种罕见的操作（仅在系统意外崩溃之后发生），所以几次冗余写入无须担心[①]。

批处理日志更新

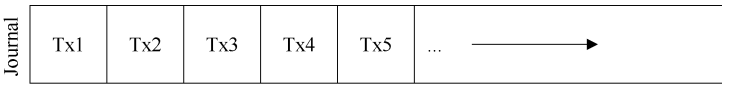
你可能已经注意到，基本协议可能会增加大量额外的磁盘流量。例如，假设我们在同一目录中连续创建两个文件，称为 file1 和 file2。要创建一个文件，必须更新许多磁盘上的结构，至少包括：inode 位图（分配新的 inode），新创建的文件 inode，包含新文件目录条目的父目录的数据块，以及父目录的 inode（现在有一个新的修改时间）。通过日志，我们将所有这些信息逻辑地提交给我们的两个文件创建的日志。因为文件在同一个目录中，我们假设在同一个 inode 块中都有 inode，这意味着如果不小心，我们最终会一遍又一遍地写入这些相同的块。

为了解决这个问题，一些文件系统不会一次一个地向磁盘提交每个更新（例如，Linux ext3）。与此不同，可以将所有更新缓冲到全局事务中。在上面的示例中，当创建两个文件时，文件系统只将内存中的 inode 位图、文件的 inode、目录数据和目录 inode 标记为脏，并将它们添加到块列表中，形成当前的事务。当最后应该将这些块写入磁盘时（例如，在超时 5s 之后），会提交包含上述所有更新的单个全局事务。因此，通过缓冲更新，文件系统在许多情况下可以避免对磁盘的过多的写入流量。

使日志有限

因此，我们已经了解了更新文件系统磁盘结构的基本协议。文件系统缓冲内存中的更新一段时间。最后写入磁盘时，文件系统首先仔细地将事务的详细信息写入日志（即预写日志）。事务完成后，文件系统会加检查点，将这些块写入磁盘上的最终位置。

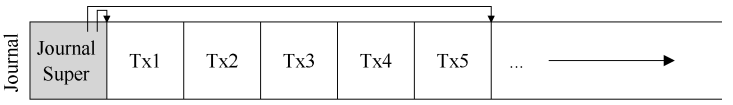
但是，日志的大小有限。如果不断向它添加事务（如下所示），它将很快填满。你觉得会发生什么？



日志满时会出现两个问题。第一个问题比较简单，但不太重要：日志越大，恢复时间越长，因为恢复过程必须重放日志中的所有事务（按顺序）才能恢复。第二个问题更重要：当日志已满（或接近满）时，不能向磁盘提交进一步的事务，从而使文件系统“不太有用”（即无用）。

为了解决这些问题，日志文件系统将日志视为循环数据结构，一遍又一遍地重复使用。这就是为什么日志有时被称为循环日志（circular log）。为此，文件系统必须在加检查点之后的某个时间执行操作。具体来说，一旦事务被加检查点，文件系统应释放它在日志中占用的空间，允许重用日志空间。有很多方法可以达到这个目的。例如，你只需在日志超级块（journal superblock）中标记日志中最旧和最新的事务。所有其他空间都是空闲的。以下是这种机制的图形描述：

① 除非你担心一切，在这种情况下我们无法帮助你。不要太担心，这是不健康的！ 但现在你可能担心自己会过度担心。



在日志超级块中（不要与主文件系统的超级块混淆），日志系统记录了足够的信息，以了解哪些事务尚未加检查点，从而减少了恢复时间，并允许以循环的方式重新使用日志。因此，我们在基本协议中添加了另一个步骤。

1．**日志写入**：将事务的内容（包括 TxB 和更新内容）写入日志，等待这些写入完成。

2．**日志提交**：将事务提交块（包括 TxE）写入日志，等待写完成，事务被认为已提交（committed）。

3．**加检查点**：将更新内容写入其最终的磁盘位置。

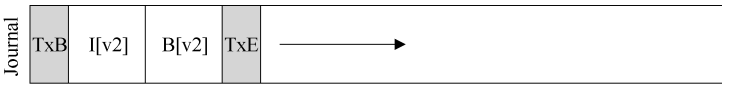
4．**释放**：一段时间后，通过更新日志超级块，在日志中标记该事务为空闲。

因此，我们得到了最终的数据日志协议。但仍然存在一个问题：我们将每个数据块写入磁盘两次，这是沉重的成本，特别是为了系统崩溃这样罕见的事情。你能找到一种方法来保持一致性，而无须两次写入数据吗？

元数据日志

尽管恢复现在很快（扫描日志并重放一些事务而不是扫描整个磁盘），但文件系统的正常操作比我们想要的要慢。特别是，对于每次写入磁盘，我们现在也要先写入日志，从而使写入流量加倍。在顺序写入工作负载期间，这种加倍尤为痛苦，现在将以驱动器峰值写入带宽的一半进行。此外，在写入日志和写入主文件系统之间，存在代价高昂的寻道，这为某些工作负载增加了显著的开销。

由于将每个数据块写入磁盘的成本很高，人们为了提高性能，尝试了一些不同的东西。例如，我们上面描述的日志模式通常称为数据日志（data journaling，如在 Linux ext3 中），因为它记录了所有用户数据（除了文件系统的元数据之外）。一种更简单（也更常见）的日志形式有时称为有序日志（ordered journaling，或称为元数据日志，metadata journaling），它几乎相同，只是用户数据没有写入日志。因此，在执行与上述相同的更新时，以下信息将写入日志：



先前写入日志的数据块 Db 将改为写入文件系统，避免额外写入。考虑到磁盘的大多数 I/O 流量是数据，不用两次写入数据会大大减少日志的 I/O 负载。然而，修改确实提出了一个有趣的问题：我们何时应该将数据块写入磁盘？

再考虑一下文件追加的例子，以更好地理解问题。更新包含 3 个块：I[v2]、B[v2]和 Db。前两个都是元数据，将被记录，然后加检查点。后者只会写入文件系统一次。什么时候应该把 Db 写入磁盘？这有关系吗？

事实证明，数据写入的顺序对于仅元数据日志很重要。例如，如果我们在事务（包含 I[v2]和 B [v2]）完成后将 Db 写入磁盘如何？遗憾的是，这种方法存在一个问题：文件系统是一致的，但 I[v2]可能最终指向垃圾数据。具体来说，考虑写入了 I[v2]和 B[v2]，但 Db 没有写入磁盘的情况。然后文件系统将尝试恢复。由于 Db 不在日志中，因此文件系统将重放

对 I[v2]和 B[v2]的写入，并生成一致的文件系统（从文件系统元数据的角度来看）。但是，I[v2]将指向垃圾数据，即指向 Db 中的任何数据。

为了确保不出现这种情况，在将相关元数据写入磁盘之前，一些文件系统（例如，Linux ext3）先将数据块（常规文件）写入磁盘。具体来说，协议有以下几个。

1．**数据写入**：将数据写入最终位置，等待完成（等待是可选的，详见下文）。

2．**日志元数据写入**：将开始块和元数据写入日志，等待写入完成。

3．**日志提交**：将事务提交块（包括 TxE）写入日志，等待写完成，现在认为事务（包括数据）已提交（committed）。

4．**加检查点元数据**：将元数据更新的内容写入文件系统中的最终位置。

5．**释放**：稍后，在日志超级块中将事务标记为空闲。

通过强制先写入数据，文件系统可以保证指针永远不会指向垃圾。实际上，这个“先写入被指对象，再写入指针对象”的规则是崩溃一致性的核心，并且被其他崩溃一致性方案[GP94]进一步利用。

在大多数系统中，元数据日志（类似于 ext3 的有序日志）比完整数据日志更受欢迎。例如，Windows NTFS 和 SGI 的 XFS 都使用无序的元数据日志。Linux ext3 为你提供了选择数据、有序或无序模式的选项（在无序模式下，可以随时写入数据）。所有这些模式都保持元数据一致，它们的数据语义各不相同。

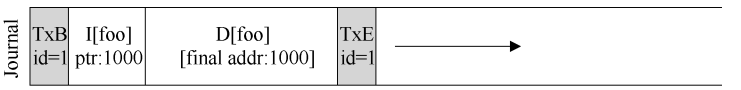
最后，请注意，在发出写入日志（步骤 2）之前强制数据写入完成（步骤 1）不是正确性所必需的，如上面的协议所示。具体来说，可以发出数据写入，并向日志写入事务开始块和元数据。唯一真正的要求，是在发出日志提交块之前完成步骤 1 和步骤 2（步骤 3）。

棘手的情况：块复用

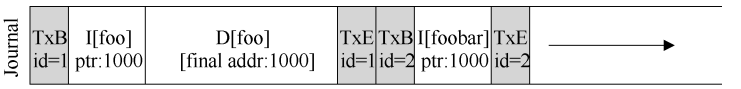
一些有趣的特殊情况让日志更加棘手，因此值得讨论。其中一些与块复用有关。正如 Stephen Tweedie（ext3 背后的主要开发者之一）说的：

“整个系统的可怕部分是什么？……是删除文件。与删除有关的一切都令人毛骨悚然。与删除有关的一切……如果块被删除然后重新分配，你会做噩梦。”[T00]

Tweedie 给出的具体例子如下。假设你正在使用某种形式的元数据日志（因此不记录文件的数据块）。假设你有一个名为 foo 的目录。用户向 foo 添加一个条目（例如通过创建文件），因此 foo 的内容（因为目录被认为是元数据）被写入日志。假设 foo 目录数据的位置是块 1000。因此日志包含如下内容：



此时，用户删除目录中的所有内容以及目录本身，从而释放块 1000 以供复用。最后，用户创建了一个新文件（比如 foobar），结果复用了过去属于 foo 的相同块（1000）。foobar 的 inode 提交给磁盘，其数据也是如此。但是，请注意，因为正在使用元数据日志，所以只有 foobar 的 inode 被提交给日志，文件 foobar 中块 1000 中新写入的数据没有写入日志。



现在假设发生了崩溃，所有这些信息仍然在日志中。在重放期间，恢复过程简单地重放日志中的所有内容，包括在块 1000 中写入目录数据。因此，重放会用旧目录内容覆盖当前文件 foobar 的用户数据！显然，这不是一个正确的恢复操作，当然，在阅读文件 foobar 时，用户会感到惊讶。

这个问题有一些解决方案。例如，可以永远不再重复使用块，直到所述块的删除加上检查点，从日志中清除。Linux ext3 的做法是将新类型的记录添加到日志中，称为撤销（revoke）记录。在上面的情况中，删除目录将导致撤销记录被写入日志。在重放日志时，系统首先扫描这样的重新记录。任何此类被撤销的数据都不会被重放，从而避免了上述问题。

总结日志：时间线

在结束对日志的讨论之前，我们总结一下讨论过的协议，用时间线来描述每个协议。表 42.1 展示了日志数据和元数据时的协议，表 42.2 展示了仅记录元数据时的协议。

表 42.1 数据日志的时间线

	日志			文件系统	
TxB	内容		TxE	元数据	数据
	（元数据）	（数据）			
发出	发出	发出			
完成					
	完成				
		完成			
			发出		
			完成		
				发出	发出
					完成
				完成	

表 42.2 元数据日志的时间线

	日志		文件系统	
TxB	内容	TxE	元数据	数据
	（元数据）			
发出	发出			发出
				完成
完成				
	完成			
		发出		
		完成		
			发出	
			完成	

在每个表中，时间向下增加，表中的每一行显示可以发出或可能完成写入的逻辑时间。例如，在数据日志协议（见表 42.1）中，事务开始块（TxB）的写入和事务的内容可以在逻辑上同时发出，因此可以按任何顺序完成。但是，在上述写入完成之前，不得发出对事务结束块（TxE）的写入。同样，在事务结束块提交之前，写入数据和元数据块的加检查点无法开始。水平虚线表示必须遵守的写入顺序要求。

对元数据日志协议也显示了类似的时间线。请注意，在逻辑上，数据写入可以与对事务开始的写入和日志的内容一起发出。但是，必须在事务结束发出之前发出并完成。

最后，请注意，时间线中每次写入标记的完成时间是任意的。在实际系统中，完成时间由 I/O 子系统确定，I/O 子系统可能会重新排序写入以提高性能。对于顺序的唯一保证，是那些必须强制执行，才能保证协议正确性的顺序。

42.4 解决方案 3：其他方法

到目前为止，我们已经描述了保持文件系统元数据一致性的两个可选方法：基于 fsck 的偷懒方法，以及称为日志的更活跃的方法。但是，并不是只有这两种方法。Ganger 和 Patt 引入了一种称为软更新[GP94]的方法。这种方法仔细地对文件系统的所有写入排序，以确保磁盘上的结构永远不会处于不一致的状态。例如，通过先写入指向的数据块，再写入指向它的 inode，可以确保 inode 永远不会指向垃圾。对文件系统的所有结构可以导出类似的规则。然而，实现软更新可能是一个挑战。上述日志层的实现只需要具体文件系统结构的较少知识，但软更新需要每个文件系统数据结构的复杂知识，因此给系统增加了相当大的复杂性。

另一种方法称为写时复制（Copy-On-Write，COW），并且在许多流行的文件系统中使用，包括 Sun 的 ZFS [B07]。这种技术永远不会覆写文件或目录。相反，它会对磁盘上以前未使用的位置进行新的更新。在完成许多更新后，COW 文件系统会翻转文件系统的根结构，以包含指向刚更新结构的指针。这样做可以使文件系统保持一致。在将来的章节中讨论日志结构文件系统（LFS）时，我们将学习更多关于这种技术的知识。LFS 是 COW 的早期范例。

另一种方法是我们刚刚在威斯康星大学开发的方法。这种技术名为基于反向指针的一致性（Backpointer-Based Consistency，BBC），它在写入之间不强制执行排序。为了实现一致性，系统中的每个块都会添加一个额外的反向指针。例如，每个数据块都引用它所属的 inode。访问文件时，文件系统可以检查正向指针（inode 或直接块中的地址）是否指向引用它的块，从而确定文件是否一致。如果是这样，一切都肯定安全地到达磁盘，因此文件是一致的。如果不是，则文件不一致，并返回错误。通过向文件系统添加反向指针，可以获得一种新形式的惰性崩溃一致性[C+12]。

最后，我们还探索了减少日志协议等待磁盘写入完成的次数的技术。这种新方法名为乐观崩溃一致性（optimistic crash consistency）[C+13]，尽可能多地向磁盘发出写入，并利用事务校验和（transaction checksum）[P+05]的一般形式，以及其他一些技术来检测不一致，如果出现不一致的话。对于某些工作负载，这些乐观技术可以将性能提高一个数量级。但是，要真正运行良好，需要稍微不同的磁盘接口[C+13]。

42.5 小结

我们介绍了崩溃一致性的问题，并讨论了处理这个问题的各种方法。构建文件系统检查程序的旧方法有效，但在现代系统上恢复可能太慢。因此，许多文件系统现在使用日志。日志可将恢复时间从 O（磁盘大小的卷）减少到 O（日志大小），从而在崩溃和重新启动后大大加快恢复速度。因此，许多现代文件系统都使用日志。我们还看到日志可以有多种形式。最常用的是有序元数据日志，它可以减少日志流量，同时仍然保证文件系统元数据和用户数据的合理一致性。

参考资料

[B07]“ZFS: The Last Word in File Systems”Jeff Bonwick and Bill Moore
实际上，ZFS 使用写时复制和日志，因为在某些情况下，对磁盘的写入记日志性能更好。

[C+12]“Consistency Without Ordering”
Vijay Chidambaram, Tushar Sharma, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau FAST ’12, San Jose, California
我们最近发表的一篇关于基于反向指针的新形式的崩溃一致性的论文。阅读它，了解令人兴奋的细节内容！

[C+13]“Optimistic Crash Consistency”
Vijay Chidambaram, Thanu S. Pillai, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau SOSP ’13, Nemacolin Woodlands Resort, PA, November 2013
我们致力于更乐观、更高性能的日志协议。对于大量调用 fsync()的工作负载，可以大大提高性能。

[GP94]“Metadata Update Performance in File Systems”Gregory R. Ganger and Yale N. Patt
OSDI ’94
一篇关于使用谨慎的写入顺序作为实现一致性的主要方法的优秀论文。后来在基于 BSD 的系统中实现。

[G+08]“SQCK: A Declarative File System Checker”
Haryadi S. Gunawi, Abhishek Rajimwale, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau OSDI ’08, San Diego, California
我们自己的论文，介绍了一种使用 SQL 查询构建文件系统检查程序的新方法。我们还展示了现有检查器的一些问题，发现了许多错误和奇怪的行为，这是 FSCK 复杂性的直接结果。

[H87]“Reimplementing the Cedar File System Using Logging and Group Commit”Robert Hagmann
SOSP ’87, Austin, Texas, November 1987
第一项工作（我们所知的）将预写日志（即日志）应用于文件系统。

[M+13]“ffsck: The Fast File System Checker”
Ao Ma, Chris Dragga, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau FAST ’13, San Jose, California, February 2013
这篇文章详细介绍了如何让 FSCK 快一个数量级。一些想法已经集成到 BSD 文件系统检查器[MK96]中，

并已部署。

[MK96]“Fsck - The UNIX File System Check Program”Marshall Kirk McKusick and T. J. Kowalski
Revised in 1996
由开发 FFS 的一些人编写的描述第一个全面的文件系统检查工具，即因此得名的 FSCK。

[MJLF84]“A Fast File System for UNIX”
Marshall K. McKusick, William N. Joy, Sam J. Leffler, Robert S. Fabry ACM Transactions on Computing Systems.
August 1984, Volume 2:3
你已经对 FFS 了解得够多了，对吗？但是，可以在书中不止一次地引用这样的论文。

[P+05]“IRON File Systems”
Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, An- drea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
SOSP ’05, Brighton, England, October 2005
该论文主要关注研究文件系统如何对磁盘故障做出反应。在最后，我们引入了一个事务校验和来加速日志，最终被 Linux ext4 采用。

[PAA05]“Analysis and Evolution of Journaling File Systems”
Vijayan Prabhakaran, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau USENIX ’05, Anaheim, California, April 2005
我们早期写的一篇论文，分析了日志文件系统的工作原理。

[R+11]“Coerced Cache Eviction and Discreet-Mode Journaling”Abhishek Rajimwale, Vijay Chidambaram, Deepak Ramamurthi, Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau
DSN ’11, Hong Kong, China, June 2011
我们自己的论文，关于磁盘缓冲写入内存缓存而不是强制它们写入磁盘的问题，即使明确告知不这样做！我们克服这个问题的解决方案：如果你想在 B 之前将 A 写入磁盘，首先写 A，然后向磁盘发送大量“虚拟”写入，希望将 A 强制写入磁盘，以便为它们在缓存中腾出空间。一个简洁但不太实际的解决方案。

[T98]“Journaling the Linux ext2fs File System”Stephen C. Tweedie
The Fourth Annual Linux Expo, May 1998
Tweedie 在为 Linux ext2 文件系统添加日志方面做了大量工作。结果毫不奇怪地被称为 ext3。一些不错的设计决策包括强烈关注向后兼容性，例如，你只需将日志文件添加到现有的 ext2 文件系统，然后将其挂载为 ext3 文件系统。

[T00]“EXT3, Journaling Filesystem”Stephen Tweedie
Talk at the Ottawa Linux Symposium, July 2000 olstrans.sourceforge.net/release/OLS2000-ext3/OLS2000-ext3.html
Tweedie 关于 ext3 的演讲的文字记录。

[T01]“The Linux ext2 File System”Theodore Ts’o, June, 2001.
一个简单的 Linux 文件系统，基于 FFS 中的想法。有一段时间它被大量使用，现在它真的只是在内核中作为简单文件系统的一个例子。

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。

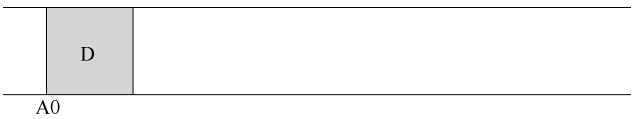
引入的新型文件系统 Rosenblum 和 Ousterhout 称为 LFS，是日志结构文件系统（Log-structured File System）的缩写。写入磁盘时，LFS 首先将所有更新（包括元数据！）缓冲在内存段中。当段已满时，它会在一次长时间的顺序传输中写入磁盘，并传输到磁盘的未使用部分。LFS 永远不会覆写现有数据，而是始终将段写入空闲位置。由于段很大，因此可以有效地使用磁盘，并且文件系统的性能接近其峰值。

关键问题：如何让所有写入变成顺序写入？

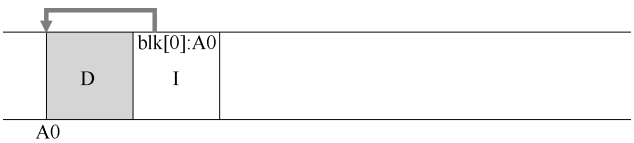
文件系统如何将所有写入转换为顺序写入？对于读取，此任务是不可能的，因为要读取的所需块可能是磁盘上的任何位置。但是，对于写入，文件系统总是有一个选择，而这正是我们希望利用的选择。

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。

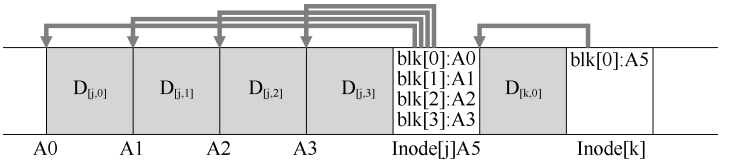
43.2 顺序而高效地写入

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。

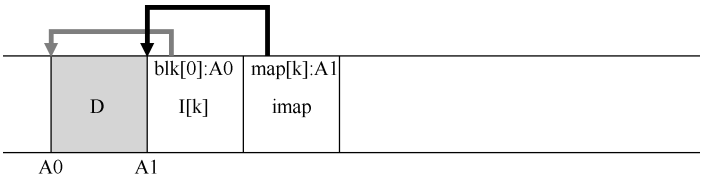
提示：使用一个间接层

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。

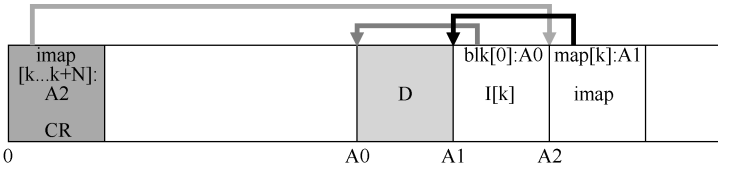
43.6 检查点区域

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。



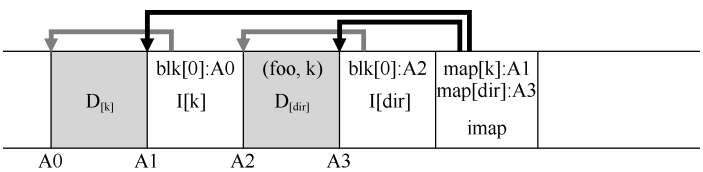
43.7 从磁盘读取文件：回顾

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。

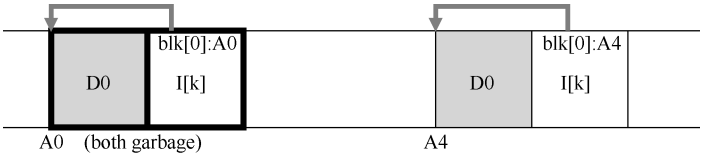
具体来说，每当更新 inode 时，它在磁盘上的位置都会发生变化。如果我们不小心，这也会导致对指向该文件的目录的更新，然后必须更改该目录的父目录，依此类推，一路沿文件系统树向上。

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

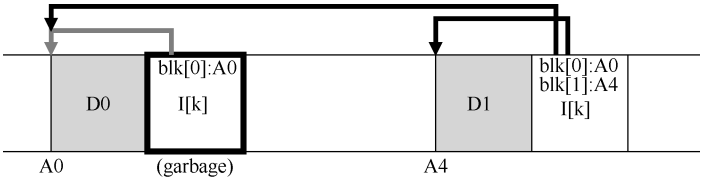
你可能已经注意到 LFS 的另一个问题；它会反复将最新版本的文件（包括其 inode 和数据）写入磁盘上的新位置。此过程在保持写入效率的同时，意味着 LFS 会在整个磁盘中分散旧版本的文件结构。我们（毫不客气地）将这些旧版本称为垃圾（garbage）。

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。