华中科技大学

研究生课程考试答题本

考生姓名 余志伟

考生学号 M202173793

系、年级 计算机 21 级

类　　别 全日制专硕

考试科目 数据中心技术

考试日期 2022 年 1 月 7 日

评　　分

题　号	得　分	题　号	得　分

总　分：	评卷人：

注：1、无评卷人签名无效。
　　2、必须用钢笔或圆珠笔阅卷、使用红色。用铅笔阅卷无效。

高级文件系统综述

余志伟 M202173793

摘 要 随着存储设备的发展以及新需求的不断出现，文件系统的开发正面临着巨大的压力。然而，文件系统的开发很慢，因为很容易引入 bugs，调试和测试却很困难。并且，缺乏对不断服务的情况下更新的支持。现有的高速开发 Linux 文件系统的方法有很大的缺点，比如 FUSE 能让开发者在用户空间开发文件系统，但是频繁的内核-用户上下文切换带来了巨大的性能损失。eBPF 能够安全地为文件系统加载新的功能，但是不适合实现具有复杂并发性和数据结构需求的文件系统。

Bento 是高速开发 Linux 内核文件系统的框架。它允许用 Rust 实现的文件系统安装到 Linux 内核中。这样实现的文件系统将绝大多数隔离在系统内部。Bento 中的文件系统可以被替换而不中断正在运行的应用程序，允许在云服务器中设置每天或每周升级。Bento 也支持用户空间的调试。用 Bento 实现的文件系统，在各项基准测试中与 VFS 原生的 ext4 表现出的性能相似。在’git clone’命令测试中，比 FUSE 版本的文件系统快 7 倍。Bento 也支持正在运行的文件系统的在线升级。

关键词 高级文件系统；安全性；高性能；兼容性；高速开发；实时在线升级

1 引言

开发和部署速度是现代云软件开发的一个关键方面。高速开发将新功能更快地推送给用户，降低了集成和调试成本，并且能够对安全漏洞更快地做出响应。然而云软件的开发速度并没有跟上操作系统。在使用最广泛的云操作系统 Linux 中，发布周期仍然以月和年为单位。在云计算的其他地方，每周甚至每天都会部署新功能。

Linux 开发缓慢可归因于这几个因素。Linux 的代码量很大，代码之间的隔离少，内部接口复杂，容易被误用。再加上用 C 语言编写正确并发代码的固有困难，这意味着新代码很可能有 bugs。内核模块之间缺乏隔离意味着这些错误通常具有非直观的影响，并且难以追踪。更不用说去实现内核级调试器和内核测试框架。受限和复杂的内核编程环境让很多开发者望而却步。最后，升级内核模块需要重新启动机器或重新启动相关模块，这两种方式都会导致机器在升级过程中不可用。

漫长的开发周期对文件系统的开发来说更为严重。存储硬件的最新变化（例如，低延迟 SSD 和 NVM，以及密度优化的 QLC SSD）使敏捷存储开发变得越来越重要。同样，应用程序的多样性和系统管理需求（例如，需要容器级 SLA，或文件出处跟踪）使得开发新功能的速度变得至关重要。事实上，由于文件系统跟不上时代的步伐，人们一直在呼吁用键值存储取代文件系统，尽管有简化的接口，但它可能面临许多相同的挑战。

现有的高速文件系统替代方案要么牺牲了性能要么牺牲了通用性。FUSE 是一个广泛用于用户空间文件系统开发和部署的系统。但是，FUSE 可能会产生巨大的性能开销，特别是对于元数据繁重的工作场景。相同的文件系统，FUSE 实现比原生内核实现在‘git clone’命令测试下慢 7 倍。另一个选择是 Linux 的可扩展架构 eBPF。eBPF 是为小型扩展而设计的，例如实现一个新的性能监测器，其中每个操作都可以在有界时间内完成静态验证。因此，它不适合实现具有复杂并发性和数据结构需求的文件系统等内核模块。

对于现有广泛使用的内核（如 Linux 内核），如何在不牺牲性能或通用性的情况下实现内核文件系统的高速开发？最终学者们提出了 Bento——高速开发 Linux 内核文件系统的框架。Bento 作为 VFS 文件系统嵌入 Linux 中，但允许文件系统动态加载和替换，无需卸载或影响正在运行的应用程序，除了短暂的性能延迟。由于 Bento 在内核中运行，它使文件系统能够重用开发良好的 Linux 功能，如 VFS 缓存、缓冲区管理、日志记录以及网络通信。文件系统是用 Rust 编写的，Rust 是一种类型安全、性能良好、无垃圾收集的语言。Bento 在 Rust 文件系统前后插入抽象层，为调用文件系统和调用其他

内核函数提供安全接口。利用现有的 Linux FUSE 接口，通过更改编译选项，就可以使 Bento 文件系统在用户空间中运行。因此，大多数测试和调试都可以在用户级别进行，当代码移植到内核中时，类型安全限制了 bug 的频率和范围。由于这个接口，移植到新的 Linux 版本只需要对 Bento 进行更改，而不需要修改文件系统本身。Bento 还支持使用内核 TCP 堆栈的网络文件系统。

2 原理与优势

Linux 文件系统在需要适应存储技术和新兴应用程序需求的快速变化。要为文件系统添加新功能，开发人员必须修改、测试代码，并将修改推送到生产集群中。

最广泛使用的方法是直接修改内核源代码。Linux 具有用于扩展其子系统的标准内核接口，如虚拟文件系统（VFS）。有时，可以使用可加载的内核模块来添加新的特性，这些特性在运行时进行集成，而无需进行内核重新编译或重新启动。几个 VFS 文件系统，包括 ext4、overlayfs 和 btrfs，都在内核源代码中实现了，并且可以作为可加载的内核模块插入。

然而，使用直接修改内核源代码的方式进行高速内核文件系统开发很难。首先，Linux 内核代码是出了名的很难改对。内核代码路径很复杂，而且很容易被误用。更糟糕的是，调试内核源代码比调试用户级代码要困难得多，因为不能使用 Posix API。升级内核模块也是一种侵入性的操作。对于文件系统，这需要关闭应用程序、卸载旧的文件系统、重新安装新的文件系统，并重新启动应用程序。

除了直接修改 Linux 内核之外，还有另外两种方法可以向 Linux 添加功能，以及它们各自的优缺点。

FUSE，把新功能开发成用户空间的服务，再将系统调用转化为对用户空间服务的调用。FUSE 将低级的内存错误如释放后使用隔离到用户进程中。开发速度更快，因为工程师可以使用熟悉的调试工具，如 gdb。但所有这些都带来了性能的损耗。此外，FUSE 文件系统不能重用许多现有的内核特性，例如通过缓冲区缓存访问磁盘。

eBPF，使用内核中的解释器来动态加载内核模块。其核心思想是允许以一种安全的方式进行内核定制。在 JIT 将虚拟机指令编译为本机代码之前，eBPF 虚拟机验证程序的内存安全性和可终止性。eBPF 可以隔离不可信的扩展，但是 eBPF 的限制使得实现更大或更复杂的功能变得非常困难。

与前面介绍的几种开发文件系统的方法相比，学者提出的 Bento 有以下 6 大特点：

1.安全：新安装的文件系统中的任何错误应该尽可能限制于使用该文件系统的应用程序或容器内。

2.高性能：相同的功能，Bento 实现和 VFS 实现的性能应该相同。

3.通用性：使用 Bento，开发者能够实现任何类型的文件系统。

4.兼容性：兼容已有的操作系统和应用程序。

5.实时升级：Bento 支持文件系统的在线升级。升级对上层应用程序透明，升级时系统不可用的空窗期很短。

6.用户空间调试：Bento 支持用户空间调试，开发者能够使用 gkb 等调试工具进行快速开发。同时调试后的代码能够直接移植到内核空间中运行。

简而言之，Bento 通过 Rust 语言实现前三个特点，通过精细的系统设计实现后三个特点。Rust 语言是一种类型安全的、非垃圾收集的、通用的语言，在内核开发中越来越被重视。

为了在不牺牲安全性的情况下提供兼容性，Bento 避免直接使用 Linux VFS 接口。因为 VFS 接口要求数据结构在文件系统和内核之间直接来回传递，这使得很难验证数据结构的所有权。首先，Bento 为文件系统引入了一个基于消息传递的 API，以强制保证所有权的安全性。其次，Bento 引入了另一种不同的 API，通过将不安全的内核接口转换为 Rust 可以安全使用的内核接口，来实现对 c 语言内核服务的安全访问。至于实时升级，Bento 包含一个组件，该组件将暂停正在运行的文件系统，然后将文件系统定义的状态传输到新实例。Bento 在文件系统之间传递内存中的所有权数据结构，使它们可以在整个升级过程中被共享。至于用户空间调试，无论是在内核中还是在用户空间中，Bento 使用相同的 API 调用。所以，使用不同的编译选项就可以编译出在用户空间运行的程序或者在内核空间运行的程序。

3 研究进展

学者们提出了 Bento 架构，如图 1 所示。

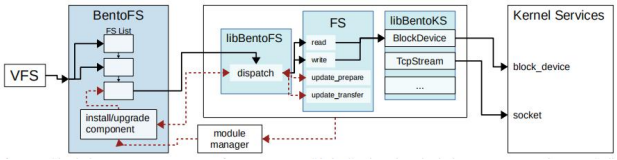

图 1　Bento 的架构

Bento 由 3 个组件构成，BentoFS、libBentoFS 以及 libBentoKS。首先 BentoFS 插在 VFS 和文件系统模块之间，管理文件系统的注册和运行。BentoFS 是用 C 语言实现，作为一个单独的内核模块插入。另外两个组件是被编译到文件系统模块中的 Rust 库。LibBentoFS 将来自 BentoFS 的不安全调用转换为由文件系统实现的安全文件操作 API 调用。LibBentoKS 为文件系统提供了一个安全的 API 来访问内核服务，例如执行 I/O 操作。文件系统本身是用安全语言 Rust 编写的，并被编译为一个 Rust 静态库，其中包括 libBentoFS 和 libBunstoKS。加载文件系统模块时，它向 BentoFS 注册，BentoFS 将其添加到活跃文件系统列表中。

BentoFS 通过一套安全的 API 与 VFS 交互。VFS 层对内存安全提出了一个根本性的挑战。例如，VFS 文件系统分配一个单个 inode 数据结构来同时保存 VFS 和文件系统特定的数据。当内核需要一个新的 inode 时，它向文件系统请求一个，文件系统从自己的内存池中分配新的 inode。双方都访问其一半的数据结构，完成后，内核将 inode 释放到文件系统，以便回收内存。不管这是否是最小化内核内存错误的良好设计模式，它与 Rust 编译时分析不一致，因此会损害学者们将 Bento 的内存错误限制在文件系统内的初衷。所以学者们设计了一套全新的安全 API 部分如表 1 所示。

Bento File Operations API (partial)
bento_init(&mut self, req, devname, fc_info)
bento_destroy(&mut self, req)
bento_read(&self, req, ino, fh, offset, size, reply)
bento_write(&self, req, ino, fh, offset, data, flags, reply)
bento_update_prepare(&mut self) -> Option<TransferOut>
bento_update_transfer(&mut, Option<TransferIn>)

表 1　Bento 文件操作 API

BentoFS 接收所有来自 VFS 层的调用，并决定哪个挂载的文件系统是调用的目标，并在内核数据结构上做必要的操作。然后，BentoFS 使用与文件系统类似的 API 向 libBentoFS 的分派函数发送请求，但使用不安全的指针，而不是 Rust 数据结构。LibBentoFS 解析请求，将指针转换为安全的数据结构，并调用文件系统中的对应的函数。这样做的核心思想是文件系统能够静态地验证它们自己的数据访问，包括 inode。要创建一个 inode，BentoFS 通过 libBentoFS 调用文件系统的接口，获得新创建 inode 的编号。反过来，BentoFS 分配并返回一个单独的内核 inode 数据结构给 VFS。BentoFS 从不触及文件系统 inode 的内容。

BentoFS 和 libBentoFS 负责确保在内存通过文件操作 API 传递时的安全性。当传递内核内存的引用给文件系统时，比如数据的读写调用，BentoFS 保证内存在调用完成之前将保持有效。如果传递了可变引用，则必须确保没有其他线程正在修改内存。当传递对结构化数据的引用时，BentoFS 和 libBentoFS 还可以确保内存的结构正确，并且永远不会被强制转换为不兼容的类型。

Bento 文件系统需要访问内核功能，比如通过块 I/O 来访问底层存储设备。这些内核接口，就像 VFS 层中的接口一样，在设计时并没有考虑到类型安全性，因此不能被 Bento 文件系统直接使用。因此，libBentoKS 实现了文件系统所需的内核数据结构和函数的安全版本。

就像许多内核接口，内核块 I/O(Linux 提供的块设备访问接口)的实现重度依赖指针。然而，直接将这些指针暴露给文件系统会导致安全错误。如果暴露给文件系统的块 I/O 函数接受一个指针，块 I/O 函数不安全，整个文件系统也不够安全。

Bento 为内核服务提供了包装抽象，以便文件系统可以安全地使用。这些抽象可以像任何其他 Rust 数据结构和函数一样使用。表 2 详细介绍了部分抽象。

Object Type	**Method**	**Kernel Equivalent**	**Description**
BlockDevice	*bread(&self, ...) -> Result<BufferHead>*	*__bread_gfp(...)*	Read a block from disk
	getblk(&self, ...) -> Result<BufferHead>	*__getblk_gfp(...)*	Get access to a block
	sync_all(&self) -> Result<i32>	*blkdev_issue_flush(...)*	Flush the block device
BufferHead	*data(&self) -> &[u8]*	*buffer_head->b_data*	Get read access to data
	data_mut(&mut self) -> &mut [u8]	*buffer_head->b_data*	Get write access to data
	drop(&mut self)	*brelse(...)*	Release the buffer
	sync_dirty_buffer(&mut self) -> Result<c_int>	*sync_dirty_buffer(...)*	Sync a block
GlobalAllocator	*alloc(&self, ...) -> *mut u8*	*__kmalloc(...)/vmalloc(...)*	Allocate memory
	dealloc(&self, ...)	*kfree(...)/vfree(...)*	Free allocated memory
RwLock<T>	*new(data:T) -> RwLock<T>*	*init_rwsem(...)*	Create a RwLock of type *T*
	read(&self) -> LockResult<ReadGuard<'_,T>>	*down_read(...)*	Acquire the read lock
	write(&self) -> LockResult<WriteGuard<'_,T>>	*down_write(...)*	Acquire the write lock
TcpStream	*connect(addr: SocketAddr) -> Result<TcpStream>*	*sock_create_kern(...)* + *kernel_connect(...)*	Create and connect
	read(&mut self, ...) -> Result<usize>	*kernel_recvmsg(...)*	Read a message
	write(&mut self, ...) -> Result<usize>	*kernel_sendmsg(...)*	Send a message
	drop(&mut self)	*sock_release(...)*	Cleanup the TcpStream
TcpListener	*bind(addr: SocketAddr) -> Result<TcpListener>*	*sock_create_kern(...)* + *kernel_bind(...)* + *kernel_listen(...)*	Create, bind, and listen
	accept(&self) -> Result<(TcpStream, SocketAddr)>	*kernel_accept(...)*	Accept a connection

表 2　部分内核服务抽象

在某些情况下，这些抽象可能会增加少量的性能开销。如果内核函数对其参数有要求，那么包装方法很可能需要执行运行时检查，以确保这些要求保持不变。

学者们也在 Bento 架构上实现了实时在线更新。为了启用对使用文件系统的应用程序透明的在线升级，必须首先明确何时升级文件系统是安全的，以及如何处理文件系统状态。如果在文件系统操作仍未完成时发生升级，可能会出现竞争条件。一些操作在旧文件系统上执行，另一些操作在新文件系统上执行，导致正确性问题。另外，任何能影响文件系统语义行为的状态，比如进程内的磁盘请求，文件系统日志，网络文件系统的 TCP 连接，在升级过程中必须妥善保存。影响性能但不影响语义的状态，比如缓存中的数据，可以选择性的保留。

Bento 通过确保旧的文件系统处于静止状态和将语义状态转移到新的文件系统来解决这些挑战。Bento 在升级期间暂停对文件系统模块的新调用，并等待正在进行的操作完成，从而使文件系统暂停。为了实现这一点，Bento 在文件系统连接上使用了一个读写锁。所有进入 libBentoFS 的调用都获得读锁，而升级则获得写锁。因此，在正常模式下，文件系统操作可以并发执行，但在升级期间将被阻塞；升级将被阻塞，直到以前的操作完成，释放读锁。

其次，对旧的文件系统的一个约束条件是，它必须能够将其语义状态传输到新的文件系统中。当然，这种状态的具体内容将因文件系统而不同。每个文件系统定义了两个数据结构：一个是在删除文件系统时返回的数据结构，另一个是在文件系统替换以前的活跃文件系统时应该返回的。这种需要编写代码来支持过去和未来版本的设计模式，在云开发中很常见。在升级过程中，BentoFS 操作数据结构的所有权将从旧的文件系统传递到新的文件系统。

详细的更新机制描述如下：

1.一个新的文件系统升级实例将被加载到内核中。在模块加载时，它调用 BentoFS 来注册自己，并表明自己是一个升级。

2.BentoFS 标识需要卸载的文件系统，并获取锁以暂停新操作并等待现有操作完成。

3.BentoFS 通过 libBentoFS 向旧的文件系统发送一个 bento_update_prepare 请求。

4.旧的文件系统实例处理 bento_update_prepare 请求，执行任何必要的清理，并通过 libBentoFS 将其定义的输出状态数据结构返回给 BentoFS。

5.BentoFS 通过 libBentoFS 向新文件系统发送 bento_update_transfer 请求，并将状态数据结构传递给新文件系统。

6.新的文件系统实例使用提供的状态初始化自己并返回。

7.BentoFS 用新的文件系统引用替换旧的文件系统引用，并释放写锁，更新操作完成。

Bento 还支持将一个新的文件系统无缝地迁移到用户空间中进行调试。这让开发者可以利用 gdb 等熟悉的工具进行高速开发。调试后的代码可以放回内核中，不需要任何修改。Bento 通过向文件系统的内核版本和用户空间版本提供相同的接口来支持此特性。文件系统是在内核中运行还是在用户空间中运行，由一个编译标志决定。该标志指定将链接哪些库以及文件系统初始化时如何注册自己。学者们利用 Linux 内核的 FUSE 支持，将文件操作转发到用户空间。因为基于 FUSE 的文件系统无法在内核中运行，所以这样做是不够的。学者们设计内核接口以镜像现有的用户空间接口，并实现用户空间库以公开额外的抽象。

许多内核接口可以被设计为与用户空间抽象相同的接口。例如，内核读写信号量的使用方式与 Rust 的 std::sync::RwLock<T>相同。在这种情况下，学者设计的内核服务 API 提供了与用户空间接口相同的接口。

然而，一些内核接口并没有明显的用户空间类似接口。例如，添加函数来实现状态转移，并传递不可变引用，以确保正确的并发行为。此外，备份存储设备上的操作与内核和用户空间不同。FUSE 文件系统通常使用文件 I/O 来访问存储设备，而内核文件系统则直接与内核缓冲区缓存进行交互。在内核中使用文件 I/O 接口严重限制了性能和功能。但是，没有一个标准的用户空间抽象可以紧密地反映内核缓冲区缓存。

为了解决这个问题，学者们提供了两个额外的库。libBentoFS 的用户空间版本将来自 FUSE 的调用转换为文件操作 API 的调用。libBentoKS 的用户空间版本实现了一个基本的缓冲区缓存，它在底层使用文件 I/O，在用户运行时为 Bento 文件系统提供块设备和缓冲区头的抽象。

学者们基于 Bento 实现了 BentoFS、其升级版附带文件追踪功能的 Bento-prov 以及在用户空间运行的 Bento-user，与 Linux 中常用的文件系统 ext4-o,ext4-j(激活日志记录功能)做对比实验测试性能和效率。

首先进行微基准测试，包括读、写以及创建和

删除操作。在所有文件系统上的读取具有相似的性能。这是因为数据在第一次读取后被快速缓存，并且所有随后的读取都在页面缓存中命中。用户空间版本在将请求转发到用户空间之前使用 FUSE 内核模块中的内核缓存，因此它的执行方式与直接内核实现类似。

对于小型写基准测试，Bento-fs 和 ext4-j 具有相当相似的写性能。由于轻微的实现差异，在大型写基准测试上 Bento-fs 的性能高于 ext4-j，与 ext4-o 性能相似。对于所有情况，Bento-user 都要慢得多，因为它会导致额外的内核交互，并发出来自用户空间的块 I/O。

在创建和删除基准测试上，ext4-j 和 Bento-fs 具有相似的性能。ext4 和 Bento-fs 的性能都优于 Bento-user，其原因与写基准测试相同。

接下来，学者们进行了不同风格的 Filebench、四种应用以及两种不同的键值对存储测试。

Filebench 测试结果如图 2 所示。在 Filebench 中，Bento-fs 的性能是 Bento-user 的 10 到 400 倍。对于 varmail 和 webserver，ext4-j 和 Bento-fs 表现出类似的性能。对于 fileserver，两者差异很大是由于对已删除文件的不同步写的处理方式不同。

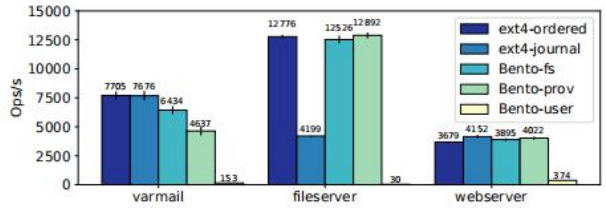

图 2　Filebench 测试结果

四种常用命令的测试结果如图 3 所示。Bento-fs 的性能为 Bento-user 的 4-36 倍。这种差异对于“untar”尤其明显，因为 untar 命令涉及到许多创造。相对于 ext4-j，Bento-fs 在“untar”、“tar”和“git clone”中的表现类似，在“grep”的要差 19%。这是由于 ext4 中的优化页面缓存功能没有在 Bento-fs 中实现。

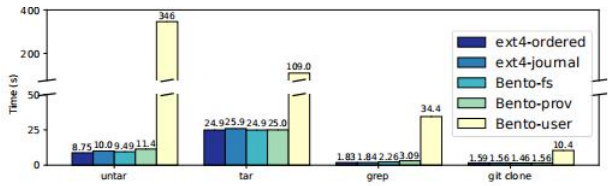

图 3　应用测试结果

键值对存储测试结果如图 4 所示。测试主要包括 Redis 的 set 和 get 和 RocksDB 的 fillrandom 和 readrandom。由于缓存，bento-user 在读取密集型工作负载上的性能与其他文件系统类似，但在写操作上的性能要差得多。Bento-fs 和便程序在读取上表现出与 ext4-j 和 ext4-o 相似的性能，但在写取上略优于它们。

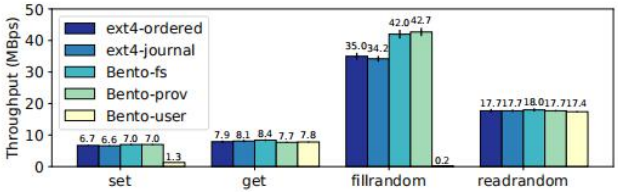

图 4　键值对存储测试结果

学者也对 Bento-fs 的实时更新功能进行了测试。测试包括两项，都是在初始包含 400000 个文件的目录下完成的。第一个测试过程中，执行着一个重复创建和删除文件的单一线程；第二个测试过程中，执行着 10 个线程，重复写入和同步 64kb 的写入到随机文件。在这两个测试中，Bento-fs 都在 0.5 秒后升级到具有来源跟踪功能的 Bento-prov，并在另外 0.5 秒后完成了测试。测试结果如图 5 所示。更新操作花费 15ms 左右，更新期间文件系统不可用，更新完成后，性能逐步恢复。测试一的性能没有回到原有水平，因为 Bento-prov 在创建删除操作中比 Bento-fs 做了更多的工作。

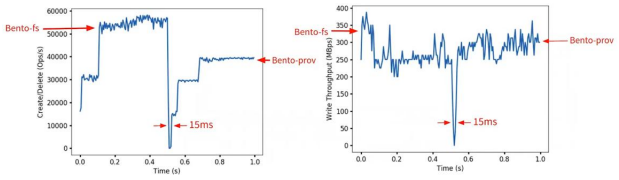

图 5　实时在线升级测试

4 总结及展望

Bento 是一个面向 Linux 内核文件系统高速开发的框架，它实现了几个目标：安全性、性能、通用性、与现有操作系统的兼容性、操作能力进行实时升级，并支持方便的调试。Bento 通过使用 Rust 语言编写文件系统来实现安全性、高性能和通用性。通过将 Linux 接口转换为具有受限内存共享的安全接口，在保持兼容性的同时不牺牲安全性。通过状态转移和读写锁来实现在线实时更新。通过统一内核空间和用户空间的接口来支持用户空间调试。

基于 Bento 实现的 Bento-fs 在测试中表现出与

ext4 相似的性能，并且性能显著优于在用户空间中运行的 Bento-fs 版本。Bento-fs 到 Bento-prov 的实时在线更新实验表明，Bento 能够实现对应用程序透明的在线更新，并且只中断服务 15ms。

Bento 的设计也存在一些局限性。虽然 Rust 的编译时分析捕捉到了许多常见的 bug 类型，但是它不会防止死锁和或语义保证，如正确的日志使用，这些错误必须在运行时进行调试。虽然在用户级别上可以进行正确性测试，但性能测试通常必须在内核中完成。此外，与其他实时升级解决方案一样，Bento 升级也要求新代码与磁盘上以前的数据兼容。Bento 目前的实现施加了一些类似于 FUSE 的可用性限制，例如，每个插入的文件系统模块只支持一个已装入的文件系统。Bento-fs 目前只是一个原型，缺乏 ext4 的很多高级功能。

参 考 文 献

[1] Miller S, Zhang K, Chen M, et al. High Velocity Kernel File Systems with Bento[C]//19th {USENIX} Conference on File and Storage Technologies ({FAST} 21). 2021: 65-79.

[2] Ashish Bijlani and Umakishore Ramachandran. Ex tension Framework for File Systems in User Space. In USENIX ATC, 2019.

[3] Abutalib Aghayev, Sage Weil, Michael Kuchnik, Mark Nelson, Gregory R. Ganger, and George Amvrosiadis. File Systems Unfit as Distributed Storage Backends: Lessons from 10 Years of Ceph Evolution. In SOSP, 2019.