Rethink the Sync!

Edmund B. Nightingale, Kaushik Veeraraghavan, Peter M. Chen, and Jason Flinn
University of Michigan

File systems serve two opposing masters: durability and performance. A file operation guarantees durability if data is written to disk before the operation completes. However, since disk writes are time-consuming, synchronous operations perform poorly. For example, use of synchronous I/O degrades performance by two orders of magnitude for disk-intensive benchmarks.

File systems often sacrifice durability to provide reasonable performance. Most current file systems provide an asynchronous I/O abstraction by default: file modifications are typically committed to disk long after a file operation returns. This is fast, but not safe. In the absence of explicit synchronization operations such as `fsync`, users often view output that depends on uncommitted modifications. If a system loses data due to crash or power failure, the viewed output is incorrect because it depends on data that has been lost.

Making all file system calls synchronous provides a cleaner abstraction. Any output seen by a user or an application running on another computer is durable; i.e., it will not be lost due to a subsequent OS crash or power failure. Synchronous I/O also guarantees ordering; i.e., if an operation A causally follows another operation B, the effects of B are never visible unless the effects of A are also visible. Finally, synchronous I/O simplifies applications since programmers do not need to manually provide ordering and durability for their data using system calls such as `fsync`. However, despite these clear benefits, most file systems eschew synchronous I/O because it assumed to be too slow. We believe this assumption is wrong.

One can view an abstraction such as synchronous I/O as a set of guarantees provided to external clients. Asynchronous I/O improves performance by substantially weakening the guarantees. Our approach is fundamentally different: we provide the same guarantees but change the client to which the guarantees are provided. Operating systems currently take an application-centric view; they guarantee durability and ordering for each system call made by an application. This is correct but too conservative. Instead, we propose a user-centric view, which we call *visible synchrony*, in which ordering and durability are guaranteed not to the application, but to any external entity that observes application output.

From the viewpoint of an external observer such as a user or an application running on another computer, the behavior of an externally synchronous system is identical to the behavior of a system that uses traditional synchronous I/O. Visible synchrony guarantees durability by buffering output until the prior file system modifications upon which that output depends have been committed to disk. Thus, an external observer never sees output that depends on uncommitted modifications. Visible synchrony guarantees ordering by writing modifications to disk in the order that they were generated by applications. This ensures that, after a crash, an external observer will not see a modification unless all other modifications that causally precede that modification are observable.

We are currently implementing visible synchrony in the Linux kernel using mechanisms developed as part of the Speculator project [1]. When an application performs a synchronous I/O operation, the operating system adds the modifications to a file system transaction (we use ext3 in data journaling mode for this purpose). The operating system returns control to the application without waiting for the transaction to commit. However, the operating system also taints the process that performed the I/O with a causal dependency that specifies that the application is not allowed to externalize any output until the transaction is committed. If the application writes to the network, screen, or other device, its output is buffered by the OS and released only when all disk transactions on which the output depends have been committed. If a process with such dependencies interacts with another process on the same computer through IPC mechanisms such as pipes, the file cache, or shared memory, the other process inherits the same dependencies so that it also cannot externalize output until the transaction commits. The performance of visible synchrony is generally quite good since applications can perform computation and initiate further I/O operations while waiting for a transaction to commit. In most cases, the output is delayed by at most the time to commit a single transaction—this is typically much less than the perception threshold of a human user.

Our results to date are promising. We have modified ext3 to provide visible synchrony for all file system operations. For I/O intensive benchmarks such as Postmark and an Andrew-style build, the performance of ext3 with visible synchrony is within 6% of the default asynchronous implementation. In contrast, conventional synchronous I/O is over two orders of magnitude slower. Even if synchronous I/O only commits modifications to the disk write cache (allowing data to be lost on power failure), it is still over 40% slower than visible synchrony.

References

[1] NIGHTINGALE, E. B., CHEN, P. M., AND FLINN, J. Speculative execution in a distributed file system. In *Proceedings of the 20th ACM Symposium on Operating Systems Principles* (Brighton, United Kingdom, October 2005), pp. 191–205.