POSIX is Dead! Long Live... errr... What Exactly?

Erez Zadok,[1] Dean Hildebrand,[2] Geoff Kuenning,[3] and Keith A. Smith[4]
[1]*Stony Brook University,* [2]*IBM Research—Almaden,* [3]*Harvey Mudd College,* [4]*NetApp*

Appears in the proceedings of the 9th USENIX Workshop on Hot Topics in Storage and File Systems (HotStorage'17)

Extended Abstract

The Problem. The POSIX system call interface is nearly 30 years old. It was designed for serialized data access to local storage, using single computers with single CPUs: today's computers are much faster, more complex, and require access to remote data. Security was not a major concern in POSIX's design, leading to numerous TOCTTOU attacks over the years and forcing programmers to work around these limitations [3]. Serious bugs are still being discovered, bugs that could have been averted with a more secure POSIX API design. POSIX's main programming model expects users to issue one synchronous call at a time and wait for its results before issuing the next. Today's programmers expect to issue many asynchronous requests at a time to improve overall throughput. POSIX's synchronous one-at-a-time API is particularly bad for accessing remote and cloud objects, where high latencies dominate.

A New Hope. By introducing compounding, a technique of packing multiple requests into one message, we have shown that we can save expensive context switches and data copies between user and kernel spaces [2]; and compounding NFS calls can save latency and improve throughput by orders of magnitude on WANs [1]. REST APIs dominate in remote/cloud stores thanks to improved efficiency (e.g., update whole file in a single PUT), but they do not go far enough (e.g., cannot PUT multiple files at once). Moreover, alternative APIs are less vulnerable to TOTTTOU attacks, for example those that do not require passing the same file name to successive system calls, but instead reuse the same open file/directory descriptor (e.g., `fstat`, `openat`).

Our Proposal. We propose to abolish and replace POSIX with a new API that is optimized for high-parallelism and high-latency, and has the following six properties. **(1)** Enabling *compounding* of any set of arbitrary calls, such as those defined by NFSv4. This would save substantial latency by eliminating many costly round-trips. As with NFSv4, the results of one call should be passed to the next. For example, one should be able to open, read, and close a file in a single compound, passing a successfully opened file descriptor from the open to the read and close calls. **(2)** Although the POSIX abstraction of files and namespaces has been very useful to users, we recognize the growing popularity of simpler cloud services that offer access to objects rather than files. Therefore, one should also be able to compound multiple operations on multiple objects in one request: read, write, create, rename, delete, etc. **(3)** All such new APIs should be asynchronous by default (and perhaps even *only* asynchronous). This will encourage (or force) users to write more efficient code. **(4)** Users should be able to define transactional semantics for their compounds. A transactional compound can eliminate many TOCTTOU bugs inherent in POSIX and improve end-to-end reliability between clients to eventual data holders (e.g., clouds). **(5)** Users should be allowed to define a compound's error-handling semantics. When one request in the middle of a compound fails, users should be allowed to designate whether to (a) stop the compound's processing and return any available results; (b) continue processing to the end and return a vector of success/failure information; or even (c) support if-then-else conditionals (*à la* NFSv4's NVERIFY call). **(6)** Users should be able to define the sequential and parallel portions of a compound. This ensures end-to-end parallelism and minimizes unwarranted serialization of operation execution to the storage device

Implementation and Transitioning. Transitioning to a new API will take time. To encourage easy adoption, we propose the following four steps: **(1)** Start by implementing the most common/useful compounds users would want (e.g., copy-file/object), adding more popular compounds as needed [1]. **(2)** Offer additional semantics as library wrappers, so as not to force all users into a new "low-level" API. **(3)** Offer a begin/end API to allow users to mark code segments to be turned into compounds, and start, commit, or abort when transactional semantics are desired. This would require R&D into compiler- and static-analysis-based techniques. **(4)** Finally, expose a low-level compounding API to enable new code to create and submit a compound with arbitrarily many operations encoded inside (with designated error, transactional, and parallelism semantics).

References

[1] M. Chen, D. Hildebrand, H. Nelson, J. Saluja, A. Subramony, and E. Zadok. vNFS: Maximizing NFS performance with compounds and vectorized I/O. In *USENIX FAST'17*, pages 301–314, 2017.

[2] A. Purohit, C. Wright, J. Spadavecchia, and E. Zadok. Cosy: Develop in user-land, run in kernel mode. In *ACM HOTOS 2003*, pages 109–114, 2003.

[3] J. Wei and C. Pu. TOCTTOU Vulnerabilities in UNIX-Style File Systems: An Anatomical Study. In *USENIX FAST'05*, pages 155–167, 2005.