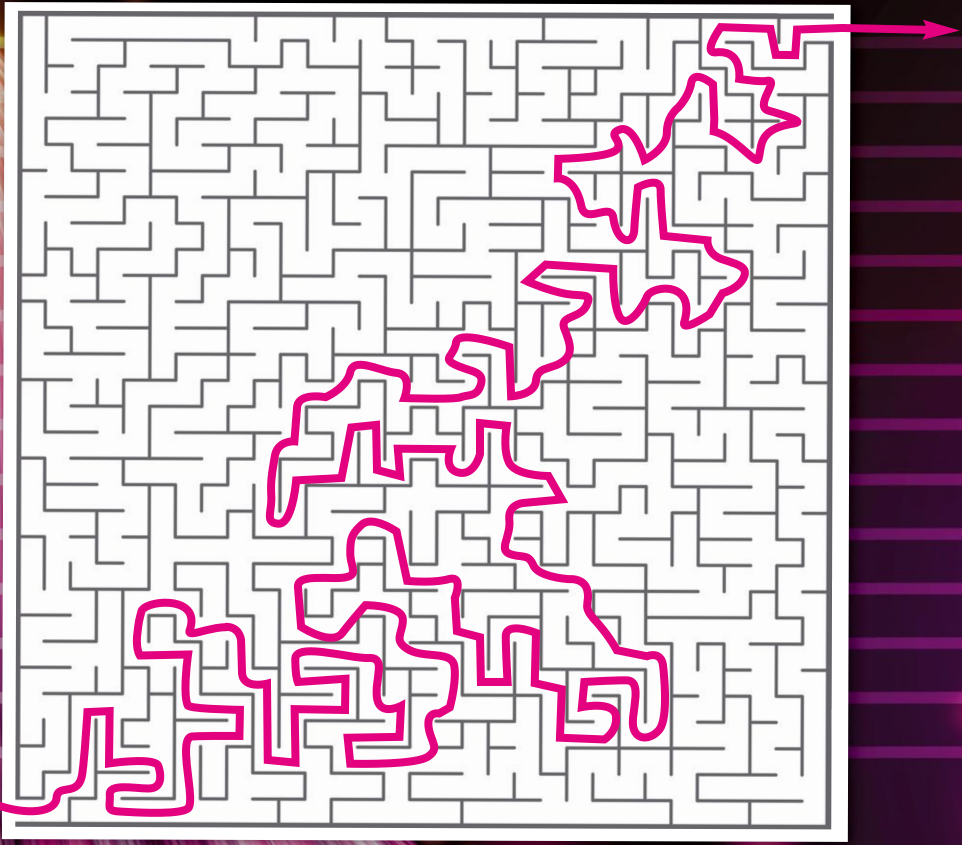
JOURNALED soft-updates

The soft-updates **DEPENDENCY TRACKING SYSTEM** was adopted by FreeBSD in 1998 as an alternative to the popular journaled file system technique.



By
Marshall Kirk McKusick
and Jeff Roberson

While the runtime performance and consistency guarantees of soft updates are comparable to journaled filesystems [Seltzer, Ganger, McKusick et al, 2000], a journaled filesystem relies on an expensive and time-consuming background filesystem recovery operation after a crash.

This article outlines a method for eliminating the expensive background or foreground whole-filesystem check operation by using a small journal that 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.

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 the whole-filesystem fsck. The journal is tiny, 16 Mbytes 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.

Compatibility with Other Implementations

It is not necessary to build a new filesystem to use soft-updates journaling. Journaling is enabled via tunefs and only requires a few spare superblock fields and 16 Mbytes 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 when mounting a previously journaled volume. When mounting a journaled filesystem, older kernels clear a flag that shows that journaling is being done, so that when the filesystem is next encountered by a kernel that does journaling, it will know that the journal is invalid and will ensure that the filesystem is consistent and clear the journal before resuming use of the filesystem.

Journal Format

The journal is kept as a circular log of segments containing records that 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 that 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 change to the way that system administrators 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 a dense representation allowing for 128 entries per 4-Kbyte 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 16 Mbytes is enough in even the most tortuous and worst-case benchmarks. A 16-Mbyte journal can cover over 500,000 namespace operations or 16 Gbyte of outstanding allocations (assuming a standard 32-Kbyte block size).

Modifications that Require Journaling

This subsection describes the operations that must be journaled so that the information needed to clean up the filesystem is available to fsck.

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 before any directory write. The journal write must occur before the inode write that updates the link count and before the bitmap write that allocates the inode if it is newly allocated.

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 before all other writes.

Unlink While Referenced

Unlinked yet referenced files pose a problem for journaled filesystems. In POSIX, 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 that 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 superblock 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 fsck need only examine a single pointer in the superblock that will already be in memory. The disadvantage is that the kernel must keep an in-memory, doubly-linked list so that it 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 before any re-use of the inode. Additions to the list must be stable before 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 following inode.

Change of Directory Offset

Any time a directory compaction moves an entry, a journal entry must be created describing 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 currently all offset changes are journaled. Without this information fsck would be unable to disambiguate multiple revisions of the same directory block.

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 are recorded using negative offsets for indirect and extended attribute blocks. Additionally, the disk block address and number of fragments are included in the journal record. The journal entry must be written to disk before any allocation or free.

When freeing an indirect block only the root of the indirect block tree is logged. Thus, for truncation we need a maximum of 15 journal entries, 12 for direct blocks and 3 for indirect blocks. 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 indirect blocks. 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 block pointers.

Additional Requirements of Journaling

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

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 that write blocks with "bmsafemap" dependencies now rollback any allocations with unwritten journal operations.

Inode Rollbacks

The inode link count must be rolled back to the link count as it existed before 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-out 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.

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 block, the cylinder group bitmap, and the block itself. Blocks pointers within indirect blocks are not stable until all parent indirect blocks are fully reachable on disk via the inode indirect block pointers. To simplify 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.

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. An 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 file system suspension is lifted and normal operations resume.

In practice, we had to create a minimal sized journal (4 Mbyte) 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 brief lasting under a second.

The Recovery Process

This subsection describes the use of the journal by fsck to clean up the filesystem after a crash.

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, however, 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 head and tail pointers. 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.

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 recorded 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 recorded 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 able to correctly assess the link count given the algorithm presented above. The chosen solution is to pre-process the journal and link together all entries related to the same inode. 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 before 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.

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 earlier in this article. The fsck program must traverse this list of unlinked inodes and free them.

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

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.

FOSSETCON

FREE AND OPEN SOURCE EXPO AND TECHNOLOGY CONFERENCE

FOSSETCON 2014

Come out and participate in the First Annual Fossetcon 2014 Florida's Only Free and Open Source Conference. With in 10 minutes of Disney Land, Universal Studios and Epcot Center.

DAY 0 FOOD, TRAINING, WORKSHOPS AND CERTIFICATIONS

DAY 1 FOOD, KEYNOTES, EXPO HALL, SPEAKER TRACKS

DAY 2 FOOD, KEYNOTES, EXPO HALL, SPEAKER TRACKS

BSD *Friendly*

FREE FOOD, TRAINING, CERTIFICATIONS AND GIVEAWAYS!!!

SEP 11 - SEPT 13

ROSEN PLAZA HOTEL ORLANDO, FL

More info at **www.fossetcon.org**

Fossetcon 2014: The Gateway To The Open Source Community

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.

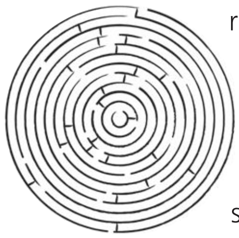
Performance

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 that 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 40-Tbyte 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 typical operation for developers is to run a parallel buildworld. Crash recovery from this case demonstrates time to recover from moderate write workload. A 250-Gbyte 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 about 27 minutes, or 1,800 times as long.

A testing volunteer with a 92% full 11-Tbyte 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 about 10 hours on this filesystem. ●

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 file system 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. He has twice been president of the board of the Usenix Association, is currently a member of the FreeBSD Foundation Board of Directors, a member of the editorial board of ***ACM Queue*** magazine and ***The FreeBSD Journal***, a senior member of the IEEE, and a member of the Usenix Association, ACM, and AAAS. 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

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).

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).