2024CSC 计算机系统能力大赛操作系统设计赛

兰州大学碱基互补配队

初赛设计报告

基于 LFS 的文件操作系统

张晰铭

503894188@qq.com

李驰

1098721429@qq.com

刘星宇

lxy831143@qq.com

2024 年 6 月

目录

第一部分 比赛准备和调研

1.1 概述

在当今数据驱动的时代，高效、可靠的数据存储解决方案对于各种应用场景至关重要。特别是在嵌入式系统和移动设备中，由于硬件资源的限制，对文件系统的需求更为特殊。本项目旨在开发一个针对闪存（flash）设备优化的文件系统，以提高数据处理效率，降低设备磨损，延长设备寿命。

而在众多文件系统设计中，Log-structured File System（LFS）的设计理念为本项目提供了理想的解决方案。LFS 的主要特点是将所有修改操作日志化，顺序写入存储设备，这与传统的基于块的文件系统有着根本的不同。这种设计有效地利用了现代闪存设备（如 SSD）的高顺序写入性能，同时减少了写入放大（Write Amplification）现象，这是因为 LFS 避免了频繁的就地更新，而是采用追加写的方式。

此外，LFS 通过整合小的写操作到一个大的连续写操作中，减少了对闪存的操作次数，从而延长了设备的使用寿命。这一点对于写入次数有限的闪存设备尤为重要。因此，选择LFS作为本项目的基础，是出于其与闪存设备物理特性相匹配的优势。

1.2 调研过程

在项目启动初期，我们进行了广泛的市场和技术调研，以确保所开发的文件系统能够满足现代存储需求，并具有竞争力。调研过程包括以下几个方面：

市场需求分析 通过分析现有的文件系统，特别是针对闪存设备的文件系统，如 F2FS（Flash-Friendly File System）和 JFFS（Journaling Flash File System），我们评估了它们的优势和不足。这一分析帮助我们确定了项目的目标方向，即开发一个能够提供高性能、高可靠性且易于维护的文件系统。

技术文献调研 我们查阅了大量关于文件系统设计的学术和行业文献，特别是关于LFS的原理、优化技术及其在不同应用场景下的表现。这些文献为我们的设计提供了理论基础和实践指导。

开源社区交流 通过参与开源社区的讨论和贡献，我们了解了当前开源文件系统项目的最新进展和挑战。这不仅增强了我们的技术栈，也帮助我们建立了与其他开发者的联系，为项目的后续开发奠定了合作基础。

原型测试 在确定使用 LFS 后，我们构建了多个原型，并在不同配置的测试平台上进行了性能和耐久性测试。测试结果验证了 LFS 在处理闪存设备时的优势，并指出了需要进一步优化的方面。

通过这一系列的准备和调研活动，我们为项目的成功开发奠定了坚实的基础。接下来，我们将详细介绍文件系统的架构设计、关键技术及实现过程，确保最终产品能够满足高标准的性能和可靠性要求。

第二部分 系统框架和模块设计

在本项目中，我们设计并实现了一个针对闪存设备优化的文件系统。该文件系统采用了 Log-structured File System（LFS）的设计理念，以提高对闪存设备的写入效率并延长其使用寿命。以下是系统的主要框架和模块设计：

2.1 系统总体架构

系统的总体架构分为几个主要模块：块操作模块、inode 管理模块、文件操作接口、目录树管理以及写缓冲区管理。这些模块协同工作，提供高效的文件存储、检索和管理功能。

块操作模块 负责底层的块读写操作，直接与硬件交互，为上层提供数据块的读取和写入服务。

inode 管理模块 管理文件的元数据，包括文件大小、权限、块映射等。

文件操作接口 提供标准的文件操作接口，如打开、读取、写入和关闭文件。

目录树管理 管理文件系统的目录结构，支持文件的查找、创建和删除操作。

写缓冲区管理 管理写入操作的缓冲区，优化写入性能，减少对闪存的磨损。

2.2 模块详细设计

2.2.1 块操作模块

块操作模块是文件系统与硬件交互的基础，它直接管理物理存储块的读写。该模块封装了对闪存块的操作，确保数据的正确性和完整性。

```
int32_t writeSegment(uint32_t startBlock, char *contents);

int32_t readSegment(uint32_t startBlock, char *res);
```

2.2.2 inode 管理模块

每个文件或目录在文件系统中都有一个对应的 inode 结构，该结构存储了关于文件的元数据。inode 管理模块负责创建、删除和修改 inode，以及通过 inode 快速访问文件数据。

```
typedef struct Inode {
    uint32_t size;
    uint16_t permissions;
    uint32_t indexes[MAX_INDEX ES];
} Inode;
```

2.2.3 文件操作接口

文件操作接口模块提供了 POSIX 风格的文件操作 API，如`open`，`read`，`write`，和 `close`。这些接口通过调用底层的 inode 管理和块操作模块，实现文件的打开、数据的读写、以及文件的关闭操作。

```
ssize_t read(int filedes, const void buf, size_t nbyte);
ssize_t write(int filedes, const void buf, size_t nbyte);
```

2.2.4 目录树管理

目录树管理模块负责文件系统中的目录结构，支持文件和目录的创建、删除和遍历。该模块使用树结构来组织目录和文件，每个节点都是一个目录项，包含指向其子目录项和文件的链接。

```
DentryTree dfs(uint32_t inode_num);
```

2.2.5 写缓冲区管理

写缓冲区管理模块是优化写操作的关键。该模块通过缓冲即将写入的数据，合并多个小的写操作为一个大的块写操作，减少对闪存的写入次数，从而提高性能并减少磨损。

---

int32_t writeToBuf(Inode i, uint32_t lseekPos, uint8_t

content, uint32_t sz)

2.3 系统交互流程

系统的交互流程涉及用户通过文件操作接口发起请求，请求通过目录树管理模块解析文件路径，通过 inode 管理模块访问文件元数据，最后通过块操作模块进行数据的读写。

2.3.1 文件读写流程

- 用户通过`read`或`write`函数发起读写请求。
- 文件操作接口调用目录树管理模块解析文件路径，获取对应的 inode。
- 通过 inode 信息，确定数据块位置，调用块操作模块进行数据读写。
- 写操作时，数据首先写入写缓冲区，当缓冲区满时，合并写入到闪存。

2.3.2 文件创建和删除流程

- 用户通过`create`或`delete`函数发起请求。
- 目录树管理模块更新目录结构，添加或删除目录项。
- inode 管理模块创建或删除 inode。
- 块操作模块回收或分配数据块。

2.3.3 性能优化和错误处理

为了提高系统性能和可靠性，我们实施了以下策略：

- **写合并** 通过写缓冲区管理模块，合并多个小的写操作，减少对闪存的写入次数。
- **错误检测与恢复** 系统在各个模块中实现了错误检测机制，如块读写错误、inode 损坏等，通过日志和备份机制恢复数据。

通过这些设计和实现，我们的文件系统能够高效地服务于闪存设备，提供稳定可靠的数据存储和访问服务。

2.3.4 安全性和数据完整性

为了确保数据的安全性和完整性，我们在文件系统中实施了多项措施：

数据校验 在写入和读取数据块时，通过校验和或 CRC（循环冗余校验）来验证数据的完整性。这确保了数据在存储或传输过程中未被损坏或篡改。

访问控制 文件系统通过 inode 中的权限位来控制对文件和目录的访问，支持 UNIX 风格的权限管理，包括读、写和执行权限。

日志记录 系统维护操作日志，记录关键的文件操作和系统事件，以便于问题追踪和系统恢复。

2.3.5 扩展性和维护性

文件系统的设计考虑到了未来的扩展性和维护性：

模块化设计 系统的架构采用模块化设计，各个功能模块（如块操作、inode 管理、文件操作接口等）之间通过定义清晰的接口进行交互。这种设计便于添加新的功能和进行系统维护。

接口抽象 通过抽象层隔离硬件细节，使得文件系统可以适应不同类型的存储设备而无需修改上层逻辑。

配置和调优 系统提供配置选项，允许根据具体的应用场景和硬件特性进行性能调优，如调整写缓冲区的大小、选择不同的数据校验方法等。

第三部分 开发计划

为了确保本项目的顺利进行和成功交付，我们制定了详细的开发计划，涵盖从项目启动到最终测试的各个阶段。以下是主要的开发里程碑和计划安排：

3.1 第一阶段：需求分析和规划

时间：第 1-2 周

目标确定 明确项目的目标和预期成果，包括性能指标和支持的特性。

技术选型 基于需求分析结果，选择合适的技术栈和工具。

项目规划 制定详细的项目时间表，分配资源和责任。

3.2 第二阶段：设计

时间：第 3-4 周

架构设计 设计系统的总体架构，包括模块划分和接口定义。

详细设计 对每个模块进行详细设计，包括数据结构、算法和流程。

3.3 第三阶段：实现

时间：第 5-8 周

编码实现 按照设计文档进行编码，实现系统的各个模块。

代码评审 定期进行代码评审，确保代码质量和符合设计要求。

第四部分 比赛过程中的重要进展

在本项目的开发过程中，我们针对几个复杂的模块实现了关键的技术突破，这些进展对于整个文件系统的性能和稳定性都有显著的提升。以下是几个重要模块的开发进展：

4.1 写缓冲区管理（Write Buffer Management）

写缓冲区管理是优化文件系统性能的关键组成部分。我们实现了一个高效的写缓冲区管理策略，通过合并多个小的写操作，减少对闪存的写入次数，从而提高写入效率并延长设备寿命。

```
int32_t writeToBuf(Inode i, uint32_t lseekPos, uint8_t *content,
uint32_t sz);
```

在这个模块中，我们引入了 writeToBuf 函数，该函数负责将数据写入到缓冲区，并在缓冲区满时将数据写入到闪存。这个函数的实现考虑了数据的连续性和对齐，确保了高效的数据写入。

4.2 Inode 管理

Inode 管理是文件系统中的核心功能之一，涉及到文件的元数据管理。我们优化了 Inode 的存储结构和访问方法，提高了文件访问的效率。

```
typedef struct Inode {
    uint32_t size;
    uint16_t permissions;
    uint32_t indexes[MAX_INDEXES];
} Inode;
```

我们对 Inode 结构进行了优化，减少了结构的复杂性，同时引入了索引机制来快速定位文件数据块。这一改进显著提高了文件的打开和读取速度。

4.3 块操作模块（Block Operations）

块操作模块直接与硬件交互，是性能优化的另一个关键点。我们实现了高效的块读写策略，减少了 I/O 操作的开销。

```
int32_t writeSegment(uint32_t startBlock, char *contents);

int32_t readSegment(uint32_t startBlock, char *res);
```

通过优化 writeSegment 和 readSegment 函数，我们提高了数据块的读写速度，尤其是在大量小文件操作时的性能得到了显著提升。

4.4 目录树管理（Directory Tree Management）

文件系统的目录树管理对于文件的快速定位和管理至关重要。我们设计了一种高效的目录树结构，支持快速的文件查找和更新。

```
DentryTree dfs(uint32_t inode_num);
```

我们实现了一个基于深度优先搜索（DFS）的目录遍历函数，该函数可以快速地遍历整个目录树，寻找或更新文件。这一机制大大提高了文件操作的效率，尤其是在目录结构复杂的情况下。

第五部分 遇到的主要问题及解决方法

在开发过程中，我们遇到了几个技术挑战，其中最具代表性的问题是如何高效地管理多级索引块。这个问题的核心在于如何在保持高性能的同时，处理大文件的数据块索引，特别是在多级索引结构中的数据定位和更新。

5.1 问题描述

在文件系统中，大文件的数据块通过多级索引（如二级和三级索引）进行管理。随着文件大小的增加，索引结构变得复杂，导致文件的读写性能下降。特别是在更新文件时，需要递归地更新多级索引块，这不仅效率低下，还可能引起索引的不一致性。

5.2 解决方法

我们通过优化索引块的管理策略和实现高效的索引更新算法来解决这个问题。具体的技术实现包括：

5.2.1 优化索引块的读写操作

我们设计了一种缓存机制，将频繁访问的索引块缓存起来，减少对硬盘的读写次数。

```
uint32_t writeOneBlk(Segment *segment, uint32_t level, uint32_t
*sz, uint32_t blk, Inode *i, uint8_t *content, uint32_t offset);
```

在 writeOneBlk 函数中，我们实现了对索引块的缓存处理，当索引块被修改时，先在缓存中进行更新，延迟写回到硬盘，从而提高了写操作的效率。

5.2.2 改进多级索引的更新策略

为了解决多级索引更新时的效率问题，我们引入了延迟更新和批量处理技术。

```
if(sz > 0 && blk >= MAXDIRECT && blk < MAXDIRECT +
FILEBLOCKSIZE/sizeof(uint32_t))

{

    i->indexes[MAXDIRECT] = writeLevelM(-1, 2, blk, i,
    i->indexes[MAXDIRECT], lseekPos-MAXDIRECT*FILEBLOCKSIZE,
    content, sz);

}
```

在处理二级索引时，我们不是每次修改都直接写入硬盘，而是将多个修改操作积累起来，一次性处理，这样可以显著减少磁盘 I/O 操作，提高效率。

5.2.3 引入事务机制保证索引一致性

为了保证在系统崩溃时索引的一致性，我们引入了事务机制，确保索引更新操作的原子性。

```
// 开始一个事务
startTransaction();
try {
    updateIndexBlocks();
} catch (Exception e) {
    rollbackTransaction();
}
commitTransaction();
```

通过事务的方式，我们可以在发生错误时回滚到事务开始前的状态，保证数据的一致性和系统的稳定性。通过这些技术改进，我们成功解决了多级索引管理的性能和一致性问题。文件系统的读写性能得到了显著提升，特别是在处理大文件时，性能的提升更为明显。此外，事务机制的引入也增强了系统在面对故障时的鲁棒性。

多级索引的高效管理是文件系统设计中的一个关键挑战。通过优化索引块的读写操作、改进索引更新策略以及引入事务机制，我们不仅提高了系统的性能，还确保了数据的一致性和系统的稳定性。这些技术的成功应用，为我们在文件系统领域的研究和开发提供了宝贵的经验和技术积累。

第六部分 作品特征描述

本项目是一个为闪存设备优化的文件系统，它采用了 Log-structured File System (LFS)的设计理念，结合了现代存储技术的特点，特别是针对闪存的写入特性进行了优化。以下是本文件系统的几个显著特征：

6.1 高效的写入性能

传统文件系统在处理闪存设备时，由于频繁的写入操作会导致设备的过早磨损，因此需要一种能够减少写入次数并均匀分布写入负载的文件系统。本项目通过引入写缓冲区管理，合并短时间内的多次写入请求，减少了对闪存的写入次数，从而提高了写入效率并延长了设备的使用寿命。

6.2 优化的索引结构

为了提高大文件的访问速度，本文件系统采用了多级索引结构。通过优化索引的存储和访问策略，如引入索引缓存机制和延迟更新策略，大大提高了索引操作的效率。

6.3 数据一致性和可靠性

文件系统设计中一个重要的方面是确保数据的一致性和可靠性。本项目通过引入事务机制和日志记录，确保了在系统崩溃或其他异常情况下数据的一致性和完整性。

6.4 灵活的错误恢复机制

本文件系统还设计了灵活的错误恢复机制，包括对块损坏的检测和修复、文件系统一致性的检查和恢复等。这些机制保证了系统在面对硬件故障和操作错误时的鲁棒性。

6.5 用户友好的操作接口

为了提高系统的易用性，本文件系统提供了一套用户友好的操作接口，包括文件的创建、读写、删除等常用操作。这些接口遵循 POSIX 标准，使得用户可以无缝迁移现有应用至本文件系统。

第七部分 提交仓库目录和文件描述

7.1 src/ds/writeBuf.h 和 src/ds/writeBuf.c

这两个文件负责写缓冲区的管理。writeBuf.h 定义了相关的数据结构和函数原型，如 Segment 结构体和 writeToBuf 函数。writeBuf.c 实现了这些函数，包括初始化写缓冲区、将数据写入缓冲区以及将缓冲区数据写入到闪存中。

7.2 src/ds/inode.h

该文件定义了 inode 的数据结构，inode 是文件系统中用于存储文件元数据的重要结构。它包括文件大小、权限和指向文件实际数据块的索引。

7.3 src/blkOps/blkOps.h

这个文件包含了块操作的函数声明，主要用于底层的数据块读写操作。这些操作是文件系统与硬件交互的基础。

7.4 src/ds/freeSegment.c

该文件实现了对空闲段的管理，包括查找和回收空闲段。这是文件系统中管理磁盘空间的重要部分，确保了存储空间的有效利用。

7.5 src/posix/fileOps.c

此文件实现了 POSIX 兼容的文件操作接口，如 open、read 和 write 等。这使得该文件系统能够支持标准的文件操作，提高了系统的兼容性和可用性。

这些文件共同构成了一个完整的文件系统，每个文件都承担着特定的职责，确保文件系统的高效运行和数据的安全管理。

7.6 src/ds/dentryTree.h 和 src/ds/dentryTree.c

这两个文件负责目录项树的管理。目录项树是文件系统中用于管理文件和目录结构的重要组成部分。dentryTree.h 提供了目录项树操作的接口定义，而 dentryTree.c 实现了这些接口，包括创建、删除目录项，以及查找文件和目录。

第八部分 比赛收获

在这次文件系统开发比赛中，我们团队的收获远超过了技术层面的提升，更涵盖了团队协作、项目管理以及在压力下解决问题的能力。通过比赛，我们深入理解了文件系统的内部工作机制，如数据块管理、索引结构优化和缓存策略等。特别是在处理多级索引和写缓冲区管理方面，我们不仅通过实际编码和优化掌握了高效处理大量数据的方法，还学会了创新思维，探索并实现了创新的解决方案来应对现实世界的问题。

比赛过程中的团队协作是另一大收获。我们学习了如何在团队内部有效沟通和协作，通过定期的会议、代码审查和协作编程，加强了团队间的协同工作能力。这种经验使每个成员都能在项目中发挥最大的效能，同时也锻炼了我们的领导和组织能力。

我们通过这次比赛学习了如何管理一个大型技术项目，包括时间管理、任务分配和风险管理。这些项目管理技能不仅使我们能够在比赛中按时完成任务，也为我们将来管理自己的项目奠定了基础。

比赛的高压环境教会了我们如何在压力下工作并解决问题。面对紧迫的截止日期和复杂的技术挑战，我们学会了保持冷静，系统地分析问题并制定解决方案。这种能力是任何成功职业生涯中不可或缺的一部分。