

Journalled Soft-updates

Marshall Kirk McKusick

Author and Consultant

Jeff Roberson

Consultant

ABSTRACT

This paper describes the work to add “journaling lite” to soft updates and its incorporation into the FreeBSD fast filesystem. Because soft updates prevent most inconsistencies, the journaling need only deal with tracking those inconsistencies that soft updates fails to address. Specifically, the journal contains the information needed to recover the block and inode resources that have been freed but whose freed status failed to make it to disk before a system failure. After a crash, a variant of the venerable *fsck* program runs through the journal to identify and free the lost resources. Only if a corruption of the log is detected is it necessary to run background *fsck*. The journal is tiny, 16Mb is usually enough independent of filesystem size. Although journal processing needs to be done before restarting, the processing time is typically just a few seconds and in the worst case a minute. The addition or deletion of soft-updates journaling to existing fast filesystems is done using the *tunefs* program.

1. Background and Introduction

The soft updates dependency tracking system was adopted by FreeBSD in 1998 as an alternative to the popular journalled-filesystem technique [Ganger & Patt, 1994; McKusick, Bostic, Karels, & Quarterman, 1996]. While the runtime performance and consistency guarantees of soft updates are comparable to journalled filesystems [Seltzer et al, 2000], it relies on an expensive and time-consuming background filesystem recovery operation after a crash [McKusick, 2002]. This paper outlines a method for eliminating the necessity of an expensive background or foreground whole-filesystem check operation through the use of a small journal which logs the only two inconsistencies possible in soft updates. The first is allocated but unreferenced inodes; the second is allocated but unreferenced blocks [Ganger, McKusick, & Patt,]. This journal allows a journal-analysis program to complete recovery in just a few seconds independent of filesystem size.

2. Compatibility with Other Implementations

Journaling is enabled via *tunefs* and only requires a few spare superblock fields and 16Mb of free blocks for the journal. These minimal requirements make it easily enabled on existing FreeBSD filesystems. The journal’s filesystem blocks are placed in an inode named *.sujournal* in the root of the filesystem and filesystem flags are set such that older non-journaling kernels will trigger a full filesystem check upon mounting a previously journalled volume. When mounting a journalled filesystem, older kernels clear a flag indicating that journaling is being done so that when the filesystem is next encountered by a kernel that does journaling, it will know that that the journal is invalid and will ensure that the filesystem is consistent and clear the journal before resuming use of the filesystem.

3. Journal Format

The journal is kept as a circular buffer of segments containing records which are never allowed to overflow. If the journal fills, the filesystem must complete enough operations to expire journal entries before allowing new operations. In practice, the journal almost never fills.

Each journal record contains a sequence number which uniquely identifies the record in addition to the context required for that journal entry. Journal entries are aggregated into segments to minimize the number of writes to the journal. Each segment contains a header with the sequence number of the first and last entry in the segment along with the last valid sequence number at the time it was written. Segments are fixed to some multiple of the disk block size and must be written all at once to avoid read/modify/write cycles in running filesystems.

The journal-analysis has been incorporated into the *fsck* program. This incorporation into the existing *fsck* program has several benefits. The existing startup scripts already call *fsck* to see if it needs to be run in foreground or background. For filesystems running with journaled soft updates, *fsck* can request to run in foreground and do the needed journaled operations before the filesystem is brought online. If the journal fails for some reason, it can instead report that a background *fsck* needs to be run as the traditional fallback. Thus, this new functionality can be introduced without any need for system administrators to change the way that they start up their systems. Finally, the invoking of *fsck* means that after the journal has been processed, it is possible for debugging purposes to fall through and run a complete check of the filesystem to ensure that the journal is working properly.

The journal entry size is 32 bytes, providing quite a dense representation allowing for 16 entries per-sector. The journal is created in a single area of the filesystem in as contiguous an allocation as is available. We considered spreading it out across cylinder groups to optimize locality for writes but it ended up being so small that this approach was not practical and would make scanning the entire journal during cleanup too slow. The journal blocks are claimed by a named immutable inode. This approach allows user-level access to the journal for debugging and statistics gathering purposes as well as providing backwards compatibility with older kernels that do not support journaling. We have found that a journal size of 16Mb is sufficient in even the most tortuous and worst-case benchmarks. Each 32Kb of journal space can cover 1024 namespace operations or 16MB of outstanding allocations (assuming a standard 16Kb block size).

4. Modifications that Require Journaling

The next subsections describe the operations that must be journaled so that the information needed to clean up the filesystem is available to *fsck*.

4.1. Increased Link Count

A link count may be increased through a hard link or file creation. The link count is temporarily increased during a rename. Here, the operation is the same. The inode number, parent inode number, directory offset, and initial link count are all recorded in the journal. Soft updates guarantees that the inode link will be increased and stable on disk prior to any directory write. The journal write must occur prior to the inode write that updates the link count and prior to the bitmap write that allocates the inode if it is newly allocated.

4.2. Decreased Link Count

The inode link count is decreased through unlink or rename. The inode number, parent inode, directory offset, and initial link count are all recorded in the journal. The deleted directory entry is guaranteed to be written before the link is adjusted down. As with increasing the link count, the journal write must happen prior to all other writes.

4.3. Unlink While Referenced

Unlinked yet referenced files pose a unique problem for journaled filesystems. Simply leaving the journal entry valid while waiting for applications to close their dangling references is untenable as it will easily exhaust journal space. A solution which scales to the total number of inodes in the filesystem is required. At least two approaches are possible, a replication of the inode allocation bitmap, or a linked list of inodes to be freed. We have chosen to use the linked-list approach.

In the linked-list case which is employed by several filesystems (xfs, ext4, etc.), the super-block contains the inode number that serves as the head of a singly linked list of inodes to be freed, with each inode storing the next pointer. The advantage of this approach is that at recovery time you need only examine a single pointer. The disadvantage is that you must keep an in memory doubly-linked list so that you can rapidly remove an inode once it is unreferenced. This approach ingrains a filesystem-wide lock in the design and incurs non-local writes when maintaining the list. In practice we have found that unreferenced inodes occur rarely enough that this approach is not a bottleneck.

Removal from the list may be done lazily but must be completed prior to any re-use of the inode. Additions to the list must be stable prior to the final unlink but irrespective of the journal write. Addition and removal involves only a single write.

4.4. Change of Directory Offset

Any time a directory compaction moves an entry, a journal entry must be created indicating the old and new locations of the entry. The kernel does not know at the time of the move whether a remove will follow it, so at this time all offset changes are journaled.

4.5. Block Allocation and Free

For either block allocation or free, whether it is a fragment, indirect block, directory block, direct block, or extended attributes the operation is the same. The inode number of the file and the offset of the block within the file is recorded using negatives for indirects and extents as is done with “getblk”. Additionally, the disk block address and number of fragments is included in the journal record. The journal entry must be written to disk prior to any allocation or free.

5. The Recovery Process

The next subsections describe the use of the journal by *fsck* to clean up the filesystem after a crash.

5.1. Scanning the Journal

To do recovery, the *fsck* program must first scan the journal from start to end to discover the oldest valid sequence number. We contemplated keeping journal head and tail pointers, but that would require extra writes to the superblock area. Because the journal is small, the extra time spent scanning it to identify the head and tail of the valid journal seemed a reasonable tradeoff to reduce the run-time cost of maintaining the journal. So, the *fsck* program must discover the first segment containing a still valid sequence number and work from there. Journal records are then resolved in order. Journal records are marked with a timestamp that must match the filesystem mount time as well as a CRC to protect the validity of the contents.

5.2. Adjusting Link Counts

For each journal record recording a link increase, *fsck* needs to examine the directory at the offset provided and see whether the directory entry exists on disk. If it does not exist, but the inode link count was increased, then the recorded link count needs to be decremented.

For each journal record recording a link decrease, *fsck* needs to examine the directory at the offset provided and see whether the directory entry exists on disk. If it has been deleted on disk, but the

inode link count has not been decremented, then the recorded link count needs to be decremented.

Compaction of directory offsets for entries that are being tracked complicates the link adjustment scheme presented above. Since directory blocks are not written synchronously, *fsck* must look up each directory entry in all its possible locations.

When an inode is added and removed from a directory multiple times *fsck* is not be able to correctly assess the link count given the algorithm presented above. The chosen solution is to pre-process the journal and link all entries related to the same inode together. In this way, all operations not known to be committed to stable store can be examined concurrently to determine how many links should exist relative to the known stable count that existed prior to the first journal entry. Duplicate records that occur when an inode is added and deleted at the same offset many times are discarded, resulting in a coherent count.

5.3. Updating the Allocated Inode Map

Once the link counts have been adjusted, *fsck* must free any inodes whose link count has fallen to zero. In addition, *fsck* must free any inodes that were unlinked but still in use at the time that the system crashed. The head of the list of unreferenced inode is in the superblock as described in section 4.3. above. The *fsck* program must traverse this list of unlinked inodes and free them.

The first step in freeing an inode is to add all of its blocks to list of blocks that need to be freed. Next the inode needs to be zero'ed to show that it is not in use. Finally, the inode bitmap in its cylinder group must be updated to reflect that it is available and all the appropriate filesystem statistics updated to reflect its availability.

5.4. Updating the Allocated Block Map

Once the journal has been scanned, it provides a list of blocks that were intended to be freed. The journal entry lists the inode from which the block was to be freed. For recovery, *fsck* processes each free record by checking to see if the block is still claimed by its associated inode. If it finds that the block is no longer claimed, it is freed.

For each block that is freed either by the deallocation of an inode, or through the identification process described above, the block bitmap in its cylinder group must be updated to reflect that it is available and all the appropriate filesystem statistics updated to reflect its availability. When a fragment is freed, the fragment availability statistics must also be updated.

6. Performance

We have not yet collected performance comparisons of soft updates versus soft updates with journaling. We hope to have time to get at least some preliminary results by the time of the BSDCan conference.

7. Future Work

The next subsection describes some areas that we have not yet explored that may give further performance improvements to our implementation.

7.1. Rollback of Directory Deletions

Doing a rollback of a directory addition is easy. The new directory entry has its inode number set to zero to indicate that it is not really allocated. However, rollback of directory deletions is much more difficult as the space may have been claimed by a new allocation. There are times when being able to roll back a directory deletion would be very convenient. For example, preventing the removal of an old name prior to a new name reaching stable store when a file is renamed. Here, we have considered using a distinguished inode number that the filesystem internally would recognize as being in use, but that would not be returned to the user application. However, at present we cannot rollback deletes, which requires any delete journaling to be written to disk prior to the writing of affected directory blocks.

7.2. Truncate and Weaker Guarantees

As a potential optimization, “truncate” may choose to instead record the intended file size and operate more lazily, relying on the log to recover any partially completed operations correctly. This approach also allows us to do partial truncations asynchronously. Further, the journal allows for the weakening of other soft dependency guarantees although we have not yet been fully explored these reduced guarantees and do not know if they provide any real benefit.

8. Types of Journal Records

For those with an interest in the details of the implementation, this section catalogs the data structures that have been added to the standard soft updates structures to support the journaling.

A *jseg* structure tracks all the journal records written in a single disk write.

A *jsegdep* structure tracks a single reference to a written journal segment so the journal space can be reclaimed when all dependencies have been written.

A *jaddref* structure tracks a new reference (link count) on an inode and prevents the link count increase and bitmap allocation until a journal entry has been written.

A *jremref* structure tracks a removed reference (unlink) on an inode and prevents the directory remove from proceeding until the journal entry is written.

A *jmvref* structure tracks name relocations within the same directory block that occur as a result of directory compaction. This is used by the recovery code to update the expected offsets for added and removed names.

A *jnewblk* structure tracks a newly allocated block or fragment and prevents the direct or indirect block pointer as well as the cylinder-group bitmap from being written until it is logged to the journal.

A *jfreeblk* structure tracks the journal write for freeing a block or tree of blocks. The block pointer cannot be cleared in the inode or indirect prior to the *jfreeblk* being written.

A *jfreefrag* tracks the freeing of a single block when a fragment is extended or an indirect page is replaced. It is only needed if the fragment is not part of a larger *freeblks* operation.

A *jtrunc* journals the intent to truncate an inode to a non-zero value. This operation is done synchronously prior to the synchronous partial truncation process. The associated *jsegdep* is not released until the truncation is complete and the truncated inode has been written to disk.

A *freework* structure handles the release of a tree of blocks or a single block. Each indirect block in a tree is allocated its own *freework* structure so that the indirect block may be freed only when all its children have been freed. Thus, we enforce the rule that an allocated block must have a valid path to a root that is journaled.

A *sbdep* structure tracks the head of the list of inodes whose names have been deleted, but are still being held open by a process. This *sbdep* structure ensures that the superblock is always pointing at the first possible unlinked inode for the recovery process.

References

- Ganger, McKusick, & Patt, .
G. Ganger, M. McKusick, & Y. Patt, “Soft Updates: A Solution to the Metadata Update Problem in Filesystems,” *ACM Transactions on Computer Systems* (in preparation).

Ganger & Patt, 1994.

G. Ganger & Y. Patt, "Metadata Update Performance in File Systems," *USENIX Symposium on Operating Systems Design and Implementation*, p. 49–60 (November 1994).

McKusick, Bostic, Karels, & Quarterman, 1996.

M. McKusick, K. Bostic, M. Karels, & J. Quarterman, *The Design and Implementation of the 4.4BSD Operating System*, p. 269–271, Addison Wesley Publishing Company, Reading, MA (1996).

McKusick, 2002.

M. K. McKusick, "Running Fsync in the Background," *Proceedings of the BSDCon 2002 Conference*, pp. 55-64 (February 2002).

Seltzer et al, 2000.

M. Seltzer, G. Ganger, M. K. McKusick, K. Smith, C. Soules, & C. Stein, "Journaling versus Soft Updates: Asynchronous Meta-data Protection in File Systems," *Proceedings of the San Diego Usenix Conference*, pp. 71-84 (June 2000).

9. Biographies

Dr. Marshall Kirk McKusick writes books and articles, consults, and teaches classes on UNIX- and BSD-related subjects. While at the University of California at Berkeley, he implemented the 4.2BSD fast filesystem, and was the Research Computer Scientist at the Berkeley Computer Systems Research Group (CSRG) overseeing the development and release of 4.3BSD and 4.4BSD. His particular area of interest is the filesystem. He earned his undergraduate degree in Electrical Engineering from Cornell University, and did his graduate work at the University of California at Berkeley, where he received master's degrees in computer science and business administration and a doctoral degree in computer science. He has twice been president of the board of the Usenix Association, is currently a member of the editorial board of ACM's Queue magazine, and is a member of the Usenix Association and ACM, and is a senior member of the IEEE.

In his spare time, he enjoys swimming, scuba diving, and wine collecting. The wine is stored in a specially constructed wine cellar (accessible from the web at <http://www.mckusick.com/~mckusick/>) in the basement of the house that he shares with Eric Allman, his domestic partner of 30-and-some-odd years. You can contact him via email at <mckusick@mckusick.com>.

Jeff Roberson is a consultant who lives on the island of Maui in the Hawai'ian island chain. When he is not cycling, hiking, or otherwise enjoying the island, he gets paid to improve FreeBSD. He is particularly interested in problems facing server

installations and has worked on areas as varied as the kernel memory allocator, thread scheduler, file systems interfaces, and network packet storage among others. You can contact him via email at <jrober-son@jrober-son.net>.