**《操作系统课程设计》**

**（2024/2025学年第一学期）**

**指导教师： 田秋红**

**郭奕亿**

**班级：22计算机科学与技术3班**

**学号：2022334323029**

**姓名：张雅瑞**

**一、题目：**

Linux内核代码分析、新增系统调用、多用户文件系统。

**二、设计目的：**

操作系统原理是计算机专业的核心课程。本课程设计的目的旨在加深学生对计算机操作系统内核的理解，提高对操作系统内核的分析与扩展能力。在课程理论教学中，较多地是讲解操作系统理论和实现原理，为将来在基于Linux的嵌入式系统开发或在Java虚拟机上的软件开发工作奠定基础。

**三、计划安排：**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **时间** | | **地点** | **工作内容** | **指导教师** |
| 11.27 | 下午 | 10-414 | 任务布置，相关要求介绍与选题 | 田秋红、郭奕亿 |
| 晚上 | 10-414 | 完成选题报告和需求分析文档 | 田秋红、郭奕亿 |
| 12.4 | 下午 | 10-414 | 算法分析与基本设计 | 田秋红、郭奕亿 |
| 晚上 | 10-414 | 完成基本设计文档 | 田秋红、郭奕亿 |
| 12.11 | 下午 | 10-414 | 编写代码 | 田秋红、郭奕亿 |
| 晚上 | 10-414 | 编写代码 | 田秋红、郭奕亿 |
| 12.18 | 下午 | 10-414 | 程序测试 | 田秋红、郭奕亿 |
| 晚上 | 10-414 | 程序测试 | 田秋红、郭奕亿 |
| 12.25 | 下午 | 10-414 | 撰写课程设计报告，答辩 | 田秋红、郭奕亿 |
| 晚上 | 10-414 | 撰写课程设计报告，答辩 | 田秋红、郭奕亿 |

**四、参考资料**

1. Gray Nutt．Kernel Projects for Linux（影印版）．北京：机械工业出版社,2002
2. 李善平，郑扣根．Linux操作系统计实验教程．北京：机械工业出版社,1999
3. 印旻．Java语言与面向对象程序设计．北京：清华大学出版社，2000
4. https://blog.csdn.net/qq\_40989769/article/details/125223591超专业解析|linux文件系统的底层架构及其工作原理
5. https://www.cnblogs.com/theseventhson/p/15622853.html[linux源码解读（三）：文件系统——inode](https://www.cnblogs.com/theseventhson/p/15622853.html)
6. https://www.cnblogs.com/theseventhson/p/15643658.html[linux源码解读（五）：文件系统——文件和目录的操作](https://www.cnblogs.com/theseventhson/p/15643658.html)

**目 录**

[1. 引言 3](#_Toc186474030)

[1.1 任务要求 3](#_Toc186474031)

[1.2 选题 3](#_Toc186474032)

[2 需求分析与设计 4](#_Toc186474033)

[2.1需求分析 4](#_Toc186474034)

[2.1.1 新增系统调用 4](#_Toc186474035)

[2.1.2 文件系统 5](#_Toc186474036)

[2.2系统框架和流程 5](#_Toc186474037)

[2.2.1 Linux文件系统源码解读绘制功能框架图 5](#_Toc186474038)

[2.2.2 本系统功能框架和流程图 7](#_Toc186474039)

[2.3 文件系统流程和模块描述 8](#_Toc186474040)

[2.3.1 类图 8](#_Toc186474041)

[2.3.2 用例图 9](#_Toc186474042)

[2.3.3 顺序图 9](#_Toc186474043)

[3. 数据结构 10](#_Toc186474044)

[3.1 “生产者-消费者”问题 10](#_Toc186474045)

[3.1.1  共享缓冲区 10](#_Toc186474046)

[3.1.2 消费者 11](#_Toc186474047)

[3.1.3 生产者 11](#_Toc186474048)

[3.2 文件系统的数据结构 12](#_Toc186474049)

[3.2.1 用户类 12](#_Toc186474050)

[3.2.2 文件系统节点类 13](#_Toc186474051)

[3.2.3 文件系统类 14](#_Toc186474052)

[3.2.4 命令行界面类 15](#_Toc186474053)

[4.关键技术 16](#_Toc186474054)

[4.1 “生产者-消费者”问题 16](#_Toc186474055)

[4.1.1 多线程环境中的共享资源访问 16](#_Toc186474056)

[4.1.2多线程环境下生成随机线程 18](#_Toc186474057)

[4.2 多用户文件系统 20](#_Toc186474058)

[4.2.1 多用户权限 20](#_Toc186474059)

[4.2.2 文件的存储和索引 22](#_Toc186474060)

[4.2.3 文件保护 24](#_Toc186474061)

[5. 运行结果 26](#_Toc186474062)

[5.1 运行环境 26](#_Toc186474063)

[5.2 服务模式 26](#_Toc186474064)

[5.3 运行结果 27](#_Toc186474065)

[5.3.1 “生产者-消费者”问题 27](#_Toc186474066)

[5.3.2 多用户文件系统 31](#_Toc186474067)

[6. 调试和改进 37](#_Toc186474068)

[6.1 调试过程 37](#_Toc186474069)

[6.2 改进措施 38](#_Toc186474070)

[7. 心得和结论 38](#_Toc186474071)

[7.1 结论和体会 38](#_Toc186474072)

[7.2 进一步改进方向 38](#_Toc186474073)

[7.3 分析设计方案对系统安全的影响 38](#_Toc186474074)

# 1. 引言

操作系统作为计算机系统的中枢，负责管理和调度硬件资源与软件程序，确保系统的高效稳定运行。在信息技术迅猛发展的当下，操作系统的稳定性、安全性和性能成为衡量计算机系统优劣的重要标准。对于计算机专业的学生而言，深入理解操作系统的内核机制和文件系统原理至关重要，这不仅有助于他们掌握计算机系统的工作原理，也是未来从事相关领域工作的必要基础。本课程设计的目的在于通过具体的实践项目，加深学生对操作系统内核的理解，并提高他们分析和扩展操作系统内核的能力，为未来在Linux嵌入式系统开发或Java虚拟机上的软件开发等领域的工作打下坚实的基础。

## 1.1 任务要求

本次课程设计要求学生完成对Linux内核代码的深入分析，掌握其工作原理和实现机制。学生需要设计并实现一个新的系统调用，包括确定其名称、参数、功能和预期效果，并将其成功集成到Linux内核中。此外，学生还需设计并实现一个多用户文件系统，该系统需要支持用户权限管理、文件和目录操作、文件保护以及数据持久化等高级功能。课程设计还包括撰写详细的报告文档，涵盖需求分析、设计文档、测试报告和用户手册等。最后，学生需要准备答辩，清晰地向评审展示项目的设计思路、实现过程和最终结果。

## 1.2 选题

1. Linux内核代码分析
2. Linux内核代码的层次分析
3. 调度程序代码段的分析
4. 系统调用内部数据结构以及执行过程的分析
5. 内核调试基本技术
6. 用JAVA语言模仿“生产者—消费者问题”
7. 通过Java语言中的wait（）和notify（）命令模拟操作系统中的P/V操作；
8. 为每个生产者／消费者产生一个线程，设计正确的同步算法
9. 每个生产者和消费者对有界缓冲区进行操作后，即时显示有界缓冲区的当前全部内容、当前指针位置和生产者／消费者线程的自定义标识符。
10. 生产者和消费者各有两个以上。
11. 多个生产者或多个消费者之间须共享对缓冲区进行操作的函数代码。

（3）多用户文件系统

设计一个多用户文件系统，理解文件系统的层次结构，完成基本的文件系统create、open、close、read/write等基本功能，并实现文件保护操作。实现以此为基础加入自己设计功能的小型文件系统 。

# 2 需求分析与设计

## 2.1需求分析

### 2.1.1 新增系统调用

#### 2.1.1.1 功能需求

（1）新系统调用的设计与实现

设计一个新的系统调用，包括确定其名称、参数、功能和预期的效果。

新系统调用能够实现将Hello, Linux输出到系统日志的功能。

（2）系统调用的集成

将新的系统调用集成到Linux内核中。

确保系统调用与内核其他部分协同工作，并不会引入冲突或不稳定性。

（3）测试程序开发

编写应用程序测试新的系统调用，确保其按照预期工作。

测试程序能够验证系统调用的功能，并且能够输出测试结果供分析。

#### 2.1.1.2 性能需求

新系统调用应该高效执行，不会对系统性能造成显著影响。必须确保系统调用在高负载下依然能够保持良好性能。

#### 2.1.1.3 兼容性需求

新系统调用应该与现有的Linux内核版本兼容，不破坏现有的用户空间程序。

### 2.1.2 文件系统

#### 2.1.2.1功能需求

（1）多用户访问控制

设计和实现一个健全的用户权限管理机制，确保不同用户的文件访问权限得到合理分配和管理。

实现用户认证功能，以支持用户登录和权限验证。

（2）文件和目录操作

支持文件和目录的创建、删除、读取和写入操作。

提供文件搜索、移动和复制等高级功能。

（3）文件保护

实现文件锁定机制，防止文件在使用过程中被其他用户篡改。

设计文件权限系统，允许用户设置文件的读、写、执行权限。

（4）数据持久化

使用序列化和反序列化将文件系统数据结构存储到硬盘，支持终端多开。

#### 2.1.1.2 性能需求

文件系统的读写操作需要高效，减少I/O延迟，提高数据吞吐率。

#### 2.1.1.3 安全需求

确保文件数据不会因为系统故障或其他原因而丢失。文件系统应能够抵抗常见的安全攻击，如路径遍历、权限绕过等。

## 2.2系统框架和流程

### 2.2.1 Linux文件系统源码解读绘制功能框架图

**用户空间和内核空间**

* **用户空间**：是普通用户程序运行的区域，像文本编辑器、编译器这类程序都在此运行。
* **内核空间**：为操作系统核心部分运行的场所，它能直接与硬件交互，并向用户空间提供系统服务。

**虚拟文件系统（VFS）**

VFS 处于 Linux 内核之中，是一个抽象层，其作用在于给不同的文件系统打造统一接口。如此一来，用户在面对不同类型文件系统时，能运用诸如 open、read、write 等相同的系统调用。同时，VFS 也定义了一套用于文件系统操作的通用数据结构以及 API。

**具体文件系统的实现**

这部分涵盖了具体文件系统的实现代码。各个文件系统都具备自身独特的数据结构与算法，用来管理磁盘上的数据。它们负责实际的数据存储、检索、元数据管理、权限控制以及其他和文件相关的各类操作。

**块设备层**

块设备层主要承担管理物理存储设备（如硬盘、SSD 等）的职责，提供读写块设备的接口，能把文件系统的读写请求转变为对应的物理设备操作。此外，该层可能还包含块缓存或者页面缓存，以此提升文件操作方面的性能。

**硬件层**

硬件层属于物理存储设备所处的层级，像硬盘驱动器、固态硬盘、USB 存储设备等都在这一层。这些设备借助硬件接口和计算机的其他组件实现通信。

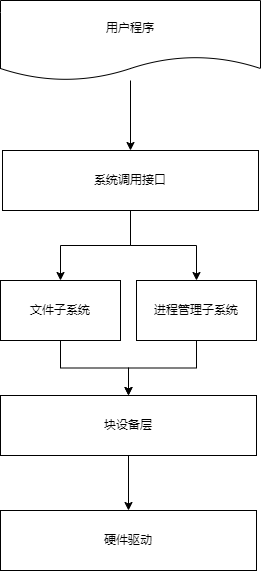


图2.1 Linux文件系统框架图

### 2.2.2 本系统功能框架和流程图

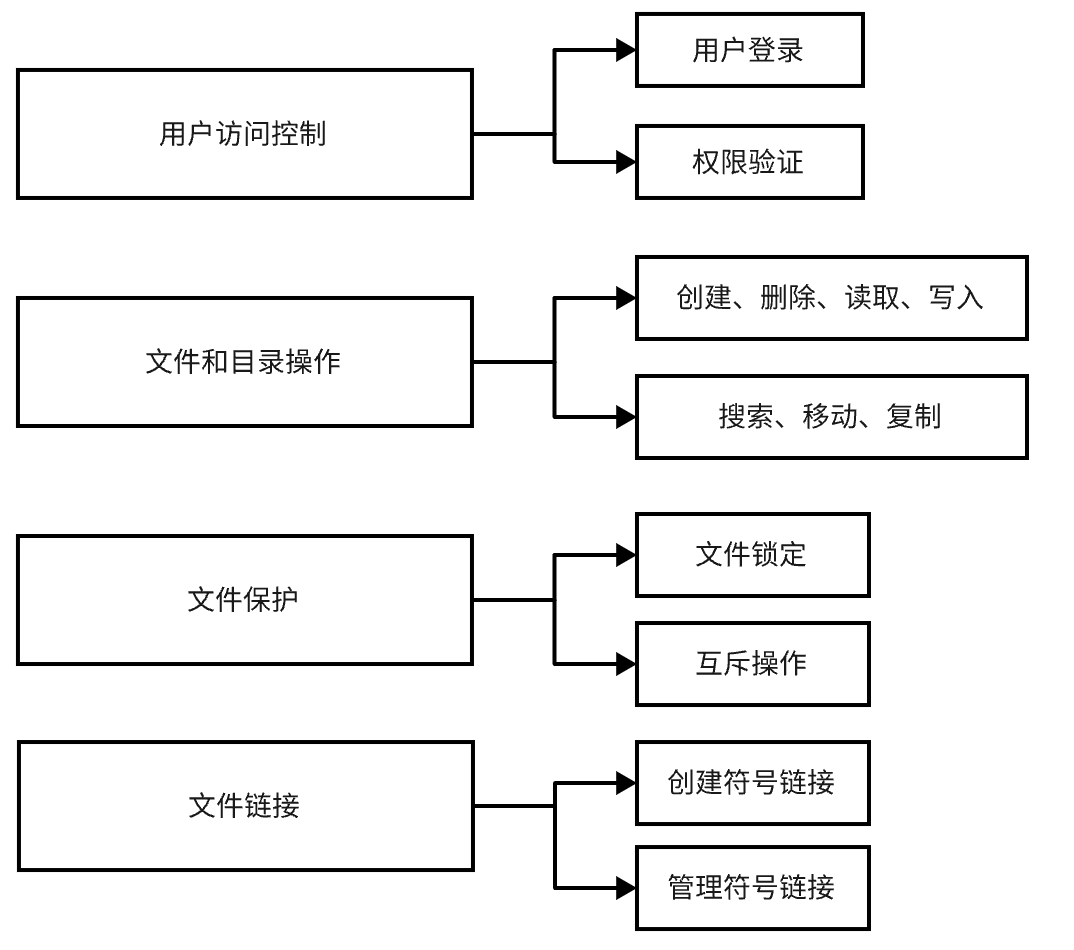


图2.2 功能框架图

该功能框架图展示了文件系统的主要功能模块，包括用户访问控制（用户登录和权限验证）、文件和目录操作（创建、删除、读取、写入、搜索、移动、复制）、文件保护（文件锁定和互斥操作）以及文件链接（创建和管理符号链接）。每个模块及其子功能共同确保文件系统的安全性、可用性和有效管理。

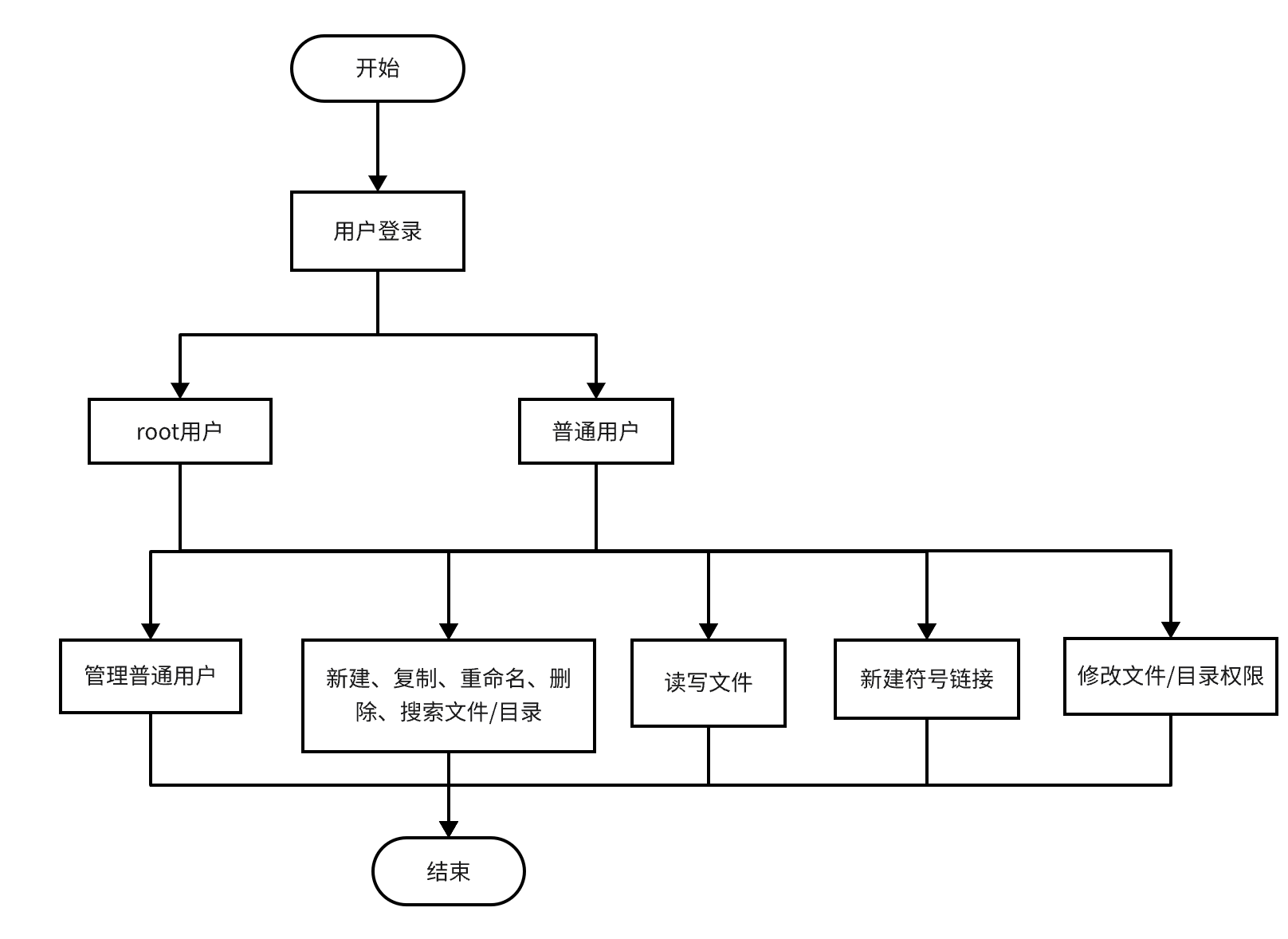


图2.3 系统流程图

该流程图展示了用户登录后的操作流程。用户登录后，根据身份分为root用户和普通用户。root用户可以管理普通用户，并进行文件和目录的各种操作（如创建、复制、重命名、删除、搜索等）。普通用户则可以进行文件的读写、新建符号链接以及修改文件/目录权限的操作。操作完成后，流程结束。

## 2.3 文件系统流程和模块描述

### 2.3.1 类图

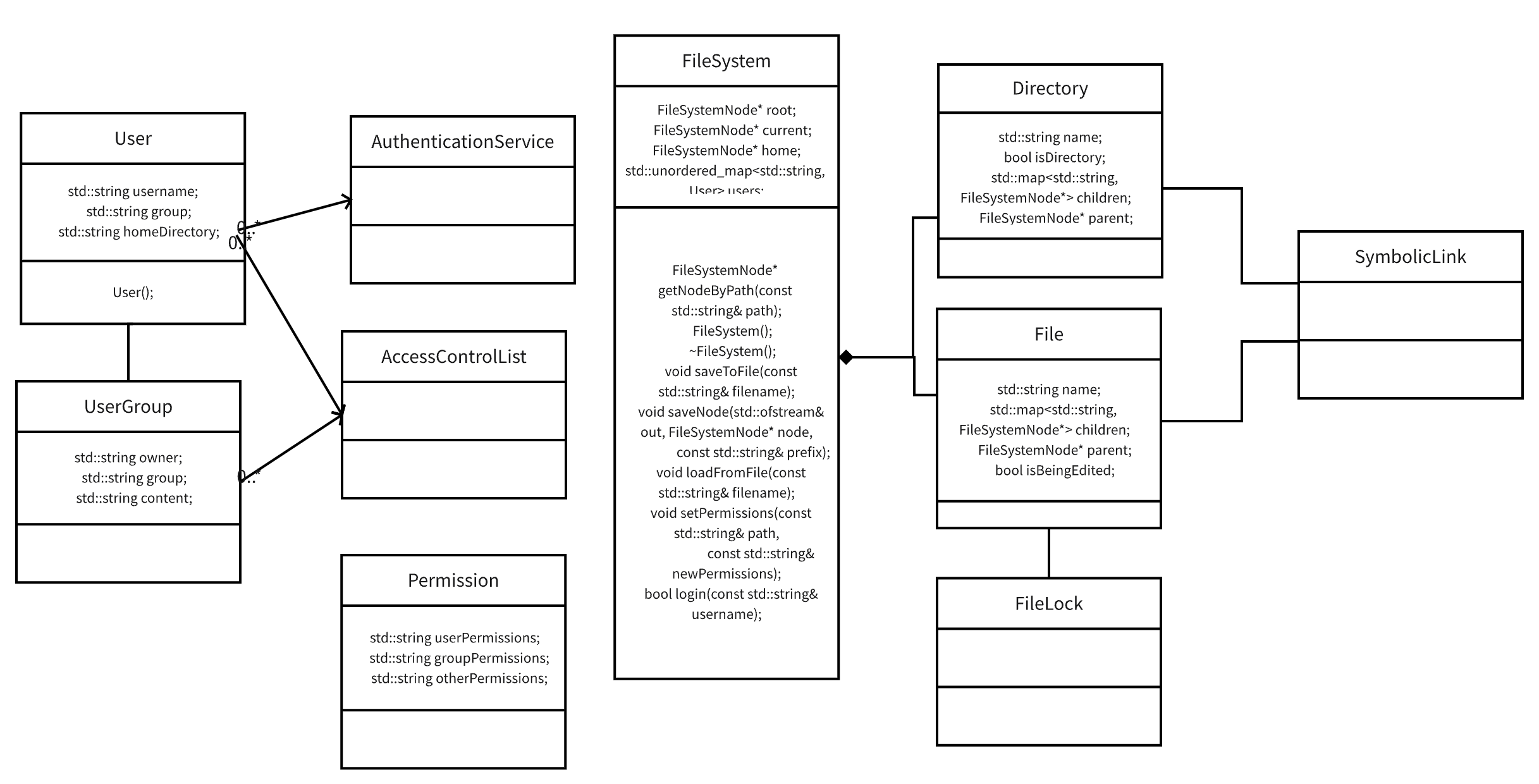


图2.3 类图

该类图展示了文件系统的主要组成部分和它们之间的关系。主要包括用户（User）及其所属用户组（UserGroup），用户权限（Permission），用于验证用户身份的认证服务（AuthenticationService），访问控制列表（AccessControlList），以及文件系统（FileSystem）中的目录（Directory）、文件（File）、符号链接（SymbolicLink）和文件锁（FileLock）。文件系统管理文件和目录的层次结构，并提供相应的操作方法。用户通过认证服务和访问控制列表来进行权限验证和访问控制。

### 2.3.2 用例图

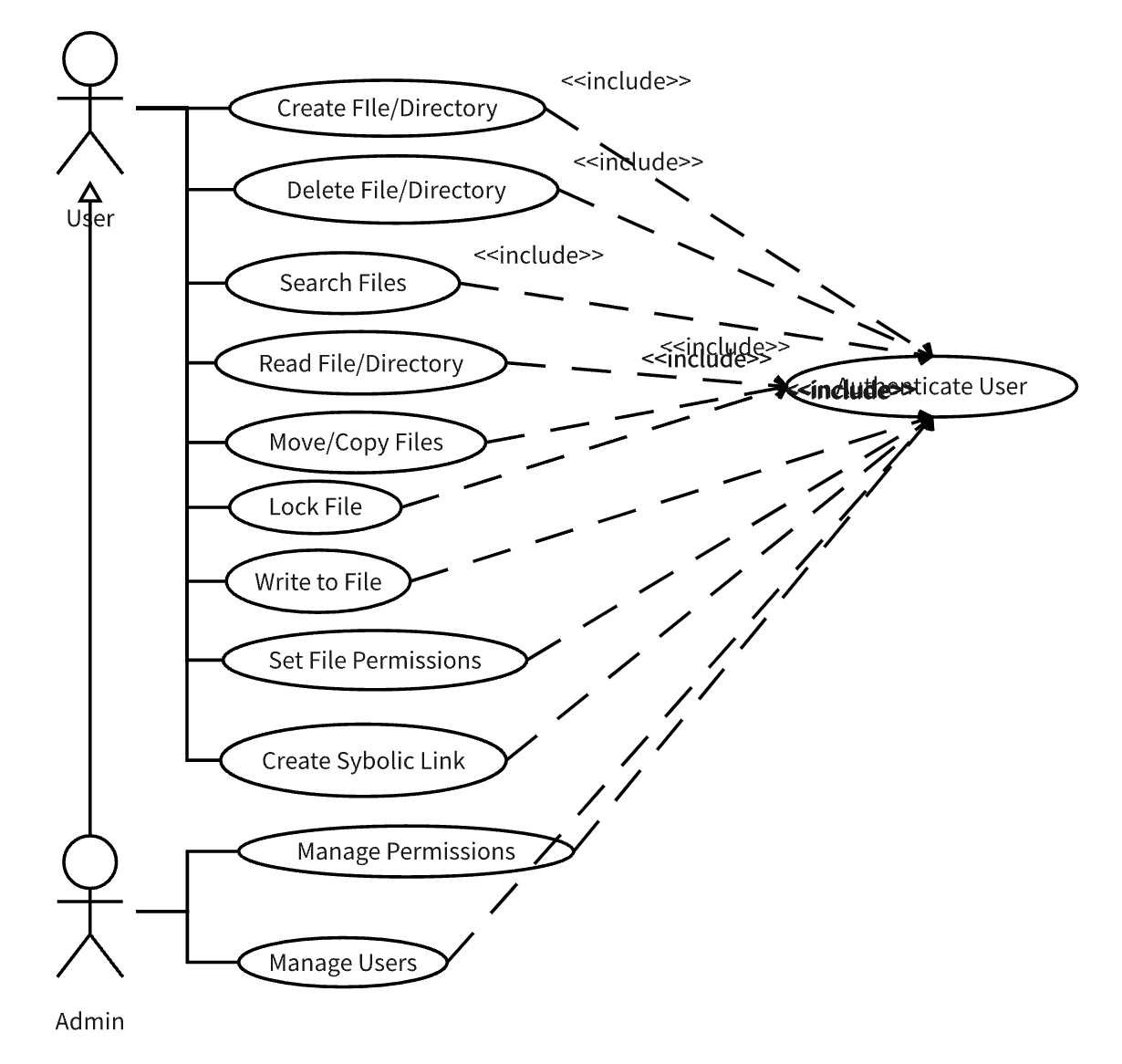


图2.4用例图

### 2.3.3 顺序图

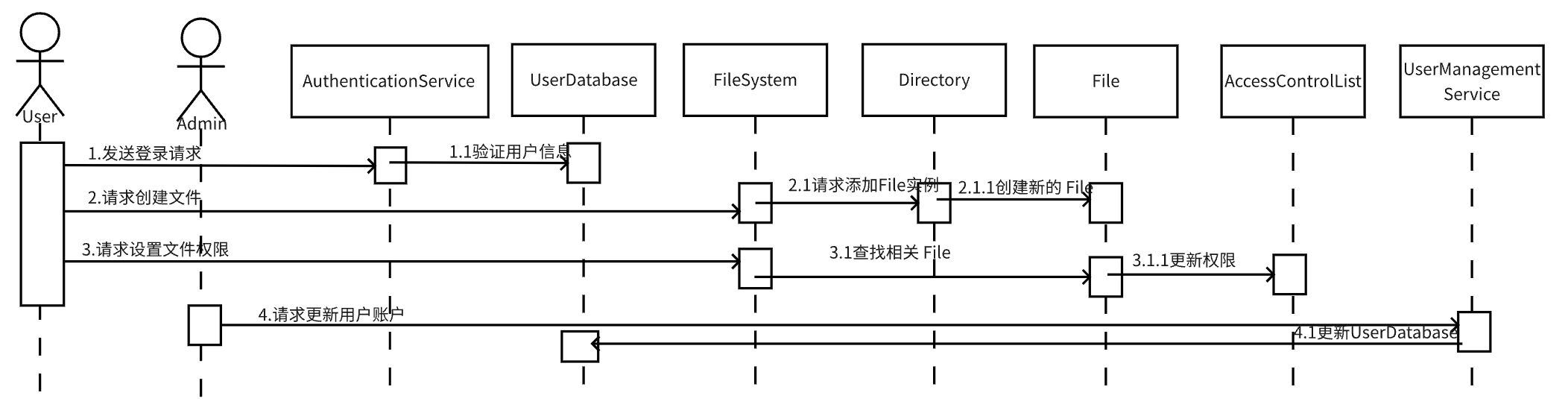


图2.5顺序图

# 3. 数据结构

## 3.1 “生产者-消费者”问题

### 3.1.1  共享缓冲区

共享缓冲区的数据结构在上述代码中是一个 List<Integer> 类型的列表，用于存储生产者生产的产品编号。

* **共享缓冲区（list）**：这是一个 List<Integer> 类型的列表，用于存储生产者生产的产品编号。它是生产者和消费者之间的共享资源。
* **同步控制**：由于这是一个共享资源，必须对其进行同步控制，确保在多线程环境下的线程安全性。
* **显示缓冲区状态**：Controller 类中的 showList 方法用于显示缓冲区的当前状态，帮助观察生产者和消费者对缓冲区的操作情况。

**数据结构**

1. **类型**：
   * List<Integer>：这是一个泛型列表，专门用于存储整数类型的数据。在这个上下文中，每个整数代表一个产品的编号。
2. **用途**：
   * 存储生产者生产的产品编号。
   * 作为生产者和消费者之间的共享缓冲区，生产者将产品编号添加到该列表中，消费者从该列表中移除产品编号进行消费。
3. **特点**：
   * **共享资源**：这是一个共享资源，多个生产者和消费者线程会并发访问它。
   * **同步控制**：需要对该列表进行同步控制，以避免并发访问引起的数据不一致或其他线程安全问题。

通过这些机制，生产者和消费者能够在多线程环境下安全、高效地进行生产和消费操作，同时能够实时监控缓冲区的状态。

**示例输出**

假设缓冲区中有三个产品编号，分别为 123, 456, 789，则 showList 方法的输出如下：

|  |
| --- |
| =============== 缓冲区情况 =============  当前指针位置：0 -------- 对应商品编号：R123  当前指针位置：1 -------- 对应商品编号：R456  当前指针位置：2 -------- 对应商品编号：R789  ====================================== |

### 3.1.2 消费者

1. **共享缓冲区（list）**：
   * 类型：List<Integer>
   * 用途：用于存储生产者生产的产品。消费者从该列表中移除产品进行消费。
   * 特点：这是一个共享资源，多个生产者和消费者线程会并发访问它，因此需要进行同步控制。
2. **消费次数（TIMES）**：
   * 类型：int
   * 用途：表示每个消费者线程消费产品的次数。在代码中，这个值被设置为3，即每个消费者线程会消费3个产品。

### 3.1.3 生产者

1. **共享缓冲区（list）**：
   * 类型：List<Integer>
   * 用途：用于存储生产者生产的产品。生产者将生产的产品添加到该列表中，消费者从该列表中移除产品进行消费。
   * 特点：这是一个共享资源，多个生产者和消费者线程会并发访问它，因此需要进行同步控制。
2. **最大容量（MAX\_SIZE）**：
   * 类型：int
   * 用途：表示缓冲区的最大容量。当缓冲区中的产品数量达到该值时，生产者线程会进入等待状态，直到有消费者消费产品并腾出空间。
3. **生产次数（TIMES）**：
   * 类型：int
   * 用途：表示每个生产者线程生产产品的次数。在代码中，这个值被设置为3，即每个生产者线程会生产3个产品。
4. **随机数生成器（RANDOM）**：
   * 类型：ThreadLocalRandom
   * 用途：用于生成随机的产品编号。ThreadLocalRandom 是一个线程安全的随机数生成器，适合在多线程环境下使用。

## 3.2 文件系统的数据结构

在本课程设计的多用户文件系统中，数据结构是实现高效文件管理和访问控制的关键。以下是主要使用的数据结构及其描述：

### 3.2.1 用户类

此类用于管理用户的信息和权限。它包含以下属性：

* username（用户名）：标识用户的唯一字符串。
* group（用户组）：用户所属的组，用于实现组级别的权限管理。
* homeDirectory（家目录）：用户的主目录路径。

|  |
| --- |
| class User {  public:  std::string username;  std::string group;  std::string homeDirectory;  User();  }; |

### 3.2.2 文件系统节点类

此类表示文件系统中的一个节点，可以是文件或目录。它包含以下属性和方法：

* name（名称）：节点的名称。
* isDirectory（是否为目录）：标识该节点是文件还是目录。
* owner（所有者）：文件或目录的所有者用户名。
* group（用户组）：文件或目录的所属用户组。
* content（内容）：仅文件节点有效，存储文件的内容。
* children（子节点）：仅目录节点有效，存储子节点的映射（子节点名称到子节点对象的映射）。
* userPermissions（用户权限）、groupPermissions（组权限）、otherPermissions（其他权限）：存储不同类别用户对节点的访问权限。
* parent（父节点）：指向父节点的指针，用于导航和路径解析。
* isBeingEdited（是否正在被编辑）：标识文件是否正在被编辑，用于文件锁定。

|  |
| --- |
| class FileSystemNode {  public:  std::string name;  bool isDirectory;  std::string owner;  std::string group;  std::string content;  std::map<std::string, FileSystemNode\*> children;  std::string userPermissions;  std::string groupPermissions;  std::string otherPermissions;  FileSystemNode\* parent;  bool isBeingEdited;  FileSystemNode();  ~FileSystemNode();  void setPermissions(const std::string& newPermissions);  std::string getPermissions();  std::string formatPermissions();  }; |

## 3.2.3 文件系统类

模拟文件系统的主体类，包含以下关键属性和方法：

* root（根节点）、current（当前节点）、home（家目录）：文件系统的核心节点。
* users（用户管理）：存储系统中所有用户的信息。
* currentUser（当前用户）：当前登录的用户。
* getNodeByPath（根据路径获取节点）：辅助方法，用于根据路径查找特定的文件系统节点。
* saveToFile / loadFromFile：用于持久化文件系统状态，将文件系统状态保存到文件或从文件中加载。
* setPermissions / getPermissions：用于设置和获取文件或目录的权限。
* addUser / login：用于添加新用户和用户登录。
* checkPermissions：检查用户对特定节点的访问权限。
* create / remove：用于创建或删除文件或目录。
* changeDirectory / listDirectory：用于更改当前工作目录和列出目录内容。

|  |
| --- |
| class FileSystem {  private:  FileSystemNode\* root;  FileSystemNode\* current;  FileSystemNode\* home;  std::unordered\_map<std::string, User> users;  public:  User\* currentUser = nullptr;  FileSystemNode\* getNodeByPath(const std::string& path);  FileSystem();  ~FileSystem();  void saveToFile(const std::string& filename);  void saveNode(std::ofstream& out, FileSystemNode\* node,  const std::string& prefix);  void loadFromFile(const std::string& filename);  void setPermissions(const std::string& path,  const std::string& newPermissions);  bool login(const std::string& username);  bool checkPermissions(FileSystemNode\* node, char permissionType);  void create(const std::string& path, bool isDirectory, const std::string& owner, const std::string& group);  void remove(const std::string& path);  void changeDirectory(const std::string& path);  void listDirectory(bool longFormat = false);  void writeFile(const std::string& path, const std::string& content);  void readFile(const std::string& path);  std::string getCurrentDirectoryPath();  }; |

## 3.2.4 命令行界面类

这是一个简单的命令行界面类，用于模拟shell环境，允许用户执行诸如文件操作、用户管理等命令。它包含以下主要方法：

* split：用于将字符串按照给定的分隔符分割为子字符串。
* executeCommand：解析和执行用户输入的命令。
* run：启动shell的主循环，等待用户输入并响应。

|  |
| --- |
| class SimpleShell {  private:  FileSystem fs;  std::vector<std::string> split(const std::string& str, char delimiter);  void adduser(const std::vector<std::string>& args);  void su(const std::vector<std::string>& args);  void login(const std::vector<std::string>& args);  void editFile(const std::string& filename);  std::string convertNumericToSymbolic(const std::string& numeric);  void executeCommand(const std::string& command, const std::vector<std::string>& args);  public:  void run();  }; |

这些数据结构在整个文件系统模拟中起着至关重要的作用，不仅支持基本的文件操作，还实现了用户权限管理、文件保护和数据持久化等高级功能。

# 4.关键技术

## 4.1 “生产者-消费者”问题

“生产者-消费者”问题是一个经典的多线程同步问题，涉及多个生产者线程和多个消费者线程通过一个共享缓冲区进行数据的生产和消费。下面将详细分析代码是如何解决这一问题的。

### 4.1.1 多线程环境中的共享资源访问

通过 synchronized 关键字对共享资源 list 进行同步控制，使用 wait() 和 notifyAll() 方法实现线程间的通信和协调，确保线程安全。

在多线程环境中，访问共享资源需要进行同步控制，以避免数据不一致和竞争条件。代码中使用了以下技术来实现这一点：

1. **共享资源**：List<Integer> list 是生产者和消费者之间的共享缓冲区，用于存储生产的产品编号。
2. **同步块**：使用 synchronized 关键字对共享资源 list 进行同步控制，确保同一时间只有一个线程可以访问 list，避免数据竞争和不一致。
3. **条件变量**：使用 wait() 和 notifyAll() 方法实现线程间的通信和协调。
   * **wait()**：当缓冲区满（生产者）或空（消费者）时，线程会进入等待状态，释放锁，等待其他线程唤醒。
   * **notifyAll()**：当生产者生产了新产品或消费者消费了产品后，唤醒所有等待的线程，以便它们继续执行。

|  |
| --- |
| // Producer 类中的 run 方法  @Override  public void run() {  for (int i = 0; i < TIMES; i++) {  synchronized (list) {  while (list.size() == this.MAX\_SIZE) {  try {  System.out.println("生产者：" + Thread.currentThread().getName() + "\t仓库已达最大容量，进入阻塞状态");  list.wait();  System.out.println("生产者：" + Thread.currentThread().getName() + "\t被唤醒，退出阻塞状态");  } catch (Exception e) {  e.printStackTrace();  }  }  int number = RANDOM.nextInt(8000);  System.out.println("生产者：" + Thread.currentThread().getName() + "\t生产的产品编号为：R" + number);  list.add(number);  Controller.showList(list);  list.notifyAll();  try {  Thread.sleep(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  System.out.println("生产者：" + Thread.currentThread().getName() + "\t停止生产，进程结束！");  }  // Customer 类中的 run 方法  @Override  public void run() {  for (int i = 0; i < TIMES; i++) {  synchronized (list) {  while (list.isEmpty()) {  try {  System.out.println("消费者：" + Thread.currentThread().getName() + "\t仓库已空，进入阻塞状态");  list.wait();  System.out.println("消费者：" + Thread.currentThread().getName() + "\t被唤醒，退出阻塞状态");  } catch (Exception e) {  e.printStackTrace();  }  }  Integer removed = list.remove(0);  System.out.println("消费者：" + Thread.currentThread().getName() + "\t购买的商品编号为：R" + removed);  Controller.showList(list);  list.notifyAll();  try {  Thread.sleep(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  System.out.println("消费者：" + Thread.currentThread().getName() + "\t消费结束，进程结束！");  } |

### 4.1.2多线程环境下生成随机线程

使用 ThreadLocalRandom 类生成随机数，确保在多线程环境下的性能和线程安全。

在多线程环境中生成随机数，可以使用 ThreadLocalRandom 类，它是线程安全的，并且比 Random 类在多线程环境下性能更好。ThreadLocalRandom 提供了每个线程独立的随机数生成器，避免了多线程竞争。

以下是代码中使用 ThreadLocalRandom 生成随机数的部分：

|  |
| --- |
| import java.util.concurrent.ThreadLocalRandom;  class Producer implements Runnable {  private static final ThreadLocalRandom RANDOM = ThreadLocalRandom.current();  private static final int TIMES = 3;  private final List<Integer> list;  private final int MAX\_SIZE;  Producer(List<Integer> list, int size) {  this.list = list;  this.MAX\_SIZE = size;  }  @Override  public void run() {  for (int i = 0; i < TIMES; i++) {  synchronized (list) {  while (list.size() == this.MAX\_SIZE) {  try {  System.out.println("生产者：" + Thread.currentThread().getName() + "\t仓库已达最大容量，进入阻塞状态");  list.wait();  System.out.println("生产者：" + Thread.currentThread().getName() + "\t被唤醒，退出阻塞状态");  } catch (Exception e) {  e.printStackTrace();  }  }  int number = RANDOM.nextInt(8000);  System.out.println("生产者：" + Thread.currentThread().getName() + "\t生产的产品编号为：R" + number);  list.add(number);  Controller.showList(list);  list.notifyAll();  try {  Thread.sleep(500);  } catch (InterruptedException e) {  e.printStackTrace();  }  }  }  System.out.println("生产者：" + Thread.currentThread().getName() + "\t停止生产，进程结束！");  }  } |

## 4.2 多用户文件系统

### 4.2.1 多用户权限

在操作系统和文件系统中，多用户权限的功能是至关重要的安全特性。它有以下几个主要目的：

* 保护用户数据：确保用户的文件和数据不被未授权的用户访问或修改。
* 系统安全：通过限制对关键系统文件和目录的访问，保护系统免受恶意软件或未授权用户的破坏。
* 隐私保护：确保用户的隐私信息不被其他用户窥探。
* 资源管理：合理分配和管理系统资源，避免某些用户的过度使用对其他用户造成不公平或影响系统稳定性。

#### 4.2.1.1 基本思想和步骤

1. 基本思想

用户权限管理基于三个主要概念：用户（User）、用户组（Group）、权限（Permissions）。每个文件或目录都有一个所有者（Owner）和一个关联的用户组。权限通常分为三类：读（Read）、写（Write）、执行（Execute），分别对应不同的访问级别。

1. 基本步骤

用户和组识别：在系统中识别不同的用户和用户组。

权限分配：为每个文件和目录指定一个所有者和用户组，分配相应的权限。

访问控制：当用户尝试访问文件或目录时，系统检查该用户是否拥有足够的权限。

权限修改：允许文件或目录的所有者更改权限设置。

#### 4.2.1.2代码实现

|  |
| --- |
| 定义权限：在 FileSystemNode 类中定义了用户权限（userPermissions）、组权限（groupPermissions）和其他用户权限（otherPermissions）。  std::string userPermissions;  std::string groupPermissions;  std::string otherPermissions;  设置和获取权限：通过 setPermissions 和 getPermissions 方法来设置和获取节点的权限。  void setPermissions(const std::string& newPermissions) {  if (newPermissions.size() == 9) {  userPermissions = newPermissions.substr(0, 3);  groupPermissions = newPermissions.substr(3, 3);  otherPermissions = newPermissions.substr(6, 3);  }  }  std::string getPermissions() const {  return userPermissions + groupPermissions + otherPermissions;  }  检查权限：在 FileSystem 类中实现 checkPermissions 方法来检查用户是否有权执行特定操作。  bool checkPermissions(FileSystemNode\* node, char permissionType) {  if (currentUser->username == "root") return true;  std::string permissions = node->getPermissions();  bool hasPermission = false;  // 检查用户权限  if (node->owner == currentUser->username) {  hasPermission = permissions[permissionType == 'r' ? 0 : (permissionType == 'w' ? 1 : 2)] != '-';  }  // ... 检查组权限 ...  // ... 检查其他用户权限 ...  return hasPermission;  }  执行权限检查：在执行像更改目录（cd）、读取文件（cat）等操作之前，先通过 checkPermissions 方法检查用户是否有相应的权限。  void changeDirectory(const std::string& path) {  if (path.empty() || path[0] != '/') {  std::cerr << "Error: Only absolute paths are supported\n";  return;  }  FileSystemNode\* node = getNodeByPath(path);  if (!node || !node->isDirectory) {  std::cerr << "Error: Directory does not exist\n";  return;  }  if (!checkPermissions(node, 'r')) {  std::cerr << "Error: Permission denied\n";  return;  }  current = node;  } |

通过这种方式，代码有效地模拟了一个基本的用户权限管理系统，允许对文件系统中的文件和目录进行访问控制。

### 4.2.2 文件的存储和索引

文件的存储和索引是文件系统中至关重要的功能。它涉及如何在磁盘上组织和管理文件数据，以便高效地存储和快速访问。

#### 4.2.2.1 基本思想和步骤

文件系统通过树状结构来组织文件和目录。每个文件或目录都是一个节点，节点之间通过指针或引用互相连接，形成层次结构。

* 节点结构：定义文件和目录的基本数据结构。
* 存储布局：确定文件数据在磁盘上的存储布局。
* 索引机制：实现文件和目录的索引，以便快速查找。
* 数据恢复：在系统重启或崩溃后，能够恢复文件系统的状态。

#### 4.2.2.1 代码实现

|  |
| --- |
| 保存文件系统状态：遍历文件系统的节点，并将节点的信息（如名称、类型、权限等）写入到文件中。  void saveToFile(const std::string& filename) {  std::ofstream out(filename);  if (!out) {  std::cerr << "Error opening file for writing: " << filename << std::endl;  return;  }  saveNode(out, root, "");  out.close();  }  保存节点信息：递归函数，用于保存每个节点的详细信息。  void saveNode(std::ofstream& out, FileSystemNode\* node, const std::string& prefix) {  if (node == nullptr) return;  out << prefix << node->name << " " << (node->isDirectory ? "d" : "f") << " " << node->owner << " " << node->group << " " << node->userPermissions << node->groupPermissions << node->otherPermissions << " " << node->isBeingEdited << "\n";  for (auto& child : node->children) {  saveNode(out, child.second, prefix + node->name + "/");  }  }  从文件加载文件系统状态:读取文件中的数据，根据这些数据恢复文件系统的状态。  void loadFromFile(const std::string& filename) {  std::ifstream in(filename);  if (!in) {  std::cerr << "Error opening file for reading: " << filename << std::endl;  return;  }  std::string line;  while (std::getline(in, line)) {  std::istringstream iss(line);  std::string path, type, owner, group, permissions;  bool isBeingEdited;  iss >> path >> type >> owner >> group >> permissions >> isBeingEdited;  FileSystemNode\* node = getNodeByPath(path);  if (node) {  // ... 更新已存在节点的属性 ...  } else {  // ... 创建新节点 ...  }  }  in.close();  } |

通过这种方式，代码实现了文件系统状态的持久化，保证了文件系统的信息在程序关闭后仍然可以被恢复和访问。这是确保数据安全和持久性的关键步骤。

### 4.2.3 文件保护

文件保护是计算机操作系统中一个重要的组成部分，它的主要目的是确保文件系统中的数据的安全性和完整性。文件保护包括以下几个方面：

读读允许（Read-Read Allow）：允许多个用户或进程同时读取同一个文件。这不会对文件内容造成影响，因此通常是安全的。

读写互斥（Read-Write Mutual Exclusion）：当一个用户或进程正在写入文件时，其他用户或进程不能读取该文件。这是为了防止在写入过程中读取到部分更新的数据，从而确保数据的一致性。

写写互斥（Write-Write Mutual Exclusion）：防止多个用户或进程同时写入同一个文件，因为这可能导致数据损坏或不一致。

#### 4.2.3.1 基本思想和步骤

实现文件保护的基本思想是通过文件权限和锁机制来控制对文件的访问。基本步骤包括：

设置文件权限：根据用户、用户组和其他用户设置不同的读、写、执行权限。

检查权限：在文件被访问时（如读取、写入、执行等），检查当前用户是否有足够的权限。

锁定文件：在写操作期间，锁定文件，防止其他写操作或不允许的读操作。

记录状态：记录文件是否正在被编辑，以便在其他用户或进程尝试访问时进行检查。

#### 4.2.3.2 代码实现

|  |
| --- |
| 权限控制：FileSystemNode 类中有 userPermissions、groupPermissions 和 otherPermissions 成员变量，用于存储不同类型用户的权限。  std::string userPermissions;  std::string groupPermissions;  std::string otherPermissions;  检查权限：FileSystem 类中的 checkPermissions 方法用于检查当前用户是否有权限执行特定操作。此方法已在 4.1 多用户权限章节中展示。  文件锁定：FileSystemNode 类中的 isBeingEdited 成员变量用于表示文件是否正在被编辑。如果 isBeingEdited 为 true，则其他用户无法进行写操作。  bool isBeingEdited;  读写操作：在执行读写操作之前，通过 checkPermissions 方法和 isBeingEdited 状态来决定是否允许该操作。  void editFile(const std::string& filename) {  FileSystemNode\* fileNode = fs.getNodeByPath(filename);  if (!fileNode) {  std::cout << "File does not exist. Creating new file: " << filename << std::endl;  fs.create(filename, false, fs.currentUser->username, fs.currentUser->group);  fileNode = fs.getNodeByPath(filename);  }  if (fileNode->isDirectory) {  std::cerr << "Error: Path is a directory\n";  return;  }  fs.loadFromFile("filesystem\_state.txt");  if (fileNode->isBeingEdited) {  std::cerr << "Error: File is currently being edited\n";  return;  }  fileNode->isBeingEdited = true;  fs.saveToFile("filesystem\_state.txt");  std::cout << "Entering insert mode for file: " << filename << std::endl;  std::cout << fileNode->content;  // ... 编辑文件内容 ...  fileNode->isBeingEdited = false;  }  void readFile(const std::string& path) {  FileSystemNode\* node = getNodeByPath(path);  if (!node || node->isDirectory) {  std::cerr << "Error: Invalid file path\n";  return;  }  loadFromFile("filesystem\_state.txt");  if (node->isBeingEdited) {  std::cerr << "Error: File is currently being edited\n";  return;  }  std::cout << node->content;  } |

通过这些机制，代码实现了文件的读读允许、读写互斥和写写互斥保护。

# 5. 运行结果

## 5.1 运行环境

“生产者-消费者”部分在ubuntu22.04完成，多用户系统在win11系统完成。

显卡型号NVIDIA GeForce RTX 3060 Laptop ，CPU型号12th Gen Intel(R) Core(TM) i7-12700H

## 5.2 服务模式

“生产者-消费者”问题采用仿Linux操作系统的终端控制服务模式，多用户文件系统采用类Windows操作系统的页面交互服务模式。

多用户文件系统主要基于客户端-服务器架构，通过命令行界面提供服务。用户通过CLI与文件系统交互，执行各种文件操作，如创建、删除、读取、写入文件和目录，以及管理用户权限和文件保护等。

## 5.3 运行结果

### 5.3.1 “生产者-消费者”问题

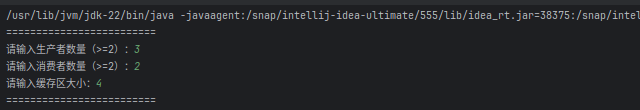
为更好描述“生产者-消费者”问题，使用现实生活中商品生产与消费的场景模拟该进程问题。用户可以输入生产者和消费者的数量。不同商品生产者有对应的生产者进程的线程编号，同理不同消费者也有对应的消费者进程下的线程编号。同时，对商品编号处理，以更好地区分缓冲区中的生产消费操作。

不同生产者生产商品，并给其编号放入缓冲区中；生产一定数量后生产商停止生产；并行的，消费者购买仓库中的商品，若消费者进程结束停止购买。

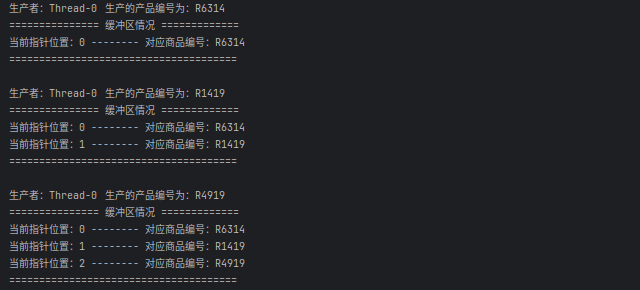
#### 5.3.1.1 初始化

用户可以对生产者和消费者的数量、缓冲区的大小进行初始化。

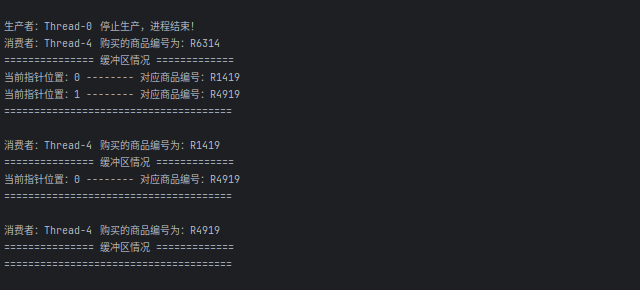
根据用户初始化定义，生产者对应线程Thread-0 ~ Thread-2，消费者对应线程Thread-3 ~ Thread-4。



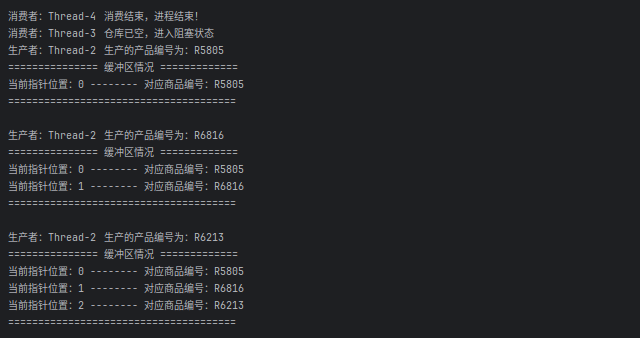
#### 5.3.1.2 运行结果及分析

时间片0-3时，生产者Thread-0到达，生产者线程创建产品并将其放入缓冲区

时间片4-6时，生产者Thread-0停止生产，进程结束。消费者线程Thread-4从缓冲区中取出产品并消费。

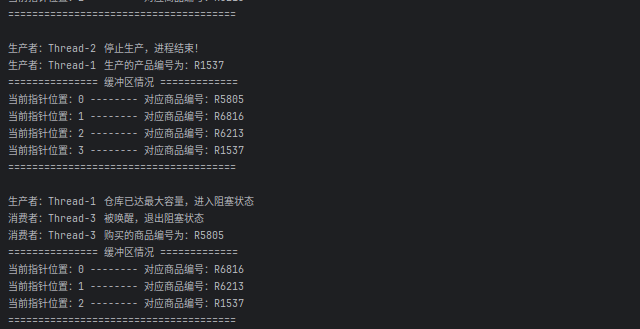


时间片7时，缓冲区为空，消费者Thread-3进入阻塞状态，等待后续唤醒。时间片7-9时，生产者Thread-2到达，生产产品R5805、R6816、R6213，依次从尾部存入缓冲区。



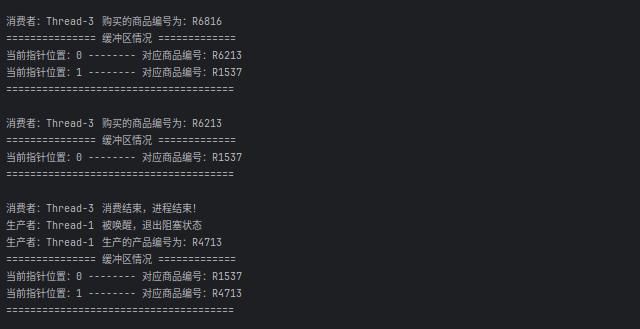
时间片10时，生产者Thread-2停止生产，进程结束；生产者Thread-1到达，生产产品R1537，存入缓冲区尾部。

时间片11时，生产者Thread-1仍要继续生产，但此时缓冲区已满，无法存入新的数据，故Thread-1进入阻塞状态，等待后续唤醒。此时唤醒消费者Thread-3，使其退出阻塞状态，购买缓冲区头部商品R5805。

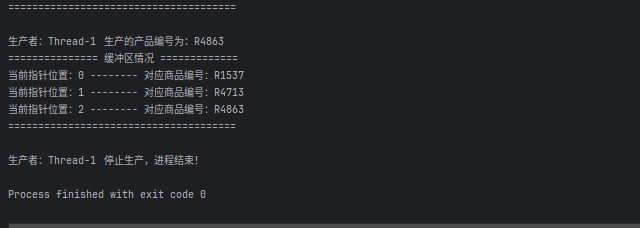


时间片12时，消费者Thread-3继续消费，购买缓冲区头部商品R6816。

时间片14时，消费者进程结束，结束消费。生产者Thread-1被唤醒，生产商品。



时间片15时，生产者继续生产。时间片16时，生产者Thread-1停止生产，进程结束。此时，所有进程结束，程序结束。



程序在时间片16时结束的原因是所有生产者和消费者线程都完成了各自的任务。生产者 Thread-1 在生产完最后一个产品 R4863 后停止生产，标志着它的任务完成。由于所有线程都完成了任务，主线程检测到这一点后，正常结束了程序，返回代码为0。

### 5.3.2 多用户文件系统

#### 5.3.2.1 初始界面

本系统为类linux终端的命令行界面。

#### 5.3.2.2 多用户系统

本系统为多用户系统，可定义文件或目录的访问权限；

（1）多用户登录和权限定义

|  |
| --- |
| 使用my-login指令，实现用户登录    使用my-mkdir指令，创建文件；  使用my-chmod指令，定义文件访问权限    使用my-getperm指令，查看文件权限    使用exit指令，登出系统 |

（2）多用户权限保护

|  |
| --- |
| User1允许访问该文件夹“only-for-user1”    Root用户无法访问文件夹“only-for-user1” |

#### 5.3.2.3 文件的存储和索引

（1）文件存储和文件索引

|  |
| --- |
| 使用my-touch指令，创建文件    退出系统后重进，文件依然存在，能够模拟文件的存储和索引    使用my-ls指令，能够展示多层文件信息 |

（2）树状文件结构

|  |
| --- |
| /  ├── 1.txt (user1 users)  ├── dir (root root)  │ ├── for-all-users (user1 users)  │ ├── only-for-user1 (user1 users)  │ └── subdir (root root)  ├── root (root root)  └── user1 (user1 users) |

#### 5.3.2.4 文件-目录的基本命令

|  |
| --- |
| 使用my-cd命令，实现定位到指定目录    使用my-mkdir指令，能够创建指定文件夹    使用my-touch指令，能够添加文件    使用my-vim指令，能够实现文件内容修改    使用my-rm指令，能实现文件删除 |

#### 5.3.2.5文件保护

（1）读读允许

|  |
| --- |
| 同时登录user1和root用户，同时读取“/dir/only-for-user1-write.txt”能够查看相应文件内部信息   * 写写互斥 |

（2）读写互斥

|  |
| --- |
| 同时登录user1和root用户，user1对文件进行写操作时，root无法读操作， |

（3）写写互斥

|  |
| --- |
| 同时登录user1和root用户，user1对文件进行写操作时，root无法写操作 |

# 6. 调试和改进

在本课程设计的实施过程中，调试和改进是确保项目质量和性能的关键步骤。以下是对Linux内核代码分析、新增系统调用以及多用户文件系统的调试和改进措施的详细描述。

## 6.1 调试过程

调试工作主要围绕代码审查、单元测试和集成测试展开。代码审查侧重于检查新增系统调用和文件系统代码的逻辑正确性和代码风格。单元测试针对每个模块和函数进行，确保它们能够独立正确运行。集成测试则关注模块间的交互和整体系统的行为。

## 6.2 改进措施

针对发现的问题，我们采取了以下改进措施：

* **代码优化**：对冗余代码进行精简，提高代码的执行效率。
* **异常处理**：增强系统调用和文件操作的异常处理能力，确保系统在遇到错误时能够给出清晰的反馈并保持稳定。
* **性能调优**：通过分析系统调用的性能瓶颈，优化算法和数据结构，以提高系统性能。
* **用户反馈**：根据用户测试反馈，调整用户界面和用户体验，使其更加友好。

# 7. 心得和结论

## 7.1 结论和体会

通过本次课程设计，我们深入理解了Linux内核的工作原理和文件系统的构建方法。设计和实现新增系统调用以及多用户文件系统的过程，不仅加深了我们对操作系统核心机制的认识，而且锻炼了我们的编程能力和问题解决能力。实践中，我们学会了如何平衡系统性能和安全性，以及如何在多线程环境下确保数据的一致性和完整性。

## 7.2 进一步改进方向

尽管本项目已取得了一定的成果，但仍存在改进空间：

* **性能优化**：进一步分析系统性能瓶颈，探索更高效的算法和数据结构。
* **功能扩展**：增加更多系统调用和文件操作功能，以支持更复杂的文件系统操作。
* **用户界面改进**：设计更直观的用户界面，提升用户体验。
* **安全性增强**：加强对系统调用和文件访问的安全性检查，防止潜在的安全威胁。

## 7.3 分析设计方案对系统安全的影响

本课程设计的设计方案对系统安全具有以下影响：

* **增强的权限管理**：通过实现细粒度的权限控制，有效防止未授权访问，增强了系统的安全性。
* **数据一致性保障**：通过文件锁定机制和读写互斥控制，确保了数据的一致性，防止了数据损坏。
* **系统稳定性提升**：通过异常处理和错误反馈机制，提高了系统的稳定性，减少了系统崩溃的风险。
* **潜在风险识别**：在设计和实现过程中，我们识别了多线程环境下的潜在风险，并采取了相应的同步和互斥措施，降低了系统出错的可能性。

综上所述，本课程设计的实施方案在提升系统性能的同时，也加强了系统的安全性和稳定性。未来的工作将进一步探索如何平衡性能和安全，以构建更加健壮和可靠的操作系统环境。

**考核成绩评定表**

|  |  |
| --- | --- |
| 指导教师考核成绩 |  |
| 答辩成绩 |  |
| 总成绩 |  |

签字：

**年 月 日**