

TONGJI UNIVERSITY

《操作系统课程设计》

实验报告

|  |  |
| --- | --- |
| 姓 名 |  |
| 学 号 |  |
| 学院（系） |  |
| 专 业 |  |
| 指导教师 |  |
| 日 期 |  |

**目录**

[模拟二级文件系统软件需求规格说明文档 4](#_Toc10742482)

[1.引言 4](#_Toc10742483)

[1.1 目的 4](#_Toc10742484)

[1.2 范围 4](#_Toc10742485)

[1.3 参考文献 4](#_Toc10742486)

[2.总体描述 4](#_Toc10742487)

[2.1 软件功能 4](#_Toc10742488)

[2.2 约束 4](#_Toc10742489)

[2.3 假设和依赖 4](#_Toc10742490)

[3.详细需求描述 4](#_Toc10742491)

[3.1 对外接口需求 4](#_Toc10742492)

[3.1.1 用户界面 4](#_Toc10742493)

[3.1.2 通信接口 4](#_Toc10742494)

[3.2 功能需求 4](#_Toc10742495)

[3.2.1 格式化文件卷 4](#_Toc10742496)

[3.2.2 列出目录 5](#_Toc10742497)

[3.2.3 创建目录 5](#_Toc10742498)

[3.2.4 创建文件 5](#_Toc10742499)

[3.2.5 打开文件 5](#_Toc10742500)

[3.2.6 关闭文件 5](#_Toc10742501)

[3.2.7 读文件 5](#_Toc10742502)

[3.2.8 写文件 5](#_Toc10742503)

[3.2.9 定位文件读写指针 5](#_Toc10742504)

[3.2.10删除文件 5](#_Toc10742505)

[3.3 非功能需求 6](#_Toc10742506)

[3.3.1 可靠性 6](#_Toc10742507)

[3.3.2 易用性 6](#_Toc10742508)

[3.3.3 数据需求 6](#_Toc10742509)

[3.3.3.1 数据定义 6](#_Toc10742510)

[3.3.3.2 默认数据 6](#_Toc10742511)

[模拟二级文件系统体系结构描述文档 7](#_Toc10742512)

[1.引言 7](#_Toc10742513)

[1.1 编制目的 7](#_Toc10742514)

[1.2 参考文献 7](#_Toc10742515)

[2.逻辑视角 7](#_Toc10742516)

[3.接口视角 8](#_Toc10742517)

[3.1 层次的职责 8](#_Toc10742518)

[3.2 展示层的分解 8](#_Toc10742519)

[3.2.1 展示层模块的职责 9](#_Toc10742520)

[3.2.2 展示层类的接口规范 9](#_Toc10742521)

[3.3 逻辑层的分解 10](#_Toc10742522)

[3.3.1 逻辑层类的职责 10](#_Toc10742523)

[3.3.2 逻辑层类的接口规范 10](#_Toc10742524)

[3.4 数据层的分解 11](#_Toc10742525)

[3.4.1 数据层类的职责 12](#_Toc10742526)

[4.信息视角 12](#_Toc10742527)

[4.1 数据持久化对象 12](#_Toc10742528)

[模拟二级文件系统软件详细设计文档 13](#_Toc10742529)

[1.引言 13](#_Toc10742530)

[1.1 编制目的 13](#_Toc10742531)

[1.2 参考资料 13](#_Toc10742532)

[2.产品概述 13](#_Toc10742533)

[3.结构视角 13](#_Toc10742534)

[3.1 展示层的分解 13](#_Toc10742535)

[3.1.1 MainForm类 13](#_Toc10742536)

[3.1.2 InputNameForm类 14](#_Toc10742537)

[3.2 逻辑层的分解 14](#_Toc10742538)

[3.2.1 FileManager类 14](#_Toc10742539)

[3.3 数据层的分解 33](#_Toc10742540)

[模拟二级文件系统软件测试分析报告 35](#_Toc10742541)

[1.引言 35](#_Toc10742542)

[1.1 编制目的 35](#_Toc10742543)

[1.2 参考文献 35](#_Toc10742544)

[2.测试用例 35](#_Toc10742545)

[2.1 基础功能测试 35](#_Toc10742546)

[2.2 大文件存储测试 35](#_Toc10742547)

[2.3 多文件存储测试 35](#_Toc10742548)

[3.调试过程 36](#_Toc10742549)

[3.1 基础功能测试 36](#_Toc10742550)

[3.2 大文件存储测试 40](#_Toc10742551)

[3.3 多文件存储测试 43](#_Toc10742552)

[模拟二级文件系统用户使用说明 49](#_Toc10742553)

[1.安装说明 49](#_Toc10742554)

[2.用户界面说明 49](#_Toc10742555)

[3.功能使用说明 50](#_Toc10742556)

[3.1 文件卷管理功能 50](#_Toc10742557)

[3.2 文件/文件夹创建功能 51](#_Toc10742558)

[3.3 文件/文件夹删除功能 51](#_Toc10742559)

[3.4 文件/文件夹打开/关闭功能 51](#_Toc10742560)

[3.5 列出目录功能 51](#_Toc10742561)

[3.6 文件读写指针定位及读写功能 51](#_Toc10742562)

[课程设计总结 52](#_Toc10742563)

模拟二级文件系统软件需求规格说明文档

# 1.引言

## 1.1 目的

本文档描述了模拟二级文件系统的功能需求和非功能需求。模拟二级文件系统的实现与验证工作都以此文档为依据。

## 1.2 范围

模拟二级文件系统是本次课程设计要求开发的软件系统，开发的目标是实现一个类似UNIX系统文件系统的模拟文件系统。

## 1.3 参考文献

1）IEEE标准

2）丁二玉,刘钦.软件开发的技术基础

3）操作系统课程讲义

# 2.总体描述

## 2.1 软件功能

SF1:实现基本的磁盘文件结构，能够保存不同类型大小的文件，并实现目录结构。

SF2:实现基本的文件系统操作，能够进行格式化、创建目录、列出目录、创建文件、打开文件、关闭文件、读写文件、随机读写、删除文件的功能。

SF3:提供一个友好的用户界面，供用户进行上述操作。

## 2.2 约束

CON1: 系统将运行在具有C++运行库的Windows 10操作系统上。

CON2:系统不使用控制台界面，而是图形界面。

CON3:在开发中，需要提交软件需求规格说明文档、设计描述文档和测试报告。

## 2.3 假设和依赖

AE1:用户了解用户界面的使用方法。

AE2:用户不会进行非法操作。

# 3.详细需求描述

## 3.1 对外接口需求

3.1.1 用户界面

UI:系统应采用Windows Form风格的界面，帮助用户进行各项操作。

3.1.2 通信接口

CI:系统使用二进制文件流对文件卷进行操作。

## 3.2 功能需求

3.2.1 格式化文件卷

（1）特性描述

用户发送格式化文件卷指令时，应当重新构建文件系统的控制结构，即将文件系统完全重置并使其处于可以接受命令的状态。

3.2.2 列出目录

（1）特性描述

调用此命令时，应当列出当前文件系统的结构。即列出文件系统内所有的文件夹和文件。

3.2.3 创建目录

（1）特性描述

调用此命令时，应当根据提供的目录名在相应的位置创建指定文件夹。若上级目录不存在，应当提示错误。

3.2.4 创建文件

（1）特性描述

调用此命令时，应当根据提供的路径名在相应的位置创建指定文件。若路径不存在，应当提示错误。

3.2.5 打开文件

（1）特性描述

调用此命令时，应当根据提供的路径名打开相应文件。若路径不存在，应当提示错误。

3.2.6 关闭文件

（1）特性描述

调用此命令时，应当关闭当前打开的文件。若当前没有打开文件，应当提示错误。

3.2.7 读文件

（1）特性描述

调用此命令时，应当根据提供的读取长度从当前打开文件的文件指针处读取相应长度的数据并导出到文件卷外。若当前没有打开文件或长度过长，应当提示错误。

3.2.8 写文件

（1）特性描述

调用此命令时，应当将提供的数据写入当前打开的文件目前的文件指针处。若当前没有打开文件或写入数据过大，应当提示错误。

3.2.9 定位文件读写指针

（1）特性描述

调用此命令时，应当移动读写指针到指定的位置以实现随机读写的功能。若当前没有打开文件或位置非法，应当提示错误。

3.2.10删除文件

（1）特性描述

调用此命令时，应当根据提供的路径从文件系统中删除指定的文件并释放空间。若指定的文件不存在，应当提示错误。

## 3.3 非功能需求

3.3.1 可靠性

Reliability1:系统应当对用户可能的操作都作出正确的应对。

Reliability2:系统应当防止数据溢出造成文件系统结构被破坏。

3.3.2 易用性

Usability1:图形界面的字体大小应设置到能在1米之内看清。

Usability2:图形界面应该足够友好，让用户能在2分钟内学会使用。

3.3.3 数据需求

3.3.3.1 数据定义

DR1:系统使用的文件卷以名称为Disk.img的文件在系统内存储。

DR2:系统若产生错误应输出错误信息到磁盘内。

3.3.3.2 默认数据

Default1:默认的磁盘数据块大小为512字节。

Default2:默认的磁盘块总数量为262144个，即文件卷大小为128MB。

Default3:默认的SuperBlock区占用2个磁盘块

Default4:默认的Inode区占用1024个磁盘块，每个DiskInode占用64字节，因此文件系统的最大文件数量为8192个

Default5:默认的目录结构大小为32字节，其中路径名占用28字节，在本文件系统采用Unicode编码的情况下路径名最长含14个字符。

Defalult6:文件系统索引表长度默认为11，其中0~5位置用于直接索引，6~7位置用于一次间接索引，8~9位置用于二次间接索引，10位置用于三次间接索引。四种索引方式对应的文件大小分别最大为3KB，131KB，16.13MB和1040.13MB的文件。

模拟二级文件系统体系结构描述文档

# 1.引言

## 1.1 编制目的

本报告详细完成对模拟二级文件系统的概要设计。达到指导详细设计和开发的目的，同时实现和测试人员及用户的沟通。

## 1.2 参考文献

1）IEEE标准

2）丁二玉,刘钦.软件开发的技术基础

3）操作系统课程讲义

4）微软官方.net文档

# 2.逻辑视角

在模拟二级文件系统中，我使用了分层体系结构风格，将系统分为3层（展示层、逻辑层、数据层）能够很好地示意整个高层抽象。展示层包括GUI页面的实现、逻辑层包含所有文件系统功能的实现、数据层包含了文件系统持久数据存储结构的实现。分层体系结构的逻辑视角和逻辑设计方案图2.1和图2.2所示。

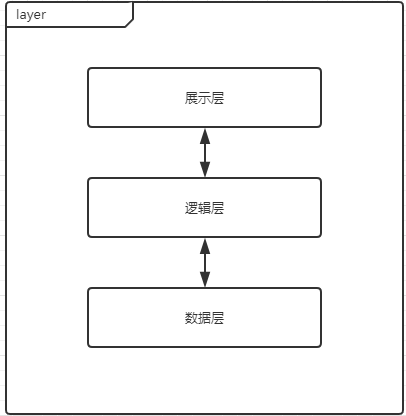


图2.1 分层体系结构的逻辑视角

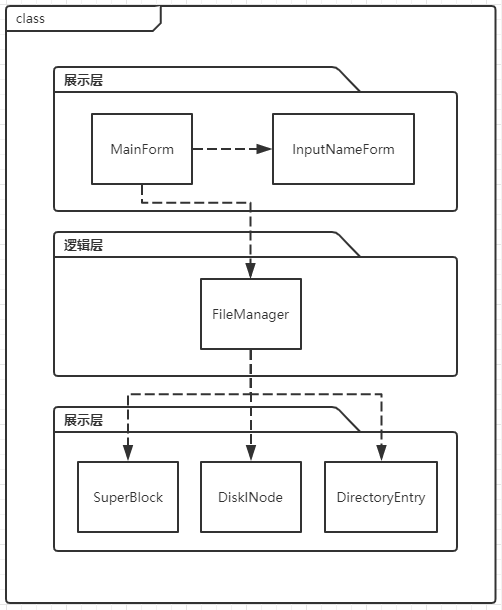


图2.2 软件体系结构逻辑设计方案

# 3.接口视角

## 3.1 层次的职责

系统各个层次的职责如表3.1所示。

表3.1 各层的职责

|  |  |
| --- | --- |
| 层 | 职责 |
| 展示层 | 接收用户命令并调用相应逻辑层方法，并将运行结果反馈给用户。 |
| 逻辑层 | 被展示层调用后，与数据层进行交互，进行相应命令的处理。 |
| 数据层 | 负责实现文件系统的存储结构，为逻辑层的实现打下基础。 |

## 3.2 展示层的分解

根据需求，系统需要4个用户界面，系统主界面、读取界面、保存界面、文件名输入界面。界面跳转图如图3.1所示。

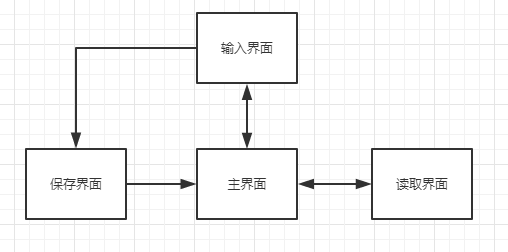


图3.1 用户界面跳转

需要额外编写类实现的界面仅有主界面和读取界面，读取、保存界面可以使用System.Windows.Forms命名空间内的OpenFileDialog和SaveFileDialog直接实现。因此展示层的类设计如图3.2所示。

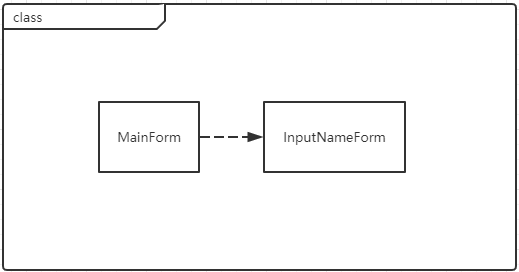


图3.2 展示层的类设计

3.2.1 展示层模块的职责

展示层各类的职责如表3.1所示

表3.1 展示层各类的职责

|  |  |
| --- | --- |
| 类 | 职责 |
| MainForm | 负责实现主界面的显示和对逻辑层相应功能的调用 |
| InputNameForm | 负责实现输入界面的显示并捕获用户的输入 |

3.2.2 展示层类的接口规范

展示层各个类的对外接口规范如表3.2所示。

表3.2 展示层类的对外接口规范

|  |  |  |
| --- | --- | --- |
| MainFrom | 语法 | MainFrom() |
| 前置条件 | 无 |
| 后置条件 | 初始化主界面各控件 |
| InputNameForm | 语法 | InputNameForm() |
| 前置条件 | 无 |
| 后置条件 | 初始化输入界面各控件 |

展示层实现所依赖的命名空间及其提供的服务如表3.3所示。

表3.3 展示层类需要的服务

|  |  |
| --- | --- |
| 命名空间 | 服务 |
| System.Windows.Forms | 实现界面所需的各控件以及读取、保存窗口 |
| System.IO | 提供读写磁盘文件功能 |
| System.Collections.Generic | 提供程序设计所需的基础数据结构（如队列等） |

## 3.3 逻辑层的分解

逻辑层使用类FileManager抽象包含了文件系统数据操作的所有功能。逻辑层的设计如图3.3所示。

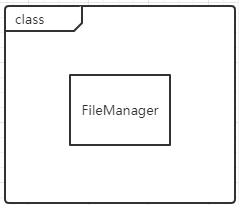


图3.3 逻辑层的类设计

3.3.1 逻辑层类的职责

逻辑层类的职责如表3.4所示。

表3.4 逻辑层类的职责

|  |  |
| --- | --- |
| 类 | 职责 |
| FileManager | 负责实现对数据层静态数据的所有操作 |

3.3.2 逻辑层类的接口规范

逻辑层类对展示层开放的所有接口规范如表3.5所示。

表3.5 逻辑层类的对外接口规范

|  |  |  |
| --- | --- | --- |
| FileManager.LoadImage | 语法 | public void LoadImage(string path) |
| 前置条件 | path对应的文件在磁盘内存在 |
| 后置条件 | 保存对之前文件卷的更改并载入新的文件卷 |
| FileManager.ImageFormatted | 语法 | public bool ImageFormatted() |
| 前置条件 | 已载入文件卷 |
| 后置条件 | 判断该文件卷是否已经被格式化并返回结果 |
| FileManager.FormatImage | 语法 | public bool ImageFormatted() |
| 前置条件 | 已载入文件卷 |
| 后置条件 | 格式化当前载入的文件卷 |
| FileManager.LoadSuperBlockInstance | 语法 | public void LoadSuperBlockInstance() |
| 前置条件 | 文件卷已被格式化 |
| 后置条件 | 读取该文件卷的超级块到内存中 |
| FileManager.OpenFile | 语法 | public int OpenFile(string path) |
| 前置条件 | 文件卷已被格式化 |
| 后置条件 | 尝试打开给定路径的文件并返回其父目录的Inode编号 |
| FileManager.CloseFile | 语法 | public void CloseFile() |
| 前置条件 | 文件卷已被格式化 |
| 后置条件 | 保存对当前文件的更改后关闭文件 |
| FileManager.Seek | 语法 | public void Seek(int offset) |
| 前置条件 | 已打开文件且offset没有超出文件范围 |
| 后置条件 | 修改文件指针到指定的位置 |
| FileManager.ReadFile | 语法 | public int ReadFile(byte[] read\_buffer, int offset, int need\_count) |
| 前置条件 | 已打开文件且read\_buffer足够容纳读出的字节 |
| 后置条件 | 从当前文件读取指定数量的字节后从offset开始置入缓冲区，返回实际读出的字节数量 |
| FileManager.WriteFile | 语法 | public int WriteFile(byte[] write\_buffer, int offset, int need\_count) |
| 前置条件 | 已打开文件且write\_buffer包含了足够写入的字节 |
| 后置条件 | 向当前文件写入缓冲区中从offset开始指定数量的字节，返回实际写入的字节数量 |
| FileManager.CreateFile | 语法 | public void CreateFile(string path, bool allow\_overlay, DiskInode.FileType type) |
| 前置条件 | 有空闲Inode，给出的路径正确且在文件重复时allow\_overlay为真 |
| 后置条件 | 创建指定位置的文件，若文件已存在且允许覆盖，将其覆盖 |
| FileManager.DeleteFile | 语法 | public void DeleteFile(string path) |
| 前置条件 | path指向的文件在文件卷内存在 |
| 后置条件 | 删除指定的文件，释放其占用的所有数据块、间接索引块和Inode，并在上级目录中删去其目录 |
| FileManager.ReadFileType | 语法 | public DiskInode.FileType ReadFileType(int inode\_no) |
| 前置条件 | inode\_no没有超出Inode区范围 |
| 后置条件 | 读出指定Inode对应的文件类型并返回 |
| FileManager.GetInodeInstance | 语法 | public DiskInode GetInodeInstance() |
| 前置条件 | 无 |
| 后置条件 | 返回当前打开的内存Inode实例 |
| FileManager.SaveAllChanges | 语法 | public void SaveAllChanges() |
| 前置条件 | 无 |
| 后置条件 | 保存所有更改 |

逻辑层实现依赖的命名空间及其提供的服务如表3.6所示。

表3.6 逻辑层类需要的服务

|  |  |
| --- | --- |
| 命名空间 | 服务 |
| System.IO | 提供读写磁盘文件功能 |
| System.Collections.Generic | 提供程序设计所需的数据结构（如队列等） |

## 3.4 数据层的分解

数据层负责实现数据静态保存结构的抽象及其在内存中的表示，方便对文件卷数据结构的管理。数据层类的设计如图3.4所示。

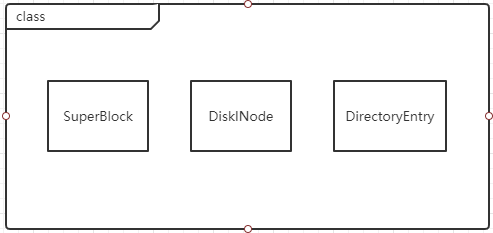


图3.4 数据层的类设计

3.4.1 数据层类的职责

数据层类的职责如表3.7所示

表3.7 数据层类的职责

|  |  |
| --- | --- |
| 类 | 职责 |
| SuperBlock | 负责实现管理文件系统空余空间的超级块的数据结构 |
| DiskInode | 负责实现管理文件的结点的数据结构 |
| DirectoryEntry | 负责实现目录文件目录项的数据结构 |

# 4.信息视角

## 4.1 数据持久化对象

系统使用img文件作为数据持久化保存文件，根据需求分析中的默认数据，文件的具体结构设计如图4.1所示。

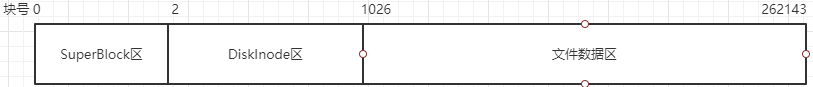


图4.1 Img文件的结构设计

模拟二级文件系统软件详细设计文档

# 1.引言

## 1.1 编制目的

本报告详细完成对类C编译器的详细设计，达到指导后续软件构造的目的，同时实现和测试人员及用户的沟通。

本报告面向开发人员，测试人员及最终用户而编写，是了解系统的导航。

## 1.2 参考资料

1）IEEE标准

2）丁二玉,刘钦.软件开发的技术基础

3）操作系统课程讲义

4）微软官方C# .net文档

# 2.产品概述

参考模拟二级文件系统软件需求规格说明文档中对产品的概括描述。

# 3.结构视角

## 3.1 展示层的分解

展示层的类设计图参见软件体系结构文档图3.2。

3.1.1 MainForm类

（1）类概述

MainForm类承担的需求参见需求规格说明文档功能需求及相关非功能需求。

MainForm类的职责及接口参见软件系统结构描述文档表3.1和表3.2。

（2）外观设计

MainForm的外观设计使用标准Windows窗口的外观设计，窗口样式如图3.1所示。

图3.1 MainForm的外观设计

（3）类内部函数接口规范

类内部供调用的函数接口如表3.1所示。类中其余函数为各控件的事件响应函数，由系统生成代码调用，因此略去接口规范，相应功能可以参考软件使用说明。

表3.1 MainForm类内部函数接口

|  |  |  |
| --- | --- | --- |
| MainForm.ListFile | 语法 | private void ListFile(TreeNode node) |
| 前置条件 | node是文件夹结点 |
| 后置条件 | 列出指定结点代表的目录下所有文件和文件夹，并生成结点作为输入结点的子结点 |
| MainForm.RemoveFile | 语法 | private void RemoveFile(TreeNode node) |
| 前置条件 | node不为null |
| 后置条件 | 递归删除该节点路径之后的所有文件 |
| MainForm.CheckSpaceUsage | 语法 | private void CheckSpaceUsage() |
| 前置条件 | 已载入文件卷并读入超级块 |
| 后置条件 | 读取文件卷的数据区和Inode区使用情况 |

3.1.2 InputNameForm类

（1）类概述

InputNameForm类承担的需求参见需求规格说明文档功能需求及相关非功能需求。

InputNameForm类的职责及接口参见软件系统结构描述文档表3.1和表3.2。

（2）外观设计

InputNameForm的外观设计如图3.2所示。

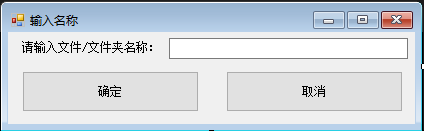


图3.2 InputNameForm的外观设计

## 3.2 逻辑层的分解

3.2.1 FileManager类

（1）类概述

FileManager类承担的需求参见需求规格说明文档功能需求及相关非功能需求。

FileManager类的职责及接口参见软件系统结构描述文档表3.4和表3.5。

（2）类内部变量设计

类内部变量设计及其注释如代码3.1所示。

代码3.1 FileManager类的内部变量说明

public const int NULL\_NO = -1; //代表无效的编号

public const int BLOCK\_SIZE = 512; //物理块大小

public const int INODE\_PER\_BLOCK = BLOCK\_SIZE / DiskInode.INODE\_SIZE; //每个物理块能存放的INODE数量

public const int MAX\_INODE = 8192; //最多DiskInode数量

public const int MAX\_DATA\_BLOCK = 261118; //最多一般数据块数量

public const int FREE\_BLOCK\_GROUP\_SIZE = 128; //空闲盘块组每组数量

public const int INODE\_BEGIN = SuperBlock.FULL\_SIZE; //INode区起始物理地址

public const int DATA\_BEGIN = INODE\_BEGIN + MAX\_INODE \* DiskInode.INODE\_SIZE; //数据区起始物理地址，即Inode区起始地址加上Inode区所占空间

public const int IMAGE\_SIZE = BLOCK\_SIZE \* MAX\_DATA\_BLOCK + DATA\_BEGIN; //默认文件卷大小,当前设置下为128MB

private FileStream CurOpenFileStream;

//当前打开文件卷的流

private int FilePtr;

//文件读写指针

private DiskInode CurFileInode;

//当前打开的文件对应的Inode

private int CurFileInodeNo;

//当前打开文件Inode在Inode区的编号

public SuperBlock SuperBlockInstance;

//内存中的超级块实例

（3）类内部函数接口规范

类的外部接口参见软件系统结构描述文档表3.5，类的内部函数接口参见表3.2。

表3.2 FileManager类内部函数接口

|  |  |  |
| --- | --- | --- |
| FileManager.AllocBlock | 语法 | private int AllocBlock() |
| 前置条件 | 已读取超级块，有空余数据块 |
| 后置条件 | 获取一个空闲数据块并返回块号 |
| FileManager.FreeBlock | 语法 | private void FreeBlock(int block\_no) |
| 前置条件 | 已读取超级块，block\_no合法 |
| 后置条件 | 释放指定数据块 |
| FileManager.AllocInode | 语法 | private int AllocInode() |
| 前置条件 | 已读取超级块且有空余Inode |
| 后置条件 | 获取一个空闲Inode并返回Inode编号 |
| FileManager.FreeInode | 语法 | private void FreeInode(int inode\_no) |
| 前置条件 | 已读取超级块，inode\_no合法 |
| 后置条件 | 释放指定Inode |
| FileManager.Find | 语法 | private int Find(int index\_inode, string name) |
| 前置条件 | index\_inode对应一个目录文件 |
| 后置条件 | 在指定inode对应的目录文件内寻找指定名字的文件或目录，返回其inode编号 |
| FileManager.ReadDiskInode | 语法 | private DiskInode ReadDiskInode(int inode\_no) |
| 前置条件 | 已载入文件卷 |
| 后置条件 | 读出指定位置的inode并返回 |
| FileManager.WriteDiskInode | 语法 | private void WriteDiskInode(DiskInode inode, int inode\_no) |
| 前置条件 | 已载入文件卷，inode\_no合法 |
| 后置条件 | 将inode写入到inode区指定位置 |
| FileManager.GetPhyBlockNo | 语法 | private int GetPhyBlockNo(int logic\_no) |
| 前置条件 | 已打开文件，logic\_no没有超出文件大小范围 |
| 后置条件 | 给定一个逻辑块号，获取当前打开文件中对应的物理块号并返回 |
| FileManager.ExpandFile | 语法 | private void ExpandFile() |
| 前置条件 | 当前打开文件占用的所有数据块空间已经用尽，系统有足够剩余空间 |
| 后置条件 | 为当前打开文件扩容一个数据块，并设置相应索引结构 |
| FileManager.AbsorbFile | 语法 | private void AbsorbFile() |
| 前置条件 | 当前打开文件末尾的数据块已不再被占用 |
| 后置条件 | 释放文件末尾的数据块，若有间接索引块因此不再被占用，将其一并释放 |
| FileManager.UpdateSuperBlock | 语法 | private void UpdateSuperBlock() |
| 前置条件 | 已载入文件卷，内存中有SuperBlock实例 |
| 后置条件 | 将内存中的SuperBlock更新到文件卷中 |

（4）算法设计

①空闲盘块管理算法：系统采用分组链式索引法对空闲数据块进行管理，详细算法参见操作系统讲义P331~P333。有一点微小的区别在于本系统去掉了数据区空闲块中索引表的nfree值，改为每一个空闲数据块存储128个索引，这可以略微减少I/O次数。根据这一算法编写出AllocBlock函数和FreeBlock函数代码如代码3.2所示。

private int AllocBlock() //获取一个空闲数据块

{

if (SuperBlockInstance.FreeBlockIndex[SuperBlockInstance.FreeBlockCount - 1] == SuperBlock.NULL\_NO) //若即将分配的盘块是代表无效的盘块，说明所有空闲盘块已被分配完，抛出异常

throw new InvalidOperationException("磁盘无空余空间！");

int ReturnValue = SuperBlockInstance.FreeBlockIndex[--SuperBlockInstance.FreeBlockCount]; //分配的盘块为表尾盘块，表长度-1

if (SuperBlockInstance.FreeBlockCount == 0) //若当前表已空，将即将分配空闲块内的表载入超级块

{

byte[] ReadBuffer = new byte[FREE\_BLOCK\_GROUP\_SIZE \* sizeof(int)];

CurOpenFileStream.Seek(ReturnValue \* BLOCK\_SIZE + DATA\_BEGIN, SeekOrigin.Begin);

CurOpenFileStream.Read(ReadBuffer, 0, FREE\_BLOCK\_GROUP\_SIZE \* sizeof(int));

for (int i = 0; i < FREE\_BLOCK\_GROUP\_SIZE; i++)

SuperBlockInstance.FreeBlockIndex[i] = BitConverter.ToInt32(ReadBuffer, i \* sizeof(int));

SuperBlockInstance.FreeBlockCount = FREE\_BLOCK\_GROUP\_SIZE; //载入完毕，设置直接管理的空闲块为一整组的数量

}

SuperBlockInstance.BlockCount--;

return ReturnValue; //返回该空闲块

}

private void FreeBlock(int block\_no) //释放指定数据块

{

if (block\_no <= 0) //0号数据块用于存储根目录文件，无法被释放

return;

CurOpenFileStream.Seek(DATA\_BEGIN + block\_no \* BLOCK\_SIZE, SeekOrigin.Begin);

for (int i = 0; i < BLOCK\_SIZE; i++) //用0覆写数据块

CurOpenFileStream.WriteByte(0x0);

if (SuperBlockInstance.FreeBlockCount == FREE\_BLOCK\_GROUP\_SIZE) //若索引表已满，将索引表存入释放的数据块，并将超级块索引表设置为只有新释放的块

{

long WriteOffset = DATA\_BEGIN + block\_no \* BLOCK\_SIZE; //设置写入开始位点

for (int i = 0; i < SuperBlockInstance.FreeBlockIndex.Length; i++)

{

CurOpenFileStream.Seek(WriteOffset, SeekOrigin.Begin);

CurOpenFileStream.Write(BitConverter.GetBytes(SuperBlockInstance.FreeBlockIndex[i]), 0, sizeof(int));

WriteOffset += sizeof(int);

SuperBlockInstance.FreeBlockIndex[i] = 0; //空闲的索引表位置赋值为0

}

SuperBlockInstance.FreeBlockIndex[0] = block\_no; //将索引表0号位置设置为新释放的块

SuperBlockInstance.FreeBlockCount = 1; //超级块直接管理1个空闲块

}

else //否则，在索引表尾加入该数据块即可

{

SuperBlockInstance.FreeBlockIndex[SuperBlockInstance.FreeBlockCount] = block\_no;

SuperBlockInstance.FreeBlockCount++;

}

SuperBlockInstance.BlockCount++;

}

代码3.2 空闲数据块管理函数代码

②空闲Inode管理算法：本系统仅在超级块中直接管理123个空闲Inode。当需要分配Inode时，直接从空闲Inode表尾取出一个分配，若表空，则去Inode取寻找最多123个空闲Inode填充到空闲Inode表内，若搜索完整个Inode表都没有空闲Inode，说明Inode已分配完。当释放Inode时，将其置入空闲Inode表，若表满，则任由其散落在Inode区。根据上述算法，编写出AllocInode和FreeInode函数代码如代码3.3所示。

private int AllocInode() //获取一个空闲Inode

{

if (SuperBlockInstance.FreeInodeCount == 0) //若空闲Inode索引表已空，需要去Inode区寻找散落的空闲Inode并置入索引表

{

for (int i = 0; i < MAX\_INODE; i++) //逐个遍历Inode区的Inode

{

CurOpenFileStream.Seek(INODE\_BEGIN + i \* DiskInode.INODE\_SIZE, SeekOrigin.Begin);

if (!Convert.ToBoolean(CurOpenFileStream.ReadByte())) //从Inode头读入Occupied变量的字节，若为否，则Inode未被占用

SuperBlockInstance.FreeInodeIndex[SuperBlockInstance.FreeInodeCount++] = i;

if (SuperBlockInstance.FreeInodeCount == SuperBlock.INODE\_INDEX\_SIZE) //若索引表已满，结束搜索

break;

}

}

if (SuperBlockInstance.FreeInodeCount == 0) //若仍无空闲Inode,说明整个系统已经没有空闲Inode，抛出异常

throw new InvalidOperationException("文件数量已到达上限！");

SuperBlockInstance.InodeCount--;

return SuperBlockInstance.FreeInodeIndex[--SuperBlockInstance.FreeInodeCount]; //返回索引最后一项，表长自减

}

private void FreeInode(int inode\_no) //释放指定Inode

{

if (SuperBlockInstance.FreeInodeCount != SuperBlock.INODE\_INDEX\_SIZE) //若索引表不满，将其置入索引表。否则任由其散落在Inode区

{

SuperBlockInstance.FreeInodeIndex[SuperBlockInstance.FreeInodeCount] = inode\_no;

SuperBlockInstance.FreeInodeCount++;

}

SuperBlockInstance.InodeCount++;

}

代码3.3 空闲Inode管理函数代码

③文件扩容算法：当写入数据写满一个数据块后需要继续写入时，需要为文件扩容一个数据块以供写入。扩容算法流程图如图3.3所示。

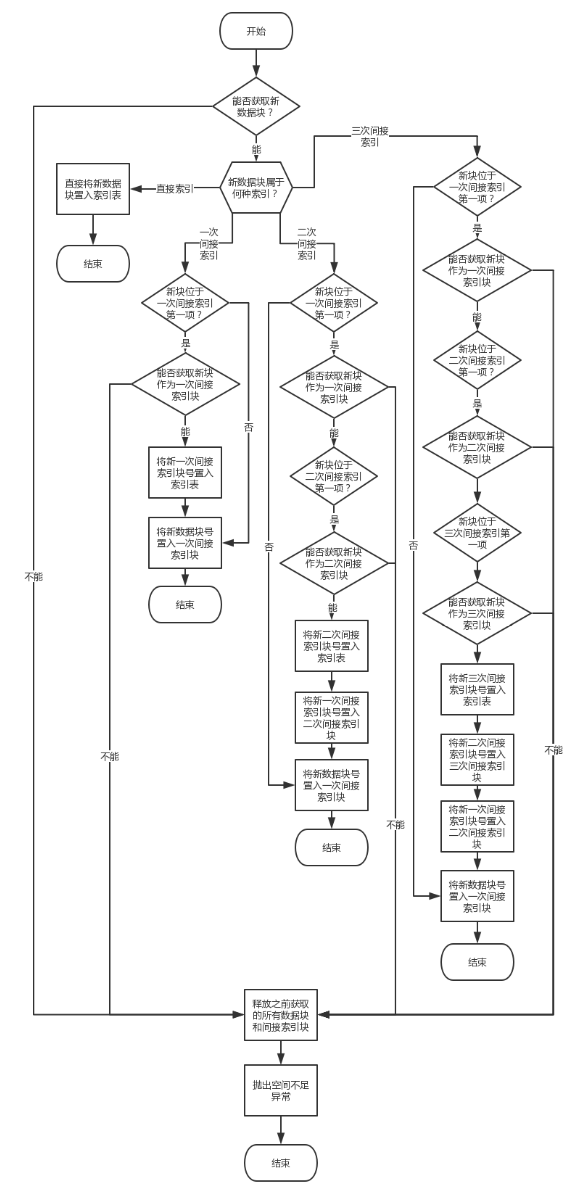


图3.3 文件扩容算法流程图

根据以上算法，编写出文件扩容函数Expand如代码3.4所示。

private void ExpandFile() //当文件所有数据块空间已经用尽时，为当前打开文件扩容一个数据块

{

if (CurFileInode.Size % BLOCK\_SIZE != 0) //若仍有剩余空间，抛出异常

throw new NotSupportedException("当前文件占用数据块空间尚未用尽！");

int LogicBlockNo = CurFileInode.Size / BLOCK\_SIZE; //新数据块的逻辑块号

int NewBlockNo; //新数据块的块号

/\* 尝试获取新数据块以存放数据 \*/

try

{

NewBlockNo = AllocBlock();

}

catch

{

throw;

}

if (LogicBlockNo < DiskInode.MAX\_SMALL\_FILE) //若新块属于直接索引，数据块直接置入索引表

CurFileInode.IndexTable[LogicBlockNo] = NewBlockNo;

else if (LogicBlockNo < DiskInode.MAX\_MEDIUM\_FILE) //若新块属于一次间接索引，找到一次间接索引块并置入

{

int SingleIndirectNo; //一次间接索引块号

if (((LogicBlockNo - DiskInode.MAX\_SMALL\_FILE) % DiskInode.INDEX\_PER\_BLOCK) == 0) //若新块位于一次间接索引表的第一项，需要再申请一个数据块作为间接索引块

{

try

{

SingleIndirectNo = AllocBlock();

}

catch //若获取间接索引块失败，将之前获取的数据块释放

{

FreeBlock(NewBlockNo);

throw;

}

CurFileInode.IndexTable[(LogicBlockNo - DiskInode.MAX\_SMALL\_FILE) / DiskInode.INDEX\_PER\_BLOCK + DiskInode.SINGLE\_INDIRECT\_BEGIN] = SingleIndirectNo; //将一次间接索引块置入索引表

}

else //否则直接从索引表获取一次间接索引块号

SingleIndirectNo = CurFileInode.IndexTable[(LogicBlockNo - DiskInode.MAX\_SMALL\_FILE) / DiskInode.INDEX\_PER\_BLOCK + DiskInode.SINGLE\_INDIRECT\_BEGIN];

int SingleIndirectIndex = (LogicBlockNo - DiskInode.MAX\_SMALL\_FILE) % DiskInode.INDEX\_PER\_BLOCK; //索引在索引块的位置

CurOpenFileStream.Seek(DATA\_BEGIN + SingleIndirectNo \* BLOCK\_SIZE + SingleIndirectIndex \* sizeof(int), SeekOrigin.Begin); //设置要写入索引的位置

CurOpenFileStream.Write(BitConverter.GetBytes(NewBlockNo), 0, sizeof(int)); //写入新获取的数据块到相应索引位置

CurOpenFileStream.Flush();

}

else if (LogicBlockNo < DiskInode.MAX\_LARGE\_FILE) //若新块属于二次间接索引

{

int SingleIndirectNo; //一次间接索引块号

int DoubleIndirectNo = CurFileInode.IndexTable[((LogicBlockNo - DiskInode.MAX\_MEDIUM\_FILE) / (DiskInode.INDEX\_PER\_BLOCK \* DiskInode.INDEX\_PER\_BLOCK)) + DiskInode.DOUBLE\_INDIRECT\_BEGIN]; //二次间接索引块号,默认从索引表取

int DoubleBlockIndex = ((LogicBlockNo - DiskInode.MAX\_MEDIUM\_FILE) % (DiskInode.INDEX\_PER\_BLOCK \* DiskInode.INDEX\_PER\_BLOCK)) / DiskInode.INDEX\_PER\_BLOCK; //物理块对应一次间接索引块号在二次间接索引块内位置

int SingleBlockIndex = (LogicBlockNo - DiskInode.MAX\_MEDIUM\_FILE) % (DiskInode.INDEX\_PER\_BLOCK); //物理块号在一次间接索引块内位置

if (SingleBlockIndex == 0) //若新块位于一次间接索引表的第一项，需要再申请一个数据块作为一次间接索引块

{

try

{

SingleIndirectNo = AllocBlock();

}

catch //若获取间接索引块失败，将之前获取的数据块释放

{

FreeBlock(NewBlockNo);

throw;

}

if (DoubleBlockIndex == 0) //若新建的一次间接索引块同时还位于二次间接索引块的第一个，需要再申请一个数据块作为二次间接索引块

{

try

{

DoubleIndirectNo = AllocBlock();

}

catch //若获取间接索引块失败，将之前获取的数据块和一次间接索引块释放

{

FreeBlock(NewBlockNo);

FreeBlock(SingleIndirectNo);

throw;

}

/\* 将二次间接索引块置入索引表 \*/

CurFileInode.IndexTable[(LogicBlockNo - DiskInode.MAX\_MEDIUM\_FILE) / (DiskInode.INDEX\_PER\_BLOCK \* DiskInode.INDEX\_PER\_BLOCK) + DiskInode.DOUBLE\_INDIRECT\_BEGIN] = DoubleIndirectNo;

}

/\* 将一次间接索引块号写入二次间接索引块 \*/

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* DoubleIndirectNo + DoubleBlockIndex \* sizeof(int), SeekOrigin.Begin);

CurOpenFileStream.Write(BitConverter.GetBytes(SingleIndirectNo), 0, sizeof(int));

CurOpenFileStream.Flush();

}

else //否则从二次间接索引块获取一次间接索引块号

{

byte[] ReadBuffer = new byte[sizeof(int)];

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* DoubleIndirectNo + DoubleBlockIndex \* sizeof(int), SeekOrigin.Begin);

CurOpenFileStream.Read(ReadBuffer, 0, sizeof(int));

SingleIndirectNo = BitConverter.ToInt32(ReadBuffer, 0);

}

/\* 将新数据块号写入到一次间接索引块 \*/

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* SingleIndirectNo + SingleBlockIndex \* sizeof(int), SeekOrigin.Begin);

CurOpenFileStream.Write(BitConverter.GetBytes(NewBlockNo), 0, sizeof(int));

CurOpenFileStream.Flush();

}

else //若新块属于三次间接索引

{

int SingleIndirectNo; //一次间接索引块号

int DoubleIndirectNo; //二次间接索引块号

int TripleIndirectNo = CurFileInode.IndexTable[DiskInode.TRIPLE\_INDIRECT\_BEGIN]; //三次间接索引块号，默认直接从索引表取

int TripleBlockIndex = (LogicBlockNo - DiskInode.MAX\_LARGE\_FILE) / (DiskInode.INDEX\_PER\_BLOCK \* DiskInode.INDEX\_PER\_BLOCK); //二次间接索引块在三次间接索引块内位置

int DoubleBlockIndex = ((LogicBlockNo - DiskInode.MAX\_LARGE\_FILE) % (DiskInode.INDEX\_PER\_BLOCK \* DiskInode.INDEX\_PER\_BLOCK)) / DiskInode.INDEX\_PER\_BLOCK; //物理块对应一次间接索引块号在二次间接索引块内位置

int SingleBlockIndex = (LogicBlockNo - DiskInode.MAX\_LARGE\_FILE) % (DiskInode.INDEX\_PER\_BLOCK); //物理块号在一次间接索引块内位置

if (SingleBlockIndex == 0) //若新块位于一次间接索引表的第一项，需要再申请一个数据块作为一次间接索引块

{

try

{

SingleIndirectNo = AllocBlock();

}

catch //若获取间接索引块失败，将之前获取的数据块释放

{

FreeBlock(NewBlockNo);

throw;

}

if (DoubleBlockIndex == 0) //若新建的一次间接索引块同时还位于二次间接索引块的第一个，需要再申请一个数据块作为二次间接索引块

{

try

{

DoubleIndirectNo = AllocBlock();

}

catch //若获取间接索引块失败，将之前获取的数据块和一次间接索引块释放

{

FreeBlock(NewBlockNo);

FreeBlock(SingleIndirectNo);

throw;

}

if (TripleBlockIndex == 0) //若新建的二次间接索引块位于三次间接索引第一个，需要再申请一个数据块作为三次间接索引块

{

try

{

TripleIndirectNo = AllocBlock();

}

catch //若获取失败，将之前获取的数据块和一、二次间接索引块释放

{

FreeBlock(NewBlockNo);

FreeBlock(SingleIndirectNo);

FreeBlock(DoubleIndirectNo);

throw;

}

CurFileInode.IndexTable[DiskInode.TRIPLE\_INDIRECT\_BEGIN] = TripleIndirectNo; //将三次间接索引块号置入索引表

}

/\* 将新建的二次间接索引块号写入三次间接索引块 \*/

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* TripleIndirectNo + TripleBlockIndex \* sizeof(int), SeekOrigin.Begin);

CurOpenFileStream.Write(BitConverter.GetBytes(DoubleIndirectNo), 0, sizeof(int));

CurOpenFileStream.Flush();

}

else //否则直接从三次间接索引块获取二次间接索引块号

{

byte[] ReadBuffer = new byte[sizeof(int)];

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* TripleIndirectNo + TripleBlockIndex \* sizeof(int), SeekOrigin.Begin);

CurOpenFileStream.Read(ReadBuffer, 0, sizeof(int));

DoubleIndirectNo = BitConverter.ToInt32(ReadBuffer, 0);

}

/\* 将新建的一次间接索引块号写入二次间接索引块 \*/

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* DoubleIndirectNo + DoubleBlockIndex \* sizeof(int), SeekOrigin.Begin);

CurOpenFileStream.Write(BitConverter.GetBytes(SingleIndirectNo), 0, sizeof(int));

CurOpenFileStream.Flush();

}

else //否则先从三次间接索引块取得二次间接索引块，再从二次间接索引块取得一次间接索引块

{

byte[] ReadBuffer = new byte[sizeof(int)];

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* TripleIndirectNo + TripleBlockIndex \* sizeof(int), SeekOrigin.Begin);

CurOpenFileStream.Read(ReadBuffer, 0, sizeof(int));

DoubleIndirectNo = BitConverter.ToInt32(ReadBuffer, 0);

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* DoubleIndirectNo + DoubleBlockIndex \* sizeof(int), SeekOrigin.Begin);

CurOpenFileStream.Read(ReadBuffer, 0, sizeof(int));

SingleIndirectNo = BitConverter.ToInt32(ReadBuffer, 0);

}

/\* 将新数据块号写入到一次间接索引块 \*/

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* SingleIndirectNo + SingleBlockIndex \* sizeof(int), SeekOrigin.Begin);

CurOpenFileStream.Write(BitConverter.GetBytes(NewBlockNo), 0, sizeof(int));

CurOpenFileStream.Flush();

}

}

代码3.4 文件扩容函数代码

④文件收缩算法：当删除文件时，会缩小上级目录文件的大小，当大小缩小了一整个数据块时，就需要释放掉末尾的那个数据块。当释放数据块的同时导致了一、二或三级索引块为空时，需要将其同时释放。根据以上思路编写出文件搜索函数如代码3.5所示。

private void AbsorbFile() //若当前文件末尾的数据块已不再被占用，将其释放。若有间接索引块因此不再被占用，将其一并释放

{

if (CurFileInode.Size % BLOCK\_SIZE != 0) //要缩小的文件大小必然是数据块大小的整数倍

throw new InvalidOperationException();

int LogicBlockNo = CurFileInode.Size / BLOCK\_SIZE;

FreeBlock(GetPhyBlockNo(LogicBlockNo));

byte[] ReadBuffer = new byte[sizeof(int)];

int DoubleIndirectNo; //二次间接索引块号

int TripleIndirectNo = CurFileInode.IndexTable[DiskInode.TRIPLE\_INDIRECT\_BEGIN]; //三次间接索引块号，默认直接从索引表取

if (LogicBlockNo >= DiskInode.MAX\_LARGE\_FILE) //若释放的数据块属于三次间接索引

{

int TripleBlockIndex = (LogicBlockNo - DiskInode.MAX\_LARGE\_FILE) / (DiskInode.INDEX\_PER\_BLOCK \* DiskInode.INDEX\_PER\_BLOCK); //二次间接索引块在三次间接索引块内位置

int DoubleBlockIndex = ((LogicBlockNo - DiskInode.MAX\_LARGE\_FILE) % (DiskInode.INDEX\_PER\_BLOCK \* DiskInode.INDEX\_PER\_BLOCK)) / DiskInode.INDEX\_PER\_BLOCK; //物理块对应一次间接索引块号在二次间接索引块内位置

if ((CurFileInode.Size - DiskInode.MAX\_LARGE\_FILE \* BLOCK\_SIZE) % (BLOCK\_SIZE \* DiskInode.INDEX\_PER\_BLOCK) == 0) //一次间接索引块空，将其释放

{

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* TripleIndirectNo + TripleBlockIndex \* sizeof(int), SeekOrigin.Begin);

CurOpenFileStream.Read(ReadBuffer, 0, sizeof(int));

DoubleIndirectNo = BitConverter.ToInt32(ReadBuffer, 0);

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* DoubleIndirectNo + DoubleBlockIndex \* sizeof(int), SeekOrigin.Begin);

CurOpenFileStream.Read(ReadBuffer, 0, sizeof(int));

FreeBlock(BitConverter.ToInt32(ReadBuffer, 0));

}

if ((CurFileInode.Size - DiskInode.MAX\_LARGE\_FILE \* BLOCK\_SIZE) % (DiskInode.INDEX\_PER\_BLOCK \* DiskInode.INDEX\_PER\_BLOCK \* BLOCK\_SIZE) == 0) //二次间接索引块空，将其释放

{

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* TripleIndirectNo + TripleBlockIndex \* sizeof(int), SeekOrigin.Begin);

CurOpenFileStream.Read(ReadBuffer, 0, sizeof(int));

FreeBlock(BitConverter.ToInt32(ReadBuffer, 0));

}

if (CurFileInode.Size == DiskInode.MAX\_LARGE\_FILE) //文件现大小恰好为大型文件最大值，说明三次间接索引块空，将其释放

{

CurFileInode.IndexTable[DiskInode.TRIPLE\_INDIRECT\_BEGIN] = DiskInode.NULL\_NO;

FreeBlock(TripleIndirectNo);

}

}

else if (LogicBlockNo >= DiskInode.MAX\_MEDIUM\_FILE) //若释放的数据块属于二次间接索引

{

DoubleIndirectNo = CurFileInode.IndexTable[(LogicBlockNo - DiskInode.MAX\_MEDIUM\_FILE) / (DiskInode.INDEX\_PER\_BLOCK \* DiskInode.INDEX\_PER\_BLOCK) + DiskInode.DOUBLE\_INDIRECT\_BEGIN]; //二次间接索引块号,默认从索引表取

int DoubleBlockIndex = ((LogicBlockNo - DiskInode.MAX\_MEDIUM\_FILE) % (DiskInode.INDEX\_PER\_BLOCK \* DiskInode.INDEX\_PER\_BLOCK)) / DiskInode.INDEX\_PER\_BLOCK; //物理块对应一次间接索引块号在二次间接索引块内位置

if ((CurFileInode.Size - DiskInode.MAX\_MEDIUM\_FILE \* BLOCK\_SIZE) % (BLOCK\_SIZE \* DiskInode.INDEX\_PER\_BLOCK) == 0) //一次间接索引块空，将其释放

{

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* DoubleIndirectNo + DoubleBlockIndex \* sizeof(int), SeekOrigin.Begin);

CurOpenFileStream.Read(ReadBuffer, 0, sizeof(int));

FreeBlock(BitConverter.ToInt32(ReadBuffer, 0));

}

if ((CurFileInode.Size - DiskInode.MAX\_MEDIUM\_FILE \* BLOCK\_SIZE) % (DiskInode.INDEX\_PER\_BLOCK \* DiskInode.INDEX\_PER\_BLOCK \* BLOCK\_SIZE) == 0) //二次间接索引块空，将其释放

{

CurFileInode.IndexTable[(CurFileInode.Size - DiskInode.MAX\_MEDIUM\_FILE \* BLOCK\_SIZE) / (DiskInode.INDEX\_PER\_BLOCK \* DiskInode.INDEX\_PER\_BLOCK \* BLOCK\_SIZE) + DiskInode.DOUBLE\_INDIRECT\_BEGIN] = DiskInode.NULL\_NO;

FreeBlock(DoubleIndirectNo);

}

}

else if (LogicBlockNo >= DiskInode.MAX\_SMALL\_FILE) //若释放的数据库属于一次间接索引

{

if ((CurFileInode.Size - DiskInode.MAX\_SMALL\_FILE \* BLOCK\_SIZE) % (DiskInode.INDEX\_PER\_BLOCK \* BLOCK\_SIZE) == 0) //若一次间接索引块空，将其释放

{

FreeBlock(CurFileInode.IndexTable[(LogicBlockNo - DiskInode.MAX\_SMALL\_FILE) / DiskInode.INDEX\_PER\_BLOCK + DiskInode.SINGLE\_INDIRECT\_BEGIN]);

CurFileInode.IndexTable[(LogicBlockNo - DiskInode.MAX\_SMALL\_FILE) / DiskInode.INDEX\_PER\_BLOCK + DiskInode.SINGLE\_INDIRECT\_BEGIN] = DiskInode.NULL\_NO;

}

}

else //否则直接将索引表这项设置为NUll\_NO即可

CurFileInode.IndexTable[LogicBlockNo] = DiskInode.NULL\_NO;

}

代码 3.5 文件收缩函数代码

⑤文件读写算法：文件读写算法的基本思路是每次写一个基本块，当读到结尾时返回并给出实际读写字节数。当写到末尾时对文件进行扩容后继续写入，当无法扩容时返回并给出实际读写字节数。具体流程可以参考操作系统讲义P354的图7.31和图7.32。根据以上算法编写出文件读写函数ReadFile和WriteFile如代码3.6所示。

public int ReadFile(byte[] read\_buffer, int offset, int need\_count) //从当前打开文件当前的文件指针处读出指定数量的字节,并从缓冲区的指定位置开始存放,返回实际读出的字节数

{

int RealCount = need\_count;

if (FilePtr + RealCount > CurFileInode.Size) //若要读取的字节超出范围，计算实际能读出的字节数

RealCount = CurFileInode.Size - FilePtr;

int LogicBlockNo = FilePtr / BLOCK\_SIZE; //设置起始读写逻辑块号

int BlockStartIndex = FilePtr % BLOCK\_SIZE; //起始读写块内位置

int NeedToRead = RealCount; //剩余需读写字节数量

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* GetPhyBlockNo(LogicBlockNo) + BlockStartIndex, SeekOrigin.Begin); //将文件流读写指针调整到起始位置

if (BLOCK\_SIZE - BlockStartIndex > RealCount) //若要读出的字节数量没有超出起始数据块，读出需要的数据即可

{

CurOpenFileStream.Read(read\_buffer, offset, RealCount);

NeedToRead = 0;

}

else //否则读完这个数据块

{

CurOpenFileStream.Read(read\_buffer, offset, BLOCK\_SIZE - BlockStartIndex);

NeedToRead-= BLOCK\_SIZE - BlockStartIndex;

}

while (NeedToRead > 0) //不断读取数据块直至读完

{

LogicBlockNo++; //逻辑块号+1

int pno = GetPhyBlockNo(LogicBlockNo);

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* GetPhyBlockNo(LogicBlockNo), SeekOrigin.Begin); //设置读写指针到物理块开头

if (NeedToRead < BLOCK\_SIZE) //若剩余需读取的数据不足一块，读取剩余数据

{

CurOpenFileStream.Read(read\_buffer, RealCount - NeedToRead + offset, NeedToRead);

NeedToRead = 0;

}

else //否则将整块读取

{

CurOpenFileStream.Read(read\_buffer, RealCount - NeedToRead + offset, BLOCK\_SIZE);

NeedToRead -= BLOCK\_SIZE;

}

}

FilePtr += RealCount; //修改文件指针位置

return RealCount;

}

public int WriteFile(byte[] write\_buffer, int offset, int need\_count) //从当前打开文件当前的文件指针处写入缓冲区内从指定位置开始指定数量的字节,返回实际写入的字节数

{

int WrittenCount = 0;

int LogicBlockNo = FilePtr / BLOCK\_SIZE; //设置起始读写逻辑块号

int BlockStartIndex = FilePtr % BLOCK\_SIZE; //起始读写块内位置

if (CurFileInode.Size == 0 && need\_count > 0 && CurFileInodeNo != 0) //若当前打开文件为空且不是根目录文件，说明该文件没有被分配数据块，需要新为其分配一个数据块以写入数据

{

/\* 尝试获取一个数据块并置入索引表0号位置，若获取不到，则返回写入数据量为0 \*/

try

{

CurFileInode.IndexTable[0] = AllocBlock();

}

catch

{

return 0;

}

}

while (WrittenCount < need\_count) //循环写入直至写完

{

int WriteCount = Math.Min(BLOCK\_SIZE - BlockStartIndex, need\_count - WrittenCount); //在本逻辑块可剩余可写字节数和还需写入字节数中选取最小值作为本次写入的字节数量

int pno = GetPhyBlockNo(LogicBlockNo);

CurOpenFileStream.Seek(DATA\_BEGIN + BLOCK\_SIZE \* GetPhyBlockNo(LogicBlockNo) + BlockStartIndex, SeekOrigin.Begin); //将文件流调整到本次写入的起始位置

CurOpenFileStream.Write(write\_buffer, offset + WrittenCount, WriteCount); //写入本次应写入的字节

CurOpenFileStream.Flush();

WrittenCount += WriteCount; //已写入的字节增加本次写入的字节数

if (FilePtr + WrittenCount > CurFileInode.Size) //若当前位置已经超出原文件长度，修改文件长度

{

CurFileInode.Size = FilePtr + WrittenCount;

if (CurFileInode.Size == DiskInode.MAX\_FILE \* BLOCK\_SIZE) //当文件大小到达系统支持的最大值时，结束写入并返回

{

FilePtr += WrittenCount;

return WrittenCount;

}

/\* 若文件长度超出了当前文件长度类型的最大长度，需要修改文件长度类型 \*/

if (CurFileInode.Size > DiskInode.MAX\_LARGE\_FILE \* BLOCK\_SIZE)

CurFileInode.\_FileLengthType = DiskInode.FileLengthType.Huge;

else if (CurFileInode.Size > DiskInode.MAX\_MEDIUM\_FILE \* BLOCK\_SIZE)

CurFileInode.\_FileLengthType = DiskInode.FileLengthType.Large;

else if (CurFileInode.Size > DiskInode.MAX\_SMALL\_FILE \* BLOCK\_SIZE)

CurFileInode.\_FileLengthType = DiskInode.FileLengthType.Medium;

if (WrittenCount < need\_count)

{

try //若仍有数据需要写入，尝试为该文件进行扩容

{

ExpandFile();

}

catch //若无法进行扩容，终止写入并返回已写入的字节数量

{

CurFileInode.LastModifyTime = DateTime.Now.Ticks; //设置文件最后修改时刻

WriteDiskInode(CurFileInode, CurFileInodeNo); //将修改后的DiskInode写回磁盘

FilePtr += WrittenCount;

return WrittenCount;

}

}

}

LogicBlockNo++; //逻辑块号+1

BlockStartIndex = 0; //除第一次以外，均从数据块的开头开始写入

}

CurFileInode.LastModifyTime = DateTime.Now.Ticks; //设置文件最后修改时刻

WriteDiskInode(CurFileInode, CurFileInodeNo); //将修改后的DiskInode写回磁盘

FilePtr += WrittenCount;

return WrittenCount; //写入结束，返回

}

**代码3.6 文件读写函数代码**

**⑥文件删除算法：当删除文件时，需要先逐个删除数据块，然后删除中间的所有间接索引块。随后需要打开删除文件的上级目录文件，然后将删除文件从目录文件中删去。当删去的目录不是最后一条目录时，需要从目录末尾取出一条目录补回空位。根据以上算法，编写出文件删除函数DeleteFile如代码3.7所示。**

public void DeleteFile(string path) //删除指定路径名的文件

{

DiskInode OriginInode = CurFileInode; //备份当前打开Inode

int OriginFilePtr = FilePtr; //备份当前文件指针

int OriginNo = CurFileInodeNo; //备份当前Inode在Inode区编号

int ParentInodeNo; //上级目录文件编号

try //先尝试打开该文件

{

ParentInodeNo = OpenFile(path);

}

catch

{

throw;

}

if (CurFileInode.\_FileType == DiskInode.FileType.Index && CurFileInode.Size != 0) //不允许删除仍有目录项的目录文件，因此还原现场并抛出异常

{

CurFileInode = OriginInode;

CurFileInodeNo = OriginNo;

FilePtr = OriginFilePtr;

throw new ArgumentException("该目录下仍然存在文件！");

}

/\* 开始删除文件 \*/

for (int i = 0; i < Math.Ceiling((double)CurFileInode.Size / BLOCK\_SIZE); i++) //逐个释放本文件占用的所有数据块

FreeBlock(GetPhyBlockNo(i));

if (CurFileInode.Size > DiskInode.MAX\_SMALL\_FILE \* BLOCK\_SIZE) //对于大于小型文件的文件，释放一次间接索引块

{

for (int i = DiskInode.SINGLE\_INDIRECT\_BEGIN; i < DiskInode.DOUBLE\_INDIRECT\_BEGIN; i++)

FreeBlock(CurFileInode.IndexTable[i]);

}

if (CurFileInode.Size > DiskInode.MAX\_MEDIUM\_FILE \* BLOCK\_SIZE) //对于大于中型文件的文件，释放一次和二次间接索引块

{

byte[] ReadBuffer = new byte[sizeof(int)];

int BlockToFree;

for (int i = 0; i < Math.Ceiling((double)(CurFileInode.Size - DiskInode.MAX\_MEDIUM\_FILE \* BLOCK\_SIZE) / (DiskInode.INDEX\_PER\_BLOCK \* BLOCK\_SIZE)); i++) //逐个释放一次间接索引块

{

if (i >= (DiskInode.TRIPLE\_INDIRECT\_BEGIN - DiskInode.DOUBLE\_INDIRECT\_BEGIN) \* DiskInode.INDEX\_PER\_BLOCK) //控制释放的一次间接索引块不能超出中型文件的最大数量

break;

CurOpenFileStream.Seek(DATA\_BEGIN + CurFileInode.IndexTable[DiskInode.DOUBLE\_INDIRECT\_BEGIN + i / DiskInode.INDEX\_PER\_BLOCK] \* BLOCK\_SIZE + (i % DiskInode.INDEX\_PER\_BLOCK) \* sizeof(int), SeekOrigin.Begin);

CurOpenFileStream.Read(ReadBuffer, 0, sizeof(int)); //读入下一个一次间接索引块

BlockToFree = BitConverter.ToInt32(ReadBuffer, 0);

FreeBlock(BlockToFree); //释放一次间接索引块

}

for (int i = DiskInode.DOUBLE\_INDIRECT\_BEGIN; i < DiskInode.TRIPLE\_INDIRECT\_BEGIN; i++) //释放二次间接索引块

FreeBlock(CurFileInode.IndexTable[i]);

}

if (CurFileInode.Size > DiskInode.MAX\_LARGE\_FILE \* BLOCK\_SIZE) //对于大于大型文件的文件，释放一次、二次和三次间接索引块

{

byte[] ReadBuffer = new byte[sizeof(int)];

int TripleIndirectNo = CurFileInode.IndexTable[DiskInode.TRIPLE\_INDIRECT\_BEGIN]; //获取三次间接索引块号

int TripleBlockIndex = 0; //二次间接索引块在三次间接索引的位置

int DoubleIndirectNo = 0; //当前正在处理的二次间接索引块号

int BlockToFree;

for (int i = 0; i < Math.Ceiling((double)(CurFileInode.Size - DiskInode.MAX\_LARGE\_FILE \* BLOCK\_SIZE) / (DiskInode.INDEX\_PER\_BLOCK \* BLOCK\_SIZE)); i++) //逐个释放一、二次间接索引块

{

if (i % DiskInode.INDEX\_PER\_BLOCK == 0) //每遍历完一个二次间接索引块，释放该块并重新设置二次间接索引块

{

if (i != 0) //除第一次以外，释放上一个二次间接索引块

FreeBlock(DoubleIndirectNo);

CurOpenFileStream.Seek(DATA\_BEGIN + TripleIndirectNo \* BLOCK\_SIZE + TripleBlockIndex \* sizeof(int), SeekOrigin.Begin); //从三次间接索引块中读出一个新的二次间接索引块号

CurOpenFileStream.Read(ReadBuffer, 0, sizeof(int));

DoubleIndirectNo = BitConverter.ToInt32(ReadBuffer, 0);

TripleBlockIndex++;

}

CurOpenFileStream.Seek(DATA\_BEGIN + DoubleIndirectNo \* BLOCK\_SIZE + (i % DiskInode.INDEX\_PER\_BLOCK) \* sizeof(int), SeekOrigin.Begin); //将读写指针调整二次间接索引块相应位置

CurOpenFileStream.Read(ReadBuffer, 0, sizeof(int)); //读入下一个一次间接索引块

BlockToFree = BitConverter.ToInt32(ReadBuffer, 0);

FreeBlock(BlockToFree); //释放一次间接索引块

}

FreeBlock(DoubleIndirectNo); //释放最后的二次间接索引块

FreeBlock(TripleIndirectNo); //释放三次间接索引块

}

/\* 设置本文件的Inode未被占用并保存，然后释放该Inode \*/

CurFileInode.Occupied = false;

WriteDiskInode(CurFileInode, CurFileInodeNo);

FreeInode(CurFileInodeNo);

/\* 打开上级目录，并从目录里删去这个目录项 \*/

int DeleteInodeNo = CurFileInodeNo; //记录要删除目录项Inode号

CurFileInode = ReadDiskInode(ParentInodeNo);

CurFileInodeNo = ParentInodeNo;

FilePtr = 0;

byte[] IndexReadBuffer = new byte[DirectoryEntry.DIR\_SIZE];

for (int i = 0; i < CurFileInode.Size / DirectoryEntry.DIR\_SIZE; i++) //遍历所有目录项

{

ReadFile(IndexReadBuffer, 0, DirectoryEntry.DIR\_SIZE);

if (BitConverter.ToInt32(IndexReadBuffer, 0) == DeleteInodeNo) //若找到了要删除的目录项,删除该项，若不是最后一项目录项，从末尾取一个目录项补到这个位置

{

if (i != CurFileInode.Size / DirectoryEntry.DIR\_SIZE - 1)

{

FilePtr = (CurFileInode.Size / DirectoryEntry.DIR\_SIZE - 1) \* DirectoryEntry.DIR\_SIZE; //移动文件指针到末尾的目录项

ReadFile(IndexReadBuffer, 0, DirectoryEntry.DIR\_SIZE); //读出末尾的目录项

FilePtr = i \* DirectoryEntry.DIR\_SIZE;

WriteFile(IndexReadBuffer, 0, DirectoryEntry.DIR\_SIZE); //将末尾的目录项写回要删除的目录项

}

Seek(CurFileInode.Size - DirectoryEntry.DIR\_SIZE);

WriteFile(new byte[DirectoryEntry.DIR\_SIZE], 0, DirectoryEntry.DIR\_SIZE); //用0覆写末尾的目录项

CurFileInode.Size -= DirectoryEntry.DIR\_SIZE; //修改文件大小

if (CurFileInode.Size % BLOCK\_SIZE == 0) //若一整个目录文件块为空，收缩文件

AbsorbFile();

WriteDiskInode(CurFileInode, CurFileInodeNo); //将上级目录文件Inode存回磁盘

break; //目录删除完成，结束循环

}

}

/\* 文件删除完毕，若删除文件的目录不是原先打开的文件，还原现场 \*/

if (CurFileInodeNo != OriginNo)

{

CurFileInode = OriginInode;

CurFileInodeNo = OriginNo;

FilePtr = OriginFilePtr;

}

}

**代码3.7 文件删除函数代码**

**⑦文件创建算法：创建文件时，需要尝试获取一个Inode，然后先打开文件的上级目录，尝试将新文件的目录项写入目录文件。最后将新文件的Inode存入文件卷。根据以上算法编写出文件创建函数CreateFile如代码3.8所示。**

public void CreateFile(string path, bool allow\_overlay, DiskInode.FileType type) //创建指定路径名的文件，路径用\分隔

{

if (path.Substring(path.LastIndexOf('\\')).Length > DirectoryEntry.NAME\_MAX\_LENGTH) //若名称过长，抛出异常

throw new ArgumentException("文件名不能超过14个字符！");

int NewInode;

string CreateFileName = path.Substring(path.LastIndexOf('\\') + 1); //取出要创建的文件名

DiskInode OriginInode = CurFileInode; //备份当前打开Inode

int OriginFilePtr = FilePtr; //备份当前文件指针

int OriginNo = CurFileInodeNo; //备份当前Inode在Inode区编号

try //尝试打开上级目录文件，若出错，抛出抓取到的异常

{

OpenFile(path.Substring(0, path.Length - CreateFileName.Length - 1)); //将不含文件名的目录打开，同时也不含最后一个\

}

catch

{

throw;

}

if (CurFileInode.Size == DiskInode.MAX\_FILE \* BLOCK\_SIZE) //若上级目录文件大小已达上限，抛出异常

throw new InvalidOperationException("目录内无更多空间！");

if (CurFileInode.\_FileType != DiskInode.FileType.Index) //若上级文件不是目录文件，抛出异常

throw new ArgumentException("无法在文件下创建文件！");

bool FileDuplicate = true; //默认有重复

try //尝试在上级目录下寻找这个文件，若找不到，则创建该文件。若找到，则根据是否允许覆盖决定是否覆盖

{

Find(CurFileInodeNo, CreateFileName);

}

catch(FileNotFoundException)

{

FileDuplicate = false; //若找不到文件，说明不重复

}

if (!FileDuplicate || allow\_overlay) //若文件不重复或文件允许覆盖，创建文件

{

try //尝试获取一个新空闲Inode位置,若失败，抛出异常

{

NewInode = AllocInode();

}

catch

{

throw;

}

if (FileDuplicate) //若文件重复

DeleteFile(path); //删除该文件

Seek(CurFileInode.Size); //将读写指针移动到目录文件末尾

try

{

WriteFile(BitConverter.GetBytes(NewInode), 0, sizeof(int)); //写入Inode号到目录

WriteFile(Encoding.Unicode.GetBytes(CreateFileName), 0, CreateFileName.Length \* sizeof(char)); //紧接Inode号写入文件名

WriteFile(new byte[DirectoryEntry.NAME\_SIZE - CreateFileName.Length \* sizeof(char)], 0, DirectoryEntry.NAME\_SIZE - CreateFileName.Length \* sizeof(char)); //用0填充剩余部分

}

catch //若写入失败，说明无空余空间，释放Inode并抛出异常

{

FreeInode(NewInode);

throw;

}

/\* 配置新文件的DiskInode \*/

DiskInode CreateFileInode = new DiskInode();

CreateFileInode.Occupied = true;

CreateFileInode.\_FileType = type;

CreateFileInode.LastModifyTime = DateTime.Now.Ticks;

WriteDiskInode(CreateFileInode, NewInode); //写入到Inode区

}

/\* 若Inode不同，还原原先的Inode、编号和文件指针 \*/

if (CurFileInodeNo != OriginNo)

{

CurFileInode = OriginInode;

CurFileInodeNo = OriginNo;

FilePtr = OriginFilePtr;

}

if (FileDuplicate && !allow\_overlay)

throw new ArgumentException("文件名重复！");

}

**代码3.8 文件创建函数代码**

## ****3.3 数据层的分解****

数据层为纯数据结构定义，其承担的职责参见软件体系结构设计文档表3.7。类的详细定义参见代码3.9。

class SuperBlock

{

public const int NULL\_NO = -1; //代表无效的块号

public const int OFFSET = 4; //类存储结构开始位置的偏移量

public const int TRUE\_SIZE = 1024; //类实际大小

public const int PADDING\_SIZE = 0; //类尾填充大小

public const int FULL\_SIZE = 1024; //类总大小

public const int INODE\_INDEX\_SIZE = 123; //空闲Inode索引表大小

public const int MAGIC\_NUM = 0x12345678; //用于表示文件系统已经格式化的魔数

public int InodeCount; //可用Inode盘块总数

public int BlockCount; //可用一般数据盘块总数

public int FreeBlockCount; //直接管理的空闲盘块数量

public int[] FreeBlockIndex; //空闲盘块索引

public int FreeInodeCount; //直接管理的空闲Inode数量

public int[] FreeInodeIndex; //空闲Inode索引

public SuperBlock()

{

/\* 设置空闲盘块和空闲Inode索引表大小 \*/

FreeBlockIndex = new int[FileManager.FREE\_BLOCK\_GROUP\_SIZE];

FreeInodeIndex = new int[INODE\_INDEX\_SIZE];

}

}

class DiskInode

{

public const int NULL\_NO = -1; //无效块号，表示索引表该位置未被使用

public const int INODE\_SIZE = 64; //DiskINode长度

public const int INDEX\_TABLE\_SIZE = 11; //索引表长度

public const int SINGLE\_INDIRECT\_BEGIN = 6; //一次间接索引在索引表开始位置

public const int DOUBLE\_INDIRECT\_BEGIN = 8; //二次间接索引在索引表开始位置

public const int TRIPLE\_INDIRECT\_BEGIN = 10; //三次间接索引在索引表开始位置

public const int INDEX\_PER\_BLOCK = FileManager.BLOCK\_SIZE / sizeof(int); //每个间接索引块能存放的索引数量

/\* 小、中、大和巨大文件最多能占用的数据块数 \*/

public const int MAX\_SMALL\_FILE = SINGLE\_INDIRECT\_BEGIN;

public const int MAX\_MEDIUM\_FILE = MAX\_SMALL\_FILE + (DOUBLE\_INDIRECT\_BEGIN - SINGLE\_INDIRECT\_BEGIN) \* INDEX\_PER\_BLOCK;

public const int MAX\_LARGE\_FILE = MAX\_MEDIUM\_FILE + (TRIPLE\_INDIRECT\_BEGIN - DOUBLE\_INDIRECT\_BEGIN) \* INDEX\_PER\_BLOCK \* INDEX\_PER\_BLOCK;

public const int MAX\_FILE = MAX\_LARGE\_FILE + (INDEX\_TABLE\_SIZE - TRIPLE\_INDIRECT\_BEGIN) \* INDEX\_PER\_BLOCK \* INDEX\_PER\_BLOCK \* INDEX\_PER\_BLOCK;

public enum FileType: byte

{

Data,

Index

}

public enum FileLengthType: short

{

Small,

Medium,

Large,

Huge

}

public bool Occupied; //该Inode是否被占用

public FileType \_FileType; //文件类型

public FileLengthType \_FileLengthType; //文件长度类型

public int LinkCount; //路径名数量

public int Size; //文件大小

public int[] IndexTable; //索引表

public long LastModifyTime; //最后修改时间

public DiskInode()

{

IndexTable = new int[INDEX\_TABLE\_SIZE];

for (int i = 0; i < INDEX\_TABLE\_SIZE; i++) //设置索引表所有位置均未被使用

IndexTable[i] = NULL\_NO;

}

}

class DirectoryEntry

{

public const int NULL\_NO = 0; //无效Inode编号

public const int DIR\_SIZE = 32; //目录结构长度

public const int NAME\_SIZE = 28; //路径名最多占用空间

public const int NAME\_MAX\_LENGTH = NAME\_SIZE / sizeof(char); //路径名最多字符数量

public int InodeNo;

public char[] FileName;

public DirectoryEntry()

{

FileName = new char[NAME\_MAX\_LENGTH];

}

}

代码3.9 数据层数据结构定义

模拟二级文件系统软件测试分析报告

# 1.引言

## 1.1 编制目的

本文档通过模拟不同的软件使用场景，并通过设置一些极端情况对文件系统进行测试，以判断程序代码的正确性。

## 1.2 参考文献

1）IEEE标准

2）丁二玉,刘钦.软件开发的技术基础

3）操作系统课程讲义

# 2.测试用例

## 2.1 基础功能测试

这一用例为课程设计ppt中对主程序功能的要求，抄录如下：

·格式化文件卷；

·创建子目录，建立如图2.1所示的目录结构；

·把随意的一个纯文本文件，你的课设报告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；

·新建文件/test/Jerry，打开该文件，写入800个字节；

·将文件读写指针定位到第500字节，读出20个字节。

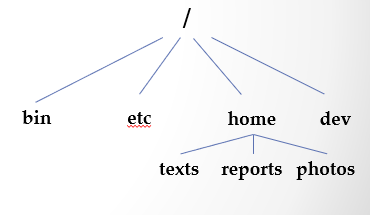


图2.1 基础功能测试的目录结构

## 2.2 大文件存储测试

这一用例用于测试程序处理大文件的能力，步骤如下：

·在2.1目录结构中创建/home/video文件夹；

·将一个较大的视频文件存入/home/video，然后读出，播放观察数据是否正确读写；

·再存入若干个较大的视频文件直至空间用尽；

·删除所有的视频文件，随后再次存入大量数据以观察所有数据块是否被正确释放。

## 2.3 多文件存储测试

这一用例用于测试文件系统处理大量文件的能力，步骤如下：

·在根目录下创建一个文件夹，然后在文件夹下创建文件直至所有inode用完；

·删除文件夹，然后重新创建一个5层的目录结构，除第一个文件夹，每一个文件夹下都有5个文件夹；

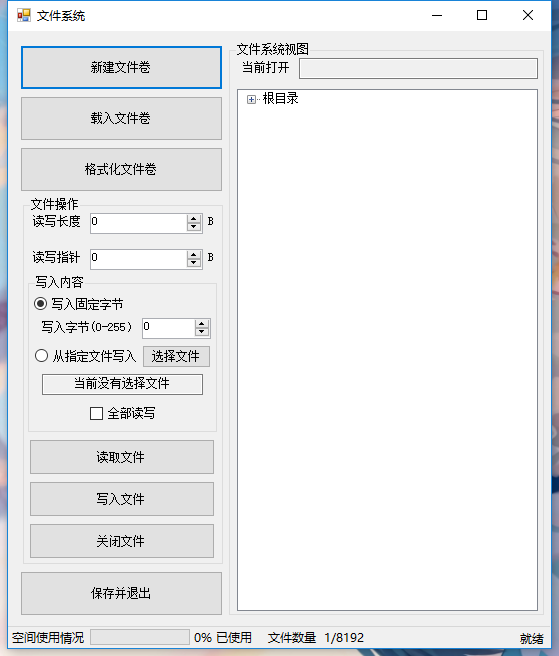
·随意取一个文件夹，在这些文件夹创建一个文件，并写入一个较大的文件；

·删除包含后面所有文件夹和文件的第一个文件夹。

# 3.调试过程

## 3.1 基础功能测试

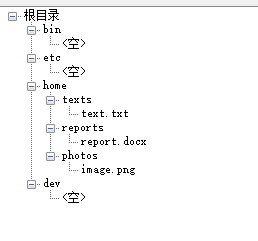
①创建并格式化一个文件卷



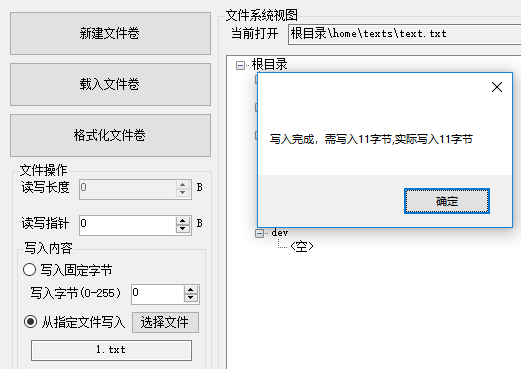
②创建所需的目录结构

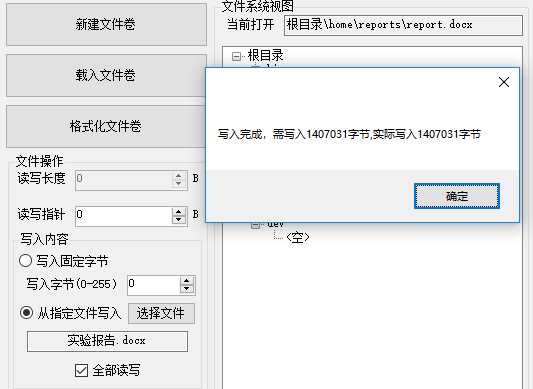


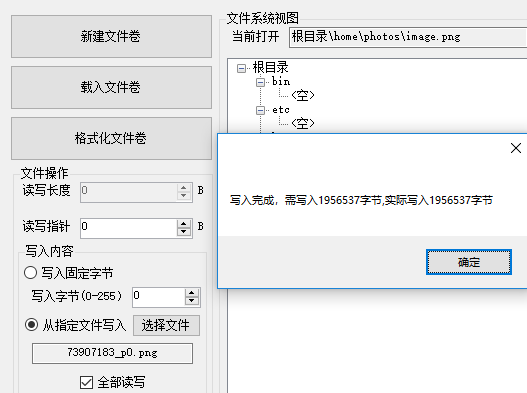
③在/home/texts ，/home/reports和/home/photos文件夹下各创建一个文件，用于写入纯文本文件、课设报告和图片



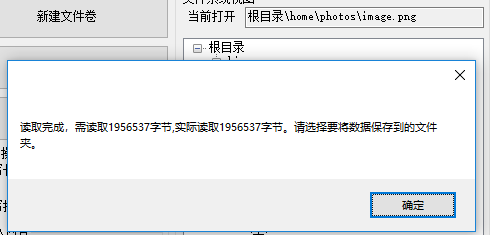
④选择“从指定文件写入”，勾选全部读写，将一个文本文件，一份课设报告和一张图片存入上面的文件。

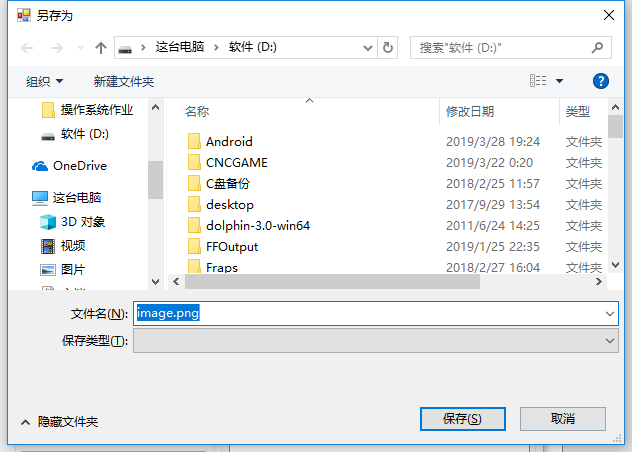


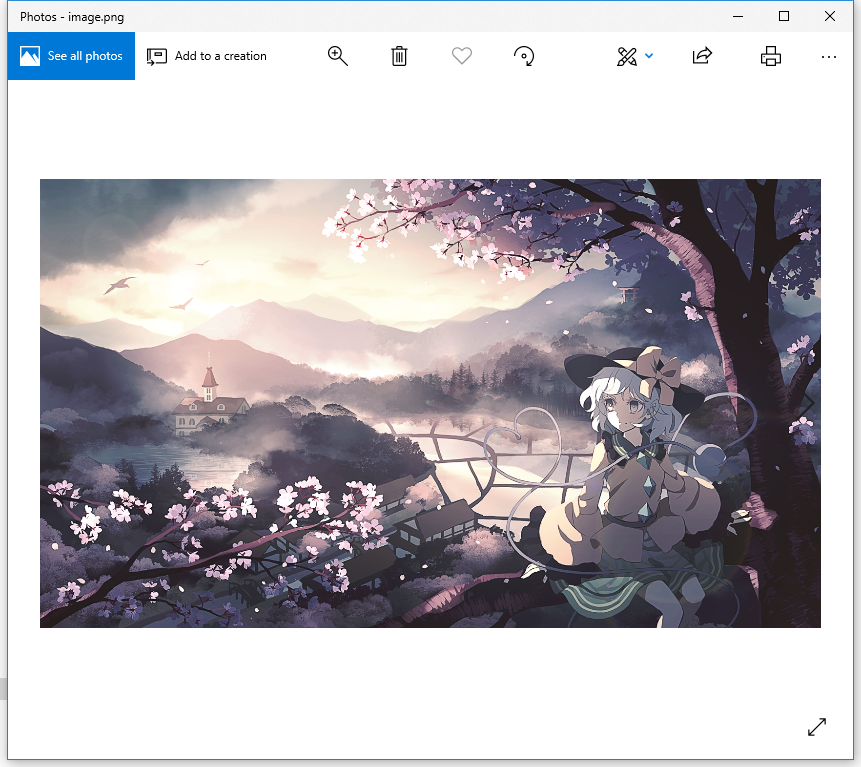




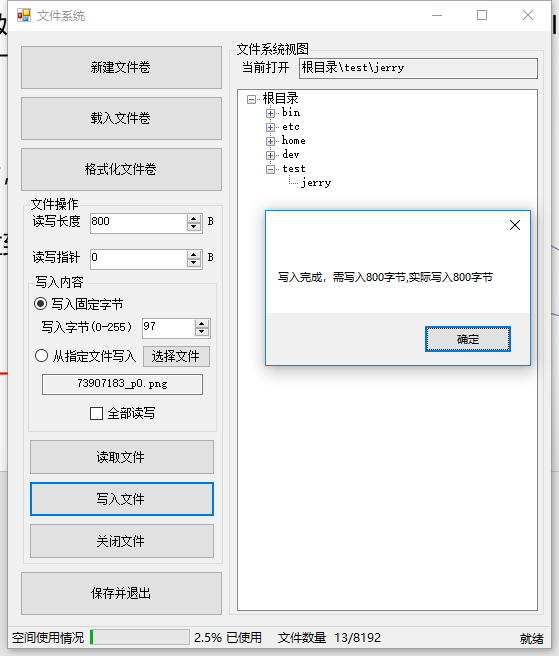
⑤将image.png从文件系统中读出，图片可以正常打开，说明文件存取正常。



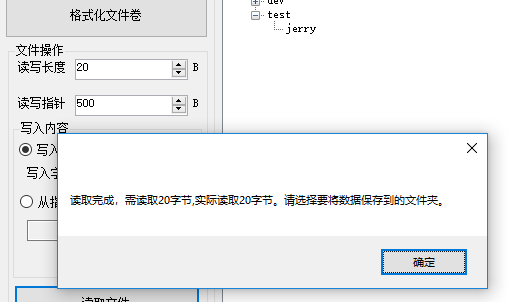


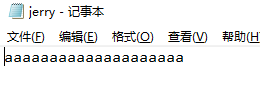


⑥创建test/jerry，并向其写写入800个97d（0x61）字节，也就是'a'的ASCII码



⑦随后从500字节处读出20个字节并导出，用记事本打开，能够查看到20个'a'，说明读写正确



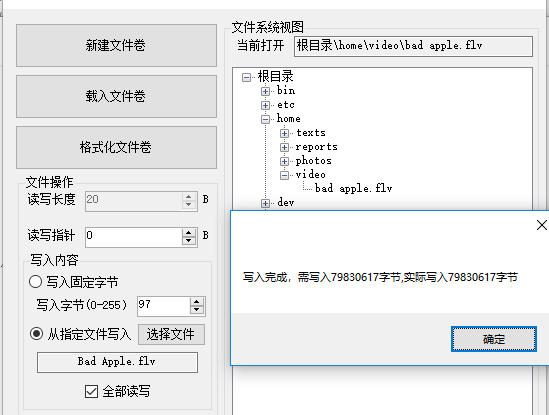


## 3.2 大文件存储测试

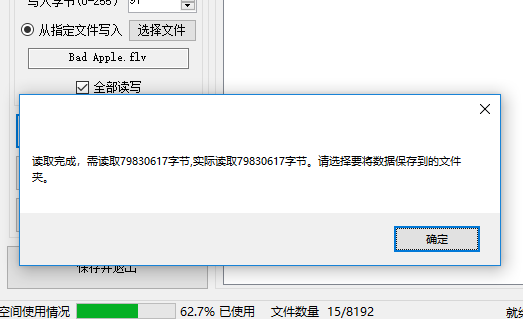
①创建home/video并创建一个文件供写入

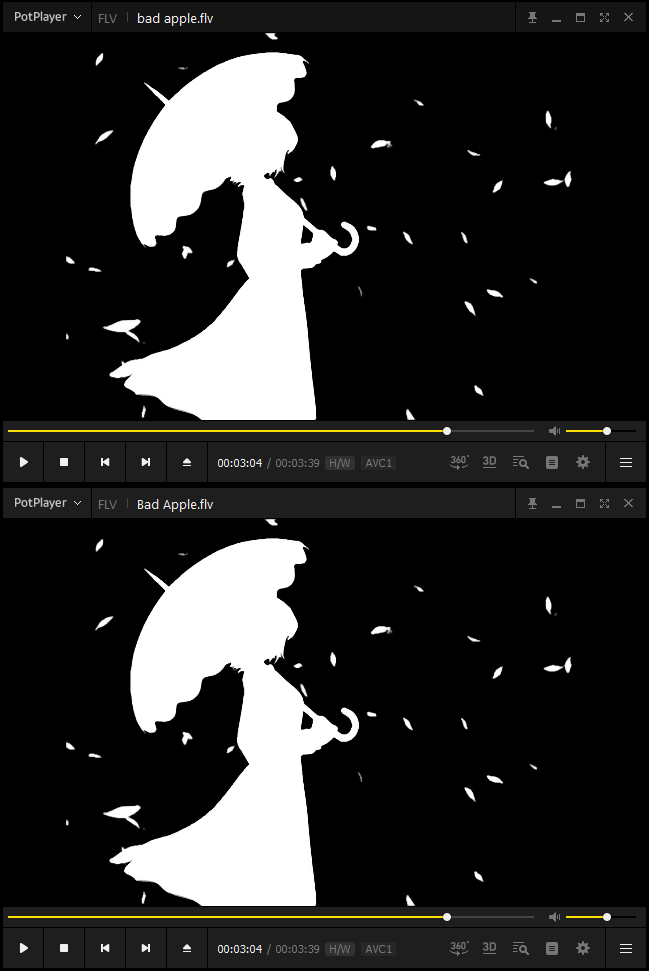


②写入一个大小约76MB的视频文件，该文件将要用到三次间接索引。测试可以正常写入。

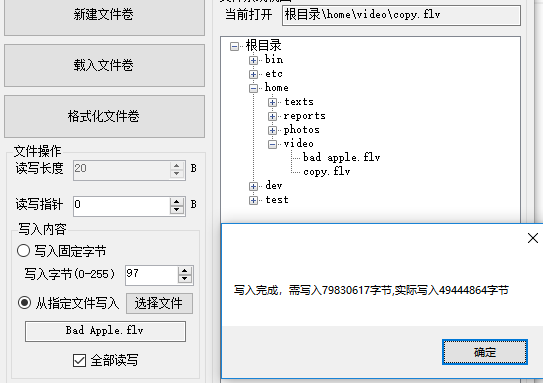


③将其读出并与原版视频对比，发现能够正常播放并且完全一致，说明大文件读写正常。





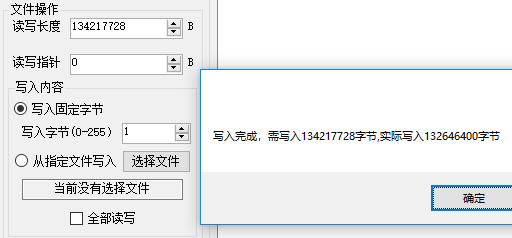
④再次创建并写入一个相同的视频文件，由于空间不足，只能写入部分



⑤删除home文件夹和test文件夹以删除所有文件，释放了所有空间



⑥创建一个新的文件并尝试写入128MB的字节，可以看到除了一些空间被间接索引块占据，基本能够全部写入，说明空间已经释放成功。



## 3.3 多文件存储测试

①首先进行将文件夹装满文件直至inode区满的测试。由于手动创建文件效率过低，因此在右键菜单中添加了一个调试按键，并编写按键响应函数如代码3.1所示。

private void 装满文件ToolStripMenuItem\_Click(object sender, EventArgs e)

{

for (int i = 0; i < FileManager.MAX\_INODE; i++)

{

StatusTB.Text = "正在创建" + FileSystemTV.SelectedNode.FullPath + '\\' + i.ToString();

Refresh();

try //不断尝试创建文件直至异常

{

FileManagerInstance.CreateFile(FileSystemTV.SelectedNode.FullPath + '\\' + i.ToString(), false, DiskInode.FileType.Data);

}

catch

{

break;

}

}

StatusTB.Text = READY\_TEXT;

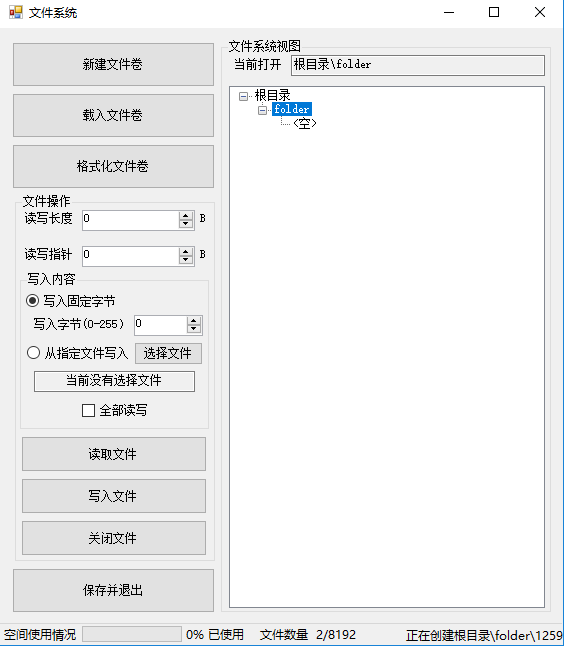
CheckSpaceUsage();

}

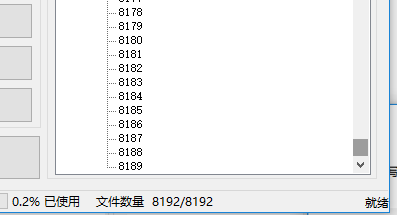
}

代码3.1 装满文件函数

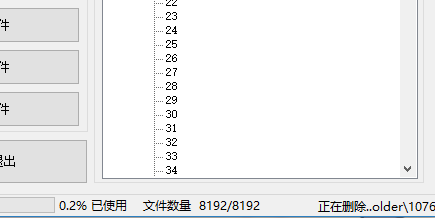
②创建新的文件夹folder并点击该按钮以装满文件。



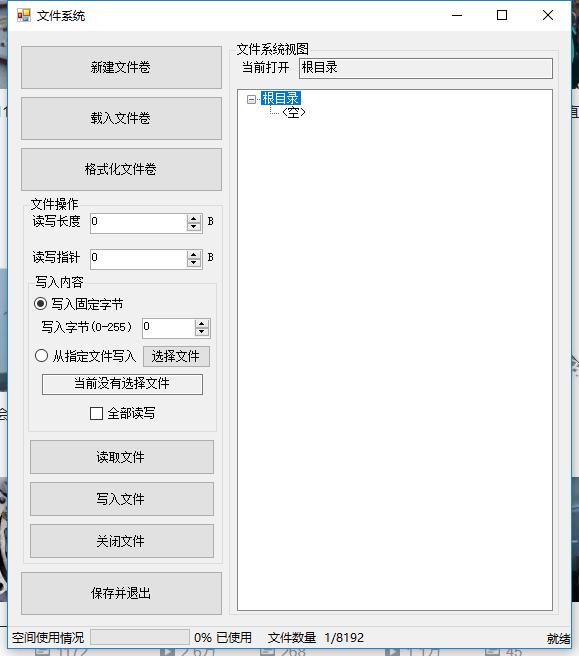
③最终成功创建0-8189号文件供8190个文件，加上根目录文件和folder目录文件共8192个文件，正好用完所有Inode。



④接着删除folder文件夹，将会同时递归删除文件夹内所有文件。



⑤最终删除成功并释放了所有文件夹内文件占用的Inode。



⑥接下来测试5层复杂目录结构。首先编写递归目录创建函数和调试用按钮响应函数如代码6.2所示。

private void 创建五层文件夹ToolStripMenuItem\_Click(object sender, EventArgs e)

{

RecurseCreate(FileSystemTV.SelectedNode.FullPath, 0);

CheckSpaceUsage();

}

private void RecurseCreate(string path,int layer)

{

if (layer >= 5) //最多创建5层

return;

for (int i = 0; i < 5; i++)

{

try

{

FileManagerInstance.CreateFile(path + "\\folder" + i.ToString(), false, DiskInode.FileType.Index);

}

catch

{

break;

}

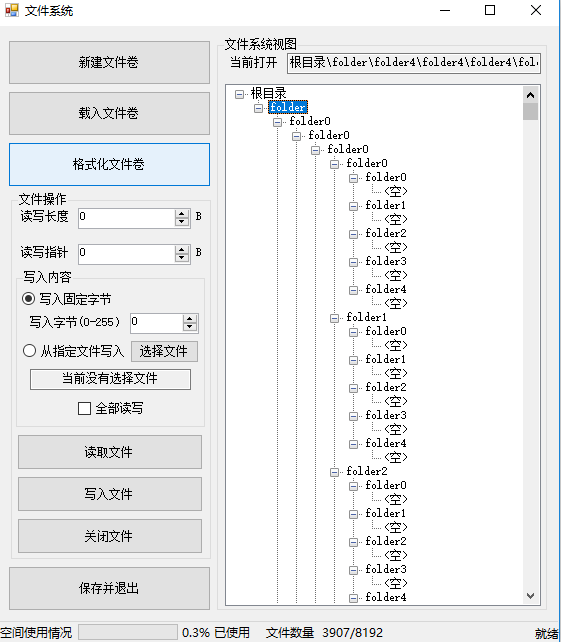
RecurseCreate(path + "\\folder" + i.ToString(), layer + 1); //对创建的每个文件夹再次执行本函数

}

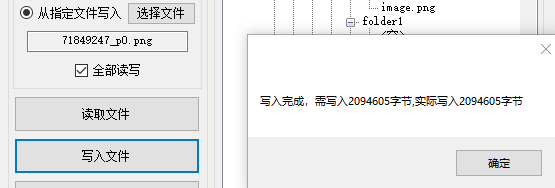
}

代码6.2 创建五层文件目录函数

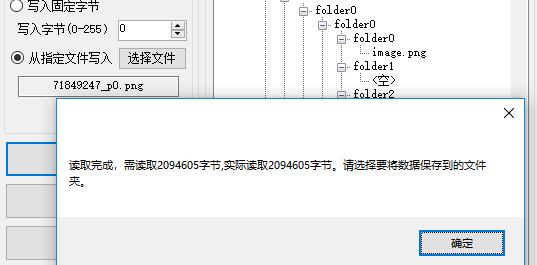
⑦创建一个文件夹folder,然后在其之下创建五层目录并全部展开如下。

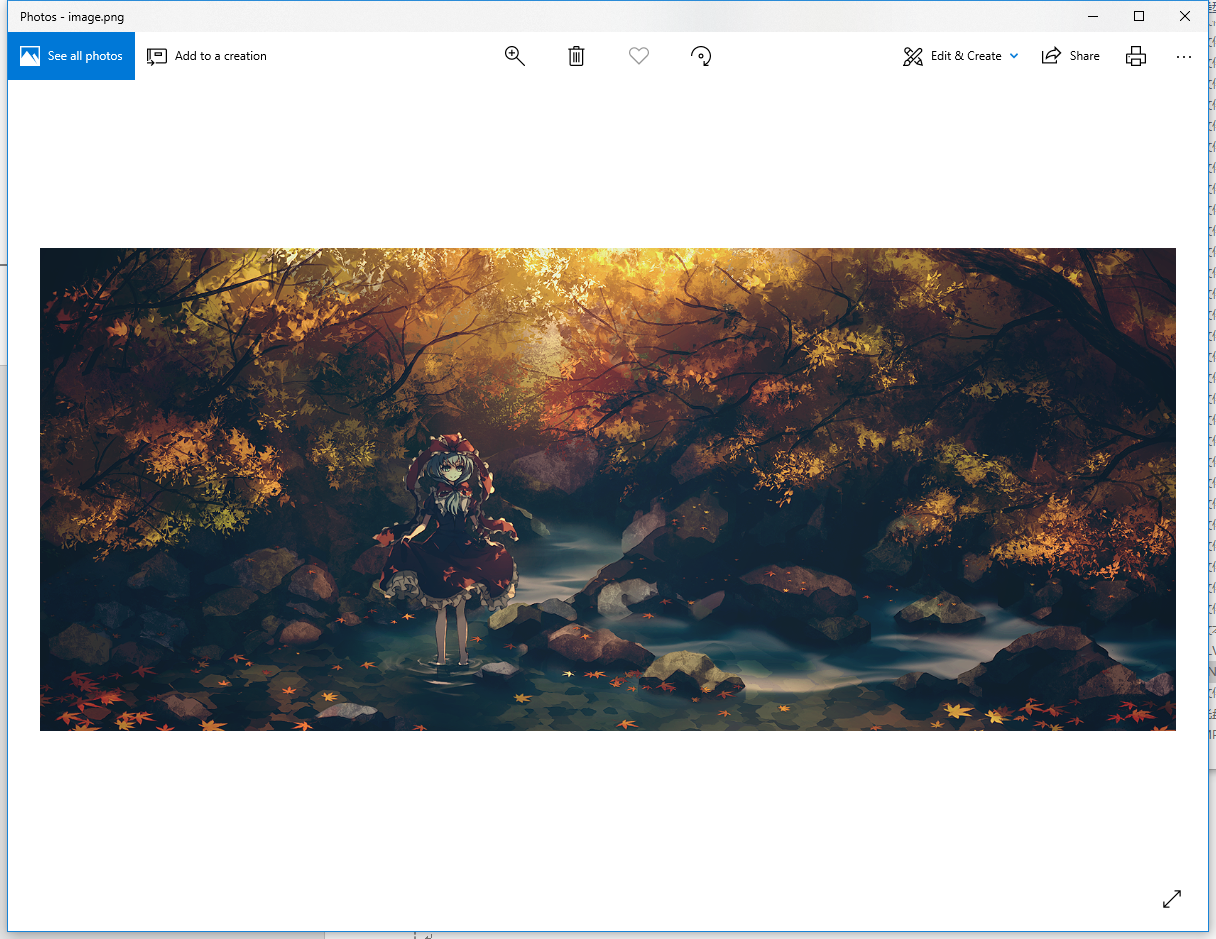


⑧随意取一个文件夹，在那个文件夹下创建了一个文件，然后写入图片。能正常写入。

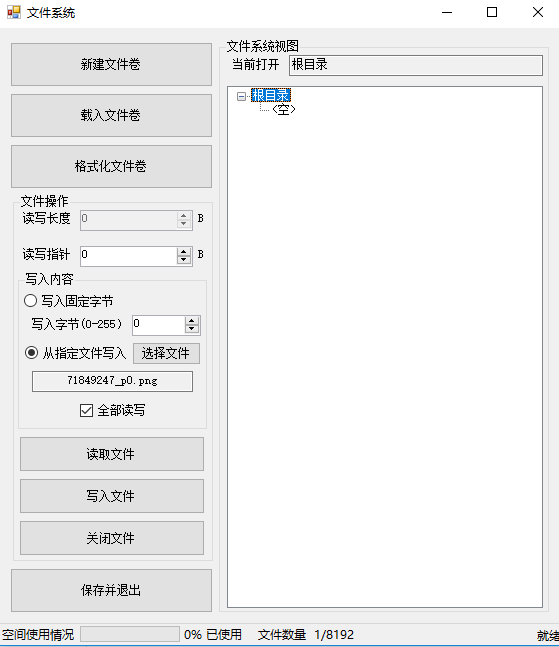


⑨再读出文件，图片能够正常打开，说明数据读写正确。





⑩最后删除整个folder目录，可以正确回收所有Inode和所有数据块。



模拟二级文件系统用户使用说明

# 1.安装说明

①使用本软件前，需要先安装.Net Framework 4.5.2以上运行库。

②运行FileSystem.exe即可使用本软件。

# 2.用户界面说明

实际用户界面参见图2.1。

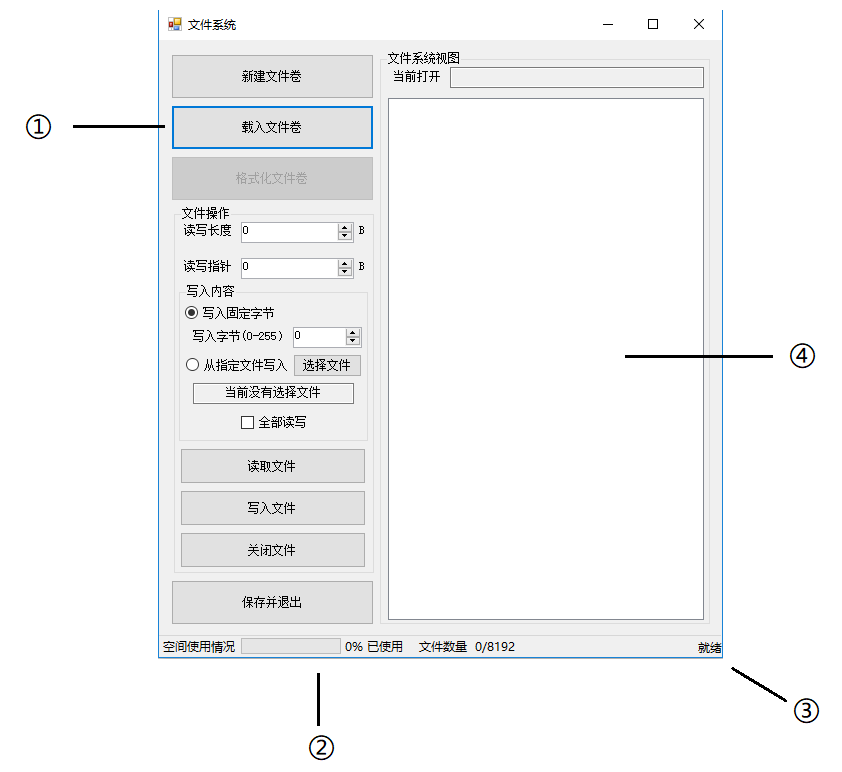


图2.1 文件系统主界面

区域描述：

①操作区域

②空间使用情况显示区域

③系统状态显示区域

④文件系统树状图显示区域

在区域④内点击右键可以进行文件系统的其他操作，右键菜单如图2.2所示。



图2.2 文件系统的右键菜单

# 3.功能使用说明

## 3.1 文件卷管理功能

●文件卷创建：点击“新建文件卷”按钮会打开一个存储文件对话框如图3.1所示。选择保存位置后即可在该位置创建一个文件卷。创建成功后会询问是否直接载入并格式化文件卷。

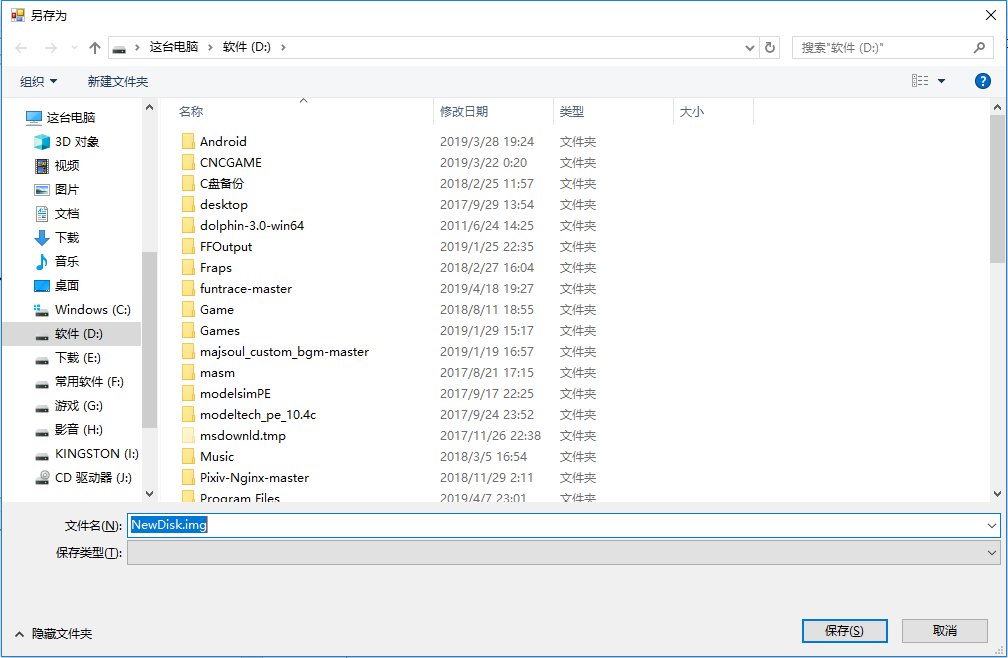


图3.1 存储文件对话框

●文件卷载入：点击“载入文件卷”按钮会打开一个打开文件对话框如图3.2所示。选择打开文件后可以载入该文件卷。

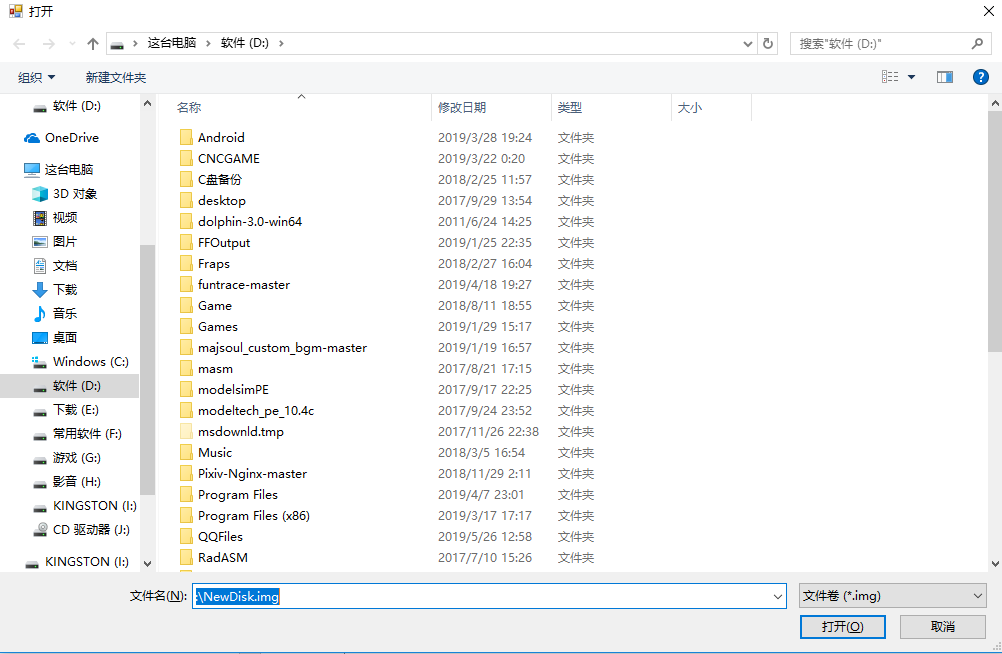


图3.2 打开文件对话框

●格式化文件卷：点击“格式化文件卷”，会弹出确认对话框。确认后即可格式化文件卷。也可以在创建文件卷后点击确认直接格式化。

## 3.2 文件/文件夹创建功能

单击文件系统树状视图中的一个文件夹结点，再点击右键打开右键菜单，点击“新建文件”或“新建文件夹”或按下F1或F2键后会弹出输入窗口如图3.3所示。输入文件/文件夹的名称后点击“确定”即可创建文件/文件夹。

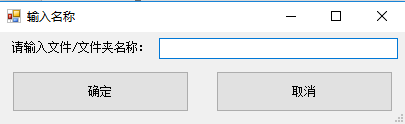


图3.3 名称输入窗口

## 3.3 文件/文件夹删除功能

单击文件系统树状视图中的一个文件夹结点，右键打开菜单，点击“删除”或按下del键即可删除文件/文件夹。当删除文件夹时系统会提示将删除文件夹下所有文件和文件夹，若点击确定即可递归删除整个文件夹。

## 3.4 文件/文件夹打开/关闭功能

双击文件系统树状视图中除“<空>”节点以外的任意一个节点均会打开该节点对应的数据文件或目录文件。当前打开的文件会在树状视图上方的“当前打开”文本框中显示。点击“关闭文件”即可关闭当前打开文件。

## 3.5 列出目录功能

点击文件夹结点旁边的“+”按钮或双击文件夹结点即可展开该文件夹，系统将会列出该文件夹目录内的所有文件/文件夹。需要注意的时这同时会将打开文件切换至该目录文件。在选中文件夹结点时右键打开菜单点击“全部展开”将会递归展开该文件夹和该文件夹内所有文件夹。这时打开文件会切换至最后打开的文件夹对应的目录文件。

## 3.6 文件读写指针定位及读写功能

设置“读写指针”数字框内数值即可设置文件读写指针。在“写入内容”内可以选择要写入的内容。若选择“写入固定字节”将会写入数字框内表示的字节，。若选择“从指定文件写入”则需要先点击“选择文件”以选择一个要获取数据的文件。也可以通过将文件直接拖动到主窗口内以设置文件。这时会从文件开头获取数据写入文件。读写的字节数量取决于是否勾选了“全部读写”。当未勾选时，读写字节数量为“读写长度”数字框内的数量。当勾选时，写数量为选择的外部文件长度，读数量为打开的系统内文件长度。当读取文件完成时，会出现图3.1的对话框以保存读出的数据。

课程设计总结

本次课程设计，我从零开始独立设计了一个具有友好图形操作界面的模拟二级文件系统。在系统的设计过程中，我对于操作系统中的文件系统有了较为深刻的理解。同时我也在充分理解了Unix V6++文件系统的前提下，对文件系统做出了一些改进，比如将空闲块能够索引的空闲块数量提升到128，充分利用SuperBlock占用了两个数据块，使用Unicode以允许所有语言的文件名，提供树状视图的文件系统视图以取代命令行等等。同时，使用C#进行程序的编写，在方便了我进行程序设计的同时，也让我对面向对象语言的特性理解变得更加深刻。总的来说，完成这次课程设计的过程让我的算法分析能力，编程能力和文档编写能力都有了长足的进步，十分具有意义。