Operating System Transactions

Blind submission to ASPLOS 2009, *Please do not distribute*

Abstract

Operating systems should provide *system transactions* to user applications, in which user-level processes execute a series of system calls atomically and in isolation from other processes on the system. System transactions provide a simple tool for programmers to express safety conditions during concurrent execution. This paper describes TxOS, a variant of Linux 2.6.22, which is the first operating system to implement system transactions on commodity hardware with strong isolation and fairness between transactional and non-transactional system calls.

System transactions provide a simple and expressive interface for user programs to avoid race conditions on system resources. For instance, system transactions eliminate time-of-check-to-time-of-use (TOCTTOU) race conditions in the file system which are a class of security vulnerability that are difficult to eliminate with other techniques. System transactions also provide transactional semantics for user-level transactions that require system resources, allowing applications using hardware or software transactional memory system to safely make system calls. While system transactions may reduce single-thread performance, they can yield more scalable performance. For example, enclosing `link` and `unlink` within a system transaction outperforms `rename` on Linux by 14% at 8 CPUs.

1 Introduction

The prevalence of concurrency due to the proliferation of multicore processors has created a problem for the system call API; current operating systems provide insufficient mechanisms for applications to synchronize these operations. 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. In this paper, we show that system transactions are a similar, broadly applicable abstraction and that transactions belong in the system-call API. Without system transactions, important functionality is impossible or difficult to express.

System transactions provide a simple tool for programmers to express safety conditions during concurrent execution. During a system transaction, the kernel insures that from the user's perspective, no other transaction or non-transactional system call occurs. The user experiences serial execution of the transactional code, eliminating race condtions. Within a system transaction, a series of system calls either execute completely or not at all (atomicity), and in-progress results are not visible (isolation). For example, if one program executes a system transaction that performs two writes to a file,

then any program reading that file either sees both writes or neither. System transactions have a simple interface: the user starts a system transaction with the `sys_xbegin()` system call, ends a transaction with the `sys_xend()` system call, and aborts the current transaction with the `sys_xabort()` system call.

This paper introduces TxOS, a variant of Linux 2.6.22 that supports system transactions on commodity hardware. TxOS is the first operating system to support transactions in which any sequence of system calls can execute atomically and in isolation. Unlike historical attempts to add transactions to the OS, transactions in TxOS have stronger semantics, are more efficient, and support a flexible contention management policy between transactional and non-transactional operations. TxOS is unique in its ability to enforce transactional isolation even for non-transactional threads, which is key for making system transactions practical, allowing the OS to balance scheduling and resource allocation fairly between transactional and non-transactional operations. TxOS achieves these goals by using modern, software transactional memory (STM) techniques.

Current operating systems address race conditions in the system API by adding complicated, *ad hoc* extensions to the API, whereas system transactions provide a simple, expressive interface for synchronizing access to system resources. For instance, time-of-check-to-time-of-use (TOCTTOU) race conditions are easier to exploit on multicore platforms [41]. In the past few years, Linux has added file system APIs that take file descriptor arguments to address TOCTTOU races (`openat`, `renameat`, `faccessat`, and ten others), and it has redesigned its signal-handling API (`sigaction`, `sigprocmask`, `pselect`, `epoll_pwait` and others). Such APIs are intended to solve specific races in a concurrent execution environment, but they have complex semantics and are difficult to learn and master. System transactions provide a single, easily understood, general mechanism that can express safe operations using simpler APIs, such as `open` or `signal`. Instead of fixing particular race conditions with new system calls, system transactions provide a general mechanism to eliminate race conditions completely.

TxOS adds transactions to its system call API, while TxLinux [36] uses hardware transactions to implement the same API as unmodified Linux. TxOS does not use hardware transactions at all, and by themselves hardware transactions are not sufficient to implement a transactional system call API. This paper describes the challenges to providing such an API.

TxOS lets user-level transactions make system calls with full transactional semantics. It provides a simple and semantically complete way for user-level transactions, such as those provided by hardware or software transactional memory systems, to access system resources. User-level transactions cannot make most system calls without violating isolation because system call results become visible to the rest of the system. Attempts to address this limitation are discussed in Section 2.2.1, but they either compromise transactional semantics or greatly increase the complexity and decrease the performance of the transactional memory implementation. In Section 3.4, we show how to coordinate user and system level transactions into a seamless whole while maintaining full transactional semantics.

To support system transactions, the kernel must isolate and undo updates to shared resources. This process adds

Victim **Attacker**

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

(A)

Victim **Attacker**

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

(B)

Figure 1: **(A)** An example of a TOCTTOU attack. **(B)** Eliminating the race with system transactions. The attacker's symlink is serialized (ordered) either before or after the transaction.

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, system transactions can provide better performance scalability than locks as we show with a web server in Section 5.5, which uses transactions to increase throughput 47% over a server that uses fine-grained locking.

This paper makes the following contributions:

- Describes a new approach to implementing system transactions on commodity hardware, which provides strong atomicity and isolation guarantees, while maintaining a low performance overhead.
- Demonstrates that system transactions can express useful safety conditions by eliminating race conditions while maintaining scalable performance. The performance of TxOS TOCTTOU elimination is superior to the current state-of-the-art user-space technique [40]. Placing `link` and `unlink` in a transaction can outperform `rename` on Linux by 14% at 8 CPUs.
- Shows how to maintain transactional semantics for user-level transactions that modify system state, and measures performance for integrating a software and a hardware transactional memory system with TxOS. We show that TxOS resolves a memory leak in genome (a STAMP benchmark [9]) without modification of the program or libc.

The paper is organized as follows: Section 2 motivates system transactions with examples of the problems they solve. 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 summarizes our findings.

2 Overview and motivation

This section motivates the need for system transactions by describing how they eliminate race conditions and how they complete the programming model of user-level transactions.

2.1 Eliminating races for security

Time-of-check-to-time-of-use (TOCTTOU) race conditions are a current source of serious security vulnerabilities, and a good example of the kinds of race conditions that system transactions can eliminate. Its most (in)famous instance is the