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

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