Proceedings of the Linux Symposium

Volume One

July 19th–22nd, 2006
Ottawa, Ontario
Canada

Conference Organizers

Andrew J. Hutton, *Steamballoon, Inc.*
C. Craig Ross, *Linux Symposium*

Review Committee

Jeff Garzik, *Red Hat Software*
Gerrit Huizenga, *IBM*
Dave Jones, *Red Hat Software*
Ben LaHaise, *Intel Corporation*
Matt Mackall, *Selenic Consulting*
Patrick Mochel, *Intel Corporation*
C. Craig Ross, *Linux Symposium*
Andrew Hutton, *Steamballoon, Inc.*

Proceedings Formatting Team

John W. Lockhart, *Red Hat, Inc.*
David M. Fellows, *Fellows and Carr, Inc.*
Kyle McMartin

Reducing fsck time for ext2 file systems

Val Henson
Intel, Inc.
val_henson@linux.intel.com

Zach Brown
Oracle, Inc.
zach.brown@oracle.com

Theodore Ts'o
IBM, Inc.
tytso@alum.mit.edu

Arjan van de Ven
Intel, Inc.
arjan@linux.intel.com

Abstract

Ext2 is fast, simple, robust, and fun to hack on. However, it has fallen out of favor for one major reason: if an ext2 file system is not cleanly unmounted, such as in the event of kernel crash or power loss, it must be repaired using `fsck`, which takes minutes or hours to complete, during which time the file system is unavailable. In this paper, we describe some techniques for reducing the average fsck time on ext2 file systems. First, we avoid running fsck in some cases by adding a filesystem-wide dirty bit indicating whether the file system was being actively modified at the time it crashed. The performance of ext2 with this change is close to that of plain ext2, and quite a bit faster than ext3. Second, we propose a technique called linked writes which uses dependent writes and a list of dirty inodes to allow recovery of an active file system by only repairing the dirty inodes and avoiding a full file system check.

1 Introduction

The Second Extended File System, ext2, was implemented in 1993 by Remy Card, Theodore T'so, and Stephen Tweedie, and for many years was the file system of choice for Linux systems. Ext2 is similar in on-disk structure to the Berkeley FFS file system [14], with the notable exception of sub-block size fragments [2]. In recent years, ext2 has been overtaken in popularity by the ext3 [6, 16] and reiser3 [4] file systems, both journaling file systems. While these file systems are not as fast as ext2 in some cases [1], and are certainly not as simple, their recovery after crash is very fast as they do not have to run fsck.

Like the original Berkeley FFS, ext2 file system consistency is maintained on a post hoc basis, by repair after the fact using the file system checker, fsck [12]. Fsck works by traversing the entire file system and building up a consistent picture of the file system metadata, which it then writes to disk. This kind of post hoc data repair has two major drawbacks. One, it tends to be fragile. A new set of test and repair functions had to be written for every common kind of corruption. Often, fsck had to fall back to manual mode—that is, asking the human to make decisions about repairing the file system for it. As ext2 continued to be used and new tests and repairs were added to the fsck code base, this occurred less and less often, and now most users can reasonably expect fsck to com-

plete unattended after a system crash.

The second major drawback to fsck is total running time. Since fsck must traverse the entire file system to build a complete picture of allocation bitmaps, number of links to inodes, and other potentially incorrect metadata, it takes anywhere from minutes to hours to complete. File system repair using fsck takes time proportional to the size of the file system, rather than the size of the ongoing update to the file system, as is the case for journaling file systems like ext3 and reiserfs. The cost of the system unavailability while fsck is running is so great that ext2 is generally only used in niche cases, when high ongoing performance is worth the cost of occasional system unavailability and possible greater chance of data loss.

On the other hand, ext2 is fast, simple, easy to repair, uses little CPU, performs well with multi-threaded reads and writes, and benefits from over a decade of debugging and fine tuning. Our goal is to find a way to keep these attributes while reducing the average time it takes to recover from crashes—that is, reducing the average time spent running fsck. Our target use case is a server with many users, infrequent writes, lots of read-only file system data, and tolerance for possibly greater chance of data loss.

Our first approach to reducing fsck time is to implement a filesystem-wide dirty bit. While writes are in progress, the bit is set. After the file system has been idle for some period of time (one second in our implementation), we force out all outstanding writes to disk and mark the file system as clean. If we crash while the file system is marked clean, fsck knows that it does not have to do a full fsck. Instead, it does some minor housekeeping and marks the file system as valid. Orphan inodes and block preallocation added some interesting twists to this solution, but overall it remains a simple change. While this approach does not improve worst case fsck time, it does improve average fsck time. For comparison purposes, recall that ext3 runs a full fsck on the file system every 30 mounts as usually installed.

Our second approach, which we did not implement, is an attempt to limit the data fsck needs to examine to repair the file system to a set of dirty inodes and their associated metadata. If we add inodes to an on-disk dirty inode list before altering them and correctly order metadata writes to the file system, we will be able to correct allocation bitmaps, directory entries, and inode link counts without rebuilding the entire file system, as fsck does now.

Some consistency issues are difficult to solve without unsightly and possibly slow hacks, such as keeping the number of links consistent for a file with multiple hard links during an `unlink()` operation. However, they occur relatively rarely, so we are considering combining this approach with the filesystem-wide dirty bit. When a particular operation is too ugly to implement using the dirty inode list, we simply mark the file system as dirty for the duration of the operation. It may be profitable to merely narrow the window during which a crash will require a full fsck rather than to close the window fully. Whether this can be done and still preserve the properties of simplicity of implementation and high performance is an open question.

2 Why ext2?

Linux has a lot of file systems, many of which have better solutions for maintaining file system consistency than ext2. Why are we working on improving crash recovery in ext2 when so many other solutions exist? The answer is a combination of useful properties of ext2 and drawbacks of existing file systems.

First, the advantages of ext2 are simplicity, robustness, and high performance. The entire ext2 code base is about 8,000 lines of code; most programmers can understand and begin altering the codebase within days or weeks. For comparison, most other file systems come in anywhere from 20,000 (ext3 + jbd) to 80,000 (XFS) lines of code. Ext2 has been in active use since 1993, and benefits from over a decade of weeding out bugs and repairing obscure and seldom seen failure cases. Ext2 performance is quite good overall, especially considering its simplicity, and definitely superior to ext3 in most cases.

The main focus of file systems development in Linux today is ext3. On-disk, ext3 is almost identical to ext2; both file systems can be mounted as either ext2 or ext3 in most cases. Ext3 is a journalled file system; updates to the file system are first written as compact entries in the on-disk journal region before they are written to their final locations. If a crash occurs during an update, the journal is replayed on the next mount, completing any unfinished updates.

Our primary concern with ext3 is lower performance from writing and sharing the journal. Work is being done to improve performance, especially in the area of multi-threaded writes [9], but it is hard to compete in performance against a file system which has little or no restrictions in terms of sharing resources or write ordering. Our secondary concern is complexity of code. Journaling adds a whole layer of code to open transactions, reserve log space, and bail out when an error occurs. Overall, we feel that ext3 is a good file system for laptops, but not very good for write-intensive loads.

The reiser3 [4] file system is the default file system for the SuSE distribution. It is also a journaling file system, and is especially good for file systems with many small files because it packs the file together, saving space. The performance of reiser3 is good and in some cases better than ext2. However, reiser3 was developed outside the mainstream Linux community and never attracted a community developer base. Because of this and the complexity of the implementation, it is not a good base for file system development. Reiser4 [4] has less developer buy-in, more code, worse performance in many cases [1], and may not be merged into the mainline Linux tree at all [3].

XFS is another journaling file system. It has many desirable properties, and is ideal for applications requiring thousands or millions of files in one directory, but also suffers from complexity and lack of developer community. Performance of more common case operations (such as file create) suffers for the benefit of fast look ups in directories with many entries [1].

Other techniques for maintaining file system complexity are soft updates [13] and copy-on-write [11]. Without a team of full-time programmers and several years to work on the problem, we did not feel we could implement either of these techniques. In any case, we did not feel we could maintain the simplicity or the benefits of more than a decade of testing of ext2 if we used these techniques.

Given these limitations, we decided to look for "90% solutions" to the file system consistency problem, starting with the ext2 code base. This paper describes one technique we implemented, the filesystem-wide dirty bit, and one we are considering implementing, linked writes.

3 The fsck program

Cutting down crash recovery time for an ext2 file system depends on understanding how the

file system checker program, fsck works. After Linux has finished booting the kernel, the root file system is mounted read-only and the kernel executes the init program. As part of normal system initialization, fsck is run on the root file system before it is remounted read-write and on other file systems before they are mounted. Repair of the file system is necessary before it can be safely written.

When fsck runs, it checks to see if the ext2 file system was cleanly unmounted by reading the state field in the file system superblock. If the state is set as VALID, the file system is already consistent and does not need recovery; fsck exits without further ado. If the state is INVALID, fsck does a full check of the file system integrity, repairing any inconsistencies it finds. In order to check the correctness of allocation bitmaps, file nlinks, directory entries, etc., fsck reads every inode in the system, every indirect block referenced by an inode, and every directory entry. Using this information, it builds up a new set of inode and block allocation bitmaps, calculates the correct number of links of every inode, and removes directory entries to unreferenced inodes. It does many other things as well, such as sanity check inode fields, but these three activities fundamentally require reading every inode in the file system. Otherwise, there is no way to find out whether, for example, a particular block is referenced by a file but is marked as unallocated on the block allocation bitmap. In summary, there are no back pointers from a data block to the indirect block that points to it, or from a file to the directories that point to it, so the only way to reconstruct reference counts is to start at the top level and build a complete picture of the file system metadata.

Unsurprisingly, it takes fsck quite some time to rebuild the entirety of the file system metadata, approximately O(total file system size + data stored). The average laptop takes several minutes to fsck an ext2 file system; large file servers can sometimes take hours or, on occasion, days! Straightforward tactical performance optimizations such as requesting reads of needed blocks in sequential order and read-ahead requests can only improve the situation so much, given that the whole operation will still take time proportional to the entire file system. What we want is file system recovery time that is O(writes in progress), as is the case for journal replay in journaling file systems.

One way to reduce fsck time is to eliminate the need to do a full fsck at all if a crash occurs when the file system is not being changed. This is the approach we took with the filesystem-wide dirty bit.

Another way to reduce fsck time is to reduce the amount of metadata we have to check in order to repair the file system. We propose a method of ordering updates to the file system in such a way that full consistency can be recovered by scanning a list of dirty inodes.

4 Implementation of filesystem-wide dirty bit

Implementing the fs-wide dirty bit seemed at first glance to be relatively simple. Intuitively, if no writes are going on in the file system, we should be able to sync the file system (make sure all outstanding writes are on disk), reset the machine, and cleanly mount the unchanged file system. Our intuition is wrong in two major points: orphan inodes, and block preallocation. Orphan inodes are files which have been unlinked from the file system, but are still held open by a process. On crash and recovery, the inode and its blocks need to be freed. Block preallocation speeds up block allocation by preallocating a few more blocks than were actually

requested. Unfortunately, as implemented, preallocation alters on-disk data, which needs to be corrected if the file is not cleanly closed. First we'll describe the overall implementation, then our handling of orphan inodes and preallocated blocks.

4.1 Overview of dirty bit implementation

Our first working patch implementing the fs-wide dirty bit included the following high-level changes:

- Per-mount kernel thread to mark file system clean
- New `ext2_mark_*_dirty()` functions
- Port of ext3 orphan inode list
- Port of ext3 reservation code

The ports of the ext3 orphan inode list and reservation code were not frivolous; without them, the file system would be an inconsistent state even when no writes were occurring without them.

4.2 Per-mount kernel thread

The basic outline of how the file system is marked dirty or clean by the per-mount kernel thread is as follows:

- Mark the file system dirty whenever metadata is altered.
- Periodically check the state of the file system.
- If the file system is clean, sync the file system.
- If no new writes occurred during the sync, mark the file system clean.

The file system is marked clean or dirty by updating a field in the superblock and submitting the I/O as a barrier write so that no writes can pass it and hit the disk before the dirty bit is updated. The update of the dirty bit is done asynchronously, so as to not stall during the first write to a clean file system (since it is a barrier write, waiting on it will not change the order of writes to disk anyway).

In order to implement asynchronous update of the dirty bit in the superblock, we needed to create an in-memory copy of the superblock. Updates to the superblock are written to the in-memory copy; when the superblock is ready to be written to disk, the superblock is locked, the in-memory superblock is copied to the buffer for the I/O operation, and the I/O is submitted. The code implementing the superblock copy is limited to the files `ext2/super.c` and one line in `ext2/xattr.c`.

One item on our to-do list is integration with the laptop mode code, which tries to minimize the number of disk spin-up and spin-down events by concentrating disk write activity into batches. Marking the file system clean should probably be triggered by the timeout for flushing dirty data in laptop mode.

4.3 Marking the file system dirty

Before any metadata changes are scheduled to be written to disk, the file system must first be marked dirty. Ext2 already uses the functions `mark_inode_dirty()`, `mark_buffer_dirty()`, and `mark_buffer_dirty_inode()` to mark changed metadata for write-out by the VFS and I/O subsystems. We created ext2-specific

versions of these functions which first mark the file system dirty and then call the original function.

4.4 Orphan inodes

The semantics of UNIX file systems allow an application to create a file, open it, unlink the file (removing any reference to it from the file system), and keep the file open indefinitely. While the file is open, the file system can not delete the file. In effect, this creates a temporary file which is guaranteed to be deleted, even if the system crashes. If the system does crash while the file is still open, the file system contains an orphan inode—an inode which marked as in use, but is not referenced by any directory entry. This behavior is very convenient for application developers and a real pain in the neck for file system developers, who wish they would all use files in tmpfs instead.

In order to clean up orphan inodes after a crash, we ported the ext3 orphan inode list to ext2. The orphan inode list is an on-disk singly linked list of inodes, beginning in the orphan inode field of the superblock. The `i_dtime` field of the inode, normally used to store the time an inode was deleted, is (ab)used as the inode number of the next item in the orphan inode list. When fsck is run on the file system, it traverses the linked list of orphan inodes and frees them. Fortunately for us, the code in fsck that does this runs regardless of whether the file system is mounted as ext2 or ext3.

Our initial implementation followed the ext3 practice of writing out orphan inodes immediately in order to keep the orphan inode list as up-to-date as possible on disk. This is expensive, and an up-to-date orphan inode list is superfluous except when the file system is marked clean. We modified the orphan inode code to only maintain the orphan inode list in memory, and write it out to disk on file system sync. We will need to add a patch to keep fsck from complaining about a corrupted orphan inode list.

4.5 Preallocated blocks

The existing code in ext2 for preallocating blocks unfortunately alters on-disk metadata, such as the block group free and allocated block counts. One solution was to simply turn off preallocation. Fortunately, Mingming Cao implemented new block preallocation code for ext3 which reserves blocks without touching on-disk data, and is superior to the ext2 preallocation code in several other ways. We chose to port Mingming Cao's reservation code to ext2, which in theory should improve block allocation anyway. ext2 and ext3 were similar enough that we could complete the port quickly, although porting some parts of the new `get_blocks()` functionality was tricky.

4.6 Development under User-mode Linux

We want to note that the implementation of the filesystem-wide dirty bit was tested almost entirely on User-mode Linux [10], a port of Linux that runs as a process in Linux. UML is well suited to file system development, especially when the developer is limited to a single laptop for both development host and target platform (as is often the case on an airplane). With UML, we could quickly compile, boot, crash, and reboot our UML instance, all without worrying about corrupting any important file systems. When we did corrupt the UML file system, all that was necessary was to copy a clean file system image back over the file containing the UML file system image. The loopback device made it easy to mount, fsck, or otherwise examine the UML file system using tools on the host machine. Only one bug required running

on a non-UML system to discover, which was lack of support for suspend in the dirty bit kernel thread.

However, getting UML up and running and working for file system development was somewhat non-intuitive and occasionally baffling. Details about running UML on recent 2.6 kernels, including links to a sample root file system and working `.config` file, can be found here:

`http://www.nmt.edu/~val/uml_tips.html`

5 Performance

We benchmarked the filesystem-wide dirty bit implementation to find out if it significantly impacted performance. On the face of it, we expected a small penalty on the first write, due to issuing an asynchronous write barrier the first time the file system is written.

The benchmarks we ran were *kuntar*, *postmark*, and *tiobench* [7]. Kuntar simply measures the time to extract a cached uncompressed kernel tarball and sync the file system. Postmark creates and deletes many small files in a directory and is a metadata intensive workload. We ran it with `numbers = 10000` and `transactions = 10000`. We also added a `sync()` system call to postmark before the final timing measurement was made, in order to measure the true performance of writing data all the way to the disk. Tiobench is a benchmark designed to measure multi-threaded I/O to a single file; we ran it mainly as a sanity check since we didn't expect anything to change in this workload. We ran tiobench with 16 threads and a 256MB file size.

The file systems we benchmarked were ext2, ext2 with the reservations-only patch, ext2 with reservations only with reservations turned off, ext2 with the fs-wide bit patch and reservations, ext3 with defaults, and ext3 with `data=writeback` mode. All file systems used 4KB blocks and were mounted with the `noatime` option. The kernel was 2.6.16-mm1. The machine had two 1533 MHZ AMD Athlon processors and 1GB of memory. We recored elapsed time, sectors read, sectors written, and kernel ticks. The results are in Table 1.

The results are somewhat baffling, but overall positive for the dirty bit implementation. The times for the fs-wide dirty bit are within 10% of those of plain ext2 for all benchmarks except postmark. For postmark, writes increased greatly for the fs-wide dirty bit; we are not sure why yet. The results for the reservations-only versions of ext2 are even more puzzling; we suspect that our port of reservations is buggy or suboptimal. We will continue researching the performance issues.

We would like to briefly discuss the `noatime` option. All file systems were mounted with the `noatime` option, which turns off updates to the "last accessed time" field in the inode. We turned this option off not only because it would prevent the fs-wide dirty bit from being effective when a file system is under read activity, but also because it is a common technique for improving performance. `noatime` is widely regarded as the correct behavior for most file systems, and in some cases is shipped as the default behavior by distributions. While correct access time is sometimes useful or even critical, such as in tracing which files an intruder read, in most cases it is unnecessary and only adds unnecessary I/O to the system.

6 Linked writes

Our second idea for reducing fsck time is to order writes to the file system such that the file

		ext2	ext2r	ext2rnor	ext2fw	ext3	ext3wb
kuntar	secs	20.32	21.03	19.06	18.87	20.99	32.02
	read	5152	5176	5176	5176	168	168
	write	523272	523272	523288	523304	523256	544160
	ticks	237	269	357	277	413	402
krmtar	secs	9.79	10.92	9.99	10.90	55.64	9.74
	read	20874	20842	20874	20874	20866	20874
	write	5208	5176	5208	5960	36296	10560
	ticks	61	61	62	61	7943	130
postmark	secs	33.98	49.34	42.93	50.46	43.48	41.82
	read	2568	2568	2568	2568	56	48
	write	168312	168392	168392	240720	260704	173936
	ticks	641	650	838	674	1364	1481
tiobench	secs	37.48	35.22	33.68	33.57	35.16	36.69
	read	32	32	32	32	24	112
	write	64	64	64	72	136	136
	ticks	441	450	456	463	452	463

kuntar: expanding a cached uncompressed kernel tarball and syncing
krmtar: rm -rf on cold untarred kernel tree, sync
postmark: postmark + sync() patch, numbers = 10000, transactions = 10000
tiobench: tiobench: 16 threads, 256m size
ext2: ext2
ext2r: ext2, reservations
ext2rnor: ext2, reservations, -o noreservation option
ext2fw: ext2, reservations, fswide
ext3: ext3, 256m journal
ext3wb: ext3, 256m journal, data=writeback

Table 1: Benchmark results

system can repaired to a consistent state after processing a short list of dirty inodes. Before an operation begins, the relevant inodes are added to an on-disk list of dirty inodes. During the operation, we only overwrite references to data (such as indirect blocks or directory entries) after we have finished all updates that require that information (such as updating allocation bitmaps or link counts). If we crash half-way through an operation, we examine each inode on the dirty inode list and repair any consistencies in the metadata it points to. For example, if we were to crash half-way through allocating a block, we would check if each block were marked as allocated in the block allocation bitmap. If it was not, we would free that block from the file (and all blocks that it points to). We call this scheme *linked writes*—a write erasing a pointer is linked or dependent on the write of another block completing first.

Some cases are ambiguous as to what operation was in progress, such as truncating and extending a file. In these cases, we will take the safest action. For example, in an ambiguous truncate/extend, we would assume a truncate operation was in progress, because if we were wrong,

the new block would contain uninitialized data, resulting in a security hole. It might be possible to indicate which operation was in progress using other metadata, such as inode size, but if that is not possible or would harm performance, we have this option as a fail safe. The difference between restoring one or the other of two ambiguous operations is the difference between restoring the file as of a short time before the crash versus restoring it as of after the completion of the operation in progress at the time of crash. Either option is allowed; only calling `sync()` defines what state the file is in on-disk at any particular moment.

Some operations may not be recoverable only by ordering writes. Consider removing one hard link to a file with multiple hard links from different directories. The only inodes on the dirty inode list are the inode for the directory we are removing the link from, and the file inode—not the inodes for the other directories with hard links to this file. Say we decrement the link count for the inode, and then crash. In the one link case, when we recover, we will find an inode with link count equal to 0, and a directory with an entry pointing to this inode. Recovery is simple; free the inode and delete the directory entry. But if we have multiple hard links to the file, and the inode has a link count of one or more, we have no way of telling whether the link count was already decremented before we crashed or not. A solution to this is to overwrite the directory entry with an invalid directory entry with a magic record that contains the inode's correct link count which is only replayed if the inode has not already been updated. This regrettably adds yet another linked write to the process of deleting an entry. On the other hand, adding or removing links to files with link counts greater than one is painful but blessedly uncommon. Typically only directories have a link count greater than one, and in modern Linux, directory hard links are not allowed, so a directory's link count can be recalculated simply by scanning the directory itself.

Another problem is circular dependencies between blocks that need to be written out. Say we need to write some part of block A to disk before we write some part of block B. We update the buffers in memory and mark them to be written out in order A, B. But then something else happens, and now we need to write some part of block B to disk before some part of block A. We update the buffers in memory—but now we can't write either block A or block B. Linked writes doesn't run into this problem because (a) every block contains only one kind of metadata, (b) the order in which different kinds of metadata must be written is the same for every option. This is equivalent to the lock ordering solution to the deadlock problem; if you define the order for acquiring locks and adhere to it, you can't get into a deadlock.

Ordinarily, writing metadata in the same order according to type for all operations would not be possible. Consider the case of creating a file versus deleting it. In the create case, we must write the directory entry pointing to the inode before updating the bitmap in order to avoid leaking an inode. In the delete case, we must write the bitmap before we delete the entry to avoid leaking an inode. What gets us out of this circular dependency is the dirty inode list. If we instead put the inode to be deleted on the dirty inode list, then we can delete the directory entry before the bitmap, since if we crash, the inode's presence on the dirty inode list will allow us to update the bitmap correctly. This allows us to define the dependency order "write bitmaps before directory entries." The order of metadata operations for each operation must be carefully defined and adhered to.

When writing a buffer to disk, we need to be sure it does not change in flight. We have two options for accomplishing this: either lock the buffer and stall any operations that need to

write to it while it is in flight, or clone the buffer and send the copy to disk. The first option is what soft updates uses [13]; surprisingly performance is quite good so it may be an option. The second option requires more memory but would seem to have better performance.

Another issue is reuse of freed blocks or inodes before the referring inode is removed from the dirty inode list. If we free a block, then reuse it before the inode referring to it is removed from the dirty list, it could be erroneously marked as free again at recovery time. To track this, we need a temporary copy of each affected bitmap showing which items should not be allocated, in addition to the items marked allocated in the main copy of the bitmap. Overall, we occasionally need three copies of each active bitmap in memory. The required memory usage is comparable to that of journaling, copy-on-write, or soft updates.

6.1 Implementing write dependencies

Simply issuing write barriers when we write the first half of a linked write would be terribly inefficient, as the only method of implementing this operation that is universally supported by disks is: (1) issue a cache flush command; (2) issue the write barrier I/O; (3) wait for the I/O to complete; (4) issue a second cache flush command. (Even this implementation may be an illusion; reports of IDE disks which do not correctly implement the cache flush command abound.) This creates a huge bubble in the I/O pipeline. Instead, we want to block only the dependent write. This can be implemented using asynchronous writes which kick off the linked write in the I/O completion handler.

6.2 Comparison of linked writes

Linked writes bears a strong resemblance to soft updates [13]. Indeed, linked writes can be thought of as soft updates from the opposite direction. Soft updates takes the approach of erring on the side of marking things allocated when they are actually free, and then recovering leaked inodes and blocks after mount by running a background fsck on the file system. Linked writes errs on the side of marking things unallocated when they are still referenced by the file system, and repairing inconsistencies by reviewing a list of dirty inodes. Soft updates handles circular buffer dependencies (where block A must be written out before block B and vice versa) by rolling back the dependent data before writing the block out to disk. Linked writes handle circular dependencies by making them impossible.

Linked writes can also be viewed as a form of journaling in which the journal entries are scattered across the disk in the form of inodes and directory entries, and linked together by the dirty inode list. The advantages of linked writes over journaling is that changes are written once, no journal space has to be allocated, writes aren't throttled by journal size, and there are no seeks to a separate journal region.

6.3 Reinventing the wheel?

Why bother implementing a whole new method of file system consistency when we have so many available to us already? Simply put, frustration with code complexity and performance. The authors have had direct experience with the implementation of ZFS [8], ext3 [6], and ocfs2 [5] and were disappointed with the complexity of the implementation. Merely counting lines of code for reiser3 [4], reiser4 [4], or XFS [15] incites dismay. We have not yet encountered someone other than the authors of the original soft updates [13] implementation who claims to understand it well enough to re-implement from scratch. Yet ext2, one of the smallest, simplest file systems out there, continues to be

the target for performance on general purpose workloads.

In a sense, ext2 is cheating, because it does not attempt to keep the on-disk data structures intact. In another sense, ext2 shows us what our rock-bottom performance expectations for new file systems should be, as relatively little effort has been put into optimizing ext2.

With linked writes, we hope for a file system a little more complex, a lot more consistent, and nearly the same performance as ext2.

6.4 Feasibility of linked writes implementation

We estimate that implementing linked writes would take on the order of half the effort necessary to implement ext3. Adjusting for programmer capability and experience (translation: I'm no Kirk McKusick or Greg Ganger), we estimate that implementing linked writes would take one fifth the staff-years required by soft updates.

We acknowledge that the design of linked writes is half-finished at best and may end up having fatal flaws, nor do we expect our design to survive implementation without major changes—"There's many a slip 'twixt cup and lip."

7 Failed ideas

Linked writes grew out of our original idea to implement per-block group dirty bits. We wanted to restrict how much of the file system had to be reviewed by fsck after a crash, and dividing it up by block groups seemed to make sense. In retrospect, we realized that the only checks we could do in this case would start with the inodes in this block group and check file system consistency based on the information they point to. On the other hand, given a block allocation bitmap, we can't check whether a particular block is correctly marked unless we rebuild the entire file system by reading all of the inodes. In the end, we realized that per-bg dirty bits would basically be a very coarse hash of which inodes need to be checked. It may make sense to implement some kind of bitmap showing which inodes need to be checked rather than a linked list, otherwise this idea is dead in the water.

Another idea for handling orphan inodes was to implement a set of "in-memory-only" bitmaps that record inodes and blocks which are allocated only for the lifetime of this mount—in other words, orphan inodes and their data. However, these bitmaps would in the worst case require two blocks per cylinder group of unreclaimable memory. A workaround would be to allocate space on disk to write them out under memory pressure, but we abandoned this idea quickly.

8 Availability

The most recent patches are available from:

`http://www.nmt.edu/~val/patches.html`

9 Future work

The filesystem-wide dirty bit seems worthwhile to polish for inclusion in the mainline kernel, perhaps as a mount option. We will continue to do work to improve performance and test correctness.

Implementing linked writes will take a significant amount of programmer sweat and may not

be considered, shall we say, business-critical to our respective employers. We welcome discussion, criticism, and code from interested third parties.

10 Acknowledgments

Many thanks to all those who reviewed and commented on the initial patches. Our work was greatly reduced by being able to port the orphan inode list from ext3, written by Stephen Tweedie, as well as the ext3 reservation patches by Mingming Cao.

11 Conclusion

The filesystem-wide dirty bit feature allows ext2 file systems to skip a full fsck when the file system is not being actively modified during a crash. The performance of our initial, untuned implementation is reasonable, and will be improved. Our proposal for linked writes outlines a strategy for maintaining file system consistency with less overhead than journaling and simpler implementation than copy-on-write or soft updates.

We take this opportunity to remind file system developers that ext2 is an attractive target for innovation. We hope that developers rediscover the possibilities inherent in this simple, fast, extendable file system.

References

[1] Benchmarking file systems part II LG #122. `http://linuxgazette.net/122/piszcz.html`.

[2] Design and implementation of the second extended filesystem. `http://e2fsprogs.sourceforge.net/ext2intro.html`.

[3] Linux: Reiser4 and the mainline kernel. `http://kerneltrap.org/node/5679`.

[4] Namesys. `http://www.namesys.com/`.

[5] OCFS2. `http://oss.oracle.com/projects/ocfs2/`.

[6] Red hat's new journaling file system: ext3. `http://www.redhat.com/support/wpapers/redhat/ext3/`.

[7] Threaded I/O tester. `http://sourceforge.net/projects/tiobench`.

[8] ZFS at OpenSolaris.org. `http://www.opensolaris.org/os/community/zfs/`.

[9] Mingming Cao, Theodore Y. Ts'o, Badari Pulavarty, Suparna Bhattacharya, Andreas Dilger, and Alex Tomas. State of the art: Where we are with the ext3 filesystem. In *Ottawa Linux Symposium 2005*, July 2005.

[10] Jeff Dike. User-mode linux. In *Ottawa Linux Symposium 2001*, July 2001.

[11] Dave Hitz, James Lau, and Michael A. Malcolm. File system design for an nfs file server appliance. In *USENIX Winter*, pages 235–246, 1994.

[12] T. J. Kowalski and Marshall K. McKusick. Fsck - the UNIX file system check program. Technical report, Bell Laboratories, March 1978.

[13] 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.

[14] Marshall K. McKusick, William N. Joy, Samuel J. Leffler, and Robert S. Fabry. A fast file system for unix. *ACM Trans. Comput. Syst.*, 2(3):181–197, 1984.

[15] Adam Sweeney, Doug Doucette, Wei Hu, Curtis Anderson, Michael Nishimoto, and Geoff Peck. Scalability in the XFS file system. In *Proceedings of the 1996 USENIX Technical Conference*, 1996.

[16] Stephen Tweedie. Journaling the Linux ext2fs filesystem. In *LinuxExpo '98*, 1998.