



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 是一款对 f2fs 文件系统进行并行检查和修复的工具。它以 fsck.f2fs 作为原型，并参考 pfsck 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。

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；
- ★ 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%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	未完成（≈0%）	
3	未完成（≈0%）	

目 录

1 概述

1.1 项目背景及意义

wfscck 是一款对 f2fs 文件系统进行检查和修复的工具。它将 pfscck 的思想移植到了 fsck.f2fs 上。而 pfscck 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。

首先，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。pfscck 则是在 e2fsck 的基础上引入了并发机制，加速其执行过程。本项目的 pfscck.f2fs 将在 fsck.f2fs 的基础上引入并发机制，来达到加速的目的。

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

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作用[41]。

先前的大量工作表明,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解决方案。先前的工作[21,27]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[11,21,27]、定期维护或强制安全升级[37]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不一致来工作。不一致可能出现在索引节点、数据和索引节点位图、链接或目录条目结构中。众所周知且广泛使用的工具,如 e2fsck (Ext4 的文件系统检查器)[2]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[24,38-41],有时持续数小时[37],甚至数周[11]。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[10,20,21,39]。虽然 wFSCK 借鉴了先前的任务并行研究[35,45],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,将每个 Node 的检查工作分解,重新设计数据结构,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)[31,49]。另一方面, 像英特尔的 DC Optane[5]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。大量先前的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[34]、nvme[44]的文件系统, 为 nvm 优化传统 Ext4 和 XFS 文件系统的开源努力[48], 以及其他研究工作[30,33,50]。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[9,28]。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, 但它们尚未被优化以提取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轻潜在的文件系统不一致性, 但它们是有限的, 因为它们不能修复由软件错误或过去由故障磁盘、位翻转、过热或相关崩溃等事件引起的错误[13-15,29,51]。在这些情况下, 使用流行的 C/R 工具, 如 fsck、e2fsck 和 xfs_repair[42], 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、目录条目一致性以及 inode 和块的一致引用计数, 来检测和修复损坏和错误。

C/R 使用:

在实际环境中, 文件系统 C/R 的频率和运行时有很大的不同。虽然缺乏记录良好的 C/R 最佳实践, 但在当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[7,21,27,29]、硬件或内核升级, 甚至在强制安全更新之后运行。不频繁的 C/R 和存储维护会将系统停机时间延长至 3 小时[37], 在极端情况下, 在 pb 级文件系统上, 停机时间长达数周[11]。

1.2.3 检查和修复工具 e2fsck

E2fsck 对 C/R 使用五次连续的遍历:第一次遍历(称为 pass -1)检查索引节点元数据的一致性;Pass-2 检查目录一致性;Pass-3 检查目录连通性;pass-4 检查参考计数;最后, 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 pfscck

pfscck 只要采用四种方式来加速,分别是①通过多核和数据并行性最大化潜在带宽。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系统影响。

①通过多核和数据并行性最大化潜在带宽。

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,pfscck 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,pfscck 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)划分为多个工作线程池,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的数据结构隔离,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而增加对共享结构的争用,并限制了 CPU 的可伸缩性。为了减少串行化开销,pfscck 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。pfscck 管理每个 pass 的线程池,使用分割和合并方法隔离 pass 间的共享结构,将检查与 inode 的实际认证区分开来,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(见\$3)。为了克服上述挑战,我们设计了一个 C/R 线程调度器,它可以动态地分配和迁移线程,以便在发现不同的文件系统对象时处理它们。

④通过资源利用感知减少系统影响。文件系统 C/R 可能与其他共享 cpu 的应用程序一起运行,同时在单独的磁盘上执行检查。考虑到 pfscck 的目标是利用可用的 cpu,它可能会影响其他协同运行的应用程序。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 pfscck 的影响,为 pfscck 的调度器配备了资源感知功能,以便动态识别在任何单个时间点要使用的内核数量,以最大限度地减少对其他共同运行的应用程序和 pfscck 性能的潜在影响。

1.2.5 f2fs 文件系统基本结构

1.2.5.1 为什么要 F2FS

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)。该设计一直专注于解决 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，还支持选择分配和清理算法。

1.2.5.2 F2FS 特性

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 的流浪树问题。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）

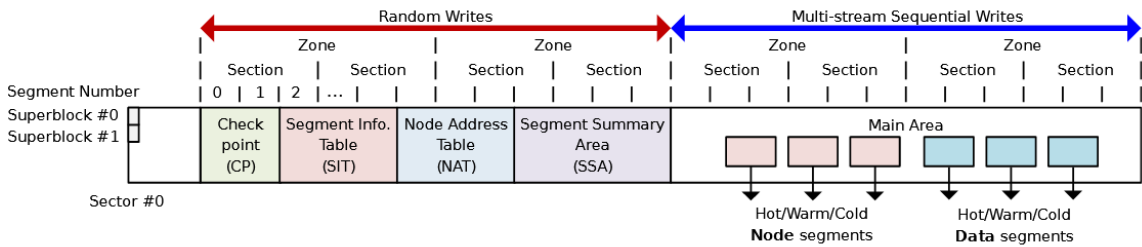
F2FS 基本上创建在仅追加日志记录（append-only logging）之上，将随机写入转换为顺序写入。然而，在高存储利用率下，它将日志记录策略更改为穿插日志记录（threaded logging），以避免长时间的写入延迟。实质上，穿插日志记录将新数据写入脏段中的可用空间，而不在前台清理它。此策略在现代闪存设备上效果很好，但在 HDD 上可能不起作用。

⑤ 通过前滚恢复机制（roll-forward recovery）加速

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。

1.2.5.3 F2FS 磁盘布局

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,如下图所示:



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。

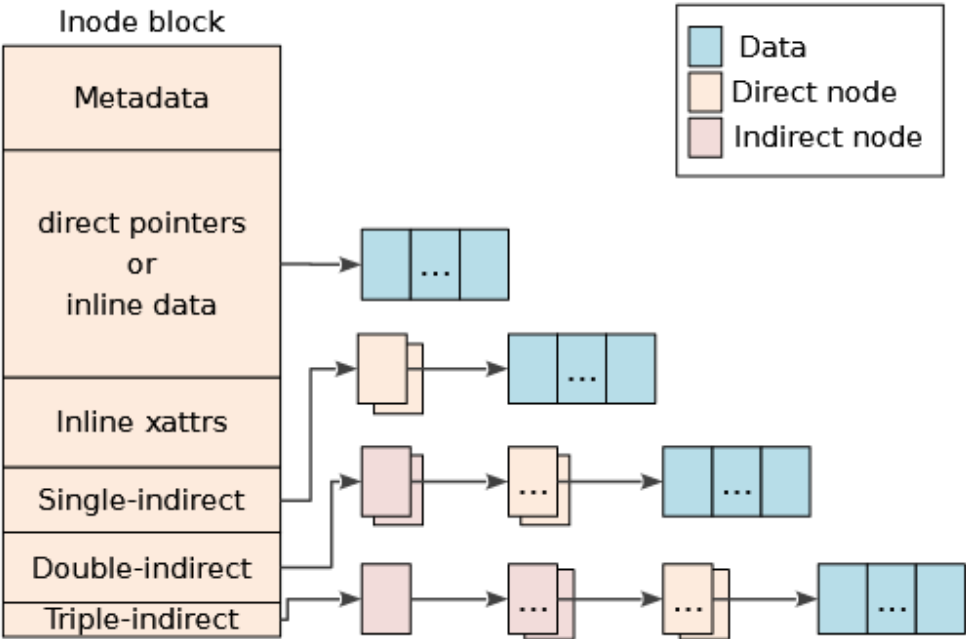
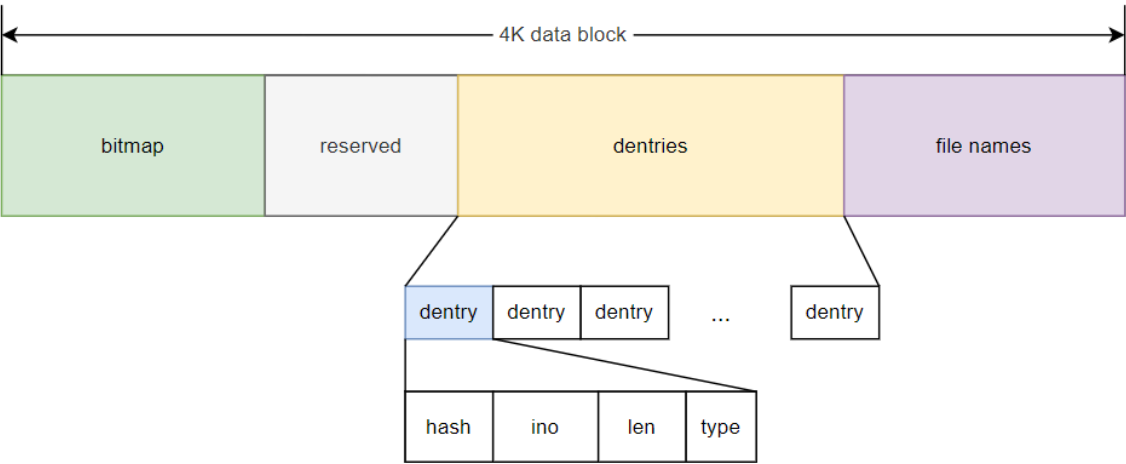


Figure 2: File structure of F2FS.

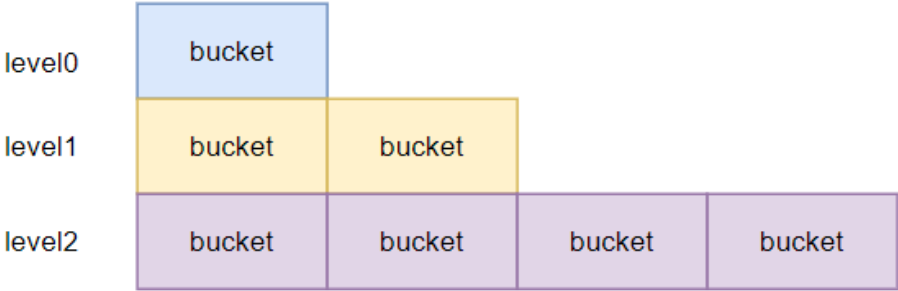
② 目录结构

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。下图展示了一个目录的数据块（目录条目块）在中硬盘中的布局。



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 $O(\log(\# \text{ of entries}))$ 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。下图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。



- attention:
- 1. one bucket may contains multiple blocks
 - 2. bucket number to scan in level #n = (hash value) % (# of buckets in level #n)

目录多级哈希表

③ 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 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。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。

Table 1: Separation of objects in multiple active segments.

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；

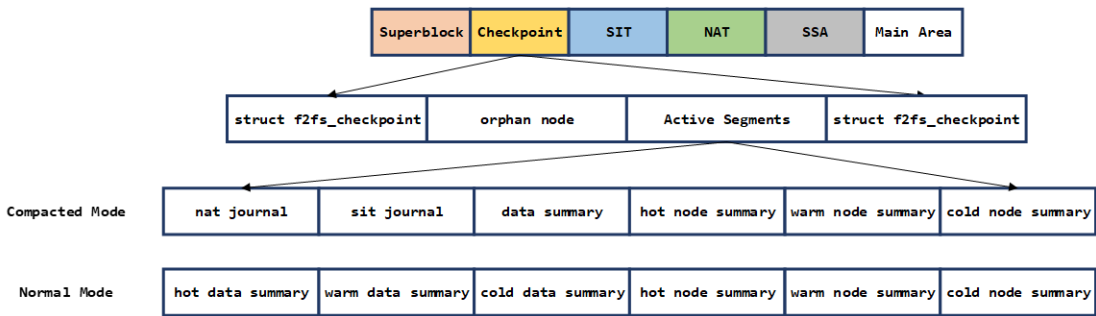
NAT 和 SIT 位图表示包含当前 pack 的 NAT 和 SIT 块的集合；

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 在硬盘中的表示如下图所示，可以看出 checkpoint 分为两种模式，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)。



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。

1.2.6 检查和修复工具 fsck.f2fs

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**

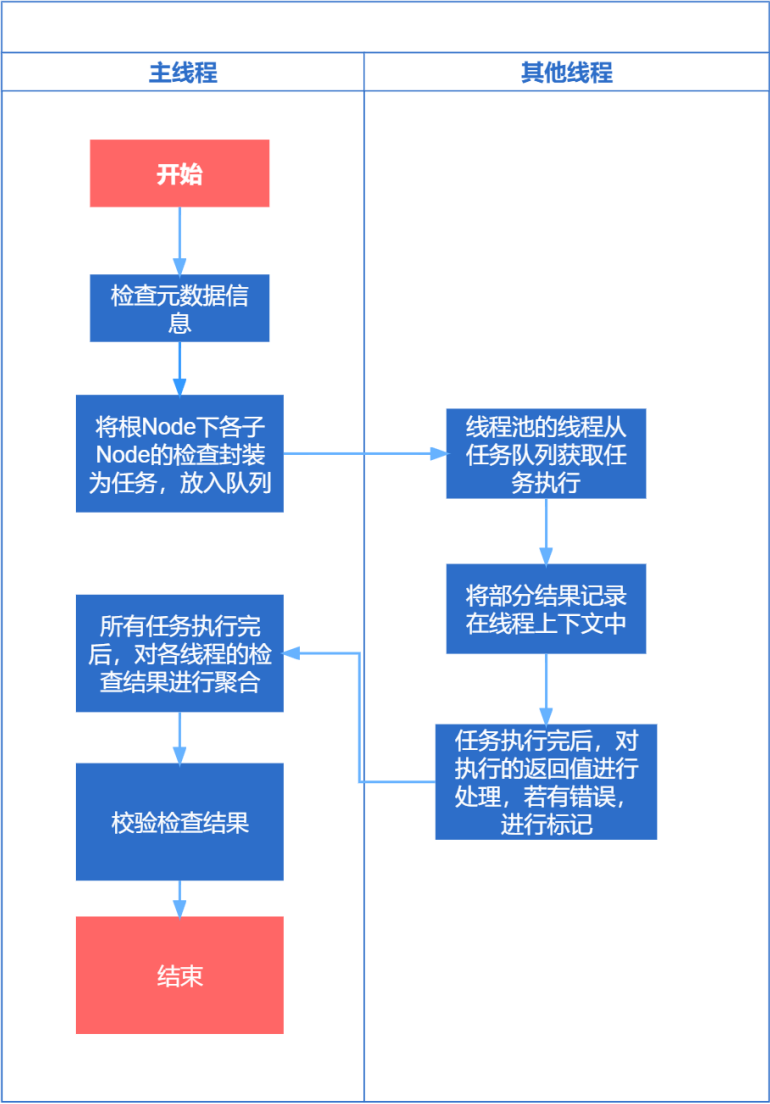
在 C/R 时，已经遍历了各文件的元数据信息，通过将这些信息记录下来，对后续读写等情况进行优化。

3.1 系统整体架构设计

系统整体运行流程如图所示。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。

全局变量安全和高效的访问。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。对全局变量的处理有两种，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全局变量的访问，在改为并发逻辑后，需要保证访问全局变量的原子性。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。



3.2 子模块设计

3.2.1 线程池设计

3.2.2 任务设计

3.2.3 线程上下文和锁设计

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
    } 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 4.1 显示了各变量的含义。

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
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4.2.1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 4.2.1 job 核心变量

表4.2.1显示了job各核心变量的含义。

表4.2.1 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4.2.1所示。

图 4.2.1 jobqueue 核心变量

表4.2.1显示了jobqueue各核心变量的含义。

表4.2.1 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4.2.1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 4.2.1 thread 核心变量

表4.2.1显示了thread各核心变量的含义。

表4.2.1 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
<i>sbi</i>	<i>f2fs_sb_info</i>	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 4.3 所示。

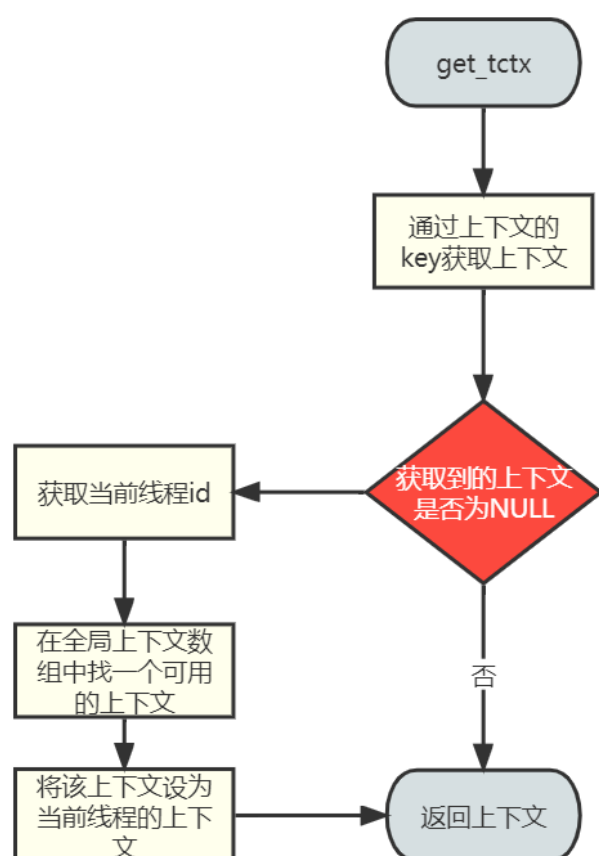


图 4.3 get_tctx 流程图

★ 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
<i>thread_p</i>	<i>struct thread*</i>	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 4.4 所示。

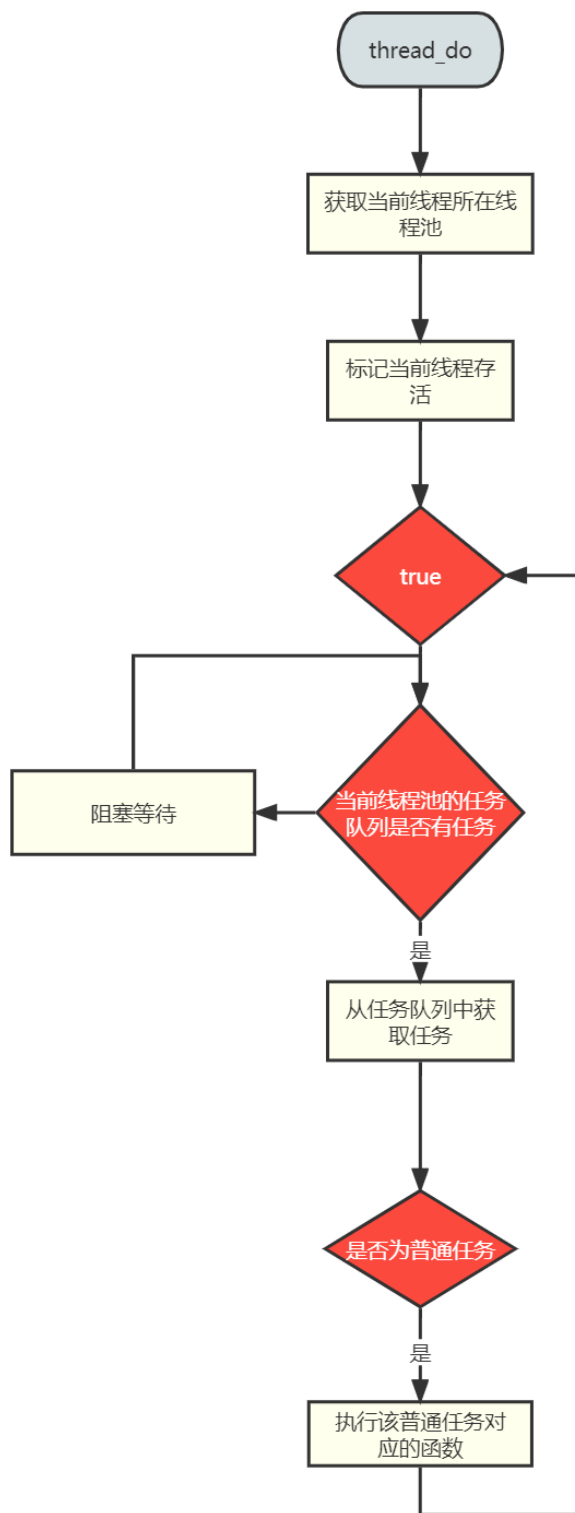


图 4.3 thread_do 流程图

5 系统测试

5.1 测试准备

5.2 测试方法

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 **pfscctof2fs** 能否正确检测并修复损坏的文件系统。为创建用于测试的损坏的文件系统，我们编写了损坏程序。性能测试主要通过测试各项评价指标评估 **pfscctof2fs** 的性能。评价指标参考原论文[1]，包括运行时间、存储带宽利用率等。

5.3 测试结果