Working with write barriers and journaling filesystems

IMPOSING ORDER

Your journaling filesystem is carefully tracking write operations – but what happens when the data gets to the disk? A write barrier request can help protect your data.

BY MARTIN STEIGERWALD

Journaling filesystems offer some important benefits for the user, but they also pose some subtle problems. One problem is that the filesystem must keep a record that reflects the state of write operations to the storage device, but the storage device may actually change the order of those write requests in an effort to optimize performance. If the system fails at a point where the journal is out of step with the true sequence of write operations, your data may not be as safe as you think.

Scott Maxwell, Fotolia

Filesystem developers and disk vendors are well aware of this problem, and a number of solutions and workarounds have emerged. One brute-force solution is simply to flush the write cache before and after each write request, which effectively eliminates the write cache without disabling it at the device level. A better and faster solution that has gained favor among developers is to ensure that write requests are written to disk in a predictable order using what is called a *write barrier* request. Although write barrier support is becoming much more common, the question of whether you can use write barriers – and whether your journaling filesystem may already be using write barriers – depends on your filesystem, kernel version, and storage device.

I experienced three filesystem crashes within a week on my IBM ThinkPad T23, which uses XFS and kernel 2.6.16 with the write buffer activated [1]. The problems stopped when I deactivated the write cache. The funny thing was that previous kernels had been stable with the write buffer. Finally, I installed a new kernel 2.6.17, and its write barrier functionality gave me the stability I needed.

The rapid development and uneasy integration of write barriers with kernel versions, filesystem drivers, and storage devices means that, if you ever troubleshoot a journaling filesystem, you'd better start with some basic knowledge of write barriers. This article explores the intricacies of write barrier support.

How a Journaling Filesystem Works

A journaling filesystem provides life insurance for your data by recording every single change.

A *data journaling* (or *full* journaling) filesystem guarantees the consistency of the file *contents* (see the box titled "Data Journaling"). This approach is

THE AUTHOR

Martin Steigerwald is a trainer, consultant, and systems engineer with team(ix) GmbH in Nürnberg, Germany. Part of his job includes second level customer support for Linux business desktops owned by team(ix) customers. He installed Linux on his Amiga 4000 many years ago, and he still uses Linux whenever he can.

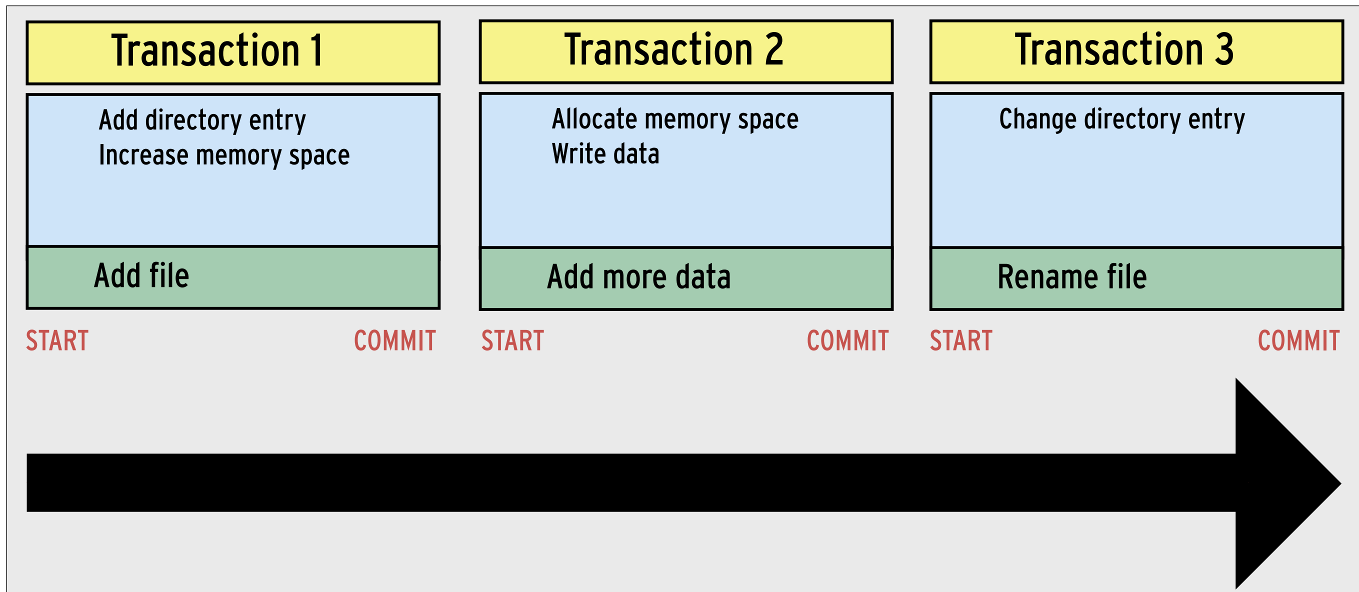

Figure 1: The filesystem writes individual transactions, such as creating a file, successively to the journal. Transactions can have one of two possible states: complete or not started. Completed transactions are tagged by a contiguous write operation.

very thorough, but typically at the cost of performance.

A faster technique known as *metadata journaling* guarantees the consistency of your filesystem *structure* by simply monitoring metadata information, such as file and directory names, file sizes, permissions, and storage locations. The filesystem stores this information in special blocks for administrative information known as inodes.

If the filesystem is interrupted while modifying the metadata, the metadata can become inconsistent because most changes comprise multiple steps, only half of which may have been completed. For example, when the filesystem creates a new file, it needs to create a directory entry, allocate storage space, write the data, and remember where it stored the file.

If an interruption happens, the storage space for the file may be occupied, although the filesystem may not have created a directory entry at the time of the interruption.

A filesystem without a journal only knows that it was not shut down correctly when you later reboot. A special program, such as *fsck*, must check whether the metadata is intact, and if you have a large filesystem with numerous directories and files, this can be a slow process.

In contrast to this, a journaling filesystem writes the changes for a complete operation, such as creating a file, to the journal as a transaction (see Figure 1). Transactions are atomic, that is, contiguous operations that can have one of two possible states: a transaction is either complete or it did not happen at all. Assuming the transaction is complete, the filesystem will tag it in an invisible write operation.

Journaling filesystems come with either of two distinct storage formats. A physical journal, like the one Ext3 uses, fills complete blocks with metadata. The Ext3 filesystem uses the Journal Block Device (JBD) [2] to do this. A filesystem with a logical journal like XFS, ReiserFS 3, or JFS, will store the metadata in its own, more compact format.

If you remount a journaling filesystem after an unexpected interruption to a write operation, it will try to evaluate the information in the journal to restore a consistent state. If a transaction, like a file creation, is still tagged as incomplete in the journal, the filesystem will discard the transaction.

The filesystem will process completed transactions step by step, checking which changes have been written out to disk, and writing changes that have not yet happened. The filesystem will not tag the transaction as complete until all changes are written to disk, at which point the journal space is freed.

As the journaling filesystem only needs to check the stored journal entries, there is no need to check the whole of the metadata structure, and this means that the recovery process will not take more than a few seconds under normal circumstances. If the filesystem is again interrupted at some point during the recovery process, the filesystem will just continue with the last incomplete transaction when everything return to normal.

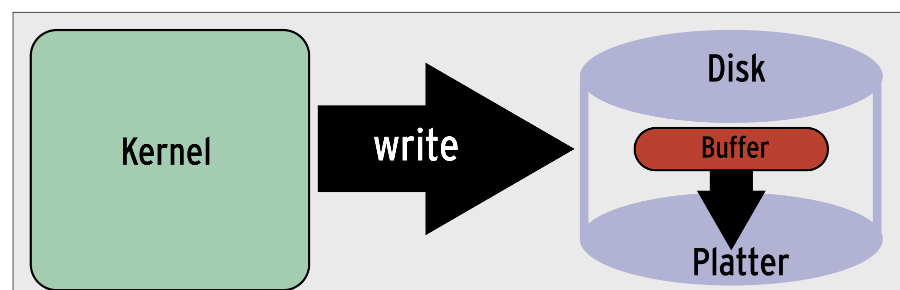

Figure 2: You have no control over the write order if you have a write buffer [3].

This approach does not guarantee that all your changes will survive a crash, however, it does ensure that the filesystem structure will be consistent, if all the write operations take place in the right order. The filesystem starts by writing the transaction to the journal. The filesystem then performs the metadata changes and finally tags the transaction as complete. If metadata changes reach the disk before the journal entry, and the process is then interrupted, the filesystem will be unable to locate entries for the changes in the journal during the recovery process. In this case, the filesystem is obviously in an inconsistent state.

Transactions checked off as complete before the metadata changes have been written to disk can cause similar issues. Thus, the filesystem has to ensure that changes are always written in a specific order. There is no guarantee that this will happen if you have a hard disk with a disk cache.

The hard disk will first cache any data it needs to write into a temporary memory buffer that resides between the fast RAM and the slow mechanical write mechanisms of the distribution. The disk firmware then decides in which order to write the write buffer contents (see Figure 2).

Figure 3: The hdparm command tells you whether the write cache for a disk is enabled.

One approach for making sure a specific write order is kept involves the filesystem telling the driver to flush the cache before and after writing a transaction. The second approach involves the filesystem using the kernel's write barrier functionality to arrange write operations in a specific order [4].

A barrier write request tells the Linux block layer to keep to the following write order for write operations: all write requests prior to the barrier request are processed normally; the barrier request then follows; and following the barrier request, all write requests are again processed normally.

This approach has two distinct advantages. There is no need for an immediate cache flush; the cache flush can occur directly prior to the barrier request. For another thing, the driver can leave the request order partly or entirely in the hands of intelligent storage devices.

The kernel block layer distinguishes between drives based on two criteria: the request order, and the write buffer type (see the Device Classes box). For example, Forced Unit Access type drives

Device Classes

The write barrier functionality in the block layer guarantees a specific order when processing I/O requests. Barrier requests thus need to have two properties [3]:

Request Order

The following variants are possible:

- Devices with support for multiple queued requests, and for sequenced requests (TCQ devices) – such as modern SCSI controllers and drives. The block layer passes the barrier request on as a sequenced request. Low level drivers, controllers, and drives are responsible for keeping to the correct sequence. TCQ is the abbreviation for Tagged Command Queueing, that is, the ability of a drive to queue multiple requests. This option is not currently enabled on Linux as the SCSI subsystem's dispatch function on kernels up to up 2.6.17 does not pass requests atomically to the SCSI controller, and this means that the request order may change.
- Devices that support multiple queued requests, but not sequenced requests – this is typical of older SCSI controllers and drives, as well as SATA drives: the block layer guarantees the correct order.
- Devices that handle requests sequentially – very old SCSI devices and IDE drives; again, the block layer guarantees the correct order.

Write Cache

The write cache configuration can fall in any of the following cases:

- No write cache: it is sufficient to organize requests in the correct order.
- Writeback cache without cache flushing: there can be no guarantee of the correct write order and no support for write barriers. You need to disable the write cache on drives of this kind to provide stable support for abnormal termination on journaling filesystems.
- Write cache and cache flushing, without Force Unit Accesses (FUA): the block subsystem triggers a cache flush before and after the barrier request.
- Write cache, cache flushing, and Forced Unit Access (FUA): the block layer triggers a cache flush before the barrier request. The barrier request passes the flush on as an FUA request. FUA stands for Forced Unit Access and tells the drive to write the request out to disk immediately and not use the write cache while doing so.

XFS Notes

If you want to achieve higher speeds for XFS metadata operations – especially for deleting large numbers of directories and files – you can mount the filesystem with the *logbufs=8* option to enable a larger number of buffers for the journal (default 2, max. 8) [7]

If you need to use kernel 2.6.17 with XFS, you should use version 2.6.17.7 or higher, as this version includes a fix for XFS [10], [11], [12]. You can also install the patch manually. Without the patch, minor filesystem defects, which the current version of *xfs_repair* may not be able to repair, can occur. You'll find a patch for *xfs_repair* that apparently solves this issue [13].

do not need a cache flush after the barrier request.

Enabling write barriers for a journaling filesystem can improve stability and performance – as long as your kernel version, your hard disk and your filesystem all offer write barrier support.

Practical Applications

The first thing you'll need if you want to use write barriers is a kernel that supports write barrier functionality. The various journaling filesystems unveiled write barrier support with different kernel versions.

For XFS, you'll need kernel version 2.6.16 or later (or preferably kernel version 2.6.17.7 or later). There is a kernel 2.6.5 patch for Ext3 and ReiserFS 3, but if you don't have the patch, write barrier support for Ext3 and ReiserFS 3 is officially available as of kernel version 2.6.9, however, kernel 2.6.16 still has a number of changes and bug fixes for write barriers. The XFS filesystem supports write barriers on version 2.6.15 or later (see the Write Barrier History box).

Even if you have a recent kernel, it is a good idea to check whether you can actually use write barriers with your combination of kernel version, drivers, filesystem, controller, and drive.

A few things might help you find your way here: IDE drives without write buffers, IDE drives with cache flush, SCSI drives without write buffers, SCSI drives with cache flush, or SCSI drives with cache flush and FUA (Forced Unit Access) should support write barriers (see the Device Classes box).

Write barriers should also work with software RAID and MD/RAID1, as long as the controller and all the drives support cache flushing. Other RAID variants are not supported as of this writing.

Write barrier functionality is enabled by default for XFS as of kernel 2.6.17. Reiser 4 uses write barriers if supported, and synchronous write operations, that is, direct cache flushes if not. JFS uses only synchronous write operations. For the other filesystems, and XFS prior to 2.6.17, you may need to specify a mount option (see Table 1).

Write barrier support is evolving very rapidly. If you are uncertain whether your system supports write barriers, use one of the mount options shown in Table 1 until you can determine the default behavior.

A quick glance at the system protocol, when mounting a filesystem with write barrier functionality enabled, tells you whether things have gone well. For example, depending on the scenario, XFS will give you one of three error messages if things have not worked out [5]. The Journal Block Device, which Ext3 uses, tells you: *JBD: barrier-based sync failed on %s -- disabling barriers*. The other filesystems output similar messages.

The *hdparm -I /dev/hda* command shows you whether the write cache is active: *Write cache* below *Command/features* (see Figure 3).

The *hdparm -W0 /dev/hda* command disables the write cache, *hdparm -W1 /dev/hda* enables the cache, and *hdparm -I /dev/hda* tells you the vendor's default setting in *WriteCache*.

System Support

On some systems, you can test whether write barriers are supported by

Data Journaling

Journaling filesystems that only write metadata to the journal have one drawback. If write operations are interrupted, files can be trashed by incomplete write operations [14]. The filesystem might have allocated additional data blocks for the file before completing the write operations in these blocks. Or a write operation designed to overwrite data in a file may not have been completed.

A filesystem with data journaling resolves this by writing the data to the journal first. If a crash occurs, the filesystem uses the information in the journal to restore a state where the metadata matches the data in files, and individual write operations are either complete or have not taken place.

The Ext3, ReiserFS 3, and Reiser 4 filesystems all support data journaling, whereas XFS and JFS do not support it, as of this writing.

Write operations are far slower if you enable data journaling, and this is understandable, as all write operations actually take place twice. The data is first written to the journal and then to the final storage location on the disk. Reiser 4 is the only filesystem to write the data at the final location and to superimpose a wandering journal over the data [15].

The Ext3 and ReiserFS 3 filesystems offer an interim solution that does without data journaling: the filesystem writes the data blocks for a metadata transaction before entering the transaction in the journal. This solution ensures that newly allocated data blocks in a file will always have valid data. However, a partly finished operation that overwrites data in a file will always lead to an inconsistent state.

Data journaling does not guarantee the integrity of your application data if a write process terminates abnormally. This is why many database and server programs, such as mail servers, have their own mechanisms for guaranteeing data integrity in case of a crash or power outage. This kind of application-specific data journaling is typically based on writing data in a specific order, and it typically relies on atomic write operations. Other programs, such as the KDE PIM applications KAddressBook, Korganizer, and Akkregator, create backups of critical files.

Alternative Approaches

Journaling is not the only means for guaranteeing filesystem integrity. Other filesystems employ techniques such as:

- Soft updates – instead of duplicating metadata in the journal, the filesystem organizes write operations for metadata in a way that guarantees filesystem integrity at all times. This technology, which was originally devised for FreeBSD, is now available for other BSD variants [16].
- Persistent write cache and an uninterruptible power supply – the controller, or driver, stores the data to be written to disk in a non-volatile memory area. In case of a crash, a backup power supply gives the system enough time to write the data out to disk. Various RAID controllers and storage appliances, like the Fabric Attached Storage (FAS) appliance from Netapp, which is also called Filer, use this technology [17].
- Log-structured filesystems – The whole filesystem is a journal – this removes the need for duplicate writing of data and metadata [18]. The UDF DVD filesystem is an example of this. Reiser 4 or WAFL (Write Anywhere Layout), as used by the NetApp FAS storage appliance, use a number of log-structured techniques [15], [19].

by mounting the filesystem via a loop device. The last time I looked, write barriers were not supported with loop devices. Thus, if you attempt to mount the filesystem via a loop device and you get an error in syslog or dmesg, it may be because your filesystem is attempting to use write barriers.

Without write barrier functionality, journaling filesystems simply keep their own integrity in case of an unexpected interruption, if the disk write cache is switched off, or if the filesystem writes transactions synchronously.

Drives with write buffers that do not support cache flushing are by design incapable of supporting a specific order for write requests. In that case, the only way for users to achieve data safety with journaling filesystems is to disable the write buffer.

Controllers and drives with persistent write buffers (NVRAM) do not typically need write barrier support, since these devices can write out data to disk even after a crash or power outage. In fact, write buffer support can sometimes interfere with an NVRAM device. The XFS FAQ even recommends disabling write barriers for devices with persistent write buffers [6].

Conclusion

Enabling write barriers solved the stability problems I was having with XFS on my notebook. The alternative of disabling the write caches also helped, but at least in theory, write barriers provide better performance – especially for complex write operations.

If you are using a journaling filesystem and you want to experiment with write barriers, first make sure you have a version of the kernel that offers write barrier support for your filesystem. If both your kernel and your storage device support write barriers, you may find that write barriers are enabled by default. If you're not sure, try the mount options shown in Table 1. ■

Table 1: Mount Options

Functionality	Ext3	ReiserFS3	Reiser 4	XFS	JFS
Write Barrier	barrier=1	barrier=flush	Standard	barrier (standard as of 2.6.17)	–
No write Barrier	barrier=0	barrier=none	–	nobarrier	–
Data journaling	ordered=journal	ordered=journal	Standard (wandering logs)	–	–
Data before metadata	ordered=data	ordered=data	–	–	–
Writeback mode	ordered=writeback	ordered=writeback	–	Standard behavior	Standard behavior

Table 2: Write Barrier History

Kernel Version	Date	Change
2.6.5	March 2004	First write barrier patchset. Supported filesystems: Ext3 and ReiserFS 3 [8].
2.6.9	October 2004	Write barrier support for IDE, SCSI, MD, device mapper, Ext3, ReiserFS is officially added to kernel.
2.6.10	December 2004	Fix for mount errors with barrier on SATA disks; fix for multiple CPU support on various platforms (SGI Challenge, Origin, and Altix).
2.6.12	June 2005	Write barrier support for DASD controllers (S/390).
2.6.13	August 2005	Fix for IO scheduler CFQ (Completely Fair Queueing) with barriers (regression from 2.6.12); Device Mapper Multipath does not support write barriers.
2.6.14	October 2005	MD/RAID still does not support write barriers at this point.
2.6.15	Januar 2006	Write barrier support for XFS with MD/RAID1.
2.6.15.4	Februar 2006	Sequenced write operations with cache flush for SCSI disabled.
2.6.16	March 2006	New implementation of barrier request handling IDE / SCSI: will now complete barrier requests atomically or not at all; FUA support (Forced Unit Access) for SCSI drives; write barrier for XFS enabled by default, then disabled again due to I/O problems; update of barrier documentation[3]; Device Mapper targets Snapshot and Origin do not support write barriers.
2.6.17	June 2006	Write Barrier for XFS enabled by default for MD/RAID1; improved detection when barrier support is missing Device Mapper; fix for hang on barrier requests with MD/RAID1.

INFO

[1] Kernel Bug #6380: *http://bugzilla.kernel.org/show_bug.cgi?id=6380*

[2] The Journaling Block Device: *http://kerneltrap.org/node/6741*

[3] Diskio visualization: *http://developer.osdl.jp/projects/doubt/diskio/documents/2005-10-19-visualizing_diskio.pdf*

[4] Tejun Heo, I/O Barriers, Documentation for Linux Kernel 2.6.17.1: block/barriers.txt

[5] XFS FAQ, How can I address the problem with the write cache?: *http://oss.sgi.com/projects/xfs/faq.html#wcache_fix*

[6] XFS FAQ, Should barriers be enabled with storage which has a persistent write cache?: *http://oss.sgi.com/projects/xfs/faq.html#wcache_persistent*

[7] Filesystem performance tweaking with XFS on Linux: *http://everything2.com/index.pl?node_id=1479435*

[8] Barrier Patch Set: *http://lwn.net/Articles/76540/*

[9] Kernel Changelogs: *http://www.kernel.org/pub/linux/kernel/v2.6/*

[10] XFS Corruption Fix: *http://marc.theaimsgroup.com/?t=115315520200004&r=1&w=2*

[11] Patch by Mandy Kirkconnell: *http://bugzilla.kernel.org/show_bug.cgi?id=6757*

[12] XFS FAQ, What is the issue with directory corruption in Linux 2.6.17?: *http://oss.sgi.com/projects/xfs/faq.html#dir2*

[13] XFS Repair Fixes: *http://oss.sgi.com/archives/xfs/2006-07/msg00374.html*

[14] Advanced filesystem Implementor's Guide: *http://www.gentoo.org/doc/en/articles/l-afig-p8.xml*

[15] Wandering journal in Reiser 4: *http://en.wikipedia.org/wiki/Reiser4*

[16] Soft updates: *http://en.wikipedia.org/wiki/Soft_updates*

[17] Netapp: *http://www-uk.netapp.com/index.html*

[18] Log-structured file system: *http://en.wikipedia.org/wiki/Log-structured_filesystem*

[19] Write Anywhere File Layout: *http://en.wikipedia.org/wiki/Write_Anywhere_File_Layout*