



哈爾濱工業大學(深圳)

HARBIN INSTITUTE OF TECHNOLOGY, SHENZHEN

wFSCK 开发设计文档

项目成员：

姓名	年级	联系方式（QQ）
刘航（队长）	研一	512962645
刘强	研一	1614545312
孙志航	研一	965868276

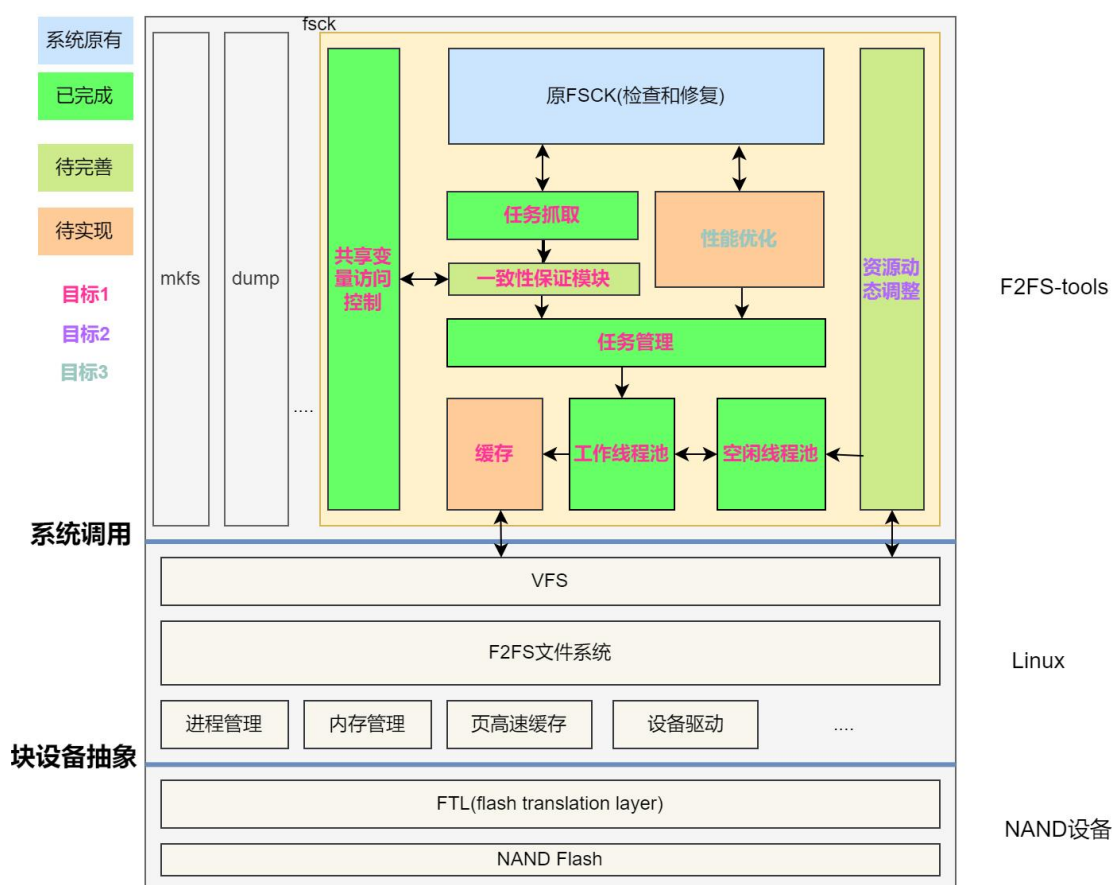
指导教师：夏文、李诗逸、仇洁婷

2023 年 05 月

摘要

wfsck 是一个智能的文件系统检查恢复和后台任务管理框架。主要包含三个方面，第一个是加速对文件系统的检查恢复。以 fsck. f2fs 作为原型，并参考 pfscck 引入多线程并发机制，加速检查和修复的过程，而不影响 C/R（检查和恢复）的正确性。并且，支持线程的 I/O cache 管理，动态调整线程个数，以减少对其他程序的影响。第二个方面是完成资源感知的动态调整后台任务线程数量的管理框架，例如动态调整后台 GC 任务线程，来减少对其他程序的影响。第三个方面是在 fsck. f2fs 检查过程中收集信息，优化后续读写等情况。fsck. f2fs 在检查文件系统的过程中，会遍历所有的文件元数据信息，这信息可以被记录下来，用于优化后续对文件系统的读写。

本项目的架构图如图所示。



本项目以第三届 OS 竞赛为驱动，旨在完成下面三个目标：

- 目标 1：完成 fsck. f2fs 的并发加速；

难点 1：如何将 pFSCK 的并发思想移植到 fsck. f2fs 中。pFSCK 的检查逻辑与 fsck. f2fs 是不一样的。pFSCK 是分阶段地进行检查，先检查 inode，再检查目录等等。但 fsck. f2fs 的检查是直接从根 Node 递归地对整个 Node 树进行检查。

难点 2：如何划分任务。理解 fsck. f2fs 原本的检查逻辑，从中划分出不同的任务。这个任务划分不能是简单的将每个 Node 的检查作为一个任务，因为每个 Node 需要对每个子 Node 检查的返回值进行处理，这样划分会影响检查的正确性。

难点 3：如何正确地对每个任务的返回值进行处理。每个任务执行完成后需要对其返回值进行处理，在并发环境下，如何保证处理逻辑的正确是一个难点。

难点 4：如何高效安全访问共享数据结构。并发情况下，如果只是简单加一个粗粒度的锁，对共享数据结构进行串行化访问，会成为系统的瓶颈。如果只是细化锁的粒度，每个变量对应一个锁，也会比较低效。需要对不同的共享数据结构进行区分，重新设计数据结构。例如有的变量是只写的，可以添加到线程私有数据，为线程设计一个类似于线程上下文的结构存放私有数据。有的变量，又读又写，加细粒度的锁处理。有的变量如目录项虽然又读又写，但是只与某个 Node 为根的 Node 树检查有关，需要添加到线程私有数据中，才能保证正确性。

难点 5：如何更智能地动态调整线程个数，使得检查工具能感知资源使用情况，做出调整，减少对其他程序的影响。

难点 6：如何充分利用硬盘 I/O。当前系统的 I/O 缓存不是为并发设计的，不同线程访问缓存可能导致低效的驱逐情况，需要为每个线程设计一个 I/O 缓存。

- 目标 2：完成资源感知的动态调整后台任务线程数量的管理框架；
- 目标 3：完成在 fsck. f2fs 检查过程中收集信息，优化后续读写等情况；

初赛的文档主要针对目标 1 进行阐述。目前，我们的赛题完成度如下：

目标编号	基本完成情况	额外说明
1	基本完成（≈80%）	① 可节省 25%到 50%左右的运行时间； ② 可动态调整线程个数 ③ 线程的 I/O cache 管理待实现

		④ 可能存在未发现的 BUG
2	初步完成 (≈10%)	完成调研, 计划将当前的动态调整线程逻辑移植到操作系统的许多后台任务中, 使其更智能。
3	初步完成 (≈10%)	完成调研, 计划通过记录 fsck. f2fs 检查获得的额外信息, 如哪些文件系统的空闲区域有哪些, 优化后续的 write 调用, 使得 write 更智能。

目 录

1. 概述	5
1.1 项目背景及意义	5
1.2 国内外研究概况	7
1.2.1 硬件和软件趋势	7
1.2.2 文件系统检查和修复	7
1.2.3 检查和修复工具 e2fsck	7
1.2.4 检查和修复工具 pfsck	8
1.2.5 F2FS 文件系统	9
1.3 项目的主要工作	19
2. 需求分析	20
3. 系统设计	21
3.1 系统整体架构设计	21
3.1.1 架构概述	21
3.1.2 系统整体运行流程	23
3.2 子模块设计	24
3.2.1 任务设计	24
3.2.2 线程池与资源动态调整设计	25
3.2.3 共享变量访问控制	26
4. 系统实现	28
4.1 核心数据结构	28
4.2 关键函数实现	34
5. 系统测试	42
5.1 测试准备	42
5.2 测试方法与测试结果	43
5.2.1 功能测试	43
5.2.2 性能测试	43
6. 总结与展望	48
6.1 工作总结	48
6.2 创新点	49
6.3 未来展望	49
参考文献	50

1. 概述

1.1 项目背景及意义

本小节主要针对 wfsc 的第一个目标加速对文件系统的检查和恢复介绍背景和意义。

wfsc 是一个智能的文件系统检查恢复和后台任务管理框架。它将 pfsc 的思想移植到了 fsck.f2fs 上。而 pfsc 是基于 e2fsck 引入了并发机制。接下来将依次介绍前面提到的几个工具，并介绍加速文件系统检查恢复的意义。

首先介绍文件系统检查恢复的几种工具。

e2fsck 是检查和修复 ext2、ext3、ext4 等文件系统的工具。而 fsck.f2fs 是检查和修复 f2fs 这个 flash 文件系统的工具。pfsc 则是在 e2fsck 的基础上引入了并发机制，加速其执行过程。并且支持额外特性。包括动态调整线程个数，减少对其他程序影响，重新设计缓存，充分利用存储设备 I/O。本项目的 wfsc 将在 fsck.f2fs 的基础上引入并发机制和相关特性，来达到加速和智能的目的。其中前面提到的四种工具特性如表 1-1 所示。

表 1-1 各工具特性

工具	原型	适用的文件系统	支持并发
e2fsck	无	ext 系列	否
pfsc	e2fsck	ext 系列	是
fsck.f2fs	无	f2fs	否
wfsc	fsck.f2fs	f2fs	是

接下来将介绍加速文件系统检查恢复的意义。

现代超高速存储设备(如 ssd、NVMe 和可字节寻址的 NVM 存储技术)提供比硬盘更高的带宽能力和更低的延迟。在 I/O 访问性能提高的同时，存储硬件和软件错误也在不断增加。长期以来，文件系统检查和修复工具(以下简称 C/R)通过识别和纠正文件系统不一致性，在提高软件存储的可靠性和系统可用性方面发挥了关键的作用^[1]。

以往的大量研究表明，在数据中心发生系统崩溃或故障的情况下，C/R 通常被用作系统恢复的第一个补救方案。以往的研究^[2,3]表明，C/R 可以跨各种场景运行。这包括由于硬件或软件错误^[2,3,4]、定期维护或强制安全升级^[5]而导致的重启期间的问题。当 C/R 以脱机方式在磁盘分区上运行时，该分区的数据不可

用。一些 C/R 支持在线检查，但至关重要，它们不会干扰使用同一设备的其他应用程序。因此，提高 C/R 性能和灵活性对于系统可用性和减少对其他应用程序的性能影响至关重要。

文件系统 C/R 工具通过识别和修复文件系统元数据的结构不一致来工作。不一致可能出现在索引节点、数据和位图、链接或目录条目结构中。广泛使用的 C/R 工具，如 `e2fsck` (Ext4 的文件系统检查工具)^[6] 将 C/R 划分为多个阶段 (通常称为 `pass`)，每个 `pass` 负责检查一个文件系统结构 (例如，目录、文件、链接)。然而，C/R 速度非常慢，随着文件和目录数量的增加，C/R 时间呈线性增长^[7, 8-11]，有时持续数小时^[5]，甚至数周^[4]。尽管现代 `flash` 和 `NVM` 技术提供了更低的延迟和带宽，但当前的 C/R 工具无法利用这些硬件功能或多核 CPU 并行性。

为了克服这些限制，我们提出了 `wFSCK`，一种并行 C/R，它利用 CPU 并行性和现代存储的高带宽来加速离线和在线形式的文件系统 C/R，从而减少系统停机时间，提高数据 (和系统) 的可靠性和可用性^[2, 9, 12, 13]。虽然 `wFSCK` 借鉴了先前的任务并行研究^[14, 15]，但它必须解决 C/R 特有的几个挑战，包括在复杂文件系统布局 and 共享文件系统结构 (例如通用位图) 中提高可扩展性，而不影响正确性，并减少 C/R 对其他应用程序的影响。`wFSCK` 引入了并行性，将 `root Node` 下的子 `Node` 的检查封装为任务，交给线程执行，这使得执行速度比传统的 C/R 要快得多。`pFSCK` 采用数据并行性，将对 `Node` 树的检查工作分解，并允许多个线程并行执行检查。虽然数据并行加速了检查，但每个任务中对全局数据的访问需要同步以确保检查的正确性，简单加锁影响了效率。对此，需要重新设计数据结构，对不同的数据进行区别，有的需要放入线程的私有数据中，有的则需要通过细粒度的锁访问。

当前 C/R 中的 I/O 缓存和预读机制等 I/O 优化不是为多线程并行设计的，我们通过设计线程感知的 I/O 缓存来解决这个问题，从而大大减少 I/O 等待时间。最后，为了在不影响共享 CPU 或访问 C/R 检查 (在线检查) 的相同磁盘的其他协同运行应用程序的性能的情况下利用多核并行性，我们设计了一个资源感知的 `wFSCK` 调度器，它通过监视系统的总 CPU 利用率来动态地扩展 C/R 线程。

1.2 国内外研究概况

我们首先简要介绍当前硬件趋势和 C/R 工具的背景，然后介绍 fsck 工具和加速后的 pfsck 工具，最后介绍我们将对其改进的 fsck.f2fs 工具。

1.2.1 硬件和软件趋势

现代超高速存储设备如 ssd 和 NVMe 不仅提供高带宽(8- 16gb /s)，而且与传统硬盘相比，存储访问延迟降低了两个数量级(< 20usec)。另一方面，像英特尔的 DC Optane^[16]这样的快速存储类内存和其他字节可寻址的持久内存技术正在发展，访问延迟< 1usec。近年来，一些新的文件系统已经发展到可以利用这些硬件优势。大量以往和正在进行的研究正在开发优化的文件系统来支持快速存储硬件。这包括 ssd^[17]、nvme^[18]的文件系统，为 nvm 优化传统 Ext4 和 XFS 文件系统的开源努力^[19]，以及其他研究工作^[20, 21, 22]。然而，减少这些文件系统的数据损坏和错误需要几年的生产使用^[23, 24]。虽然文件系统 C/R 工具将在这些文件系统中发挥关键作用，但它们尚未充分利用硬件存储优势和多核并行性。

1.2.2 文件系统检查和修复

自从文件系统出现以来，一致性一直是一个问题。尽管诸如日志记录、写时复制、日志结构写入和软更新等存储机制已经被开发出来以减少不一致性的情况，但它们是有限的，因为它们不能修复由软件错误或由故障磁盘、位翻转、过热或崩溃等事件引起的错误^[25-29]。这时候，使用流行的 C/R 工具，如 fsck、e2fsck 和 xfs_repair^[30]，通过遍历文件系统的布局并检查 inode 一致性、目录一致性、文件和目录连接性、目录项一致性以等，来检测和修复文件系统的损坏和错误。

在实际环境中，文件系统 C/ R 的频率变化很大。虽然缺乏 C/R 最佳实践，但在目前的大型和个人计算系统中，fsck、e2fsck 和 xfs_repair 等 C/R 工具对于数据可靠性仍然至关重要，因为它们通常在系统错误^[31, 2, 3, 28]、硬件或内核升级，或在强制安全更新之后运行。不频繁的 C/R 会将系统停机时间延长至 3 小时^[5]，在极端情况下，在 pb 级文件系统上，停机时间长达数周^[4]。

1.2.3 检查和修复工具 e2fsck

E2fsck 是针对 ext 系列文件系统的 C/R 工具。它的检查流程分为五个阶段。对第一个阶段 pass -1 检查索引节点元数据的一致性；第二个阶段 Pass-2 检查目录一致性；第三个阶段 Pass-3 检查目录连通性；第四个阶段 pass-4 检查引用计数；最后，第五个阶段 Pass-5 检查数据和元数据位图的一致性。

1.2.4 检查和修复工具 pfsck

pfsck 是基于 e2fsck 进行了优化。pfsck 采用四种方式来加速和优化，分别是①通过数据并行性并发执行检查任务。②通过减少 pass 间的依赖关系来启用 pass 并行性。③通过动态线程调度适应文件系统配置。④通过资源利用感知减少对系统的影响。

①通过数据并行性并发执行检查任务。

为了克服当前 C/R 工具在磁盘、卷或逻辑组级别使用串行或粗粒度并行化技术的瓶颈，pFSCK 引入了细粒度数据并行化。由于 Pass-1 和 Pass-2 占文件和目录密集型文件系统运行时的 90%以上，pFSCK 将重点放在这两个 pass 上。将更精细的文件系统结构(如 inode、目录块和目录)划分为多个任务，并在一次 pass 中并发地执行 C/R。虽然看起来很简单，但实现数据并行性需要跨线程进行数据结构隔离，以减少同步瓶颈。

②通过减少 pass 间的依赖关系来启用 pass 并行性。

虽然数据并行性加速了 C/R，但是每个 pass(例如，目录检查)都必须等待前一个 pass(例如，索引节点检查)完成。具体来说，在 C/R 中，使用了几个跨 pass 全局数据结构来构建文件系统的一致视图并识别不一致性(例如位图)。因此，对共享全局结构的更新必须序列化，从而随着线程数的增加，对共享全局结构的争用也在增加，限制了并行速度。为了减少串行化开销，pFSCK 设计了并行 pass，打破了 pass 之间串行执行的局限，允许多个 pass 同时执行，并减少 I/O 等待时间。

③通过动态线程调度适应文件系统配置。

数据和 pass 并行性都需要在不同的 pass 上分配线程。由于缺乏有关元数据类型(文件、目录、链接)，各 pass 的工作量的信息，CPU 线程的静态划分不是最优的。简单的检查，如文件数量与目录索引节点的信息是不够的，因为目录处理是复杂和耗时的。为了克服这些问题，pFSCK 设计了一个 C/R 线程调度器，它可以动态地分配和迁移线程，以便适应不同的文件系统配置。

④通过资源利用感知减少对系统的影响。文件系统 C/ R 可能与其他应用程序一起运行。考虑到 pFSCK 的目标是利用可用的 cpu，它可能会影响其他一起运行的应用程序。类似地，C/R 也可以运行在其他应用程序用来存储数据的磁盘上。为了减少整个系统对共同运行的应用程序和 pFSCK 的影响，为 pFSCK 的调度器配备了资源感知功能，以便动态识别在不同时间要使用的内核数量，以最大限度地减少对其他共同运行的应用程序和 pFSCK 性能的潜在影响。

1.2.5 F2FS 文件系统

(1) 为什么要 F2FS

要明白 F2FS 的设计原理,明白 F2FS 为什么好,我们必须从实际的背景出发。我们首先要弄清楚 F2FS 文件系统是针对什么存储设备提出的(是什么), 其次是目前的文件系统存在哪些问题(为什么), 最后是 F2FS 是如何设计来解决这些问题的(怎么做)。只有这样,我们才能体会到 F2FS 设计的精妙之处。接下来,我们将从“观察”和“总结”两个层面,循序渐进,引出 F2FS 文件的必要性。

观察:

① 基于 NAND Flash (NAND 闪存) 的存储介质, 比如 SSD, eMMC 以及 SD 卡, 相比硬盘(HDD)来说, 具有更低的访问延迟, 在随机读方面, 更是比硬盘的访问速度高出一个数量级。因此, Flash 存储介质已经被广泛地应用于从移动端设备到服务器端的各类系统。但是, Flash 存储介质仍存在一些限制, 比如: 写前擦除、有限的擦除次数, 这使得 Flash 需要按顺序写入擦除的块, 并且尽量使得各个块擦除次数一致(磨损均衡)。

② 在早期, 许多消费电子设备直接将“bare” NAND 闪存连接到一个系统。然而, 随着存储需求的增长, 使用通过专用控制器连接多个闪存芯片的解决方案越来越普遍。控制器上运行的固件通常称为 FTL (闪存转换层), 解决了 NAND 闪存的限制, 并提供了通用的块设备抽象。这种闪存解决方案的示例包括 eMMC (嵌入式多媒体卡), UFS (通用闪存) 和 SSD (固态驱动器)。通过 FTL 的抽象, 我们可以将一个 NAND 闪存设备当做一个块设备, 此时, 当前存在的针对块设备的文件系统, 可以不加修改地运行在 NAND 闪存中, 但是, 由于 Flash 本身固有的特性(写前擦除等), 大量频繁的随机写将会大大降低 NAND 闪存的性能并降低其寿命。更糟糕的是, 随机写的场景在移动端设备十分常见。

③ 20 世纪 90 年代初提出的日志结构文档系统 (LFS), 是为了缓解硬盘随机写引发的多次寻道所带来高开销而提出, 通过以类似日志的结构按顺序将所有修改写入磁盘, 从而加快了文件写入和崩溃恢复的速度, 这是一种对文件数据异地更新的方法。日志是磁盘上的唯一结构; 它包含索引信息, 以便可以有效地从日志中读回文件。为了在磁盘上维护较大的可用区域以进行快速写入, 还将日志划分为多个 segment, 并使用 segment 清理器压缩来自严重碎片化 segment 的实时信息。但是 LFS 仍存在着众所周知的漫游树 (wandering tree) 和高清理开销 (high cleaning overhead) 的问题。LFS 的思想虽然是针对硬盘首次提出, 却能够在多年后与 NAND Flash 存储介质完美结合, 解决 NAND Flash 上随机写的问题。

总结：

① 由观察 1，我们知道 NAND 闪存介质应用十分广泛，针对 NAND 闪存介质进行优化十分必要；由观察 2，FTL 可以将 NAND 闪存抽象为一个块设备，可是，针对传统块设备的文件系统不能很好地应用在 NAND 闪存介质上；由观察 3，LFS 文件系统异地更新、顺序写入的结构给了我们很好的启发，我们可以应用此思想解决在 Flash 上随机写入的问题，然而 LFS 存在漫游树和高清理开销的问题。因此，我们可以明白设计 f2fs 文件系统的必要性，即 f2fs 是一种 Flash-aware 的新型文件系统，基于 LFS，并能够解决其潜在的问题。

② F2FS 是一个利用基于 NAND 闪存的存储设备的文件系统，它基于日志结构文档系统（LFS）。该设计专注于解决 LFS 中的基本问题，即漫游树的滚雪球效应和高清理开销。由于基于 NAND 闪存的存储设备根据其内部几何形状或闪存管理方案（即 FTL）表现出不同的特性，因此 F2FS 及其工具不仅支持各种参数，用于配置磁盘布局，还支持动态调整分配和清理算法。

(2) F2FS 特性

了解了 F2FS 存在的必要性，我们就能得到 F2FS 需要拥有的一些重要特征，后面 F2FS 的磁盘布局和数据组织都是围绕着这些特性展开的。

① 闪存友好的磁盘布局：F2FS 是一种 Flash-aware 的文件系统，其磁盘数据结构经过精心布局，以匹配底层 NAND 闪存的组织和管理方式。F2FS 采用 3 个可配置单元：segment、section、zone。它以 segment 为单位从多个单独的 zone 分配存储块。它以 section 为单位进行清理，引入这些单元是为了与底层 FTL 的操作单元保持一致，以避免不必要且成本高昂的数据复制。

② 高效的索引结构：解决了 LFS 的漫游树问题。LFS 将数据和索引块写入新分配的可用空间。如果叶数据块已更新（并写入某处），则其直接索引块也应更新。写入直接索引块后，应再次更新其间接索引块。这种递归更新会导致写入链，从而产生“漫游树”问题。为了解决这个问题，F2FS 给出了一种新的索引表，称为节点地址表（Node address Table）。当叶数据块更新时，只需要更新相应索引块在节点地址表中对应的块地址即可。

③ 多头日志记录（Multi-head logging）：缓解了 LFS 高清理开销的问题。F2FS 设计了一种有效的热/冷数据分离方案，应用于日志时间（即块分配时间）。它同时运行多个活动日志段（logging segment），并根据预期的更新频率将数据和元数据附加到单独的日志段中。由于闪存设备利用介质并行性，因此多个活动段可以同时运行，而无需频繁的管理操作，因此由于多个日志记录（与单段日志记录相比）而导致的性能下降微不足道。通过 Multi-head logging，冷的日志数据段里面的 block，通常处于稳定的状态，不会被移动；而热的日志数据段，由于经常发生变化，在清理时，大部分 block 已经处于无效的状态，需要移动的有

效 block 较少，大大降低了清理的时间。

④ 自适应日志记录（Adaptive logging）：F2FS 基本上创建在仅追加日志记录（append-only logging）之上，将随机写入转换为顺序写入。然而，在高存储利用率下，它将日志记录策略更改为穿插日志记录（threaded logging），以避免长时间的写入延迟。实质上，穿插日志记录将新数据写入脏段中的可用空间，而不在前台清理它。此策略虽然在机械硬盘上不起作用，却在现代闪存设备上效果很好。

⑤ 通过前滚恢复机制（roll-forward recovery）加速：解决移动端随机写和 fsync 频繁造成 Flash 设备寿命短、延迟高的问题。F2FS 通过最小化所需的元数据写入并使用高效的前滚机制恢复同步数据，优化小型同步写入以减少 fsync 请求的延迟。通常，上层调用 fsync 时，文件系统需要将所有缓存数据同步到硬盘，在存在大量 fsync 的场景下，此操作会带来巨大的开销，F2FS 实现了高效的前滚恢复机制来增强 fsync 性能。关键思想是仅写入数据块及其直接节点块，不包括其他节点或 F2FS 元数据块。为了在回滚到稳定检查点后有选择地查找数据块，F2FS 在直接节点块内保留一个特殊标志。

(3) F2FS 磁盘布局

后面两小节将分别介绍 F2FS 的磁盘布局和数据组织，这对于后续理解 fsck.f2fs 的工作原理至关重要。首先是磁盘布局，F2FS 的磁盘布局和上文介绍的 F2FS 特性互相呼应，特性要求为磁盘布局提供了设计原则，而磁盘布局为特性提供了一个具体实现。

F2FS 将整个 volume 划分为多个 segment，每个 segment 的尺寸固定为 2MB。section 由连续 segment 组成，zone 由一组 section 组成。默认情况下，section 和 zone 大小都设置为一个 segment 大小，但用户可以通过 mkfs 轻松修改该大小。F2FS 将整个 volume 划分为六个区域，除超级块外，所有区域都由多个 segment 组成，如图 1-1 所示：

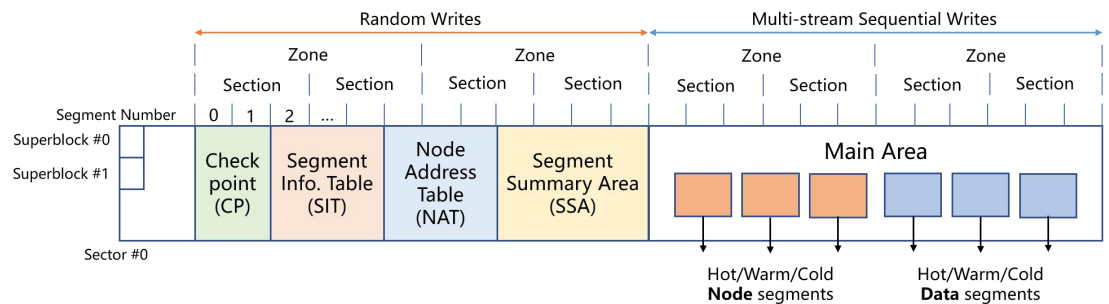


图 1-1 F2FS 布局

Superblock (SB)：它位于分区(partition，注意：文件系统构建在硬盘分区之中)的开头，并且存在两个副本以避免文件系统崩溃。它包含基本的分区信息

和 f2fs 的一些默认参数。

Checkpoint (CP): 它包含文件系统信息、有效 NAT/SIT 集的位图、孤立索引节点列表 (orphan inode lists) 和当前活动段的摘要条目 (summary entries of current active segments)。

Segment Information Table (SIT): 它包含 segment 的信息, 例如有效块计数和所有块有效性的位图。

Node Address Table (NAT): 它由存储在 main area 的所有 node blocks 的块地址表组成。

Segment Summary Area (SSA): 它包含存储在 main area 中的所有 data blocks 和 node blocks 的所有者信息的摘要条目。清理时需要根据此信息找到 main area 中某个 block 所属的 node 节点。

Main Area: 由两种类型的 block 组成, node block 或者 data block。其中 node block 包括 inode 或者 data block 的索引块, 而数据块包含目录或文件的具体数据。

为了避免文件系统和基于闪存的存储之间不一致, F2FS 将 CP 的起始块地址与 segment 大小对齐。此外, 它还通过在 SSA 区域中保留某些 segments, 将 main area 的起始块地址与 zone 的大小对齐。

(4) F2FS 数据组织

接下来介绍 F2FS 的数据组织。F2FS 的数据组织是在磁盘布局的基础之上, 对文件系统重要功能具体实现的介绍。这里不会牵涉到太多细节, 却能让我们从宏观的角度对整个文件系统设计有一定程度的把握, 这对后面明白 fsck.f2fs 工作原理也是必要的。

① 文件结构: 如图 1-2 所示, F2FS 使用基于指针的文件索引和直接、间接节点块来消除更新传播 (即“漫游树”问题)。在传统的 LFS 设计中, 如果一个叶子数据被更新, 它的直接和间接指针块被递归地更新。而 F2FS 只更新一个直接节点块及其对应的 NAT 表项, 有效解决了漫游树问题。一个 inode 块包含指向文件数据块的直接指针、两个单间接指针、两个双间接指针和一个三重间接指针。F2FS 支持内联数据和内联扩展属性, 将小数据或扩展属性嵌入到 inode 块本身。内联减少了空间需求并提高了 I/O 性能。请注意, 许多系统都具有小文件和少量扩展属性。默认情况下, 如果文件大小小于 3,692 字节, F2FS 会激活数据内联。F2FS 在一个 inode 块中预留 200 字节用于存储扩展属性。

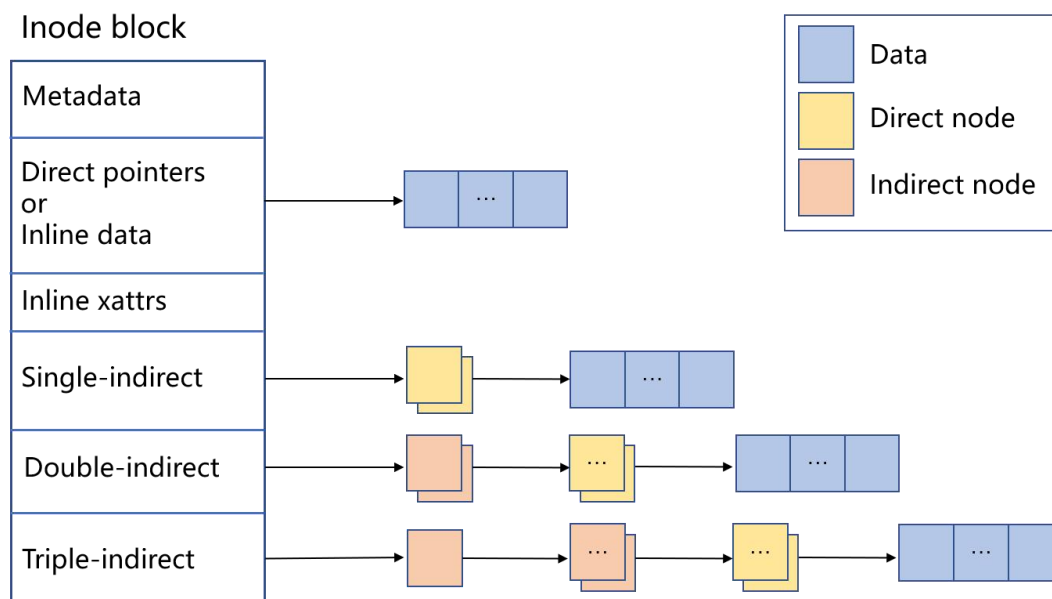


图 1-1-2 F2FS 文件组织结构

② 目录结构：在 F2FS 中，一个 4KB 目录条目（dentry）块由一个位图和两个成对的插槽（dentry 结构体、名称）数组组成。bitmap 指示每个插槽是否有效。dentry 结构体具有哈希值、索引节点号、文件名长度和文件类型（例如，普通文档、目录和符号链接）属性；而 name 是一个大小为 8 的字符数组，由于文件名的长度可能大于 8，因此，一个目录项可能会占用多个插槽。图 1-3 展示了一个目录的数据块（目录条目块）在中硬盘中的布局。

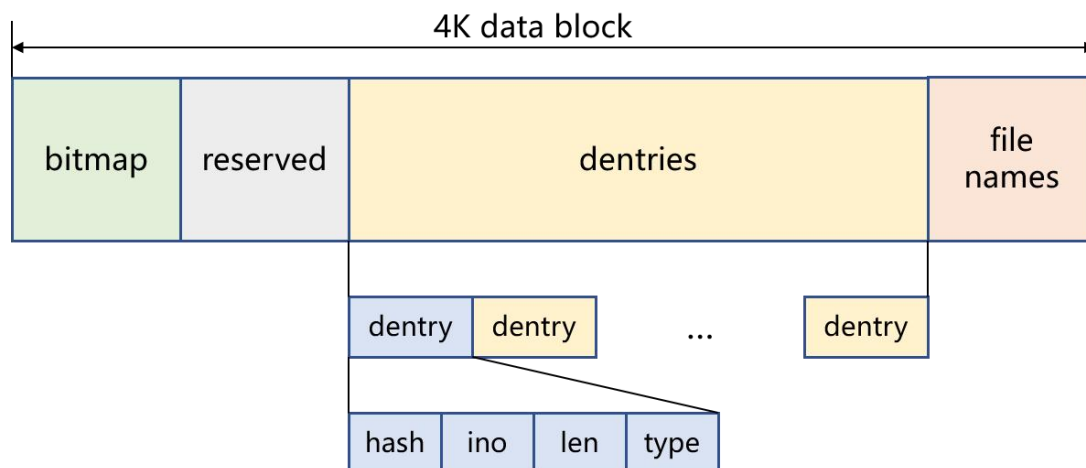


图 1-3 目录条目块 4K block

目录构造多级哈希表，以有效地管理大量目录项。当 F2FS 在目录中查找给定的文件名时，它首先计算文件名的哈希值。然后，它以增量方式遍历构造的哈希表，从级别 #0 到索引节点中记录的最大分配级别。在每个级别中，它扫描一个包含两个或四个目录条目块的存储桶，导致 $O(\log(\# \text{ of dentries}))$ 复杂性。为了更快地查找条目，它会按顺序依次比较位图、哈希值和文件名。当需要海量

的目录项时（例如，在服务器环境中），用户可以配置 F2FS 在最初时分配更多的目录项，即使用较低级别的较大哈希表，这样，F2FS 可以更快地到达目标条目。图 1-4 给出了一个多级哈希表的示意图，目录的目录条目块组织成一个多级哈希表，每一级由多个 bucket 组成，而每个 bucket 包含了多个目录条目块。搜索某个文件时，在每一级对应的 bucket 中依次搜索目录项，且每一级只会搜索一个 bucket。举一个例子：当 F2FS 在目录中查找某个文件名时，首先计算文件名的哈希值。然后，F2FS 扫描级别 #0 中的哈希表，以查找由文件名及其索引节点号组成的条目。如果未找到，F2FS 将扫描级别 #1 中的下一个哈希表。通过这种方式，F2FS 从 0 到 N 以增量方式扫描每个级别的哈希表。

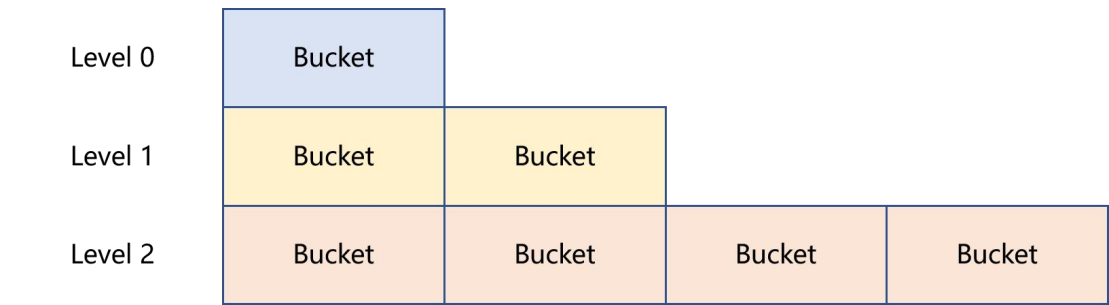


图 1-4 目录多级哈希表

③ 日志记录（Logging）：与只有一个大日志区域的 LFS 不同，F2FS 维护六个主要日志区域，以最大限度地实现冷热数据分离的效果。F2FS 静态地为节点和数据块定义了三种温度级别——hot、warm 和 cold，如表 1-2 所示。直接节点块被认为比间接节点块更热，因为它们更新得更频繁，而间接节点块包含节点 ID（指向了下一个节点块），仅在增加或删除特定节点块时写入。目录的直接节点块和数据块被认为是热的，因为与普通文件的块相比，它们具有明显不同的写入模式。某些数据块被认为是冷的，如多媒体数据，因为它们一般不会被写入，通常是只读的。

表 1-2 不同对象的划分

Type	Temp.	Objects
Node	Hot	Direct node blocks for directories
	Warm	Direct node blocks for regular files
	Cold	Indirect node blocks
Data	Hot	Directory entry blocks
	Warm	Data blocks made by users
	Cold	Data blocks moved by cleaning; Cold data blocks specified by users; Multimedia file data

LFS 有两种空闲空间管理方案：穿插日志(threaded log)和仅追加日志(append

log)。仅追加日志方案非常适合具有非常好的顺序写入性能的设备，因为空闲段一直用于写入新数据。然而，在高利用率的情况下，它会受到清理开销的影响。相反，穿插日志方案不得不采用随机写，这会降低写入性能，但不需要清理过程。F2FS 采用混合模式，默认采用仅追加日志，但根据文件系统状态动态更改策略为穿插日志模式（如空闲的 segment 数量少于 K 时，变换为穿插日志模式，而 K 是一个预定义的值）。

为了使 F2FS 与底层基于闪存的存储保持一致，F2FS 以 section 为单位分配 segment。F2FS 期望 section 大小与 FTL 中垃圾收集的单位大小相同。此外，对于 FTL 中的映射粒度，F2FS 尽可能地在不同的 zone 中分配活动日志，否则，由于 FTL 可以根据其映射粒度将活动日志中的数据写入一个分配单元，这就违背 multi-head logging 的初衷，并且无法缓解系统清理开销。

④ 清理：F2FS 可以根据需要（on demand）和在后台（in the background）进行清理。当没有足够的空闲段来服务 VFS 调用时，触发按需清理。后台清理器由内核线程操作，在系统空闲时触发清理作业。

F2FS 支持两种受害者选择策略（victim selection policies）：贪心算法和成本-收益（cost-benefit）算法。在贪心算法中，F2FS 选择有效块数量最少的受害段（victim segment）。在成本效益算法中，F2FS 根据 segment 的年龄和有效块的数量选择受害段，以解决贪心算法中的日志块抖动问题。F2FS 按需清理采用贪心算法，后台清理采用成本效益算法。这是因为用户需等待按需清理完成，此时间必须足够短，因而采用贪心算法；而后台清理是系统空闲时进行，系统有足够的时间做出最优决策，这时可以选择时间长但效果更好的成本效益算法。

为了识别受害段中的数据是否有效，F2FS 管理一个位图。每个位代表一个块的有效性，位图由覆盖 main area 所有块的位流（bit stream）组成。此位图保存在 SIT 表中。

⑤ 检查点和恢复：F2FS 实现检查点，以便在突然电源故障或系统崩溃时提供一致的恢复点。当它需要在 sync、umount 和前台清理等事件中保持一致状态时，F2FS 触发一个检查点过程，如下：(1)刷新页面缓存中的所有脏节点和 dentry 块；(2)暂停普通的写活动，包括 create、unlink 和 mkdir 等系统调用；(3)将文件系统元数据（NAT、SIT 和 SSA）写入磁盘上各自的专用区域；(4)最后，F2FS 写一个检查点包（checkpoint pack）到 CP 区域，其包括以下信息：

Header 和 **Footer** 分别写在 pack 的开始和结束。F2FS 在 Header 和 Footer 中维护一个版本号，该版本号在创建检查点时递增。版本号在挂载期间区分两个记录的 pack 之间的最新的稳定的 pack；

NAT 和 SIT 位图表示包含当前 pack 的 NAT 和 SIT 块的集合；

NAT 和 SIT 日志包含少量最近修改的 NAT 和 SIT 条目，以避免频繁的 NAT 和 SIT 更新；

活动段的摘要块（**summary block**）由内存中的 SSA 块组成，这些块将在将来被刷新到 SSA 区域；

孤儿块（**orphan blocks**）保存“孤儿 inode”信息。如果一个 inode 在关闭之前被删除（例如，两个进程打开一个公共文件，一个进程删除它），它应该被注册为孤立 inode，以便 F2FS 可以在突然断电后恢复它。

checkpoint 在硬盘中的表示如图 1-5 所示，可以看出 checkpoint 分为两种模式——Normal 模式和 Compacted 模式。在 Normal 模式下，每个日志区域都有一个块来保存其摘要信息；而 Compacted 模式下，多出了保存 nat journal 和 sit journal 的块，而所有的数据（hot、warm、cold）的日志区域共享一个摘要块。上面提到的 NAT 和 SIT 的位图，在图中没有显示出来，通过查看 f2fs 源码可知，其位于 f2fs_checkpoint 结构体的后面（整个 f2fs_checkpoint 的大小不足 4K，后面的部分充当 NAT、SIT 的位图）。

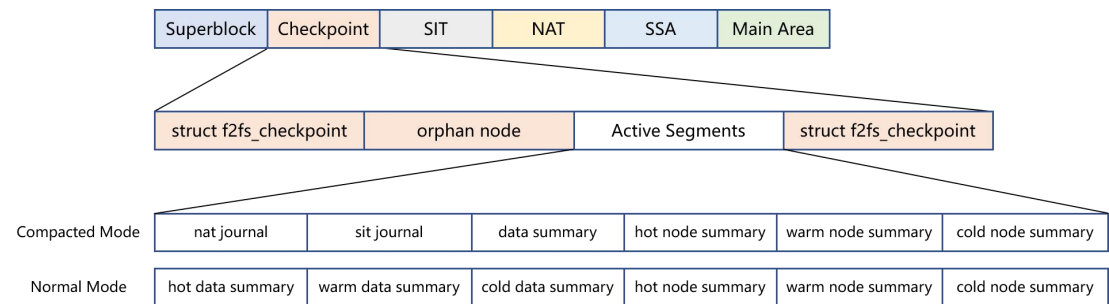


图 1-5 Checkpoint 结构

后向回退修复（**Roll-Back Recovery**）：在突然断电后，F2FS 回滚到最近的一致检查点。为了在创建新包（Pack）时保持至少一个稳定的检查点包，F2FS 维护两个检查点包。如果检查点包在 Header 和 Footer 中具有相同的内容，F2FS 认为它是有效的。否则，它将被丢弃。在挂载时的恢复过程中，F2FS 通过检查 Header 和 Footer 来搜索有效的检查点包。如果两个检查点包都有效，F2FS 通过比较它们的版本号来选择最新的一个。一旦选择了最新的有效检查点包，它就会检查孤儿 inode 块是否存在。如果是这样，它将截断它们引用的所有数据块，最后也释放孤儿 inode。最终，在前滚恢复过程成功完成之后（在下文介绍），F2FS 使用一组一致的由位图引用的 NAT 和 SIT 块启动文件系统服务。

前向回滚修复（**Roll-Forward Recovery**）：像数据库（例如 SQLite）这样的应用程序经常将小数据写入文件并进行 fsync 以保证持久性。支持 fsync 的最简单的方法是触发检查点并使用后向回退模型恢复数据。然而，这种方法会导致较差的性能，因为检查点涉及到写入与数据库文件无关的所有节点和 dentry 块。F2FS

实现了高效的前滚恢复机制，提高了 `fsync` 性能。关键思想是只写数据块及其直接节点块，不包括其他节点或 F2FS 元数据块。为了在回滚到稳定检查点后选择性地查找数据块，F2FS 在直接节点块中保留了一个特殊标志。

(5) 检查和修复工具 `fsck.f2fs`

`fsck.f2fs` 的工作流程如图 1-6 所示，检查主要分了三个步骤：初始化、修改以及验证。初始化流程主要是根据读取硬盘得到的信息，对 NAT 区域位图、SIT 区域位图和 MainArea 位图信息进行一个初始化，在后续修改的步骤，会使用这些初始化的信息，对文件系统的一致性进行验证；修改步骤是检查的核心，也是我们对 `fsck.f2fs` 修改得最多的部分，这里的逻辑是从根目录对应的 `inode` 出发，遍历整个文件系统，并且在此过程中记录对应的信息，比如，记录下整个 Main Area 区域中有效的块数、每个文件的链接数，Main Area 区域中各个 segment 中对应块有无被使用的位图等，通过此类信息，我们将能够验证文件系统的有效性；验证步骤即对前面收集到的信息与硬盘记录的元数据信息进行比对，来对文件系统进行检查，如果检查到了文件系统的不一致，用户可以选择是否调用 `fsck_xxx_func` 类型的函数对文件系统进行修复。

由于我们的工作核心在于 `fsck_chk_node_blk` 函数，也就是上面提到的修改步骤，我们在下面再重点介绍一下此步骤。如图 1-6 步骤 2 所示，`fsck` 程序通过硬盘保存的元数据信息（`do_mount` 后保存在 `sbi` 结构体中），我们可以获取根目录对应的 `node id`。通过 `fsck_chk_node_blk` 对此 `node id` 进行检查，这个过程是递归的。具体来说，在 `fsck_chk_node_blk` 里，会根据不同的 `node` 节点类型（上面提到 `node` 分为 `inode` 类型、`dinode` 直接数据块、`idinode` 间接数据块等），调用不同的检查函数进行检查，比如根目录，对应的 `node` 节点为 `inode` 类型，因此会调用 `fsck_chk_inode_blk` 进行检查，在 `fsck_chk_inode_blk` 中，首先根据读取到的 `inode` 节点信息，记录 `inode` 硬链接数等，后面将会对此类元数据进行验证；然后是对数据的处理，会根据此 `inode` 是否存在内联的数据来决定如何进行下一步操作，如果不存在内联的数据，这会遍历每个数据块，通过调用 `fsck_chk_data_blk` 进行检查；除了 `inode` 节点存在的数据指针直接指向数据块外，`inode` 节点还存在 5 个 `node` 指针间接地指向了数据块。因此，如果存在这类 `node` 指针，`fsck_chk_inode_blk` 中会调用 `fsck_chk_node_blk` 对 `node` 进行检查，形成了一个递归的结构。如果是目录的数据块，里面保存了目录项的信息，因此，在 `fsck_chk_data_blk` 里要进一步调用 `fsck_chk_dentry_blk` 对目录项进行检查，`fsck_chk_dentry_blk` 收集了目录项的信息后，调用 `_chk_dentries` 真正进行各个目录项的检查操作。如果某个目录项是有效的，`_chk_dentries` 内还会对此目录项记录的 `inode` 信息进一步调用 `fsck_chk_node_blk` 进行检查，再次形成了一个递归的结构。图 1-6 步骤 2 蓝色标记的块和相应箭头描述了我们上面所述的各个函数以

及他们之间的关系。

注意，我们没有刻画 fsck.f2fs 中对扩展块，xattr 块等的描述和检查操作，流程图也省略了很多细节，因为我们只想勾勒出 fsck.f2fs 检查的具体轮廓和步骤，太多细节反倒会让人迷惑。但是，实际 fsck.f2fs 检查要考虑很多细节，并且数据之间的依赖也错综复杂，在其中引入并行性不是一件容易的事情，我们已经做了大量的努力来维护系统的一致性，包括但不限于引入细粒度锁、线程独立的私有空间、线程等待和数据聚合等。

do_fsck

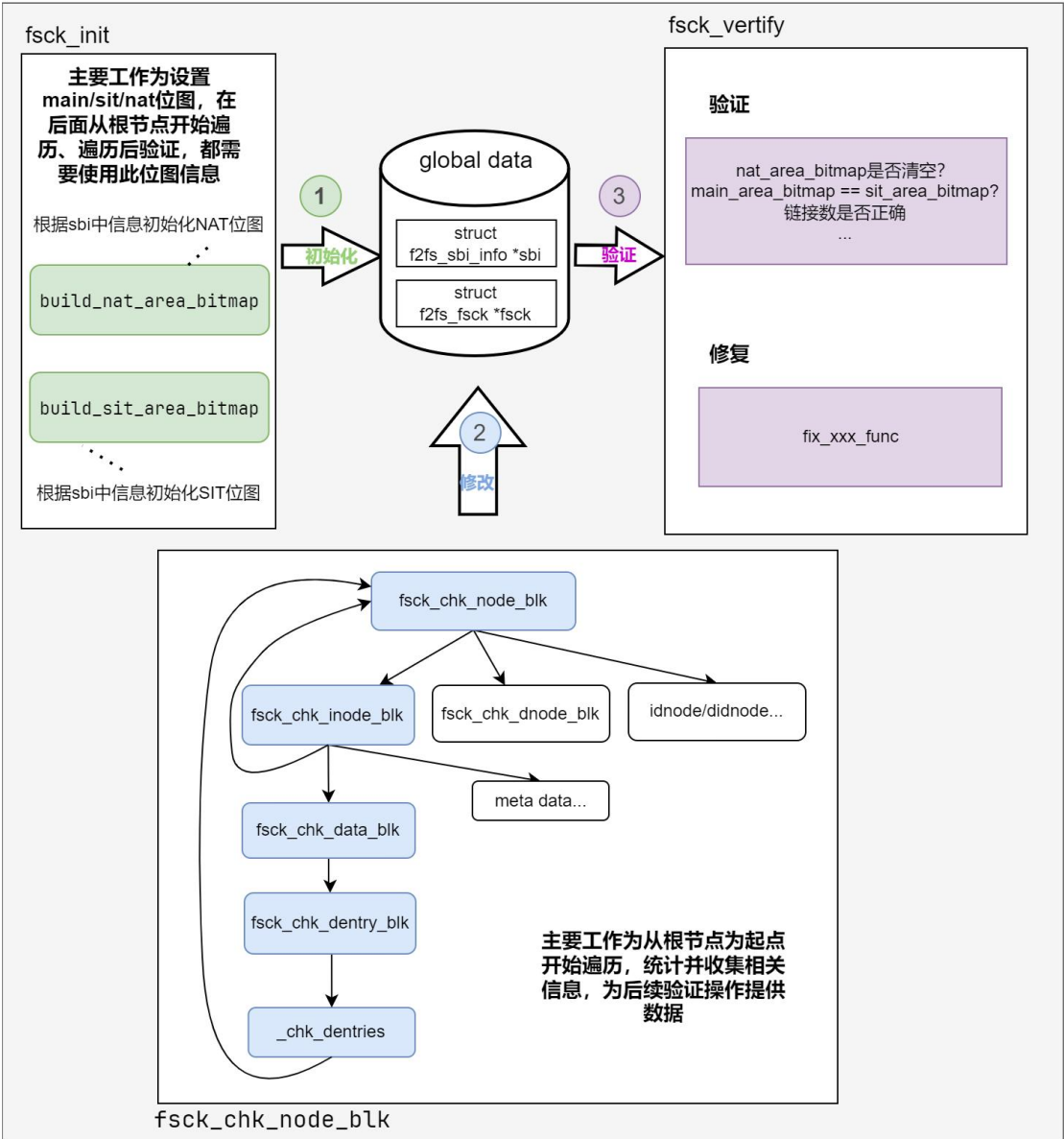


图 1-6 fsck.f2fs 工作示意图

1.3 项目的主要工作

项目的主要工作为以下三个题目:

- **题目一: fsck.f2fs的并发加速**

实现fsck.f2fs进行C/R的加速, 并动态调整线程数量, 减少对其他程序的影响。

- **题目二: 后台任务线程数量动态调整框架**

linux中有许多后台任务, 比如gc任务, 为了不影响其他进程运行, 需要通过监控系统资源使用情况, 动态调整其运行的线程数量。

- **题目三: fsck.f2fs检查过程中收集信息, 优化后续读写等情况**

在 C/R 时, 已经遍历了各文件的元数据信息, 通过将这些信息记录下来, 对后续读写等情况进行优化。

2. 需求分析

分析赛题可知，我们需要实现更智能的 flash 文件系统，而检查与恢复对于提高 flash 文件系统的可靠性十分重要。而如今 C/R 的时间不断增长，随着文件和目录的增加，C/R 的时间呈线性增长，有时持续数小时，甚至数周。尽管现代 flash 和 NVM 技术提供了更低的延迟和带宽，但当前的 C/R 工具无法充分利用这些硬件 I/O 或多核 CPU 并行性。

为了使其更快速，更智能。我们需要引用并发机制加速 fsck.f2fs 的执行速度。通过引用智能感知资源的机制，让 fsck.f2fs 智能调整线程个数，减少对系统其他程序的影响。

同时实现一个智能的资源感知的动态调整后台任务线程数量的管理框架，将系统后台任务都智能地管理起来。

最后需要充分利用 fsck.f2fs 检查过程中收集信息，优化后续读写等情况。

3. 系统设计

3.1 系统整体架构设计

系统的整体架构图如图 3-1 所示。其中字体颜色表明了模块属于哪个目标，填充的颜色表明了模块的完成情况。接下来将对架构进行概述，并介绍系统整体运行流程。

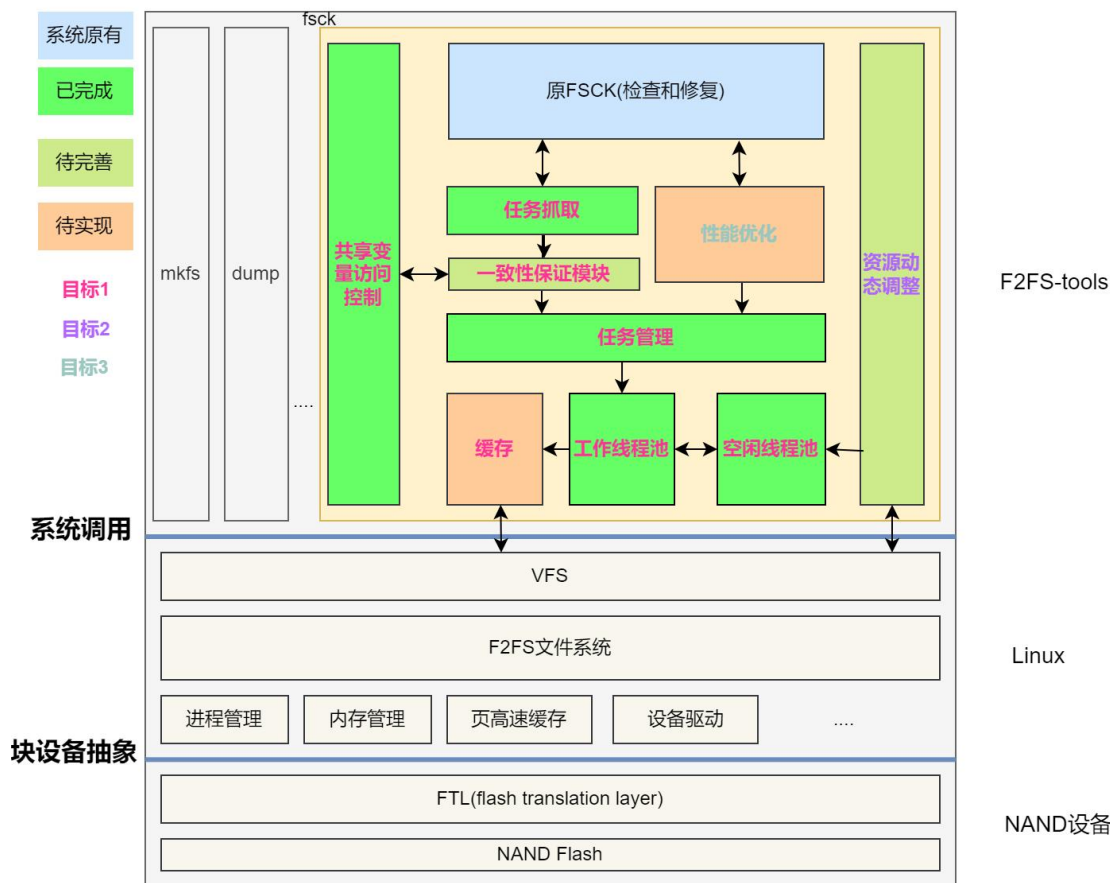


图 3-1 系统整体架构图

3.1.1 架构概述

● 任务管理

任务的管理包括检查任务和调度任务。检查任务就是执行文件系统检查的任务。调度任务是动态调整工作线程个数的任务。检查任务如何划分是一个难点。需要将原始从根 Node 开始的递归地对整个 Node 树的检查进行拆分，拆分成不同的任务，同时对任务的返回值正确地处理。检查任务会被加入到工作线程池，以先入先出的方式被工作线程池里的线程执行。调度任务会直接加入到线程池工作队列的头部，拿到该调度任务的线程将被调度到另一个线程池上去。

● 资源动态调整

这里的资源指线程个数，调度器线程会根据系统的 CPU 利用率和当前进程的 CPU 利用率动态调整当前进程的线程个数。对于检查工具就是调整执行检查任务的线程个数。同时我们计划完成一个后台任务管理的框架，将系统其他的许多后台任务也加入到资源动态调整的管理中，使得后台任务是资源感知地在运行，更加智能地执行任务，而减少对系统的影响。

- 共享变量访问控制

共享变量访问控制需要安全和高效。安全指的是各线程并发读写不会影响结果的正确性，高效是指执行速度尽量快。对共享数据结构的处理有两种，一种是加锁，一种是变为线程的私有数据。由于原本的单线程逻辑涉及到大量对共享数据结构的访问，在改为并发逻辑后，需要保证访问共享数据结构的原子性。一种简单的实现方式就是加锁，但是若对所有变量都加同一把锁，反而会使得执行时间变得更长。进一步的优化则是对不同的变量加不同的锁。更进一步则是将全局数据分散到各个线程的私有数据中去，通过调用线程库，实现一个类似于线程上下文的东西。该线程对全局数据的访问或更新改为对线程上下文中私有数据的访问或更新。进一步加快执行速度。但不是所有数据都能变成线程的私有数据，若该数据既被各线程读又被各线程写且不只与该线程处理的 Node 有关，则该数据只能加锁处理，常见的是 bit map 相关的数据结构。而有的数据，如目录项链表，又被读又被写，但是是由各线程动态向链表中添加目录项，删除目录项，最后目录项链表会为空，该数据可以加入到线程私有数据中。同时需要在所有任务执行完后，对线程私有数据进行结果的聚合。如各线程记录了该线程遍历到的有效的 inode 个数，在最后需要将各线程有效的 inode 个数相加，得到系统总的有效 inode 个数。

- 一致性保证模块

保证整个 C/R 过程的正确性。这需要正确地做到共享变量访问控制和任务管理。包括对任务返回值的正确处理，正确地记录调用任务前的上下文，正确地将控制不同线程访问共享变量，正确地将各个线程的检查结果进行聚合。

- I/O 缓存

当前检查工具的 I/O 缓存不是为并发环境而设计的。不同线程访问的是磁盘的不同位置。并发情况下，很可能会导致缓存的一些错误驱逐的情况，无法充分利用现代存储设备的 I/O。对此需要重新设计数据结构，每个线程一个 I/O 缓存。并且需要智能地自适应调整预取窗口地大小，以免预读过多不需要的内容。

● 性能优化

在文件系统的检查过程中，需要遍历文件的元数据信息，而这些信息很可能是可以用于其他优化的。比如收集当前文件系统空闲区域的情况，优化后续的读写等。

3.1.2 系统整体运行流程

目标 1 的系统整体运行流程如图 3-1 所示。原本的单线程的检查流程是①检查元数据信息。②从根 Node 出发，对根 Node 下的各子 Node 进行递归检查，并对递归的返回值进行处理。③对检查结果进行核对。而引入并发机制后，将②中的递归调用都封装为一个任务，放入任务队列。线程池里的线程会不断从任务队列里拿任务进行处理。由于原来的逻辑会对递归调用的返回值进行处理，所以任务执行完成后，会对返回值进行处理。同时调度器线程会周期性运行，根据系统资源使用情况，动态调整线程个数。调整的方式为向空闲线程池中放入调度任务，空闲线程池中线程拿到调度任务后会把自己加入工作线程，执行其任务队列里的任务。

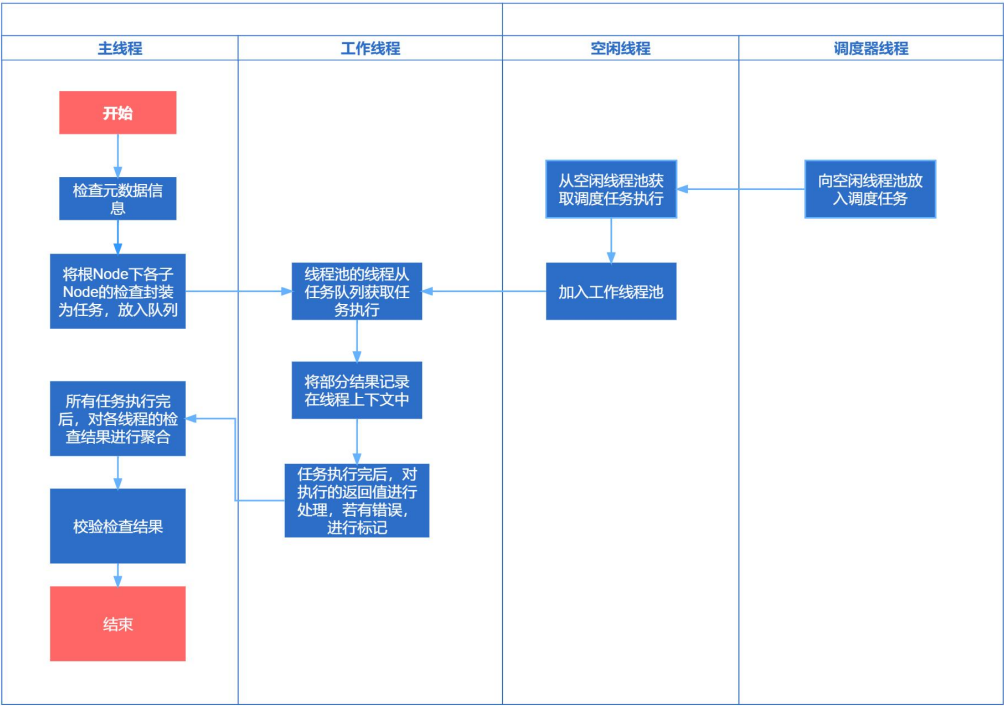


图 3-2 系统整体运行流程

3.2 子模块设计

本小节将介绍各子模块的设计，包括任务设计，线程池设计，线程上下文和锁设计。分别对应了检查的不同步骤，①将任务划分出来加入线程池/任务返回值处理。②线程从线程池里取出任务执行。③在执行过程中需要加锁访问特殊变量或者访问线程的上下文。其中各模块的具体实现放在第 4 节中进行讲解。

3.2.1 任务设计

本小节介绍每个线程执行的任务是如何划分，如何既高效又能保证正确性。

如前文所说，任务的划分是一个难点。首先 fsck.f2fs 的检查逻辑是从根 Node 开始递归地对整个 Node 树进行检查，并且每个 Node 都要对子 Node 的返回值进行处理。若只是将每个 Node 都作为一个任务加入线程池，那么任务的返回值将不好处理。比如 Node 需要检查完其所有子 Node 后将有效子 Node 个数和父 Node 中记录的个数进行比对。所有不能随意将每个 Node 都作为一个任务加入到任务队列中。

对此，我们将 root Node 下一级的子 Node，也就是图 3-2 中绿色的 Node 作为任务添加到任务队列。注意，仅仅是下一级，不包括图 3-2 中红色的 Node。这样这些子 Node 对返回值的处理就可以复用原始的代码，不用修改。接下来只需要考虑如何在 root Node 中对所有子 Node 的返回值进行处理。对此，我们将每个任务执行前的一些上下文保存下来，在任务结束后，使用这些上下文，对任务的返回值进行处理。

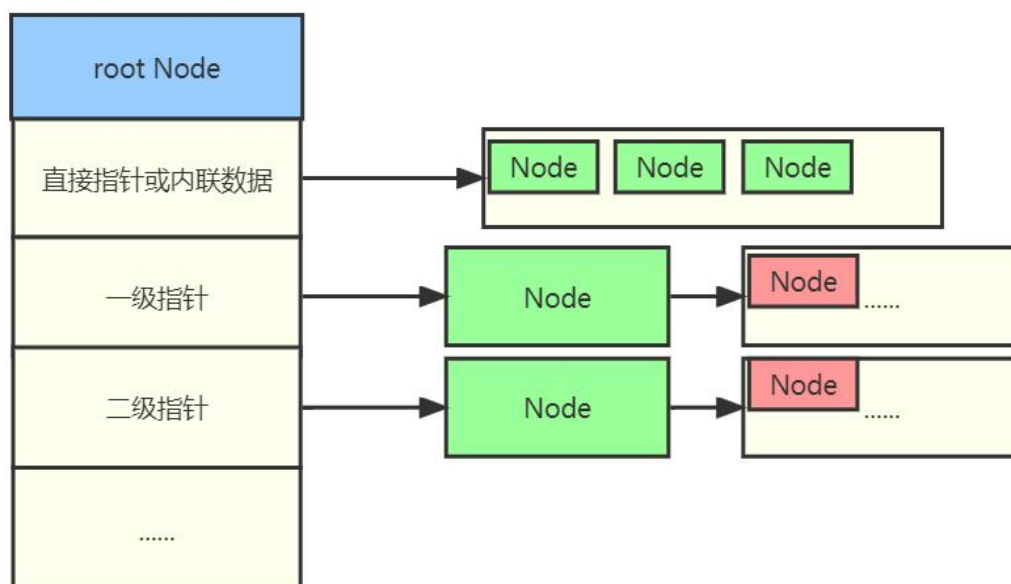


图 3-3 任务划分示意图

3.2.2 线程池与资源动态调整设计

本小节介绍线程池是如何设计，线程数量是如何动态调整的。

线程池设计如图 3-3 所示，包含两个线程池。一个是工作线程池，该线程池里的线程负责执行检查任务。工作线程池里的任务队列会包含检查或调度任务，空闲线程池里的任务队列在大部分情况下是空的，这样空闲线程池里的线程获取不到任务，就会让出 CPU。有时空闲线程池里会有调度任务，线程获取到调度任务就是调度自己到其他线程池。线程池里的初始线程个数可通过运行时参数指定。同时调度器线程会周期地运行，通过获取系统的资源使用情况结合当前检查工具的资源使用情况，动态调整线程个数。这个动态调整就是往线程池的任务队列里放一个调度任务，拿到这个任务的线程就会将自己转移到另一个线程池中去。比如图所示，当前工作线程池有两个线程在不断从任务队列获取任务执行。而此时调度器线程休眠的时间结束又到了工作的周期，它监控系统资源情况，发现系统的 CPU 利用率并不高，选择从空线程池中转移线程 3 到工作线程池。调度器线程就会向空闲线程池的任务队列添加一个调度任务。该调度任务包含了，任务类型，标识其为一个调度任务，标识了目标线程池，当线程 3 拿到这个调度任务时，就会将自己所属的线程池改为工作线程池。因为线程是在一个死循环里不断获取自己所属的线程池任务队列里的任务，所以当下一次循环开始时，线程 3 获取的就是工作线程池的任务队列里的任务，这样线程 3 就从空闲线程池转移到了工作线程池。当然，空闲线程池中不一定是线程 3 获取到调度任务，也可能是线程 4 获取到调度任务，无论如何，只会有一个线程获取到调度任务，从而转移到工作线程池中。

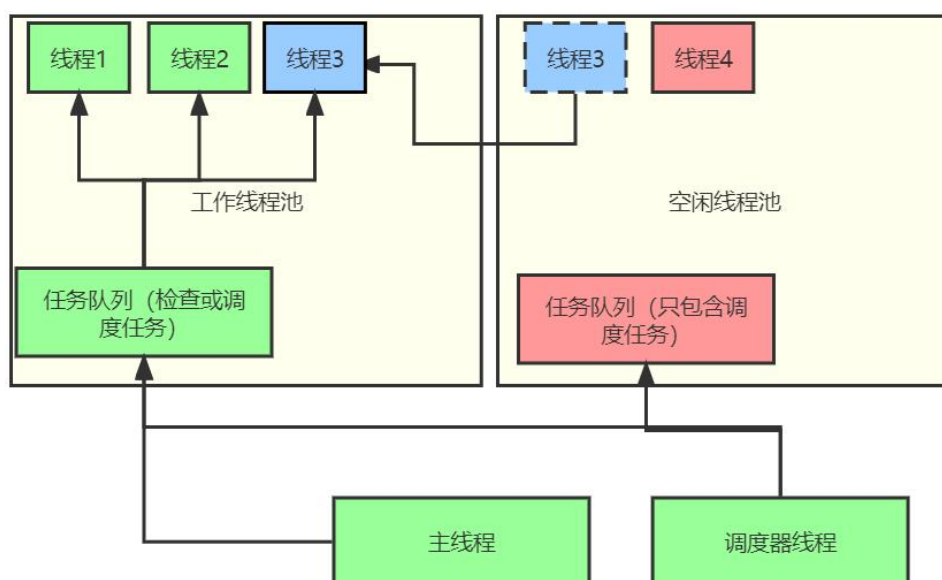


图 3-4 线程池设计

3.2.3 共享变量访问控制

本小节介绍并发情况下对共享变量访问控制，使用的手段包括线程上下文和锁。

并发情况下，为了保证 C/R 的正确性，重新设计数据结构，解耦数据是一个关键问题。根据测试，如果只是简单对所有共享数据结构的访问加一把大锁，那么对共享数据结构的串行化访问会成为系统的瓶颈，执行的时间反而增加。对此，我们的优化是细化锁的粒度和设计线程上下文。如图 3-4 所示，变量 A 本来是共享数据结构，但在并发环境下对其的访问需要加锁。变量 B 本来是一个共享数据结构，但是每个线程的上下文中都有一份私有的变量 B，线程在对该变量 B 进行读写时，是对该线程上下文的私有变量 B 进行读写。主线程会在所有任务执行完成后，将各个线程的私有变量进行聚合，比如变量 B 是一个计数值，主线程会将所有线程的私有变量 B 累加到共享数据结构 B 中。这些变量记录了检查结果，后续主线程会使用该结果进行最后的校验。同时，各个线程对磁盘的读写也要加锁处理，保证正确性。

各个变量不同处理分为以下类型：

- 只写的数据：

放入线程上下文。各线程可以将该变量记录在自己的线程上下文，所有任务执行完后，由主线程聚合结果。

- 只读的数据：

未额外处理。因为各线程只对该变量进行读操作，不影响 C/R 的结果。

- 又读又写的数据：

加锁处理。因为该变量被修改后，又可能被其他线程读到，所以还是作为共享数据结构，只是访问要加锁。这类变量大多是 bitmap 位图之类的数据结构。

- 只有放入线程上下文才能保证正确性的数据：

放入线程上下文。比如目录项的链表，只与某个 Node 为根的 Node 树有关。在检查时线程会不断向这个链表加入或删除结点，检查结束后，这个链表为空。从变量的含义上看，并发环境下，这个变量需要作为每个变量的私有数据，放入线程上下文，才能保证逻辑正确。

- 磁盘的数据：

磁盘上的数据的访问也要加锁处理，否则会出现错误。

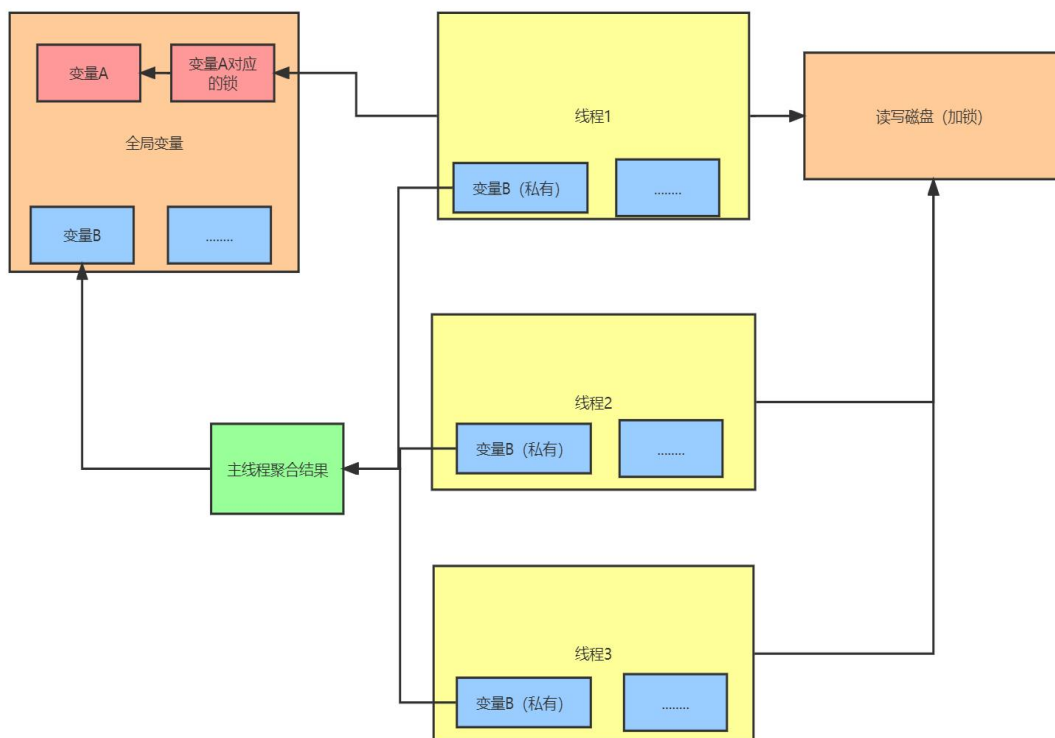


图 3-5 共享变量访问控制

4. 系统实现

4.1 核心数据结构

以下将给出重要数据结构的定义。

- **thread_ctx**

thread_ctx 代表线程的私有数据，类似于线程上下文，用于保存该线程检查得到的结果。图 4-1 显示了其核心变量。

```
struct thread_ctx {  
  
    //Thread information  
    int tid;  
    u64 checked_node_cnt;  
    u64 valid_blk_cnt; // ** main area中有效的block数目: node + data  
    u32 valid_node_cnt; // ** main area中有效的node block数目  
    u32 valid_inode_cnt; // ** main area中有效的inode block数目 (node block的子集)  
    u32 multi_hard_link_files; // 有多个 hard_link 文件的个数  
  
    u32 dentry_depth;  
    struct f2fs_dentry *dentry;  
    struct f2fs_dentry *dentry_end;  
  
};
```

图 4-1 thread_ctx 核心变量

表 4-1 显示了各变量的含义。

表 4-1 thread_ctx 核心变量含义

变量名	数据类型	变量描述
tid	int	线程 id
checked_node_cnt	u64	已检查的 node 数目
valid_blk_cnt	u64	有效的 block 数目
valid_node_cnt	u32	有效的 node block 数目
valid_inode_cnt	u32	有效的 node block 数目
valid_blk_cnt	u64	有效的 block 数目
valid_node_cnt	u32	有效的 node block 数目
valid_inode_cnt	u32	有效的 node block 数目
multi_hard_link_files	u32	有多个硬链接的文件数目
dentry_depth	u32	目录项深度
dentry	struct f2fs_dentry *	目录项链表
dentry_end	struct f2fs_dentry *	目录项链表表尾

● f2fs_fsck

f2fs_fsck 代表检查过程的全局数据。图 4-2 显示了其核心变量。

```
struct f2fs_fsck {
    struct thread_ctx *tctx_array; // list of thread contexts
    pthread_mutex_t tctx_array_lock; // lock for iterating over tctx array
    pthread_key_t tctx_key; // thread key for getting threads specific tctx
    uint32_t thread_number; // how many thread do you want to make? default is 1

    struct thpool* threadpool; //添加线程池
    pthread_mutex_t fsck_lock;
    pthread_mutex_t qf_szchk_type_lock;
    pthread_mutex_t qf_last_blkofs_lock;

    struct f2fs_sb_info sbi;

    struct chk_result {
        u64 checked_node_cnt;
        u64 valid_blk_cnt; // ** main area中有有效的block数目: node + data
        u32 valid_node_cnt; // ** main area中有有效的node block数目
        u32 valid_inode_cnt; // ** main area中有有效的inode block数目 (node block的子集)
        u32 multi_hard_link_files; // 有多个 hard_link 文件的个数
    } chk;

    struct hard_link_node *hard_link_list_head;
    pthread_mutex_t hard_link_list_head_lock;
    char *main_area_bitmap; // 记录在遍历过程中所访问到的所有的block (和sit_bitmap对标), 最终两者要相同才对
    pthread_mutex_t main_area_bitmap_lock;
    char *nat_area_bitmap; // 指向 nat_entry位图
    pthread_mutex_t nat_area_bitmap_lock;
    u32 dentry_depth;
    struct f2fs_dentry *dentry;
    struct f2fs_dentry *dentry_end;
};
```

图 4-2 f2fs_fsck 核心变量

表 4-2 显示了各变量的含义。

表 4-2 f2fs_fsck 核心变量及描述

变量名	数据类型	变量描述
tctx_array	thread_ctx	保存所有线程上下文
tctx_array_lock	pthread_mutex_t	tctx_array 对应的锁
tctx_key	pthread_key_t	用于获取线程上下文的 key
thread_number	uint32_t	线程池中线程数量
threadpool	thpool_*	线程池
qf_szchk_type_lock	pthread_mutex_t	qf_szchk_type 对应的锁
qf_last_blkofs_lock	pthread_mutex_t	qf_last_blkofs 对应的锁
sbi	f2fs_sb_info	super block 的信息
chk	chk_result	全局检查结果
hard_link_list_head	hard_link_node *	硬链接链表
hard_link_list_head_lock	pthread_mutex_t	hard_link_list_head 对应的锁

续表 4-2 f2fs_fsck 核心变量及描述

变量名	数据类型	变量描述
main_area_bitmap	char *	main area 的位图
main_area_bitmap_lock	pthread_mutex_t	main area 位图对应的锁
nat_area_bitmap	char *	nat area 的位图
nat_area_bitmap_lock	pthread_mutex_t	nat area 位图的锁
dentry_depth	u32	目录项的深度
dentry	f2fs_dentry *	目录项链表
dentry_end	f2fs_dentry *	目录项链表尾

● **job**

job代表了一个任务。类型为0时为普通任务，即为对从某一Node开始递归对整个Node树进行检查。其核心变量如图4-3所示。

```
/* Job */
typedef struct job{
    struct job* prev;                /* pointer to previous job */
    void (*function)(void* arg);    /* function pointer */
    void* arg;                      /* function's argument */

    //////////////////////////////////////
    int type; // 0=regular, 1=transfer protocol, -1=giveup protocol
    struct thpool_* new_tpool; //new thread pool if there
    //////////////////////////////////////
} job;
```

图 4-3 job 核心变量

表 4-3 显示了 job 各核心变量的含义。

表 4-3 job 核心变量

变量名	数据类型	变量描述
PRE	STRUCT JOB*	指向前一个任务的指针；
VOID (*FUNCTION) (VOID* ARG)	FUNCTION	任务对应要调用的函数；
ARG	VOID*	函数参数；
TYPE	INT	任务类型；
NEW_TPOOL	STRUCT THPOOL_*	新的线程池，当任务为迁移线程时，将线程迁移到该目标线程池；

● **jobqueue**

jobqueue 为任务队列，每个线程池会有一个任务队列，线程池里的线程会不断从该任务队列取出任务执行。其核心变量如图 4-4 所示。

```
/* Job queue */
typedef struct jobqueue{
    pthread_mutex_t rwmutex;           /* used for queue r/w access */
    job *front;                        /* pointer to front of queue */
    job *rear;                         /* pointer to rear of queue */
    bsem *has_jobs;                   /* flag as binary semaphore */
    int len;                          /* number of jobs in queue */
} jobqueue;
```

图 4-4 jobqueue 核心变量

表 4-4 显示了 jobqueue 各核心变量的含义。

表 4-4 jobqueue 核心变量

变量名	数据类型	变量描述
RWMutex	PTHREAD_MUTEX_T	访问队列的锁；
FRONT	JOB *	队头的任务；
REAR	JOB *	队尾的任务；
LEN	INT	队列中的任务个数；

● **thread**

thread 是对真正的线程进行了封装，记录了额外信息，如线程对应的线程池，是否借出，原来的线程池等。其核心变量如图 4-5 所示。

```
/* Thread */
typedef struct thread{
    int id; /* friendly id */
    pthread_t pthread; /* pointer to actual thread */
    struct thpool_* thpool_p; /* access to thpool */

    //////////////////////////////////////
    int borrowed; // signifies if this thread is borrowed
    struct thpool_* orig_tpool; // originating threadpool
    //////////////////////////////////////
} thread;
```

图 4-5 thread 核心变量

表 4-5 显示了 thread 各核心变量的含义。

表 4-5 thread 核心变量

变量名	数据类型	变量描述
ID	INT	线程ID;
PTHREAD	PTHREAD_T	指向真正的线程;
THPOOL_P	STRUCT THPOOL_*	线程所属的线程池;
BORROWED	INT	线程是否借出;
ORIG_TPOOL	STRUCT THPOOL_*	线程原本的线程池

4.2 关键函数实现

● get_tctx

① 函数原型

```
struct thread_ctx* get_tctx(struct f2fs_sb_info *sbi)
```

② 函数功能

获取当前线程的上下文

③ 参数说明

变量名	数据类型	变量描述
sbi	struct f2fs_sb_info *	F2FS超级块信息

④ 返回值说明

返回当前线程的上下文

⑤ 函数流程

通过线程上下文的 key 获取当前线程上下文。若是第一次获取，还会将该上下文记录在全局上下文数组中。其流程图如图 4.6 所示。

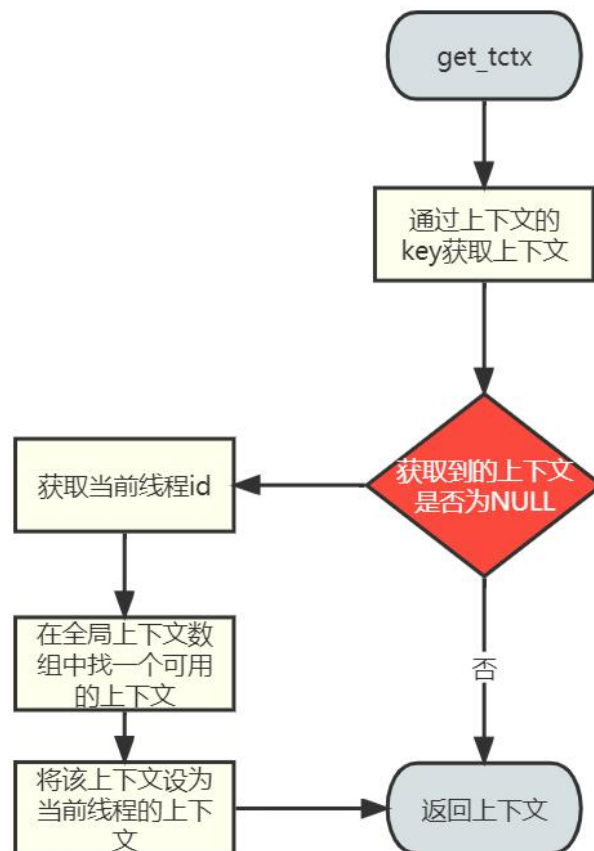


图 4-6 get_tctx 流程图

● **thread_do**

① 函数原型

`static void* thread_do(struct thread* thread_p)`

② 函数功能

这是线程池里的线程被创建后一直执行的函数。

③ 参数说明

变量名	数据类型	变量描述
thread_p	struct thread *	线程

④ 返回值说明

无。

⑤ 函数流程

主要流程为在死循环里不断获取当前线程池里任务进行执行。如图 4.7 所示。

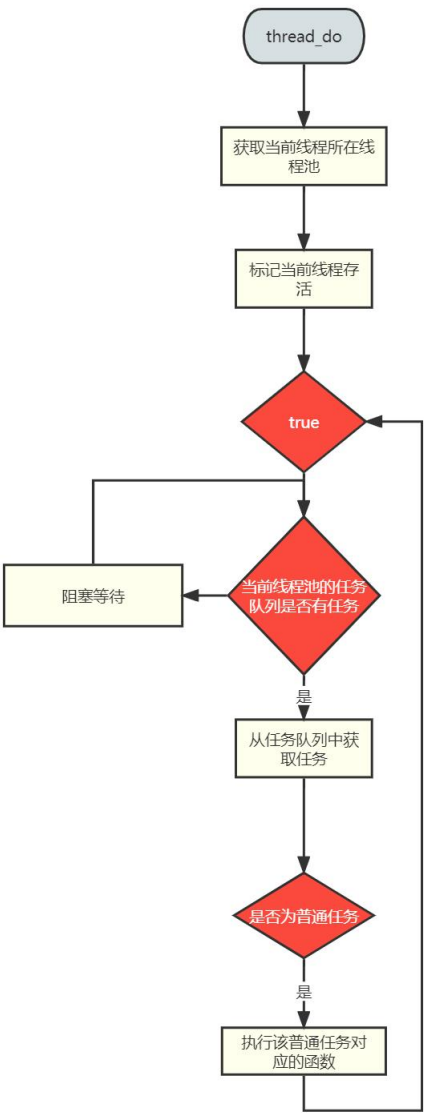


图 4-7 thread_do 流程图

● scheduler_thread

① 函数原型

```
void scheduler_thread(struct f2fs_sb_info *sbi)
```

② 函数功能

调度器线程，周期运行，动态调整线程数目。

③ 参数说明

变量名	数据类型	变量描述
sbi	struct f2fs_sb_info *	f2fs超级块信息

④ 返回值说明

无。

⑤ 函数流程

主要流程为在死循环里每隔一段时间调用 schedule 函数。如图 4.8 所示。

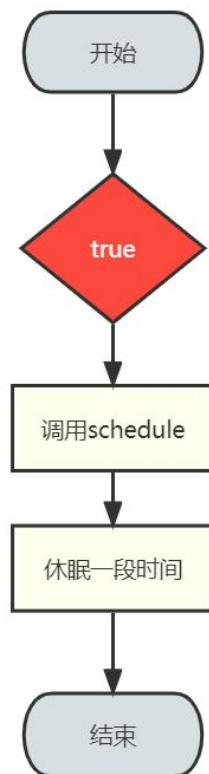


图 4-8 scheduler_thread 流程图

● schedule

① 函数原型

```
void schedule(struct f2fs_sb_info *sbi)
```

② 函数功能

根据当前资源使用情况，动态调整线程个数。

③ 参数说明

变量名	数据类型	变量描述
sbi	struct f2fs_sb_info *	f2fs超级块信息

④ 返回值说明

无。

⑤ 函数流程

主要流程为根据 top 命令获取系统整体CPU利用率和当前进程的CPU利用率，进一步算出线程数的预算，并向线程池添加“调整线程数”的任务。如图 4.9 所示。

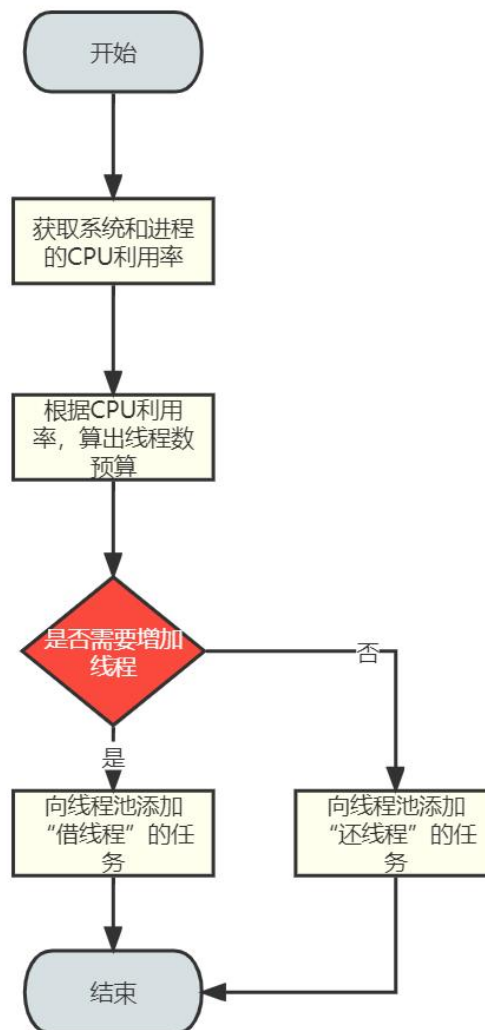


图 4-9 schedule 流程图

● adjust_core_count

① 函数原型

```
int adjust_core_count(int cores, int prev_cores, float total_util, float process_util)
```

② 函数功能

根据当前资源使用情况，算出应该使用的线程数。

③ 参数说明

变量名	数据类型	变量描述
sbi	struct f2fs_sb_info *	超级块信息
cores	int	应调整到的线程数
prev_cores	int	之前的线程数
float	total_util	系统总CPU利用率
float	process_util	wFSCK进程的CPU利用率

④ 返回值说明

返回应调整到的线程数。

⑤ 函数流程

主要流程为根据系统 CPU 利用率高低和进程 CPU 利用率高低，决定增加或减少或不变动线程数。如图 4.10 所示。

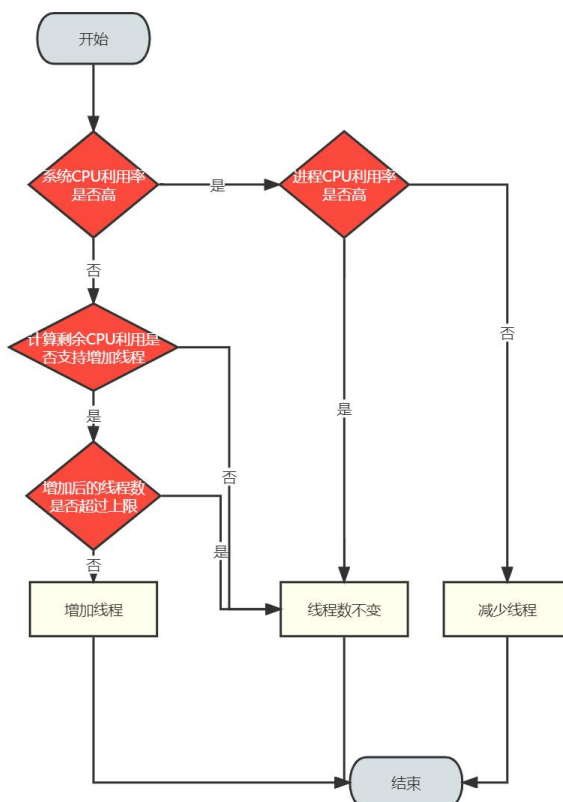


图 4-10 adjust_core_count 流程图

● borrow_threads

① 函数原型

```
int borrow_threads(thpool_* thpool1, thpool_* thpool2, int n)
```

② 函数功能

从线程池 thpool1 中借 n 个线程给线程池 thpool2，执行检查任务。

③ 参数说明

变量名	数据类型	变量描述
thpool1	thpool_*	线程池1
thpool2	thpool_*	线程池2
n	int	要借的线程数

④ 返回值说明

成功返回 0，失败返回-1。

⑤ 函数流程

主要流程为向要借出线程的线程池 1 中添加一个任务，线程池 1 中的线程在拿到这个任务执行时，会把自己切换到线程池 2 中去。如图 4.11 所示。



图 4-11 borrow_threads 流程图

● fsck_increase_valid_inode_cnt_atomicly

① 函数原型

```
void fsck_increase_valid_inode_cnt_atomicly(struct f2fs_sb_info *sbi)
```

② 函数功能

取出当前线程上下文的某变量的值将其加一，

③ 参数说明

变量名	数据类型	变量描述
sbi	struct f2fs_sb_info *	f2fs超级块信息

④ 返回值说明

无。

⑤ 函数流程

主要流程为取出当前线程上下文的某变量的值将其加一。类似的函数还有很多，这类函数封装了对线程私有数据的操作，用于方便地替换原始直接访问共享数据结构的代码。

● post_work_inode

① 函数原型

```
void post_work_inode(int ret, void *args)
```

② 函数功能

fsck_chk_inode 函数里对间接 block 进行的检查完成后，对返回值进行处理。

③ 参数说明

变量名	数据类型	变量描述
ret	int	检查的返回值，0代表成功，-EINVAL代表失败。
args	void *	处理返回值时需要的上下文，在执行该检查任务前已保存好。

④ 返回值说明

无。

⑤ 函数流程

主要流程如图 4.12 所示。

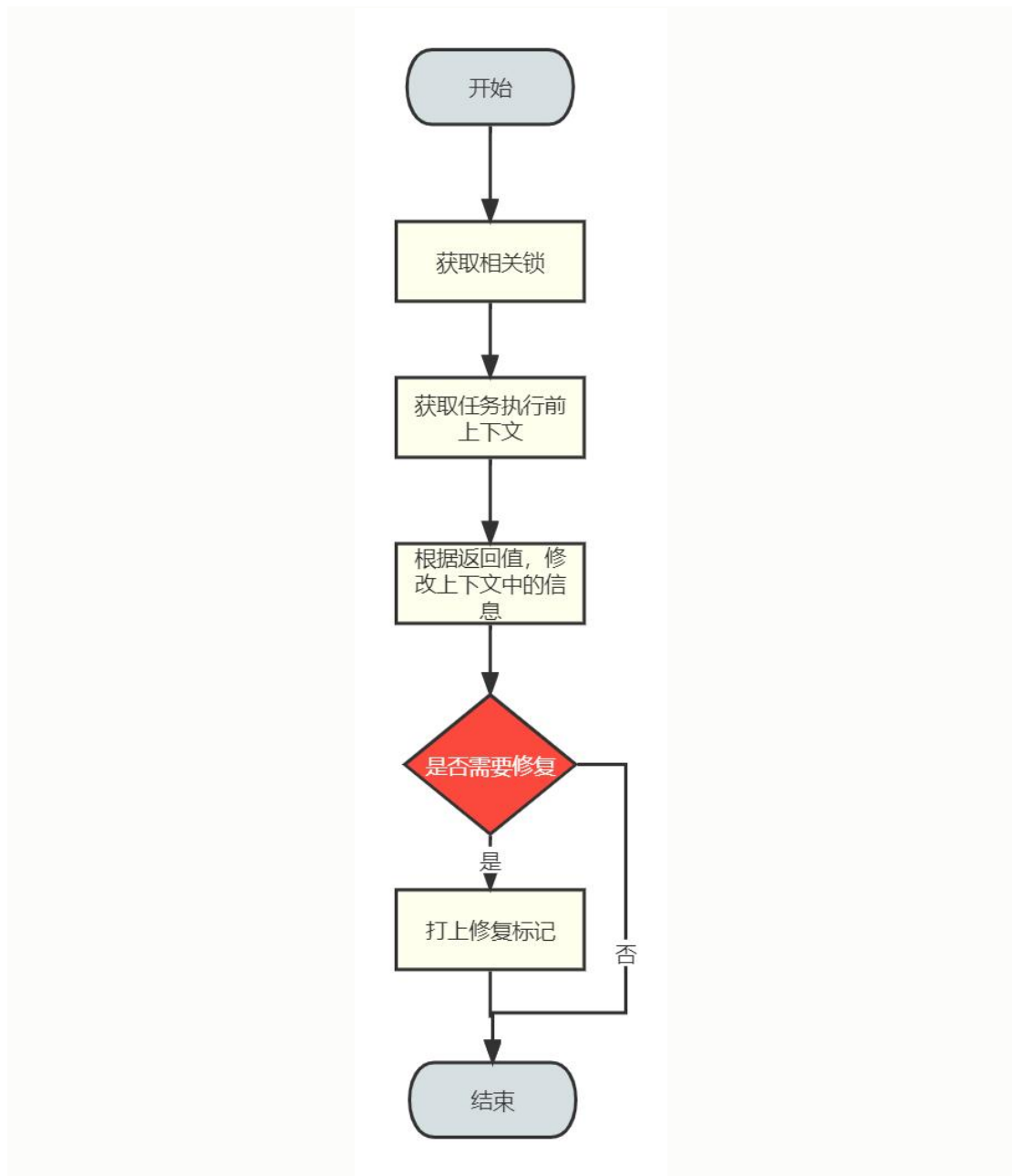


图 4-12 `post_work_inode` 流程图

5. 系统测试

5.1 测试准备

对 wfscck 的测试分为功能测试与性能测试两个方面。功能测试主要测试 wfscck 能否正确检测并修复损坏的文件系统。性能测试主要通过测试 wfscck 在不同场景下的运行时间评估其性能，并将其与原始 fsck 工具对比。

文件系统中的文件和目录数量可能会变化并取决于应用程序。论文^[32]对 RocksDB(键值存储)、视频服务器、Web 服务器和邮件服务器等工作负载进行分析，发现文件数量占主导地位(99%的文件占 1%的目录)。测试中我们同样按此配置设置文件系统，即一个目录下包含 100 个文件。

实验在两台机器上进行，一台物理机，一台虚拟机。选择了一台虚拟机是因为运用虚拟机的场景在云计算时代越来越普遍，在虚拟机上的运行结果具有很高的参考价值。另一方面由于我们手上没有空闲 Nand Flash 存储介质，因此在回环设备上模拟了 Nand Flash 硬盘，回环设备运行在 ext4 的文件系统上，机器的具体信息如表 5-1 所示：

表 5-1 机器配置

Target	System	Storage Device
物理机*16 核	CPU: Intel(R) Core(TM) i9-9900K CPU @ 3.60GHz Memory: 128G OS: Ubuntu 20.04.6 LTS	Samsung SSD 860 4B6Q 500G 在此硬盘上创建了一个大小为 450G 的分区，文件系统为 ext4。 在此文件系统中挂载了大小为 1/10/50G 的回环设备
VirtualBox 虚拟机*4 核	CPU: Intel(R) Core(TM) i5-10500 CPU @ 3.10GHz Memory: 8G OS: Ubuntu 22.04.1 LTS	vdi 硬盘 150G（在 Seagate HDD 1T 上创建）。硬盘上创建了一个 130G 的分区，文件系统为 ext4。在次文件系统中挂载了大小为 1/2/5G 的回环设备

在性能测试中，为了在文件系统中的生成测试文件，我们使用 fs_mark 基准测试工具进行填充。针对 1G 的文件系统，我们填充 512000*1 个大小为 1024 字节的文件，这将占用 500M 的空间，这符合一般文件系统的使用情况。以此类推，针对 2G 的文件系统，我们将填充 512000*2 个大小为 1024 字节的文件。实验表明，我们的 wFSCCK 在开启大于一个工作线程时，检测性能相比原来的 fsck. f2fs

将普遍提升超过 25%。

在功能测试中，为了验证我们加速后的 fsck.f2fs 与原 fsck.f2fs 的行为一致，我们对文件系统的某些区域进行破坏，然后使用 fsck.f2fs 来找出被破坏的部分并进行纠正。来验证 wFSCK 在加速的同时并未破坏原功能完整性。

5.2 测试方法与测试结果

5.2.1 功能测试

功能测试的测试流程如下：

- (1) 生成指定大小的文件系统镜像并运行 fs_mark 程序为文件系统镜像填充文件；
- (2) 运行 destroy.f2fs 程序将文件系统镜像指定区域的数据用随机数据替换；
- (3) 将被破坏后的镜像拷贝两份；
- (4) 运行原始 fsck 程序检验文件系统损坏情况；
- (5) 运行 wfsck 程序检测并修复损坏的文件系统镜像；
- (6) 通过 4、5 得到的日志比较两者行为是否一致；
- (7) 若文件系统检测到损坏，回到第 4 步，分别通过各种的修复程序进行修复，然后再次对日志进行对比；否则，退出程序。

图 5-1 给出了一次功能测试的流程图。注意第 3 步，我们将镜像拷贝了两份，这样当原 fsck 与 wfsck 行为不一致时，我们可以在修复 wfsck 后，通过留存的镜像，再次进行测试，只有当两者行为一致，留存的镜像才能被安全的删除。

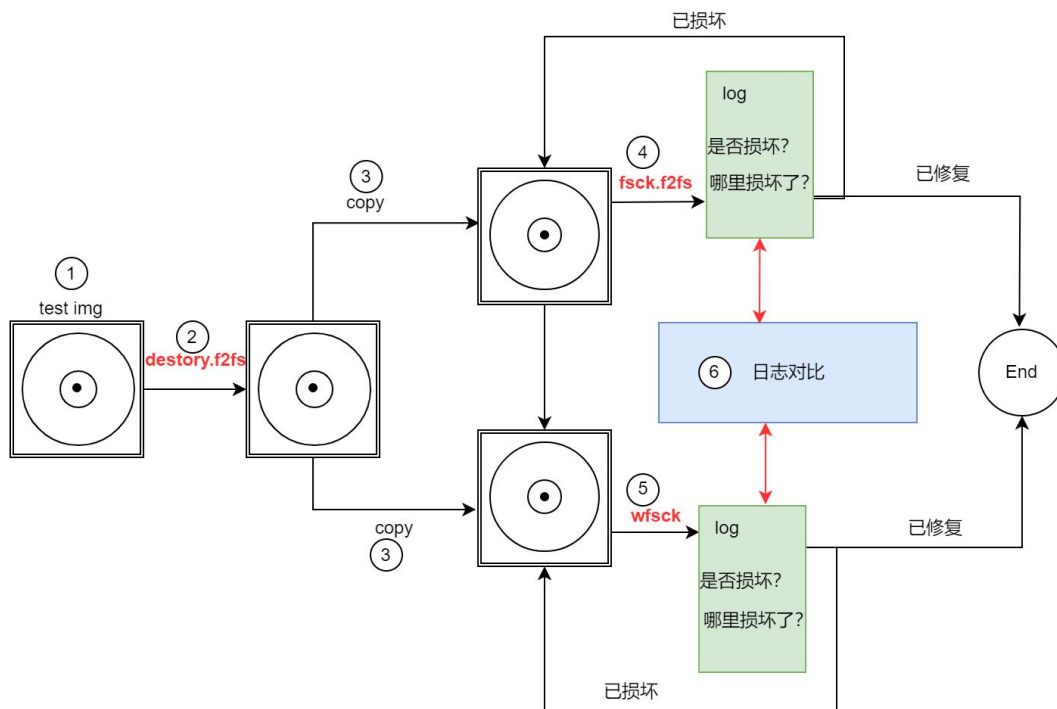


图 5-1 功能测试流程

测试中我们生成了一大小为 1GB 的文件系统镜像，向该镜像的 sit 或 nat 或 ssa 区域随机写入数据，破坏该镜像。该区域包含 100 字节的垃圾数据。使用原始 fsck 工具与 wfsck 分别进行检查和修复，然后通过 diff 工具对过滤后的日志进行对比，若两者的日志一样，我们则认为两者的行为是一致的。目前，测试分了 20 轮，都未检测到 fsck 和 wfsck 之间行为的不一致，我们暂且认为 wfsck 和原 fsck 是一致的，后面我们会进行更为详尽的测试，比如破坏 Main Area 区域中 node 节点的数据、使用更精确的破坏工具破坏指定区域指定字段等。

5.2.2 性能测试

性能测试的测试流程如下：

- 1) 生成指定大小的文件系统镜像用于测试；
- 2) 运行 fs_mark 程序为文件系统镜像按指定配置填充文件；
- 3) 通过 time 程序捕获 fsck 程序的运行时间。

情景 1：物理机

我们在物理机上创建了大小为 1GB，10GB，50GB 的文件系统镜像，分别测试了原始的 fsck 程序，以及使用不同线程数的 wfsck 程序的运行时间，并取 3 次运行的平均时间作为最终运行时间。结果统计如图 5-2 所示。图中纵坐标为程序运行时间，横坐标为 wfsck 使用的线程数，红色的虚线为原始 fsck 程序的运行时间，作为 baseline 进行比较。

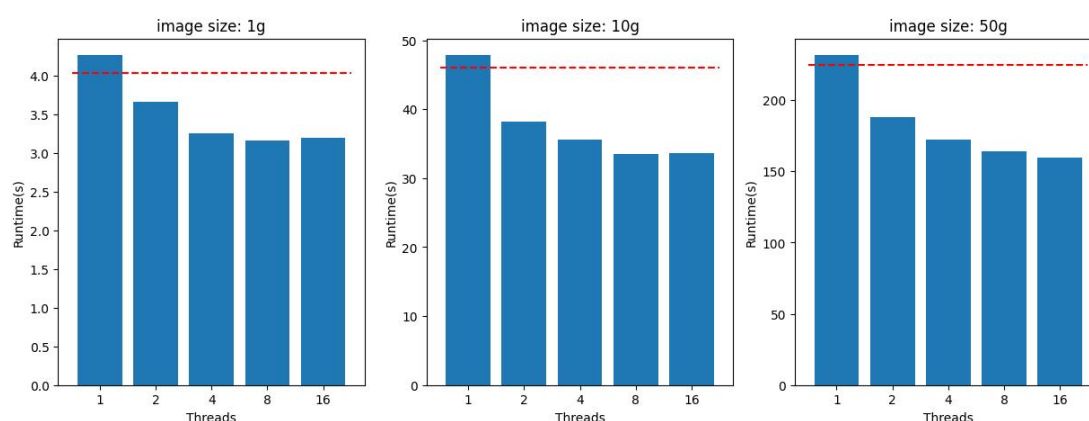


图 5-2 wfsck 在物理机上运行时间对比

可以看出，wfsck 使用多线程并行时，运行速度超过原始 fsck 程序。随着线程数的增加，wfsck 的运行速度也同样会加快，且在文件系统规模增大时，依然能保持这种趋势。说明 wfsck 成功利用多线程并行实现加速。其中线程数为 1 时，即不使用并行时，wfsck 的运行速度略有下降，这是由于并行操作在程序中引入了额外的开销，使得性能略有下降。总体而言，在使用 8 线程并行时，wfsck

程序可以加速约 25%，最大可加速约 29%。

现根据指标进一步分析 wfsck。在大小为 50gb 的文件系统镜像上测试得到的指标如下表。其中 Total time 为程序运行总时间，在单核机器上等于用户态运行时间（User time），内核态运行时间（System time），与 IO 时间之和，即

$$Total\ time = system\ time + user\ time + IO\ time$$

而在多核机器上，我们有：

$$Total\ time * core = system\ time + user\ time + IO\ time$$

CPU usage 为程序运行期间对 CPU 计算资源的占用情况。

表 5-2 物理机性能测试结果(image size: 50GB)

指标	fsck	wfsck -1thread	wfsck -2thread	wfsck -4thread	wfsck -8thread	wfsck -16thread
Total time (s)	224	232	188	172	163	159
Total time * 16 cores	3584	3712	3008	2752	2608	2544
User time (s)	11.39	18.62	23.24	32.47	35.45	33.98
System time (s)	36.78	38.50	49.10	65.48	69.99	75.45
CPU usage (%)	21	24	38	56	64	68

可以看出，随着 wfsck 使用线程数的增加，程序的用户态运行时间与内核态运行时间逐渐增加，而总体时间却逐渐下降。对此现象的解释是，wfsck 通过多线程并行处理数据，提高了 IO 的利用率，降低了 IO 的等待时间，从而加速了程序运行。但与此同时，wfsck 对于 CPU 的利用率非常低，在 16 线程的 CPU 上，使用率没超过 100%，因此，系统的瓶颈在于 I/O 设备上，我们接下来的工作，将会对每个线程设立独立 I/O 缓存，以此提高 I/O 效率，达到更高的加速比。

情景 2：虚拟机

我们在虚拟机上创建了大小为 1GB，2GB，5GB 的文件系统镜像，进行与情景 1 相同的测试，结果统计如图 5-5 所示。图中纵坐标为程序运行时间，横坐标为 wfsck 使用的线程数，红色的虚线为原始 fsck 程序的运行时间，作为 baseline 进行比较。

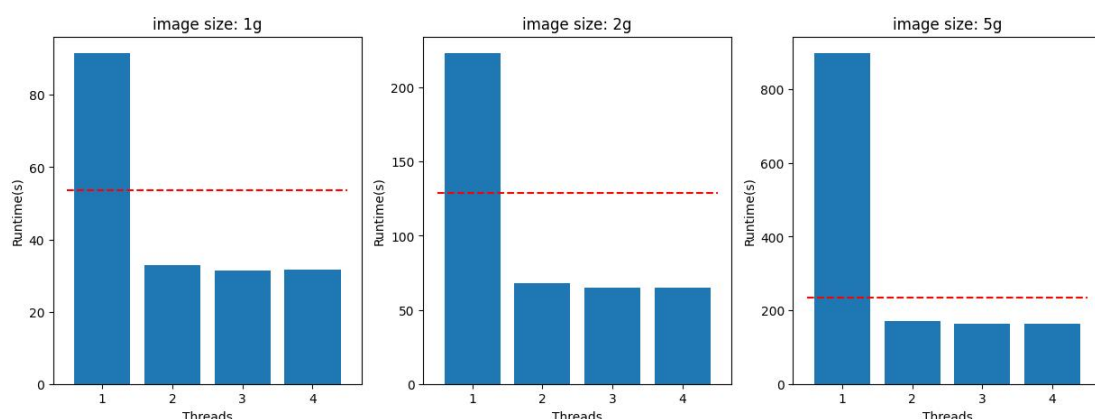


图 5-3 wfsck 在虚拟机上运行时间对比

可以看出，与物理机类似，wfsck 使用多线程并行加速时，运行速度超过原始 fsck 程序。随着文件系统大小的增加，这种趋势同样成立。可是随着线程数的增加，wfsck 的运行速度增长缓慢，我们猜测在此小硬盘上，两个线程已经能够达到最大的加速比，超过 25%，而增加额外的线程受限于 I/O 的速度和系统基础开销，而不再有明显增益。值得一提的是，开启一个线程的 wfsck 开销要远远高于原 fsck，这让人非常意外，我们初步认定是由于这是在虚拟机环境的缘故，受限于虚拟机监控程序和 Host 操作系统，清缓存不见得总是能清理得非常干净，而 wfsck_1 总是生成磁盘后进行测试的第一个程序，将磁盘加载到内存耗费了大量时间，而后续由于缓存缘故，测试时间就减慢了许多。后续，我们将使用不同的线程数作为第一个测试程序来验证这种影响，并且提供缓解措施来避免这些错误数据导致的误解。但是总体而言，wfsck 程序在使用 2 个工作线程及以上时，均可加速超过 25%；而在 2G 硬盘的情况下，加速超过了 50%。

现根据指标进一步分析 wfsck。在大小为 5gb 的文件系统镜像上测试得到的指标如下表。

表 5-3 虚拟机性能测试结果(image size:5GB)

指标	fsck	wfsck -1thread	wfsck -2thread	wfsck -3thread	wfsck -4thread
Total time (s)	234	857	169	162	162
Total time * 4 cores	936	3428	676	648	648
User time (s)	2.42	4.26	2.81	2.39	2.40
System time (s)	224.94	220.22	327.26	465.97	479.71
CPU usage (%)	96	25	194	288	295

可以看到 wfsck 开启 2 个线程即以上的时候，CPU 利用率非常高，证明我们充分地利用了 CPU 资源，I/O 的开销相比于上述物理机的情况少了许多，这也有可能是上述介绍的宿主机缓存的影响。但是，实验仍表明我们充分利用了 CPU

并行性，对文件系统检测和修复显著加速的效果。

6. 总结与展望

6.1 工作总结

在 wFSCK 项目中，我们完成了：

- 针对 f2fs 复杂的文件系统结构，将 pFSCK 的并发思想移植到 fsck.f2fs，将 fsck.f2fs 对 Node 树的检查拆分，并行执行检查。
- 对 fsck.f2fs 的检查任务进行划分，同时不影响 C/R 的正确性。
- 保存任务执行前的上下文，对任务的返回值进行处理，保证了 C/R 一致性。
- 对共享文件系统结构访问进行控制，重新设计数据结构，解决同步访问共享结构的瓶颈。
- 智能感知系统资源使用情况，动态调整工作线程个数。

待完成：

- 充分利用存储设备 I/O，使 I/O 管理适用并发环境。
- 智能资源感知的动态调整后台任务线程数量的管理框架，例如动态调整后台 GC 任务线程，来减少对其他程序的影响。
- fsck.f2fs 检查过程中收集信息，优化后续读写等情况。fsck.f2fs 在检查文件系统的过程中，会遍历所有的文件元数据信息，这信息可以被记录下来，用于优化后续对文件系统的读写。

下表显示了对应赛题完成情况：

目标编号	基本完成情况	额外说明
1	基本完成（≈80%）	⑤ 可节省 25%到 50%左右的运行时间； ⑥ 可动态调整线程个数 ⑦ 线程的 I/O cache 管理待实现 ⑧ 可能存在未发现的 BUG
2	初步完成（≈10%）	完成调研，计划将当前的动态调整线程逻辑移植到操作系统的许多后台任务中，使其更智能。

3	初步完成 (≈10%)	完成调研, 计划通过记录 <code>fsck.f2fs</code> 检查获得的额外信息, 如哪些文件系统的空闲区域有哪些, 优化后续的 <code>write</code> 调用, 使得 <code>write</code> 更智能。
---	-------------	---

6.2 创新点

我们总结 wFSCK 相较于 `fsck.f2fs` 的主要创新点如下:

已实现:

- 参考 pFSCK 引用了并发机制, 加速 `fsck.f2fs` 的执行过程, 可节省 25%-50% 的时间, 并能智能感知系统资源使用情况, 动态调整线程个数, 减少对系统其他程序的影响。

待实现:

- 充分利用存储设备 I/O, 进一步加速 `fsck.f2fs` 的执行过程。
- 实现智能资源感知的动态调整后台任务线程数量的管理框架。
- 在 `fsck.f2fs` 检查过程中收集信息, 优化后续读写等情况。

6.3 未来展望

虽然 wFSCK 做了很多工作, 但还有很多的 Future Works, 例如:

- 如何划分任务, 使其任务量更均衡, 进一步加速 `fsck.f2fs`。
- 如何调度任务的执行, 当前任务调度策略是 FIFO 先入先出, 可以引入其他策略, 进一步加速 `fsck.f2fs`。
- 如何充分利用检查过程中收集到的信息, 结合 `f2fs` 的特性, 做更多的性能优化。

参考文献

- [1] Marshall Kirk McKusick, Willian N Joy, Samuel J Lefffler, and Robert S Fabry. Fsk- the unix† ffile system check program. *Unix System Manager's Manual-4.3 BSD Virtual VAX-11 Version*, 1986.
- [2] Haryadi S. Gunawi, Riza O. Suminto, Russell Sears, Casey Golliher, Swaminathan Sundararaman, Xing Lin, Tim Emami, Weiguang Sheng, Nematollah Bidokhti, Caitie McCaffrey, Gary Grider, Parks M. Fields, Kevin Harms, Robert B. Ross, Andree Jacobson, Robert Ricci, Kirk Webb, Peter Alvaro, H. Birali Runesha, Mingzhe Hao, and Huaicheng Li. Fail-slow at scale: Evidence of hardware performance faults in large production systems. In 16th USENIX Conference on File and Storage echnologies (FAST 18), pages 1–14, Oakland, CA, 2018. USENIX Association.
- [3] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating ffile system reliability on solid state drives. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 783–798, Renton, WA, July 2019. USENIX Association.
- [4] William (Bill) E. Allcock. Parallel File Systems at HPC Centers: Usage, Experiences, and Recommendations. <https://www.nersc.gov/assets/Uploads/W01-DataIntensiveComputingPanel.pdf>.
- [5] HPC-Users Mailing List. Outages in HPC Systems. <https://maillists.uci.edu/pipermail/hpc-users/2019-December/000095.html>
- [6] e2fsck: fsck for ext4. <https://linux.die.net/man/8/e2fsck>.
- [7] Val Henson, Zach Brown, and Arjan van de Ven. Reducing fsck time for ext2 ffile systems. 04 2019.
- [8] M. Lu, T. Chiueh, and S. Lin. An incremental ffile system consistency checker for block-level cdp systems. In *2008 Symposium on Reliable Distributed Systems*, pages 157–162, Oct 2008.
- [9] Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci Dusseau, and Marshall Kirk Mckusick. Ffsck: The fast ffile-system checker. *Trans. Storage*, 10(1):2:1–2:28, January 2014.
- [10] Marshall K. McKusick. Improving the performance of fsck in freebsd. *ogin*., 38(2), 2013.
- [11] Marshall Kirk McKusick, Willian N Joy, Samuel J Lefffler, and Robert S Fabry. Fsk- the unix† ffile system check program. *Unix System Manager's Manual-4.3 BSD Virtual VAX-11 Version*, 1986.

- [12] Ramnathan Alagappan, Aishwarya Ganesan, Yuvraj Patel, Thanu malayan Sankaranarayana Pillai, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Correlated crash vulnerabilities. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI' 16, pages 151 – 167, Berkeley, CA, USA, 2016. USENIX Association.
- [13] Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. Sqck: A declarative file system checker. In Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation, OSDI' 08, pages 131 – 146, Berkeley, CA, USA, 2008. USENIX Association.
- [14] Daan Leijen, Wolfram Schulte, and Sebastian Burckhardt. The design of a task parallel library. OOPSLA ' 09, page 227 – 242, New York, NY, USA, 2009. Association for Computing Machinery.
- [15] Henry Qin, Qian Li, Jacqueline Speiser, Peter Kraft, and John Ousterhout. Arachne: Core-aware thread management. In Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation, OSDI' 18, pages 145 – 160, Berkeley, CA, USA, 2018. USENIX Association.
- [16] Intel-Micron Memory 3D XPoint. <http://intel.ly/1eICR0a>.
- [17] Changman Lee, Dongho Sim, Joo-Young Hwang, and Sangyeun Cho. F2FS: A New File System for Flash Storage. In Proceedings of the 13th USENIX Conference on File and Storage Technologies, FAST' 15, Santa Clara, CA, 2015.
- [18] Jiabin Ou, Jiwu Shu, and Youyou Lu. A high performance file system for non-volatile main memory. In Proceedings of the Eleventh European Conference on Computer Systems, pages 1 – 16, 2016.
- [19] Matthew Wilcox and Ross Zwisler. Linux DAX. <https://www.kernel.org/doc/Documentation/filesystems/dax.txt>.
- [20] Rohan Kadekodi, Se Kwon Lee, Sanidhya Kashyap, Taesoo Kim, Aasheesh Kolli, and Vijay Chidambaram. Splitfs: reducing software overhead in file systems for persistent memory. In Proceedings of the 27th ACM Symposium on Operating Systems Principles, pages 494 – 508, 2019.
- [21] Youngjin Kwon, Henrique Fingler, Tyler Hunt, Simon Peter, Emmett Witchel, and Thomas Anderson. Strata: A Cross Media File System. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, 2017.

- [22] Jian Xu and Steven Swanson. NOVA: A Log-structured File System for Hybrid Volatile/Non-volatile Main Memories. In Proceedings of the 14th Unix Conference on File and Storage Technologies, FAST' 16, 2016.
- [23] 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 Proceedings of the 27th ACM Symposium on Operating Systems Principles, pages 353 – 369, 2019.
- [24] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating file system reliability on solid state drives. In 2019 {USENIX} Annual Technical Conference ({USENIX}{ATC} 19), pages 783 – 798, 2019.
- [25] Lakshmi N Bairavasundaram, Andrea C Arpaci-Dusseau, Remzi H Arpaci-Dusseau, Garth R Goodson, and Bianca Schroeder. An analysis of data corruption in the storage stack. ACM Transactions on Storage (TOS), 4(3):8, 2008.
- [26] Lakshmi N. Bairavasundaram, Garth R. Goodson, Shankar Pasupathy, and Jiri Schindler. An analysis of latent sector errors in disk drives. Proceedings of the 2007 ACM SIGMETRICS international conference on Measurement and modeling of computer systems - SIGMETRICS 07, 2007.
- [27] Vijay Chidambaram, Thanumalayan Sankaranarayanan Pillai, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Optimistic crash consistency. In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pages 228 – 243. ACM, 2013.
- [28] Shehbaz Jaffer, Stathis Maneas, Andy Hwang, and Bianca Schroeder. Evaluating file system reliability on solid state drives. In 2019 USENIX Annual Technical Conference (USENIX ATC 19), pages 783 – 798, Renton, WA, July 2019. USENIX Association.
- [29] Mai Zheng, Joseph Tucek, Feng Qin, Mark Lillibridge, Bill W. Zhao, and Elizabeth S. Yang. Reliability analysis of ssds under power fault. ACM Trans. Comput. Syst., 34(4):10:1 – 10:28, November 2016.
- [30] Mtanski. [mtanski/xfsplogs github.com/mtanski/xfsplogs/preadv2/repair](https://github.com/mtanski/xfsplogs/tree/preadv2/repair). <https://github.com/mtanski/xfsplogs/tree/preadv2/repair>, Feb 2015.
- [31] StackExchange - Extremely long time for an ext4 fsck. <https://unix.stackexchange.com/questions/78785/extremely-long-time-for-an-ext4-fsck>, Mar 2013.

- [32] Domingo D, Kannan S. pFSCK: Accelerating File System Checking and Repair for Modern Storage[C]//Proceedings of the 19th USENIX Conference on File and Storage Technologies. 2021.