Journaling the Linux ext2fs Filesystem

Stephen C. Tweedie

sct@dcs.ed.ac.uk

Abstract

This paper describes a work-in-progress to design and implement a transactional metadata journal for the Linux ext2fs filesystem. We review the problem of recovering filesystems after a crash, and describe a design intended to increase ext2fs's speed and reliability of crash recovery by adding a transactional journal to the filesystem.

Introduction

Filesystems are central parts of any modern operating system, and are expected to be both fast and exceedingly reliable. However, problems still occur, and machines can go down unexpectedly, due to hardware, software or power failures.

After an unexpected reboot, it may take some time for a system to recover its filesystems to a consistent state. As disk sizes grow, this time can become a serious problem, leaving a system offline for an hour or more as the disk is scanned, checked and repaired. Although disk drives are becoming faster each year, this speed increase is modest compared with their enormous increase in capacity. Unfortunately, every doubling of disk capacity leads to a doubling of recovery time when using traditional filesystem checking techniques.

Where system availability is important, this may be time which cannot be spared, so a mechanism is required which will avoid the need for an expensive recovery stage every time a machine reboots.

What's in a filesystem?

What functionality do we require of any filesystem? There are obvious requirements which are dictated by the operating system which the filesystem is serving. The way that the filesystem appears to applications is one−operating systems typically require that filenames adhere to certain conventions and that files possess certain attributes which are interpreted in a specific way.

However, there are many internal aspects of a filesystem which are not so constrained, and which a filesystem implementor can design with a certain amount of freedom. The layout of data on disk (or alternatively, perhaps, its network protocol, if the filesystem is not local), details of internal caching, and the algorithms used to schedule disk IO−these are all things which can be changed without necessarily violating the specification of the filesystem's application interface.

There are a number of reasons why we might choose one design over another. Compatibility with older filesystems might be an issue: for example, Linux provides a UMSDOS filesystem which implements the semantics of a POSIX filesystem on top of the standard MSDOS on-disk file structure.

When trying to address the problem of long filesystem recovery times on Linux, we kept a number of goals in mind:

- Performance should not suffer seriously as a result of using the new filesystem;
- Compatibility with existing applications must not be broken
- The reliability of the filesystem must not be compromised in any way.

记录 Linux ext2fs 文件系统 Stephen C. Tweedie sct@dcs.ed.ac.uk

摘要

本文描述了一项正在进行中的工作，旨在为 Linux ext2fs 文件系统设计和实现一个事务性元数据日志。我们回顾了在崩溃后恢复文件系统的问题，并描述了一种设计，通过向文件系统添加事务性日志来提高 ext2fs 的速度和崩溃恢复的可靠性。

引言

文件系统是任何现代操作系统的重要组成部分，人们期望它们既快速又极其可靠。然而，问题仍然会发生，由于硬件、软件或电源故障，机器可能会意外停机。

意外重启后，系统可能需要一些时间才能将其文件系统恢复到一致状态。随着磁盘容量的增长，这段时间可能会成为一个严重的问题，导致系统离线一个多小时，因为磁盘正在被扫描、检查和修复。尽管磁盘驱动器每年都在变快，但与它们容量的巨大增长相比，这种速度提升微不足道。不幸的是，当使用传统的文件系统检查技术时，磁盘容量的每次翻倍都会导致恢复时间也翻倍。

在系统可用性很重要的情况下，这可能是一些无法节省的时间，因此需要一种机制，以避免每次机器重启时都需要进行昂贵的恢复阶段。

文件系统中有哪些内容?

我们对任何文件系统需要哪些功能？操作系统为文件系统提供的服务有明显的需求。文件系统对应用程序的表现方式是-操作系统通常要求文件名遵循某些规范，并且文件需要拥有某些属性，这些属性以特定方式被解释。

文件需要拥有某些属性，这些属性以特定方式被解释。

然而，文件系统的许多内部方面并没有那么严格限制，并且文件系统实现者可以在一定程度上自由设计。磁盘上的数据布局（或者，也许，如果文件系统不是本地的，则是其网络协议），内部缓存的细节，以及用于调度磁盘IO-这些都是可以在不必然违反文件系统应用接口规范的情况下进行更改的事情。

我们可能会选择一种设计而不是另一种设计，原因有很多。与旧文件系统的兼容性可能是一个问题：例如，Linux 提供了一个 UMSDOS 文件系统，它在标准的 MSDOS 磁盘文件结构之上实现了 POSIX 文件系统的语义。

在尝试解决 Linux 系统中文件恢复时间长的问题时，我们始终牢记以下几个目标：

- 使用新的文件系统不应导致性能严重下降；
- 与现有应用程序的兼容性不能被破坏
- 文件系统的可靠性在任何方面都不能被妥协。

Filesystem Reliability

There are a number of issues at stake when we talk about filesystem reliability. For the purpose of this particular project, we are interested primarily in the reliability with which we can recover the contents of a crashed filesystem, and we can identify several aspects of this:

Preservation: data which was stable on disk before the crash should never ever be damaged. Obviously, files which were being written out at the time of the crash cannot be guaranteed to be perfectly intact, but any files which were already safe on disk must not be touched by the recovery system.

Predictability: the failure modes from which we have to recover should be predictable in order for us to recover reliably.

Atomicity: many filesystem operations require a significant number of separate IOs to complete. A good example is the renaming of a file from one directory to another. Recovery is atomic if such filesystem operations are either fully completed on disk or fully undone after recovery finishes. (For the rename example, recovery should leave either the old or the new filename committed to disk after a crash, but not both.)

Existing implementations

The Linux ext2fs filesystem offers preserving recovery, but it is non-atomic and unpredictable. Predictability is in fact a much more complex property than appears at first sight. In order to be able to predictably mop up after a crash, the recovery phase must be able to work out what the filesystem was trying to do at the time if it comes across an inconsistency representing an incomplete operation on the disk. In general, this requires that the filesystem must make its writes to disk in a predictable order whenever a single update operation changes multiple blocks on disk.

There are many ways of achieving this ordering between disk writes. The simplest is simply to wait for the first writes to complete before submitting the next ones to the device driver−the ‘‘synchronous metadata update’’ approach. This is the approach taken by the BSD Fast File System[1], which appeared in 4.2BSD and which has inspired many of the Unix filesystems which followed, including ext2fs.

However, the big drawback of synchronous metadata update is its performance. If filesystems operation require that we wait for disk IO to complete, then we cannot batch up multiple filesystem updates into a single disk write. For example, if we create a dozen directory entries in the same directory block on disk, then synchronous updates require us to write that block back to disk a dozen separate times.

There are ways around this performance problem. One way to keep the ordering of disk writes without actually waiting for the IOs to complete is to maintain an ordering between the disk buffers in memory, and to ensure that when we do eventually go to write back the data, we never write a block until all of its predecessors are safely on disk−the ‘‘deferred ordered write’’ technique.

One complication of deferred ordered writes is that it is easy to get into a situation where there are cyclic dependencies between cached buffers. For example, if we try to rename a file between two directories and at the same time rename another file from the second directory into the first, then we end up with a situation where both directory blocks depend on each other: neither can be written until the other one is on disk.

Ganger’s ‘‘soft updates’’ mechanism[2] neatly sidesteps this problem by selectively rolling back specific updates within a buffer if those updates still have outstanding dependencies when we first try to write that buffer out to disk. The missing update will be restored later once all of its own dependencies are satisfied. This allows us to write out buffers in any order we choose when there are circular dependencies. The soft update mechanism has been adopted by FreeBSD and will be available as part of their next major kernel version.

All of these approaches share a common problem, however. Although they ensure that the state of the disk is in a predictable state all the way through the course of a filesystem operation, the recovery process still has to scan the entire disk in order to find and repair any uncompleted operations. Recovery becomes more reliable, but is not necessarily any faster.

It is, however, possible to make filesystem recovery fast without sacrificing reliability and predictability. This is typically done by filesystems which guarantee atomic completion of filesystem updates (a single filesystem update is usually referred to as a *transaction* in such systems). The basic principle

文件系统可靠性

当讨论文件系统可靠性时，涉及许多问题。就这个特定项目而言，我们主要关注的是能够以多高的可靠性恢复崩溃文件系统中的内容，并且可以识别出几个相关方面：

保存：在崩溃前磁盘上稳定的数 据永远不应该被损坏。显然，在崩溃时正在写入的文件无法保证完全完好无损，但任何在磁盘上已经安全的文件都必须不被恢复系统触碰。

可预测性：我们必须从可预测的故障模式中恢复，才能可靠地恢复。

原子性：许多文件系统操作需要大量独立的 I/O 操作才能完成。一个很好的例子是将文件从一个目录重命名为另一个目录。如果文件系统操作在磁盘上完全完成或在恢复完成后完全撤销，则恢复是原子的。（以重命名为例，在崩溃后，恢复应该将旧文件名或新文件名中的一个提交到磁盘，但不能两者都提交。）

现有实现

Linux ext2fs文件系统提供了可恢复性，但它不是原子性的，且不可预测。可预测性实际上比表面看起来要复杂得多。为了能够在崩溃后可预测地清理，恢复阶段必须能够推断出文件系统在遇到表示磁盘上不完整操作的冲突时当时试图做什么。通常，这要求文件系统在每次单个更新操作更改多个磁盘块时，以可预测的顺序将写入写入磁盘。

实现这种磁盘写入顺序的方法有很多。最简单的方法就是等待第一批写入完成后再将后续的写入提交给设备驱动程序-，即“同步元数据更新”方法。这是BSD快速文件系统[1], 采用的方法，该系统出现在4.2BSD版本中，并启发了许多后续的Unix文件系统，包括ext2fs。

然而，同步元数据更新的主要缺点是性能问题。如果文件系统操作需要我们等待磁盘IO完成，那么我们就无法将多个文件系统更新合并为一次磁盘写入。例如，如果在磁盘的同一目录块中创建十几个目录条目，那么同步更新就需要我们将该块单独写回磁盘十几次。

有办法可以绕过这个问题。一种在不实际等待IO完成的情况下保持磁盘写入顺序的方法是，在内存中的磁盘缓冲区之间维护一个顺序，并确保当我们最终要写回数据时，永远不会在它的所有前驱都安全地写入磁盘之前写入一个块-即“延迟有序写入”技术。

延迟有序写入的一个复杂问题是，很容易出现缓存缓冲区之间存在循环依赖的情况。例如，如果我们尝试在两个目录之间重命名一个文件，同时将第二个目录中的另一个文件重命名为第一个目录，那么最终会得到一个两个目录块相互依赖的情况：任何一个都不能在另一个写入磁盘之前完成写入。

Ganger的“软更新”机制[2] 通过在尝试将缓冲区写入磁盘时，如果这些更新在首次尝试写入时仍然存在未完成的依赖关系，就选择性地回滚缓冲区中的特定更新，巧妙地绕过了这个问题。缺失的更新将在其所有依赖关系都满足后稍后恢复。这允许我们在存在循环依赖时按任意顺序写入缓冲区。软更新机制已被FreeBSD采用，并将作为其下一个主要内核版本的一部分提供。

然而，所有这些方法都存在一个共同问题。尽管它们确保了磁盘状态在整个文件系统操作过程中都处于可预测的状态，但恢复过程仍然需要扫描整个磁盘，以查找和修复任何未完成的操作。恢复变得更加可靠，但不一定更快。

然而，可以在不牺牲可靠性和可预测性的前提下，使文件系统恢复变得快速。这通常是通过那些保证文件系统更新原子完成的文件系统来实现的（在这样系统中，单个文件系统更新通常被称为事务）。基本原理

behind atomic updates is that the filesystem can write an entire batch of new data to disk, but that those updates do not take effect until a final, *commit* update is made on the disk. If the commit involves a write of a single block to the disk, then a crash can only result in two cases: either the commit record has been written to disk, in which case all of the committed filesystem operations can be assumed to be complete and consistent on disk; or the commit record is missing, in which case we have to ignore any of the other writes which occurred due to partial, uncommitted updates still outstanding at the time of the crash. This naturally requires a filesystem update to keep both the old and new contents of the updated data on disk somewhere, right up until the time of the commit.

There are a number of ways of achieving this. In some cases, filesystems keep the new copies of the updated data in different locations from the old copies, and eventually reuse the old space once the updates are committed to disk. Network Appliance's WAFL filesystem[6] works this way, maintaining a tree of filesystem data which can be updated atomically simply by copying tree nodes to new locations and then updating a single disk block at the root of the tree.

Log-Structured Filesystems achieve the same end by writing *all* filesystem data−both file contents and metadata−to the disk in a continuous stream (the ''log''). Finding the location of a piece of data using such a scheme can be more complex than in a traditional filesystem, but logs have the big advantage that it is relatively easy to place marks in the log to indicate that all data up to a certain point is committed and consistent on disk. Writing to such a filesystem is also particularly fast, since the nature of the log makes most writes occur in a continuous stream with no disk seeks. A number of filesystems have been written based on this design, including the Sprite LFS[3] and the Berkeley LFS[4]. There is also a prototype LFS implementation on Linux[5].

Finally, there is a class of atomically-updated filesystems in which the old and new versions of incomplete updates are preserved by writing the new versions to a separate location on disk until such time as the update has be committed. After commit, the filesystem is free to write the new versions of the updated disk blocks back to their home locations on disk.

This is the way in which ''journaling'' (sometimes referred to as ''log enhanced'') filesystems work. When metadata on the disk is updated, the updates are recorded in a separate area of the disk reserved for use as a journal. Filesystem transactions which complete have a commit record added to the journal, and only after the commit is safely on disk may the filesystem write the metadata back to its original location. Transactions are atomic because we can always either undo a transaction (throw away the new data in the journal) or redo it (copy the journal copy back to the original copy) after a crash, according to whether or not the journal contains a commit record for the transaction. Many modern filesystems have adopted variations on this design.

Designing a new filesystem for Linux

The primary motivation behind this new filesystem design for Linux was to eliminate enormously long filesystem recovery times after a crash. For this reason, we chose a filesystem journaling scheme as the basis for the work. Journaling achieves fast filesystem recovery because at all times we know that all data which is potentially inconsistent on disk must be recorded also in the journal. As a result, filesystem recovery can be achieved by scanning the journal and copying back all committed data into the main filesystem area. This is fast because the journal is typically very much smaller than the full filesystem. It need only be large enough to record a few seconds-worth of uncommitted updates.

The choice of journaling has another important advantage. A journaling filesystem differs from a traditional filesystem in that it keeps transient data in a new location, independent of the permanent data and metadata on disk. Because of this, such a filesystem does not dictate that the permanent data has to be stored in any particular way. In particular, it is quite possible for the ext2fs filesystem's on-disk structure to be used in the new filesystem, and for the existing ext2fs code to be used as the basis for the journaling version.

As a result, we are not designing a new filesystem for Linux. Rather, we are adding a new feature−transactional filesystem journaling−to the existing ext2fs.

原子更新的背后原理是，文件系统可以将一批新数据写入磁盘，但这些更新只有在磁盘上执行最终提交更新后才会生效。如果提交涉及向磁盘写入单个块，那么崩溃只可能导致两种情况：要么提交记录已被写入磁盘，在这种情况下，可以假定所有已提交的文件系统操作在磁盘上都是完整且一致的；要么提交记录丢失，在这种情况下，我们必须忽略由于部分未提交更新在崩溃时仍然未完成的任何其他写入。这自然要求文件系统更新在提交时间之前，将更新数据的新旧内容都保留在磁盘上的某个地方。

有几种方法可以实现这一点。在某些情况下，文件系统会将更新数据的新副本保存在与旧副本不同的位置，并在更新提交到磁盘后最终重用旧空间。Network Appliance 的 WAFL 文件系统[6] 就是这种方式，它维护一个文件系统数据的树状结构，只需将树节点复制到新位置并更新树根的一个磁盘块即可原子性地更新数据。

日志结构文件系统通过将所有文件系统数据−（包括文件内容和元数据−）以连续流的形式写入磁盘（即“日志”）来实现相同的目的。使用这种方案查找数据的位置可能比传统文件系统更复杂，但日志的一大优势是相对容易在日志中放置标记，以指示所有数据在某个点之前都已提交且在磁盘上保持一致。写入这种文件系统也特别快，因为日志的特性使得大多数写入都发生在连续的流中，无需磁盘寻道。基于这种设计已编写了多种文件系统，包括 Sprite LFS[3] 和 Berkeley LFS[4]。Linux[5]上也有一个 LFS 原型实现。

最后，存在一类原子更新文件系统，其中旧版本和新版本的未完成更新通过将新版本写入磁盘上的单独位置来保留，直到更新被提交为止。提交后，文件系统可以自由地将更新后的磁盘块的新版本写回磁盘上的原始位置。

这就是‘‘日志记录’’（有时也称为‘‘日志增强’’）文件系统的工作方式。当磁盘上的元数据被更新时，更新会被记录在磁盘上为日志保留的单独区域中。完成的文件系统事务会在日志中添加提交记录，并且只有当提交安全地写入磁盘后，文件系统才会将元数据写回其原始位置。事务是原子的，因为无论日志是否包含该事务的提交记录，我们都可以在崩溃后撤销事务（丢弃日志中的新数据）或重做事务（将日志副本复制回原始副本）。许多现代文件系统都采用了这种设计的变体。

为 Linux 设计新的文件系统

Linux 新文件系统设计的主要动机是消除崩溃后极其漫长的文件系统恢复时间。为此，我们选择文件系统日志记录方案作为工作基础。日志记录之所以能实现快速文件系统恢复，是因为我们始终知道所有可能不一致的磁盘数据都必须记录在日志中。因此，文件系统恢复可以通过扫描日志并将所有已提交数据复制回主文件系统区域来实现。这很快，因为日志通常远小于整个文件系统。它只需要足够大，能够记录几秒钟的未提交更新。

日志记录的选择还有另一个重要优势。日志记录文件系统与传统文件系统的区别在于，它将临时数据保存在新位置，独立于磁盘上的永久数据和元数据。由于这个原因，这种文件系统并不强制永久数据必须以任何特定方式存储。特别是，ext2fs 文件系统的磁盘结构完全可以用于新文件系统，现有的 ext2fs 代码可以作为日志记录版本的基础。

因此 ，我们并不需要 gning 一个新的文件系统 for Linux。相反， we are 我们只是 a new 特性-事务性文件系统日志记录-到
添加现有的 ext2fs。

Anatomy of a transaction

A central concept when considering a journaled filesystem is the transaction, corresponding to a single update of the filesystem. Exactly one transaction results from any single filesystem request made by an application, and contains all of the changed metadata resulting from that request. For example, a write to a file will result in an update to the modification timestamp in the file's inode on disk, and may also update the length information and the block mapping information if the file is extended by the write. Quota information, free disk space and used block bitmaps will all have to be updated if new blocks are allocated to the file, and all this must be recorded in the transaction.

There is another hidden operation in a transaction which we have to be aware about. Transactions also involve reading the existing contents of the filesystem, and that imposes an ordering between transactions. A transaction which modifies a block on disk cannot commit after a transaction which reads that new data and then updates the disk based on what it read. The dependency exists even if the two transactions do not ever try to write back the same blocks−imagine one transaction deleting a filename from one block in a directory and another transaction inserting the same filename into a different block. The two operations may not overlap in the blocks which they write, but the second operation is only valid after the first one succeeds (violating this would result in duplicate directory entries).

Finally, there is one ordering requirement which goes beyond ordering between metadata updates. Before we can commit a transaction which allocates new blocks to a file, we have to make absolutely sure that all of the data blocks being created by the transaction have in fact been written to disk (we term these data blocks *dependent data*). Missing out this requirement would not actually damage the integrity of the filesystem's metadata, but it could potentially lead to a new file still containing a previous file contents after crash recovery, which is a security risk as well as being a consistency problem.

Merging transactions

Much of the terminology and technology used in a journaled filesystem comes from the database world, where journaling is a standard mechanism for ensuring atomic commits of complex transactions. However, there are many differences between the traditional database transaction and a filesystem, and some of these allow us to simplify things enormously.

Two of the biggest differences are that filesystems have no transaction abort, and all filesystem transactions are relatively short-lived. Whereas in a database we sometimes want to abort a transaction halfway through, discarding any changes we have made so far, the same is not true in ext2fs−by the time we start making any changes to the filesystem, we have already checked that the change can be completed legally. Aborting a transaction before we have started writing changes (for example, a create file operation might abort if it finds an existing file of the same name) poses no problem since we can in that case simply commit the transaction with no changes and achieve the same effect.

The second difference−the short life term of filesystem transactions−is important since it means that we can simplify the dependencies between transactions enormously. If we have to cater for some very long-term transactions, then we need to allow transactions to commit independently in any order as long as they do not conflict with each other, as otherwise a single stalled transaction could hold up the entire system. If all transactions are sufficiently quick, however, then we can require that transactions commit to disk in strict sequential order without significantly hurting performance.

With this observation, we can make a simplification to the transaction model which can reduce the complexity of the implementation substantially while at the same time increasing performance. Rather than create a separate transaction for each filesystem update, we simply create a new transaction every so often, and allow all filesystem service calls to add their updates to that single system-wide compund transaction.

There is one great advantages of this mechanism. Because all operations within a compound transaction will be committed to the log together, we do not have to write separate copies of any metadata blocks which are updated very frequently. In particular, this helps for operations such as creating new files, where typically every write to the file results in the file being extended, thus updating the same quota, bitmap blocks and inode blocks continuously. Any block which is updated many times during the life of a compound transaction need only be committed to disk once.

事务的解剖

在考虑日志记录文件系统时，一个核心概念是事务，它对应于文件系统的一次更新。任何应用程序发起的任何单个文件系统请求都会产生一个事务，该事务包含由该请求导致的所有更改的元数据。例如，向文件写入内容会导致磁盘上文件inode的修改时间戳更新，如果文件因写入而扩展，还可能更新文件长度信息和块映射信息。如果向文件分配新块，配额信息、可用磁盘空间和使用块位图都将需要更新，所有这些都必须记录在事务中。

事务中还有一个隐藏的操作需要我们注意。事务也涉及读取现有文件系统的内容，这会在事务之间建立顺序关系。修改磁盘上块的交易不能在读取该新数据并基于读取内容更新磁盘的事务之后提交。即使两个事务永远不会尝试写回相同的块，依赖关系也存在-想象一个事务从一个目录块的某个块中删除一个文件名，而另一个事务将同一个文件名插入到不同的块中。这两个操作在它们写入的块中可能不会重叠，但第二个操作只有在第一个操作成功后才是有效的（违反这一点会导致目录条目重复）。

最后，有一个顺序要求超出了元数据更新之间的顺序关系。在我们能够提交一个为文件分配新块的事务之前，我们必须绝对确保事务创建的所有数据块实际上都已写入磁盘（我们将这些数据块称为依赖数据）。遗漏这一要求实际上不会损害文件系统元数据的完整性，但它可能会在崩溃恢复后导致新文件仍然包含先前文件的内容，这既是一个安全风险，也是一个一致性问题。

合并事务

一个日志文件系统的许多术语和技术都源自数据库领域，在那里，日志记录是确保复杂事务原子提交的标准机制。然而，传统数据库事务和文件系统之间存在许多差异，其中一些差异使我们能够极大地简化问题。

传统数据库事务和文件系统之间存在许多差异，其中一些差异使我们能够极大地简化问题。

最大的两个区别在于：文件系统没有事务中止功能，且所有文件系统事务都是相对短暂的。而在数据库中，我们有时希望在中途中止事务，丢弃已做的任何更改，但在 ext2fs−中并非如此。一旦我们开始对文件系统进行任何更改，就已经检查过该更改可以合法完成。在我们开始写入更改之前中止事务（例如，如果创建文件操作发现存在同名文件，可能会中止）不会构成问题，因为在这种情况下，我们可以简单地提交一个无更改的事务，达到相同的效果。

第二个区别-文件系统事务的短暂生命周期-非常重要，因为它意味着我们可以极大地简化事务之间的依赖关系。如果我们需要支持一些非常长期的事务，那么我们需要允许事务以任何顺序独立提交，只要它们不相互冲突，否则单个停滞的事务可能会使整个系统瘫痪。然而，如果所有事务都足够快，那么我们可以要求事务以严格的顺序提交到磁盘，而不会显著影响性能。

基于这一观察，我们可以对交易模型进行简化，从而大幅降低实现复杂度，同时提升性能。与其为每个文件系统更新创建单独的交易，我们只需定期创建一个新的交易，并允许所有文件系统服务调用将其更新添加到这个单一的、系统级的复合交易中。

这种机制有一个巨大的优势。由于复合事务中的所有操作都会一起提交到日志中，我们不必为频繁更新的任何元数据块编写单独的副本。特别是，这有助于创建新文件等操作，通常每次写入文件都会导致文件扩展，从而持续更新相同的配额、位图块和索引节点块。在复合事务的生命周期中多次更新的任何块只需提交到磁盘一次。

The decision about when to commit the current compound transaction and start a new one is a policy decision which should be under user control, since it involves a trade-off which affects system performance. The longer a commit waits, the more filesystem operations can be merged together in the log and so less IO operations are required in the long term. However, longer commits tie up larger amounts of memory and disk space, and leave a larger window for loss of updates if a crash occurs. They may also lead to storms of disk activity which make filesystem response times less predictable.

On-disk representation

The layout of the journaled ext2fs filesystem on disk will be entirely compatible with existing ext2fs kernels. Traditional UNIX filesystems store data on disk by associating each file with a unique numbered *inode* on the disk, and the ext2fs design already includes a number of reserved inode numbers. We use one of these reserved inodes to store the filesystem journal, and in all other respects the filesystem will be compatible with existing Linux kernels. The existing ext2fs design includes a set of compatibility bitmaps, in which bits can be set to indicate that the filesystem uses certain extensions. By allocating a new compatibility bit for the journaling extension, we can ensure that even though old kernels will be able to successfully mount a new, journaled ext2fs filesystem, they will not be permitted to write to the filesystem in any way.

Format of the filesystem journal

The journal file's job is simple: it records the new contents of filesystem metadata blocks while we are in the process of committing transactions. The only other requirement of the log is that we must be able to atomically commit the transactions it contains.

We write three different types of data blocks to the journal: metadata, descriptor and header blocks.

A journal metadata block contains the entire contents of a single block of filesystem metadata as updated by a transaction. This means that however small a change we make to a filesystem metadata block, we have to write an entire journal block out to log the change. However, this turns out to be relatively cheap for two reasons:

- Journal writes are quite fast anyway, since most writes to the journal are sequential, and we can easily batch the journal IOs into large clusters which can be handled efficiently by the disk controller;
- By writing out the entire contents of the changed metadata buffer from the filesystem cache to the journal, we avoid having to do much CPU work in the journaling code.

The Linux kernel already provides us with a very efficient mechanism for writing out the contents of an existing block in the buffer cache to a different location on disk. Every buffer in the buffer cache is described by a structure known as a *buffer_head*, which includes information about which disk block the buffer's data is to be written to. If we want to write an entire buffer block to a new location without disturbing the buffer_head, we can simply create a new, temporary buffer_head into which we copy the description from the old one, and then edit the device block number field in the temporary buffer head to point to a block within the journal file. We can then submit the temporary buffer_head directly to the device IO system and discard it once the IO is complete.

Descriptor blocks are journal blocks which describe other journal metadata blocks. Whenever we want to write out metadata blocks to the journal, we need to record which disk blocks the metadata normally lives at, so that the recovery mechanism can copy the metadata back into the main filesystem. A descriptor block is written out before each set of metadata blocks in the journal, and contains the number of metadata blocks to be written plus their disk block numbers.

Both descriptor and metadata blocks are written sequentially to the journal, starting again from the start of the journal whenever we run off the end. At all times, we maintain the current head of the log (the block number of the last block written) and the tail (the oldest block in the log which has not been unpinned, as described below). Whenever we run out of log space−the head of the log has looped back round and caught up with the tail−we stall new log writes until the tail of the log has been cleaned up to free more space.

Finally, the journal file contains a number of header blocks at fixed locations. These record the current head and tail of the journal, plus a sequence number. At recovery time, the header blocks are scanned to find the block with the highest sequence number, and when we scan the log during recovery we just

关于何时提交当前复合事务并开始新事务的决定是一项政策决策，应受用户控制，因为它涉及影响系统性能的权衡。提交等待时间越长，日志中可以合并的文件系统操作就越多，因此从长远来看所需的 I/O 操作就越少。然而，较长的提交会占用更多的内存和磁盘空间，并且在发生崩溃时留下更大的更新丢失窗口。它们还可能导致磁盘活动激增，使文件系统响应时间更不可预测。

磁盘表示

日志 ext2fs 文件系统在磁盘上的布局将与现有的 ext2fs 内核完全兼容。传统的 UNIX 文件系统通过将每个文件与磁盘上的一个唯一编号的 i node 关联来存储数据，而 ext2fs 设计已经包含了一些保留的 inode 编号。我们使用其中一个保留的 inode 来存储文件系统的日志，并且在其他所有方面，文件系统将与现有的 Linux 内核兼容。现有的 ext2fs 设计包含一组兼容性位图，其中位可以被设置为指示文件系统使用某些扩展。通过为日志扩展分配一个新的兼容性位，我们可以确保即使旧的内核能够成功挂载新的、带日志的 ext2fs 文件系统，它们也不会被允许以任何方式写入文件系统。

文件系统日志的格式

日志文件的任务很简单：在提交事务的过程中，它记录文件系统元数据块的新内容。日志的另一个要求是，我们必须能够原子性地提交其中包含的事务。

我们将三种不同类型的数据块写入日志：元数据块、描述符块和头块。

一个日志元数据块包含一个文件系统元数据块的全部内容，该内容由事务更新。这意味着无论我们对文件系统元数据块做出多么微小的更改，都必须将整个日志块写入日志以记录该更改。然而，事实证明这相对便宜，原因有两个：

- 日志写入速度很快，因为大多数写入日志的操作都是顺序的，而且我们可以轻松地将日志IO批量处理成大型簇。
- 通过将文件系统缓存中已更改的元数据缓冲区全部内容写入日志，我们避免了在日志代码中进行大量CPU工作。

Linux内核已经为我们提供了一种非常高效的机制，用于将缓冲区缓存中现有块的内容写入磁盘上的不同位置。缓冲区缓存中的每个缓冲区都由一个称为buffer_head的结构描述，其中包含有关缓冲区数据要写入哪个磁盘块的信息。如果我们想在不干扰buffer_head的情况下将整个缓冲区块写入新位置，我们可以简单地创建一个新的临时buffer_head，将旧缓冲区的描述复制到其中，然后编辑临时缓冲区头中的设备块号字段，使其指向日志文件中的一个块。然后我们可以将临时buffer_head直接提交给设备IO系统，并在IO完成后将其丢弃。

描述符块是用于描述其他元数据块的日志块。每当我们要将元数据块写入日志时，都需要记录元数据通常位于哪些磁盘块上，以便恢复机制可以将元数据复制回主文件系统。每个日志中的元数据块集之前都会写入一个描述符块，其中包含要写入的元数据块数量及其磁盘块号。

描述符块和元数据块都按顺序写入日志，当写入到日志末尾时，会重新从日志开头开始。我们始终维护当前日志的头部（最后写入的块号）和尾部（日志中尚未被解除固定的最老块，如下所述）。每当日志空间不足-日志头部已经循环回绕并追上尾部-时，我们会暂停新的日志写入，直到日志尾部被清理以释放更多空间。

最后，日志文件在固定位置包含多个头部块。这些块记录了当前日志的头部和尾部，以及一个序列号。在恢复时，会扫描这些头部块以找到序列号最高的块，而在恢复过程中扫描日志时，我们只需按照该头部块中记录的顺序，从尾部到头部遍历所有日志块。

run through all journal blocks from the tail to the head, as recorded in that header block.

Committing and checkpointing the journal

At some point, either because we have waited long enough since the last commit or because we are running short of space in the journal, we will wish to commit our outstanding filesystem updates to the log as a new compound transaction.

Once the compound transaction has been completely committed, we are still not finished with it. We need to keep track of the metadata buffers recorded in a transaction so that we can notice when they get written back to their main locations on disk.

Recall that when we commit a transaction, the new updated filesystem blocks are sitting in the journal but have not yet been synced back to their permanent home blocks on disk (we need to keep the old blocks unsynced in case we crash before committing the journal). Once the journal has been committed, the old version on the disk is no longer important and we can write back the buffers to their home locations at our leisure. However, until we have finished syncing those buffers, we cannot delete the copy of the data in the journal.

To completely commit and finish checkpointing a transaction, we go through the following stages:

1. Close the transaction. At this point we make a new transaction in which we will record any filesystem operations which begin in the future. Any existing, incomplete operations will still use the existing transaction: we cannot split a single filesystem operation over multiple transactions!

2. Start flushing the transaction to disk. In the context of a separate log-writer kernel thread, we begin writing out to the journal all metadata buffers which have been modified by the transaction. We also have to write out any dependent data at this stage (see the section above, Anatomy of a transaction).

   When a buffer has been committed, mark it as pinning the transaction until it is no longer dirty (it has been written back to the main storage by the usual writeback mechanisms).

3. Wait for all outstanding filesystem operations in this transaction to complete. We can safely start writing the journal before all operations have completed, and it is faster to allow these two steps to overlap to some extent.

4. Wait for all outstanding transaction updates to be completely recorded in the journal.

5. Update the journal header blocks to record the new head and tail of the log, committing the transaction to disk.

7. When we wrote the transaction's updated buffers out to the journal, we marked them as pinning the transaction in the journal. These buffers become unpinned only when they have been synced to their homes on disk. Only when the transaction's last buffer becomes unpinned can we reuse the journal blocks occupied by the transaction. When this occurs, write another set of journal headers recording the new position of the tail of the journal. The space released in the journal can now be reused by a later transaction.

Collisions between transactions

To increase performance, we do not completely suspend filesystem updates when we are committing a transaction. Rather, we create a new compound transaction in which to record updates which arrive while we commit the old transaction.

This leaves open the question of what to do if an update wants access to a metadata buffer already owned by another, older transaction which is currently being committed. In order to commit the old transaction we need to write its buffer to the journal, but we cannot include in that write any changes which are not part of the transaction, as that would allow us to commit incomplete updates.

If the new transaction only wants to read the buffer in question, then there is no problem: we have created a read/write dependency between the two transactions, but since compound transactions always commit in strict sequential order we can safely ignore the collision.

Things are more complicated if the new transaction wants to write to the buffer. We need the old copy of the buffer to commit the first transaction, but we cannot let the new transaction proceed without letting it modify the buffer.

The solution here is to make a new copy of the buffer in such cases. One copy is given to the new transaction for modification. The other is left owned

按照该头部块中记录的顺序，从尾部到头部遍历所有日志块。

提交和检查点日志

在某个时刻，无论是由于距离上次提交等待了足够长的时间，还是因为日志空间不足，我们都希望将未完成的文件系统更新提交到日志中，作为一个新的复合事务。

复合事务完全提交后，我们仍然没有完成它。我们需要跟踪事务中记录的元数据缓冲区，以便在它们被写回磁盘上的主要位置时能够注意到。

回想一下，当我们提交一个事务时，新的更新文件系统块位于日志中，但尚未同步回磁盘上的永久家块（我们需要保持旧块未同步，以防我们在提交日志之前崩溃）。一旦日志被提交，磁盘上的旧版本就不再重要了，我们可以随时将缓冲区写回它们的家位置。然而，在我们完成同步这些缓冲区之前，我们不能删除日志中的数据副本。

要完全提交并完成事务的检查点，我们会经历以下阶段：

1. 关闭交易。此时我们将创建一个新的交易，用于记录未来开始的任何文件系统操作。任何现有的、未完成的操作仍将使用现有的交易：我们无法将单个文件系统操作拆分到多个交易中！

2. 开始将事务刷新到磁盘。在单独的日志写入器内核线程的上下文中，我们开始将所有被事务修改的元数据缓冲区写入日志。我们还需要在这个阶段写入任何依赖数据（参见上文，“事务的解剖结构”部分）。

   当缓冲区已提交时，将其标记为锁定事务，直到它不再脏（它已通过常规的回写机制写回主存储）。

3. 等待此事务中所有未完成的文件系统操作完成。我们可以安全地

在所有操作完成前就开始写日记，并且允许这些更快
这两个步骤在一定程度上有重叠。

4. 等待所有未处理的交易更新完全记录在账簿中。

5. 更新日志头块以记录新日志的头和尾，并将事务提交到磁盘。

7. 当我们写入事务更新后的buffers到日志时，我们将它们标记为在日志中固定事务。这些缓冲区只有在同步到磁盘上的家时才会被解除固定。只有当事务的最后一个缓冲区被解除固定时，我们才能重用事务占用的日志块。当这种情况发生时，写入另一组日志头以记录日志尾的新位置。日志中释放的空间现在可以被后续事务重用。

事务之间的冲突

为了提高性能，当我们提交事务时，我们不会完全暂停文件系统更新。相反，我们会创建一个新的复合事务，在其中记录在提交旧事务期间到达的更新。

这留下了一个问题：如果一个更新想要访问由另一个较旧的事务当前正在提交时拥有的元数据缓冲区，该怎么办。为了提交旧事务，我们需要将它的缓冲区写入日志，但我们不能在这次写入中包含任何不属于事务的更改，因为那样将允许我们提交不完整的更新。

如果新事务只想读取该缓冲区，那么没有问题：我们已经在两个事务之间创建了读写依赖关系，但由于复合事务总是以严格的顺序提交，因此我们可以安全地忽略冲突。

如果新事务想要写入该缓冲区，情况就复杂了。我们需要旧版本的缓冲区来提交第一个事务，但我们不能在不允许它修改缓冲区的情况下让新事务继续进行。

这里的解决方案是在这种情况下创建缓冲区的新副本。一个副本交给新的事务进行修改。另一个则保留所有权

by the old transaction, and will be committed to the journal as usual. This copy is simply deleted once that transaction commits. Of course, we cannot reclaim the old transaction's log space until this buffer has been safely recorded elsewhere in the filesystem, but that is taken care of automatically due to the fact that the buffer must necessarily be committed into the next transaction's journal records.

Project status and future work

This is still a work-in-progress. The design of the initial implementation is both stable and simple, and we do not expect any major revisions in design to be necessary in order to complete the implementation.

The design described above is relatively straightforward and will require minimal modifications to the existing ext2fs code other than the code to handle the management of the journal file, the association between buffers and transactions and the recovery of filesystems after an unclean shutdown.

Once we have a stable codebase to test, there are many possible directions in which we could extent the basic design. Of primary importance will be the tuning of the filesystem performance. This will require us to study the impact of arbitrary parameters in the journaling system such as commit frequencies and log sizes. It will also involve a study of bottlenecks to determine if performance might be improved through modifications to the design of the system, and several possible extensions to the design already suggest themselves.

One area of study may be to consider compressing the journal updates of updates. The current scheme requires us to write out entire blocks of metadata to the journal even if only a single bit in the block has been modified. We could compress such updates quite easily by recording only the changed values in the buffer rather than logging the buffer in its entirety. However, it is not clear right now whether this would offer any major performance benefits. The current scheme requires no memory-to-memory copies for most writes, which is a big performance win in terms of CPU and bus utilisation. The IOs which result from writing out the whole buffers are cheap−the updates are contiguous and on modern disk IO systems they are transferred straight out from main memory to the disk controller without passing through the cache or CPU.

Another important possible area of extension is the support of fast NFS servers. The NFS design allows a client to recover gracefully if a server crashes: the client will reconnect when the server reboots. If such a crash occurs, any client data which the server has not yet written safely to disk will be lost, and so NFS requires that the server must not acknowledge completion of a client's filesystem request until that request has been committed to the server's disk.

This can be an awkward feature for general purpose filesystems to support. The performance of an NFS server is usually measured by the response time to client requests, and if these responses have to wait for filesystem updates to be synchronised to disk then overall performance is limited by the latency of on-disk filesystem updates. This contrasts with most other uses of a filesystem, where performance is measured in terms of the latency of in-cache updates, not on-disk updates.

There are filesystems which have been specifically designed to address this problem. WAFL[6] is a transactional tree-based filesystem which can write updates anywhere on the disk, but the Calaveras filesystem[7] achieves the same end through use of a journal much like the one proposed above. The difference is that Calaveras logs a separate transaction to the journal for each application filesystem request, thus completing individual updates on disk as quickly as possible. The batching of commits in the proposed ext2fs journaling sacrifices that rapid commit in favour of committing several updates at once, gaining throughput at the expense of latency (the on-disk latency is hidden from applications by the effects of the cache).

Two ways in which the ext2fs journaling might be made more fit for use on an NFS server may be the use of smaller transactions, and the logging of file data as well as metadata. By tuning the size of the transactions committed to the journal, we may be able to substantially improve the turnaround for committing individual updates. NFS also requires that data writes be committed to disk as quickly as possible, and there is no reason in principle why we should not extend the journal file to cover writes of normal file data.

Finally, it is worth noting that there is nothing in this scheme which would prevent us from sharing a single journal file amongst several different filesystems. It would require little extra work to allow multiple filesystems to be journaled to a log on a

根据旧事务，并将像往常一样提交到日志中。一旦该事务提交，此副本就会被简单地删除。当然，在缓冲区在其他文件系统中的位置安全记录之前，我们无法重新收回旧事务的日志空间，但由于缓冲区必然要提交到下一个事务的日志记录中，所以这一点会自动处理。

项目状态与未来工作

这仍然是一个进行中的项目。初始实现的设 计既稳定又简单，我们预计在完成实现之前，设计 方面不需要进行任何重大修订。

上述设计相对直接，除了处理日志文件管 理的代码、缓冲区与事务的关联以及非正常关机 后文件系统的恢复代码之外，对现有的 ext2fs 代码 需要的最小修改。

一旦我们拥有一个稳定的代码库进行测试，就 有许多可能的方向可以扩展基本设计。最重要的是调整文件系统的性能。这将需要我们研究日志系统 中任意参数的影响，例如提交频率和日志大小。它 也将涉及对瓶颈的研究，以确定是否可以通过修 改系统设计来提高性能，而且已经有几个可能的扩 展建议自己浮现出来。

一个研究方向可能是考虑压缩更新日志的更新。当前方案要求我们即使块中只有一个比特被修改，也要将整个元数据块写入日志。我们可以通过在缓冲区中仅记录更改的值来轻松压缩此类更新，而不是记录整个缓冲区。然而，目前尚不清楚这种方法是否能够带来显著的性能提升。当前方案在大多数写入操作中无需内存到内存的复制，这在CPU和总线利用率方面是一个巨大的性能优势。由于更新是连续的，因此从整个缓冲区写入产生的IO在现代磁盘IO系统中可以直接从主内存传输到磁盘控制器，而无需经过缓存或CPU。−

另一个重要的扩展方向是支持快速NFS服务器。NFS设计允许客户端在服务器崩溃时优雅地恢复：当服务器重启时，客户端将重新连接。如果发生此类崩溃，任何服务器尚未安全写入磁盘的客户端数据都将丢失，因此NFS要求服务器必须在客户端的文件系统请求完成并提交到服务器磁盘后才能确认完成。

这对通用文件系统来说可能是一个尴尬的功能。NFS 服务器的性能通常通过其对客户端请求的响应时间来衡量，如果这些响应必须等待文件系统更新同步到磁盘，那么整体性能就会受到磁盘文件系统更新的延迟限制。这与大多数其他文件系统使用方式形成对比，在这些方式中，性能是以缓存更新的延迟来衡量，而不是磁盘更新的延迟。

有一些文件系统是专门设计用来解决这个问题的。WAFL[6] 是一种基于事务的树形文件系统，它可以在磁盘的任何位置写入更新，而Calaveras文件系统[7] 则通过使用类似于上面提出的日志来实现相同的目标。不同之处在于，Calaveras为每个应用程序文件系统请求在日志中记录一个单独的事务，从而尽可能快地在磁盘上完成单个更新。提出的ext2fs日志的提交批处理牺牲了这种快速提交，以换取一次提交多个更新，从而在牺牲延迟（应用程序通过缓存的效果隐藏了磁盘延迟）的情况下提高了吞吐量。

ext2fs日志系统可以通过两种方式使其更适合在NFS服务器上使用：使用更小的交易，以及记录文件数据和元数据。通过调整提交到日志的交易大小，我们可能能够显著提高提交单个更新的周转时间。NFS还要求数据写入尽可能快地提交到磁盘，原则上没有理由不将日志文件扩展以覆盖正常文件数据的写入。

最后，值得注意的是，本方案中没有任何内容会阻止我们将单个日志文件共享到多个不同的文件系统中。让多个文件系统将日志记录到完全专用于此目的的单独磁盘上，只需要少量额外的工作，这在许多已记录日志的文件系统都处于高负载的情况下，可能会显著提升性能。单独的日志磁盘几乎完全按顺序写入，因此可以在不影响主文件系统磁盘可用带宽的情况下，维持高吞吐量。

separate disk entirely reserved for the purpose, and this might give a significant performance boost in cases where there are many journaled filesystems all experiencing heavy load. The separate journal disk would be written almost entirely sequentially, and so could sustain high throughput without hurting the bandwidth available on the main filesystem disks.

Conclusions

The filesystem design outlined in this paper should offer significant advantages over the existing ext2fs filesystem on Linux. It should offer increased availability and reliability by making the filesystem recover more predictably and more quickly after a crash, and should not cause much, if any, performance penalty during normal operations.

The most significant impact on day-to-day performance will be that newly created files will have to be synced to disk rapidly in order to commit the creates to the journal, rather than allowing the deferred writeback of data normally supported by the kernel. This may make the journaling filesystem unsuitable for use on /tmp filesystems.

The design should require minimal changes to the existing ext2fs codebase: most of the functionality is provided by a new journaling mechanism which will interface to the main ext2fs code through a simple transactional buffer IO interface.

Finally, the design presented here builds on top of the existing ext2fs on-disk filesystem layout, and so it will be possible to add a transactional journal to an existing ext2fs filesystem, taking advantage of the new features without having to reformat the filesystem.

References

[1] A fast file system for UNIX. McKusick, Joy, Leffler and Fabry. *ACM Transactions on Computer Systems*, vol. 2, Aug. 1984

[2] Soft Updates: A Solution to the Metadata Update Problem in File Systems. Ganger and Patt. *Technical report CSE-TR-254-95*, Computer Science and Engineering Division, University of Michigan, 1995.

[3] The design and implementation of a log-structured file system. Rosenblum and Ousterhout. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, Oct. 1991

[4] An implementation of a log-structured file system for Unix. Seltzer, Bostic, McKusick and Staelin. *Proceedings of the Winter 1993 USENIX Technical Conference*, Jan. 1993

[5] Linux Log-structured Filesystem Project. Deuel and Cook. *http://collective.cpoint.net/prof/lfs/*

[6] File System Design for an NFS File Server Appliance. Dave Hitz, James Lau and Michael Malcolm. *http://www.netapp.com/technology/level3/30-02.html#preface*

[7] Metadata Logging in an NFS Server. Uresh Vahalia, Cary G. Gray, Dennis Ting. *Proceedings of the Winter 1995 USENIX Technical Conference*, 1995: pp. 265-276

单独的日志磁盘几乎完全按顺序写入，因此可以在不影响主文件系统磁盘可用带宽的情况下，维持高吞吐量。

结论

本文中概述的文件系统设计应比 Linux 上的现有 ext2fs 文件系统提供显著优势。它应通过使文件系统在崩溃后更可预测且更快速地恢复，从而提高可用性和可靠性，并且在正常操作期间不应造成太多（如果有的话）性能损失。

对日常性能影响最大的是，新创建的文件必须快速同步到磁盘，以便将创建内容提交到日志，而不是像内核通常支持的那样允许数据延迟写回。这可能使日志文件系统不适合用于 /tmp 文件系统。

该设计应要求对现有的 ext2fs 代码库进行最小更改：大部分功能是由一个新的日志机制提供的，该机制将通过一个简单的交易式缓冲区 IO 接口与主 ext2fs 代码接口。

最后，这里提出的设计建立在现有的 ext2fs 磁盘文件系统布局之上，因此将能够向现有的 ext2fs 文件系统添加事务性日志，利用新功能而无需重新格式化文件系统。

参考文献

[1] A fast file system for UNIX. McKusick, Joy, Leffler and Fabry. *ACM Transactions on Computer Systems*, vol. 2, Aug. 1984

[2] Soft Updates: A Solution to the Metadata Update Problem in File Systems. Ganger and Patt. *Technical report CSE-TR-254-95*, Computer Science and Engineering Division, University of Michigan, 1995.

[3] The design and implementation of a log-structured file system. Rosenblum and Ousterhout. *Proceedings of the Thirteenth ACM Symposium on Operating Systems Principles*, Oct. 1991

[4] An implementation of a log-structured file system for Unix. Seltzer, Bostic, McKusick and Staelin. *Proceedings of the Winter 1993 USENIX Technical Conference*, Jan. 1993

[5] Linux Log-structured Filesystem Project. Deuel and Cook. *http://collective.cpoint.net/prof/lfs/*

[6] File System Design for an NFS File Server Appliance. Dave Hitz, James Lau and Michael Malcolm. *http://www.netapp.com/technology/level3/30-02.html#preface*

[7] Metadata Logging in an NFS Server. Uresh Vahalia, Cary G. Gray, Dennis Ting. *Proceedings of the Winter 1995 USENIX Technical Conference*, 1995: pp. 265-276