Repair–Driven File System Design

Valerie Henson
Open Source Technology Center
Intel Corporation
val_henson@linux.intel.com

File system design has traditionally focused on the on–line performance of the file system while the performance and reliability of file system check, repair, and restore has been for the most part neglected. Journaling and other techniques handle the most common case of file system corruption—half–finished updates interrupted by system crash or other event—but repairing any other form of metadata corruption requires reading and checking the entire file system's metadata. Unfortunately, disk capacity is growing faster than disk bandwidth, seek times are hardly budging, and the overall chance of an I/O error occurring somewhere on the disk is increasing. The result: the traditional file system check and repair cycle will be not only longer, but also more frequent, with disastrous consequences for data availability. Data reliability will also decline with frequency of corruption. With this reality in mind, we propose repair–driven file system design, the practice of designing file systems with the explicit goal of fast and reliable file system check, repair, and restore.

1 Introduction

File system repair is usually an afterthought for file system designers. In our experience with ZFS[2], file system programmers tend not to be excited about writing fsck; XFS was even initially shipped without any file system repair tool at all[13]. One reason is that repair is difficult and annoying to reason about. It's neither possible nor worthwhile to fix all error modes so we must focus our efforts on the ones that commonly occur, yet we don't know what they are until we encounter them in the wild. In practice, most file system repair code is written in response to an observed corruption mode. File system repair is annoying because, by definition, something went wrong and we must think outside the state space of our beautifully designed system. In the end, designing a file system is more fun than designing a file system checker.

	2006	2009	2013	Change
Capacity (GB)	500	2000	8000	16x
Bandwidth (Mb/s)	1000	2000	5000	5x
Seek time (ms)	8	7.2	6.5	1.2x

Table 1: Projected disk hardware trends[9].

For many years, we could brush off the importance of making file system repair fast and reliable with the following chain of reasoning: File system corruption is a rare event, and when it does occur, repairing it takes only a few minutes or maybe a few hours of downtime, and if repair is too difficult or time–consuming, "That's what backups are for." Unfortunately, if this reasoning was ever valid, it is being eroded by some inconvenient truths about disk hardware trends.

As Table 1 shows, Seagate projects that during the same time that disk capacity increases by 16 times, disk bandwidth will increase by only 5 times, and seek times will remain nearly unchanged. This is good news for many common workloads—we can store more data and read and write more of it at once. But it is terrible news for any workload that is (a) proportional to the size of the disk, (b) throughput–intensive, and (c) seek–intensive. One workload that fits this profile is file system check and repair. We calculate that file system check and repair time will increase by approximately a factor of 10 between 2006 and 2013 with today's file system formats (see § 2.2).

At the same time that capacity is increasing, the per–bit error rate is improving. However, for an overall improvement in the error rate for operations that read data proportional to the file system size (such as fsck), the per–bit error rate must improve as fast as capacity grows, which seems unlikely. We conclude that the frequency of file system corruption and necessary check and repair or restore is more likely to increase than decrease. This

combination of increasing fsck time and increasing fsck frequency is what we call the **fsck time crunch**.

Our response to this approaching train wreck is to propose **repair–driven file system design**: the practice of designing file systems with repair and recovery as a first–class goal. The on–disk format should be designed for performance and reliability for both normal on–line workloads and for repair workloads. While we cannot repair every possible corruption, we can design on–disk formats that are robust and fast to repair.

2 Motivation

To make a convincing argument for repair–driven file system design, we must first investigate fsck in depth: how it works, what the major performance factors are, how performance can be improved. (For more depth, see [8].) The short version is that most checkers operate in several passes, each of which checks a different kind of consistency and/or builds up data structures for later passes. The things fsck checks may include superblock sanity, block group summaries, intra–inode consistency, inode link counts, block/inode allocation bitmaps, directory connectivity, orphaned inodes, directory consistency, duplicate block allocations, indirect blocks/extents/trees, and checksums. To complete all of these checks, fsck must read every piece of metadata in the file system—every inode, every bitmap, every directory entry, every indirect block, every block group summary.

2.1 Fsck performance

The fundamental limiting factors in the performance of fsck are amount of data read, number of separate I/Os, how scattered the data is on disk, number of dependent reads, and CPU time required to check and repair the data read. The amount of memory available is a factor as well, though most fsck programs operate on an all–or–nothing basis: Either there is enough memory to fit all the needed metadata for a particular checking pass or the checker simply aborts.

The time required to read the file system metadata is partially constrained by the bandwidth of the disk. Depending on the file system, some kinds of file system data, such as blocks of statically allocated inodes or block group summaries, are located in contiguous chunks at known locations. Reading this data off disk is relatively swift.

Other kinds of file system data are dynamically allocated, such as directory entries, indirect blocks, and extents, and hence are scattered all over the disk. Many modern file systems allocate nearly all their metadata dynamically. The location of much of this kind of metadata is not known until the block of metadata pointing to it is read off the disk, introducing many levels of dependent reads. This portion of the file system check is usually the most time consuming, as we must issue a set of small scattered reads, wait for them to complete, read the address of the next block, then issue another set of reads. Some optimization of this stage has been done; mainly grouping sets of independent I/Os together, reordering them, and issuing them in order to the disk to minimize seeks. However, good disk performance requires issuing many closely grouped I/Os at once; as long as we don't know which blocks we need to read it is difficult to optimize the manner in which we retrieve them.

Finally, we need CPU time and sufficient memory to actually compute the consistency checks on the data we have read off disk. This takes a relatively small amount of time compared to the time spent doing what are effectively random 4 KB or similar–sized reads, although more complex file systems may burn more CPU time in computing checksums or similar tasks.

In summary, the ways to reduce fsck time, in rough order of effectiveness, are to reduce seeks, reduce dependent reads, reduce the amount of metadata that needs to be read (either by reducing the overall quantity or the amount that needs to be read), and to reduce the complexity of the consistency checks themselves.

2.2 Projecting file system check time

To get a rough estimate of how 16x capacity increase, 5x bandwidth increase, and almost no change in seek time will affect fsck time in 2013, we ran fsck on the /home partition (ext3 formatted) of a development laptop (see Table 2 for details on the disk used). We measured the elapsed time and CPU time using the `time` command, and the number of individual I/O operations and the total data read using `iostat`. Using this data and projected changes in disk hardware, we made a rough estimate of the time needed to complete a file system check on a moderately sized laptop file system in 2013.

First, we estimated how the elapsed time is divided between using the CPU, reading data off the disk, and head travel between blocks (seek time plus rotational latency). The total elapsed time to check a 37 GB file system with 21 GB of data is 450 seconds, 15 seconds of which are spent in CPU time. That leaves 435 seconds in I/O. We measured 1.3GB of data read. We estimated the amount of time to read 1.3 GB of data off the disk by using `dd` to transfer that amount of data from the partition into /dev/null, which took 42 seconds at optimal read size. The remaining time, 393 seconds, we assume is spent seeking between tracks and waiting for the data to rotate under the head. We measured 340,000 separate I/O requests. Given that the time spent seeking is 393 seconds, the average seek time for this disk is 12 ms, and the average rotational latency is 7.4 ms, we estimate that fsck required about $393/(0.010 + 0.0074) \approx 20000$ seeks (about one seek per every 15-20 I/Os).

To estimate how long fsck will take in 2013, we use

Total size	80 GB
Partition size	40 GB
Used	23 GB
Files	850,000
Avg. seek time	12 ms
Avg. rotational latency	7.4 ms

Table 2: Experiment disk characteristics (GB $= 10^9$ bytes

Year	Elapsed time	I/Os	Data read
2006	7.5 m	340,000	1.3 GB
2013 (est.)	80 m	5,400,000	21 GB

Table 3: Projected fsck time.

Seagate's projections for changes in capacity, bandwidth, average seek time, and average rotational latency. We generously assume that the CPU time is negligible. We assume that the file system is 16 times bigger, and checking requires reading 16 times the data and 16 times the I/Os. We'll assume that bandwidth is 5 times bigger, and that the seek time is 10 ms, 1.2 times faster than 12 ms. The trend in rotational latency is about a 1.5x improvement[9], so we will assume a 4.7 ms rotational latency.

The time required to read the data off the disk is $42s \times (16/5) \approx 130s$. The time required to execute the seeks is $16 \times 20000 \times (0.010s + 0.0047s) \approx 4700s$. The total time is $4700s + 130s \approx 4800s$, or about 80 minutes, compared to about 7.5 minutes today, an increase of a factor of about 10.

The above experiment is also best case in that ext3 is an older file system with lots of statically allocated metadata and no checksums or complex structures. Most modern file systems have more dynamically allocated metadata and take more CPU time to verify their more complex on–disk structures. Running fsck on a full 300 GB reiserfs file system with 60,000,000 files takes about 2.5 hours! In any case, we do not expect that a 10–fold increase in the time it takes to check and repair file systems will be acceptable to file system users.

2.3 Increasing file system corruption

Disk manufacturers currently specify an uncorrectable read error rate (UER) of 1 in 10^{13} to 10^{16} bits read[6], depending on quality, or one per 1–1,000 TB of data read—far from a shockingly large amount of data to read by today's standards. Empirical measurements of disk reliability vary from 1-2 orders of magnitude below specification for a small sample size ($n \approx 20$)[6] to a 1% per year rate of disk replacement due to "substantial sector failure" for a sample size in the thousands[12]. Improvements in the specified UER are predicted[4], but their relationship to shipped product is uncertain. In any case, we cannot depend on an improvement in the error rate when the consequences of failure are hours of downtime on even modestly–sized file systems, and days on any significantly sized file system. Along with hardware errors, increases in the complexity of file systems bring their own inevitable increase in file system corruption due to bugs.

Complete disk failure has been measured at a rate of 1-6% per year, depending on disk batch[12, 6], making speedy file system restore from backup a priority. The author's web site was recently down for a week when the RAID system on a file server with several thousand user accounts failed catastrophically. The file–based restore took over a week to restore only 320 GB of data.

2.4 The fsck time crunch

The combination of a 10–fold increase in fsck time with, if anything, an increase in file system corruption requiring fsck creates the fsck time crunch. File systems will spend an ever increasing proportion of time unavailable simply due to being checked and repaired. Repair–driven file system design is one solution to the fsck time crunch.

3 Repair–driven file system design

How can we take file system check, repair, and restore into account when designing file systems? We discuss several general classes of useful techniques: (1) Divide and isolate metadata, (2) make file system traversal fast, (3) use more (relatively cheap) disk space, (4) keep it simple, (5) make restore fast.

3.1 Divide and isolate metadata

One of the primary reasons that file system check and repair are so painful is that they are O(file system size). All on–disk file system layouts we are aware of fundamentally require examining every piece of metadata in the system to determine, for example, whether a particular data block is free or allocated. If, on the other hand, we can divide up the metadata and create isolation boundaries that limit the effects of metadata corruption inside the boundary on things outside the boundary, we will be able to make file system check O(size of isolation group)—a constant factor. This can be thought of as block groups on steroids. Any references that cross isolation group boundaries must have a simple, direct, constant time method to determine the state of the relevant out–of–group metadata, such as back pointers. One way of doing this is to allow only inodes inside an isolation group to point to data blocks inside of it.

A limitation is that we need to be able to find the exact location of file system corruption. Useful methods include checksums, disk error reporting, dirty bits or marks on isolation groups showing that an update is

in progress, and error–checking code such as asserts and sanity checks in the file system code (such as in [11] and [2]). In the case of completely silent corruption or corruption spread over the entire file system, this technique will not be of much help. Isolation group boundaries need to be aligned with underlying device fault boundaries, such as RAID stripes or disks.

A corollary to divide and isolate is structuring file system metadata so that changes and repairs do not propagate excessively to other parts of metadata. Many complex tree structures require rebalancing and removal or addition of one node may affect many other nodes. The more work we have to do during repair, the longer it takes and the more fragile the file system.

Another benefit of divide and isolate is that the checker requires less memory, since it can check most of the file system in pieces, sequentially. If enough memory, CPUs, and disks are available, checking can easily be multi–threaded.

3.2 Make file system traversal fast

Even if we only need to read part of the metadata, it is still useful to make reading that metadata fast. Probably the most useful idea is to add another bitmap: The block metadata/data bitmap. This is a bitmap that marks whether a particular block contains metadata or data, regardless of the type of metadata. This allows us to read all the metadata in one sweep of the disk arm without waiting for dependent reads. In the best case, this could convert our 2013 fsck time to $42s \times (16/5) \approx 134s$ — only a third as long as current fsck.

The metadata bitmap also improves reliability because it gives an extra piece of information about whether a block is data or metadata; currently fsck has to guess in some cases. This allows us to rely more heavily on magic numbers in metadata during repair, as magic numbers (used as a sanity check to identify different metadata types) are more useful if we have some kind of out–of–band indication of whether a block is data or metadata. Consider the reiserfs bug in which any data block containing what looked like a consistent inode, complete with magic number, was considered a lost inode and put in lost+found. The scheme failed spectacularly when a disk image of a reiserfs file system was stored in a file in a reiserfs file system, since the inodes stored in the contents of the file looked exactly like lost inodes.

Greater fan–out and back pointers (at least for common, single link cases) may speed up file system traversal. Another idea is to have hints about which blocks to read next—they may not need to be exactly accurate, as long as on average they increase the speed at which metadata is read into memory.

3.3 Space is cheap

Modern disks have about 50% free space on average[5], and capacity will probably continue to outstrip actual data stored. We should use more of that space to make file system repair fast and reliable. In general, optimizations designed to save on–disk space are missing the forest for the trees; the real point of reducing the size of on–disk data is to save bandwidth, if anything. Checksums and magic numbers are the most obvious examples of useful additional on–disk structures. Multiple copies of metadata are another. Red zones to detect out–of–bounds writes and buffers between isolation groups may also be useful.

3.4 Keep it simple

The more knobs, gew–gaws, and ornamental curlicues in the on–disk format, the more difficult it is to repair errors correctly and the more failure modes multiply. The concept of having completely pluggable on–disk file formats, as in Reiser4[1], is a file system repair nightmare. On–disk structures that have long tentacles of cause–and–effect reaching into and changing other on–disk structures also increase fragility.

The problem with very simple, straight–forward disk formats is that they often do things like lookups in large directories very slowly. One way to have your cake and eat it too is to keep the on–disk format extremely simple, but build complex lookup structures in memory as data is read in off the disk. The first operation may be slow, but subsequent operations will go as fast as if the lookup structure was stored on disk. Another option is to cache complex structures on disk, but have redundant simpler structures which we can fall back on if the complex structure is damaged or non–existent.

3.5 Make restore fast

When all else fails—fsck has a fatal bug, the disk is smoking in the chassis, your RAID system ate itself—we need to restore from backup. An interesting thing about restoring from backup is that the application knows exactly how many files it will create, what size they will be, and what the directory structure will look like. If we have an interface to communicate this information in advance of the actual creat(), write(), mkdir() calls, the file system might be able to complete the operations faster. It will also benefit applications like tar that also have detailed information about what file system operations it will be doing.

4 Chunkfs

We are working on a repair–driven file system design called chunkfs[7]. The file system is divided into chunks which can be checked and repaired mostly independent of other chunks. The metadata inside each chunk looks like fairly standard file system metadata layout—inodes, indirect blocks, etc.. However, any references that cross chunk boundaries must have both forward and back pointers, so that external chunk references can be quickly checked. The only references that cross

chunk boundaries are when files or directories outgrow a chunk; when this happens, we create a "continuation inode" in another chunk and set up forward and back pointers between the original inode and its continuation. Only chunks that are damaged or dirty need to be checked, in addition to a cross–chunk reference checking pass. Chunks can also be checked as needed and on demand, so that the entire file system need not be down while only parts of it are being checked.

5 Related work

Some very basic optimizations for fsck, long implemented in most checkers, are found in [3]. Many file systems use journaling, copy–on–write, or soft updates to recover from partially finished file system updates, as often occurs with system panic or power loss. These techniques work only for completing in–progress updates; they have no effect on fsck time when repairing file system corruption from other sources. Peacock, et al. modified Solaris UFS to speed up fsck through a variety of techniques[10]. Most techniques were intended to solve the fast crash recovery problem; some, like recording the last used inode, are useful for all sources of file system corruption. Their work was constrained by the need for partial backwards compatibility with existing tools.

6 Conclusion

Traditional file system check and repair will become both more frequent and more time–consuming due to disk hardware trends—the fsck time crunch. To combat this, we need repair–driven file system design to become the norm. Examples of repair–driven design are making file system traversal fast using techniques such as metadata bitmaps, dividing and isolating metadata so it can be checked incrementally and so that corruption propagates as little as possible, keeping the on–disk format as simple as possible, and taking advantage of ample disk space to add checksums and redundancy to ease repair.

7 Acknowledgments

Thanks to Arjan van de Ven, Zach Brown, Andreas Dilger, Kristal Pollack, Theodore Y. T'so, Brian Warner, and Ric Wheeler for discussions, ideas, insightful comments, and suggestions.

Note to reviewers: Repair-driven file system design has been briefly discussed in two previous publications: "Chunkfs: Using divide–and–conquer to improve file system reliability and repair," appeared in HotDep '06, and in a review of the Linux File Systems Workshop '06 in Linux Weekly News.

References

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

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

[3] Eric J. Bina and Perry A. Emrath. A faster fsck for BSD UNIX. In *USENIX Winter Technical Conference*, pages 173–185, 1989.

[4] Kurt Chan. A comparison of disk drives for enterprise computing. *;login: The Usenix Magazine*, 31(3), 2006.

[5] John R. Douceur and William J. Bolosky. A large-scale study of file-system contents. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 1999.

[6] Jim Gray and Catharine van Ingen. Empirical measurements of disk failure rates and error rates. Technical Report MSR-TR-2005-166, Microsoft Research, 2005.

[7] Val Henson, Arjan van de Ven, Amit Gud, and Zach Brown. Chunkfs: Using divide-and-conquer to improve file system reliability and repair. In *Hot Topics in System Dependability*, 2006.

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

[9] Mark Kryder. Future storage technologies: A look beyond the horizon. `http://www.snwusa.com/documents/presentations-s06/MarkKryder.pdf`.

[10] J. Kent Peacock, Ashvin Kamaraju, and Sanjay Agrawal. Fast consistency checking for the Solaris file system. In *USENIX Annual Technical Conference*, 1998.

[11] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON file systems. In *SOSP '05*, pages 206–220, 2005.

[12] Thomas Schwarz, Mary Baker, Steven Bassi, Bruce Baumgart, Wayne Flagg, Catherine van Ingen, Kobus Joste, Mark Manasse, and Mehul Shah. `http://www.hpl.hp.com/personal/Mary_Baker/publications/wip.pdf`, 2006.

[13] 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.