

Double the Metadata, Double the Fun: A COW-like Approach to File System Consistency

Val Henson

IBM, Inc.

val@nmt.edu

Theodore Ts'o

IBM, Inc.

tytso@alum.mit.edu

Abstract

Modern file systems maintain on-disk consistency through a variety of competing techniques, including after the fact repair, logging or journaling, soft updates, and copy-on-write (COW) techniques. The space of solutions has been well explored in the literature, and one or more of these techniques are in use in nearly all modern file systems. However, these techniques all have shortcomings which we explore in this paper. We argue that COW file systems are the best solution to maintaining file system consistency, and propose a COW-like filesystem design, doublefs, which overcomes some of drawbacks of COW file systems. Doublefs preallocates a second copy of all metadata, writes updates to the unused copy, and atomically switches to the new set of metadata when it is consistent on disk.

1 Introduction

One of the most difficult problems to solve in file systems is dealing with unexpected system crashes. Since a single logical operation, such as creating a file, requires writing many different blocks on disk, if the system crashes in the middle of these writes, the on-disk data will be inconsistent. The original solution to this problem was to use a file system checker after an unclean shutdown. However, this approach proved to be inadequate with increasingly larger file systems, and demands for system availability meant that boot times measured in hours were unacceptable.

More elegant solutions to the problem of file system consistency have been proposed and implemented. With logging or journaling, updates to the file system are written to a separate log area of the disk, and marked to show when the contents of the log are consistent. The updates are then written to the file system itself. At the next mount of the file system, the contents of the log are replayed if necessary, overwriting any half-finished

updates and resulting in a consistent file system. Soft updates carefully manage the effects of file system operations on the in-memory copies of metadata blocks on disk, and periodically write out updated metadata blocks in specific order to present a mostly consistent on-disk picture. Copy-on-write (COW) file systems never overwrite an allocated disk block; instead they allocate a new block whenever an existing block is modified and only free the old block when a completely consistent set of new blocks has been written out to disk.

Each of these solutions has advantages and drawbacks, many of which are not obvious until implementation and widespread use. In this paper, we review these advantages and drawbacks, and argue that COW techniques are the only fully consistent and maintainable solution to the problem of file system consistency in the face of crashes. We also propose doublefs, a COW-like file system that avoids some of the known drawbacks of existing COW file systems. Doublefs preallocates a second copy of all metadata, writes updates to the unused copy, and atomically switches to the new set of metadata when it is consistent on disk.

2 File system consistency techniques

2.1 File system consistency checkers

One of the most basic ways of keeping a file system consistent is to simply fix things up (or try to fix things up) after an unclean shutdown. This is, of course, the time-honored technique of using a file system checker, `fsck`, which runs whenever the system boots after an unclean shutdown and checks the entire file system for consistency and repairs the on-disk data if necessary. Unfortunately, the time to perform a file system check is proportional to the size of the entire file system, and for very large file systems this could take hours or even days, which would lead to unacceptable down times after an unclean shutdown. As a result, file systems which rely

on file system checking for consistency have been generally abandoned.

2.2 Logging

In file system logging (also known as journaling), updates to the file system are written to a separate log area as log entries. The log may be located in a special part of the disk, on another disk, in NVRAM, or some other device. Each log entry ends with a checkpoint region to show that the log entry is complete; incomplete log entries are discarded during log replay. After an update has been written to the log, the system then writes the update to the file system itself. If the update does not complete in the primary file system, the contents of the log are replayed on the next mount of the file system, overwriting any half-finished updates and producing a consistent file system. Examples include XFS, ext3, logging UFS, JFS, and VxFS.

Logging comes in two main varieties: block-level logging and intent logging. Additionally, some file systems log only metadata while others log both metadata and data. In block-level logging, each block that will be modified during a file system update is written to the log. In intent logging, a logical representation of the update recording the “intent” of the update is written to the log. For example, an intent log entry might include the operation name (“create”), the inode it modifies (“23”), and any other associated data (then name of the file, its permissions, etc.). Intent logging requires much smaller log entries, but block-level logging is sometimes simpler and often more robust in the face of errors. For example, if power is cut in the middle of writing a block, some disks will continue to write while the power supply to the contents of the track buffer memory drops below critical, resulting in garbage in the memory and therefore garbage written to the remainder of the block. Block-level logging will rewrite the entire block, covering up the error, while intent logging will only rewrite part of the block.

Logging eliminates the necessity of running `fsck` after an unclean shutdown. Instead, the outstanding log entries are replayed, which take time proportional to the number of outstanding operations when the crash occurred rather than the size of the entire file system. Logging often also results in improved file system performance despite writing the data twice. Log entries are written to sequential blocks on disk and may require fewer blocks, while the update to the primary file system usually requires multiple seeks. The system call can return after only the log entry has hit disk, reducing latency, and the subsequent in-file-system updates can then be batched, increasing throughput.

The drawbacks of logging are less obvious. The file system is not actually consistent until it has been mounted

for the first time and the log has been replayed. This usually occurs when the file system is mounted by the operating system, late in boot. Before this time, both the boot loader and the operating system itself often read files out of the inconsistent, pre-replay file system. GRUB, the most common Linux boot loader, reads the file system in order to find its configuration file and the kernel and initial ramdisk. The Solaris operating system reads a great variety of files from the file system before log replay occurs, including kernel modules and driver configuration files (the Solaris boot loader also reads the file system). The authors have never seen a case where Linux failed to boot because GRUB could not read an inconsistent file system, probably because the usual method for updating a kernel is to create an entirely new file, leaving the old one in place, and crashes don’t usually occur while updating the GRUB configuration file. We have seen multiple cases where Solaris failed to boot for this reason, mainly because Solaris reads so many different files during boot, all of which are required to exist and contain the right data. Also, using the `add_drv` command to add a driver to the system almost invites this kind of corruption, since `add_drv` updates vital system files, such as `/etc/name_to_major` at the same time it loads the driver module[2]. Often during driver development, loading the driver causes a system panic, just after the files have been updated and written to the log, but not written to the primary file system. On boot, the kernel tries to read the file using the corrupted primary file system, and gets garbage, which prevents it from finishing booting. The problem is fixed by booting some alternate media (such as a CD-ROM or netbooting) and mounting the root file system, which causes the log to be replayed.

A more subtle issue is a well-known principle of software engineering: infrequently used code is more likely to be buggy. Log replay happens only occasionally since it is error path code, and so is not as well tested as the rest of the file system code. For example, the FiSC model checker found that the log replay code in all the file systems it examined incorrectly ordered log clearing and writes to the primary file system. For the same reason, `fsck` code was also a common source of bugs in their analysis[13].

Finally, with a logging file system, there is no longer such a concept as a truly read-only mount, since the log must be replayed on an unclean file system in order to be able to read it. Read-only mounts are a useful tool, not only for file system developers, but also for file system users. For example, a system administrator which suspects that a filesystem may have been corrupted via hardware failure may wish to be able to mount the filesystem read-only to be sure that no further damage will be done to the

filesystem. Linux's ext3 filesystem will replay the journal, even on a read-only mount. This may be the right thing to do, since otherwise the invalid filesystem data-structures may cause serious system malfunctions; however, it certainly violates the principle of least surprise.

2.3 Soft Updates

Soft Updates is a technique which tracks dependencies to determine which disk blocks must be written to disk before other disk blocks, and uses delayed writes to assure that file system on disk is mostly consistent [4]. It has been popularized by an implementation written for NetBSD, FreeBSD, and its descendants [6].

In the Soft Updates approach, the file system image on disk is mostly consistent at all times (except that the allocation bitmaps may wrongly mark certain blocks and inodes as being in use when they are actually free). This is enforced by keeping track of all modifications to the file system metadata blocks, and ordering the writeback of the metadata blocks in order to guarantee this consistency guarantee. However, since an inode bitmap, or an inode table block, may be modified by multiple transactions, there may be cyclical dependencies that must be resolved by locking a disk block, rolling back operations that have dependencies that cannot yet be resolved by writing out some other block, writing the block, and then redoing the operations that had been rolled back, and finally releasing the disk buffer lock. This requires over a dozen dependency soft updates dependency tracking structures, and specialized code to evaluate dependencies and roll-back/forward changes as needed.

The performance of Soft Updates is significantly better than traditional FFS for most workloads, although there are some workloads which place a significant heavy load on the file system where Soft Updates's performance will be worse than either a journaling file system or traditional FFS. This can happen when the data set exceeds the size of the buffer cache, cache evictions will force extra writes which when rolled-back, written, and then rolled-forward, will remain dirty in the buffer cache despite being written to disk.[9]

Soft Update file systems do not strictly speaking require a file system consistency check after an unclean shutdown, although there may be freed blocks and inodes that will not become available. While the fsck can be skipped, it must be eventually done, or the file system will run out of blocks or inodes due to an increasing number of lost blocks and inodes after each unclean shutdown. The BSD implementation introduces a clever optimization in which fsck is run against a snapshot and the list of freed blocks is then applied to the running file system[6]. However, the file system consistency check still requires the disk i/o of a full fsck run, which on a

very large file system can take many hours; even if it is done while the system is running, system performance will be significantly degraded while the file system consistency check is running in the background. In some embedded applications, this degraded performance may not be acceptable.

Soft Update file systems are also significantly complicated, and hard to get right, due to the need to track dependencies and to roll back/forward logical operations to file system metadata. While this may not be a problem if sufficiently smart and careful implementors are found to implement Soft Updates; it also means that future enhancements to the file system will also require equally skilled and subtle implementors. Perhaps it is for this reason that the Soft Updates-enhanced BSD file systems still do not support some kind of B-tree or hashed directory indexing feature¹ — the complexities of handling rollback of various tree operations (in particular tree insert operations leading to interior node splits) are certainly not trivial.

2.4 Copy-on-write file systems

Copy-on-write file systems solve the file system consistency problem by never updating a currently in-use block in place. Whenever an update alters an existing block, it is copied and a new block is allocated elsewhere on the disk. Once a consistent set of new blocks has been written out to disk, the file system performs an atomic switch from the old, fully consistent set of blocks to the new, fully consistent set of blocks. This is often done by rewriting a single block which forms the root of a tree of block pointers; back-ups of this block are kept in case the block is corrupted mid-write. Once the atomic switch has been accomplished, the old blocks can be freed and reused. Examples include LFS[8], WAFL[5], and ZFS[1].

COW file systems are always consistent on-disk — no fsck or log replay needed. They can be read at any point, including during early boot prior to the first official mount of the file system. Read-only mounts and snapshots are fully supported. The COW logic is implemented once at the block level, rather than repeatedly for each kind of file system operation. No badly tested error path code is required to restore file system consistency after a crash. In our opinion, COW is the most reliable method of implementing file system consistency.

COW has its drawbacks as well. The vast LFS segment cleaner literature (e.g., [3, 7, 12] shows us the difficulty of preventing fragmentation in a COW file system, since blocks must constantly be reallocated and cannot be immediately reused. A related problem is the difficulty of

¹In contrast Linux's ext3 file system has had both journaling as well as directory indexing for several years

estimating how much space a particular write will (perhaps temporarily) require. A write to one block, e.g., extending the length of a file by one block, can easily cause changes that ripple up an entire tree of blocks. Each one of these blocks must have a new block allocated in order for the original write to complete. A write that would require only one block of disk space to be allocated in an update-in-place file system can easily result in tens or hundreds of block allocations in a COW file system, making it hard to know when the write request occurs whether there is enough space to satisfy it. For example, the original implementation of LFS for Sprite would return success for writes even when there was not enough free disk space[10]. In the authors' experience, correctly bounding on-disk space requirements is one of the trickiest problems to get right in a COW file system.

Finally, while it is simpler and less error-prone to implement COW once at the block level than to implement per-operation roll back and roll forward, it is still a difficult job. The implementor must choose a point in time at which to "freeze" the set of file system changes which will be written to disk as consistent set of blocks. Any file system operations which occur after this point must not be allowed to alter the set of blocks being written to disk, or else a "future leak" occurs.

3 General Principles

3.1 Simplicity is Good

There is a very famous quote by Brian Kernigham which applies as much to file system designs as it does to coding practices: "Debugging is twice as hard as writing the code in the first place. Therefore, if you write the code as cleverly as possible, you are, by definition, not smart enough to debug it."

As we have seen in previous sections, the complexities of Soft Updates schemes and Copy-on-write schemes are significant, which can impede new features from being added to these file systems, and make it more difficult to find and fix bugs as they are discovered. In the authors' experience, who have experience both in the Open Source development world as well as the engineering department of a proprietary UnixTM system, finding and recruiting skilled developers is extremely challenging.

As a result, it is the authors' contention that if file system complexity must be justified by *significant* improvements in performance or in features that enable significant new functionality or performance in userspace applications.

3.2 Be consistent (on disk)

It is highly desirable that the filesystem image on the storage media be completely consistent at all times. This

property allows boot loaders and filesystem debugging utilities to be able to operate reliably on filesystems without the need to run some specialized log replay code, and without needing to modify the filesystem first.

Of all of the file system consistency techniques explored in this paper, only copy on write completely satisfies this principle. Soft Updates comes close, but still require an resource-intensive filesystem check in order to release orphaned inodes and blocks. While this does not present a problem for boot loaders, the requirement to run fsck may not be convenient or appropriate on some production workloads — for example, an underpowered embedded system, or a constantly (24x7) heavily loaded file or database server.

4 COW-like file systems

Given the forgoing, we propose research into a new class of file system, dubbed COW-like file systems. The question we want to answer is, can we get most of the benefits of COW file systems (always consistent on-disk, implement consistency once for the entire file system), while avoiding the drawbacks (uncertainty of space allocation, fragmentation on writes)?

Our proposal is doublefs, which updates data blocks in place, avoiding the fragmentation problems of filesystems such as LFS. Metadata blocks are treated in a copy-on-write-like fashion, but we avoid the complexity of predicting space requirements for file system updates by preallocating a second copy of all file system metadata — the superblock, inode table, allocation bitmaps, indirect blocks, and directories. While this doubles the space used for metadata, note that logging file systems do essentially the same thing in reserving space on-disk for the log. It is also common practice for file systems to reserve 5-10% of on-disk space simply to improve performance, allow root to create files on a full file system, or to provide a cushion for errors in space allocation. Currently, an ext2 filesystem has an overhead of approximately 1.5% to 3%, depending on how many directory blocks are in use in the filesystem. With the typical disk in a personal computer only about half full, an estimated overhead of 3% to 5% seems quite reasonable.

Each block of metadata in doublefs has a 64-bit generation number, initially set to 1 at file system creation time. File system operations are, as is standard, applied to an in-memory copy of the metadata block. As each metadata block is modified, the in-memory copy is marked with a generation number which is one greater than the current valid generation number. When a sync point is triggered, either by a timeout, memory pressure, or explicit user sync request, all of the modified metadata blocks are "frozen" and are written out to disk, over-

writing the oldest copy of the metadata (the one with the oldest copy of the metadata). During this time, further filesystem updates may take place, using an incremented generation number; however, the “frozen” version of the metablock must be copied and saved, if an attempt is made to further modify that metadata block until the frozen version has been written to disk. Once the full set of blocks is written, the file system superblock is updated to show what the latest valid generation number is. Whenever a metadata block is read, both copies (which are allocated adjacent to each other) are read and the copy with the latest valid generation number is used.

After an unclean shutdown, there may be metadata blocks with a generation number of $n+1$, that represent the half-finished update that was in progress at the time of the crash. These metadata blocks must be cleared before the filesystem can be modified, since the otherwise the next generation update will allow these half-finished update from the previous boot to become visible. Therefore, we maintain a per-blockgroup list of modified metadata blocks/regions, so these half-finished updates can be cleared before the filesystem can be mounted for read/write operations.

We plan to implement doublefs using ext2 as a starting base. The ext2 filesystem is simple (less than 10,000 lines of code), robust, and has excellent performance. In addition, this will allow us to compare the performance of doubles with ext3[11], the journaling variant of ext2.

5 Conclusion

The problem of maintaining file system consistency has been approached from many directions, including post facto repair, logging, soft updates, and copy-on-write. Copy-on-write is the only technique that keeps the file system always fully consistent on-disk; however, traditional COW filesystems are highly complex beast that are difficult to implement and more difficult to debug and keep bug-free. We propose research into COW-like file systems, which are not completely copy-on-write but retain many of their nice properties. Our own COW-like file system design is doublefs, which is copy-on-write for metadata only, and pre-allocates space for two copies of all metadata. This results in a design which is simpler than traditional filesystem consistency techniques. We plan to implement doublefs using the Linux ext2 file system code as a starting point.

References

- [1] ZFS – the last word in file systems. <http://www.sun.com/2004-0914/feature/>.
- [2] *Writing Device Drivers (Solaris 10)*. <http://docs.sun.com>, 2004.
- [3] Trevor Blackwell, Jeffrey Harris, and Margo I. Seltzer. Heuristic cleaning algorithms in log-structured file systems. In *USENIX Winter*, pages 277–288, 1995.
- [4] Gregory R. Ganger and Yale N. Patt. Metadata update performance in file systems. In *OSDI*, pages 49–60, 1994.
- [5] Dave Hitz, James Lau, and Michael A. Malcolm. File system design for an nfs file server appliance. In *USENIX Winter*, pages 235–246, 1994.
- [6] Marshall K. McKusick and Gregory R. Ganger. Soft updates: A technique for eliminating most synchronous writes in the fast filesystem. In *USENIX Annual Technical Conference, FREENIX Track*, pages 1–17. USENIX, 1999.
- [7] John T. Robinson. Analysis of steady-state segment storage utilizations in a log-structured file system with least-utilized segment cleaning. *Operating Systems Review*, 30(4):29–32, 1996.
- [8] Mendel Rosenblum and John K. Ousterhout. The design and implementation of a log-structured file system. In *SOSP*, pages 1–15, 1991.
- [9] Margo Seltzer, Greg Ganger, M. Kirk McKusick, Keith Smith, Craig Soules, and Christopher Stein. Journaling versus soft updates: Asynchronous meta-data protection in file systems. In *USENIX Annual Technical Conference*, pages 18–23, June 2000.
- [10] Margo I. Seltzer, Keith Bostic, Marshall K. McKusick, and Carl Staelin. An implementation of a log-structured file system for unix. In *USENIX Winter*, pages 307–326, 1993.
- [11] Stephen Tweedie. Journaling the Linux ext2fs filesystem. In *LinuxExpo '98*, 1998.
- [12] Jun Wang and Yiming Hu. Profs-performance-oriented data reorganization for log-structured file system on multi-zone disks. In *MASCOTS*, pages 285–292, 2001.
- [13] Junfeng Yang, Paul Twohey, Dawson R. Engler, and Madanlal Musuvathi. Using model checking to find serious file system errors. In *OSDI*, pages 273–288, 2004.