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.

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 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.

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.

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 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 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 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/level3/3002.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