MOONSHOT TRACK · NETDEV 0x1A

Why Syscalls Fail

Bilateral Commit at the Linux Kernel/Userspace Boundary

Manav Singhai
Measurements & Findings

Paul Borrill
Framework & Argument · DÆDÆLUS

Università Roma TRE · 13–16 July 2026

Two speakers, one argument

PB

Paul Borrill

Framework & argument

- The page-cache illusion & FITO
- The Kernel Acknowledgment Spectrum
- Forensic case studies (fsync, io_uring, fork, signals)
- Existing bilateral mechanisms & the proposal

MS

Manav Singhai

Measurements & findings

- Linux 6.8 test harness (E1–E4, R1)
- Quantitative silent-loss rates
- The Bilateral Commit Queue shim
- What the numbers say about the Spectrum

Closing: joint Q&A

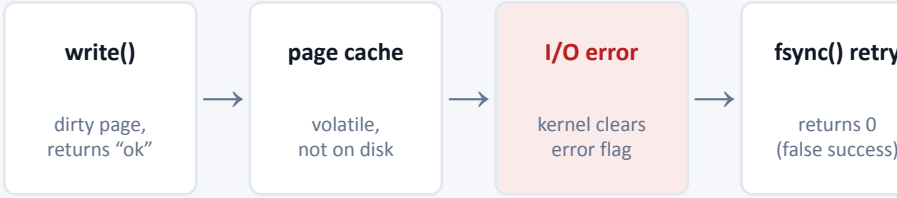
write() is a lie

PB

The kernel copies data into the page cache and reports success.

That is not durability — it's volatile memory.

The caller assumes forward progress. The kernel makes no commitment.



PostgreSQL vs. ext4, 2018

A disk I/O error clears fsync()'s error flag after one report. On retry, fsync() returns success — data is gone, PostgreSQL believes the WAL is durable.

“Written”

calls an operation that has written nothing durable. The interface lies, and the programmer believes the lie.

PB

The same structural hole, four times over

1

Email

SMTP

relay confirms transport, not that the recipient read it

2

Messaging

SMS / iMessage

“Delivered” confirms silicon, not a mind

3

Ethernet

conventional L2

frame transmitted ≠ frame delivered

4

Syscalls

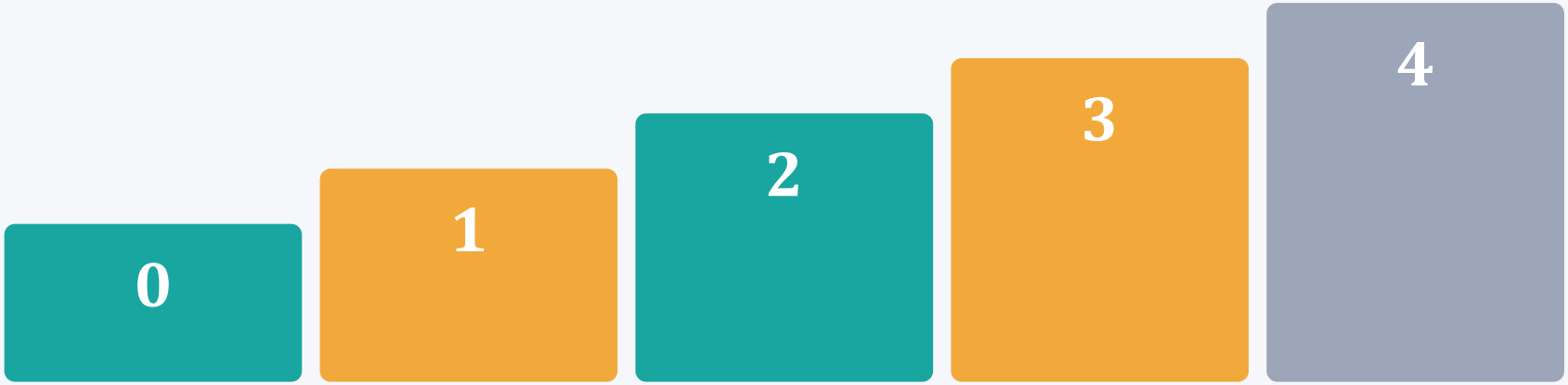
write / send / fork

kernel receipt ≠ kernel commitment

FITO — Forward-In-Time-Only: every one of these assumes that once an operation is initiated, time will carry it forward to completion.

The Kernel Acknowledgment Spectrum

PB



0	1	2	3	4
Fire-and-Forget	**Kernel Receipt**	**Completion Notification**	**Application Consumption**	**Bilateral Semantic Commit**
write() to page cache w/o fsync()	write(), send() return a count, not durability	fsync(), io_uring CQEs	io_uring_cqe_seen(), userfaultfd	no general-purpose syscall reaches this

THEOREM

Most syscalls cap at Level 1

PB

$$L\square_{ax}(F) \leq 1$$

for F ∈ {write, read, send, recv, mmap, fork, brk}

No mechanism reaches Level 4 — the kernel never learns whether the application actually verified the result.

Mechanism	Elevates	Level
fsync(fd)	write() durability	2
io_uring_cqe_seen()	I/O completion consumption	3
userfaultfd	page fault resolution	3
O_SYNC / O_DSYNC	write() durability	2

No bolt-on mechanism reaches Level 4.

Four families, one structural weakness

PB

write() / fsync()

PostgreSQL vs. ext4 (2018): error flag cleared after one report; retry fsync() lies.

io_uring

Completion queue entries can be silently overwritten before userspace reads them.

fork()

OOM killer can end a child before the parent's PID is even schedulable.

Signals

SIGCHLD coalesces: three children exit, one signal arrives. The kernel knows; it doesn't tell.

MS

MEASUREMENTS & FINDINGS

Is the Spectrum measurable on real Linux?

Four reproducers and one remedy, run on Linux 6.8.0 / aarch64

Presented by Manav Singhai

The harness

MS

- **Ubuntu 24.04 / Linux 6.8.0, aarch64**
- Lima/Vz virtualization on Apple M2
- io_uring FEAT_NODROP active
- Each reproducer logs JSON Lines, aggregated by harness/aggregate.py

E1 **fsync() error-clear**
dm-flakey fault injection

E2 **io_uring CQE overrun**
N = 16 → 1024 nops

E3 **SIGCHLD coalescing**
N = 2 → 256 children

E4 **fork()/OOM stale-PID race**
64 MiB cgroup-v2 leaf

R1 **Bilateral Commit Queue shim**
≈100-line liburing wrapper

fsync() lies after the first report

MS

100%

error_writes
(25 trials)

100%

drop_writes
(25 trials)

50/50 trials: the retry fsync() returned 0 while the block was absent from disk — in every single trial.

Read-back used O_DIRECT after drop_caches, so the cached page couldn't mask on-disk loss. No 6.8 mitigation exists — the same bug as 2018.

Level 1 masquerading as Level 2: report-once-then-forget error semantics violate the bilateral contract.

Completions the application never sees

MS

16 of 20 cells
show strictly fewer completions than submitted.

1,008 / 1,024

completions held invisibly in the kernel overflow list at cq=16, N=1024

FEAT_NODROP routes overflow to a kernel-side list instead of dropping it — but the application still has no syscall-level signal that state is accumulating.

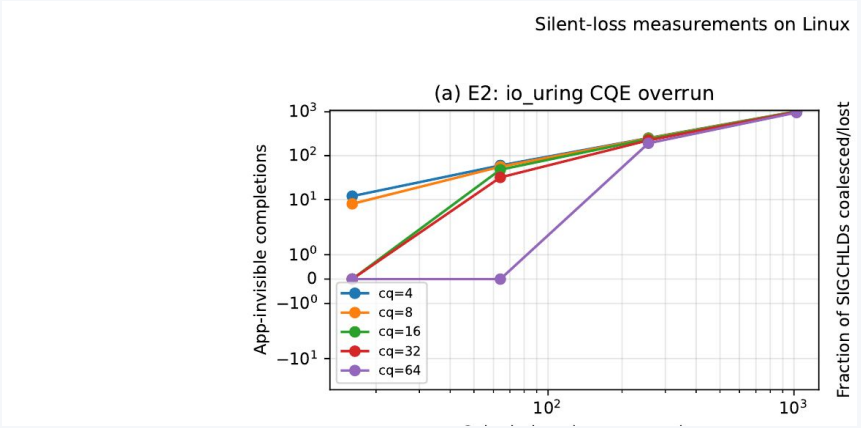


Figure 1(a): app-invisible completions vs. submissions, per CQ size

One signal for three exits

MS

36–153

SIGCHLDs delivered for 256 simultaneous exits (median 130)

0.5–0.8

coalesced-loss fraction at N = 256

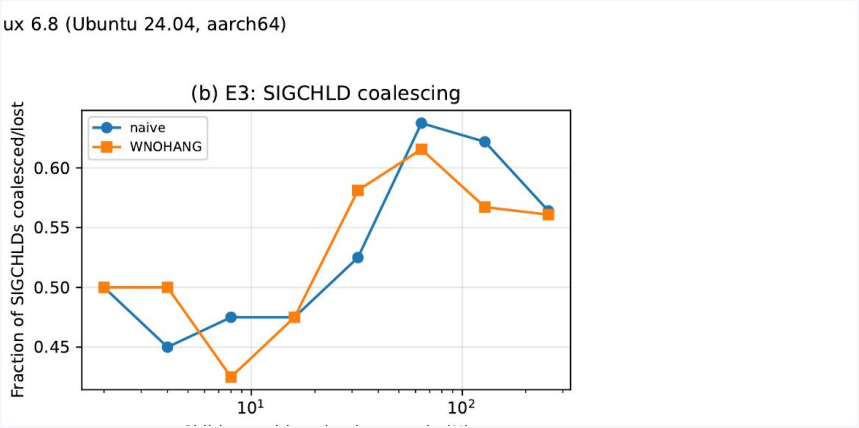


Figure 1(b): fraction of SIGCHLDs coalesced/lost vs. simultaneous exits

A robust WNOHANG loop reaps all children — but that's the application compensating. The kernel still chose to deliver one signal, knowing three exited.

A PID for a process that's already dead

MS

100%

50 / 50 trials

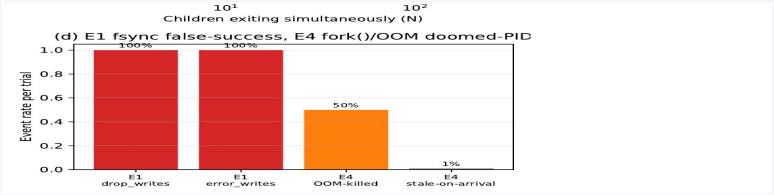
child SIGKILLed by the OOM-killer in a 64 MiB cgroup-v2 leaf

7.7 ms

median fork-to-terminal time — the child never executed a single instruction of useful work.

fork() returns a valid PID as a forward-time projection of viability — the kernel has no protocol to retract it once the OOM killer acts.

Figure 1(d): E1 & E4 event rates



MS

Making silent CQE loss structurally impossible

A ~100-line userspace shim wraps liburing with a per-process Commit Queue: it refuses submissions that would overcommit the CQ, and advances the kernel-visible head only on commq_commit().

Path	Silent loss	Rate	Overhead
Naked (no cqe_seen)	21,280	80% of cells	—
Correct (cqe_seen)	0	0%	baseline
CommQ shim	**0**	**0%**	**+39%**

The shim can't exceed the ring's Level-2 device ceiling, but it raises the application-facing interface to Level 3 — refusing to overcommit, rather than dropping or hiding completions.

FIGURE 1

All four measurements, together

MS

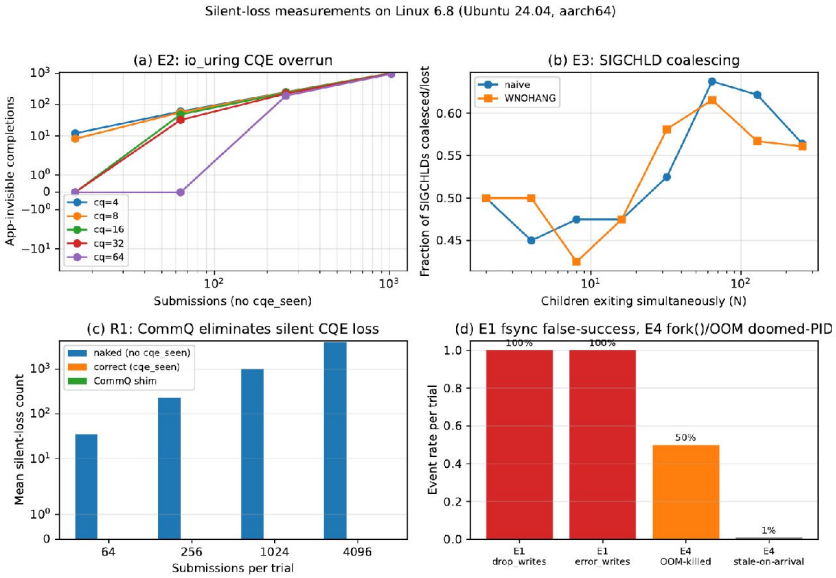


Figure 1: Silent-loss measurements on Linux 6.8 (Ubuntu 24.04 aarch64, Lima/Vz). (a) E2: completions invisible to the application vs. submissions, per CQ size. (b) E3: fraction of SIGCHLDs coalesced vs. simultaneous exits. (c) R1: silent-loss counts for naked, correct, and CommQ paths — CommQ stays at zero. (d) E1 `fsync()` false-success and E4 `fork()`/OOM doomed-PID rates.

MS

The Spectrum isn't theoretical — it's measurable, and current

100%

false fsync() durability across 50 trials

1,008

silently buffered io_uring completions in a single ring

80%

SIGCHLD loss at modest concurrency

100%

doomed-PID forks into a constrained cgroup

On Linux 6.8 / aarch64, over 3–25 trials per cell (E1: 25/mode; E2/R1: 5/cell over 20/16 cells; E3: 5/(N,mode); E4: 50).

The forensic record and the measured rates make bilateral syscall semantics practically necessary — not just theoretically desirable.

The Kernel Cognition Gap

PB

$R(O) \not\Rightarrow A(O)$

R(O): the kernel's delivery of a result (a value in rax, bytes in a buffer, a CQE)
A(O): the application's consumption and validation of it

No kernel mechanism below the application layer can close the gap — A(O) is outside the kernel's system boundary.

The messaging Cognition Gap, one layer down

WhatsApp's "Delivered"
confirms data reached silicon, not a mind.

A syscall return
confirms data reached a register, not application logic.

Who has tried this before?

PB

Mechanism	Approach	Level
seL4 synchronous IPC	kernel blocks sender until receiver calls Recv()	3
POSIX fsync()	bolt-on; error reporting is unilateral (report once, clear)	2
Linux userfaultfd	genuine bilateral handshake for page-fault resolution	3
Windows TxF	kernel-enforced atomic transactions — deprecated in Win 8	4*

The TxF Paradox: the highest-safety mechanism was too complex for humans to use correctly — the API must be simple enough to use correctly, or bilateral semantics become dead code.

PB

A four-phase bilateral protocol

1

Propose

userspace submits;
kernel returns a transaction handle

→

2

Execute

kernel performs the operation;
failure reported via the handle

→

3

Offer

result placed in a buffer,
remains uncommitted

→

4

Commit / Rollback

application validates and signals;
kernel finalizes or reclaims

Pipelined, not doubled: the COMMIT for operation n bundles with the PROPOSE for n+1 — one extra kernel crossing per sequence, not per syscall.

Three paths, increasing ambition

PB

1

O_BILATERAL flag

opt-in bilateral semantics on write()/read() for a given fd. Analogous to O_SYNC — no change for apps that don't opt in.

2

Bilateral io_uring

a Commit Queue (CommQ) alongside SQ/CQ. The kernel doesn't reuse a CQE slot until the app writes a commit/rollback entry.

3

Bilateral microkernel IPC

adopt seL4's synchronous IPC model for critical interactions, with pipelined commit to amortize latency.

Design principle: bilateral by default, opt-out for performance. seL4's synchronous IPC costs ≈ 0.2 µs per round trip on ARM64.

PB

A category mistake the kernel community can still fix

Treating kernel receipt as kernel commitment is the mistake underlying PostgreSQL's lost transactions, io_uring's overwritten completions, and doomed forks into an OOM cgroup.

The cost of fixing it is bounded: our CommQ shim reaches zero silent loss at ~39% overhead — a lower bound on what an in-kernel implementation could do.

The question for Netdev

Does the kernel community recognise this category mistake before the next PostgreSQL-scale incident — or after?

Questions

MS **Manav Singhai**
manavsinghai11@gmail.com

PB **Paul Borrill**
paul@daedaelus.com

Netdev 0x1A · Università Roma TRE · 13–16 July 2026