# Safe Asynchronous System Calls

**Brock-Nannestad, Laust; Karlsson, Sven**

*Publication date:*
2014

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

# 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

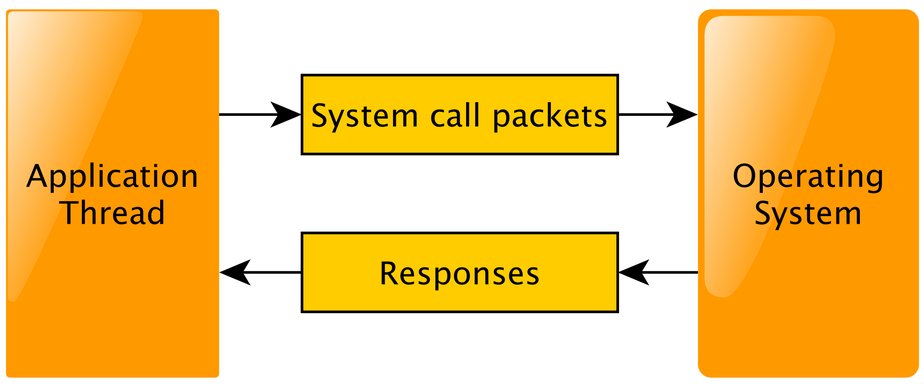


Figure : Queues in *shared memory* replace traditional system call traps.

- 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_WR);

/* 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.

```
char *fn = "example.dat";
send(begin);
send(access(fn, W_OK));
send(open(fh, O_WR));
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.

- Non-blocking primitives
  - **send**
  - **get**
- **barrier**s make data dependencies explicit
  - Thread is descheduled at a **barrier**.
  - Thread is rescheduled once all preceding responses are ready

## Per-process transaction management

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

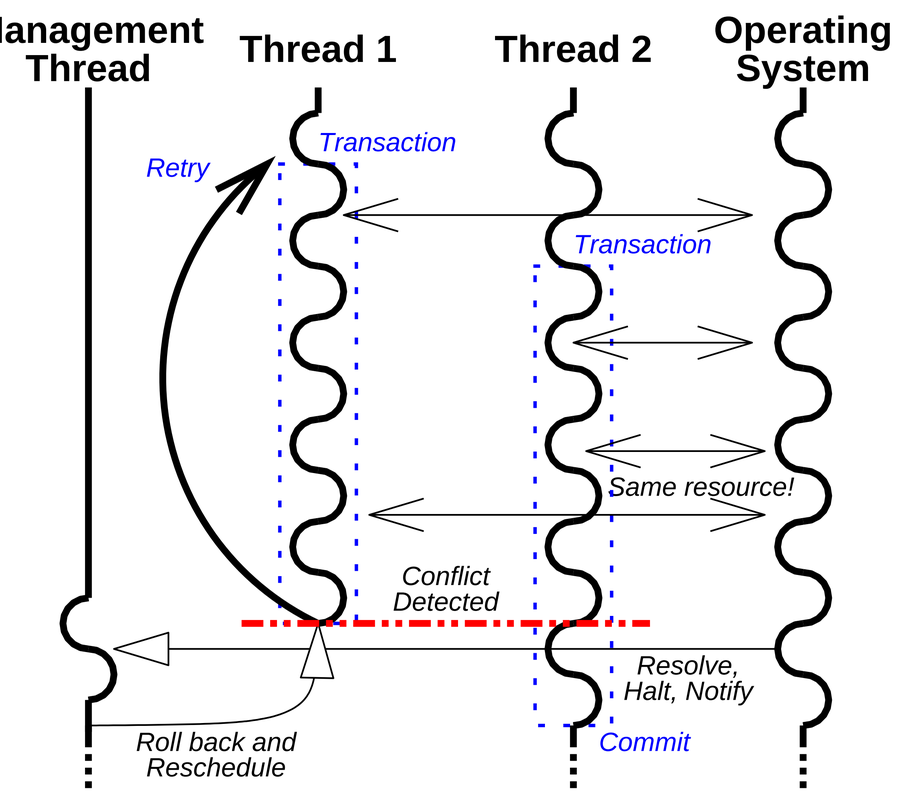


Figure : a per-process thread is responsible for transaction house keeping

## 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. Tocttou vulnerabilities in unix-style file systems: An anatomical study. In *FAST* (2005).