Extending a verified file system with concurrency

Tej Chajed (student)
Adam Chlipala, M. Frans Kaashoek, Nickolai Zeldovich (advisors)

2 pages

Abstract

Concurrency is a major source of bugs in file systems, and it remains one of the challenging open problems in applying formal verification in that domain. This research contributes CFSCQ, the first concurrent file system with a formal proof that its implementation meets its specification. CFSCQ uses a novel approach, called *optimistic system calls*, to reuse the implementation, specification, and proof of a verified sequential file system.

1. Introduction

Many applications, including databases, rely on file systems to store their data persistently. However, file systems have a long history of bugs that lead to incorrect result, data corruption, or data loss. An extensive study of bugs in several file systems from the Linux 2.6 kernel [20] found that most bugs (over 50%) are semantic bugs that require an understanding of file-system semantics to find or fix, and the next most common source of bugs is concurrency issues (about 20%), such as inadvertently updating shared state in a supposedly read-only operation that holds no locks. File systems are particularly prone to concurrency bugs because all threads must access the same complex shared state: both the persistent state as well as various in-memory caches.

Formal verification is a promising approach to address file-system bugs, because it allows developers to specify the precise semantics that a file system should provide, and because it ensures that these semantics are followed in all possible corner cases. Prior work has shown that this can work for a sequential file system [1, 5, 25], thus addressing the first category of bugs mentioned above. However, since prior work is limited to single-threaded file systems, it cannot address the next most prominent source of file system bugs, namely concurrency issues.

This research aims to produce the first formally verified concurrent file system, CFSCQ (Concurrent FSCQ). CFSCQ supports concurrent I/O and concurrent readers, but does not support concurrent writers. Concurrent I/O allows the disk[1] to service several pending reads while the CPU is executing another system call. CFSCQ also allows multiple processes to execute read-only system calls on multiple cores, in parallel with one read-write system call, which is a good fit for read-mostly workloads. However, CFSCQ is not as sophisticated as modern file systems such as Linux ext4, which allow concurrent writers to different parts of the file system.

Even CFSCQ's concurrency is difficult to get right. With I/O concurrency, one process may start reading a directory block from disk, but while that process is waiting, another process may either write to that directory block or delete the directory altogether. With concurrent readers, it is important to ensure that updates of caches are done safely, such as a name lookup updating a directory cache; in other words, system calls that are semantically read-only (such as `stat`ing a file) can actually write shared state (such as updating the directory cache as a result of looking up the file).

To verify CFSCQ, our work uses ideas from optimistic transactions [18] [23, §9.4.3] in an approach we call *optimistic system calls* (OSC). In this approach, system calls are executed as atomic actions in some sequential order. Every system call optimistically runs without locks, but if it makes any changes (such as scheduling disk I/O or modifying memory), it is aborted and retried while holding a global file-system lock. Readers obtain a snapshot of the system state when they start executing, and appear to execute instantaneously at that point. The key benefit of OSC is that it allows reusing existing implementations, specifications, and proofs from a sequential verified file system.

2. Approach

CFSCQ supports two forms of concurrency: disk reads concurrent with system calls executing on the CPU and concurrent read-only operations that use only in-memory caches. We aim to provide a linearizable specification, where each system call executes atomically, and the system calls appear to execute in some linear order [16]. This specification fits naturally with our approach: the execution behavior resembles this description and the atomic behavior of each system call is taken from the sequential file system specification. However, it precludes sophisticated optimizations that arise from fine-grained locking within system calls and result in a relaxed but more concurrent specification.

[1] By "disk" we mean any storage device, including rotational disks and SSDs.

Even with our modest concurrency goals, the file system must have a plan for coordinating the concurrent execution of system calls. We use a global lock for serializing the execution of writers, and use snapshots to ensure consistent execution of readers. For concurrent disk I/O, we explicitly yield when waiting for I/O completion, so that another system call can execute in the meantime, possibly issuing additional concurrent I/O requests to the disk.

The file system does not know upfront if a given system call will require I/O (due to reading an address not in the cache) or update the system state (by writing to memory). Instead, it optimistically initially executes in read-only mode and aborts; if any update is required, it retries with a global write lock. This allows read-only system calls to run concurrently on different calls, when possible, but still provides correct execution when some file-system state needs to be updated. When a disk read is required, we achieve I/O concurrency by releasing the global lock while waiting for the disk, returning to the system call that needed the disk block after updating the cache.

There are two nuances to aborting and retrying which we must correctly handle when writing an OSC. The first is that before aborting, a system call may have made some changes. Exposing these changes to other system calls would break the atomicity of system calls. We address this problem by taking a snapshot of in-memory state and rolling back to this snapshot upon abort; this is easy to implement since the file system is written in a functional language with the state in a persistent data structure. The second issue is that between restarting and acquiring the global lock other threads may run. It is important to ensure that if the system call was safe to run initially it continues to be safe even after other threads run. We address this concern by assuming a global protocol on file system usage where each thread is restricted to a directory disjoint from the others and thus does not unduly modify other threads' state. Note that the protocol places no restriction on what directories threads may *read* from, only writes.

3. Related work

There is a great deal of related work we build upon and extend, encompassing file systems work, verified systems, program logics and other formal methods, and techniques for concurrent programming. We give a necessarily brief overview here.

Modern file systems have taken an incremental approach to supporting concurrency, going from a big kernel lock to finer-grained locking over time. A good recent example of applying this incremental approach is NetApp's Waffinity file system [10]. The approach taken in that work has a similar goal of adding concurrency while re-using a (large) existing implementation, though in an unverified context.

There are several verified file systems, including Yggdrasil [25] and Cogent [1], but the file system CFSCQ builds upon is FSCQ [5], where we were able to make the necessary changes to the execution model and verification framework. Related to our work is other work on verifying concurrent systems, though none of these have been applied to file systems. These include work on verifying Hyper-V [6–8], a concurrent garbage collector [15], the CertiKOS concurrent operating system [4, 13], and verified distributed systems [14, 19, 26]. These approaches differ widely in how they approach, model, and verify concurrent systems; none could be retro-fitted onto an existing verified file system, without significant changes to the code and especially proofs.

There are many formal methods techniques relevant to our work. These include linearizability [16], work on specifying file systems [21, 22], and concurrent logics [2, 11, 12, 17].

Our OSC approach is in some ways similar to a long line of work on optimistic concurrency control [18, 23] as well as software transactional memory (STM) [24].

4. Preliminary results

We have implemented and verified CFSCQ in the Coq proof assistant [9], building on top of FSCQ's implementation and proofs. The approach follows FSCQ in many respects. We wrote a verification framework to model cooperative concurrency, a high-level view of shared memory, and disk I/O operations with asynchronous reads. The model of disk writes is the same as in FSCQ's verification framework. CFSCQ includes a translator that translates FSCQ system calls into its concurrent verification framework, while also translating proofs. These result in verified optimistic system calls. Finally, CFSCQ wraps each system call in a retry loop and exposes a linearizable specification based on the FSCQ specification. Since the approach uses FSCQ's implementation in a black-box manner, we were able to use the latest version of FSCQ (as described in an upcoming SOSP 2017 paper [3]), benefitting from its significantly improved performance with no additional burden in our implementation or proofs.

The remaining work is to improve the performance of the CFSCQ prototype. The prototype uses Coq's extraction facility to generate executable Haskell code, which we link with a hand-written concurrent runtime. OSCs and the concurrent runtime have overheads and concurrency bottlenecks that we have not yet fixed. On the other hand, microbenchmarks of the I/O concurrency do achieve better performance than FSCQ's blocking reads on a slow-enough disk.

References

[1] Amani, S., Hixon, A., Chen, Z., Rizkallah, C., Chubb, P., O'Connor, L., Beeren, J., Nagashima, Y., Lim, J., Sewell, T., Tuong, J., Keller, G., Murray, T., Klein, G., and Heiser, G. COGENT: Verifying high-assurance file system implementations. In *Proceedings of the 21st International Conference on Architectural Support for*

Programming Languages and Operating Systems (ASPLOS) (Atlanta, GA, Apr. 2016), pp. 175–188.

[2] BROOKES, S. A semantics for concurrent separation logic. *Theoretical Computer Science 375*, 1–3 (May 2007). Festschrift for John C. Reynolds's 70th Birthday.

[3] CHEN, H., KONRADI, A., WANG, S., CHAJED, T., İLERI, A., CHLIPALA, A., KAASHOEK, M. F., AND ZELDOVICH, N. Using the metadata-prefix specification to verify a high-performance crash-safe file system. In *Proceedings of the 26th ACM Symposium on Operating Systems Principles (SOSP)* (Shanghai, China, Oct. 2017).

[4] CHEN, H., WU, X., SHAO, Z., LOCKERMAN, J., AND GU, R. Toward compositional verification of interruptible os kernels and device drivers. In *Proceedings of the 2016 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Santa Barbara, CA, June 2016), pp. 431–447.

[5] CHEN, H., ZIEGLER, D., CHAJED, T., CHLIPALA, A., KAASHOEK, M. F., AND ZELDOVICH, N. Using Crash Hoare Logic for certifying the FSCQ file system. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)* (Monterey, CA, Oct. 2015), pp. 18–37.

[6] COHEN, E., DAHLWEID, M., HILLEBRAND, M. A., LEINENBACH, D., MOSKAL, M., SANTEN, T., SCHULTE, W., AND TOBIES, S. VCC: A practical system for verifying concurrent C. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics* (Munich, Germany, Aug. 2009).

[7] COHEN, E., MOSKAL, M., SCHULTE, W., AND TOBIES, S. A practical verification methodology for concurrent programs. Tech. Rep. MSR-TR-2009-2019, Microsoft Research, Feb. 2009.

[8] COHEN, E., MOSKAL, M., SCHULTE, W., AND TOBIES, S. Local verification of global invariants in concurrent programs. In *Proceedings of the 22nd International Conference on Computer Aided Verification (CAV)* (Edinburgh, UK, July 2010).

[9] COQ DEVELOPMENT TEAM. *The Coq Proof Assistant Reference Manual, Version 8.6.1*. INRIA, July 2017. `http://coq.inria.fr/distrib/current/refman/`.

[10] CURTIS-MAURY, M., DEVADAS, V., FANG, V., AND KULKARNI, A. To Waffinity and beyond: A scalable architecture for incremental parallelization of file system code. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)* (Savannah, GA, Nov. 2016), pp. 419–434.

[11] DINSDALE-YOUNG, T., BIRKEDAL, L., GARDNER, P., PARKINSON, M., AND YANG, H. Views: Compositional reasoning for concurrent programs. In *Proceedings of the 40th ACM Symposium on Principles of Programming Languages (POPL)* (Rome, Italy, Jan. 2013), pp. 287–300.

[12] FENG, X. Local rely-guarantee reasoning. In *Proceedings of the 36th ACM Symposium on Principles of Programming Languages (POPL)* (Savannah, GA, Jan. 2009).

[13] GU, R., SHAO, Z., CHEN, H., WU, X. N., KIM, J., SJÖBERG, V., AND COSTANZO, D. CertiKOS: An extensible architecture for building certified concurrent OS kernels. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)* (Savannah, GA, Nov. 2016), pp. 653–669.

[14] HAWBLITZEL, C., HOWELL, J., KAPRITSOS, M., LORCH, J. R., PARNO, B., ROBERTS, M. L., SETTY, S., AND ZILL, B. IronFleet: Proving practical distributed systems correct. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)* (Monterey, CA, Oct. 2015), pp. 1–17.

[15] HAWBLITZEL, C., PETRANK, E., QADEER, S., AND TASIRAN, S. Automated and modular refinement reasoning for concurrent programs. In *Proceedings of the 27th International Conference on Computer Aided Verification (CAV)* (San Francisco, CA, July 2015).

[16] HERLIHY, M. P., AND WING, J. M. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages Systems 12*, 3 (1990), 463–492.

[17] JONES, C. B. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems 5*, 4 (Oct. 1983), 596–619.

[18] KUNG, H. T., AND ROBINSON, J. T. On optimistic methods for concurrency control. *ACM Transactions on Database Systems 6*, 2 (June 1981), 213–226.

[19] LESANI, M., BELL, C. J., AND CHLIPALA, A. Chapar: Certified causally consistent distributed key-value stores. In *Proceedings of the 43rd ACM Symposium on Principles of Programming Languages (POPL)* (St. Petersburg, FL, Jan. 2016), pp. 357–370.

[20] LU, L., ARPACI-DUSSEAU, A. C., ARPACI-DUSSEAU, R. H., AND LU, S. A study of Linux file system evolution. *ACM Transactions on Storage 10*, 1 (Jan. 2014), 31–44.

[21] NTZIK, G., AND GARDNER, P. Reasoning about the POSIX file system: Local update and global pathnames. In *Proceedings of the 2015 Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications* (Pittsburgh, PA, Oct. 2015), pp. 201–220.

[22] RIDGE, T., SHEETS, D., TUERK, T., GIUGLIANO, A., MADHAVAPEDDY, A., AND SEWELL, P. SibylFS: formal specification and oracle-based testing for POSIX and real-world file systems. In *Proceedings of the 25th ACM Symposium on Operating Systems Principles (SOSP)* (Monterey, CA, Oct. 2015), pp. 38–53.

[23] SALTZER, J. H., AND KAASHOEK, M. F. *Principles of Computer System Design: An Introduction*. Morgan Kaufmann, 2009.

[24] SHAVIT, N., AND TOUITOU, D. Software transactional memory. In *Proceedings of the 14th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing* (Ottawa, Canada, Aug. 1995), pp. 204–213.

[25] SIGURBJARNARSON, H., BORNHOLT, J., TORLAK, E., AND WANG, X. Push-button verification of file systems via crash refinement. In *Proceedings of the 12th Symposium on Operating Systems Design and Implementation (OSDI)* (Savannah, GA, Nov. 2016), pp. 1–16.

[26] WILCOX, J. R., WOOS, D., PANCHEKHA, P., TATLOCK, Z., WANG, X., ERNST, M. D., AND ANDERSON, T. Verdi:

A framework for implementing and formally verifying distributed systems. In *Proceedings of the 2015 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)* (Portland, OR, June 2015), pp. 357–368.