View metadata, citation and similar papers at core.ac.uk
brought to you by CORE
provided by Online Research Database In Technology

Technical University of Denmark

Safe Asynchronous System Calls - extended abstract

Brock-Nannestad, Laust; Karlsson, Sven

Publication date:
2014

Document Version
Publisher's PDF, also known as Version of record

Link back to DTU Orbit

Citation (APA):
Brock-Nannestad, L., & Karlsson, S. (2014). Safe Asynchronous System Calls - extended abstract. Abstract from International Symposium on Code Generation and Optimization, CGO 2014, Orlando, Florida, United States.

DTU Library
Technical Information Center of Denmark

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

Safe Asynchronous System Calls
extended abstract

Laust Brock-Nannestad and Sven Karlsson

November 15, 2013

1 Introduction

The typical interface between applications and the operating system is the *system call* or *trap*. It allows the application to request a service of the operating system. To handle the trap, the processor switches from the unprivileged mode of the application to the privileged mode of the kernel, who then handles the call. While the kernel handles the call, the caller is *blocked* and does not execute. Non-blocking calls which return before the operation has completed exist, for example the `aio` calls on UNIX systems, but are still implemented using the trap mechanism. Non-blocking calls expose some amount of parallelism in the caller, but each call incurs an overhead as the operating system saves and restores the context of the processor whenever it interrupts the application.

The individual nature of system calls makes them prone to programming errors and there is a long history of race condition exploits and file locking schemes used especially by mail systems. Many of these problems can be prevented if sequences of operations can be composed and executed atomically.

2 Design

We propose an alternate way of performing system calls with an asynchronous interface. We replace the traditional system call mechanism with a pair of queues implemented in shared memory. The caller sends packets conveying the same information as normal system calls. The operating system in turn polls this queue and executes the calls. Responses from the OS flow in the opposite direction using a similar queue. The queues provide an

asynchronous interface allowing the caller to enqueue operations and continue with other work while they are executed. This is illustrated in figure 1.

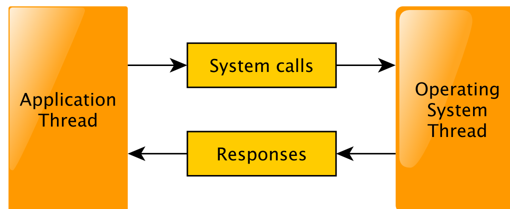

Figure 1: An application thread and the operating system communicate through shared queues.

The caller may depend on the result of an operation before it can continue. Rather than busy waiting, we introduce a *barrier*. Like a memory barrier it blocks the caller until all preceding system calls have completed. The barrier may be implemented as a trap or it may simply relinquish execution to a user mode scheduler.

To safely allow the grouping of related operations, we borrow the concept of transactions from database systems [2]. We encapsulate sequences of system calls as transactions and rely on the OS to ensure that each transaction either completes successfully or not at all and that concurrent transactions do not interfere. The OS is responsible for detecting *conflicts* and informing the caller of the failed transaction.

Figure 2 shows a code fragment implemented using traditional blocking system calls and with our approach. `send` and `get` are non-blocking operations on the queue. Conventional system call traps are highlighted in **bold**. `begin` and `commit` encapsulate the set of system calls inside a transaction.

3 Related work

The overhead introduced by frequent context switches has been investigated in the areas of inter-process communication and system calls. Shared memory buffers for efficient communication between processes was suggested by Bershad et al. [1]. Several scalable research operating systems, such as *Barrelfish* and *fos*, employ this technique.

Libflexsc by Soares and Stumm [4] implements asynchronous system calls on top of Linux. Instead of a queue they allocate an array of system call contexts in shared memory. The caller scans for a free context and writes

```
int fd=open("test.txt")
int *addr=mmap(fd,"RW")
*addr = 128
msync(addr)
close(fd)
```

(a) Trap based

```
send(begin)
send(open("test.txt"))
barrier()
int fd = get()
send(mmap(fd, "RW"))
barrier()
int *addr = get()
*addr = 128
send(msync(addr))
send(close(fd))
send(commit)
barrier()
```

(b) Queue based

Figure 2: Pseudocode for traditional and queue based calls. Trap based calls are marked in bold.

a system call into it. The kernel periodically scans contexts, finding new system calls to execute, and replacing the context with the result once done. Their experiments found that removing context switches primarily avoided data cache pollution.

Encapsulating system calls in transactions has been explored by Porter et al. in TxOS [3]. TxOS implements transactional system calls on top of the Linux kernel with an acceptable overhead. The implementation retains the trap-based interface and requires a context switch for each call even inside transactions.

4 Conclusion

This abstract proposes an asynchronous transactional system call interface. An implementation effort is underway on top of a small operating system. An evaluation will show if the removal of context switches, but addition of transaction overhead, remains acceptable.

References

[1] Brian N. Bershad et al. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2), 1991.

[2] Jim Gray et al. The transaction concept: Virtues and limitations. In *VLDB*, volume 81, 1981.

[3] Donald E. Porter et al. Operating system transactions. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*. ACM, 2009.

[4] Livio Soares and Michael Stumm. Exception-less system calls for event-driven servers. In *Proc. of USENIX Annual Tech. Conf*, 2011.