Disks from the Perspective of a File System

Disks lie. And the controllers that run them are partners in crime.

Marshall Kirk McKusick

Most applications do not deal with disks directly, instead storing their data in files in a file system, which protects us from those scoundrel disks. After all, a key task of the file system is to ensure that the file system can always be recovered to a consistent state after an unplanned system crash (for example, a power failure). While a good file system will be able to beat the disks into submission, the required effort can be great and the reduced performance annoying. This article examines the shortcuts that disks take and the hoops that file systems must jump through to get the desired reliability.

While the file system must recover to a consistent state, that state usually reflects the one that the file system was in some time before the crash. Often data written in the minute before the crash may be lost. The reason for this loss is that the file system has not yet had the opportunity to write that data to disk. When an application needs to ensure that data can be recovered after a crash, it does an `fsync` system call on the file(s) that contain the data in need of long-term stability. Before returning from the `fsync` system call, the file system must ensure that all the data associated with the file can be recovered after a crash, even if the crash happens immediately after the return of the `fsync` system call.

The file system implements `fsync` by finding all the dirty (unwritten) file data and writing it to the disk. Historically, the file system would issue a write request to the disk for the dirty file data and then wait for the write-completion notification to arrive. This technique worked reliably until the advent of track caches in disk controllers. Track-caching controllers have a large buffer in the controller that accumulates the data being written to the disk. To avoid losing nearly an entire revolution to pick up the start of the next block when writing sequential disk blocks, the controller issues a write-completion notification when the data is in the track cache rather than when it is on the disk. The early write-completion notification is done in the hope that the system will issue a write request for the next block on the disk in time for the controller to be able to write it immediately following the end of the previous block.

This approach has one seriously negative side effect. When the write-completion notification is delivered, the file system expects the data to be on stable store. If the data is only in the track cache but not yet on the disk, the file system can fail to deliver the integrity promised to user applications using the `fsync` system call. In particular, semantics will be violated if the power fails after the write-completion notification but before the data is written to disk. Some vendors eliminate this problem by using nonvolatile memory for the track cache and providing microcode restart after power failure to determine which operations need to be completed. Because this option is expensive, few controllers provide this functionality.

Newer disks resolve this problem with a technique called *tag queueing*, in which each request passed to the disk driver is assigned a unique numeric tag. Most disk controllers that support tag queueing will accept at least 16 pending I/O requests. After each request is finished—possibly in a different order than the one in which they were presented to the disk—the tag of the completed request is returned as part of the write-completion notification. If several contiguous blocks are presented to the disk controller, it can begin work on the next block while notification for the tag of the previous one is being returned. Thus, tag queueing allows applications to be accurately notified when their data has reached stable store without incurring the penalty of lost disk revolutions when writing contiguous blocks. The `fsync` of a file is implemented by sending all the modified blocks of the file to the disk and then waiting until the tags of all those blocks have been acknowledged as written.

Tag queueing was first implemented in SCSI disks, enabling them to have both reliability and speed. ATA disks, which lacked tag queueing, could be run either with their write cache enabled (the default) to provide speed at the cost of reliability after a crash or with the write cache disabled, which provided the reliability after a crash but at a 50-percent reduction in write speed.

To escape this conundrum, the ATA specification added an attempt at tag queueing with the same name as that used by the SCSI specification: TCQ (Tag Command Queueing). Unfortunately, in a deviation from the SCSI specification, TCQ for ATA allowed the completion of a tagged request to depend on whether the write cache was enabled (issue write-completion notification when the cache is hit) or disabled (issue write-completion notification when media is hit). Thus, it added complexity with no benefit.

Luckily, SATA (serial ATA) has a new definition called NCQ (Native Command Queueing) that has a bit in the write command that tells the drive if it should report completion when media has been written or when cache has been hit. If the driver correctly sets this bit, then the disk will display the correct behavior.

In the real world, many of the drives targeted to the desktop market do not implement the NCQ specification. To ensure reliability, the system must either disable the write cache on the disk or issue a cache-flush request after every metadata update, log update (for journaling file systems), or `fsync` system call. Both of these techniques lead to noticeable performance degradation, so they are often disabled, putting file systems at risk if the power fails. Systems for which both speed and reliability are important should not use ATA disks. Rather, they should use drives that implement Fibre Channel, SCSI, or SATA with support for NCQ.

Another recent trend in rotating media has been a change in the sector size on the disk. From the time of their first availability in the 1950s until about 2010, the sector size on disks has been 512 bytes. In 2010, disk manufacturers began producing disks with 4,096-byte sectors.

As the write density for disks has increased over the years, the error rate per bit has risen, requiring the use of ever-longer correction codes. The errors are not uniformly distributed across the disk. Rather, a small defect will cause the loss of a string of bits. Most sectors will have few errors, but a small defect can cause a single sector to experience many bits needing correction. Thus, the error code must have enough redundancy for each sector to handle a high correction rate even though most sectors will not require it. Using larger sectors makes it possible to amortize the cost of the extra error-correcting bits over longer runs of bits. Using sectors that are eight times larger also eliminates 88 percent of the sector start and stop headers, further reducing the number of nondata bits on the disk. The net effect of going from 512- to 4,096-byte sectors is a near doubling of the amount of user data that can be stored on a given disk technology.

When doing I/O to a disk, all transfer requests must be for a multiple of the sector size. Until 2010, the smallest read or write to a disk was 512 bytes. Now the smallest read or write to a disk is 4,096 bytes.

For compatibility with old applications, the disk controllers on the new disks with 4,096-byte sectors emulate the old 512-byte sector disks. When a 512-byte write is done, the controller reads the 4,096-byte sector containing the area to be written into a buffer, overwrites the 512 bytes within the sector that is to be replaced, and then writes the updated 4,096-byte buffer back to the disk. When run in this mode, the disk becomes at least 50 percent slower because of the read and write required. Often it becomes much slower because the controller has to wait nearly a full revolution of the disk platter before it can rewrite a sector that it has just read.

File systems need to be aware of the change to the underlying media and ensure that they adapt by always writing in multiples of the larger sector size. Historically, file systems were organized to store files smaller than 512 bytes in a single sector. With the change in disk technology, most file systems have avoided the slowdown of 512-byte writes by making 4,096 bytes the smallest allocation size. Thus, a file smaller than 512 bytes is now placed in a 4,096-byte block. The result of this change is that it takes up to eight times as much space to store a file system with predominantly small files. Since the average file size has been growing over the years, for a typical file system the switch to making 4,096 bytes the minimum allocation size has resulted in a 10- to 15-percent increase in required storage.

Some file systems have adapted to the change in sector size by placing several small files in a single 4,096-byte sector. To avoid the need to do a read-modify-write operation to update a small file, the file system collects a set of small files that have changed recently and writes them out together in a new 4,096-byte sector. When most of the small files within a sector have been rewritten elsewhere, the sector is reclaimed by taking the few remaining small files within it and including them with other newly written small files in a new sector. The now-empty sector can then be used for a future allocation.

The conclusion is that file systems need to be aware of the disk technology on which they are running to ensure that they can reliably deliver the semantics that they have promised. Users need to be aware of the constraints that different disk technology places on file systems and select a technology that will not result in poor performance for the type of file-system workload they will be using. Perhaps going forward they should just eschew those lying disks and switch to using flash-memory technology—unless, of course, the flash storage starts using the same cost-cutting tricks.

LOVE IT, HATE IT? LET US KNOW

feedback@queue.acm.org

DR. MARSHALL KIRK MCKUSICK writes books and articles, teaches classes on Unix- and BSD-related subjects, and provides expert-witness testimony on software-patent, trade-secret, and copyright issues, particularly those related to operating systems and file systems. He has been a developer and committer to the FreeBSD Project since its founding in 1994. While at the University of California at Berkeley, he implemented the 4.2BSD fast file system and was the research computer scientist at the Berkeley CSRG (Computer Systems Research Group) overseeing the development and release of 4.3BSD and 4.4BSD.

© 2012 ACM 1542-7730/12/0900 $10.00