**课 程 设 计 报 告**

设计题目：多用户、多级目录结构文件系统的设计与实现

班 级：计算机1606

组长学号：20164383

组长姓名：刘学渊

指导教师：王大玲

设计时间：2019年7月

**任务分工**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **姓名** | **学号** | **任务比例** | **实现功能** | **命令格式** |
| 刘学渊 | 20164383 | 50% | 用户登录文件系统 | login |
| 用户退出文件系统 | logout |
| \*修改用户密码 | passwd |
| \*删除文件夹 | rm -c -f [^- ]+ |
| \*复制文件 | cp [^- ]+ [^- ]+ |
| \*复制文件夹 | cp -f [^- ]+ [^- ]+ |
| \*移动文件 | mv [^- ]+ [^- ]+ |
| \*移动文件夹 | mv [^- ]+ [^- ]+ |
| 改变当前目录 | cd |
| \*将文件/文件夹移入回收站 | rm (-f) [^- ]+ |
| \*还原回收站中的文件/文件夹 | rec [^- ]+ |
| \*显示回收站内容 | rec -show |
| \*清空回收站 | rec -clr |
| \*创建全局数据库优化查询 | index [^- ]+/sql |
| \*连接数据库 | - |
| \*创建数据表 | - |
| \*关闭数据库连接 | - |
| \*向数据库批量导入数据（优化） | - |
| \*解析执行用户输入SQL语句 | sql |
| \*创建索引 | index [^- ]+ |
| \*清除数据表中数据 | - |
| \*全盘搜索（普通搜索） | find -a [^- ]+ |
| \*全盘搜索（优化） | find -a [^- ]+ |
| \*在指定路径下搜索 | find [^- ]+ [^- ]+ |
| \*计算文件/文件夹大小 | - |
| \*授予用户文件权限 | chomd -a [^- ]+ [\d]+ [0-7] |
| \*收回用户文件权限 | chomd -r [^- ]+ [\d]+ [0-7] |
| \*修改用户文件权限 | chomd -r [^- ]+ [\d]+ [0-7] |
| \*对读写访问添加权限检查和限制 | - |
| 戚子强 | 20164625 | 50% | 数据结构设计与实现 | - |
| 磁盘分区设计与实现 | - |
| 文件系统启动 | - |
| 文件系统格式化 | format |
| 读写各磁盘分区 | - |
| 分配回收i节点 | - |
| 多级目录实现 | - |
| 文件读取 | cat [^- ]+ |
| 文件写入 | write [^- ]+ |
| 文件创建 | touch [^- ]+ |
| 文件打开/关闭 | open/close [^- ]+ |
| 列出文件夹目录 | ll/ls [^- ]+ |
| 创建/改变目录 | touch -f [^- ]+ / |
| 根据ID/路径寻找i节点 | - |
| 清除磁盘块 | - |
| 文件夹/文件重命名 | chomd [^- ]+ -name [^ ~]+ |
| 帮助 | help [^- ]+ |

# 摘 要

本文对操作系统课程设计中本组设计的文件系统进行了详细地阐述和分析，从总体设计、模块设计、数据结构与算法设计等方面展开论述，完整展示了一个多用户、多级目录文件系统的设计过程，对操作系统理论知识的学习和掌握具有重要意义。

本系统以EXT2文件系统为蓝本，对文件系统数据结构和执行流程进行了二次设计，力求在现有计算机硬件配置的基础上实现一个高效率的文件系统。为此，我们采用“以空间换时间”的思想，在很多地方对文件系统进行了优化，例如我们对文件i结点的操作均在内存中进行，减少磁盘I/O次数，从而提高执行速度；另外，我们利用全局数据库对文件的全局搜索进行优化，通过建立索引，极大地提高了文件查找的效率……除了实现课程要求的基本功能以外，我们还添加了大量的扩展内容，在底层构建的基础上进行二次开发，丰富了文件系统的功能，可以说本系统基本涵盖了Linux文件系统的全部指令和功能。

综上，本系统以其强大的功能和深入底层的优化取得了预期的目标，是一个令人满意的课程设计成果，在实践和学习之外具备一定的实用性。

**关键词：**文件系统；i结点；优化；搜索；权限

**目录**

[摘 要 3](#_Toc13755759)

[1. 概述 6](#_Toc13755760)

[2. 课程设计任务及要求 7](#_Toc13755761)

[2.1 设计任务 7](#_Toc13755762)

[2.2 设计要求 7](#_Toc13755763)

[3. 算法及数据结构 8](#_Toc13755764)

[3.1 算法的总体思想（流程） 8](#_Toc13755765)

[3.2 用户文件权限模块 8](#_Toc13755766)

[3.2.1 功能 8](#_Toc13755767)

[3.2.2 数据结构 8](#_Toc13755768)

[3.2.3 算法 9](#_Toc13755769)

[3.3 登入登出模块 10](#_Toc13755770)

[3.3.1 功能 10](#_Toc13755771)

[3.3.2 数据结构 11](#_Toc13755772)

[3.3.3 算法 12](#_Toc13755773)

[3.4 删除模块 12](#_Toc13755774)

[3.4.1 功能 12](#_Toc13755775)

[3.4.2 数据结构 13](#_Toc13755776)

[3.4.3 算法 13](#_Toc13755777)

[3.5 复制模块 15](#_Toc13755778)

[3.5.1 功能 15](#_Toc13755779)

[3.5.2 数据结构 15](#_Toc13755780)

[3.5.3 算法 15](#_Toc13755781)

[3.6 移动模块 16](#_Toc13755782)

[3.6.1 功能 16](#_Toc13755783)

[3.6.2 数据结构 16](#_Toc13755784)

[3.6.3 算法 16](#_Toc13755785)

[3.7 搜索模块 17](#_Toc13755786)

[3.7.1 功能 17](#_Toc13755787)

[3.7.2 数据结构 17](#_Toc13755788)

[3.7.3 算法 17](#_Toc13755789)

[3.8 数据库模块 18](#_Toc13755790)

[3.8.1 功能 18](#_Toc13755791)

[3.8.2 数据结构 19](#_Toc13755792)

[3.8.3 算法 19](#_Toc13755793)

[3.9 磁盘分区模块 20](#_Toc13755794)

[3.10 数据结构模块 21](#_Toc13755795)

[3.11 文件(夹)创建模块 23](#_Toc13755796)

[3.11.1 功能 23](#_Toc13755797)

[3.11.2 数据结构 24](#_Toc13755798)

[3.11.3 算法 24](#_Toc13755799)

[3.12 文件的打开/关闭/读/写/重命名模块 24](#_Toc13755800)

[3.12.1 功能 24](#_Toc13755801)

[3.12.2 数据结构 24](#_Toc13755802)

[3.12.3 算法 24](#_Toc13755803)

[3.13 安装系统和格式化模块 25](#_Toc13755804)

[3.13.1 功能 25](#_Toc13755805)

[3.13.2 数据结构 25](#_Toc13755806)

[3.13.3 算法 25](#_Toc13755807)

[3.9 底层服务模块 26](#_Toc13755808)

[3.9.1 功能 26](#_Toc13755809)

[3.9.2 数据结构 27](#_Toc13755810)

[3.9.3 算法 27](#_Toc13755811)

[4. 程序设计与实现 29](#_Toc13755812)

[a) 程序相关图示 29](#_Toc13755813)

[b) 程序说明 30](#_Toc13755814)

[c) 实验结果 31](#_Toc13755815)

[5. 结论 34](#_Toc13755816)

[6. 参考文献 36](#_Toc13755817)

[7. 收获、体会和建议 36](#_Toc13755818)

# 概述

EXT2文件系统以其高效稳定的特点深受用户欢迎，而后在其基础上发展而来的EXT3/4等文件系统，更是加入了日志管理功能，使系统的安全性获得了很大地提高。但是对于文件i结点的使用上，EXTX文件系统采用了相对保守的策略，虽然节省内存空间，但是未能很好地利用内存速度快的特点，大量的磁盘I/O操作很可能造成系统性能的瓶颈。正是出于这个原因，我们希望设计出一个高效能的文件系统，利用“以空间换时间”的策略，适当牺牲内存空间以换取效率的大幅提升，并且从当前计算机硬件配置的情况来看，这种牺牲带来的代价是微小的，其占用的内存空间完全不会影响其他程序的正常运行。因此，本系统是对EXTX文件系统的一个尝试性改进，收到了较好的实验效果。

# 课程设计任务及要求

## 2.1 设计任务

在下列内容中任选其一：

1、多用户、多级目录结构文件系统的设计与实现；

2、WDM驱动程序开发；

3、存储管理系统的实现，主要包括虚拟存储管理调页、缺页统计等；

4、进程管理系统的实现，包括进程的创建、调度、通信、撤消等功能；

5、自选一个感兴趣的与操作系统有关的问题加以实现，要求难度相当。

## 2.2 设计要求

1、在深入理解操作系统基本原理的基础上，对于选定的题目，以小组为单位，先确定设计方案；

2、设计系统的数据结构和程序结构，设计每个模块的处理流程。要求设计合理；

3、编程序实现系统，要求实现可视化的运行界面，界面应清楚地反映出系统的运行结果；

4、确定测试方案，选择测试用例，对系统进行测试；

5、运行系统并要通过验收，讲解运行结果，说明系统的特色和创新之处，并回答指导教师的提问；

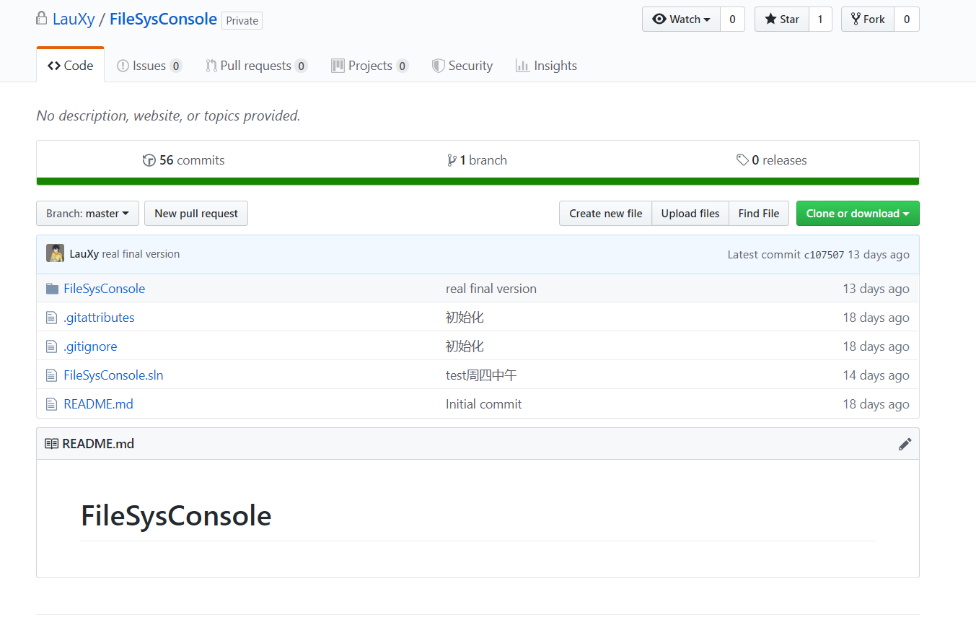
6、提交课程设计报告。

# 算法及数据结构

## 3.1 算法的总体思想（流程）

本文件系统在设计上可以分为底层和上层两大部分，底层主要解决数据结构设计、底层算法设计、接口设计等，需要与文件系统的物理实现打交道，设计者需要清晰地掌握文件系统的架构和执行流程；而上层部分属于二次开发，在底层提供的接口上进行功能的拓展、算法的优化等，充分利用软硬件优势和架构设计特点对文件系统的可靠性、安全性和执行效率进行深度优化。在设计时需要约定好底层和上层的交界面，并完整而详细地了解整个系统架构，才能设计出一个好的文件系统。

在版本控制和团队协作上，使用Git版本控制：



## 3.2 用户文件权限模块

### 3.2.1 功能

该模块主要涉及对用户文件权限的相关操作，包括授权、收权、改权、鉴权等，通过对不同用户不同文件（或目录）设置不同的权限，以实现文件的共享与保护，从而增强文件系统的安全性。

### 3.2.2 数据结构

public class DiskiNode

{

...

//用户ID，1~1000用户组1，1001~2000用户组2...(uid,authority)

public Dictionary<uint, uint> uid;

...

}

每个文件（或文件夹）的i结点中存放用户权限信息，其数据结构为List<Dictionary<uint, uint>>，List内每一个元素是一对映射关系，第一个参数是用户ID，第二个参数是该用户对该文件享有的权限值，二者均为无符号整型数值（uint），权限值的取值范围为0~7，不同范围的用户ID被划分到不同的用户组，如1~1000号用户属于用户组1，1001~2000号用户属于用户组2……依此类推，0号用户为超级管理员，是系统自创建时便存在的一个用户，该用户对所有文件及目录享有所有权限，并可以修改和收回其他用户的权限，而普通用户无权对超级管理员的权限做任何修改和限制。

**Table 3.2.2.1 用户文件权限对照表**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **权限值**  **权限** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **可读r** | 0 | 0 | 0 | 0 | 1 | 1 | 1 | 1 |
| **可写w** | 0 | 0 | 1 | 1 | 0 | 0 | 1 | 1 |
| **可执行x** | 0 | 1 | 0 | 1 | 0 | 1 | 0 | 1 |

关于授权问题，一个普通用户（ID>0）对另一个普通用户授权，只能授予少于等于己方的权限，例如A用户对File\_Exp文件享有的权限值为6，即对文件拥有读写权限，而没有执行权限，那么在授权操作中，用户A只能向用户B授予可读权限或可写权限或读写权限，即可授予的权限值集合为{2,4,6}。收权操作也是如此。

**Table 3.2.2.2 可授予权限值对照表**

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **权限值** | **0** | **1** | **2** | **3** | **4** | **5** | **6** | **7** |
| **可授权值集合** | 0 | 1 | 2 | 1,2,3 | 4 | 1,4,5 | 2,4,6 | 0~7 |

### 3.2.3 算法

* **授权算法**（伪代码描述）：

/\* Authorization Algorithm \*/

/\* --param name='filename'

--param name='username'

--param name='authval' \*/

Begin

1. inode <- GetiNodeByName(filename)

2. if CheckLegality(current\_user, username, authval) == true:

2.1 inode.uid.add(username, authval)

3. else:

3.1 return ERROR

4. return OK

End

* **收权（改权）算法**：

/\* Recycle/Change Authority Algorithm \*/

/\* --param name='filename'

--param name='username'

--param name='new\_auth' \*/

Begin

1. inode <- GetiNodeByName(filename)

2. if CheckLegality(current\_user, username, new\_auth) == true:

2.1 inode.uid.replace(username, new\_auth)

3. else:

3.1 return ERROR

4. return OK

End

* **鉴权算法：**

/\* Check Authority Algorithm \*/

/\* -- param name='filename' \*/

/\* -- param name='operation' \*/

Begin

1. inode <- GetiNodeByName(filename)

/\* 检查该文件权限列表中是否存在该用户的记录 且 用户当前操作对应的权限值在允许的权限集合范围内 \*/

2. if inode.uid.Contains(current\_user)

and inode.uid[current\_user].Contains(calval(operation)):

2.1 return OK

3. else:

3.1 return ERROR

End

## 3.3 登入登出模块

### 3.3.1 功能

该模块实现了用户登录文件系统和退出文件系统，对多用户提供管理和支持。该模块在实现中添加了大量的异常处理，以应对使用者各种可能的输入，对于错误的输入，会给出准确的提示信息，以便帮助用户快速找到错误的输入项进行修正。

### 3.3.2 数据结构

/// <summary>

/// 磁盘用户项

/// </summary>

public class User

{

public uint uid; //用户uid

public string password;

public uint current\_folder;//上次退出时所在文件夹的i节点ID(可以加一个“回到上次工作区”功能)

public User(uint uid = 0, string pwd = "")

{

this.uid = uid;

this.password = pwd;

}

}

/// <summary>

/// 内存用户项

/// </summary>

public class MemoryUser

{

//建议：用户数小于10

public uint uid; //用户uid

public uint current\_folder; //当前所在文件夹的i节点ID

public List<uint> open\_file = new List<uint>();//用户打开文件表

public string newpassword; //用户新密码

public MemoryUser(uint uid, uint cf, string pwd)

{

this.uid = uid;

this.current\_folder = cf;

this.open\_file.Add(cf);

this.newpassword = pwd;

}

}

该部分的数据结构主要包含磁盘用户项和内存用户项，相比于磁盘用户项，内存用户项又添加了一些字段，用来记录用户打开文件及当前所在目录等信息，其中用户打开文件表用于控制文件访问的互斥操作（该部分会在后面详细展开介绍）。在每次登录系统时，系统会将磁盘用户项加载到内存中，由于此时用户还未正式登录系统，为了节省内存空间，这里不生成内存用户项，仍保持磁盘用户项的数据结构。当身份核验完成后，用户进入系统，才会变为内存用户项驻留在内存中。

### 3.3.3 算法

* **登入算法：**

Begin

1. input(id, password)

2. check whether input is legal or not

3. List<DiskUser> userlist <- LoadUsersInfofromDisk()

4. if id in userlist then CheckPassword()

else throw an error to user

4.1 if password is not correct then throw the error to user

4.2 else cur\_user\_num += 1

4.3 if cur\_user\_num < MAX\_USER\_NUM then new MemoryUser

else throw exception to user

4.5 login system successfully

End

* **登出算法：**

Begin

1. 更新磁盘用户项，将内存用户项中的数据写回磁盘

2. cur\_user.release() 释放当前用户资源

3. cur\_user = null

4. cur\_user\_num -= 1

End

## 3.4 删除模块

### 3.4.1 功能

删除操作包含两类操作——彻底删除和移入回收站，每一类操作又可以分为对文件的操作和对目录的操作，后两者有着明显的区别，对文件的删除只需修改当前文件i结点和上级目录的i结点，但是对目录的删除需要使用递归算法，级联删除该目录下所有的子目录及文件。移入回收站的文件或文件夹可以还原到原位置，但彻底删除操作会回收磁盘块，该操作是不可逆的。

### 3.4.2 数据结构

/// <summary>

/// 回收站文件地址映射（用于还原）List<Dictionary<inode\_id, fore\_addr\_id>>

/// </summary>

Dictionary<uint, uint> recyclebinMap = new Dictionary<uint, uint>();

如上所示是回收站文件映射表，保存了回收站中文件的i结点id和该文件的原目录地址，用于还原操作。

/// <summary>

/// i节点

/// </summary>

public class DiskiNode

{

public uint id; //磁盘i节点ID

public string name; //文件(夹)名

public uint size; //文件(夹)大小

//用户ID，1~1000用户组1，2~2000用户组2...(uid,authority)

public Dictionary<uint, uint> uid;

//文件的磁盘块地址或者文件夹下的文件(夹)的i节点ID

public List<uint> next\_addr = new List<uint>();

public uint fore\_addr; //上层目录的i的ID

public DateTime t\_create; //文件(夹)创建时间

public DateTime t\_revise; //文件(夹)修改时间

public ItemType type; //类型：文件/文件夹

}

### 3.4.3 算法

* **删除文件（彻底删除）：**

Begin

1. inode <- GetiNodeByPath(filepath)

2. flag = CheckAuthority() 检查用户是否具有可写权限

3. if flag == true then RecycleiNode(inode) 回收i结点

else return false 该用户没有删除文件的权限

4. return true 删除操作成功

End

* **删除目录（彻底删除）：**

Begin

1. if inode.type == FILE then DeleteFile(inode)

2. else List<INode> del\_list <- GetiNode(inode.next\_addr)

3. Traverse the del\_list

3.1 DeleteFolder(item) 递归删除子文件(夹)

4. RecycleiNode(inode) 回收该文件夹的i结点

End

* **删除文件/文件夹（移入回收站）：**

Begin

1. inode <- GetiNodeByPath(filepath)

2. if CheckAuthority() == false then return ERROR

3. else 记录inode与父级目录的映射关系

4. 从inode父级目录的子域中删除inode

5. inode.fore\_addr <- 回收站i结点id

6. 回收站i结点的next\_addr中加入inode

7. return OK

End

与彻底删除相区别的是，移入回收站并不需要级联操作，只需将该文件的inode移到回收站下即可，如果当前类型为目录，移入回收站后，目录与其子级文件或目录之间的关系不发生改变，无需对其子一级i结点进行修改。需要修改的指针如下所示：

回收站

i结点

待删除文件(夹)

父级i节点

待删除文件(夹)

i结点

源地址映射表

目录

图3.4.1 移入回收站操作指针修改示意图

* **还原文件/文件夹（移出回收站）：**

Begin

1. filename <- input

2. search in recyclebin, get inode

3. if 有多个同名的文件/文件夹 then select\_file <- input

4. check authority

5. inode.fore\_addr <- recyclebinMap[inode.id]

6. inode父级子域添加inode.id

7. 将inode.id从回收站i结点的子域中移除

8. return inode

End

该算法为移入回收站算法的逆操作，这里不再赘述。

* **清空回收站：**

清空回收站即将回收站中所有文件及目录彻底删除，该操作需要获取回收站的子级i结点，然后逐个遍历，调用DeleteFolder()函数进行删除。

## 3.5 复制模块

### 3.5.1 功能

与删除操作的分类相似，复制也可分为复制文件和复制目录，但区别在于，复制操作不是对i结点指针域的简单修改，而是需要创建新的i结点，然后链接到指定的目录。复制文件仅需创建一个i结点，而复制目录需要递归地创建出该目录下所有文件和目录的i结点。因此，对复制后文件或目录的任何修改操作都不会源文件的内容。

### 3.5.2 数据结构

本模块涉及的数据结构仍为DiskiNode（i结点），该数据结构已在[模块3.4](#_3.4.2_数据结构)中列出，这里不再赘述。

### 3.5.3 算法

* **复制文件：**

Begin

1. from\_inode <- GetiNodeByPath(source\_path)

2. 判断当前用户对该文件是否有可读权限

3. to\_inode <- GetiNodeByPath(target\_path)

4. 判断当前用户对该目录是否有可写权限

5. 在目的目录中对源文件进行重名检查

6. if 存在相同文件名且类型相同的文件 return ERROR

7. 创建新的i结点，并为之申请磁盘空间

8. 将原i内存结点的信息复制到新i的内存结点

9. to\_inode.next\_addr.Add(new\_inode.id)

new\_inode.fore\_addr = to\_inode.id

10.将原i磁盘结点的信息复制到新i的磁盘结点, 更新磁盘i结点

End

值得注意的是，复制文件操作需要当前用户对源文件具有可读权限，对目的目录具有可写（可修改）权限。只有二者均具备上述要求，文件复制操作才可进行。复制时不仅需要对内存i结点操作，也需要在磁盘中复制i结点，最后要更新磁盘i结点。

* **复制文件夹：**

Begin

1. itemlist <- 源目录下所有文件/文件夹的i结点

2. if type == FOLDER，在目的目录下创建一个与源文件夹同名的文件夹

3. for item in itemlist Loop CopyFolder()

4. else if type == FILE then CopyFile()

End

## 3.6 移动模块

### 3.6.1 功能

移动是将一个文件或文件夹移到另一个指定的目录下，与复制操作不同的是，文件移动后便不在源目录下，而是存在于新的目录下，这种操作类似于我们常用的剪切和粘贴，移动是将两个操作合二为一的结果，移动文件始终是对同一个文件或目录进行操作，不需要新创建和申请i结点，但是需要修改磁盘i结点。

### 3.6.2 数据结构

移动操作涉及的数据结构也是DiskiNode，相关定义详见[模块3.4](#_3.4.2_数据结构)。

### 3.6.3 算法

Begin

1. inode <- GetiNodeByPath(source)

2. 检查当前用户对该文件是否有可修改权限，若无，退出

3. 若有，判断该文件或目录是否为回收站，若是，退出

4. 若否，target\_inode <- GetiNodeByPath(target)

5. 检查当前用户对该目录是否具有可修改权限。若无，退出

6. 若有，检查目的目录下是否存在与源文件同名同型冲突的文件，若有，退出

7. 若无，inode.fore\_addr <- target\_inode.id

8. target\_inode.next\_addr <- inode.id

9. 将变更写回磁盘，更新磁盘i结点

10.正常退出

End

移动操作要求当前用户对源文件和目的目录同时具备可写（可修改）权限，移动操作仅需修改i结点的指针域，移动文件夹无需递归。

## 3.7 搜索模块

### 3.7.1 功能

该模块实现了对文件和文件目录的查找，用户通过输入文件名或文件路径（支持绝对路径和相对路径），提交到文件系统，执行搜索操作。搜索可分为全盘搜索（全盘查找一个文件）和局部搜索（指定路径或范围的搜索），搜索返回的结果可能是单一的，也可能是多个。本文件系统为搜索提供了两种模式，一种为普通模式，速度相对较慢；另一种为优化模式，通过在全局数据库中建立索引来提高文件查找的效率。

### 3.7.2 数据结构

由于搜索返回的是文件或目录i结点，故涉及的数据结构为i结点结构，相关定义及说明详见[模块3.4](#_3.4.2_数据结构)。

### 3.7.3 算法

* **全局搜索-普通模式：**

Begin

1. 将根目录压栈 stack.push(root)

2. Loop while !stack.empty()

2.1 top <- stack.pop()

2.2 visit top, check if top meet the requirements, if so, result.Add(top)

2.3 if not, 检查其子级文件或目录是否具有可读可执行权限，若无，no action

2.4 若有，将i结点逆序压栈

3. return result 返回所有符合条件的搜索结果

End

普通模式下的全局搜索使用深度优先搜索算法，遍历全树，当搜索到第一个符合条件的文件i结点时，并不停止搜索，仍继续查找，直至遍历完成，该操作可以将所有符合搜索条件的文件或目录都找出。

* **全局搜索-优化模式：**

这部分并没有涉及到具体的算法，只是执行数据库的查找操作，然后根据用户文件权限对查找结果进行过滤筛选，选出用户可读可执行的文件返回之。

* **局部搜索：**

局部搜素会根据用户输入的路径在限定的范围内搜索，一般情况下，局部搜索的规模较小，速度较快，故这里没有特别做优化。若用户输入的路径为绝对路径则可直接定位到指定文件，若用户输入为相对路径，则需要以当前路径为root进行查找，查找所用到的算法仍为深度优先搜索。

## 3.8 数据库模块

### 3.8.1 功能

该模块主要用于对全盘搜索进行优化。搜索的效率是一个十分值得关注的问题，Windows文件系统的全盘搜索功能效率不是很高，尤其在查找一个指定文件时，搜索的过程很慢，耗时长，造成用户体验不佳。为了解决这个问题，我们借鉴了Linux文件系统的设计，并在此基础上做出修改，通过创建全局数据库，建立对文件属性项的索引，从而达到高效搜索的目的。相比于扫描全树，该方法在DBMS的优化下，时间复杂度小于O(n)。此外，数据库的引入还带来了很多额外的功能，丰富了文件系统的指令，用户可以直接输入sql语句对数据库操作，由于数据库中存放的是i结点的拷贝，故一般情况下，不会对原文件造成影响，当然，我们并不建议用户使用sql语句对数据库进行不安全的操作。

### 3.8.2 数据结构

* **数据库：**SQLite 3
* **数据表名**：InodeDb.sqlite
* **数据表项：**为了节省空间，提高查询效率，数据库中存储的是精简的i结点，并不具备i结点中所有信息，如果要获取一个文件或目录的详细信息，应该根据i结点的编号进行查找。下面给出了数据表定义语句：

create table InodeTab (

id integer primary key,

name varchar(225),

size real,

t\_create text,

t\_revise text,

type text

);

### 3.8.3 算法

* **数据库操作涉及的SQL语句：**

1. 检查数据表是否存在：

select count(\*) from sqlite\_master where type='table' and name='InodeTab'

1. 创建数据表：

create table InodeTab (id integer primary key, name varchar(225), size real, t\_create text, t\_revise text, type text)

1. 导入数据：

insert into InodeTab values ({0},'{1}',{2},'{3}','{4}','{5}')

1. 建立索引：

create index if not exists index\_{0} on InodeTab({0})

1. 清空数据表中的数据：

delete from InodeTab

1. 查找：

select \* from InodeTab where name='filename'

* **第一次使用数据库时的基本流程：**

Begin

1. 新建数据库

2. 创建连接

3. 新建数据表

4. 批量导入数据

5. 对数据进行操作

6. 关闭连接

End

* **批量导入数据：**

开启事务后，执行数据导入可大幅提高批量数据的导入效率，具体的流程为：开启事务 🡪 导入批量数据 🡪 关闭事务（提交）。我们利用事务的互斥性，如果在批量的插入操作前显式地开启一次事务，在插入操作结束后，提交事务，那么所有的操作将只执行一次事务，大大地提高了IO效率。而如果不显式开启事务，那么每次执行insert操作之前都会开启事务，执行完成后提交事务，若insert操作需要执行10000次，那么事务开启和关闭也要对应地执行10000次，显而易见，显式开启事务对于批量数据导入性能的提升具有至关重要的作用。下面的表格通过对比前后两者的执行时间，很好地表现了二者在执行效率上的差异（注：表中数据来源于网络）。

|  |  |  |
| --- | --- | --- |
| **记录数** | **单条数据插入** | **合并数据（事务）** |
| 100 | 0 min 0.109 s | 0 min 0.009 s |
| 1000 | 0 min 1.432 s | 0 min 0.026 s |
| 10000 | 0 min 13.093 s | 0 min 0.755 s |
| 注：数据基础记录数为500,000 | | |

## 3.9 磁盘分区模块

每个磁盘块1024B，文件系统大小为4GB（因使用unsigned int寻址限制，使用unsigned long可拓展至最大2^34 GB），磁盘分区情况如下所示：



超级块：0~2块，大小为3KB

用户信息：3~9块，大小为7KB

回收站map表：10~99块，大小为90KB

i节点：100~3999块，大小为3899KB(3.9MB)

数据区：4000~4294967295块，大小为4092MB(3.996GB)

## 3.10 数据结构模块

**超级块：**

public class SuperBlock

{

//uint.MAX = 1024\*1024\*1024\*4 = 4,294,967,295

public const uint BLOCK\_SIZE = 1024; //块大小

public const uint BLOCK\_SUM\_NUM = 1024 \* 1024 \* 4;//块的总数量，4GB

public const uint iNODE\_SUM\_NUM = 1024 \* 50; //i节点的总数量

public const uint BLOCK\_IN\_GROUP = 128; //每一组有128块

public const uint SB\_DISK\_START = 0; //超级块区磁盘起始块号

public const uint USER\_DISK\_START = 3; //用户信息区磁盘起始块号

public const uint RECYCLEBINMAP\_DISK\_START = 10; //回收站Map表区磁盘起始块号

public const uint iNODE\_DISK\_START = 100; //i节点区磁盘起始块号

public const uint DATA\_DISK\_START = 4000; //数据区磁盘起始块号

public uint free\_block\_num = 1023 \* 1024 \* 4; //空闲块的数量,4092MB

public uint free\_inode\_num = 1024 \* 50; //空闲i节点的数量

public uint last\_group\_block\_num = BLOCK\_IN\_GROUP;//最后一组的块的数量

public List<uint> last\_group\_addr; //最后一组的块的地址(保留区大小为4000块)

public bool change\_flag = true; //超级块修改标志

public uint max\_inode\_id = 100; //当前分配的最大i节点ID,因为有默认文件夹,所以第一次取个大一点的数值

public uint check\_byte = 707197; //校验位

}

**i节点：**

public class DiskiNode

{

public uint id; //磁盘i节点ID

public string name; //文件(夹)名

public uint size; //文件(夹)大小

public Dictionary<uint, uint> uid; //用户ID，1~1000用户组1，2~2000用户组2...(uid,priority)

public List<uint> next\_addr = new List<uint>();//文件的磁盘块地址或者文件夹下的文件(夹)的i节点ID

public uint fore\_addr; //上层目录的i的ID

public DateTime t\_create; //文件(夹)创建时间

public DateTime t\_revise; //文件(夹)修改时间

public ItemType type; //类型：文件/文件夹

}

在这里，我们的i节点设计与以往不同：i节点里记录了文件层级目录关系，去掉了“目录”的概念，i节点的fore\_addr为该节点的父节点ID，对于next\_addr数组，若此i节点的type为文件夹，则数组内存放的是该文件夹下的文件和文件夹对应的i节点ID，若此i节点的type为文件，则数组内存放的是该文件的磁盘块地址列表。

**用户：**

public class MemoryUser

{

//建议：用户数小于10，每个用户打开文件数小于40

public uint uid; //用户uid

public uint current\_folder; //当前所在文件夹的i节点ID

public List<uint> open\_file = new List<uint>();//用户打开文件表

public string newpassword; //用户新密码

}

**组长块：**

public class BlockLeader

{

public uint next\_blocks\_num; //下一组的磁盘块数量

public List<uint> block\_addr = new List<uint>();//下一组的每一块的地址

}

**多次间址：**

//虚拟间址寻址结构

public abstract class Data

{

public abstract int getItem(uint i = 0, uint j = 0);

}

//一次间址寻址结构实现

public class DataInt : Data

{

int adata;

public DataInt(int a)

{

adata = a;

}

public override int getItem(uint i = 0, uint j = 0)

{

return adata;

}

}

//二次间址寻址结构实现

public class DataIntArray : Data

{

public int[] data = new int[24];

public override int getItem(uint i = 0, uint j = 0)

{

return data[i];

}

public DataIntArray(int key, int value)

{

data[key] = value;

}

}

//三次间址寻址结构实现

public class DataIntArrayArray : Data

{

public int[,] data = new int[24, 24];

public override int getItem(uint i = 0, uint j = 0)

{

return data[i, j];

}

public DataIntArrayArray(int key1, int key2, int value)

{

data[key1, key2] = value;

}

}

## 3.11 文件(夹)创建模块

### 3.11.1 功能

创建文件或文件夹，若无重名则成果创建，否则返回错误。

### 3.11.2 数据结构

该功能相关的数据结构为i节点和超级块，详情见：3.10 数据结构模块

### 3.11.3 算法

Begin:

1. 获取要创建的文件(夹)所在目录，进行重名检测，未重名转(2)，否则报错并结束；

2. 分配i节点，若为文件则预分配一个磁盘块；

3. 更新超级块、i节点；

End

## 3.12 文件的打开/关闭/读/写/重命名模块

### 3.12.1 功能

本模块完成文件的相关操作，主要包含：文件的打开、文件的关闭、文件的读取、文件的写入、文件的重命名。

### 3.12.2 数据结构

本模块涉及的数据结构为i节点，详见3.10 数据结构章节。

### 3.12.3 算法

Begin:

Creat:

1.1 获取所创建文件(夹)父级目录i节点

1.1 重名检测，True转1.3，Flase继续

1.2 分配i节点，修改i节点列表和超级块，若为文件，预分配磁盘块，转End

1.3 报错提示

Open:

2.1 获取用户打开表、系统打开表，检测是否已存在，True转2.3，False继续

2.2 将文件i节点ID加入用户打开表、系统打开表，转End

2.3 报错提示

Close:

3.1 获取用户打开表、系统打开表，检测是否已存在，False转3.3，True继续

3.2 将文件i节点ID从用户打开表、系统打开表删除，转End

3.3 报错提示

Read:

4.1 根据路径，获取文件的i节点ID，False转4.4，True继续

4.2 根据i节点，获取磁盘块列表，null转4.4，List继续

4.3 根据磁盘块地址，依次读取到缓冲区并显示，转End

4.4 报错提示

Write:

5.1 根据路径，获取文件的i节点ID，False转5.5，True继续

5.2 从缓冲区读入修改后的文件内容并计算长度

5.3 合理分配磁盘块数量（新内容比现有磁盘块长则新加分配磁盘块，比现有磁盘块短则删减分配磁盘块）

5.4 根据磁盘块地址，依次写入到缓冲区并存储在磁盘上，转End

5.5 报错提示

Rename:

6.1 根据路径，获取文件的i节点ID，False转6.4，True继续

6.2 检测新名字是否重名冲突，False继续，True转6.4

6.3 更改i节点信息，转End

6.4 报错提示

End

## 3.13 安装系统和格式化模块

### 3.13.1 功能

提供首次一键安装文件系统服务和文件系统格式化服务，至少需要磁盘空间4.2MB，之后随着文件内容的写入，进行动态扩容，最大支持4GB空间。在安装过程中会对超级块、i节点、用户、回收站及数据区做初始化，预分配系统保留区，建立组长块成组链接机制。在格式化过程中，根据用户权限格式化对应的自己权限内的文件和文件夹，ROOT用户有权格式化所有分区，包括用户信息区。

### 3.13.2 数据结构

本模块因为要初始化系统，所以涉及几乎所有数据结构。

### 3.13.3 算法

Begin:

Install:

1.1 清空文件系统所有信息

1.2 设置默认ROOT账户

1.3 在ROOT账户模式下，执行format函数

Format:

2.1 判断用户权限，ROOT转2.2，普通用户转2.5

2.2 初始化超级块、i节点、回收站、组长块等信息

2.3 初始化用户文件夹

2.4 将上述信息写入磁盘，转End

2.5 设置当前目录为本用户根目录

2.6 将本用户文件夹下所有文件夹和文件级联删除，转End

End

## 3.9 底层服务模块

### 3.9.1 功能

作为底层实现函数，介于物理磁盘与顶层功能函数之间，屏蔽了对物理磁盘的操作，封装操作逻辑，给顶层功能函数提供一个函数接口，使物理磁盘对顶层实现函数透明化，将文件系统代码解耦。

底层服务模块主要实现函数有：

/// 将用户信息写回到磁盘

public bool StoreUserInfotoDisk(MemoryUser currentUser);

/// 从磁盘用户区加载所有用户信息到内存

public List<User> LoadUsersInfofromDisk();

/// 更新磁盘的超级块或i节点，默认全写回(true,true)，第一个参数决定超级块是否写回，第二个是i节点

public bool UpdateDiskSFi(bool sb = true, bool inode = true);

/// 分配i节点ID，正常则返回i节点ID，错误则返回0

public uint AllocAiNodeID();

/// 输入ID，返回i节点结构，错误则i节点name为.

public DiskiNode GetiNode(uint id);

/// 输入路径，返回i节点结构

public List<DiskiNode> GetiNodeByPath(string path);

/// 判断命名是否冲突，冲突返回true

public bool IsNameConflict(DiskiNode fold\_node,string name,ItemType type);

/// 分配磁盘块,正常则返回块地址,错误则返回0,未写回超级块

public uint AllocADiskBlock();

/// 清除一个磁盘块，写满\0

public bool EraseBlock(uint block\_order);

/// 回收磁盘块

public bool RecycleDiskBlock(uint block\_addr);

/// 复制旧i节点物理盘块到新i节点物理盘块，正常返回true

public bool CopyiNodeDisk(DiskiNode from, DiskiNode to);

/// 解析输入的命令

public uint ParsingCommands(string commandstr);

### 3.9.2 数据结构

本模块作为底层服务，涉及到几乎所有数据结构。

### 3.9.3 算法

* 将用户信息写回到磁盘：

for(int i=0; i<用户数量;i++)

{

找到当前用户；

更新密码；

打开文件系统文件；

将用户信息序列化；

从超级块找到用户信息区开始磁盘块号；

写入磁盘并返回True；

}

返回False;

* 从磁盘用户区加载所有用户信息到内存：

打开文件系统文件；

从超级块找到用户信息区开始磁盘块号；

开辟内存空间；

从磁盘读取数据并反序列化；

将反序列化的数据存到内存空间；

* 更新磁盘的超级块或i节点，默认全写回(true,true)，第一个参数决定超级块是否写回，第二个是i节点：

打开文件系统文件；

从超级块找到超级块区和i节点区开始磁盘块号；

根据需要，将内存数据序列化并写回磁盘；

* 分配i节点ID，正常则返回i节点ID，错误则返回0：

i节点ID = 0;

if ( uint的最大值>当前使用的最大i节点ID >= 100)

{

i节点ID = ++当前使用的最大i节点ID;

更新超级块磁盘数据;

}

返回i节点;

* 输入ID，返回i节点结构，错误则i节点name为"."：

uint temp\_id = id % 128;

新建i节点，其name为".";

选取第tem\_id个i节点列表

for (int i = 0; i <i节点数量; i++)

{

if (第i个i节点ID== id)

{

return 第i个i节点;

}

}

return 新建i节点;

* 输入路径，返回i节点结构：

flod=当前所在文件夹的i节点；

新建两个i节点list（head\_list，tail\_list）；

新建错误i节点，其name为"."；

if(输入路径是绝对路径){fold=根目录；}

根据“/”分隔字符串并将分割后的每个字符串记录在name\_list里；

head\_list.add(fold)

tail\_list.add(fold)

去除空的字符串；

for(i=0;i<name\_list.lenght,i++)

{

if(name\_list[i]=="."){}

else if(name\_list==".."){把当前结果遍历，获取每个结果上级目录并更新head\_list}

else{遍历当前结果，若匹配到了当前name且是文件夹或者是文件但当前结果不是最后一级，将其加入到head\_list，}

更新tail\_list；

}

返回head\_list；

* 分配磁盘块,正常则返回块地址,错误则返回0,未写回超级块：

若最后一组块数大于1：

更新超级块信息；

若最后一组块数=1：

使用组长块并更新，把倒数第二组加到超级块

# 程序设计与实现

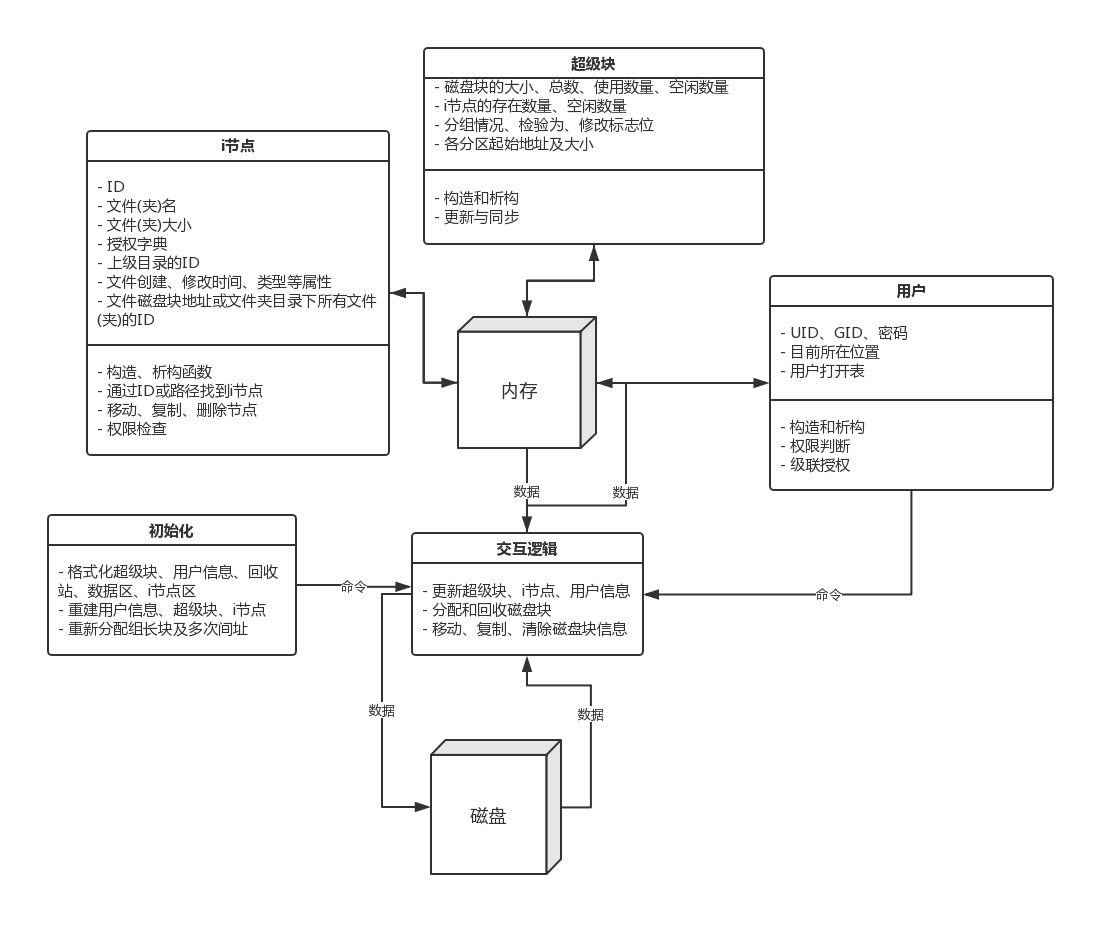
## 程序相关图示

**1. 总体流程图**

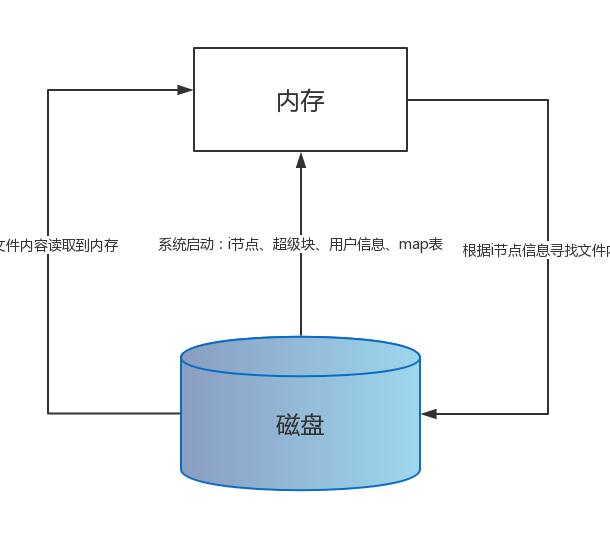
图片包含 标牌, 物体

描述已自动生成

**2. 类流程图：**



**3. 数据流动图**



## 程序说明

运行环境：Windows 2000及以上

内存占用：4.5MB~6MB

磁盘占用：至少要有4MB大小的空间

命令格式（正则表达式规范）：

@"^help$",//0:help

@"^.+ -h$",//1:help

@"^ls[ ]\*$",//2:输出当前路径下的文件(夹)，缺省为当前目录

@"^ls [^- ]+$",//3:输出某一路径下的文件(夹)

@"^ls -s (name|size|type|creat|revise)$",//4:按顺序输出当前目录

@"^ls -s (name|size|type|creat|revise) [^- ]+$",//5:按顺序输出某一目录

@"^ll[ ]\*$",//6:输出当前路径下的文件(夹)，缺省为当前目录

@"^ll [^- ]+$",//7:输出某一路径下的文件(夹)

@"^ll -s (name|size|type|creat|revise)$",//8:按顺序输出当前目录

@"^ll -s (name|size|type|creat|revise) [^- ]+$",//9:按顺序输出某一目录

@"^touch [^- ]+$",//10:创建文件，输入带新文件名的path

@"^touch -f [^- ]+$",//11:创建文件夹，输入带新文件名的path

@"^file [^- ]+$",//12:查看某文件/文件夹详情

@"^file -d [^- ]+$",//13:查看某文件/文件夹详情，含addr

@"^cat [^- ]+$",//14输出文件内容

@"^cmp [^- ]+ [^- ]+$",//15比较两个文件

@"^cmp -d [^- ]+ [^- ]+$",//16比较两个文件，含i节点（除ID）比较

@"^write [^- ]+$",//17写文件

@"^find [^- ]+$",//18当前目录查找

@"^find [^- ]+ [^- ]+$",//19某一目录下查找

@"^find -a [^- ]+$",//20全盘搜索

@"^find -r size [^- ]+ [^- ]+ [^- ]+$",//21当前限定范围目录查找

@"^find -r type [^- ]+ [^- ]+$",//22当前限定范围目录查找

@"^find -r (create|revise) (20[0-9][0-9])/(1[0-2]|[1-9])/(3[0-1]|[1-2][0-9]|[1-9]) ([0-9]|1[0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9]) (20[0-9][0-9])/(1[0-2]|[1-9])/(3[0-1]|[1-2][0-9]|[1-9]) ([0-9]|1[0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9]) [^- ]+$",//当前限定范围目录查找

@"^find -r size [^- ]+ [^- ]+ [^- ]+$",//24某一限定范围目录查找

@"^find -r type [^- ]+ [^- ]+$",//25某一限定范围目录查找

@"^find -r (create|revise) (20[0-9][0-9])/(1[0-2]|[1-9])/(3[0-1]|[1-2][0-9]|[1-9]) ([0-9]|1[0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9]) (20[0-9][0-9])/(1[0-2]|[1-9])/(3[0-1]|[1-2][0-9]|[1-9]) ([0-9]|1[0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9]) [^- ]+ [^- ]+$",//某一限定范围目录查找

@"^find -a -r size [^- ]+ [^- ]+ [^- ]+$",//27全盘限定范围目录查找

@"^find -a -r type [^- ]+ [^- ]+$",//28全盘限定范围目录查找

@"^find -a -r (create|revise) (20[0-9][0-9])/(1[0-2]|[1-9])/(3[0-1]|[1-2][0-9]|[1-9]) ([0-9]|1[0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9]) (20[0-9][0-9])/(1[0-2]|[1-9])/(3[0-1]|[1-2][0-9]|[1-9]) ([0-9]|1[0-9]|2[0-3]):([0-5][0-9]):([0-5][0-9]) [^- ]+$",//全盘限定范围目录查找

@"^index [^- ]+$",//30建立索引

@"^sql$",//31输入SQL

@"^mv [^- ]+ [^- ]+$",//32移动文件或文件夹

@"^mv -d [^- ]+ [^- ]+$",//33移动时显示都移动了哪些文件/文件夹

@"^rm [^- ]+$",//34删除文件或文件夹

@"^rm -d [^- ]+ [^- ]+$",//35删除时显示都删除了哪些文件/文件夹

@"^rm -c [^- ]+$",//36删除文件或文件夹，并不放回收站

@"^rm -c -d [^- ]+$",//37删除时显示都删除了哪些文件/文件夹，并不放回收站

@"^cp [^- ]+ [^- ]+$",//38复制文件

@"^cp -d [^- ]+ [^- ]+$",//39复制时显示都复制了哪些文件/文件夹

@"^rec [^- ]+$",//40还原某个文件

@"^rec [^- ]+ -p [^- ]+$",//41还原某个文件到指定路径

@"^rec -clr$",//42清空回收站

@"^rec -show$",//43显示回收站

@"^chomd [^- ]+ -name [^ ~]+$",//44重命名

@"^chomd -a [^- ]+ [\d]+ [0-7]$",//45更改权限

@"^login$",//46登录

@"^logout$",//47注销

@"^passwd$",//48更改密码

@"^install$",//49安装文件系统

@"^format$",//50格式化文件系统

@"^cd [^ ]+$",//51进入目录

@"^rm -f [^- ]+$",//52删除文件夹

@"^rm -c -f [^- ]+$",//53彻底删除文件夹，并不放回收站

@"^cp -f [^- ]+ [^- ]+$",//54复制文件夹

@"^chomd -r [^- ]+ [\d]+ [0-7]$",//55更改权限

## 实验结果

1、安装和格式化文件系统、多级目录的显示、文件的创建、目录更改等操作：

图片包含 文字

描述已自动生成

2、多字节读写文件：

图片包含 户外, 天空, 建筑物

描述已自动生成

3、查看文件详情、查找命令：

图片包含 屏幕截图

描述已自动生成

图片包含 屏幕截图

描述已自动生成

4、删除、移动、复制文件和文件夹

图片包含 文字, 屏幕截图

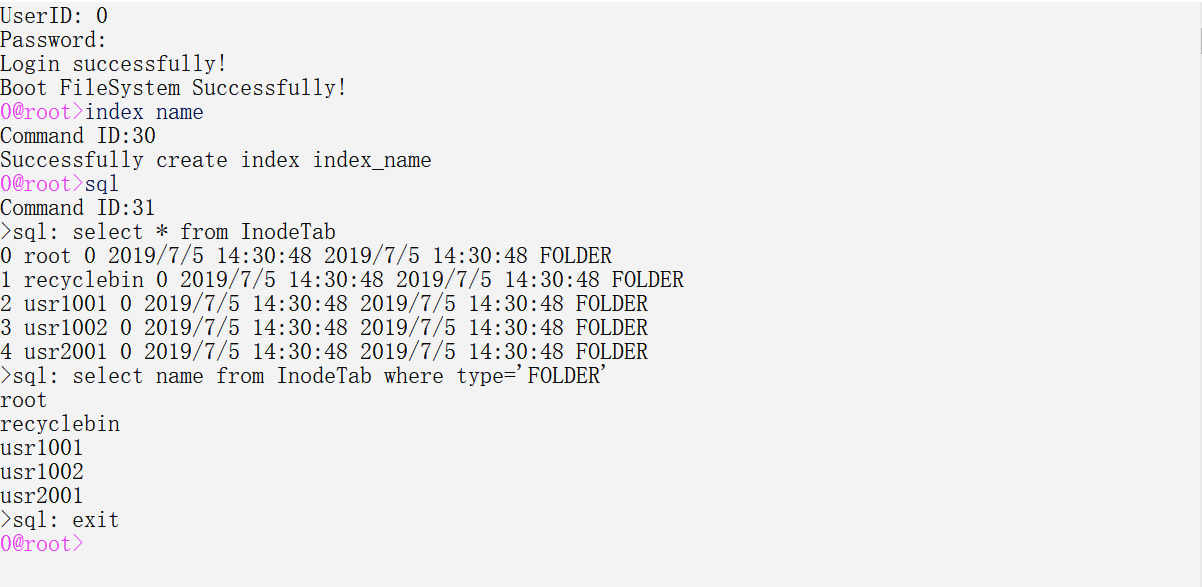
描述已自动生成

5、登录、登出以及权限操作

图片包含 文字

描述已自动生成

6、SQL相关（建立索引、SQL语句查询）



# 结论

在本次操作系统课程设计里，我们利用一周的时间，基于我们的目标出发，从底层重新设计了一个文件系统，并完成了大多数功能的实现。

首先，我们考虑到，EXTX系列的文件系统对于文件i结点的使用上，采用了相对保守的策略，分成了磁盘i节点和内存i节点，虽然节省内存空间，但是未能很好地利用内存速度快的特点，大量的磁盘I/O操作很可能造成系统性能的瓶颈。出于这个原因，我们希望设计出一个高效能的文件系统，利用“以空间换时间”的策略，适当牺牲内存空间以换取效率的大幅提升，并且从当前计算机硬件配置的情况来看，这种牺牲带来的代价是微小的，其占用的内存空间完全不会影响其他程序的正常运行。在4GB大小的文件系统里，i节点区只占了不到4MB。因此本系统是对EXTX文件系统的一个尝试性改进，收到了较好的实验效果。

之后，我们进行分工，一人从底层的数据结构和磁盘分区设计开始，完成底层功能函数实现并进行封装，为上层应用提供接口，一人负责顶层功能实现，完成目标任务。

最终，我们除了实现了课程设计基本要求的功能点：

* 多用户
* 多级目录：可有多级子目录
* 具有login (用户登录)
* 系统初始化（建文件卷、提供登录模块）
* 文件的创建： create
* 文件的打开：open
* 文件的读：read
* 文件的写：write
* 文件关闭：close
* 删除文件：delete
* 创建目录：mkdir
* 改变目录：chdir
* 列出文件目录：dir
* 退出：logout
* 格式化：format

之外，还实现了：

* 输出某一指定路径下的文件(夹)
* 按(文件名、大小、创建时间、修改时间等)顺序输出当前目录
* 比较两个文件
* 当前或指定目录查找文件（实现正则匹配）
* 全盘搜索（正则匹配方案）
* 全盘搜索（SQLite数据库索引方案）
* 建立索引、输入SQL语句查询
* 移动文件或文件夹
* 复制文件或文件夹
* 删除文件或文件夹
* 重命名文件或文件夹
* 文件和文件夹大小实时显示
* 回收站机制（回收站的还原、清空、查看等操作）
* 权限：（读/写/见）权限的授予和收回、级联授权机制
* 注销登录更改密码

美中不足的是，我们原本打算设计实现的图形化界面由于时间关系并没有做完，其界面如下：

图片包含 屏幕截图

描述已自动生成

# 参考文献

1. 张尧学, 计算机操作系统教程（第3版），清华大学出版社

2. 汤小丹等，计算机操作系统（第4版），西安电子科技大学出版社

3. 李彤等，操作系统分析与设计，云南大学出版社

4. 张琨藏，操作系统原理DOS篇，清华大学出版社

5. 陈葆玉译，UNIX操作系统设计，北京大学出版社

# 收获、体会和建议

**刘学渊**

在本次课程设计中，我们完整地实现了一个文件系统，该系统功能丰富，指令设计贴合Linux文件系统的使用习惯，尤其是对于执行效率的优化，是本系统设计的一大亮点。本系统基于EXT2文件系统，但数据结构和底层设计与EXT2文件系统有很大不同，该系统是根据我们的需求进行重新设计的，与原文件系统相比，本系统具备更高的效率。由于我们将磁盘i结点在启动时完全加载到内存中，对文件系统的操作绝大多数都在内存中进行，充分利用了内存速度快的特点，极大地减少了磁盘IO。此外，利用一个全局数据库对查找进行优化，在数据库导入数据时还使用事务进行优化，不仅保证了数据库操作的高效性，减少数据库带来的额外开销，还尽可能地提高了文件查找的效率。整个系统在设计时，我们充分权衡时间和空间的开销，最终得到了现在的方案，我们认为该方案可以称得上是一个令人满意的方案。

关于技术选型，由于一开始打算写图形用户接口，故底层选择使用C#实现，界面用xaml，设计出一个uwp应用程序，但由于时间紧迫，我们不得不放弃界面的设计，而尽量添加更多具有实在意义的功能并对系统设计做进一步的优化。然而技术选型一旦确定便不容易再做出更改，因此我们全程使用C#编写代码，由于C#不存在全局变量的概念，源文件之间的通信变得十分困难，模块间高度的数据耦合使我们难以对模块和文件进行很清晰地划分，最终除了数据库和数据结构等一些耦合度较小的代码可以单独提出来以外，其他代码（基本包含了对文件系统的所有操作）都放在了一个文件里，从这一点看，技术选型是存在一些问题的。

而至于该文件系统，我们认为它还有很多提升空间，无论从效率还是空间开销上都可以继续优化、精益求精。

综上所述，本次课程设计不仅帮助我巩固了课堂所学的知识、锻炼了数据结构设计、算法设计等基本功，还对于我学习如何解决问题、如何查阅资料、如何将所学过的知识迁移到其他可用的地方提供了很大的帮助，本次课程设计的综合性很强，是一次很好的锻炼机会，使我受益良多。

最后还要感谢老师的认真教学和耐心指导，感谢队友的全力配合和努力奋斗。

**戚子强**

在本次操作系统课程设计中，我们原本的打算是用UWP做一个漂亮的外观界面，然后底层是尽可能照顾速度，可以牺牲一些空间来换取时间上的优势。我们的分工是底层服务与顶层功能实现分离，分别由一个人完成，这样既降低了代码耦合又可以减少不必要的沟通问题，虽然之后也是这样做的，但是这个课程设计很难解耦，导致最终效果不如设想的那么美好，但完全可以接受。我主要是负责底层实现，底层实现函数介于物理磁盘与顶层功能函数之间，屏蔽了对物理磁盘的操作，封装操作逻辑，给顶层功能函数提供一个函数接口，使物理磁盘对顶层实现函数透明化，将文件系统代码解耦。

在底层的数据结构和磁盘分区设计时，我们考虑到目前磁盘和RAM容量已经比较大，内存完全可以容纳所有i节点（设计的文件系统每个磁盘块1024B，文件系统大小为4GB，这受限于使用了unsigned int寻址，使用unsigned long可拓展至最大2^34 GB），即使i节点信息区域满，总大小约为4MB,我们认为也是完全可以接收的。此外，我们去掉了EXT2中“目录”的设定，转而将目录层级逻辑放到了i节点里，为了节省空间，约定：若i节点结构中的type字段是0，则为文件夹，其next\_addr列表里存放该目录下一级的文件和文件夹的i节点ID；type为1，则为文件，其next\_addr列表里存放的是该文件磁盘地址号。这样一来，极大减少了i节点的大小，同时提高了速度，不至于每次都用到了一个i节点，发现内存里没有再去磁盘寻找，那样的话会耗费许多IO时间。

另外一个比较深的体会是：作为底层服务提供者，自己写的底层函数有的可能会被上层调用非常多的次数，而有的只是很少次数的调用。在这里就需要优先对调用频繁的底层函数做好优化，同时确保其每次运行都是正确的，不能有丝毫的漏洞存在，最终目标是使其能够以最快的速度返回正确的结果。否则将会对整个项目造成难以估量的损失和代价。

最后，本次操作系统课程设计不仅培养了我们独立思考、动手操作的能力，在各种其它能力上也都有了提高。更重要的是，我们学会了很多学习的方法，而这是以后最实用的。要面对各种不同的问题和挑战，只有不断的学习、实践，再学习、再实践，有什么不懂不明白的地方要及时请教或上网查询，只要认真钻研，动脑思考，动手实践，就没有弄不懂的知识。在这个过程中，不仅可以巩固了以前所学过的知识，而且学到了很多在书本上所没有学到过的知识。通过这次课程设计使我懂得了理论与实际相结合是很重要的，只有理论知识是远远不够的，只有把所学的理论知识与实践相结合起来才可以。所谓“纸上得来终觉浅，绝知此事要躬行。”