Journaled 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 journal need only track 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 an inconsistency between the log and filesystem is detected is it necessary to run *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. It is not necessary to build a new filesystem to use soft-updates journaling. 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 journaled-filesystem technique [Ganger & Patt, 1994; McKusick, Bostic, Karels, & Quarterman, 1996]. While the runtime performance and consistency guarantees of soft updates are comparable to journaled 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 blocks; the second is incorrectly high link counts. Incorrectly high link counts include unreferenced inodes that were being deleted and files that were unlinked but open [Ganger, McKusick, & Patt, 2000]. 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 journaled volume. When mounting a journaled 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 log of segments containing records which describe metadata operations. 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 segment contains a unique sequence number and a timestamp which identifies the filesystem mount instance so old segments can be discarded during journal processing. Journal entries are aggregated into segments to minimize the number of writes to the journal. Each segment contains the last valid sequence number at the time it was written to allow *fsck* to recover the head and tail by scanning the entire journal. Segments are variably sized as some multiple of the disk block size and are written atomically 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 full *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. A 16Mb journal can cover over 500,000 namespace operations or 8Gb 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 count 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. In UNIX, an inode’s storage is not reclaimed until after the final name is removed and the last reference is closed. 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 a pointer to the next inode in the list. The advantage of this approach is that at recovery time you need only examine a single pointer in the superblock which will already be in memory. 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 nonlocal 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 reclaiming journal space for the final unlink but otherwise may be delayed long enough to avoid needing the write at all if the file is quickly closed. Addition and removal involve only a single write to update the preceding pointer to the subsequent inode.

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. Without this information *fsck* would be unable to disambiguate multiple revisions of the same directory block.

4.5. Block Allocation and Free

When performing either block allocation or free, whether it is a fragment, indirect block, directory block, direct block, or extended attributes the record 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.

When freeing an indirect only the root of the indirect tree is logged. Thus, for truncation we need a maximum of 15 journal entries, 12 for direct blocks and 3 for indirects. These 15 journal entries allow us to free a large amount of space with a minimum of journaling overhead. During recovery, *fsck* will follow indirect blocks and free any descendants including other indirects. For this algorithm to work, the contents of the indirect block must remain valid until the journal record is free so that user data is not confused with indirect pointers.

5. Additional Requirements of Journaling

Some operations that had not previously required tracking under soft updates need to be tracked when journaling is introduced. This section describes these new requirements.

5.1. Cylinder Group Rollbacks

Soft updates previously did not require any rollbacks of cylinder groups as they were always the first or last write in a group of changes. When a block or inode has been allocated but its journal record has not yet been written to disk, it is not safe to write the updated bitmaps and associated allocation information. The routines which write blocks with *bmsafemap* dependencies now rollback any allocations with unwritten journal operations.

5.2. Inode Rollbacks

The inode link count must be rolled back to the link count as it existed prior to any unwritten journal entries. Allowing it to grow beyond this count would not cause filesystem corruption but it would prohibit the journal recovery from adjusting the link count properly. Soft updates already prevents the link count from decreasing before the directory entry is removed as a premature decrement could cause filesystem corruption.

When an unlinked file has been closed, its inode cannot be returned to the inode freelist until its zeroed block pointers have been written to disk so that its blocks can be freed and it has been removed from the on-disk list of unlinked files. The unlinked-file inode is not completely removed from the list of unlinked files until the next pointer of the inode that precedes it in the list has been updated on disk to point to the inode that follows it on the list. If the unlinked-file inode is the first inode on the list of unlinked files, then it is not completely removed from the list of unlinked files until the head-of-unlinked-files pointer in the superblock has been updated on disk to point to the inode that follows it on the list.

5.3. Reclaiming Journal Space

To reclaim journal space from previously written records, the kernel must know that the operation the journal record describes is stable on disk. This requirement means that when a new file is created, the journal record cannot be freed until writes are completed for a cylinder group bitmap, an inode, a directory block, a directory inode, and possibly some number of indirect blocks. When a new block is allocated, the journal record cannot be freed until writes are completed for the new block pointer in the inode or indirect, the cylinder group bitmap, and the block itself. Blocks pointers within indirects are not stable until all parent indirects are fully reachable on disk via the inode indirect pointers. To facilitate fulfillment of these requirements, the dependencies that describe these operations carry pointers to the oldest segment structure in the journal containing journal entries that describe outstanding operations.

Some operations may be described by multiple entries. For example, when making a new directory, its addition creates three new names. Each of these names is associated with a reference count on the inode to which the name refers. When one of these dependencies is satisfied, it may pass its journal entry reference to another dependency if another operation on which the journal entry depends is not yet complete. If the operation is complete, the final reference on the journal record is released. When all references to journal records in a journal segment are released, its space is reclaimed and the oldest valid segment sequence number is adjusted. We can only release the oldest free journal segment, since the journal is treated as a circular queue.

5.4. Handling a Full Journal

If the journal ever becomes full, we must prevent any new journal entries from being created until more space becomes available from the retirement of the oldest valid entries. A very effective way to stop the creation of new journal records is to suspend the filesystem using the mechanism in place for taking snapshots. Once suspended, existing operations on the filesystem are permitted to complete, but new operations that wish to modify the filesystem are put to sleep until the suspension is lifted.

We do a check for journal space before each operation that will change a link count or allocate a block. If we find that the journal is approaching a full condition, we suspend the filesystem and expedite the progress on the soft-updates work-list processing to speed the rate at which journal entries get retired. As the operation that did the check has already started, it is permitted to finish, but future operations are blocked. Thus, operations must be suspended while there is still enough journal space to complete operations already in progress. When enough journal entries have been freed, the filesystem suspension is lifted and normal operations resume.

In practice, we had to create a minimal sized journal (4Mb) and run scripts designed to create huge numbers of link-count changes, block allocations, and block frees to trigger the journal-full condition. Even under these tests, the filesystem suspensions were infrequent and very brief lasting under a second.

6. The Recovery Process

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

6.1. 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.

6.2. 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 for the indicated inode number 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 for the indicated inode number 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 the disk 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.

6.3. 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. 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.

6.4. 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.

7. Performance

Journaling adds extra running time and memory allocations to the traditional soft-updates requirements and also additional I/O operations to write the journal. The overhead of the extra running time and memory allocations was immeasurable in the benchmarks that we ran. The extra I/O was mostly evident in the increased delay for individual operations to complete. Operation completion time is usually only evident to an application when it does an ‘‘fsync’’ system call which causes it to wait for the file to reach the disk. Otherwise, the extra I/O to the journal only becomes evident in benchmarks that are limited by the filesystem's I/O bandwidth before journaling is enabled. In summary, a system running with journaled soft updates will never run faster than one running soft updates without journaling. So, systems with small filesystems such as an embedded system will usually want to run soft updates without journaling and take the time to run *fsck* after system crashes.

The primary purpose of the journaling project was to eliminate long filesystem check times. A 40TB volume may take an entire day and a considerable amount of memory to check. We have run several scenarios to understand and validate the recovery time.

A relatively normal case for developers is to run a parallel buildworld. Crash recovery from this case demonstrates time to recover from moderate write workload. A 250GB disk was filled to 80% with copies of the FreeBSD source tree. One copy was selected at random and an 8 way buildworld proceeded for 10 minutes before the box was reset. Recovery from the journal took 0.9 seconds. An additional run with traditional *fsck* was used to verify the safe recovery of the filesystem. The *fsck* took approximately 27 minutes, or 1800 times as long.

A testing volunteer with a 92% full 11TB volume spanning 14 drives on a 3ware RAID controller generated hundreds of megabytes of dirty data by writing random length files in parallel before resetting the machine. The resulting recovery operation took less than one minute to complete. A normal *fsck* run takes approximately 10 hours on this filesystem.

8. Future Work

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

8.1. 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 the disk 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.

8.2. Truncate and Weaker Guarantees

As a potential optimization, the ‘‘truncate’’ system call 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 know know if they provide any real benefit.

9. New Data Structures

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

9.1. New Dependency Structures

The following structures have been added to the standard soft updates structures to support the journaling. The first three records fulfill similar roles to the existing soft updates structures as they track when their filesystem resource has been written to disk, then trigger another step in the filesystem operation that they are tracking.

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. Each 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 *freedep* structure tracks the completion of a bitmap write for a *freework*. One *freedep* may cover many freed blocks so long as they reside in the same cylinder group. When the cylinder group is written, the *freedep* decrements the reference count on the *freework* which is freed when its reference count reaches zero.

A *sbdep* structure tracks the writing of the superblock that contains 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.

The remaining nine new dependency structures are used to track writes to the journal. Typically they will prevent updates to filesystem data structures until their tracked journal entry has been written to the disk. They are identified with a leading ‘‘j’’ in their name.

A *jseg* structure tracks the records within a journal segment. A segment contains all the journal records written in a single disk write. When all of the operations associated with the records in that segment have been committed to disk, the *jseg* structure allows its segment to be freed. If its segment is the oldest valid segment, that segment as well as any unused segments that follow it are returned for the use of future journal entries.

A *jsegdep* structure tracks the validity of a written journal record. When all the record’s associated dependencies have been written to disk, the *jseg* that tracks the segment in which it is contained is notified that it is no longer needed.

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 a directory block that occur as a result of directory compaction. This information is used by the recovery code to updated the expected offsets for added and removed names. The *jmvref* prevents the directory directory block in which the compaction occurred from being written to disk until the journal entry is written.

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 written 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* journal entry being written.

A *jfreefrag* structure 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. The block pointer cannot be cleared in the inode or indirect prior to the *jfreefrag* journal entry being written.

A *jtrunc* structure journals the intent to truncate an inode to a non-zero value. The *jtrunc* record must be written to the journal prior to the synchronous partial truncation process. The associated *jsegdep* that tracks the *jtrunc* is not released until the truncation is complete and the truncated inode has been written to disk.

9.2. Types of Journal Records

The following structures all exist on-disk within the journal file. Each structure is a uniform size, 32 bytes, which simplifies journal processing. Each journal record has an opcode that can further refine its operation. Records with more than one opcode have their opcode noted below.

Every 512 byte disk block starts with a *jsegrec* record that may describe more than one block of journal entries. The *jsegrec* contains a 64-bit sequence number and the oldest valid sequence number as described in section 3. It also has a count of valid records and blocks along with a timestamp that identifies the mount instance.

A *jrefrec* record uniquely describes a single link addition (opcode of JOP_ADDREF) or removal (opcode of JOP_REMREF). If the link is transitioning to or away from zero, it also affects the allocation bitmap. It contains the inode number, parent inode number, directory offset, starting link count, and file mode.

A *jmvrec* record describes a relocated directory entry when its containing directory block is compacted by the kernel. It contains an inode number, parent inode number, old directory offset and new directory offset.

A *jblkrec* record describes either an allocation (opcode of JOP_NEWBLK) or free (opcode of JOP_FREEBLK) of a block. It contains an inode number, logical block number, physical block number, and frag count. Negative logical numbers indicate extended attributes or indirect blocks.

A *jtrncrec* record is used only for partial truncation where the recovery process must evaluate the current size of the inode and complete the truncation. It contains an inode number, desired file size, and desired external attribute size. A *jtrncrec* record is not used when truncating to zero. Rather, all direct blocks and root indirects are logged as frees and the inode pointers are written to zero so that it may all be done asynchronously.

10. 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, filesystems interfaces, and network packet storage among others. You can contact him via email at `<jroberson@jroberson.net>`.

References

Ganger, McKusick, & Patt, 2000.
G. Ganger, M. McKusick, & Y. Patt, "Soft Updates: A Solution to the Metadata Update Problem in Filesystems," *ACM Transactions on Computer Systems* **18**(2), p. 127−153 (May 2000).

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 Fsck 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).