Operating Systems Should Provide Transactions

Donald E. Porter, Indrajit Roy, Andrew Matsuoka, Emmett Witchel
Department of Computer Sciences, The University of Texas at Austin, Austin, TX 78712
{porterde,indrajit,matsuoka,witchel}@cs.utexas.edu
Submission to OSDI '08, please do not distribute

Abstract

Operating systems can efficiently provide *system transactions* to user applications, in which user-level processes can execute a series of system calls atomically and in isolation from other processes on the system. The effects of system calls performed during a system transaction are not visible to the rest of the system (other threads or hardware devices) until the transaction commits. This paper is the first to implement system transactions with recent techniques from the transactional memory literature. TxOS, a variant of Linux 2.6.22 which we modified to support system transactions, can solve problems in a wide range of domains, including security, isolating extensions, and user-level transactional memory. We also show that combining semantically lightweight system calls to perform heavyweight operations can yield better performance scalability: enclosing `link` and `unlink` within a system transaction outperforms `rename` on Linux by 14% at 8 CPUs.

1 Introduction

The challenge of system API design is finding a small set of easily understood abstractions that compose naturally and intuitively to solve diverse programming and systems problems. Using the file system as the interface for everything from data storage to character devices and inter-process pipes is a classic triumph of the Unix API that has enabled large and robust applications. We show that system transactions are a similar, broadly applicable abstraction: transactions belong in the system-call API. Without system transactions, important functionality is impossible or difficult to express.

System transactions allow a user to transactionally group a sequence of system calls, for example guaranteeing that two writes to a file are either both seen by a reader or neither are seen. System transactions provide atomicity (they either execute completely or not at all) and isolation (in-progress results are not visible so transactions can be serially ordered). The user can start a system transaction with the `sys_xbegin()` system call, she can end a transaction with `sys_xend()` and abort it with `sys_xabort()`. The kernel makes sure that all system calls between an `sys_xbegin()` and an `sys_xend()` execute transactionally.

This paper introduces TxOS, a variant of Linux 2.6.22 which supports system transactions. TxOS is the first operating system to support transactions that allow any sequence of system calls to execute atomically and in isolation. It is also the first to apply current software transactional memory (STM) techniques which make transactions more efficient and which allow a flexible contention management policy among transactional and non-transactional operations. This flexibility lets the system balance scheduling and resource allocation between transactional and non-transactional operations.

We use TxOS to solve a variety of systems problems, which indicates that system transactions earn their position in the API. System transactions can eliminate time-of-check-to-time-of-use (TOCTTOU) race conditions, they can isolate an application from some misbehaviors in libraries or plugins, and they allow user-level transactions to modify system resources.

An important class of current security vulnerabilities consist of time-of-check-to-time-of-use (TOCTTOU) race conditions. During a TOCTTOU attack, the attacker changes the file system using symbolic links while a victim (such as a setuid program) checks a particular file's credentials and then uses it (e.g., writing the file). Between the credential check and the use, the attacker compromises security by redirecting the victim to another file— perhaps a sensitive system file like the password file. At the time of writing, a search of the U.S. national vulnerability database for the term "symlink attack" yields over 400 hits [28]. System transactions can eliminate TOCTTOU race conditions. If the user starts a system transaction before doing their system calls (e.g., an `access` and `open`), then the OS will guarantee that the interpretation of the path name used in both calls will not change during the transaction.

Having an API for transactions frees the system from supporting complex semantics that have accrued in their absence. For example, text editors [1] and source code control systems [3] use the `rename` system call heavily because of its strong atomicity and isolation properties— renames either successfully complete or they leave no trace of partial execution. Allowing the user to com-

bine semantically simple system calls, such as `link` and `unlink`, within a transaction more clearly expresses his intent, increases the performance scalability of the system, and reduces the implementation complexity for the operating system.

User-level transactions, such as those provided by a transactional memory system, run into trouble if they need to update system state. Such transactions cannot simply make a system call, because doing so violates their isolation guarantees. System transactions provide a mechanism for the transactional update of system state and in Section 3.4 we show how to coordinate user- and system- level transactions into a seamless whole with full transactional semantics.

In order to support system transactions, the kernel must be able to isolate and undo updates to shared resources. This adds latency to system calls, but we show that it can be acceptably low (13%–327%) within a transaction, and 10% outside of a transaction). However, using system transactions can provide better performance scalability than locks as we show with a web server in Section 5.4, uses transactions to increase throughput $2\times$ over a server that uses fine-grained locking.

This paper makes the following contributions:

- a new approach to implementating system transactions, that provides strong atomicity and isolation guarantees with low performance overhead, implemented in Linux 2.6.
- shows that semantically lightweight system calls can be combined within a transaction to provide better performance scalability and ease of implementation than complex system calls. Placing `link` and `unlink` in a transaction outperforms `rename` on Linux by 14% at 8 CPUs.
- demonstration of the use of system transactions to avoid TOCTTOU races whose performance is superior to the current state-of-the-art user-space technique [43].
- showing how system transactions can be used to isolate some faults in software plugins and libraries with minimal performance overhead.
- showing how to maintain transactional semantics for user-level transactions that modify system state.

The paper is organized as follows: Section 2 provides background on each of the example problems that we address with transactions. Section 3 describes the design of operating system transactions within Linux and Section 4 provides implementation details. Section 5 evaluates the system in the context of the target applications. Section 6 provides related work and section 7 concludes.

2 Overview and motivation

This section defines system level transactions and explains how they can be used by applications. It then describes three case studies with system transactions. The usage scenarios we choose demonstrate the applicability and advantages of system transactions across a wide range of systems problems.

2.1 System transactions

To describe system transactions, we must first make the distinction between *system state* and *application state*. Writes to the file system or forking a thread are actions that update system state, whereas updates to the data structures within an application's address space represent application state. System transactions are a novel and efficient way to provide the atomic and isolated access of transactions to system state.

A complete transactional programming model must provide an interface that is uniform with respect to both system and application state. Several techniques for supporting transactions that manage application state already exist, including transactional memory [18, 20] and recoverable virtual memory [35, 37]. We show in Section 3 that the operating system, by providing system transactions, can coordinate with multiple implementations of application transactions, giving applications the freedom to choose the best fit for their needs. For our target applications, we use both software transactional memory and an implementation of copy-on-write recovery to provide application transactions.

In adding the support for system transactions in Linux we have focused on system calls related to the file system. Transactions are implemented at the virtual file system layer. This has the advantage of providing a transactional interface to non-transactional file systems. At transaction commit, all of the changes are exposed to the file system at once, maintaining the safety guarantees of the underlying file system.

System transactions, as described in this work, commit their results to memory—the effect of a successful system transaction may not survive a reboot. Many applications can benefit from transactional semantics without durability. For instance, a file writer might want to make several updates to the contents of a file without concurrent readers seeing any of the individual writes. However, the writing application might be satisfied with the durability semantics of the underlying file system and does not require that the file contents by synced to disk at the conclusion of its update. While durability would be useful for system transactions, they are not necessary so we defer support to future work.

Previous research on incorporating transactional semantics into the operating system has focused on com-