How to (Not) Lose Your Data

Linux as a Reliable Storage Platform

Ric Wheeler
Red Hat
rwheeler@redhat.com

Abstract

Increasingly, Linux is the platform that major vendors use to implement everything from consumer grade NAS devices that you can buy at your local electronics store up to expensive, enterprise grade storage systems. This paper aims to present a high level overview of how some of these systems are put together, how to tune Linux for storage applications and what functionality is either on the horizon or yet to be started in the open source space that will enhance Linux as a storage system. The techniques presented are are also applicable to normal home users who would like to enhance data integrity.

1 Why Care about Data Integrity

Taking care of digital data used to be the worry of system administrators. If a computer went down without a backup, few people would ever notice any disruption. Today, the sheer mass of digital data that normal people have makes this a problem for just about anyone with a digital camera or a collection of digital music. Many commercial businesses use Linux-based systems for storing data about their customers like banking records, airline tickets and other critical data. Home users who turn to online sites for storing their photos, music and email on the web also, more than likely, end up using Linux-based systems indirectly.

Linux strives to maintain a unified version of its code, which means that there is just one storage and file system stack that is used by both casual end users and for servers at corporate data centers. The challenge is to provide a system that can leverage high-end storage and its features when possible, without imposing complexity and performance penalties for non-data critical applications. Given the deeply personal nature of the types of data that people store today on Linux, like digital photographs, it is really critical to give users a framework for how to store data reliably.

When designing a reliable storage system, enterprises usually invest in both reliable local storage and a way to replicate their data to storage at a remote site which must also be reliable. This paper aims to provide Linux developers a framework for thinking about how to provide reliable components for people building Linux based storage systems and weigh the trade offs appropriately. The conclusion presents a brief overview of key research in storage systems and a summary of upcoming features in the Linux storage and file system stack that will enhance both enterprise and end user's data integrity.

2 Common Causes of Data Loss in Systems

Anyone who deals with storage of digital data, especially long-term storage, understands that even the best storage systems can and will suffer data loss occasionally. This section gives a very high level overview of the most common causes of data loss.

2.1 User Errors

The most common errors are typically user errors; for example, accidentally deleting a file, forgetting where you put a specific file, upgrading your system or reformatting a whole disk. Rather than write off this class of data loss as "stupid human errors," the challenge is to design systems that are easy to use, have safe default settings and do not require expensive infrastructure like UPS backup for the servers' power needs.

At the system level, there are a few basic techniques that Linux provides which help mitigate against these types of user errors. A common practice in enterprise data centers is to create periodic snapshots of a file system, say once a day. Snapshots do not protect against storage failure at the block level, but they do give you a point in time picture of the state of your file system that can be referenced if you accidentally delete a file or do some other regrettable action to your data. In addition, users can create a snapshot of a live file system and use that snapshot as the basis for a local backup or to kick off a consistent remote copy.

2.2 Confusing Semantics for Key System Calls

Application writers need to have crisp and clear semantics for basic operations like fsync() and rename() system calls and clear documentation and guidance about how to use them to provide their users reliable data storage. How does an application know with absolute certainty that data that it writes or a new file that it has renamed will survive a power outage or system reboot? To make the challenge more interesting, different applications need different levels of granularity.

At one end of the spectrum, a database typically wants to have this type of clear promise after each commit of a transaction. At the other end of the extreme, it would usually be sufficient for an application like rsync to provide this promise that all of the data is safely stored on the remote system at the end of its execution which would allow the system to flush caches and so on only once for the entire set of files. In the rsync case, the user would be able to simply redo the rsync if the something fails during the initial run without being exposed to any data loss.

Somewhere in the middle of this spectrum are common tools like editors which want to provide atomic updates to files being processed. The rename() system call has long been used to provide this level of atomicity for updating files. For example, an editor that wants to overwrite all or part of file "foo" can do this safely by first creating a temporary copy of the contents of "foo" to a separate file, say "foo.temp." All changes are written to the temporary copy until the editor is ready to persist its changes to disk. At this point, it calls `rename("foo.temp", "foo")` with the expectation that, even in the face of a system crash or power outage, the rename will be atomic. Specifically, after a crash, the user will see either the new contents in file "foo" or the old contents, not some random mix of old and new data or an empty file.

To make this sequence really robust in a generic way, the application should issue one fsync() system call for the new "foo.temp" file and potentially a second fsync() system call on the directory that "foo" lives in to insure that the changes in the name space survive. File systems could automatically insert the appropriate fsync() calls internally, but this could degrade performance for applications that are less concerned about data integrity. How to get the balance right between data integrity and performance is an active debate in the file system developer community.

Clearly, using "fsync()" and "rename()" on every individual file while doing batch updates like the rsync example mentioned above, or when using tar to extract a large number of files, will have a large impact on performance. For some popular file systems like ext3, an effective way to avoid the performance impact of the fsync() per file technique is to have applications break up the extraction into a writing phase in which each file is written to disk without any special promises, and then a second fsync() phase in which the application iterates back over all of the files written and fsync()'s each one in turn in the reverse order that the files were written originally. This technique mitigates the heavy fsync() performance impact since the first fsync() in that second phase will push out the data for all of the preceding files that have just been written.

Different types of storage will show vastly different results since the performance is directly tied to how expensive seek operations are and whether or not the device has a volatile write cache. Testing the various methods on a common 1TB S-ATA disk can give the reader a sense of the impact using common hardware today. Using Fedora10 on a quad core desktop system and ext4, the best rate for writing 40KB files without doing any fsync() calls is around 2,600 files/second. Note that this test basically measures how quickly the file system can write to the page cache and is highly variable.

With the barrier support properly enabled on ext4, the slowest, most cautious method writes only 25 files/second by doing an open, write and and fsync on each file in turn. This rate is roughly half the rate that the drive's seek latency dictates, which corresponds well to the two cache flush operations per fsync that the fsync calls produce when running with barriers.

Finally, using the two phase technique, first writing all of the files in a batch and then iterating back over the batch of files in the reverse order to fsync them one at a time, the rate returns back up to around 143 files/second. If this is not convoluted enough, issuing a sync() system call before doing the fsync() phase will bring the rate up to around 900 files/second. Looking at the twists and turns required to get reasonable performance and data integrity clearly shows that we need to provide something better if we would like to get application programmers to improve their code.

Clearly, there is a lot of room for reducing the complexity, and giving application writers more intuitive and powerful tools. Over the past few years, file systems developers have been debating several possibilities ranging from some complex mechanisms like exposing transactional semantics to user space applications or providing a robust asynchronous fsync primitive. Like other async calls, the application would use the async fsync interface on each file in a fairly straight forward way and then have a second interface to use when it needs to wait for completion. The advantages of this async approach would be that the file system could optimize the fsync calls internally over a larger set of files.

2.3 IO Stack Bugs and Configuration Errors

System software, like the file system or the IO stack, can also be a common cause of data loss when it fails to persist data correctly before a power outage or system crash. Modern file systems and data bases often use journaled transactions as a way to provide robust storage. Transaction based systems need to be able to have a few promises from storage in order to make their transactions robust including the ability to store some information, like a transaction commit block, in a reliable way. Storage devices, including disk drives, have large, volatile write caches which is typically tens of megabytes in size. On power loss, the data stored in that cache will be lost.

To provide robust support for transactions, Linux has supported a fairly brute force mechanism called "write barriers" which effectively give the file system the ability to flush the target device write cache before sending a write with the commit block. A second flush is then initiated in order to make sure that the commit block itself is safe on persistent storage. This technique has a clear performance impact for applications that cause lots of transactions. Most file systems have mount options which enable the barriers correctly, but work is ongoing to make sure that all of the various bits of the block layer like device mapper and the more advanced RAID levels supported by MD will correctly handle barriers operations.

If the system has one of the configurations that do not support barriers properly and it has storage devices with volatile write cache devices, the only safe option is to disable the write cache on the storage devices which can be done with the hdparm command.

Note that external storage arrays typically have large, non-volatile write caches which do not require these barrier operations.Some of these arrays will silently ignore the cache flush commands issued by the Linux barrier operations, but others will honor them by flushing their potentially very large caches which is a gigantic performance hit. To prevent this overhead, file systems mounted on this class of device should be mounted with the barriers disabled.

The preceding set of considerations makes doing the right thing extremely confusing. If a system is using device mapper, the barrier operations will log an error and be disabled which leaves users exposed to potential data loss on power loss. The same story happens with RAID5 or RAID6 and MD devices. Several things need to be fixed in order to reduce the confusion. One very promising set of patches, recently posted by Martin Petersen, exports several characteristics of devices through /sys interfaces. Unfortunately, the nature of the write cache is not currently one of these characteristics, but this is a positive first step. Also, work is ongoing in the device mapper community to properly handle barrier operations. For MD users who use anything but the basic MD1 RAID, the only safe option is to disable the write cache on the individual component devices currently.

2.4 Hardware Failures

Hardware failures, specifically disk failures, are what most users would associate with data loss. Single disk drives are relatively reliable components, but can suffer from both hard failures when all data is lost or partial failures where only a portion of data is lost. Other types of hardware failure, like bad memory components, can cause data loss as well. RAID schemes, discussed briefly below, reduce the exposure to data loss by storing the data on multiple components which are assumed to have independent failures. New types of devices, like the increasingly popular SSDs, are largely immune from some of the causes of failure of traditional drives, but bring their own unique ways of failing that system designers and users will learn more about as the devices increase in number and age.

3 Data Loss Timeline

One useful way to think about keeping data safe is as a timeline. Assuming the application has figured out how to properly navigate the confusing maze detailed previously and has correctly stored the data on a storage device, a clock starts counting down for each hardware component in your system. Time runs out when the component actually sustains an error or fails completely. Designing a reliable storage device requires understanding the expected failure rates of the components used to make a system and being able to balance the cost of those components against other considerations like cost, performance and power consumption.

The high level overview of this timeline is:

Data Creation The application performs a write of data: for example, the "cp" application is used to create a copy of a file but has not called fsync(). The data is not protected against a power outage or system failure at this point in time.

Persistently Stored The data is stored and acknowledged by the storage subsystem: the data is moved from the page cache out to the storage system and the transaction is acknowledged back to the server. At this point, all is right with the data and the storage system has all of the redundant copies it needs to overcome a partial failure.

Component Failure A component of the storage system fails partially or completely: failures could be partial failures like a single bad sector on a drive, total failure of a drive or possibly a software or user error that corrupts a file. This error alone might not cause data loss or data unavailability to a user if the data is protected in a RAID group, but it does expose the user to permanent data loss if not repaired before a second failure in the same data stripe. The key consideration in building a robust storage system is to minimize the amount of time spent in this state.

Failure Detection The failure is detected by the system: an application tries to read a file back or the RAID software detects a partial or total failure. In RAID arrays, these errors are often detected by the firmware which will continually scan the surface of the individuals drives, searching for partial errors. The critical trade off here is that over aggressive scanning, while reducing the window of time that the system is exposed to a potential data loss, has a negative impact on the performance of the system's normal workload, can prematurely age the components and can consume more power since the devices are kept from entering an idle state.

Data Repair Initiated Examples include a new drive is inserted into the RAID group, a file system repair is initiated or a file is restored from tape. Note that there is a potential lag between the detection of the partial error and being able to initiate a repair. In the worst case, if you are repairing a RAID group with one completely failed drive and no spare, this repair phase is blocked until a new drive is physically inserted into the array to replace the failed component.

Data repair completed The original file is back and usable by the user. Just like the fourth stage above, there is a trade off here between completing the repair in an aggressive way by consuming the full bandwidth of the device and impacting the foreground workload.

Note that a related class of problem is data unavailability which can be caused by something as common as a power outage or by a long running, offline repair like an invocation of fsck. For time critical data, this can be as critical as permanent data loss.

The next section gives some details about common components used to build storage and gives some measure of how they rate in the time line sketched out here.

4 Reliability Building Blocks

A general principle of design for reliable systems is to build systems that tolerate a given number of failures. If you have a system with one drive, your data stands to disappear whenever that single drive fails. If you have two disks in a RAID1 mirror, your system can tolerate the failure of one disk but would still suffer failure if your CPU or DRAM fail. For this reason, enterprise class arrays have redundancy for all critical components: no single failure of a power supply, CPU, DRAM or disk would cause data loss but might cause degraded performance.

In a similar way, a single location like your home office or a data center is a single failure component which could be destroyed by a fire or other disaster. In order to reduce data loss for these catastrophic events, businesses commonly use long distance replication to store data on a remote site.

This section presents a summary of common features in storage and reviews the status of these features in Linux today.

4.1 RAID Level Tradeoffs

RAID is the most common form of data protection used today. RAID is normally done at the level of a block device, for example, a file system will send a write down to the block level which will do the appropriate RAID computations transparently. A simple, robust and inefficient RAID level is RAID1—all data is written to each member of the RAID group. For example, a system with four storage devices will write to each of the four devices on every IO. This gives the storage system great fault tolerance since the system could have as many as three of the four drives fail without incurring data loss. The down side of this scheme is that it is horribly inefficient with only 25% of the total capacity of the storage components available for storing user data. This ratio will be referred to as **effective capacity**.

Other RAID levels, with the same number of drives, improve the effective capacity. For example, RAID5 will break each IO into fragments, three data fragments and a fragment which contains parity information. Any single drive can fail and the other disks can be used to regenerate the data from the failed component. In this 4 drive system, a RAID5 scheme provides the user an effective capacity of 75%. In a similar way, RAID6 computes two different parity computations and will be able to survive any dual failure of storage devices, but decreases the effective capacity to 50% in our four drive example above.

Commercial RAID arrays offer a wide range of configurations. Low end systems aimed at consumers start with as few as 2 drives configured into a RAID1 device. Higher end consumer devices move up to a 4 drive RAID5 configuration. Enterprise class RAID arrays provide shelves full of disks. A typical mid-range storage system would have 12 to 15 drives per storage shelf with high-end systems ranging up to a couple thousand drives per array. Clearly, these larger systems present more than one RAID set out to hosts.

One type of failure that can foil any RAID system is an undetected partial failure. The above examples used the common assumption that a storage device would either work correctly or fail completely. While complete failures are not uncommon, it is also relatively common to have storage get corruption that impacts only a few sectors of storage. For example, rotational storage might have localized loss of data due to contamination like dust or lubricant on the platter while SSD devices might have localized data loss due to overuse. Regardless of the cause, the problem is the same—these partial failures can lie undetected for a very long time. In the worst case, they are detected only when a second total failure happens to a different storage device in the same RAID group. As the system tries to rebuild the RAID group, it needs to read data from all of the other components and will invariably detect all latent errors. Each of these latent errors will cause the RAID rebuild to fail for one stripe. In this case, the basic assumption about having independent failures does not protect the user since we notice the latent errors concurrently with the total failure of the other device.

The way to reduce the likelihood of failure during critical times like a RAID rebuild is to do periodic scans of the individual storage devices. For example, once every two weeks, the system will do a full surface scan of each storage device in a RAID group. If you detect an error during the scan, you can attempt to repair the data immediately by recomputing the data from the other devices in the RAID group and attempting to overwrite the failed sector. In many cases, this write will work by either correcting the data in place or by remapping the failed sectors to a pool of extra sectors kept for failures. If the data cannot be recovered, it is time to replace the failed device. In current Linux MD RAID, we have the capability to do this period scan for example.

There are some techniques used by high-end storage systems to make their RAID systems more robust. A very common technique is to have a spare device that is not an active participant in any RAID group. When a drive fails fully, the spare can be used to immediately start rebuild the contents of the missing storage component which decreases the window of time that the RAID group needs to be exposed to a possible second failure. A second trick is to suck as much data as possible from the failed component if it is still partially readable, since it allows the RAID rebuild only the data for stripes that cannot be read.

If the system needs to tolerate more than two component failures, there is a generic set of techniques called erasure encodings that can tolerate *k* failed components out of the *n* devices in your system. An example for the mid-level arrays might be an encoding that would survive any 4 failures in a 15-device system.

4.2 Remote Replication

Remote replication is another important tool for data protection and provides a remote copy of data that would survive any catastrophic event like a fire or a flood that would destroy any local storage. This section details several varieties of remote replication.

Block level replication can be built using something as simple as a RAID1 device, where one of the components is a remote device like an iSCSI target. Each write will be sent synchronously to the remote site which, depending on distance, can introduce substantial performance hurdles. A more sophisticated scheme could use LVM snapshots to avoid this performance penalty: snapshot a volume and then do the replication to the far site of the snapshot copy while local file system IO is left unhindered. Block level replication is also a feature that is frequently implemented inside of storage arrays that can use either dedicated storage links to the remote sites or direct the replication over normal connections. Block level replication is fairly common in high end data centers but can be a bit challenging to use in an intuitive way.

A more pedestrian way to replicate data is by replication at the file system layer. For anyone who is familiar with rsync, the technique is fairly intuitive: iterate over the entire file system and send the files that have changed to a remote server which will store it on disk. From the point of view of the source file system, the operation should be a fairly straight forward sequence of calls to getdents() in order to build a list of files followed by the application reading and then transmitting the file over the network to the target system. Unfortunately, there are several complications that get in the way of doing this in a straight forward manner.

Some file systems, specifically ext3, can return the file names via getdents() calls in an arbitrary order which, in turn, causes a lot of seeking as the application reads a series of small files in non-sequential order with regards to the disk layout. To improve this performance, applications can sort the list of file names by the inode order. Using a similar test, putting 1 million 40KB files in one directory results give a rate of 55 files/second when read in getdents() order and a rate of 1,381 files/second when read in sorted by inode number. Given that applications can write new files at a rate of 1277 files/second, the sorted remote replication is the only way to keep pace with ingest. Other file systems, like XFS, do a good job of returning the file names in a reasonable order. For these file systems there is no benefit from this technique but the cost of using it is not high. All of this complexity just continues to make the life of application programmers miserable. Note that doing full file system level iteration at full speed for any file system depends on minimizing head movement for traditional disk drives, so any other file system activity can have a severe impact on the performance of the replication.

4.3 Data Migration

Data migration is a special form of remote replication in which the intention is to decommission the source storage system once the data is successfully replicated at the target system. Data migration might involve a local migration from a single disk drive which has started to fail to a new drive, or be done from one high end storage array to a second one over a long distance link. It is a fairly common operation both for consumers who routinely replace or upgrade their personal systems and for data centers where high end storage is often rotated out of service after a fixed period, say every three years.

Key points of this class of replication include making absolutely certain that the remote system has a full and persistently stored copy of the data since the source will be taken offline. In the earlier rsync example, it is critical to make sure that the data is not just stored in the remote page cache. One other key consideration is that the source system is typically not new and can be in fairly rough shape, so the iteration can encounter more IO errors than a normal system would encounter. To migrate from an unhealthy source, the IO stack needs to be tuned properly to handle IO errors in a quick and deterministic way and avoid excessive retries. Applications doing the migration need to be able to be equally robust in face of errors: log any failures and keep moving good data to the new system.

Cloud storage can be thought of as another variation on remote replication at the file system layer. The difference is that the target system is normally not a typical file system or block device that users can access directly. Rather, for each file a user stores in the cloud, that user gets back a object reference that can be used to retrieve the file when needed.

5 Research in Reliable Storage Systems

File and storage system research has become one of the more active areas of academic research. One of the highlights of the year for researchers engaged in this area is the USENIX Association's annual FAST conference, where open source, industrial, and academic researchers meet to present key results. The USENIX Association has all of its papers online and freely available, along with recordings and videos of recent presentations. This section presents a very brief summary of key results presented recently.

Some key areas for storage are the analysis of what really fails and how frequent those failures are. For many years, storage companies have collected that data for their own deployed products and guarded that information as an important part of their intellectual property which made it extremely difficult to build either research or real systems based on facts. Two groundbreaking works were presented in FAST in 2007. The first paper was presented by Bianca Schroeder and Garth Gibson from Carnegie Mellon's Parallel Data Laboratory and presented the first large-scale analysis of real-world disk failures [6]. The second work at FAST that year was from Eduardo Pinheiro and his coauthors from Google [2] who shared similar data collected from the huge number of systems in Google's data centers. These works were followed in proceeding years by significant contributions by NetApp [1] and others.

For those interested in coding open source RAID6, or more robust erasure encoded systems, James Plank from the University of Tennessee and his coauthors presented an overview of open source friendly RAID6 [3] and general erasure encoding [4] algorithms. He took careful note of which algorithms he believed to be free of the known patents.

File systems also have received a fair amount of attention from the academic world, with notable contributions to Linux file systems reliability coming from the University of Wisconsin's group, which did an analysis of failures in commodity file systems and produced the prototype code used in ext4's journal checksumming [5]. Several other universities have active Linux based research projects, including Erez Zadok's group and their work on stacking file systems and the scalable file system work being done at the University of California, Santa Cruz.

6 Conclusion

The scale of storage systems is increasingly dramatically, both for home consumers and certainly for high end data centers. Current capacity for a single S-ATA drive is 2TB.In the examples used previously in this paper, this single drive will hold over 50 million 40KB files. Migration from a 2TB drive at 25 files/second would take close to 600 hours as compared to just under 12 hours running at the sorted rate of 1,300 files/second. The author has recently been testing Linux file systems on a relatively "small" 80TB LUN exported from EMC's newest Symmetrix, which can hold over two thousand drives. Clearly, scale makes using and improving the techniques discussed above an important challenge.

7 References

References

[1] Weihang Jiang, Chongfeng Hu, and Yuanyuan Zhou. Are Disks the Dominant Contributor for Storage Failures? A Comprehensive Study of Storage Subsystem Failure Characteristics. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies*, February 2008.

[2] E. Pinheiro, W. D. Weber, and L. A. Barroso. Failure trends in a large disk drive population. In *FAST-2007: 5th Usenix Conference on File and Storage Technologies*, February 2007.

[3] J. S. Plank. The RAID-6 Liberation Codes. In *FAST-2008: 6th Usenix Conference on File and Storage Technologies*, February 2008.

[4] J. S. Plank, J. Luo, C. D. Schuman, L. Xu, and Z. Wilcox-O'Hearn. A Performance Evaluation and Examination of Open-Source Erasure Coding Libraries For Storage. In *FAST-2009: 7th Usenix Conference on File and Storage Technologies*, February 2009.

[5] Vijayan Prabhakaran, Lakshmi N. Bairavasundaram, Nitin Agrawal, Haryadi S. Gunawi, Andrea C. Arpaci-Dusseau, and Remzi H. Arpaci-Dusseau. IRON File Systems. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP '05)*, pages 206–220, Brighton, United Kingdom, October 2005.

[6] Bianca Schroeder and Garth Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean too you? In *FAST-2007: 5th Usenix Conference on File and Storage Technologies*, February 2007.

Proceedings of the Linux Symposium

July 13th–17th, 2009
Montreal, Quebec
Canada