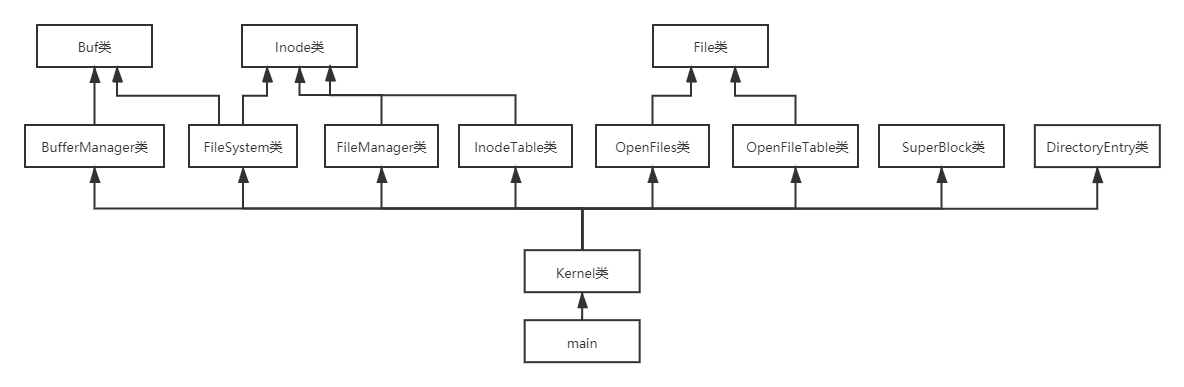
成员变量及成员函数的含义如下：

|  |  |
| --- | --- |
| 成员变量 | 含义 |
| bFreeList | 自由缓存队列控制块，自由队列队头 |
| m\_Buf | 缓存控制块数组 |
| bDevtab | 设备控制块，设备队列队头 |
| Buffer | 高速缓冲区数组 |

与Unix V6++相比，此处省略了用于进程图像在内存和盘交换区之间传递的成员变量。

## 2.4 模块间的调用关系

各模块之间的调用关系如下：

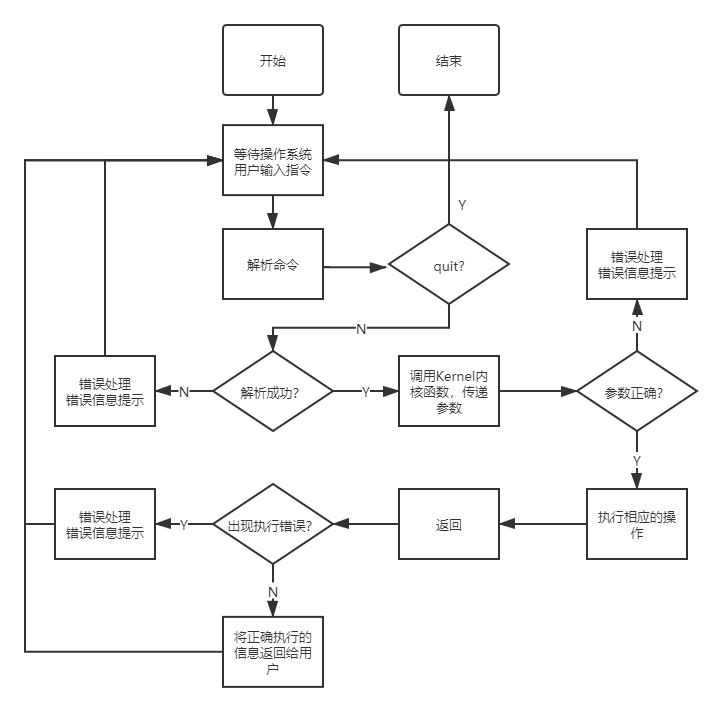


其中，Kernel类是本二级文件系统的核心类(内核)，它直接或间接的调用了前面所述的所有数据结构，而整个文件系统就是该类的一个实例化，由main函数来进行这一实例化。

## 2.5 主程序流程与算法说明

主程序的总体流程如下：main函数等待用户输入，用户输入后，对其命令进行简单解析，若出现无法解析的命令或者命令的参数错误，则进行错误处理和错误提示；若成功解析命令，则将命令及参数提供给实例化的Kernel类，由 Kernel类调用其中的类及其成员函数以实现文件系统的具体功能。

主程序的流程图如下：



# **3 详细设计**

## 3.1 重点函数与重点变量

（1）INode类

|  |  |
| --- | --- |
| 重点变量 | 说明 |
| i\_flag | 状态的标志位 |
| i\_mode | 文件工作方式信息 |
| i\_count | 引用计数 |
| i\_nlink | 文件联结计数 |
| i\_number | 外存inode区中的编号 |
| i\_size | 文件大小 |
| i\_addr | 逻辑块号和物理块号转换的基本索引表 |
| 重点函数 | 描述 |
| 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 |
| WriteI | 根据Inode对象中的物理磁盘块索引表，将数据写入文件 |
| Bmap | 将文件的逻辑块号转换成对应的物理盘块号 |
| IUpdate | 更新外存Inode |
| ITrunc | 释放Inode对应文件占用的磁盘块 |
| Clean | 清空Inode对象中的数据 |
| ICopy | 将包含外存Inode字符块中信息拷贝到内存Inode中 |

（2）DiskINode类

|  |  |
| --- | --- |
| 重点变量 | 说明 |
| d\_mode | 状态的标志位 |
| d\_nlink | 文件联结计数，即该文件在目录树中不同路径名的数量 |
| d\_size | 文件大小 |
| d\_addr | 用于文件逻辑块号和物理块号转换的基本索引表 |

（3）SuperBlock类

|  |  |
| --- | --- |
| 重点变量 | 说明 |
| s\_isize | 外存Inode区占用的盘块数 |
| s\_fsize | 盘块总数 |
| s\_nfree | 直接管理的空闲盘块数量 |
| s\_free | 直接管理的空闲盘块索引表 |
| s\_ninode | 直接管理的空闲外存Inode数量 |
| s\_inode | 直接管理的空闲外存Inode索引表 |
| s\_fmod | 内存中super block副本被修改标志 |

（4）FileSystem类

|  |  |
| --- | --- |
| 重点函数 | 描述 |
| LoadSuperBlock | 系统初始化时读入SuperBlock |
| Update | 将SuperBlock对象的内存副本更新到存储设备的SuperBlock中去 |
| IAlloc | 分配一个空闲外存INode，用于创建新的文件 |
| IFree | 释放编号为number的外存INode，用于删除文件 |
| Alloc | 分配空闲磁盘块 |
| Free | 释放编号为blkno的磁盘块 |

（5）InodeTable类

|  |  |
| --- | --- |
| 重点变量 | 说明 |
| m\_Inode | 内存Inode数组 |
| 重点函数 | 描述 |
| IGet | 根据外存Inode编号获取对应Inode |
| IPut | 减少该内存Inode的引用计数 |
| UpdateInodeTable | 将被修改过的内存Inode更新到对应外存Inode中 |
| IsLoaded | 检查编号为inumber的外存inode是否有内存拷贝 |
| GetFreeInode | 在内存Inode表中寻找一个空闲的内存Inode |

（6）File类

|  |  |
| --- | --- |
| 重点变量 | 说明 |
| f\_flag | 对打开文件的读、写操作要求 |
| f\_count | 当前引用该文件控制块的进程数量 |
| f\_inode | 指向打开文件的内存Inode指针 |
| f\_offset | 文件读写位置指针 |

（7）OpenFileTable类

|  |  |
| --- | --- |
| 重点变量 | 说明 |
| m\_File | 系统打开文件表 |
| 重点函数 | 描述 |
| FAlloc | 在系统打开文件表中分配一个空闲的File结构 |
| CloseF | 对打开文件控制块File结构的引用计数 |

（8）OpenFiles类

|  |  |
| --- | --- |
| 重点变量 | 说明 |
| k\_OpenFileTable | File对象的指针数组，指向系统打开文件表中的File对象 |
| 重点函数 | 描述 |
| AllocFreeSlot | 在内核打开文件描述符表中分配一个空闲表项 |
| GetF | 找到对应的打开文件控制块File结构 |
| SetF | 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 |

（9）DirectoryEntry类

|  |  |
| --- | --- |
| 重点变量 | 说明 |
| inode | 目录项中Inode编号部分 |
| name | 目录项中路径名部分 |

（10）FileManager类

|  |  |
| --- | --- |
| 重点变量 | 说明 |
| rootDirInode | 根目录内存Inode |
| 重点函数 | 描述 |
| NameI | 文件的路径搜索 |
| Read | 读文件 |
| Write | 写文件 |
| Seek | 改变当前读写指针的位置 |
| Delete | 删除文件 |

（11）Buf

|  |  |
| --- | --- |
| 重点变量 | 说明 |
| b\_flags | 缓存控制块标志位 |
| b\_wcount | 需传送的字节数 |
| b\_addr | 指向该缓存控制块所管理的缓冲区的首地址 |
| b\_blkno | 磁盘逻辑块号 |
| b\_forw，b\_back，  av\_forw，av\_back | 缓存控制块队列勾连指针 |

（12）BufferManager类

|  |  |
| --- | --- |
| 重点变量 | 说明 |
| bFreeList | 自由缓存队列控制块 |
| m\_Buf | 缓存控制块数组 |
| bDevtab | 设备队列控制块 |
| Buffer | 缓冲区数组 |
| 重点函数 | 描述 |
| GetBlk | 申请一块缓存，用于读写字符块blkno |
| Brelse | 释放缓存控制块buf |
| Bread | 读一个磁盘块，blkno为目标磁盘块逻辑块号 |
| Bwrite | 写一个磁盘块 |
| Bdwrite | 延迟写磁盘块 |
| Bawrite | 异步写磁盘块 |
| ClrBuf | 清空缓冲区内容 |
| Bflush | 将dev指定设备队列中延迟写的缓存全部输出到磁盘 |
| NotAvail | 从自由队列中摘下指定的缓存控制块buf |

（13）IOParameter类

|  |  |
| --- | --- |
| 重点变量 | 说明 |
| m\_Base | 当前读、写用户目标区域的首地址 |
| m\_Offset | 当前读、写文件的字节偏移量 |
| m\_Count | 当前还剩余的读、写字节数量 |

（14）Kernel类

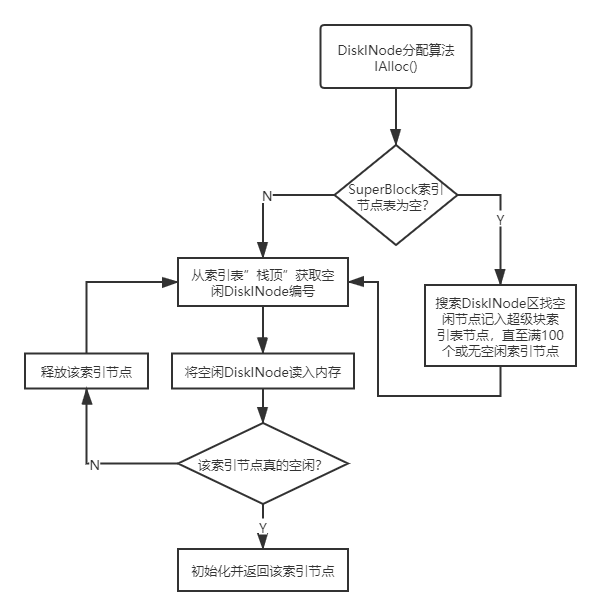
|  |  |
| --- | --- |
| 重点变量 | 说明 |
| instance | 唯一的内核类实例 |
| BufMgr | 内核的高速缓存管理实例 |
| fileSys | 内核的文件系统实例 |
| fileMgr | 内核的文件管理实例 |
| k\_InodeTable | 内核的内存Inode表 |
| s\_openFiles | 系统全局打开文件描述符表 |
| k\_openFiles | 内核的打开文件描述符表 |
| spb | 全局的SuperBlock内存副本 |
| 重点函数 | 描述 |
| initialize | 内核初始化 |
| callInit | 每个函数调用的初始化工作 |
| format | 格式化磁盘 |
| open | 打开文件 |
| create | 新建文件 |
| mkdir | 新建目录 |
| cd | 改变当前工作目录 |
| ls | 显示当前目录下的所有文件 |
| fread | 读一个文件到目标区 |
| fwrite | 根据目标区的字符写一个文件 |
| fseek | 改变读写指针的位置 |
| fdelete | 删除文件 |
| fmount | 将windows文件拷贝到磁盘某目录下 |
| close | 关闭文件 |
| clear | 系统关闭时收尾工作 |

## 3.2 重点功能的算法流程图及代码

### 3.2.1 DiskINode分配算法

DiskINode的分配由FileSystem类中的IAlloc函数来完成，该算法的核心思想是：采用栈式管理，若栈不为空，则直接分配；若栈顶指针为空，则遍历DiskINode区，将找到的空闲节点压入栈中。

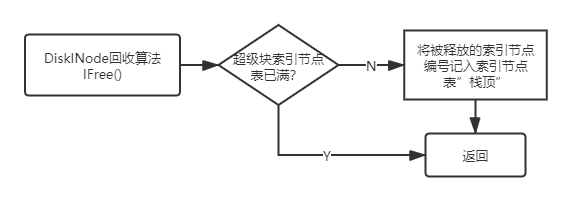
算法的流程图入下：



### 3.2.2 DiskINode回收算法

DiskINode的回收由FileSystem类中的IFree函数来完成，该算法的核心思想是：采用栈式管理，当需要释放一个DiskINode时，若超级块索引节点表中空闲DiskINode的个数小于100，则将该索引节点编号记入“栈顶”；否则直接返回，不做任何的操作。

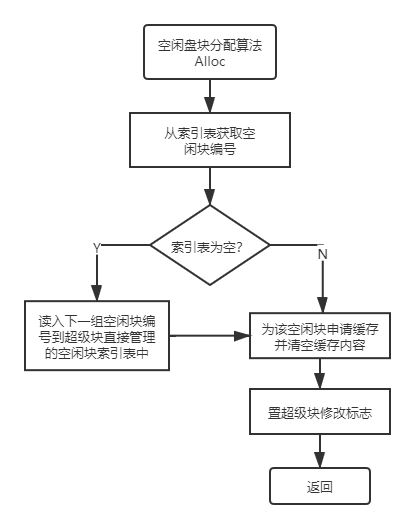
算法的流程图入下：



### 3.2.3 空闲盘块分配算法

空闲盘块的分配由FileSystem类中的Alloc函数来完成，该算法的核心思想是：空闲盘块采用栈式管理，并且采用分组链式的索引方法。每个SuperBlock块最多只能直接管理100个空闲块。SuperBlock块管理第一组空闲块，各组的索引表存放在它们上一组第一个盘块的开头中，当分配空闲盘块的时候，将当前栈顶内容弹出；如果当前栈只剩下一个索引，则换入下一组进行管理。

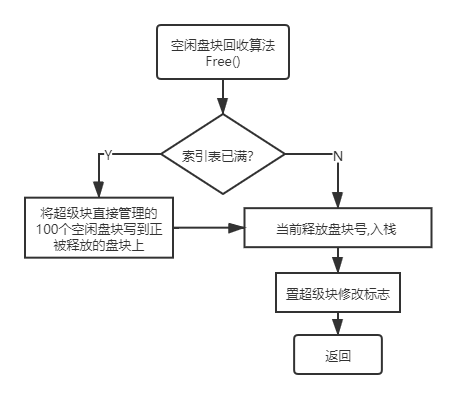
算法的流程图入下：



### 3.2.4 空闲盘块回收算法

空闲盘块的回收由FileSystem类中的Free函数来完成，该算法的核心思想是：空闲盘块采用栈式管理，并且采用分组链式的索引方法。在进行回收时，若当前的栈不满，则直接弹入栈中；若当前栈已满，则新增一组空闲块，并加在上一级结尾。

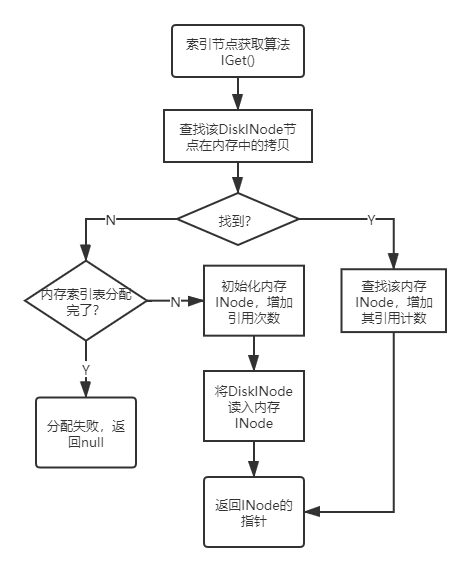
算法的流程图入下：



### 3.2.5 索引节点获取算法

索引节点的获取由InodeTable类中的IGet函数来完成，该算法的核心思想是：先直接查找该DiskINode是否有对应的内存INode，若有，则返回该INode；若没有，则申请分配一个内存INode。

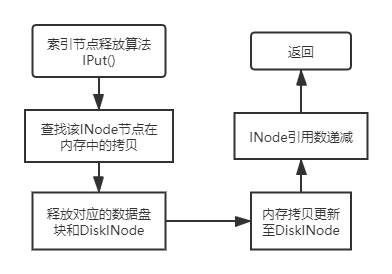
算法的流程图入下：



### 3.2.6 索引节点释放算法

索引节点的获取由InodeTable类中的IPut函数来完成，该算法的核心思想是：先直接释放对应的DiskINode，然后将INode更新拷贝至DiskINode，最后减少该INode的引用计数。

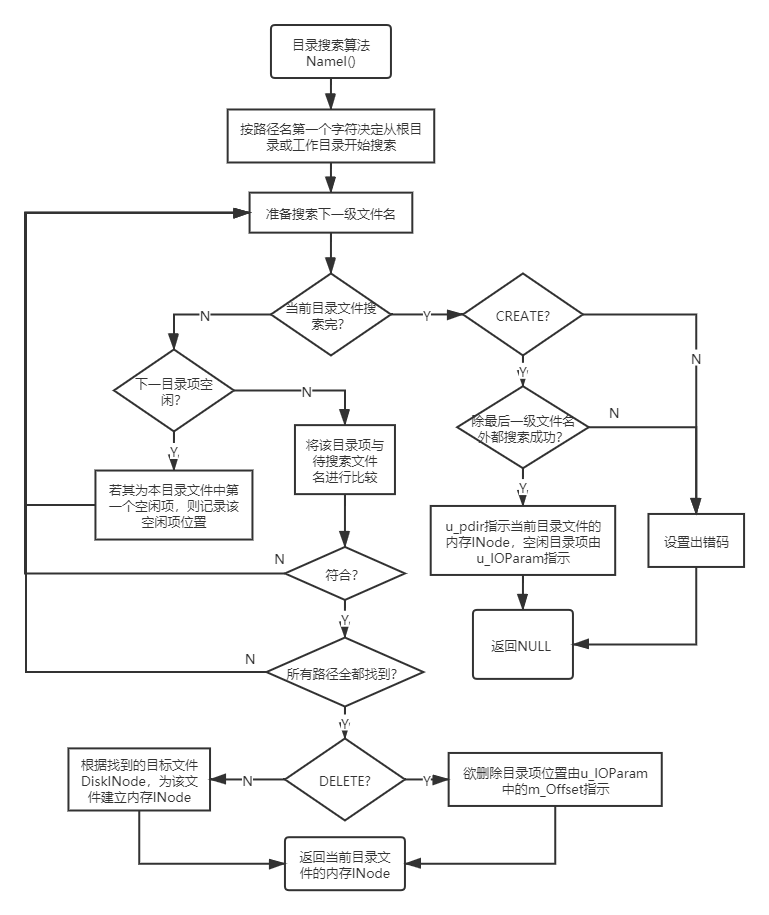
算法的流程图入下：



### 3.2.7 目录搜索算法

目录搜索由FileManager类中的NameI函数来完成，它负责将用户给出的路径名字符串转化为对应文件的索引节点。

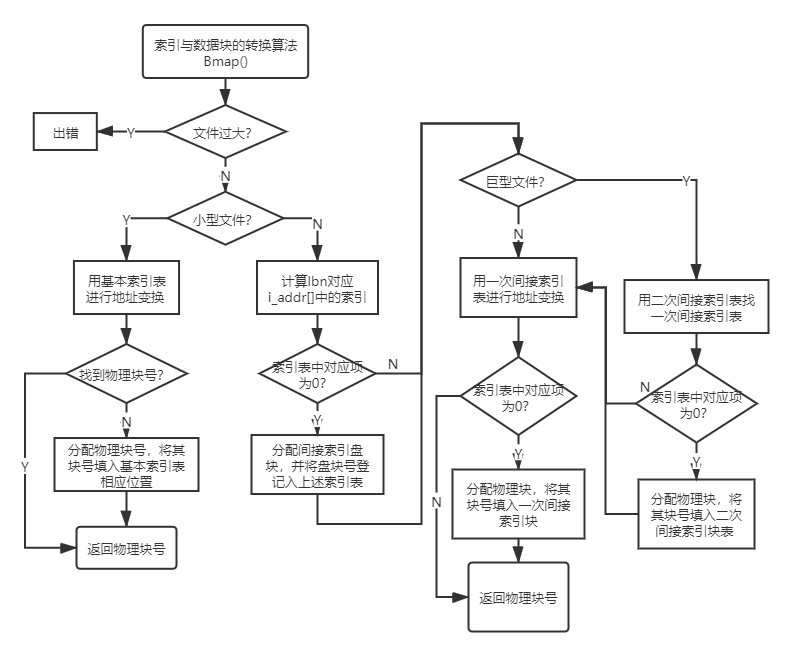
算法的流程图入下：



### 3.2.8 文件索引与数据块的转换算法

目录搜索由Inode类中的Bmap函数来完成，它负责将文件索引与对应的数据块进行一一对应，从而通过文件索引找到该文件的所有数据块。它的核心思想是：通过判断文件大小，如果是小文件，直接在索引表得到数据块索引。如果是中型文件，首先根据索引表得到一级间接索引表，然后得到数据块索引。如果是大型文件，首先根据索引表得到二级间接索引表，再通过二级间接索引表得到一级间接索引表，从而得到数据块索引。

算法的流程图入下：



### 3.2.9 高速缓存的相关算法

在本文件系统中，使用了双向循环链表模拟缓存队列，实现缓存的分配和淘汰。

（1）高速缓存淘汰算法

本文件系统采用的缓存淘汰算法是LRU算法：每当一个缓存被释放，就将它送入自由缓存队列的尾部，使得它能够尽可能保持原样较长时间而不被改动；而在分配缓存时，当一个缓存在自由队列内移动时，只要有按原状使用它的需求，就立即将他从自由队列中抽出，否则就从自由缓存队列首部取一个。这样即实现了LRU算法。

（2）高速缓存分配算法

首先从缓存队列中寻找是否有目标缓存块：若存在则分离该缓存节点，并返回该节点；若不存在则说明缓存队列中没有目标缓存块，这时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，清空标志并返回该缓存块。

## 3.3 函数调用关系

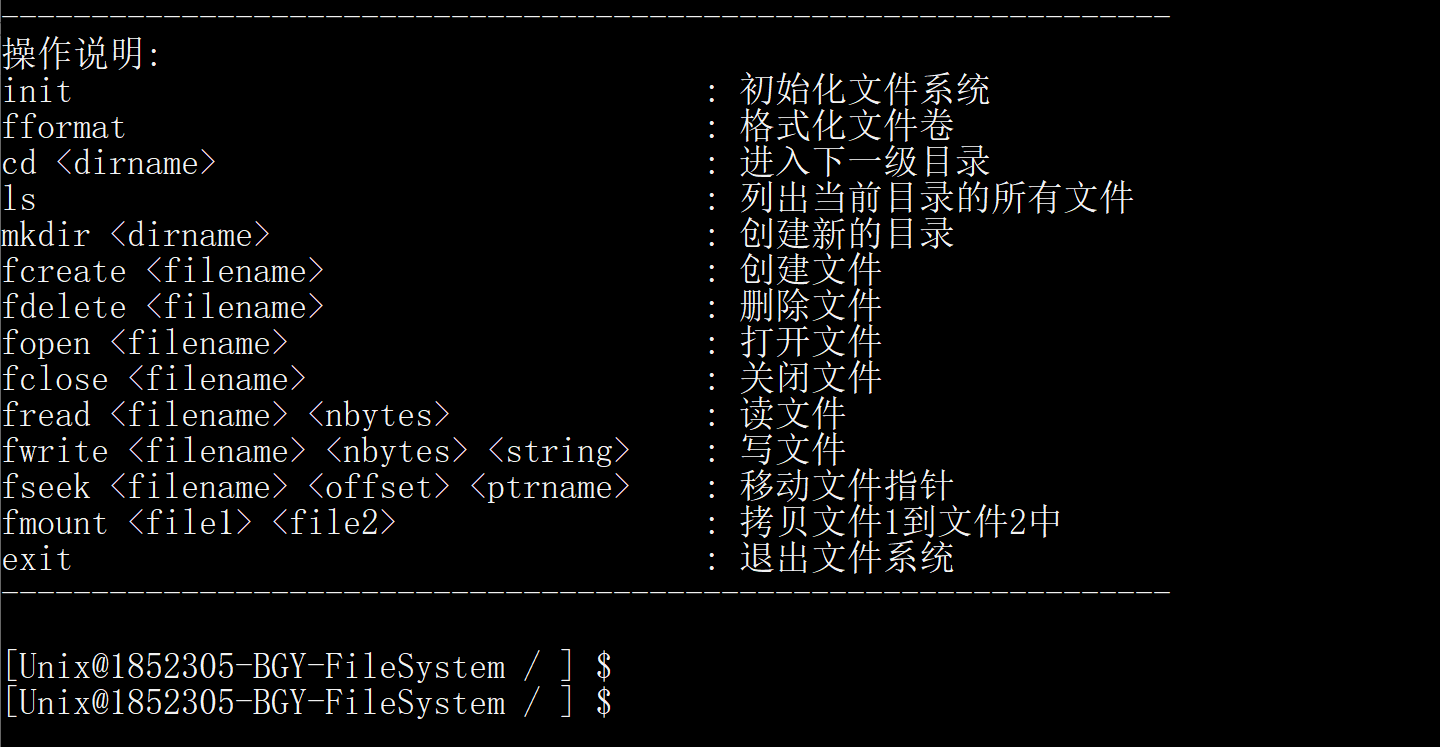
由于整个文件系统的函数调用关系过于复杂，在此仅列出用户接口函数的直接调用的类成员函数，如下：

|  |  |
| --- | --- |
| 用户接口函数 | 调用的类成员函数 |
| fformat(Kernel::format) | Utility::copy |
| cd(Kernel::cd) | Kernel::callInit  FileManager::ChDir |
| ls(Kernel::ls) | BufferManager::Brelse  BufferManager::Bread  Utility::copy |
| mkdir(Kernel::mkdir) | Kernel::callInit  FileManager::Creat  FileSystem::Update |
| fcreate(Kernel::create) | Kernel::callInit  FileManager::Creat  FileSystem::Update |
| fdelete(Kernel::fdelete) | Kernel::callInit  FileManager::Delete |
| fopen(Kernel::open) | Kernel::callInit  FileManager::Open |
| fclose(Kernel::close) | Kernel::callInit  FileManager::Close |
| fread(Kernel::fread) | Kernel::callInit  FileManager::Read |
| fwrite(Kernel::fwrite) | Kernel::callInit  FileManager::Write |
| fseek(Kernel::fseek) | Kernel::callInit  FileManager::Seek |
| fmount(Kernel::fmount) | 无 |
| exit(无) | 无 |
| init(无) | fformat(Kernel::format)  mkdir(Kernel::mkdir)  fcreate(Kernel::create)  cd(Kernel::cd)  fclose(Kernel::close)  fmount(Kernel::fmount) |

# **4 运行结果分析**

## 4.1 文件系统初始化

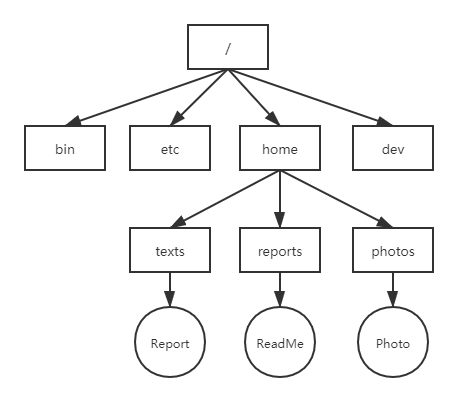
打开文件系统将看到如下界面：



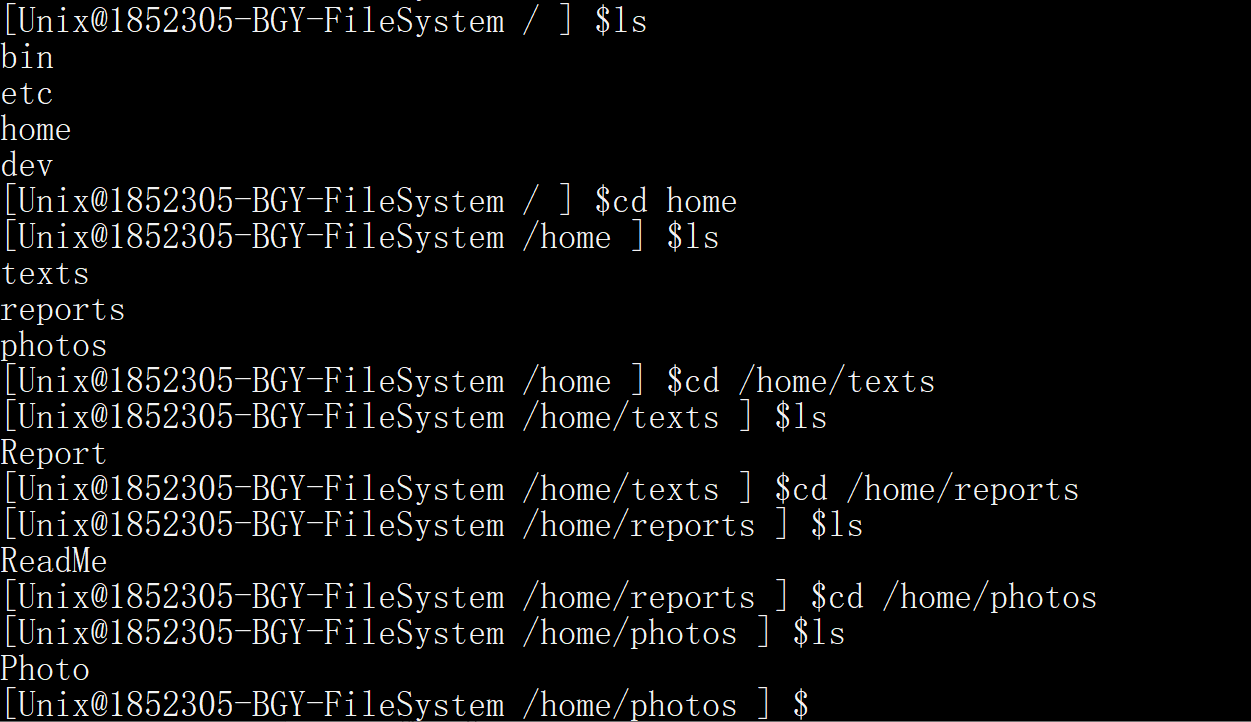
执行命令init，初始化文件系统：



得到以下结构：



查看该文件结构是否符合要求：

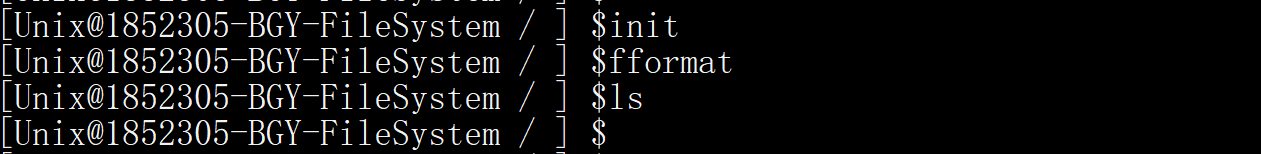


可以看到，所创建的目录和文件结构与要求的一致。

## 4.2 部分命令的测试

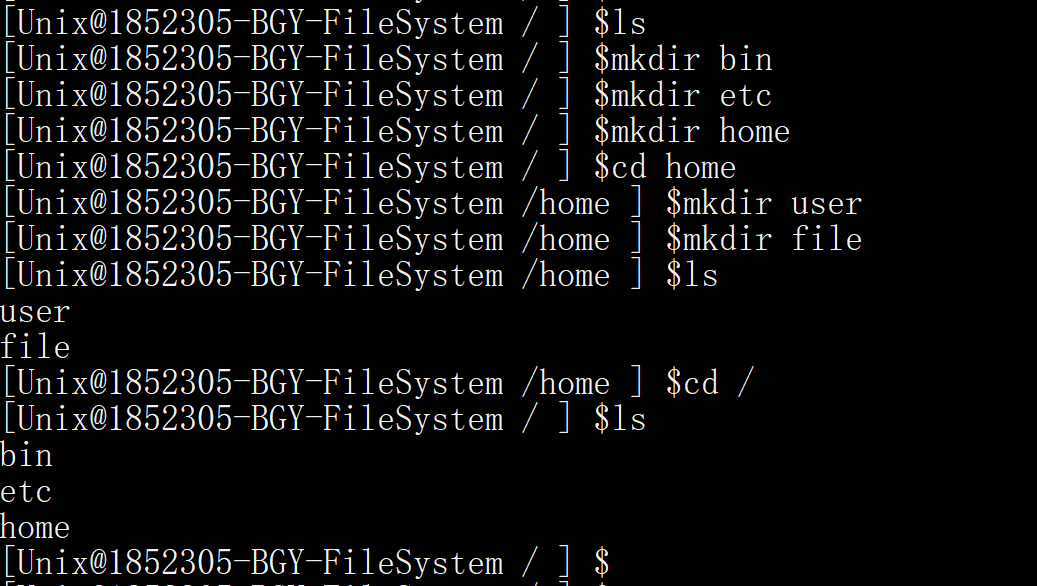
（1）测试一：fformat

如下图，先init初始化文件系统，再fformat格式化文件系统，使用ls列举出根目录下的目录结构，发现并没有任何目录，这表明格式化成功。



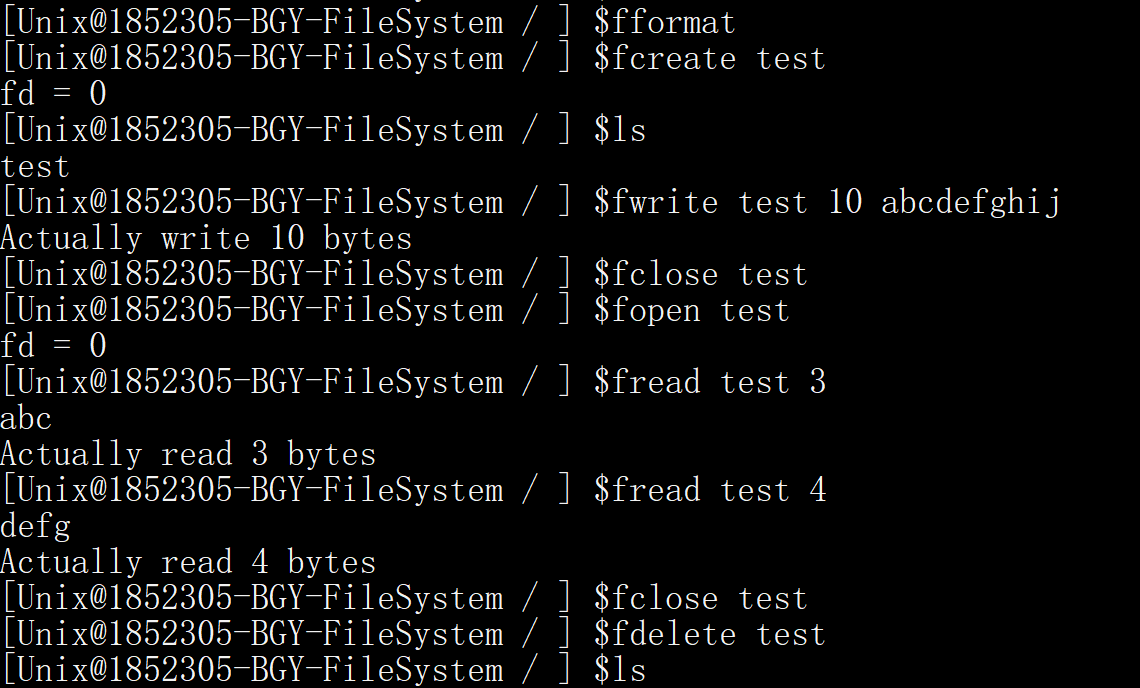
（2）测试二：mkdir, ls, cd

如下图，在格式化后的根目录下创建bin,etc,home三个文件目录，进入home目录创建user,file目录。用ls列举，发现user,file成功创建，返回根目录，用ls列举，发现bin,etc,home成功创建。



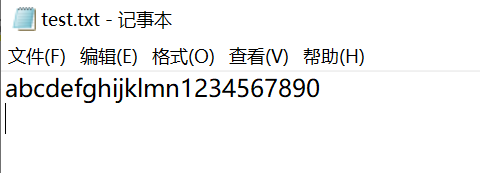
（3）测试三：fcreate,fdelete,fopen,fclose,fwrite,fread

如下图，格式化文件卷后创建test文件，向其中写入10个字符，分两次读取3个和4个字符，最后关闭并删除文件，这些操作都得到了正确的结果。



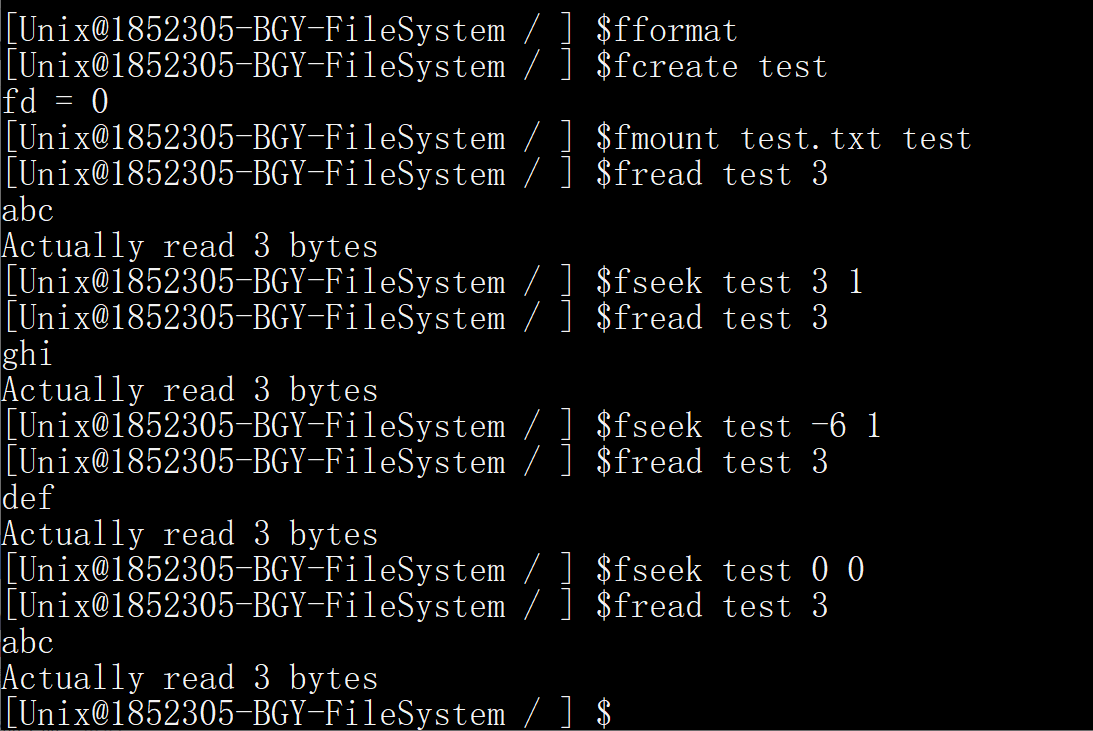
（4）测试四：fmount,fseek

在test.txt文件中输入字符串如下：



将test.txt与文件系统可执行文件放在同一目录下（若不这样，在fmount的第一个参数中请使用text.txt文件的绝对路径）。

如下图，进行fmount和fseek的测试：



可以看到，fmount将test.txt中的字符串读入到test中。直接读3个字符，得到abc；将文件指针向后移动3个字符，读三个字符为ghi，又把文件指针向前移动6个字符，读三个字符为def；最后将文件指针设置到0，读三个字符为abc。

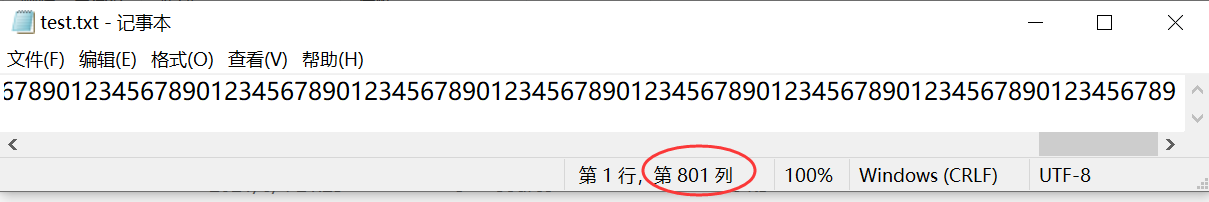
## 4.3 读写与文件指针综合测试及分析

### **4.3.1 测试要求**

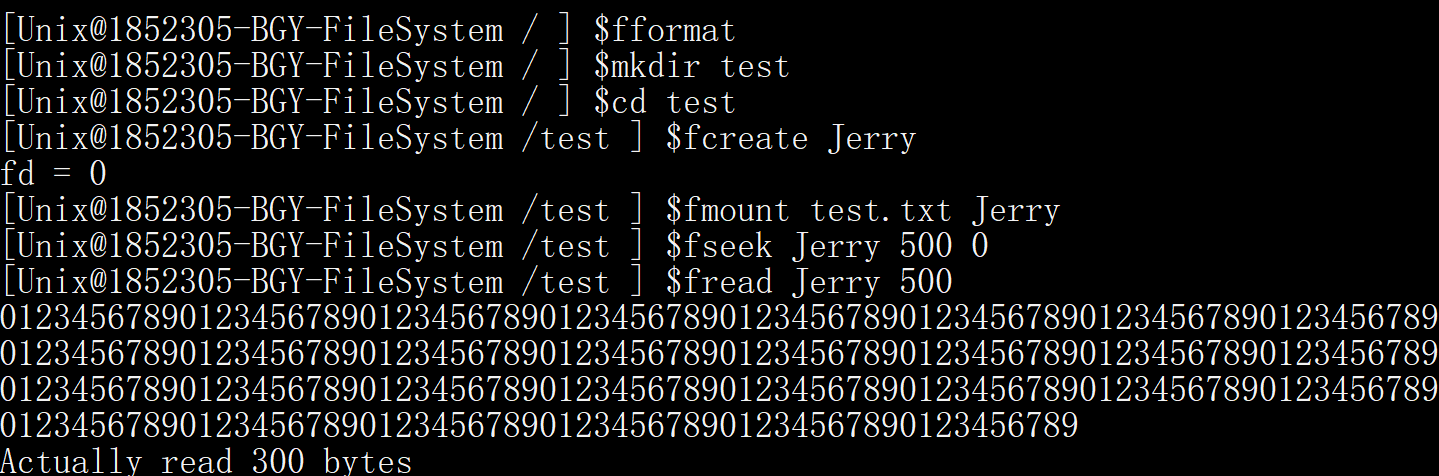
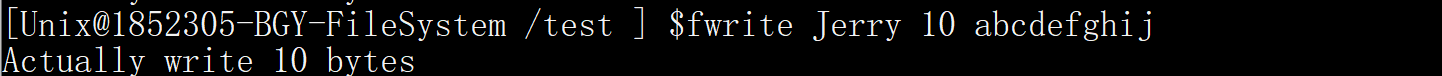
* 新建文件/test/Jerry，打开该文件，任意写入800个字节；
* 将文件读写指针定位到第500字节，读出500个字节到字符串abc。
* 将abc写回文件。

### **4.3.2 测试截图**

在test.txt中写入80遍“0123456789”，如下：



进行测试如下：



可以看到结果和预期一致。

### **4.3.2 测试分析**

**步骤一：新建文件/test/Jerry，打开该文件，任意写入 800 个字节**

**分析：**

该写操作将分两次盘块写操作完成

**第一次：**

1. 通过当前文件读写指针的位置0计算得到需要写的逻辑块号为0，块内偏移地址为0本次需写入的字节数为512

2. 通过i\_addr查询Jerry文件0号逻辑块所在的物理盘块号0，即：一个新块，所以首先申请一个新的盘块（干净的盘块），假设分配的盘块号为100，将该盘块号登记入Jerry文件的i\_addr数组；

3. 第一次是写完整的一块，因此不需要将100号盘块读入；

4. 将前512个字节写入该缓存的0到511字节；

5. 修改读写指针为512，因为刚好写满一块，所以执行异步写操作将缓存内容写入磁盘；

**第二次：**

1. 通过当前文件读写指针的位置512计算得到需写的逻辑块号为1，块内偏移地址为0，本次需写入的字节数为288；

2. 通过i\_addr查询Jerry文件1号逻辑块所在的物理盘块号0，即：一个新块，所以首先申请一个新的盘块（干净的盘块），假设分配的盘块号为101，将该盘块号登记入Jerry文件的i\_addr数组；

3. 因为288 < 512，即不是写完整的一块，所以需将101号盘块读入；

4. 对该物理盘块申请一个缓存

5. 将后288个字节写入该缓存的前288个字节；

6. 修改读写指针为800，因为一块未写满，所以将该缓存添加B\_DELWRI标志后直接释放；

7. 修改文件长度为800。

**步骤二：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**

**分析：**

该读操作将分两次盘块读操作完成。

（1）seek(fd, 500, 0);

文件的读写指针被定位在第500个字节

（2）int count = read (fd，abc, 500);

该读操作将分两次盘块读操作完成。

**第一次：**

1. 通过当前文件读写指针的位置500计算得到需读取的逻辑块号为0；

2. 通过i\_addr查询Jerry文件0号逻辑块所在物理盘块号为100；

3. 对该物理盘块申请一个缓存；

4. 将该缓冲区中500到511字节的内容读出；

5. 释放缓存；

6. 修改文件读写指针为512。

**第二次：**

1. 通过当前文件读写指针的位置512计算得到需读取的逻辑块号为1；

2. 通过i\_addr查询Jerry文件1号逻辑块所在的物理盘块101；

3. 对该物理盘块申请一个缓存，发现101号盘块的缓存在设备队列中，且有B\_DONE标志，即该缓存中的内容可以直接使用；

4. 因为文件到该缓存中的第288字节就结束了，所以将该缓冲区中0到288字节的内容读出；

5. 释放缓存；

6. 修改文件读写指针为800。

**步骤三：读出字符串写回文件。**

**分析：**

该写操作将分两次盘块写操作完成

**第一次：**

1. 通过当前文件读写指针的位置800计算得到需要写的逻辑块号为1，块内偏移地址为288本次需写入的字节数为224；

2. 通过i\_addr查询Jerry文件1号逻辑块所在的物理盘块号101；

3. 因为224 < 512，即不是写完整的一块，所以需将101号盘块读入；

4. 对该物理盘块申请一个缓存；

5. 将前224个字节写入该缓存的288到511字节；

6. 修改读写指针为1024，因为刚好写满一块，所以执行异步写操作将缓存内容写入磁盘；

**第二次：**

1. 通过当前文件读写指针的位置1024计算得到需写的逻辑块号为2，块内偏移地址为0，本次需写入的字节数为76；

2. 通过i\_addr查询Jerry文件2号逻辑块所在的物理盘块号0，即：一个新块，所以首先申请一个新的盘块（干净的盘块），假设分配的盘块号为102，将该盘块号登记入Jerry文件的i\_addr数组；

3. 因为76 < 512，即不是写完整的一块，所以需将102号盘块读入；

4. 对该物理盘块申请一个缓存；

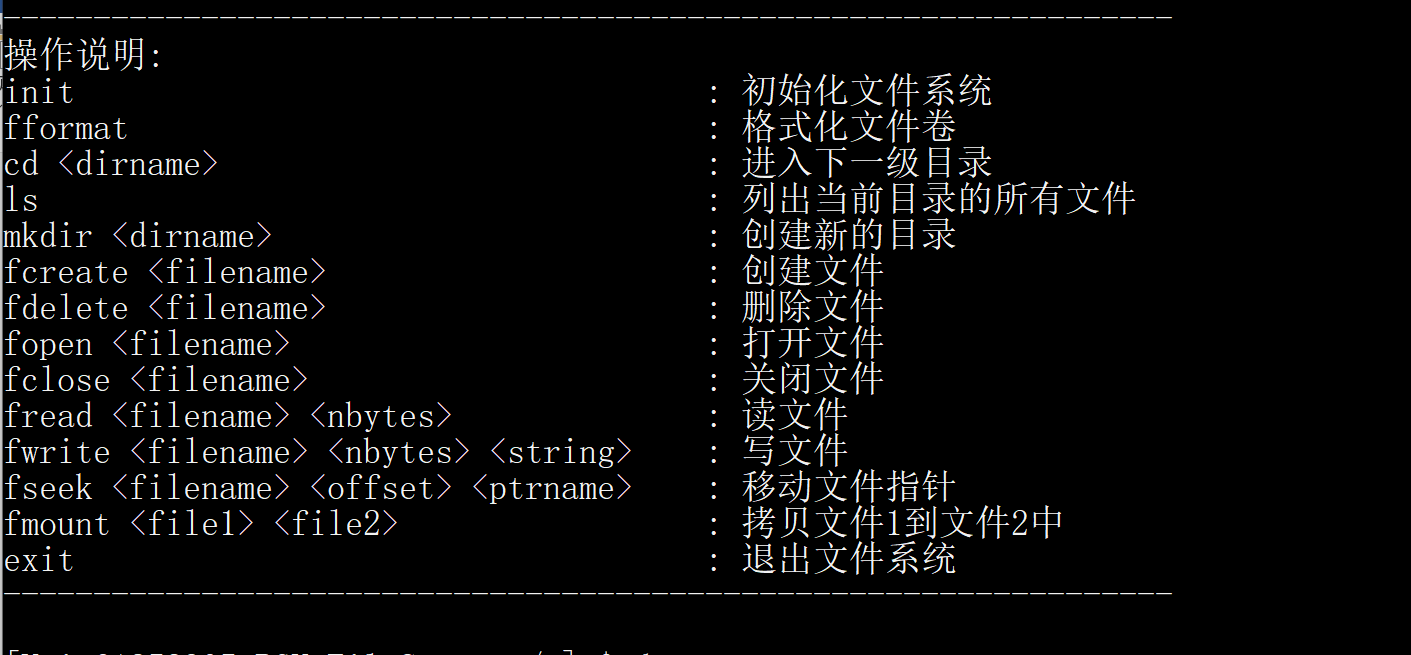
5. 将后76个字节写入该缓存的前76个字节；

6. 修改读写指针为1100，因为一块未写满，所以将该缓存添加B\_DELWRI标志后直接释放；

7. 修改文件长度为1100。

# **5 用户使用说明**

双击FileSystem.exe可执行文件即可运行该文件系统，本文件系统的界面与linux系统命令行较为相似，操作简单，容易上手使用。启动可执行文件即可看到下列的界面：



在文件系统的最上方给出了用户操作说明，可以根据该操作说明使用相应的命令进行测试和使用，可使用的命令都包含于《操作说明》之中。

**几个注意点如下：**

* 1. 在对文件系统进行操作之前，一般需要执行fformat初始化文件卷。使用结束后，使用命令exit退出文件系统；
  2. 指令cd的参数可以是当前目录下的目录文件，或者是绝对路径，无法使用“cd ..”回到上一级文件目录；
  3. 可以使用PgUp和PgDn向前或向后翻看执行过的命令；
  4. fcreate的同时也打开了文件，相当于fopen了一次，打开文件后在不使用时记得关闭文件；
  5. 关于feesk参数，fseek的第三个参数是seek方式，“0”表示设置绝对位置，此时第二个参数必须为非负数；“1”表示从当前位置移动移动，此时第二个参数可以为合理的正负数。

# **6 实验总结**

实验总结包含四个部分：收获与感悟、遇到的问题和解决方案、后续工作以及对课程的认识。

## 6.1 收获与感悟

本次课设在理论课基础上，需要实现类Unix的二级文件系统。

在实现过程中，我基本再一次梳理了一遍课本上设备管理和文件管理这两部分的所有知识点，包括DiskINode和INode的分配与回收管理，数据块的分配与回收管理，高速缓存的实现方法，以及和文件系统相关的十几个数据结构类。通过课设，我更加深入理解了Unix V6++的文件系统的组织结构和构成方式。

实验一开始时，由于我只是对课程的理论知识稍微熟悉，对于下发的Unix V6++的源代码的熟悉程度不够，在加上从来没有这样着手做过完整的内核开发的项目，我觉得十分无从下手。

在经过对各项操作系统技术的详细了解，我逐步分析并规划出了满足所有需求的文件系统。后续又完成了代码撰写、调试等工作。这个过程中，我根据课程设计要求，查找了不少的资料，也和同学、室友进行讨论。完成这个课设的过程很漫长，学习的内容也比较困难，但我还是没有放弃，最终也完成课设项目。

当然，在这一过程中，我也遇到了很多细节问题与设计思路问题，比如说自己代码习惯不是很好导致编程混乱，以及在参考源码的过程中发现源码中很多东西并不需要在这次实验中实现，所以在参考源码后进行了一定的修改与删减等。同时对于本次课设与上学期理论课程的认知，课本上学的文件系统其实还不太透彻，做课设的时候也处处碰壁，需要反复翻阅教材、源代码和其他资料。

从结果上说，本次课设收获满满，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，还是分析错误调试程序的能力，我都有较大的提升。

## 6.2 遇到的问题和解决方案

本次课设仿照Unix V6++完成二级文件系统的设计，在设计过程中，遇到了不少问题。其中之一就是需要从Unix V6++中选取需要的部分，这里所谓的“需要的部分”不仅仅指需要的类，更是指需要的类成员变量和类成员函数。在一开始时，我所编写的二级文件系统中包含了很多用不到的类成员变量和类成员函数，这无疑增大了代码的冗余量与可读性。解决方法是认真分析文件系统的需求，从中选取用得到的变量和函数。

## 6.3 后续工作

本次课设完成了基本的带有高速缓存的二级文件系统，仍有许多的可改进之处。

（1）增加文件系统支持的命令

该文件系统仅仅实现了《操作系统课设要求》中提及的fformat, ls, cd, fcreate, fdelete, fopen, fclose, fread, fwrite, fseek, fmount和exit几个基本命令，还有很大的扩展空间。例如可以加入tree, df, cmp, find等许多其他命令。

（2）多用户

该文件系统相当于只有一个用户，所有文件对于该用户而言都是直接可见的。但在实际使用中，多用户是很有必要的，文件在不同用户间的共有和私有也是极为重要的，因此，可以考虑将该文件系统作出进一步修改，实现多用户。

（3）多进程基础

在理论课中，我们学习到操作系统不同的进程打开文件是相互独立的，不同的进程拥有自己的进程文件打开表。多个进程是可以打开同一个文件的，因此为了适应这种需要，首先需要更改上述命令的接口参数，使之能够为多线程提供可能性。当前无论文件的创建、删除、打开或是关闭，参数都是文件名本身，应该修改了和进程编号与文件句柄有关。

（4）图形化界面

本文件系统使用cmd界面。cmd界面虽然简洁方便，但不够直观。因此，可以考虑制作简单的图形化界面。

## 6.4 对课设的认识

在逐步完成操作系统课设——二级文件系统的过程中，我反反复复翻看教材上关于设备管理和文件管理这两章的内容，也多次查看Unix V6++源代码，可以说，对于Unix文件管理的理论知识我能够娓娓道来，对于Unix V6++文件系统部分的代码逻辑和结构烂熟于心。通过本次课程设计，我对操作系统特别是文件系统部分有了很深入的了解，对于C++程序设计语言的使用技巧有了更多的认识，对于内核开发有了一点点的初步经验。

操作系统的课程设计无疑是一门很有意义的课程，它难度适中，在拥有一定难度和挑战性的同时也有很多的理论和技术支持。它是理论与实践结合的很好体现，需要我们将学习过的理论知识付诸于实践，落地到实际的应用中去。

# **7 参考文献**

[1]操作系统原理(讲义)[M]. 同济大学计算机系. 2017(09): 293-335

[2]Unix V6++ 源代码. 同济大学计算机系.

[3]尤晋元. Unix操作系统教程[M]. 电子科技大学出版社，1985年6月