第 43 章　日志结构文件系统

在 20 世纪 90 年代早期，由 John Ousterhout 教授和研究生 Mendel Rosenblum 领导的伯克利小组开发了一种新的文件系统，称为日志结构文件系统[RO91]。他们这样做的动机是基于以下观察。

- **内存大小不断增长**。随着内存越来越大，可以在内存中缓存更多数据。随着更多数据的缓存，磁盘流量将越来越多地由写入组成，因为读取将在缓存中进行处理。因此，文件系统性能很大程度上取决于写入性能。
- **随机 I/O 性能与顺序 I/O 性能之间存在巨大的差距，且不断扩大：传输带宽每年增加约 50%～100%**。寻道和旋转延迟成本下降得较慢，可能每年 5%～10%[P98]。因此，如果能够以顺序方式使用磁盘，则可以获得巨大的性能优势，随着时间的推移而增长。
- **现有文件系统在许多常见工作负载上表现不佳**。例如，FFS [MJLF84]会执行大量写入，以创建大小为一个块的新文件：一个用于新的 inode，一个用于更新 inode 位图，一个用于文件所在的目录数据块，一个用于目录 inode 以更新它，一个用于新数据块，它是新文件的一部分，另一个是数据位图，用于将数据块标记为已分配。因此，尽管 FFS 会将所有这些块放在同一个块组中，但 FFS 会导致许多短寻道和随后的旋转延迟，因此性能远远低于峰值顺序带宽。
- **文件系统不支持 RAID**。例如，RAID-4 和 RAID-5 具有小写入问题（small-write problem），即对单个块的逻辑写入会导致 4 个物理 I/O 发生。现有的文件系统不会试图避免这种最坏情况的 RAID 写入行为。

因此，理想的文件系统会专注于写入性能，并尝试利用磁盘的顺序带宽。此外，它在常见工作负载上表现良好，这种负载不仅写出数据，还经常更新磁盘上的元数据结构。最后，它可以在 RAID 和单个磁盘上运行良好。

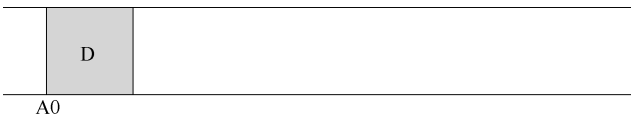
引入的新型文件系统 Rosenblum 和 Ousterhout 称为 LFS，是日志结构文件系统（Log-structured File System）的缩写。写入磁盘时，LFS 首先将所有更新（包括元数据！）缓冲在内存段中。当段已满时，它会在一次长时间的顺序传输中写入磁盘，并传输到磁盘的未使用部分。LFS 永远不会覆写现有数据，而是始终将段写入空闲位置。由于段很大，因此可以有效地使用磁盘，并且文件系统的性能接近其峰值。

关键问题：如何让所有写入变成顺序写入？

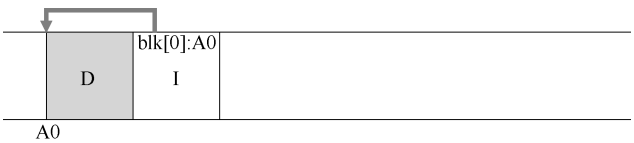
文件系统如何将所有写入转换为顺序写入？对于读取，此任务是不可能的，因为要读取的所需块可能是磁盘上的任何位置。但是，对于写入，文件系统总是有一个选择，而这正是我们希望利用的选择。

43.1 按顺序写入磁盘

因此，我们遇到了第一个挑战：如何将文件系统状态的所有更新转换为对磁盘的一系列顺序写入？为了更好地理解这一点，让我们举一个简单的例子。想象一下，我们正在将数据块 D 写入文件。将数据块写入磁盘可能会导致以下磁盘布局，其中 D 写在磁盘地址 A0：



但是，当用户写入数据块时，不仅是数据被写入磁盘；还有其他需要更新的元数据（metadata）。在这个例子中，让我们将文件的 inode（I）也写入磁盘，并将其指向数据块 D。写入磁盘时，数据块和 inode 看起来像这样（注意 inode 看起来和数据块一样大，但通常情况并非如此。在大多数系统中，数据块大小为 4KB，而 inode 小得多，大约 128B）：



提示：细节很重要

所有有趣的系统都包含一些一般性的想法和一些细节。有时，在学习这些系统时，你会对自己说，“哦，我抓住了一般的想法，其余的只是细节说明。”你这样想时，对事情是如何运作的只是一知半解。不要这样做！很多时候，细节至关重要。正如我们在 LFS 中看到的那样，一般的想法很容易理解，但要真正构建一个能工作的系统，必须仔细考虑所有棘手的情况。

简单地将所有更新（例如数据块、inode 等）顺序写入磁盘的这一基本思想是 LFS 的核心。如果你理解这一点，就抓住了基本的想法。但就像所有复杂的系统一样，魔鬼藏在细节中。

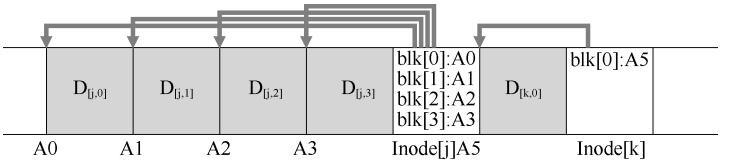
43.2 顺序而高效地写入

遗憾的是，（单单）顺序写入磁盘并不足以保证高效写入。例如，假设我们在时间 T 向地址 A 写入一个块。然后等待一会儿，再向磁盘写入地址 A+1（下一个块地址按顺序），但是在时间 T+δ。遗憾的是，在第一次和第二次写入之间，磁盘已经旋转。当你发出第二次写入时，它将在提交之前等待一大圈旋转（具体地说，如果旋转需要时间 $T_{rotation}$，则磁盘将等待 $T_{rotation}-\delta$，然后才能将第二次写入提交到磁盘表面）。因此，你可以希望看到简单地按顺序写入磁盘不足以实现最佳性能。实际上，你必须向驱动器发出大量连续写入（或一次大写入）才能获得良好的写入性能。

为了达到这个目的，LFS 使用了一种称为写入缓冲[①]（write buffering）的古老技术。在写入磁盘之前，LFS 会跟踪内存中的更新。收到足够数量的更新时，会立即将它们写入磁盘，从而确保有效使用磁盘。

LFS 一次写入的大块更新被称为段（segment）。虽然这个术语在计算机系统中被过度使用，但这里的意思是 LFS 用来对写入进行分组的大块。因此，在写入磁盘时，LFS 会缓冲内存段中的更新，然后将该段一次性写入磁盘。只要段足够大，这些写入就会很有效。

下面是一个例子，其中 LFS 将两组更新缓冲到一个小段中。实际段更大（几 MB）。第一次更新是对文件 j 的 4 次块写入，第二次是添加到文件 k 的一个块。然后，LFS 立即将整个七个块的段提交到磁盘。这些块的磁盘布局如下：



43.3 要缓冲多少

这提出了以下问题：LFS 在写入磁盘之前应该缓冲多少次更新？答案当然取决于磁盘本身，特别是与传输速率相比定位开销有多高。有关类似的分析，请参阅 FFS 相关的章节。

例如，假设在每次写入之前定位（即旋转和寻道开销）大约需要 T_{position} s。进一步假设磁盘传输速率是 R_{peak} MB / s。在这样的磁盘上运行时，LFS 在写入之前应该缓冲多少？

考虑这个问题的方法是，每次写入时，都需要支付固定的定位成本。因此，为了摊销（amortize）这笔成本，你需要写入多少？写入越多就越好（显然），越接近达到峰值带宽。

为了得到一个具体的答案，假设要写入 D MB 数据。写数据块的时间 T_{write} 是定位时间 T_{position} 的加上 D 的传输时间 $\left(\frac{D}{R_{\text{peak}}}\right)$，即

$$T_{\text{write}} = T_{\text{position}} + \frac{D}{R_{\text{peak}}} \tag{43.1}$$

因此，有效写入速率（$R_{\text{effective}}$）就是写入的数据量除以写入的总时间，即

$$R_{\text{effective}} = \frac{D}{T_{\text{write}}} = \frac{D}{T_{\text{position}} + \frac{D}{R_{\text{peak}}}} \tag{43.2}$$

我们感兴趣的是，让有效速率（$R_{\text{effective}}$）接近峰值速率。具体而言，我们希望有效速率与峰值速率的比值是某个分数 F，其中 $0 < F < 1$（典型的 F 可能是 0.9，即峰值速率的 90%）。

① 实际上，很难找到关于这个想法的好引用，因为它很可能是很多人在计算史早期发明的。有关写入缓冲的好处的研究，请参阅 Solworth 和 Orji [SO90]。要了解它的潜在危害，请参阅 Mogul [M94]。

用数学表示，这意味着我们需要 $R_{\text{effective}} = F \times R_{\text{peak}}$。

此时，我们可以解出 D：

$$R_{\text{effective}} = \frac{D}{T_{\text{position}} + \frac{D}{R_{\text{peak}}}} = F \times R_{\text{peak}} \tag{43.3}$$

$$D = F \times R_{\text{peak}} \times \left(T_{\text{position}} + \frac{D}{R_{\text{peak}}} \right) \tag{43.4}$$

$$D = (F \times R_{\text{peak}} \times T_{\text{position}}) + \left(F \times R_{\text{peak}} \times \frac{D}{R_{\text{peak}}} \right) \tag{43.5}$$

$$D = \frac{F}{1-F} \times R_{\text{peak}} \times T_{\text{position}} \tag{43.6}$$

举个例子，一个磁盘的定位时间为 10ms，峰值传输速率为 100MB/s。假设我们希望有效带宽达到峰值的 90%（$F = 0.9$）。在这种情况下，$D = 0.9 \times 100\text{MB/s} \times 0.01\text{s} = 9\text{MB}$。请尝试一些不同的值，看看需要缓冲多少才能接近峰值带宽，达到 95%的峰值需要多少，达到 99%呢？

43.4 问题：查找 inode

要了解如何在 LFS 中找到 inode，让我们简单回顾一下如何在典型的 UNIX 文件系统中查找 inode。在典型的文件系统（如 FFS）甚至老 UNIX 文件系统中，查找 inode 很容易，因为它们以数组形式组织，并放在磁盘的固定位置上。

例如，老 UNIX 文件系统将所有 inode 保存在磁盘的固定位置。因此，给定一个 inode 号和起始地址，要查找特定的 inode，只需将 inode 号乘以 inode 的大小，然后将其加上磁盘数组的起始地址，即可计算其确切的磁盘地址。给定一个 inode 号，基于数组的索引是快速而直接的。

在 FFS 中查找给定 inode 号的 inode 仅稍微复杂一些，因为 FFS 将 inode 表拆分为块并在每个柱面组中放置一组 inode。因此，必须知道每个 inode 块的大小和每个 inode 的起始地址。之后的计算类似，也很容易。

在 LFS 中，生活比较艰难。为什么？好吧，我们已经设法将 inode 分散在整个磁盘上！更糟糕的是，我们永远不会覆盖，因此最新版本的 inode（即我们想要的那个）会不断移动。

43.5 通过间接解决方案：inode 映射

为了解决这个问题，LFS 的设计者通过名为 inode 映射（inode map，imap）的数据结构，在 inode 号和 inode 之间引入了一个间接层（level of indirection）。imap 是一个结构，它将 inode 号作为输入，并生成最新版本的 inode 的磁盘地址。因此，你可以想象它通常被实现为一个

简单的数组，每个条目有 4 个字节（一个磁盘指针）。每次将 inode 写入磁盘时，imap 都会使用其新位置进行更新。

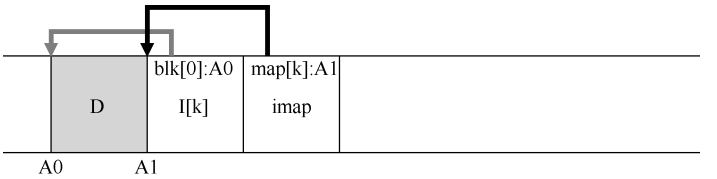
提示：使用一个间接层

人们常说，计算机科学中所有问题的解决方案就是一个间接层（level of indirection）。这显然不是真的，它只是大多数问题的解决方案。你当然可以将我们研究的每个虚拟化（例如虚拟内存）视为间接层。当然 LFS 中的 inode 映射是 inode 号的虚拟化。希望你可以在这些示例中看到间接的强大功能，允许我们自由移动结构（例如 VM 例子中的页面，或 LFS 中的 inode），而无需更改对它们的每个引用。当然，间接也可能有一个缺点：额外的开销。所以下次遇到问题时，请尝试使用间接解决方案。但请务必先考虑这样做的开销。

遗憾的是，imap 需要保持持久（写入磁盘）。这样做允许 LFS 在崩溃时仍能记录 inode 位置，从而按设想运行。因此有一个问题：imap 应该驻留在磁盘上的哪个位置？

当然，它可以存在于磁盘的固定部分。遗憾的是，由于它经常更新，因此需要更新文件结构，然后写入 imap，因此性能会受到影响（每次的更新和 imap 的固定位置之间，会有更多的磁盘寻道）。

与此不同，LFS 将 inode 映射的块放在它写入所有其他新信息的位置旁边。因此，当将数据块追加到文件 k 时，LFS 实际上将新数据块，其 inode 和一段 inode 映射一起写入磁盘，如下所示：



在该图中，imap 数组存储在标记为 imap 的块中，它告诉 LFS，inode k 位于磁盘地址 A1。接下来，这个 inode 告诉 LFS 它的数据块 D 在地址 A0。

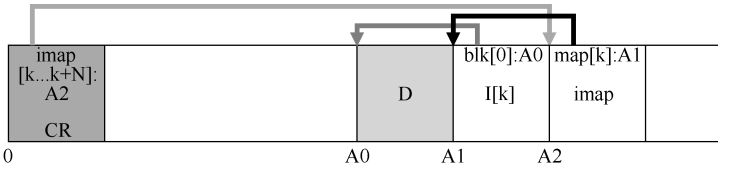
43.6 检查点区域

聪明的读者（就是你，对吗？）可能已经注意到了这里的问题。我们如何找到 inode 映射，现在它的各个部分现在也分布在整个磁盘上？归根到底：文件系统必须在磁盘上有一些固定且已知的位置，才能开始文件查找。

LFS 在磁盘上只有这样一个固定的位置，称为检查点区域（checkpoint region，CR）。检查点区域包含指向最新的 inode 映射片段的指针（即地址），因此可以通过首先读取 CR 来找到 inode 映射片段。请注意，检查点区域仅定期更新（例如每 30s 左右），因此性能不会受到影响。因此，磁盘布局的整体结构包含一个检查点区域（指向内部映射的最新部分），每个 inode 映射块包含 inode 的地址，inode 指向文件（和目录），就像典型的 UNIX 文件系统一样。

下面的例子是检查点区域（注意它始终位于磁盘的开头，地址为 0），以及单个 imap 块，

inode 和数据块。一个真正的文件系统当然会有一个更大的 CR（事实上，它将有两个，我们稍后会理解），许多 imap 块，当然还有更多的 inode、数据块等。



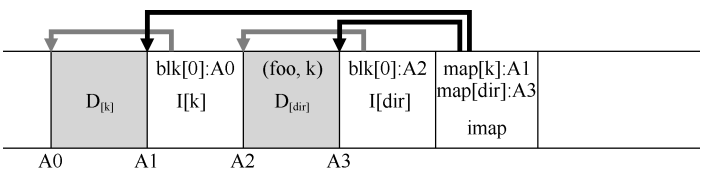
43.7 从磁盘读取文件：回顾

为了确保理解 LFS 的工作原理，现在让我们来看看从磁盘读取文件时必须发生的事情。假设从内存中没有任何东西开始。我们必须读取的第一个磁盘数据结构是检查点区域。检查点区域包含指向整个 inode 映射的指针（磁盘地址），因此 LFS 读入整个 inode 映射并将其缓存在内存中。在此之后，当给定文件的 inode 号时，LFS 只是在 imap 中查找 inode 号到 inode 磁盘地址的映射，并读入最新版本的 inode。要从文件中读取块，此时，LFS 完全按照典型的 UNIX 文件系统进行操作，方法是使用直接指针或间接指针或双重间接指针。在通常情况下，从磁盘读取文件时，LFS 应执行与典型文件系统相同数量的 I/O，整个 imap 被缓存，因此 LFS 在读取过程中所做的额外工作是在 imap 中查找 inode 的地址。

43.8 目录如何

到目前为止，我们通过仅考虑 inode 和数据块，简化了讨论。但是，要访问文件系统中的文件（例如/home/remzi/foo，我们最喜欢的伪文件名之一），也必须访问一些目录。那么 LFS 如何存储目录数据呢？

幸运的是，目录结构与传统的 UNIX 文件系统基本相同，因为目录只是（名称，inode 号）映射的集合。例如，在磁盘上创建文件时，LFS 必须同时写入新的 inode，一些数据，以及引用此文件的目录数据及其 inode。请记住，LFS 将在磁盘上按顺序写入（在缓冲更新一段时间后）。因此，在目录中创建文件 foo，将导致磁盘上的以下新结构：



inode 映射的片段包含目录文件 dir 以及新创建的文件 f 的位置信息。因此，访问文件 foo（具有 inode 号 f）时，你先要查看 inode 映射（通常缓存在内存中），找到目录 dir（A3）的 inode 的位置。然后读取目录的 inode，它给你目录数据的位置（A2）。读取此数据块为你提供名称到 inode 号的映射（foo，k）。然后再次查阅 inode 映射，找到 inode 号 k（A1）的

位置，最后在地址 A0 处读取所需的数据块。

inode 映射还解决了 LFS 中存在的另一个严重问题，称为递归更新问题（recursive update problem）[Z+12]。任何永远不会原地更新的文件系统（例如 LFS）都会遇到该问题，它们将更新移动到磁盘上的新位置。

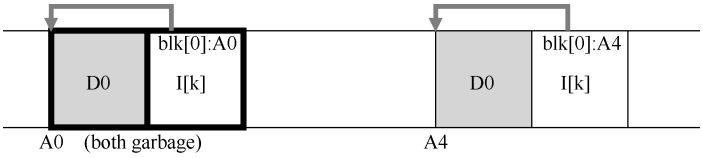
具体来说，每当更新 inode 时，它在磁盘上的位置都会发生变化。如果我们不小心，这也会导致对指向该文件的目录的更新，然后必须更改该目录的父目录，依此类推，一路沿文件系统树向上。

LFS 巧妙地避免了 inode 映射的这个问题。即使 inode 的位置可能会发生变化，更改也不会反映在目录本身中。事实上，imap 结构被更新，而目录保持相同的名称到 inumber 的映射。因此，通过间接，LFS 避免了递归更新问题。

43.9 一个新问题：垃圾收集

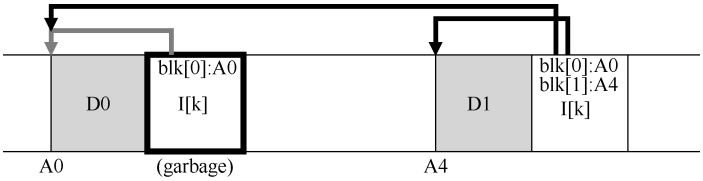
你可能已经注意到 LFS 的另一个问题；它会反复将最新版本的文件（包括其 inode 和数据）写入磁盘上的新位置。此过程在保持写入效率的同时，意味着 LFS 会在整个磁盘中分散旧版本的文件结构。我们（毫不客气地）将这些旧版本称为垃圾（garbage）。

例如，假设有一个由 inode 号 k 引用的现有文件，该文件指向单个数据块 D0。我们现在覆盖该块，生成新的 inode 和新的数据块。由此产生的 LFS 磁盘布局看起来像这样（注意，简单起见，我们省略了 imap 和其他结构。还需要将一个新的 imap 大块写入磁盘，以指向新的 inode）：



在图中，可以看到 inode 和数据块在磁盘上有两个版本，一个是旧的（左边那个），一个是当前的，因此是活的（live，右边那个）。对于覆盖数据块的简单行为，LFS 必须持久许多新结构，从而在磁盘上留下上述块的旧版本。

另外举个例子，假设我们将一块添加到该原始文件 k 中。在这种情况下，会生成新版本的 inode，但旧数据块仍由旧 inode 指向。因此，它仍然存在，并且与当前文件系统分离：



那么，应该如何处理这些旧版本的 inode、数据块等呢？可以保留那些旧版本并允许用户恢复旧文件版本（例如，当他们意外覆盖或删除文件时，这样做可能非常方便）。这样的文件系统称为版本控制文件系统（versioning file system），因为它跟踪文件的不同版本。

但是，LFS 只保留文件的最新活版本。因此（在后台），LFS 必须定期查找文件数据，索引节点和其他结构的旧的死版本，并清理（clean）它们。因此，清理应该使磁盘上的块再次空闲，以便在后续写入中使用。请注意，清理过程是垃圾收集（garbage collection）的一种形式，这种技术在编程语言中出现，可以自动为程序释放未使用的内存。

之前我们讨论过的段很重要，因为它们是在 LFS 中实现对磁盘的大段写入的机制。事实证明，它们也是有效清理的重要组成部分。想象一下，如果 LFS 清理程序在清理过程中简单地通过并释放单个数据块，索引节点等，会发生什么。结果：文件系统在磁盘上分配的空间之间混合了一些空闲洞（hole）。写入性能会大幅下降，因为 LFS 无法找到一个大块连续区域，以便顺序地写入磁盘，获得高性能。

实际上，LFS 清理程序按段工作，从而为后续写入清理出大块空间。基本清理过程的工作原理如下。LFS 清理程序定期读入许多旧的（部分使用的）段，确定哪些块在这些段中存在，然后写出一组新的段，只包含其中活着的块，从而释放旧块用于写入。具体来说，我们预期清理程序读取 M 个现有段，将其内容打包（compact）到 N 个新段（其中 $N < M$），然后将 N 段写入磁盘的新位置。然后释放旧的 M 段，文件系统可以使用它们进行后续写入。

但是，我们现在有两个问题。第一个是机制：LFS 如何判断段内的哪些块是活的，哪些块已经死了？第二个是策略：清理程序应该多久运行一次，以及应该选择清理哪些部分？

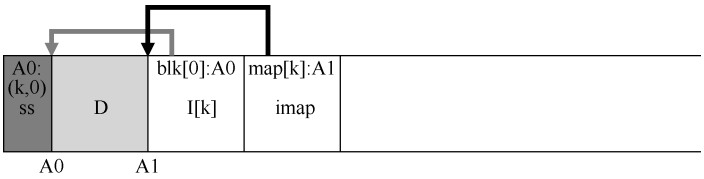
43.10 确定块的死活

我们首先关注这个问题。给定磁盘段 S 内的数据块 D，LFS 必须能够确定 D 是不是活的。为此，LFS 会为描述每个块的每个段添加一些额外信息。具体地说，对于每个数据块 D，LFS 包括其 inode 号（它属于哪个文件）及其偏移量（这是该文件的哪一块）。该信息记录在一个数据结构中，位于段头部，称为段摘要块（segment summary block）。

根据这些信息，可以直接确定块的死活。对于位于地址 A 的磁盘上的块 D，查看段摘要块并找到其 inode 号 N 和偏移量 T。接下来，查看 imap 以找到 N 所在的位置，并从磁盘读取 N（可能它已经在内存中，这更好）。最后，利用偏移量 T，查看 inode（或某个间接块），看看 inode 认为此文件的第 T 个块在磁盘上的位置。如果它刚好指向磁盘地址 A，则 LFS 可以断定块 D 是活的。如果它指向其他地方，LFS 可以断定 D 未被使用（即它已经死了），因此知道不再需要该版本。下面的伪代码总结了这个过程：

```
(N, T) = SegmentSummary[A];
inode  = Read(imap[N]);
if (inode[T] == A)
    // block D is alive
else
    // block D is garbage
```

下面是一个描述机制的图，其中段摘要块（标记为 SS）记录了地址 A0 处的数据块，实际上是文件 k 在偏移 0 处的部分。通过检查 imap 的 k，可以找到 inode，并且看到它确实指向该位置。



LFS 走了一些捷径，可以更有效地确定死活。例如，当文件被截断或删除时，LFS 会增加其版本号（version number），并在 imap 中记录新版本号。通过在磁盘上的段中记录版本号，LFS 可以简单地通过将磁盘版本号与 imap 中的版本号进行比较，跳过上述较长的检查，从而避免额外的读取。

43.11 策略问题：要清理哪些块，何时清理

在上述机制的基础上，LFS 必须包含一组策略，以确定何时清理以及哪些块值得清理。确定何时清理比较容易。要么是周期性的，要么是空闲时间，要么是因为磁盘已满。

确定清理哪些块更具挑战性，并且已成为许多研究论文的主题。在最初的 LFS 论文[RO91]中，作者描述了一种试图分离冷热段的方法。热段是经常覆盖内容的段。因此，对于这样的段，最好的策略是在清理之前等待很长时间，因为越来越多的块被覆盖（在新的段中），从而被释放以供使用。相比之下，冷段可能有一些死块，但其余的内容相对稳定。因此，作者得出结论，应该尽快清理冷段，延迟清理热段，并开发出一种完全符合要求的试探算法。但是，与大多数政策一样，这只是一种方法，当然并非“最佳”方法。后来的一些方法展示了如何做得更好[MR+97]。

43.12 崩溃恢复和日志

最后一个问题：如果系统在 LFS 写入磁盘时崩溃会发生什么？你可能还记得上一章讲的日志，在更新期间崩溃对于文件系统来说是棘手的，因此 LFS 也必须考虑这些问题。

在正常操作期间，LFS 将一些写入缓冲在段中，然后（当段已满或经过一段时间后），将段写入磁盘。LFS 在日志（log）中组织这些写入，即指向头部段和尾部段的检查点区域，并且每个段指向要写入的下一个段。LFS 还定期更新检查点区域（CR）。在这些操作期间都可能发生崩溃（写入段，写入 CR）。那么 LFS 在写入这些结构时如何处理崩溃？

我们先介绍第二种情况。为了确保 CR 更新以原子方式发生，LFS 实际上保留了两个 CR，每个位于磁盘的一端，并交替写入它们。当使用最新的指向 inode 映射和其他信息的指针更新 CR 时，LFS 还实现了一个谨慎的协议。具体来说，它首先写出一个头（带有时间戳），然后写出 CR 的主体，然后最后写出最后一部分（也带有时间戳）。如果系统在 CR 更新期间崩溃，LFS 可以通过查看一对不一致的时间戳来检测到这一点。LFS 将始终选择使用具有一致时间戳的最新 CR，从而实现 CR 的一致更新。

我们现在关注第一种情况。由于 LFS 每隔 30s 左右写入一次 CR，因此文件系统的最后一致快照可能很旧。因此，在重新启动时，LFS 可以通过简单地读取检查点区域、它指向的 imap 片段以及后续文件和目录，从而轻松地恢复。但是，最后许多秒的更新将会丢失。

为了改进这一点，LFS 尝试通过数据库社区中称为前滚（roll forward）的技术，重建其中许多段。基本思想是从最后一个检查点区域开始，找到日志的结尾（包含在 CR 中），然后使用它来读取下一个段，并查看其中是否有任何有效更新。如果有，LFS 会相应地更新文件系统，从而恢复自上一个检查点以来写入的大部分数据和元数据。有关详细信息，请参阅 Rosenblum 获奖论文[R92]。

43.13 小结

LFS 引入了一种更新磁盘的新方法。LFS 总是写入磁盘的未使用部分，然后通过清理回收旧空间，而不是在原来的位置覆盖文件。这种方法在数据库系统中称为影子分页（shadow paging）[L77]，在文件系统中有时称为写时复制（copy-on-write），可以实现高效写入，因为 LFS 可以将所有更新收集到内存的段中，然后按顺序一起写入。

这种方法的缺点是它会产生垃圾。旧数据的副本分散在整个磁盘中，如果想要为后续使用回收这样的空间，则必须定期清理旧的数据段。清理成为 LFS 争议的焦点，对清理成本的担忧[SS+95]可能限制了 LFS 开始对该领域的影响。然而，一些现代商业文件系统，包括 NetApp 的 WAFL [HLM94]、Sun 的 ZFS [B07]和 Linux btrfs [M07]，采用了类似的写时复制方法来写入磁盘，因此 LFS 的知识遗产继续存在于这些现代文件系统中。特别是，WAFL 通过将清理问题转化为特征来解决问题。通过快照（snapshots）提供旧版本的文件系统，用户可以在意外删除当前文件时，访问到旧文件。

提示：将缺点变成美德

每当你的系统存在根本缺点时，请看看是否可以将它转换为特征或有用的功能。NetApp 的 WAFL 对旧文件内容做到了这一点。通过提供旧版本，WAFL 不再需要担心清理，还因此提供了一个很酷的功能，在一个美妙的转折中消除了 LFS 的清理问题。系统中还有其他这样的例子吗？毫无疑问还有，但你必须自己去思考，因为本章的内容已经结束了。

参考资料

[B07]“ZFS: The Last Word in File Systems”Jeff Bonwick and Bill Moore
关于 ZFS 的幻灯片。遗憾的是，没有优秀的 ZFS 论文。

[HLM94]“File System Design for an NFS File Server Appliance”Dave Hitz, James Lau, Michael Malcolm
USENIX Spring ’94

WAFL 从 LFS 和 RAID 中获取了许多想法，并将其置于数十亿美元的存储公司 NetApp 的高速 NFS 设备中。

[L77]“Physical Integrity in a Large Segmented Database”
R. Lorie
ACM Transactions on Databases, 1977, Volume 2:1, pages 91-104
这里介绍了影子分页的最初想法。

[M07]“The Btrfs Filesystem”Chris Mason
September 2007
最近的一种写时复制 Linux 文件系统，其重要性和使用率逐渐增加。

[MJLF84]“A Fast File System for UNIX”
Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry ACM TOCS, August, 1984, Volume 2, Number 3
最初的 FFS 文件。有关详细信息，请参阅有关 FFS 的章节。

[MR+97]“Improving the Performance of Log-structured File Systems with Adaptive Methods”
Jeanna Neefe Matthews, Drew Roselli, Adam M. Costello,
Randolph Y. Wang, Thomas E. Anderson
SOSP 1997, pages 238-251, October, Saint Malo, France
最近的一篇论文，详细说明了 LFS 中更好的清理策略。

[M94]“A Better Update Policy”Jeffrey C. Mogul
USENIX ATC ’94, June 1994
在该文中，Mogul 发现，因为缓冲写入时间过长，然后集中发送到磁盘中，读取工作负载可能会受到影响。因此，他建议更频繁地以较小的批次发送写入。

[P98]“Hardware Technology Trends and Database Opportunities”David A. Patterson
ACM SIGMOD ’98 Keynote Address, Presented June 3, 1998, Seattle, Washington
关于计算机系统技术趋势的一系列幻灯片。也许 Patterson 很快就会制作另一个幻灯片。

[RO91]“Design and Implementation of the Log-structured File System”Mendel Rosenblum and John Ousterhout
SOSP ’91, Pacific Grove, CA, October 1991
关于 LFS 的原始 SOSP 论文，已被数百篇其他论文引用，并启发了许多真实系统。

[R92]“Design and Implementation of the Log-structured File System”Mendel Rosenblum
关于 LFS 的获奖学位论文，包含其他论文中缺失的许多细节。

[SS+95]“File system logging versus clustering: a performance comparison”
Margo Seltzer, Keith A. Smith, Hari Balakrishnan, Jacqueline Chang, Sara McMains, Venkata Padmanabhan
USENIX 1995 Technical Conference, New Orleans, Louisiana, 1995
该文显示 LFS 的性能有时会出现问题，特别是对于多次调用 fsync()的工作负载（例如数据库工作负载）。

该论文当时备受争议。

[SO90]“Write-Only Disk Caches”Jon A. Solworth, Cyril U. Orji
SIGMOD ’90, Atlantic City, New Jersey, May 1990
对写缓冲及其好处的早期研究。但是，缓冲太长时间可能会产生危害，详情请参阅 Mogul [M94]。

[Z+12]“De-indirection for Flash-based SSDs with Nameless Writes”
Yiying Zhang, Leo Prasath Arulraj, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau FAST ’13, San Jose, California, February 2013
我们的论文介绍了构建基于闪存的存储设备的新方法。由于 FTL（闪存转换层）通常以日志结构样式构建，因此在基于闪存的设备中会出现一些 LFS 中相同的问题。在这个例子中，它是递归更新问题，LFS 用 imap 巧妙地解决了这个问题。大多数 SSD 中存在类似的结构。