

Technical University of Denmark



Safe Asynchronous System Calls

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. Poster session presented at 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

Laust Brock-Nannestad and Sven Karlsson
Technical University of Denmark



Motivation

- ▶ Each traditional system call incurs a **context switch overhead**
- ▶ Traditional system calls are executed in **isolation**
 - ▶ Well known **security issues** due to lack of composability [5]
- ▶ Solutions
 - ▶ Issue multiple operations **asynchronously**
 - ▶ Receive responses asynchronously
 - ▶ Compose system calls using **transactions** [2]
 - ▶ Minimize context switch overhead
 - ▶ Beneficial for application and kernel to reside on different cores

Architecture

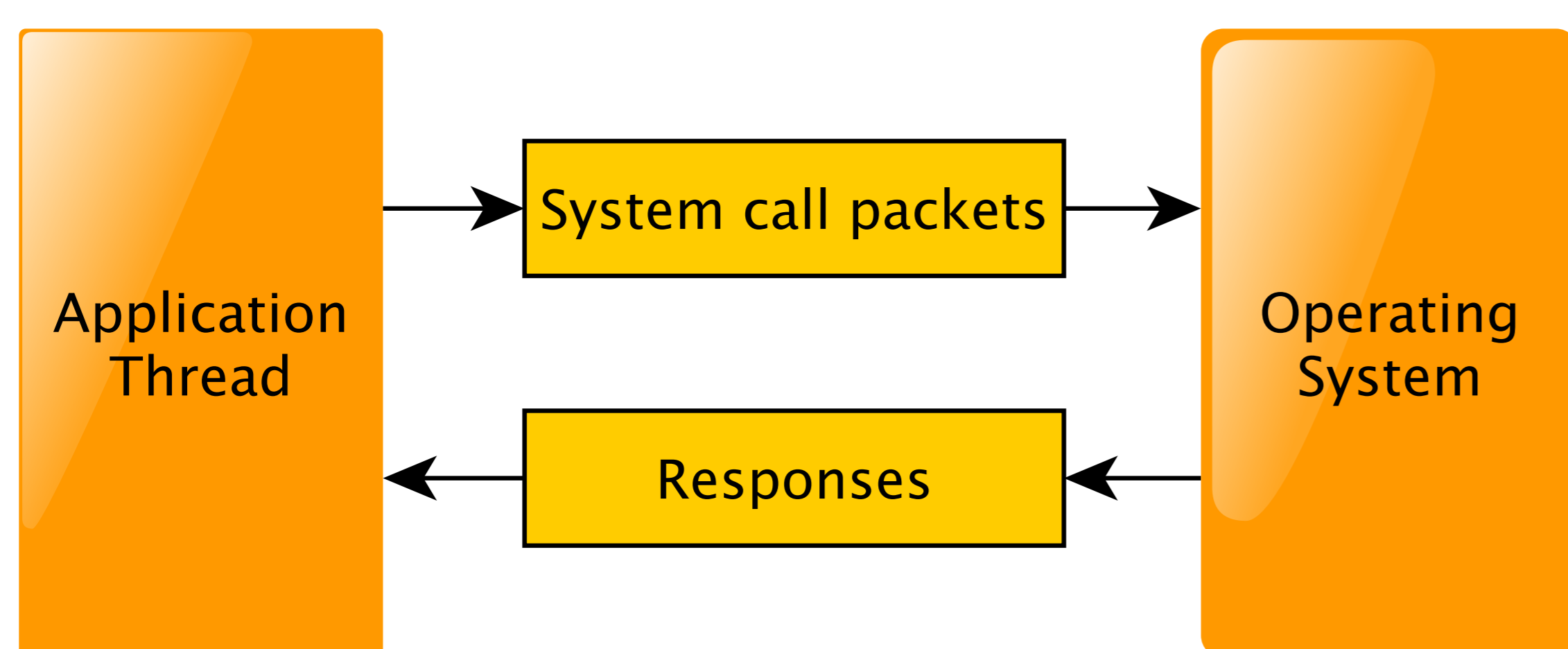


Figure : Queues in *shared memory* replace traditional system call traps.

- ▶ Queues are implemented as ring buffers in shared memory [1]
 - ▶ One sender, one receiver per buffer
 - ▶ Virtual Memory subsystem sets up shared mappings
 - ▶ Well known strategy for distributed operating systems
- ▶ No context switching required
- ▶ A transaction is local to each *thread* of a process

Code example

```
char *fn = "example.dat";  
  
int r = access(fn, W_OK);  
int fd = open(fn, O_WRONLY);  
  
/* Permissions read with  
access may no longer be  
valid */  
if (r == 0 && fd > 0)  
{  
    write(fd, ...);  
    close (fd);  
}
```

Figure : Conventional system call interface. Vulnerable to a race condition.

- ▶ Non-blocking primitives

- ▶ **send**
- ▶ **get**

- ▶ **barriers** make data dependencies explicit

- ▶ Thread is descheduled at a **barrier**.
- ▶ Thread is rescheduled once all preceding responses are ready

```
char *fn = "example.dat";  
send(begin);  
send(access(fn, W_OK));  
send(open(fn, O_WRONLY));  
barrier();  
int r = get();  
int fd = get();  
  
if (r == 0 && fd > 0)  
{  
    send(write(fd, ...));  
    send(close(fd));  
}  
send(commit);  
barrier();
```

Figure : Asynchronous system calls. Transaction ensures atomicity of access and open.

Per-process transaction management

- ▶ Each process has a thread dedicated to transaction management
- ▶ The kernel detects conflicts and manages kernel state

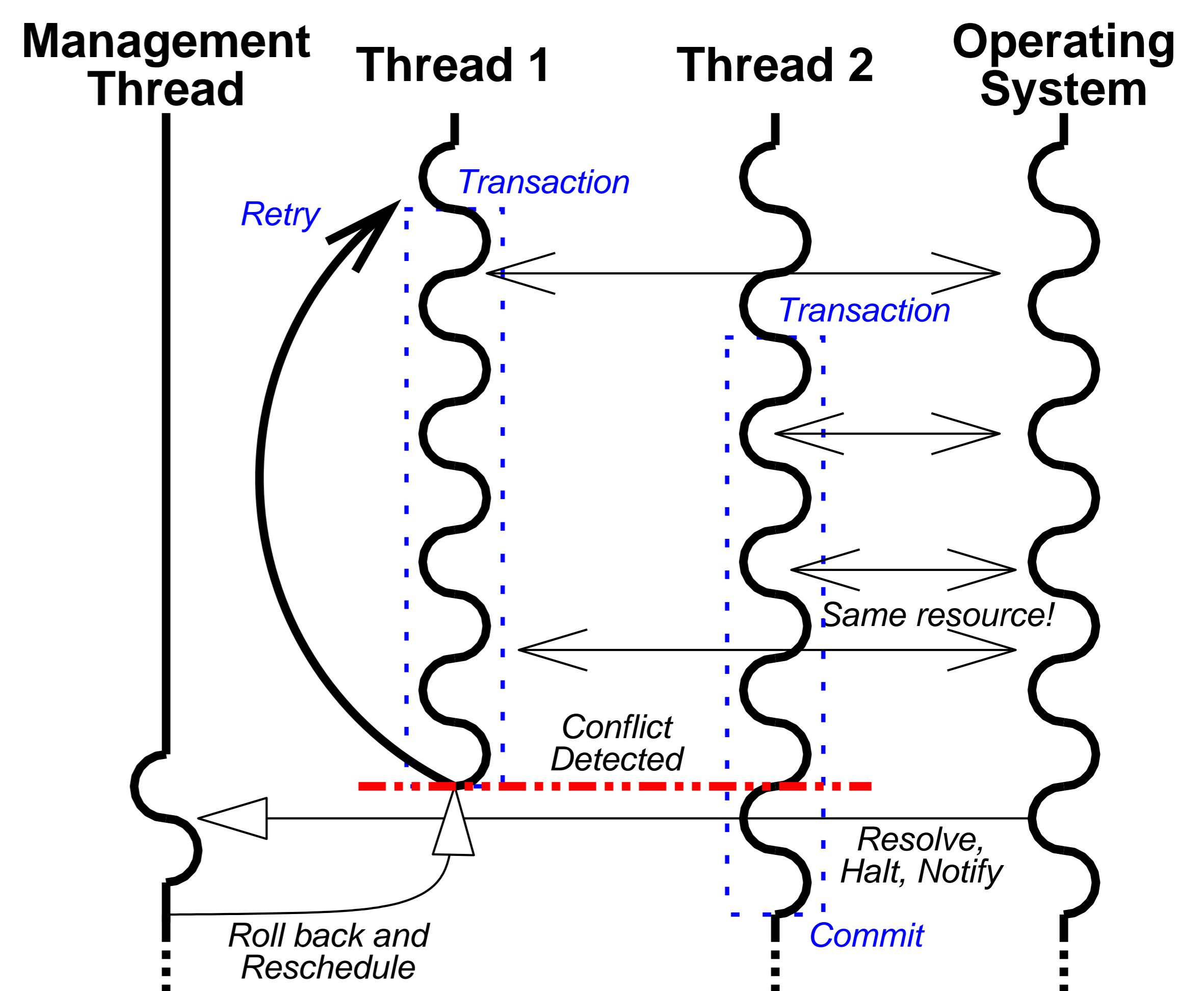


Figure : a per-process thread is responsible for transaction house keeping

Preliminary Evaluation

- ▶ Previous work shows a **22%** reduction in clocks-per-instruction [4]
 - ▶ Reduction is due to less pollution of data cache by kernel

Future Work

- ▶ Optimization of the implementation with respect to the memory hierarchy
- ▶ Integration into a fully fledged operating system
- ▶ Transaction based system libraries and run-time

Conclusion

- ▶ The traditional system call interface has several disadvantages
- ▶ Need to rethink the system call interface as part of a new operating system
- ▶ Transactions and system call communication through shared memory avoids common pitfalls
- ▶ Transactions are already central to new programming models
 - ▶ With operating system support, transactions span the entire software stack

Related work

- [1] BERSHAD, B. N., ET AL. User-level interprocess communication for shared memory multiprocessors. *ACM TOCS* 9, 2 (1991).
- [2] PORTER, D. E., ET AL. Operating system transactions. In *SIGOPS* 22 (2009), ACM.
- [3] RAJAGOPALAN, M., ET AL. Cassyopia: Compiler assisted system optimization. In *HotOS* (2003).
- [4] SOARES, L., AND STUMM, M. Exception-less system calls for event-driven servers. In *OSDI* (2011).
- [5] WEI, J., AND PU, C. Toctou vulnerabilities in unix-style file systems: An anatomical study. In *FAST* (2005).