基于数据库的文件系统设计与实现

摘要

随着计算机科学的发展，文件系统作为操作系统的核心组成部分，一直在不断地发展和演变。现代文件系统通过提供高效的数据组织和存储、可靠的容错能力和高级功能（如数据压缩、快照和镜像等）来满足不同的需求。但是，这些功能的实现通常需要复杂的算法和数据结构，并且有时需要对硬件进行特殊的优化。此外，文件系统的开发和维护也需要大量的时间和精力，现代文件系统愈加复杂，导致难以调试和纠错。数据库是一种高效、可靠且具有事务特性的数据管理系统，现实世界对数据的需求不变变化，也让数据库的发展不断更新迭代，总体而言，数据库的发展速度要远远快于文件系统。在文件系统中许多新技术都借鉴自数据库系统。传统的文件系统实现方式在近几十年的发展过程中并没有发生本质性的变化，并且在从数据库系统中引入一些特性时还需要不断地进行修改与适配，这导致了许多不必要的成本。为了解决这些问题，本文考虑利用数据库来作为数据引擎实现文件系统。基于数据库的文件系统的可以带来很多好处，例如，数据库提供了高效的查找算法和缓存机制，可以加快文件访问速度；同时，数据库也提供了高级特性，比如备份，事务等。此外，使用数据库还可以简化文件系统的实现，使得开发者不再需要关注在磁盘上的具体布局，不需要重复编写数据库中已经具备的功能，提高代码的可维护性和可扩展性。

本文对基于数据库的文件系统进行初步尝试，设计并实现了一个操作系统无关的DBFS(database file system)。本文选择了一个较为简单的key-value类型的数据库`jammdb`作为底层的数据库引擎，并对数据库进行改造去除对操作系统的依赖，在此基础上，根据数据库的结构设计了类似于ext系列的超级块、Inode等数据结构，并参考linux的VFS提供了一层与系统无关的文件操作函数。在linux系统上，本文将DBFS介入`fuse`，从而可以在linux系统上完成文件系统的功能验证和性能测试。同时，为了验证使用rust编程语言实现的dbfs可以迁移到linux内核中，本文使用rust实现了一个类linux的简易操作系统内核Alien OS，并将dbfs无缝地迁移到此内核中。实验证明，本文实现的文件系统性能可以达到linux上传统文件系统性能的一半以上。

本文的研究成果可以为文件系统的开发和研究提供一个新的思路和方法。通过利用数据库的优势，我们可以构建出高效、可靠且具有高级功能的文件系统，这可以减小文件系统的开发难度，并提高文件系统的可维护性和可扩展性，在一定程度上甚至可以提高其性能。

关键词： 文件系统、数据库、操作系统、DBFS、fuse、Alien OS

第一章 绪论

1.1 研究背景

计算机的文件系统是一种存储和组织计算机数据的方法，它使得对数据的访问和查找变得容易，文件系统使用文件和树形目录的抽象逻辑概念代替了硬盘和光盘等物理设备使用数据块的概念，用户使用文件系统来保存数据而不必关心数据实际保存在硬盘（或者光盘）的地址为多少的数据块上，只需要记住这个文件的所属目录和文件名。在写入新数据之前，用户不必关心硬盘上的哪个块地址没有被使用，硬盘上的存储空间管理（分配和释放）功能由文件系统自动完成，用户只需要记住数据被写入到了哪个文件中。随着时间的推移，存储需求不断变化，数据量不断增加， 文件系统必须是可靠的、持久的、安全的、高效的、容错的和可扩展的，[1] 为了实现这些特性并跟上不断变化的计算需求和存储需求，不同的技术和文件系统随着时间的推移而被引入到操作系统中，但是绝大多数的文件系统实现仍然采用的是树状层次结构，提供给用户的接口仍然是`open`,`read/write`等。

在文件系统快速发展的同时，数据库系统同样也在随着需求的不断变化而快速迭代。数据库是以电子方式存储的系统数据集合，它可以包含任何类型的数据，包括文字、数字、图像、视频和文件。这些数据按一定的数据模型组织、描述和存储，具有较小冗余度、较高数据独立性和易扩展性，并可为各种用户共享。通常，数据库的实现以文件系统为基础，所有的数据库操作，如增删改查、备份等最后都会转换为对磁盘上文件的操作，但有时数据库并不想受到文件系统实现带来的影响，一个广泛的现象是许多数据库系统的实现都会包

含一个缓存模块，以此来提高其性能和可靠性。除了一直占据主要地位的大型关系型数据库系统，如Mysql、SQLserver等，近年来也出现了一些嵌入式数据库和许多nosql数据库，这些数据库与传统的数据库相比，其对系统资源的占用更小，如sqlite，其与应用程序运行在同一个地址空间中，而不是像Mysql那样存在服务器端和客户端的区别，同时，新兴的nosql数据库虽然功能不如传统的大型关系型数据库完善，但根据需求的变化，这些nosql数据库在一些专有领域也有着不错的表现。

从数据库和文件系统的发展来看，两者是在互相改进，共同发展，从两者的功能来看，他们都是存储数据的一种管理方式。既然两者的功能如此相像，那一个值得思考的问题就是为什么大多数的数据库系统实现以文件系统为基础，而不能反其道而行之？文件系统的实现者是否可以重用数据库的基础设施以获得数据库提供的更多特性？更近一步地，文件系统的实现是否可以完全在一个数据库系统提供的接口上进行.

1.2 国内外研究现状

早期，数据库系统的基础是底层的文件系统，许多管理数据的特性都只是在数据库系统上得到体现，而近年来，文件系统的设计开发中越来越借鉴了数据库相关技术的思想，二者不断融合发展。日志文件系统就是在借鉴数据库系统的日志和事务特性之后发展出的一种文件系统。在对文件系统进行修改时，需要进行很多操作，比如在扩展一个文件的大小时，需要对文件的元信息和磁盘元信息都需要修改，这些操作可能中途被打断，但是，理论上这些操作不是不可中断的。如果操作被打断，就可能造成文件系统出现不一致的状态，虽然可以使用一些工具对这些情况进行修复，如fsck，但这些工具在修复时需要扫描整个磁盘，繁琐且速度较慢。日志文件系统在修改磁盘数据时利用日志信息记录修改过程，如果文件系统发生崩溃，重启后只需要根据日志信息就可以恢复数据从而保证数据的一致性和完整性。发展较早的日志文件系统JFS[]、XFS[]如今在某些系统上仍然广泛使用。而较新的Btrfs[]、Ext4[]、NTFS[]等具有日志功能的文件系统在服务器系统与桌面系统上占据着主要地位。除了对数据库的日志和简单事务的借鉴之外，许多适应新需求的文件系统实现在设计中会直接包含事务ACID特性的设计，exF2FS[]是一个事务性的日志文件系统，其允许事务跨越多个文件，应用程序可以显式指定与事务关联的文件，同时其允许使用少量内存执行事务并在一个事务中封装多个更新。DurableFS【】提供了一种受限制的事务形式：文件打开和关闭之间的操作自动形成具有原子性和持久性保证的事务。

一些文件系统的实现不再局限于在文件系统的设计上引入数据库系统的特性，而是选择直接基于数据库系统进行文件系统的实现，考虑到数据库系统提供的特性的实现复杂度，这种方案在实施难度上要远远小于重新设计文件系统。IFS【】是一个基于关系型数据库POSTGRES实现的文件系统，其向用户提供事务特性以及时间旅行功能，同时其使用多张关系表来构建层级文件系统以最大限度地支持传统POSIX接口。

Oracle iFS【】是一个运行在数据库之上的文件系统，实质上，Oracle iFS 提供了文件系统和数据库之间的范式转换，同时提高了一些高级搜索和数据备份功能。国内学者【】将Oracle iFS移植到linux的VFS模块中使得用户对其使用就如原有的文件系统一样，并且将基于内容分类、关联访问、基于内容的访问、版本控制等功能扩展到VFS中。AMINO【】是一个具有ACID语义的文件系统，其使用Berkeley Database作为后备存储，将一个易于使用但功能强大的嵌套事务API导出到用户空间。 应用程序可以开始、提交和中止事务，同时其设计了一个简单的API，使协作进程能够共享事务。 使用相同的API，单个应用程序可以支持多个并发事务。该文件系统不仅为应用程序提供了ACID的额外好处，在性能方面也与ext3相当。

1.3 研究目的和意义

基于数据库的文件系统设计与实现，是一个基于数据库技术的文件系统实现方案。其可以利用数据库的高效管理和查询能力来存储和管理文件数据，能够更好地支持大规模、多类型数据的管理和查询。此外，基于数据库的文件系统还能够支持多用户、多任务的并发操作，提高系统的可靠性和性能。本研究旨在通过将文件系统和数据库系统相结合，充分发挥两者的优势，实现高效、可扩展、易管理的文件存储与管理系统。具体地，通过将文件和相关的元数据存储在数据库中，利用数据库系统的索引机制实现高效的文件检索和访问，同时设计文件在数据库中的存储方式，实现文件的高效存储和管理，在用户态，我们也提供数据库操作的直

接抽象使得用户也可以直接使用数据库的接口进行数据的存储和访问，这种设计不仅可以提高系统的性能和可扩展性，还可以提高系统的可靠性和安全性。

基于数据库的文件系统设计与实现的研究具有重要的理论和实际意义。它可以为用户数据管理和存储提供一种新的解决方案，同时还能扩展文件系统的功能。

1.4 主要工作内容

基于数据库的文件系统设计与实现的主要工作内容是将数据库的存储和管理思想结合到文件系统中，设计并实现一个具有数据库特性的文件系统，实现对文件的高效管理和查询以及存储。具体研究内容包括：

1. 数据库技术与文件系统结合：将数据库的基本概念、数据结构、数据存储和管理技术与文件系统的数据组织、存储和访问方式相结合，构建一个具有数据库特性的文件系统。
2. 文件管理与查询优化：设计文件系统，包括文件的存储、读取、修改、删除等操作，并优化文件的查询性能，通过对文件元数据的索引和优化算法等手段，实现对文件的快速查询和检索。
3. 数据一致性与事务管理：研究文件系统和数据库之间的数据一致性问题，实现事务管理功能，确保数据的完整性和一致性。
4. 安全性与权限管理：根据linux系统的文件权限检查机制，在文件系统中实现相关的检查逻辑，保证文件系统具有正确的安全性。
5. 系统实现与性能测试：基于上述研究内容，设计并实现一个完整的基于数据库的文件系统原型，并进行性能测试和优化，验证其可行性和实用性。

本项目的所有实现使用`rust`语言完成，在实现中，本文没有选择功能强大的数据库来作为文件系统实现的底层支持，而是选择了 一个简单的`key-value`数据库。同时，因为使用`rust`来完成所有实现，而将这些实现直接放到`linux`中目前还不可行，因此本文为此实现了一个简单的类linux操作系统，并将dbfs移植到此系统上。同时为了获得它的性能数据和实现的正确性，本文在linux系统上为数据库文件系统实现了fuse的接口，并使用几个常见的性能测试工具以及几个常见的文件系统进行了测试。

总的来说，本项目总共会包含以下几个项目:

1. Alien: 使用`rust`实现的简单类linux操作系统，验证将数据库文件系统移植到操作系统内核的可能性。
2. jammdb: key-value数据库，本文选择使用的数据库。
3. rvfs: rust写的vfs框架，主要参考linux中的vfs。
4. dbfs: 本文设计实现的数据库文件系统。
5. dbop: 数据库操作抽象，一种在操作系统导出数据库接口的方法

1.5 文章结构

本文的结构安排如下：

第一章，给出本课题的研究背景和意义，并概述国内外的研究现状，本文的主要工作内容以及文章的写作结构。

在第二章，介绍数据库、文件系统、操作系统内核以及数据库文件系统相关的领域知识以及这些系统如何协同工作，同时说明本文选择的编程语言。

在第三章，详细介绍本次实现的数据库文件系统使用到的数据库，以及整个数据库文件系统的架构和设计方案。

在第四章，介绍数据库文件系统的fuse接口适配，以及将数据库文件系统接入到操作系统内核的方案。

在第五章，对本文所设计的系统进行实验测试，以评估其性能和可靠性，以及与其他相关系统进行比较分析，主要的测试讨论位于用户态的fuse实现，并简要介绍接入内核的测试。

在第六章，总结全文，并指出未来进一步的研究方向。

第二章 相关理论与方法

本章介绍了本项目设计到的相关理论和方法，并对数据库文件系统的实现技术进行深入研究和总结。2.1节主要介绍数据库系统的一些概念，比如事务，索引，缓存，并发等；2.2节主要介绍操作系统和文件系统的关系，文件系统在内核中扮演的角色，并详细介绍VFS的数据结构与方法；2.3节介绍数据库文件系统的实现技术和方法，阐述将数据库作为文件系统存储引擎并将其移植到内核中的困难之处。

2.1 数据库

2.2 操作系统内核与文件系统

2.3 数据库文件系统实现技术与方法

2.4 编程语言

为什么使用rust来实现整个项目，介绍rust的一些实用特性与特征。

本文所包含的所有项目均使用rust实现。在涉及到操作系统和文件系统这样的低级别的任务时，往往多数人选择使用c/c++来进行实现，而现在有了另一个选择，Rust 是一种非常优秀的编程语言，其相比c语言来说，有许多优势：

1. 内存安全性：在编写操作系统或文件系统时这种底层代码时，一个常见的问题是内存泄漏、空指针引用等，这些问题在Rust中是不可能的或者很少发生的，因为Rust有强大的编译时借用检查和内存管理机制。
2. 零成本抽象: Rust具有零成本抽象，这意味着我们可以在Rust中使用高级语言特性，同时不会增加额外的运行时开销。比如我们可以使用Rust的trait和泛型来创建高效的抽象数据类型，这可以使项目的代码更容易阅读和维护。
3. 性能: Rust被设计为一种高性能的语言。在操作系统和文件系统等场景下，性能是一个非常关键的因素。Rust通过使用内存安全性和零成本抽象等技术，可以在不牺牲性能的前提下提高开发效率。
4. 社区支持：Rust拥有一个强大的开源社区，提供了许多有用的工具和库，这些工具和库可以帮助我们更轻松地构建操作系统和文件系统。

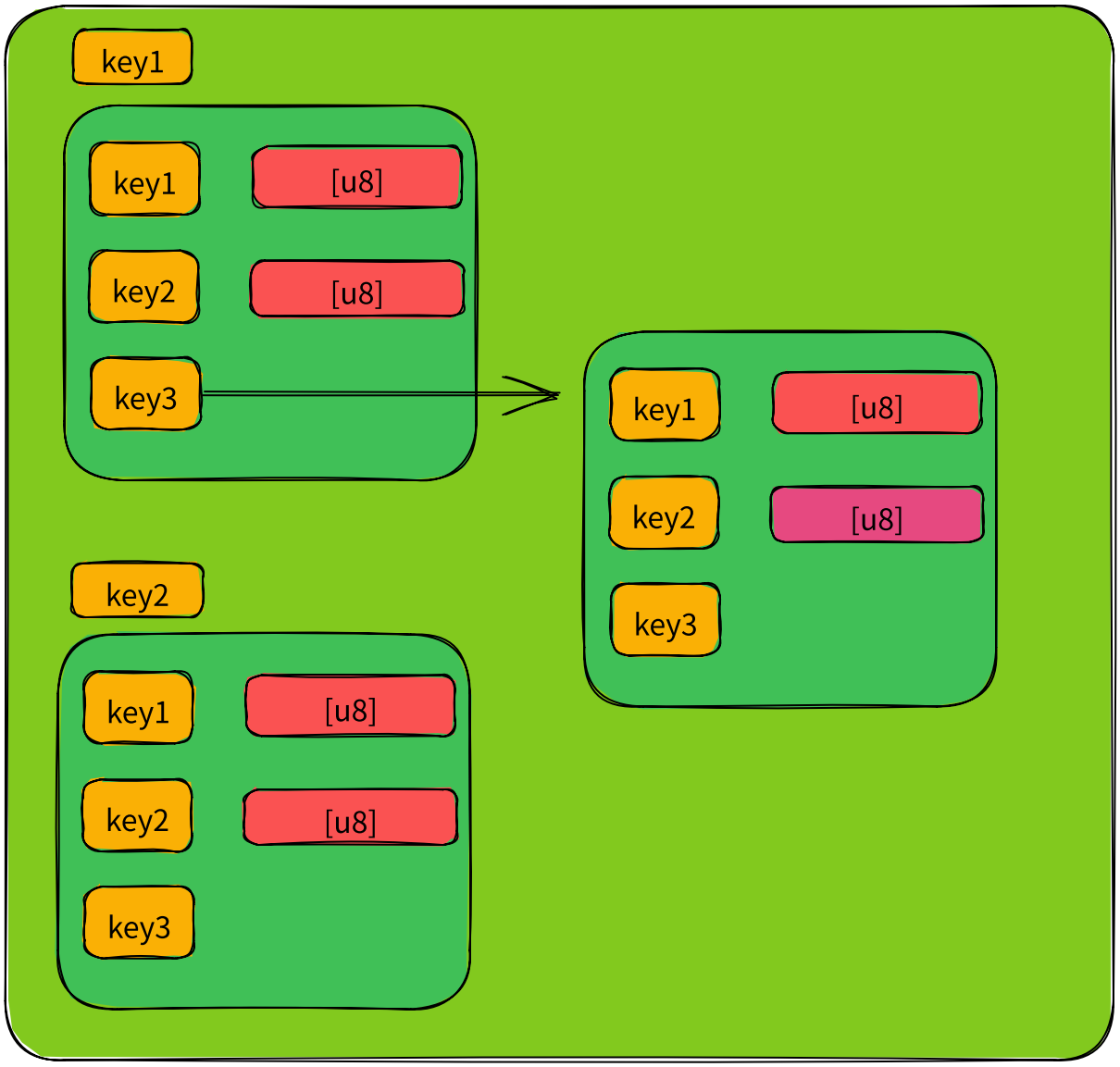
2.5 本章小结

第三章 dbfs设计与实现

4.1 数据库结构的分析

jammdb是一个嵌入式、单文件的key-value数据库，其提供ACID特性，支持多个并发读取和多个单个写入。所有的数据被组织成一棵B+树，随机和顺序读取速度很快。其对文件的操作基于内存映射。选择这个数据库作为dbfs实现的原因是简单，因为复杂的数据库难以不仅难以移植到裸机平台，要将其放到内核态需要大量的工作，但简单带来的一个坏处就是数据库的功能不够强大，并且数据库本身没有对磁盘设备作出相应的优化，对于毕设来说，我们主要进行dbfs的探索和尝试，因此不必选择复杂的数据库。为了在no_std环境下使用jammdb，本文fork了原项目仓库，删除了原项目中依赖的标准库或操作系统接口，例如一些标准库算法以及mmap系统调用接口，并将这些依赖重新定义。这样一来用户只需要提供这些接口的实现，就可以使用此数据库，而为了检查作出的修改是否影响了原实现的正确性，我们在其内部实现了一个内存模拟的文件，并实现抽象出来的接口，在通过了所有测试后，可以说明我们的修改没有引入错误。

数据库的数据结构如下所示:



数据库的内部基于桶`bucket`实现。如上面所示，该数据库的基本结构由一个个处于全局空间的`bucket`组成，`bucket`可以存储普通的key-value数据，这里key和value都是[u8]数组，同时也可以存储嵌套的`bucket`数据结构。一个`bucket`是由一棵B+树构成，b+树将大小为4k或者其它大小的页面组织起来存储数据，这里的页面与传统的磁盘块类似，只是因为数据库使用内存映射而将存储结构设置为页大小。

4.2 目录树构建与元信息管理

讨论如何利用数据库的结构来

4.3 文件数据存储

讨论如何存储文件数据

4.4 dbfs接口设计

如何构建通用的，与os无关的文件系统接口。

第四章 Fuse与Alien OS

5.1 Fuse的linux用户态dbfs实现

Fuse（Filesystem in Userspace）是一个用户空间文件系统接口，它允许非特权用户在不修改操作系统内核的情况下创建自己的文件系统。Fuse提供了一种通用的机制，用于将用户空间的代码与内核文件系统接口连接起来，从而实现自定义文件系统的开发。

Fuse文件系统工作流程如下：

1. 应用程序调用标准库中的系统调用，例如open()，read()等。
2. 应用程序的请求传递到Fuse内核模块，该模块负责将请求路由到特定的Fuse文件系统实现。
3. Fuse文件系统实现处理请求并执行相关的操作，例如读取或写入文件。
4. 文件系统操作的结果返回给Fuse内核模块。
5. 内核模块将结果返回给应用程序，仿佛这些操作是直接由内核处理的一样。

Fuse的优点在于，它允许用户空间程序使用一种通用的API来实现自定义文件系统。这使得开发人员可以更轻松地开发和测试文件系统，而无需担心对内核进行修改。此外，Fuse还提供了一些其他功能，例如访问远程文件系统，加密文件系统和网络文件系统等。Fuse文件系统的缺点在于，由于所有文件系统操作都是通过用户空间代码执行的，因此对于某些高负载应用程序，它可能会导致性能问题

讨论在dbfs通用接口的基础之上提供linux的fuse支持，可以更直观地测量文件系统的性能并进行分析

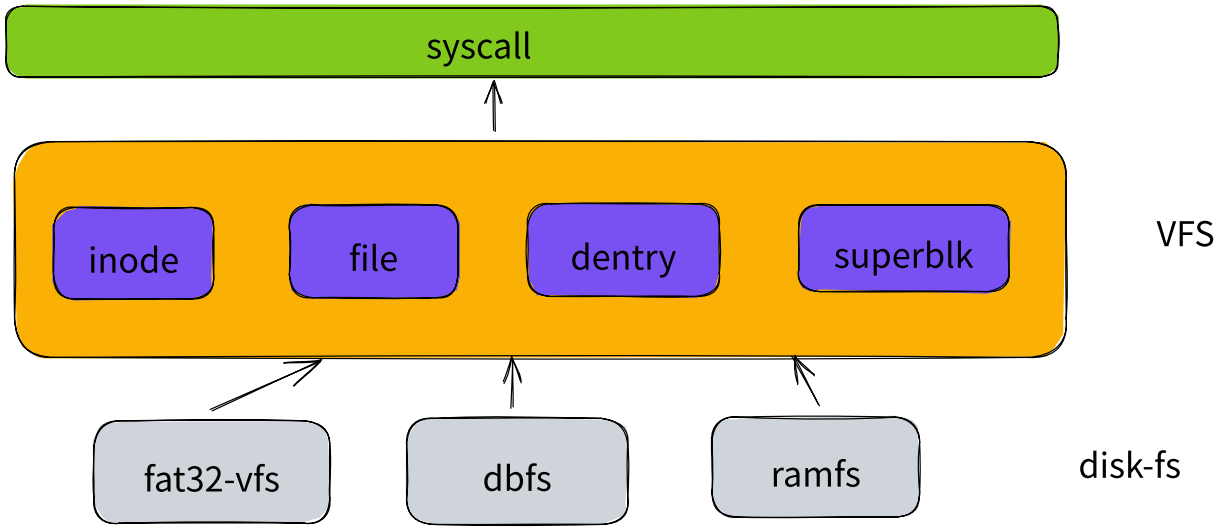
5.2 Alien OS

这是一个使用`rust`实现的简单操作系统，其基于qemu模拟器，支持`riscv64`位平台。这个简单的操作系统原本是用来探索模块化os的，在比设选择了实现文件系统后本文选择将其作为文件系统以及应用程序的运行平台。目前已经完成的功能包括内存管理，虚存管理，外部设备，中断，进程管理，文件系统，虚拟文件系统等，已经支持数十个系统调用，同时支持rust编写的应用程序和c语言程序。

讨论本文实现的一个简单os内核，内核包含了vfs与两个底层文件系统，进一步说明dbfs放入内核态的可能性。

5.2.1 VFS

rvfs是一个类似于linux的vfs的框架，其为不同的底层文件系统提供统一的抽象。



设计上，其主要包含4个数据结构,`Inode`、`File`、`Dentry`、`Superblock`以及这些数据结构相关的操作。实现此项目的原因是我们需要将文件系统在上文提到的操作系统中使用，如果没有vfs的实现，那对fat32和dbfs的操作就需要分别进行处理，并且缺乏这个类似缓存层的实现会导致用户存取和访问文件速度变慢，降低文件系统实现的性能。

vfs的实现包含了一个简单的内存文件系统，可以作为初始挂载点，将其他文件系统挂载在上面。目前vfs实现了linux中的大部分常用接口，包括诸如读写、链接、属性扩展、文件状态、重命名、截断等。

5.2.2 dbfs 在内核中的实现

第五章 性能测试与优化

5.1 测试环境和准备

机器信息:

- 8 Intel(R) Core(TM) i5-8265U CPU @ 1.60GHz
- 8GB memory
- Ubuntu 22.04.2 LTS

程序运行条件:

1. dbfs使用release模式运行, mount 参数为

```
-o auto_unmount allow_other default_permissions rw async
```

2. ext系列使用fuse2fs工具进行挂载,具体步骤为:
    1. 使用dd生成4GB大小的文件
    2. 使用mkfs工具对文件进行格式化生成对应的文件系统
    3. 使用fuse2fs工具将文件系统挂载到任意一个空目录上

   这个系列的挂载参数与dbfs的相同
3. juicefs 使用其自带的工具完成，具体步骤为:
    1. 安装jucie客户端与sqlite数据库
    2. 使用juicefs format 命令创建了一个文件系统
    3. 使用juicefs mount 命令挂载文件系统

   juicefs的挂载参为：

```
-o allow_other,default_permissions,rw,async，writeback_cache
```

在运行测试时，挂载后文件系统内的内容会被全部删除。对于功能测试，我们只详细测试了dbfs，对于其它文件系统，只作简要的说明。

所有的测试使用脚本自动进行，并用python进行绘图。

5.2 pjdfstest POSIX兼容性测试

pjdfstest是一套简化版的文件系统POSIX兼容性测试套件，它可以工作在FreeBSD, Solaris, Linux上用于测试UFS, ZFS, ext3, XFS and the NTFS-3G等文件系统。目前pjdfstest包含八千多个测试用例，基本涵盖了文件系统的接口。

pjdfstest的项目地址与编译流程位于pjdfstest

在挂载dbfs-fuse后，可以在挂载目录下运行测试程序得到dbfs的测试结果。

测试结果

test-set	interface	pass/all	error/all	error
chflags	chflags(FreeBSD)	14/14	❌	linux下不工作
chmod	chmod/stat/symlink/chown	321/327	26/327	chmod实现有误
chown	chown/chmod/stat	1280/1540	260/1540	chmod实现有误
ftruncate	truncate/stat	88/89	1/89	Access判断出错
granular	未知	7/7	❌	linux下不工作
link	link/unlink/mknod	359/359	❌	✔️
mkdir	mkdir	118/118	❌	✔️
mkfifo	mknod/link/unlink	120/120	❌	✔️
mknod	mknod	186/186	❌	✔️
open	open/chmod/unlink/symlink	328/328	❌	✔️
posix_fallocate	fallocate	21/22	1/22	Access判断错误
rename	rename/mkdir/	4458/4857	399/4857	错误返回值处理与逻辑错误
rmdir	rmdir	139/145	6/145	错误处理，权限检查
symlink	symlink	95/95	❌	✔️
truncate	truncate	84/84	❌	✔️
unlink	unlink/link	403/440	37/440	mknod中的socket,错误处理
utimensat	utimens	121/122	1/122	权限检查
		7943/8674	731/8674	92%

结果说明

在pjdfstest中，错误主要集中在rename和chown的处理当中，rename是其中最复杂的部分，本项目并没有处理所有细节，chown是权限检查的主要部分，实现细节也有待斟酌。还有出现错误的位置主要是错误处理当中，一些错误值没有按照Posix标准进行返回.

5.3 mdtest 元数据性能测试

5.4 fio读写性能测试

5.5 性能优化

从上面得到的结果中可以看到，不管是大文件的读写性能还是元数据处理能力，dbfs都与其他文件系统存在较大的差距，但是按照设计方案来说，其设计与ext系列的文件系统有相似的结构，理论上不应该存在如此大的性能差距，因此可以推断其仍然具有较大的改进空间，并且导致这些问题的一定是显而易见的一些设计缺陷。通过反复实验与测试，本文观察到了几个可以提升性能的地方。

1. flush+sync_all

在数据库中, 每当一个写事务发生，根据`jammdb`的机制，会调用`flush`和`sync_all`来将文件的元数据以及缓存中的数据写回磁盘，并更新`mmap`中的只读缓存，由于缺乏写缓存且fuse位于用户态，这导致了非常大的性能开销，因此我们需要在这两个函数中作一定的优化。

具体而言, 我们在实现`jammdb`的`File`的接口时，并不直接将`flush`和`sync_all`映射到`File`的这两个接口。原来这两个接口的实现是这样的:

```
fn flush(&mut self) -> core2::io::Result<()> {
        self.file
            .flush()
            .map_err(|_x| core2::io::Error::new(core2::io::ErrorKind::Other, "flush
error"))
}
fn sync_all(&self) -> IOResult<()> {
        self.file
            .sync_all()
            .map_err(|_x| core2::io::Error::new(core2::io::ErrorKind::Other,
"sync_all error"))
}
```

对这两个接口的初步改进方案是我们直接将对文件的调用忽略，因为操作系统会为我们更新文件的数据。

WARN： 破坏事务一致性

2. 固定文件大小

在原来的实现中，本文使用本机的文件模拟dbfs的镜像文件，但这个实现是按照数据库的申请而逐渐增大文件大小的，对于ext文件系统，一开始就分配了一个固定大小的极限文件，因此在读写过程中并不会向系统再次申请文件，而对于juicefs来说，其本身不需要读取镜像文件，而是直接使用linux的文件接口。

在这个改进中，我们在创建文件时，直接初始化这个文件的大小为一个常数值，因为所有的性能测试文件大小都在一个可预测的范围，因此整个过程中数据库只会调用这个分配接口而不需要真正的去分配。同时，对于mmap系统调用，在数据库的原实现中，每一次增大文件都会重新进行映射，然后我们的改进是固定了文件的大小，因此对这个调用也同时进行了修改，我们只在第一次进行映射时记录映射的数据，后面调用这个接口时，返回的仍然是第一个数据。

```
static MMAP:Once<Arc<IndexByPageIDImpl>> = Once::new();

impl MemoryMap for FakeMMap {
    fn do_map(&self, file: &mut File) -> IOResult<Arc<dyn IndexByPageID>> {
        if !MMAP.is_completed(){
            let file = &file.file;
            let fake_file = file.downcast_ref::<FakeFile>().unwrap();
            let res = mmap(&fake_file.file, false);
            if res.is_err() {
                return Err(core2::io::Error::new(
                    core2::io::ErrorKind::Other,
                    "not support",
                ));
            }
            let map = res.unwrap();
            let map = Arc::new(IndexByPageIDImpl{map});
            MMAP.call_once(||map);
        }
        Ok(MMAP.get().unwrap().clone())
    }
}
```

3.调整块大小

对于dbfs来说，本质上是没有块大小的说法的，这里其实指的就是我们在使用key-value键值对保存文件数据时做的优化手段，这在设计文档中也有说明。

在原实现中，我们规定了这个块大小为512字节，但是在实验中发现，其他几个文件系统使用的块大小都为4kb大小，在进行fio测试时，我们设定了每次读取的大小为1MB，对于其他几个文件系统来说，每次只需要读256个块，但是对于dbfs来说就需要读取2048个键值对，而且这些键值对是需要依次查找的，其他文件系统则会尝试分配在一个连续的空间，因此增大dbfs的块大小，理论上是可以增大吞吐量的。这里改进我们增加了几个可选的块大小。

```rust
#[cfg(feature = "sli512")]
pub const SLICE_SIZE:usize = 512;

#[cfg(feature = "sli1k")]
pub const SLICE_SIZE:usize = 1024;

#[cfg(feature = "sli4k")]
pub const SLICE_SIZE:usize = 4096;

#[cfg(feature = "sli8k")]
pub const SLICE_SIZE:usize = 8192;
```

第六章 结论

6.1 总结和归纳

6.2 研究贡献

6.3 未来工作展望

参考文献

素材

开题报告

中期报告

设计文档

测试文档

操作系统+fat32+vfs 在线文档

时间安排

每章各一天

1. File Systems for Various Operating Systems: A Review ↩