Operating Systems Should Provide Transactions

Donald E. Porter and Emmett Witchel, The University of Texas at Austin
{*porterde,witchel*}@*cs.utexas.edu*

Abstract

Current operating systems provide programmers an insufficient interface for expressing consistency requirements for accesses to system resources, such as files and interprocess communication. To ensure consistency, programmers must to be able to access system resources atomically and in isolation from other applications on the same system. Although the OS updates system resources atomically and in isolation from other processes within a single system call, not all operations critical to the integrity of an application can be condensed into a single system call.

Operating systems should support transactional execution of system calls, providing a simple, comprehensive mechanism for atomic and isolated accesses to system resources. Preliminary results from a Linux prototype implementation indicate that the overhead of system transactions can be acceptably low.

1 Introduction

Operating systems manage resources for user applications, but do not provide a mechanism for applications to group operations into logically consistent updates. The consistency of application data can be undermined by system failures and concurrency. Consistency is guaranteed by allowing critical operations occur atomically (i.e., they occur all at once or not at all) and in isolation from the rest of the system (i.e., partial results of a series of operations are not visible and cannot observe concurrent operations). Mechanisms for data consistency exist at different layers of the software stack. For instance, locks use mutual exclusion to provide consistency for user-level data structures, and database transactions provide consistent updates to database-managed secondary storage.

Unfortunately, the POSIX system call API has lagged behind in providing support for consistent updates to OS-managed resources. The OS executes a single system call atomically and in isolation, but it is difficult, if not impossible, for applications to extend these guarantees to an operation that is too complex to fit into a single system call. This paper proposes adding *system transactions* to the system call API. A system transaction executes a series of system calls in isolation from the rest of the system and atomically publishes the effects to the rest of the system. System transactions provide a simple and powerful way for applications to express consistency requirements for concurrent operations to the OS.

Only the application knows when its data is in a consistent state, yet system resources that are critical to ensuring consistent updates, such as the file system, are outside of user control. In simple cases, programmers can serialize operations by using a single system call, such as using `rename` to atomically replace the contents of a file. Unfortunately, more complex operations, such as software installation or upgrade, cannot be condensed into a single system call. An incomplete software install can leave the system in an unusable state. Executing the entire software install atomically and in isolation would be a powerful tool for the system administrator, but no mainstream operating system provides a combination of system abstractions that can express it.

In the presence of concurrency, applications must ensure consistency by isolating a series of modifications to important data from interference by other tasks. Concurrency control mechanisms exposed to the user (e.g., file locking) are clumsy and difficult to program. Moreover, they are often insufficient for protecting a series of system calls from interference by other applications running on the system, especially when the other applications are malicious.

Figure 1 shows an example where an application wants to make a single, consistent update to the file system by checking the access permissions of a file and conditionally writing it. This pattern is common in setuid programs. Unfortunately, the application cannot express to the system its need for the `access` and `open` system calls to see a consistent view of the filesystem namespace.

The inability of an application to consistently view and update system resources results in serious security and programmability problems. The example in Figure 1 illustrates a time-of-check-to-time-of-use (TOCTTOU) race condition, a major and persistent security problem in modern operating systems. During a TOCTTOU attack, the attacker changes the file system namespace using symbolic links between the victim's access control check and its actual use, perhaps tricking a `setuid` program into overwriting a sensitive system file like the password database. TOCTTOU races also arise in temporary file creation and other accesses to system resources. While conceptually simple, TOCTTOU attacks are present in much deployed software and are difficult to eliminate. At the time of writing, a search of the U.S. national vulnerability database for

```
     Victim                 Attacker

if(access('foo')){
                          symlink('secret','foo');
  fd=open('foo');
  write(fd,...);
  ...
}
```

```
     Victim                 Attacker

                          symlink('secret','foo');
sys_xbegin();
if(access('foo')){
  fd=open('foo');
  write(fd,...);
  ...
}
sys_xend();
                          symlink('secret','foo');
```

Figure 1: An example of a TOCTTOU attack, followed by an example of eliminating the race with system transactions. The attacker's symlink is serialized (ordered) either before or after the transaction, and the attacker cannot see partial updates from the victim's transaction, such as changes to `atime`.

the term "symlink attack" yields over 600 hits [3].

In practice, the lack of concurrency control in the system call API has been addressed in an *ad hoc* manner by adding new, semantically heavy system calls for each new problem that arises. Linux has been addressing TOCTTOU races by encouraging developers to traverse the directory tree in user space rather than in the kernel using the recently introduced *openat()* family of system calls. Similarly, Linux kernel developers recently added a new close-on-exec flag to fifteen system calls to eliminate a race condition between calls to `open` and `fcntl` [6]. Individual file systems have introduced new operations, such as the Google File System supporting atomic append operations [7] or Windows adding transaction support to NTFS [11]. Rather than requiring users to lobby OS developers for new system calls, why not allow users to solve their own problems by composing a series of simple system calls into an atomic and isolated unit?

In this position paper, we advocate adding system transactions to the system call API to provide the user a simple and powerful mechanism to express consistency requirements for system resources. The relative success of parallel programming with database transactions as compared to threads and locking is a strong indicator that transactions are a useful, natural abstraction for programmers to reason about consistency. By wrapping a series of system calls in a transaction, programmers can continue using the POSIX API in a secure manner, eliminating the need for many of the complicated API changes that have been recently introduced. Developers can also protect concurrency in a natural way, reducing code complexity and potentially gaining performance, e.g., eliminating lock files and allowing concurrent file updates instead of using a database. This paper also shows that system transactions can be efficient with preliminary data from *TxOS*, a prototype implementation on the Linux kernel.

2 System Transactions

System transactions provide atomicity, consistency, isolation, and durability (ACID) for system state. The only application code change required to use system transactions is to enclose the relevant code region within the appropriate system calls: `sys_xbegin()`, `sys_xabort()`, and `sys_xend()`. Placing system calls within a transaction changes the semantics of when and how their results are published to the rest of the system. Outside of a transaction, actions on system resources are visible as soon as the relevant internal kernel locks are released. Within a transaction, all updates are kept isolated until commit, when they are atomically published to the rest of the system.

2.1 Previous transactional operating systems

Locus [19] and QuickSilver [15] are historical systems that provide some system support for transactions. Both systems implement transactions using database implementation techniques, namely isolating data structures with two-phase locking and rolling back failed transactions with an undo log. One problem with this locking scheme is that simple reader-writer locks do not capture the semantics of container objects, such as directories. Multiple transactions can concurrently and safely create files in the same directory so long as none of them use the same file name and none of them read the directory. Unfortunately, creating a file in these historical systems requires a write lock on the entire directory, which needlessly serializes operations and eliminates concurrency. To compensate for the poor performance of reader/writer locks, both systems allow directory contents to change during a transaction, which reintroduces the possibility of time-of-check-to-time-of-use (TOCTTOU) race conditions that system transactions ought to eliminate.

We propose a design for system transactions that provides stronger semantics than these historical systems and helps address the problems of concurrent programming on current and future generations of multi-core hardware.

2.2 Implementation sketch

The key goal of a system transaction implementation should be to provide strong atomicity and isolation guarantees to transactions while retaining good performance. This section outlines how our TxOS prototype achieves these goals; a detailed design of our prototype is available as a technical report [12].

TxOS implements a custom, object-based software transactional memory system to checkpoint and rollback many kernel data structures, including objects that represent file system metadata or a process's address space. To isolate updates to kernel data structures, TxOS adopts *lazy version management* [8], where transactions operate on private copies of a data structure. Unlike traditional approaches that use two-phase locking, lazy versioning is a good match for an OS because no kernel locks are held when returning to the application from a transactional system call. Lazy versioning also allows a high-priority or real-time process to quickly abort a lower priority process, as a non-transactional thread does not need to wait for the transactional victim to walk its undo log.

TxOS leverages existing OS buffers to isolate data read and written by a transaction. When an application writes data to a file or device, the updates generally go into an OS buffer first, allowing the OS to optimize device accesses. By making these buffers copy-on-write for transactions, TxOS isolates transactional data accesses until commit.

Buffering updates in memory during transactions limits the size of transactions, and restricts the transactional model. For instance, if an application writes a message to the network, it will be buffered until commit and the application cannot expect a response to the message within the same transaction. Future work could examine using secondary storage to help buffer changes and extending transactions over the network; this paper argues for a more expressive system call framework that can serve as the interface for future enhancements.

2.3 Fairness

TxOS can schedule transactional and non-transactional threads more fairly than historical transactional operating systems. TxOS protects transactions from interference by non-transactional system calls by having all threads use the same locking discipline, and by having transactions annotate accessed objects. When a thread, transactional or non-transactional, accesses an object for the first time, it must check for a conflicting annotation and have the scheduler arbitrate the conflict. In many cases, this check is performed at the same time a thread acquires a lock for the object. When TxOS detects a conflict before a critical section begins, the scheduler can safely suspend a conflicting non-transactional thread or abort a transaction, giving the scheduler substantial latitude to ensure fairness and prevent starvation of tasks, whether transactional or not.

2.4 Durability

For system state to remain consistent across crashes, updates from committed transactions must be durable—they must reside on stable storage. Durability is only relevant for some system resources, like file systems on non-volatile storage. Providing durability often slows performance because of the increased latency of stable storage, so users should have the option of relaxing it when they do not need it. For instance, eliminating the TOCTTOU race described above does not require durable updates.

In the TxOS design, specific file system implementations are responsible only for not writing intermediate transaction results to disk and atomically writing a group of updates at commit. This atomic write could be implemented with journaling, copy-on-write semantics, or a transactional file system [11, 20]. Conflict detection and object versioning occur in common code shared by all file systems. Because the implementation of durability is the most thoroughly studied in previous work, this paper focuses on other aspects of system transactions.

2.5 Transactions for system state

System transactions provide programmers with atomicity and isolation for system state, as opposed to application state. System state includes OS data structures and device state, whereas application state is stored in the application's data structures within its address space. When using system transactions, the application must be able to restore its pre-transaction state if a system transaction aborts. Application state can be managed in several ways: the application state might need no explicit management (as in the TOCTTOU example), the OS can automatically checkpoint and restore the application's address space (as in Speculator [10]), or the application can implement its own checkpoint and recovery mechanism, perhaps using hardware or software transactional memory.

2.6 Related work

The system transactions supported by TxOS solve a fundamentally different problem from those solved by TxLinux [14]. TxLinux is a Linux kernel that uses hardware transactional memory as a synchronization technique within the Linux kernel, whereas TxOS provides transactions to programmers as part of the OS API on currently available hardware. The techniques used to build TxLinux support short critical regions that enforce consistency for accessing memory: these techniques are insufficient to implement TxOS, which must guarantee consistency across heterogeneous system resources, and which must support arbitrarily large transactions.

Speculator [10] applies an isolation and rollback mechanism to the operating system that is similar to transactions. This mechanism allows their system to speculate past high-latency remote file system operations. Providing transactional semantics to users is a more complicated endeavor, as transactions must be isolated from each other, whereas applications in Speculator share speculative results. If a TOCTTOU attack were to occur under Speculator, the attacker and victim would be part of the same speculation and the attack would be successful. Speculator has been extended to tasks including debugging system configuration [17], but it does not provide a general-purpose interface for users to delimit speculation, and is thus insufficient for applications like atomic software installation.

3 Applications for system transactions

This section examines several classes of applications where open problems can be cleanly addressed by grouping system calls within transactions.

3.1 Isolation for concurrent performance

By providing a simple isolation primitive, system transactions expose new opportunities for enhancing concurrent performance of computer systems. For instance, when a system administrator performs a major distribution upgrade with `apt`, a system will typically upgrade over a thousand packages, requiring an hour of down time or more. In many cases, packages update disjoint files without ordering dependences, and are thus safe to install concurrently. Because some packages cannot be installed concurrently and safely, apt performs all installs sequentially. This design harms common case performance because it cannot easily guarantee correctness in the worst case.

Similarly, simple web applications often use relatively heavyweight database management systems to provide concurrency control for a small amount of application data, leading to higher single-thread overheads as well as the management complexity and security risks that come with databases. Although databases perform an important role in providing sophisticated queries and optimizing accesses to very large data sets, system transactions can provide a lightweight solution for applications with more modest datasets and query requirements. System transactions allow an application to group multiple writes to a file while shielding readers from seeing a file in an inconsistent state during an update. That power is all many web applications need.

System transactions also allow new ways of expressing producer/consumer relationships, enhancing the usability of systems. Consider a user who is downloading music files that must be converted to a format supported by his or her player. When the producer does not easily fit into a UNIX-style pipeline (such as a GUI program), attempts to overlap the conversion with the download creates irritating partially converted files. If the producer and consumer ran in transactions designated as a producer and consumer pair, the system could coordinate the transactions so that the consumer serializes after the producer and waits for the producer, if necessary. Ramadan et al. describe the safety conditions for this sort of transaction coordination, called dependence-awareness [13]. Dependence-aware transactions improve usability by providing a simple way to coordinate data producers and consumers that do not communicate through pipes or sockets.

3.2 Eliminating races for security

System-level race conditions threaten security because attackers can change system state in between a check and a modification. Although system-level races are most commonly presented in the context of the file system namespace, they also occur in OS subsystems other than the file system. Zalewski demonstrates how races in signal handlers can be used to crack applications, including sendmail, screen, and wu-ftpd [21]. These races can be remedied by the new `sigaction` API, which allows application developers to disable signals in a manner similar to OS developers disabling interrupts during critical regions. Transactions provide a simpler alternative by serializing signal delivery before or after system transactions.

Local sockets used for IPC are vulnerable to a similar race between creation and connection. Versions of OpenSSH before 1.2.17 suffered from a socket race exploit that allowed a user to steal another's credentials [1], and the Plash sandboxing system has a similar exploit [2]. To prevent this socket race with system transactions, the transaction API must allow multiple processes to participate in a transaction.

Race conditions in a single file system can be addressed with a transactional file system, such as TxF [11], and Valor [16], but a transactional file system cannot address race conditions for other system resources, such as network sockets or signal handlers. Transactional file systems also cannot address races that involve non-transactional file systems. Races in the creation of temporary files in privileged programs provide a common attack vector [5]. Many Unix distributions use a memory-only file system for `/tmp`. On these distributions, temporary file creation is not protected even if the root file system is transactional. We propose implementing system transactions primarily at the VFS layer, allowing the OS to isolate a series of operations that span multiple, transactional and non-transactional file systems.

3.3 Transactional memory

With the exception of TxLinux, transactional memory systems are used to synchronize threads in user-level applications. An open problem for transactional memory is supporting system calls within a transaction, because some system calls can have effects that are difficult to undo if a transaction must restart[1]. There have been a number of user-level approaches to this problem that leverage logical undo complements of some system calls (e.g., `open`/`close`). These proposals include open nesting [9], escape actions [22], and xCalls [18].

There are two problems with user-level support for system calls in transactional memory. First, system calls may have side effects that are difficult to detect and rollback (e.g., `munmap` may remove the last link to a file and delete it). Second, most user-level techniques only isolate updates to system resources from other threads in the same application by using user-level locking. In order to be appropriate for tools that affect system state, such as a software installer, transactions must isolate partial updates from all processes on the system. System transactions provide a

[1]Isolation is also violated by exposing the effects to the rest of the system immediately.

Bench		Linux	TxOS	
lfs small	create	4.5	5.5	1.2×
	read	1.2	1.0	0.8×
	delete	0.1	1.2	12.0×
lfs large	write seq	1.38	0.34	0.2×
	read seq	0.04	0.13	3.2×
	write rnd	1.60	0.36	0.2×
	read rnd	0.07	0.14	2.0×

Figure 2: Execution time in seconds for the LFS benchmarks on TxOS and slowdown relative to Linux. The LFS small benchmark operates on 10,000 files of length 100 bytes, and the large benchmark reads and writes a 100MB file.

comprehensive solution for system-wide updates, as system transactions have access to internal OS data structures and can isolate all of the side effects of a system call. Similarly, system transactions can leverage OS scheduling and synchronization to ensure system-wide isolation.

4 Transaction overhead

We evaluate the feasibility of system transactions by evaluating our TxOS prototype, derived from Linux 2.6.22.6. All experiments are performed on a 4 core Intel X5355 processor running at 2.66GHz with 4 GB of RAM.

Figure 2 shows the performance of TxOS on the LFS benchmarks. For workloads that run for more than one second, the overhead of system transactions is under 20%. The LFS large phases that repeatedly write files in a transaction are more efficient than Linux because transaction commit groups the writes and presents them to the I/O scheduler at once, improving disk arm scheduling. Write-intensive workloads out-perform non-transactional writers by as much as a factor of 5×.

4.1 Implementation complexity

System transactions in TxOS add roughly 8,600 lines of code to the kernel and require about 14,000 lines of minor changes to kernel code, such as replacing pointer dereferences with wrapper functions that detect conflicts. Although adding system transactions increases the implementation complexity of the operating system, overall system code complexity can be reduced because many applications can eliminate the *ad hoc* code currently needed to approximate atomic accesses to system resources.

Scaling the performance of current systems with increasing core counts requires substantial implementation effort. High-performance concurrent programming is difficult and requires more pervasive changes than most systems research proposals of the previous decade. For instance, Corey [4] addresses a number of scaling problems in Linux by redesigning and reimplementing OS data structures and user APIs. Similary, substantial engineering effort is required to implement system transactions, but transactions can help programmers write performance scalable programs. Given how few tools programmers have for writing concurrent applications in the multi-core era, the value of a useful abstraction justifies a challenging implementation.

5 Summary

Adding efficient transactions to the Linux system call API provides a natural way for programmers to synchronize access to system resources, a problem currently solved in an *ad hoc* manner. This paper argues that system transactions elegantly solve a number of important, long-standing problems ranging from system security to performance scalability.

6 Acknowledgements

The authors thank Chris Rossbach, Mike Walfish, and the anonymous reviewers for their helpful comments.

References

[1] http://www.employees.org/ satch/ssh/faq/TheWholeSSHFAQ.html.
[2] http://plash.beasts.org/wiki/PlashIssues/ConnectRaceCondition.
[3] National Vulnerability Database. http://nvd.nist.gov.
[4] S. Boyd-Wickizer, H. Chen, R. Chen, Y. Mao, F. Kaashoek, R. Morris, A. Pesterev, L. Stein, M. Wu, Y. Dai, Y. Zhang, and Z. Zhang. Corey: An operating system for many cores. In *OSDI*, 2008.
[5] H. Chen, D. Dean, and D. Wagner. Model checking one million lines of C code, 2004.
[6] U. Drepper. Secure file descriptor handling. http://udrepper.livejournal.com/20407.html.
[7] S. Ghemawat, H. Gobioff, and S.-T. Leung. The Google file system. *SOSP*, 2003.
[8] J. Larus and R. Rajwar. *Transactional Memory*. Morgan & Claypool, 2006.
[9] M. J. Moravan, J. Bobba, K. E. Moore, L. Yen, M. D. Hill, B. Liblit, M. M. Swift, and D. A. Wood. Supporting nested transactional memory in LogTM. In *ASPLOS*, 2006.
[10] E. B. Nightingale, P. M. Chen, and J. Flinn. Speculative execution in a distributed file system. In *SOSP*, 2005.
[11] J. Olson. Enhance your apps with file system transactions. *MSDN Magazine*, July 2007.
[12] D. E. Porter, O. S. Hofmann, C. J. Rossbach, A. Benn, and E. Witchel. Operating systems transactions. Technical report, UT Austin TR 09-13, 2009.
[13] H. E. Ramadan, I. Roy, M. Herlihy, and E. Witchel. Committing conflicting transactions in an STM. *PPoPP*, 2009.
[14] C. Rossbach, O. Hofmann, D. Porter, H. Ramadan, A. Bhandari, and E. Witchel. TxLinux: Using and managing transactional memory in an operating system. In *SOSP*, 2007.
[15] F. Schmuck and J. Wylie. Experience with transactions in QuickSilver. In *SOSP*. ACM, 1991.
[16] R. Spillane, S. Gaikwad, M. Chinni, E. Zadok, and C. P. Wright. Enabling transactional file access via lightweight kernel extensions. *FAST*, 2009.
[17] Y.-Y. Su, M. Attariyan, and J. Flinn. AutoBash: Improving configuration management with operating system causality analysis. In *SOSP*, 2007.
[18] H. Volos, A. J. Tack, N. Goyal, M. M. Swift, and A. Welc. xCalls: Safe I/O in memory transactions. In *EuroSys*, 2009.
[19] M. J. Weinstein, J. Thomas W. Page, B. K. Livezey, and G. J. Popek. Transactions and synchronization in a distributed operating system. In *SOSP*, 1985.
[20] C. P. Wright, R. Spillane, G. Sivathanu, and E. Zadok. Extending ACID semantics to the file system. *Trans. Storage*, 3(2):4, 2007.
[21] M. Zalewski. Delivering signals for fun and profit. 2001. http://lcamtuf.coredump.cx/signals.txt.
[22] C. Zilles and L. Baugh. Extending hardware transactional memory to support non-busy waiting and non-transactional actions. In *TRANSACT*, Jun 2006.