MARSHALL KIRK MCKUSICK

disks from the perspective of a file system

Dr. Marshall Kirk McKusick writes books and articles, consults, and teaches classes on UNIX- and BSD-related subjects. He has twice served on the Board and as president of USENIX.

kirk@usenix.org

MOST APPLICATIONS DO NOT DEAL with disks directly. Rather, they store their data in files in a file system. One of the key tasks of the file system is to ensure that the file system can always be recovered to a consistent state after an unplanned system crash (e.g., due to a power failure).

Although the file system must recover to a consistent state, that state usually reflects the state of the file system sometime before the crash (often data written in the minute before the crash may be lost). When an application needs to ensure that data can be reliably recovered after a crash, it does an fsync system call on the file or files 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 the fsync system call by finding all the dirty (unwritten) file data and writing these data 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 the 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 are in the track cache rather than when they are 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 kernel expects the data to be on stable store. If the data are 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 are written to disk. Some vendors eliminate this problem by using nonvolatile memory for the track cache and providing microcode restart after

a 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 queuing*. With tag queuing, each request passed to the disk driver is assigned a unique numeric tag. Most disk controllers supporting tag queuing will accept at least 16 pending I/O requests. After each request is finished, 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 one while notification for the tag of the previous one is being returned. Thus, tag queuing allows applications to be accurately notified when their data have reached stable store without incurring the penalty of lost disk revolutions when writing contiguous blocks.

Tag queuing was first implemented in SCSI disks, enabling them to have both reliability and speed. ATA disks, which lacked tag queuing, could either be run 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 reliability after a crash but at about a 50% reduction in write speed.

To try to solve this conundrum, the ATA specification added an attempt at tag queuing with the same name as that used by the SCSI specification, Tag Command Queueing (TCQ). 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, with SATA there is a new definition called Native Command Queueing (NCQ) that has a bit in the write command that tells the drive if it shall report completion when media has been written or when cache has been hit. Provided that the driver correctly sets this bit, the disk will have 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. Because both of these techniques lead to noticeable performance degradation, they are often disabled, putting file systems at risk in the event of power failures. 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.