

We don't know what we're looking for until we find what we're looking for

Isolation issues in the fleet

David Dai · **Shakeel Butt** Meta

LPC 2026 · Linux System Monitoring and Observability MC · Prague, 5 October 2026

Customers complaints

What teams that share hosts with unrelated workloads tell us

“A neighbor started OOMing. Then our container started OOMing too.”

“Whenever one particular neighbor OOMs, we see stalls and hiccups.”

“Our requests timed out. Our own CPU and memory were low. A neighbor was at its memory limit.”

“A neighbor hit its memory limit. Our dead task couldn't even be killed.”

“Hung-task alerts, then our task was moved off the host.”

“Every task on the host was declared lost. The host had plenty of memory.”

“One OOM made the host agent unresponsive. Our storage tasks were moved away, and we lost data.”

These workloads did nothing wrong. Memory limits and cgroups are supposed to stop exactly this. The rest of the talk is about one big reason they don't.

What to look for?

Contention on shared resources

CPU / Memory / IO

No overcommit, dedicated CPUs, low I/O utilization

Kernel Locks

Cross every isolation boundary: cgroups, cpusets, nice levels, cpu.max.

Lock holders stuck in reclaim, **sleeping on nested locks, runnable state, rcu wait**, etc

What else?

How we measure it: runnable_lock_monitor.bt

one lock hold



When does it report?

- At unlock, if a waiter is queued and one off-CPU stretch was 5 ms or more.

What is in a report?

- Lock type, hold time, time slept, time ready but not running.
- Two stacks: where the lock was taken, and where the holder left the CPU.
- CPUs the holder may use, and CPUs on the host.

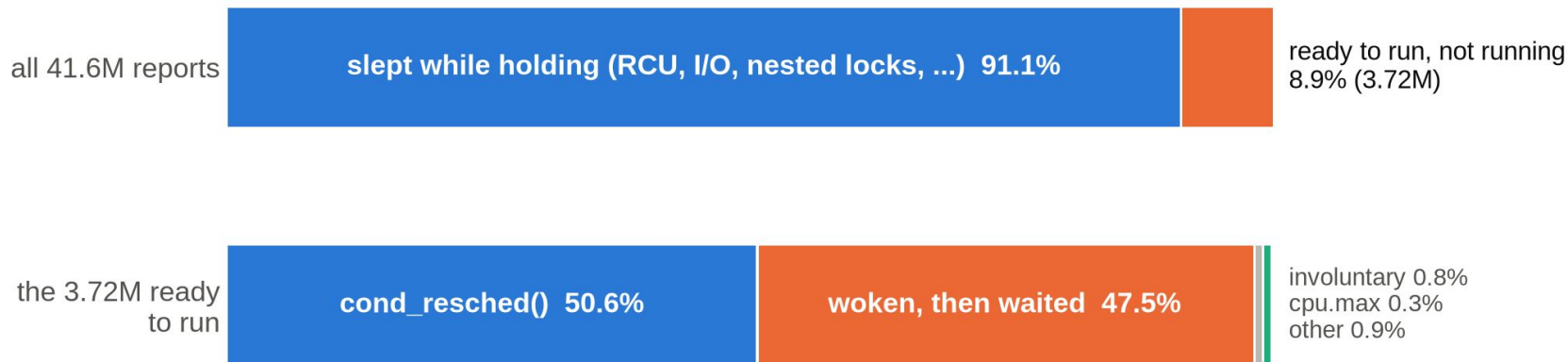
Where it runs

- bpftrace, on mutex, rwsem and percpu-rwsem (write) lock and unlock.
- Strobelight runs it on a rotating sample of hosts: tracking every lock is too costly to run everywhere.

Limits

- Counts are lower bounds, so compare shares.
- Sleeping locks only: no spinlocks, rt_mutex, percpu_down_read or trylocks.

The big picture, and the locks we keep



- 41.6M reports from about 139K hosts in 28 days (28 Aug - 25 Sep 2026). Every one had a waiter queued.
- The 9% where the holder was ready to run but not running (3.72M reports, ~92K hosts) is this talk. **It is mostly cond_resched() and waiting for a CPU after a wakeup, not throttling.**
- We keep only locks shared across workloads: one per host, device, CPU or filesystem. **About 20 locks, 47.5% of the 3.72M.** Per-process and per-file locks are set aside (mmap_lock alone is 41.5%).

Which shared locks, and what they were doing

Reports where the holder was ready to run but not running, 28 days

lock	shared by	reports	what the holder was doing
btrfs b-tree node locks	a filesystem	608K	69% on a tree's root: a container tool's tree search
rtnl_mutex	the host	483K	ethtool ioctl (92%): 348 programs polling NIC stats
rdtgroup_mutex	the host	320K	reading resctrl counters (100%): two monitoring agents
block device invalidate_lock	a device	105K	one storage service's discard ioctl, small cpuset
uprobes_mmap_mutex[]	the host	84K	exec/munmap of binaries with uprobes: health checks
cgroup_mutex	the host	57K	rmdir 32%, cgroup.procs write 32%, mkdir 28%
XFS realtime bitmap lock	a filesystem	45K	inode-GC worker pinned to one CPU, freeing files

Also: cgroup_threadgroup_rwsem 16K, zswap per-CPU mutex 13K, btrfs transaction/log locks 8.1K, wq_pool_attach_mutex 7.8K, kernfs_rwsem 5.1K, bnxt NIC firmware lock 3.7K, lru_add_drain_all lock 2.9K, ...

Isolation stories from the fleet

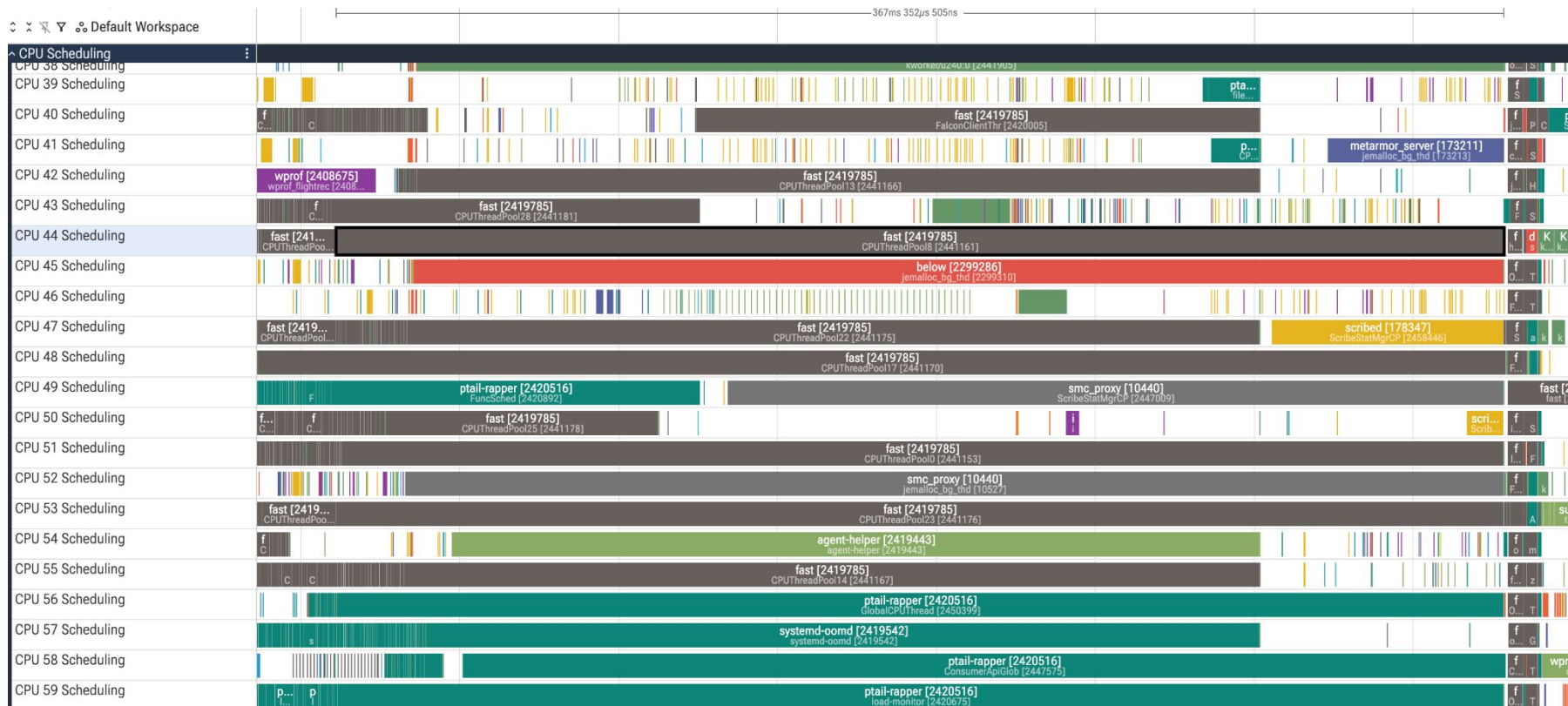
OOM killer prints to the console while holding a lock

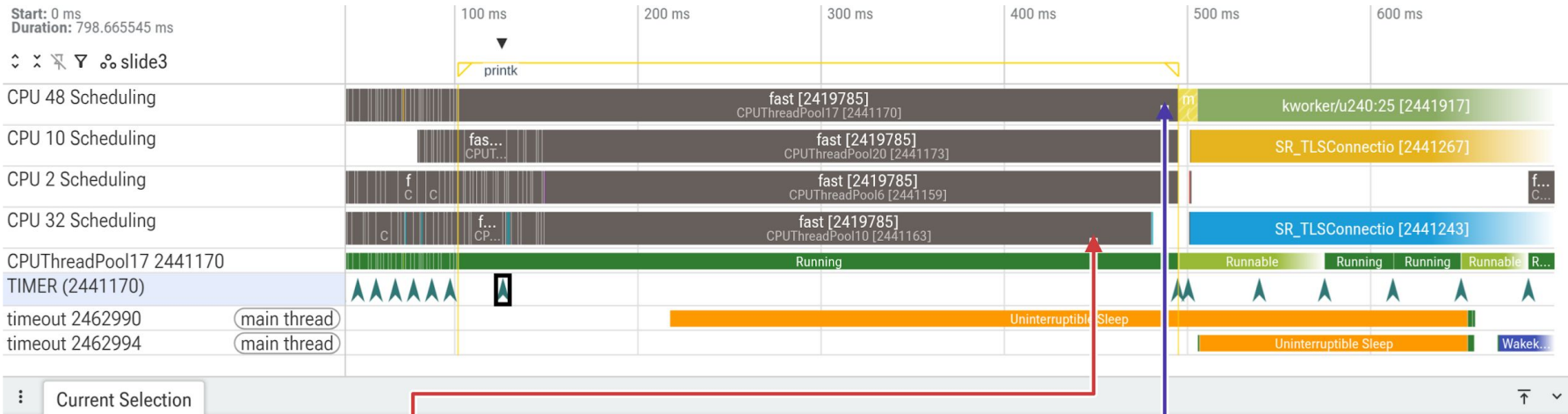
Act of monitoring causes isolation issues in the fleet for sensitive services

Hard partitioning causes lock holder from getting CPU time

Lock waiters $\leftarrow$ Lock holder sleeping on `synchronize_rcu` $\leftarrow$ CPU holding up grace period

1 - OOM killer prints to the console while holding oom_lock





Slice TIMER

```
#35 [K] mem_cgroup_charge (inlined)
#36 [K] __mem_cgroup_charge
#37 [K] charge_memcg (inlined)
#38 [K] try_charge (inlined)
#39 [K] try_charge_memcg (inlined)
#40 [K] mem_cgroup_oom (inlined)
#41 [K] mem_cgroup_out_of_memory
#42 [K] mutex_spin_on_owner
#43 [K] cpu_relax (inlined)
#44 [K] native_pause (inlined)
```

Contextual Options

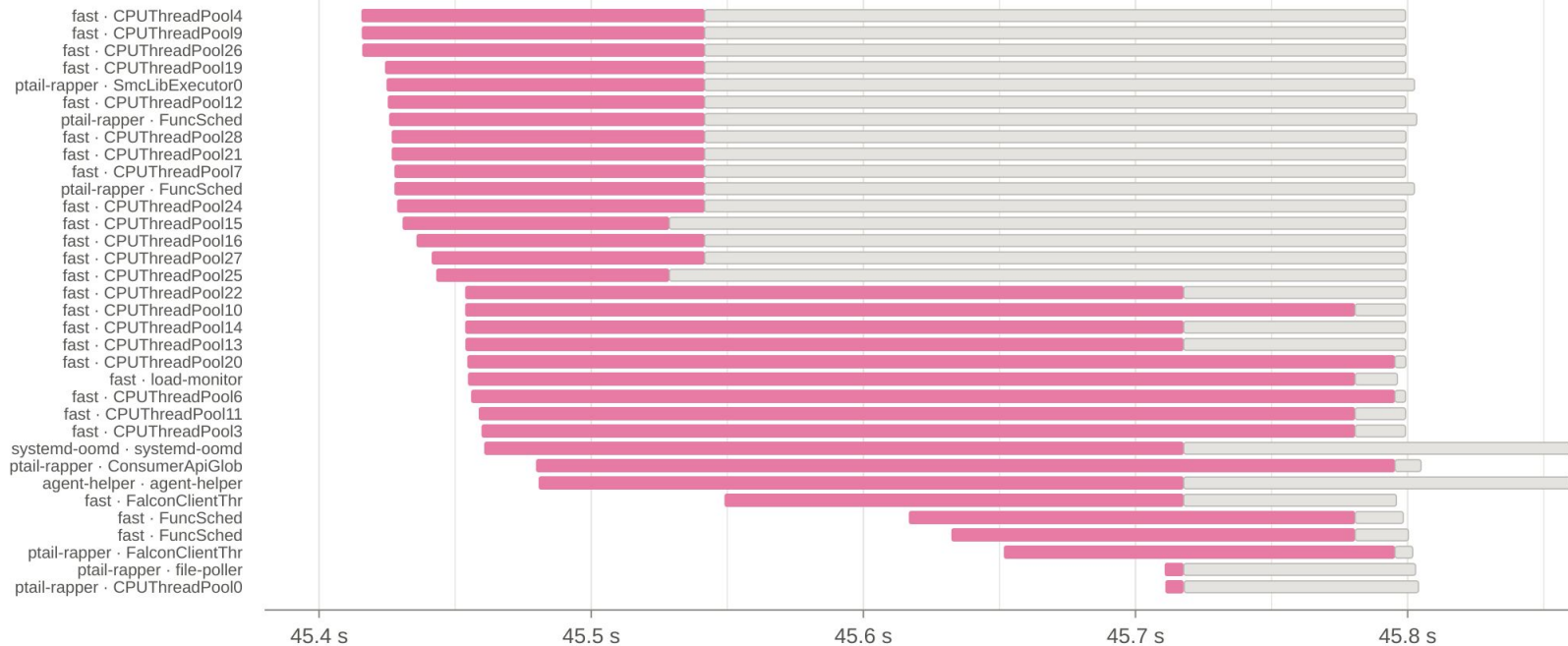
```
#39 [K] mem_cgroup_out_of_memory
#40 [K] out_of_memory
#41 [K] oom_kill_process
#42 [K] dump_header
#43 [K] _printk
#44 [K] vprintk_emit
#45 [K] console_unlock
#46 [K] __console_flush_and_unlock (inlined)
#47 [K] console_flush_all
#48 [K] console_emit_next_record (inlined)
```

■ Holds oom_lock, printing ■ Spinning on oom_lock ■ Sleeping on oom_lock

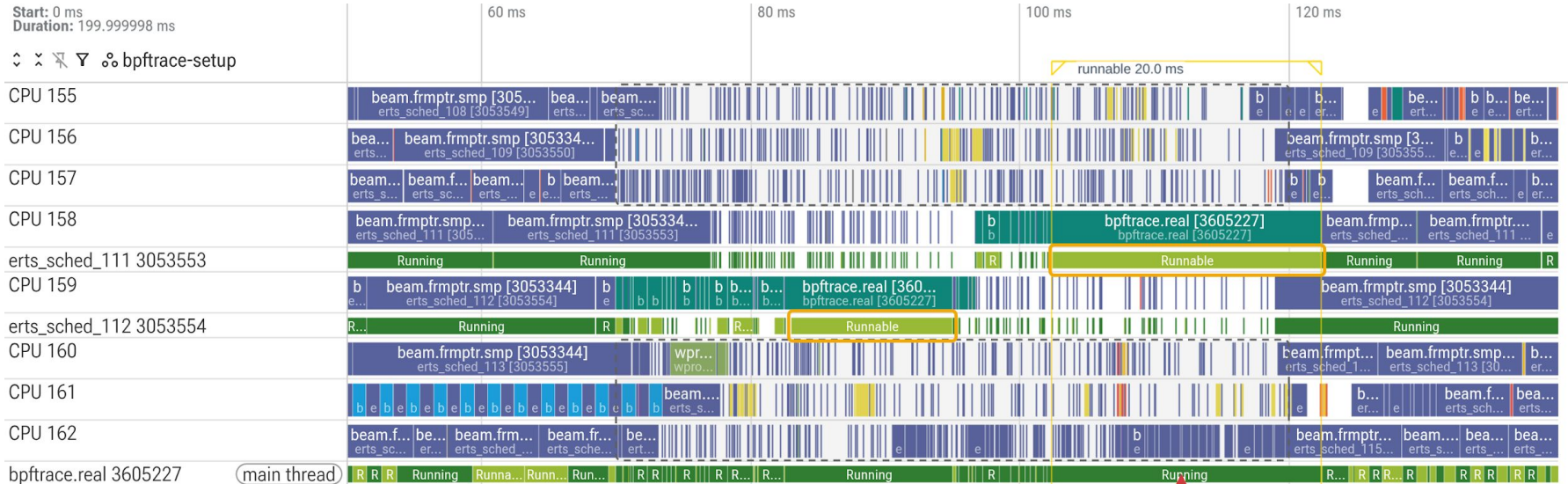
fast · CPUThreadPool17

OOM killer, holds oom_lock

Waiting on oom_lock: 34 tasks



2. Monitoring causes isolation issues in the fleet for sensitive services

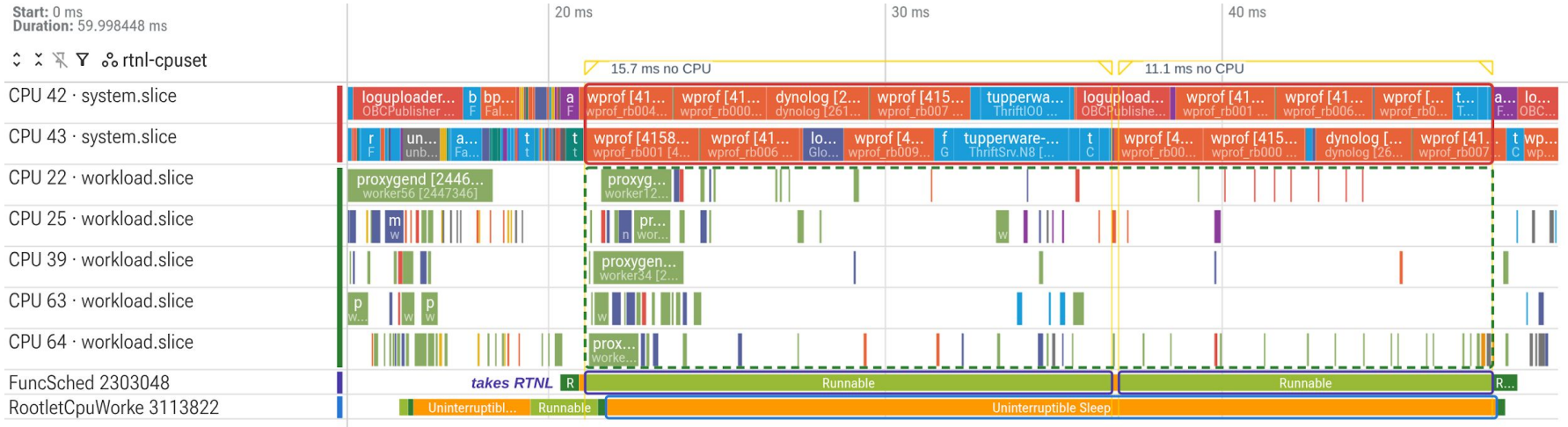


bpftrace.real · holding onto the CPU

erts_sched_111 · Erlang scheduler pinned to CPU 158 · runnable 20.0 ms, runs the instant bpftrace yields

Stalling one Erlang thread can lead to stalls elsewhere

Hard partitioning from causes lock holder unable to get CPU time



system.slice cpuset = CPUs 41–48 (8 of 80), 96% busy for all 26.8 ms

```
/sys/fs/cgroup/system.slice/cpuset.cpus = 41-48 · workload.slice = 0,9-40,49-79 ← ~28 idle workload CPUs are off-limits
```

↑ *FuncSched is runnable, but it needs one of these 8 CPUs*

Lock holder · FuncSched (attestd2) holds rtnl_mutex across an ethtool ioctl

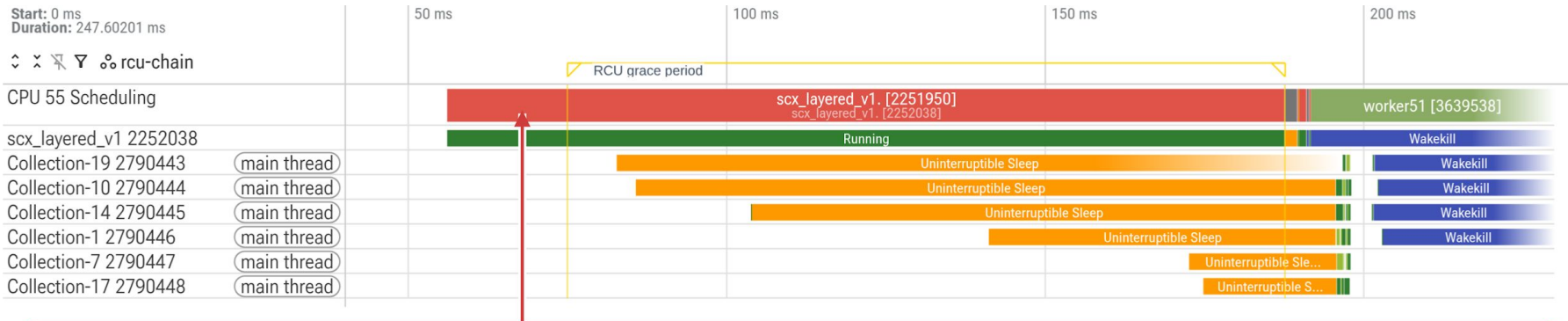
```
dev_ethtool: rtnl_lock() → mlx4_en_get_link_ksettings → mlx4_cmd_wait ← NIC replies in 0.2 ms, then 26.8 ms runnable holding RTNL
```

↑ *RootletCpuWorke is asleep on the `rtnl` mutex that `FuncSched` holds*

Waiter · RootletCpuWorke (rootlet) · 32.2 ms for one netlink request

```
netlink_sendmsg → netlink_rcv_skb → rtnetlink_rcv_msg → rtnl_lock ← asleep until FuncSched's rtnl_unlock at 48.2 ms
```

4 · The RCU chain



scx_layered_v1 · CPU 55

close() → fput_close_sync → inotify_release → fsnotify_destroy_group → fsnotify_clear_marks_by_group ← holds up the RCU grace period

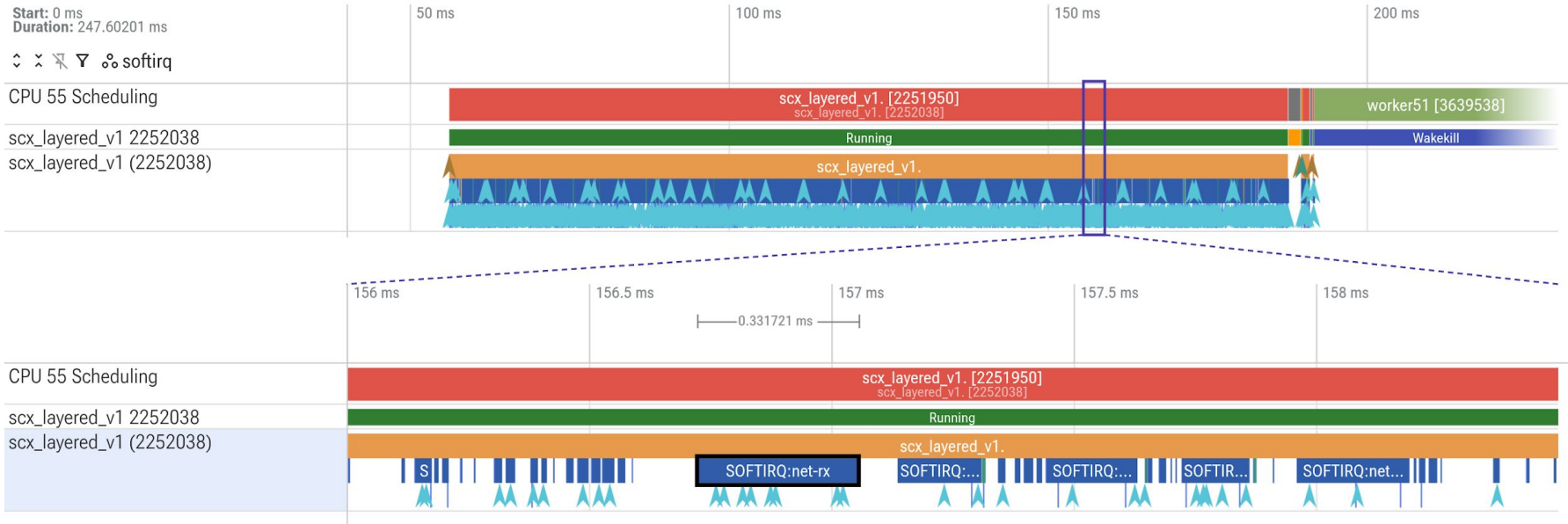
Collection-19

write() → kernfs_fop_write_iter
→ cgroup_procs_write
→ __cgroup_procs_write(threadgroup=true)
→ cgroup_kn_lock_live → cgroup_mutex ← takes cgroup_mutex
→ cgroup_procs_write_start: threadgroup_locked = pid || threadgroup
→ cgroup_attach_lock: cpus_read_lock(); percpu_down_write(&cgroup_threadgroup_rwsem)
→ rcu_sync_enter → synchronize_rcu ← slept here 112.9 ms, holding cgroup_mutex

Collection-10 (109.9 ms) · Collection-14 (91.8 ms) · Collection-1 (54.5 ms) · Collection-7 (23.1 ms) · Collection-17 (20.9 ms)

write() → kernfs_fop_write_iter → cgroup_procs_write → __cgroup_procs_write(threadgroup=true)
→ cgroup_kn_lock_live → cgroup_lock: mutex_lock(&cgroup_mutex) ← blocked here until Collection-19 releases cgroup_mutex

NET_RX softirqs on CPU 55 during scx_layered_v1's run: 3,958 · 48.1 ms of 131.6 ms (37%)



References and Q/A

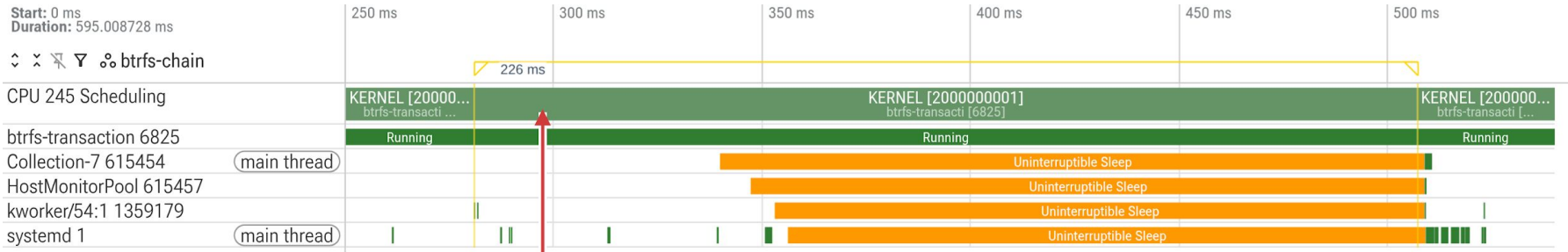
Questions

- Should the kernel expose a cheap “who is everyone waiting for” signal? (holder tracking; proxy execution's blocked_on chain)
- A standard event when a lock holder has been runnable too long, to trigger a flight recorder?
- How do other fleets find these? Is management churn your top offender too?

Tools/References

- [What Blocks This RCU Grace Period?](#) - Paul E. McKenney
- Wprof <https://github.com/anakryiko/wprof> - Andrii Nakryiko

Backup



btrfs-transaction · CPU 245

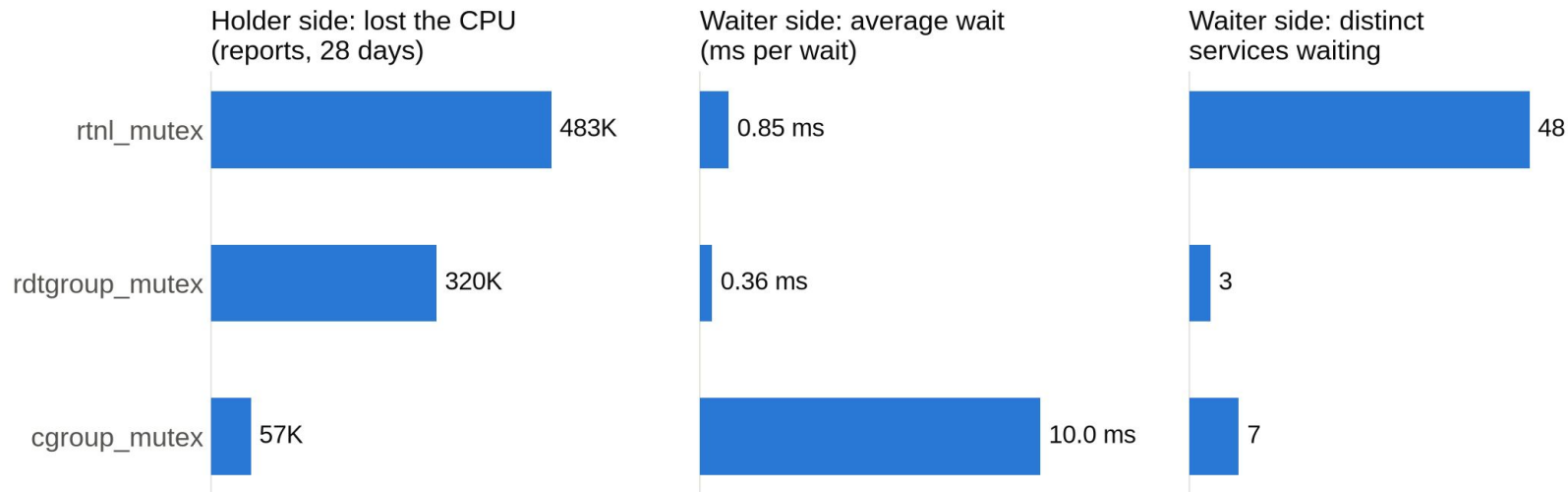
transaction_kthread → btrfs_commit_transaction → btrfs_finish_extent_commit → unpin_extent_range
→ __btrfs_add_free_space → steal_from_bitmap → steal_from_bitmap_to_front → find_next_zero_bit ← holds up the RCU grace period

Collection-7

write() → kernfs_fop_write_iter
→ cgroup_procs_write
→ __cgroup_procs_write(threadgroup=true)
→ cgroup_kn_lock_live → cgroup_mutex ← takes cgroup_mutex
→ cgroup_procs_write_start: threadgroup_locked = pid || threadgroup
→ cgroup_attach_lock: cpus_read_lock(); percpu_down_write(&cgroup_threadgroup_rwsem)
→ rcu_sync_enter → synchronize_rcu ← slept here 169.0 ms, holding cgroup_mutex

HostMonitorPool · 161.5 ms bpf(BPF_PROG_QUERY) → cgroup_bpf_prog_query → cgroup_bpf_query → cgroup_lock
kworker/54:1 · 155.8 ms css_release_work_fn → cgroup_lock ← all three blocked here until Collection-7 releases cgroup_mutex
systemd · 152.8 ms rmdir() → kernfs_iop_rmdir → cgroup_rmdir → cgroup_kn_lock_live → cgroup_lock

The waiter side re-ranks the cost



- **`cgroup_mutex`**: fewest holder reports, longest waits: **10.0 ms on average (max 221 ms)**, 12x `rtnl_mutex`. The holder is asleep in an RCU grace period.
- **`rtnl_mutex`**: the broadest. **48 services** wait on it, from host agents to a database and a cache proxy.
- The waiter side also sees spinlocks the holder side can't: `scx_tasks_lock`, `tasklist_lock`, `css_set_lock`.

Waiter side: Strobelight lockstat, a waiter-side contention profiler (like perf lock contention) on a smaller host sample. Wait times: 2 Aug - 1 Sep 2026.

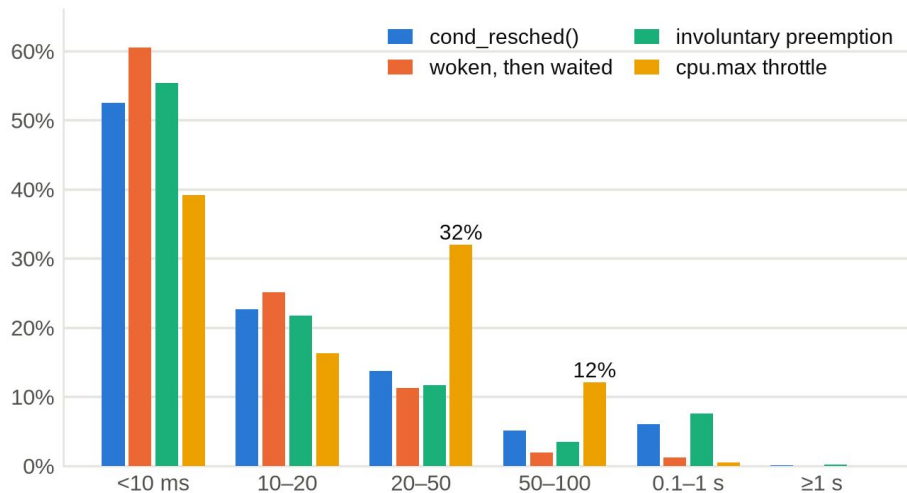
Why these locks: held across a sleep

lock	while holding it, the holder sleeps on	then lost CPU
<code>rtnl_mutex</code>	NIC firmware mailbox round trips (81% of sleeps), <code>vzalloc()</code>	82%
<code>rdtgroup_mutex</code>	a work item on the L3 domain's first CPU (<code>smp_call_on_cpu</code>)	19%
<code>cgroup_mutex</code>	an RCU grace period in <code>percpu_down_write(&cgroup_threadgroup_rwsem)</code>	0.3%
btrfs tree root node	the child node's lock (lock coupling, 94%), disk reads	26%
XFS realtime bitmap lock	a metadata read from disk, on a worker pinned to one CPU	10%

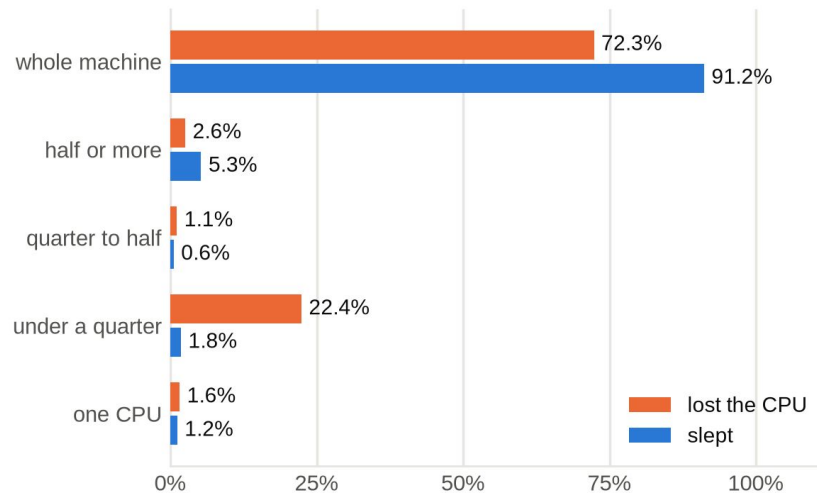
The pattern: a host-wide lock held across a sleep. The sleep ends, the holder is woken, and then it waits for a CPU. For the three global locks, **91-100% of the CPU loss is “woken, then waited”**.

- **`rtnl_mutex`**: in 82% of its reports the holder's longest wait for a CPU was longer than its longest sleep.
- The odd one out: **`kernfs_rwsem`** writers do CPU work and drop the lock to drain. They block only on the nested `kernfs_iattr_rwsem`, so on preemptible kernels they lose the CPU only to preemption.

Why the holder can't get the CPU back



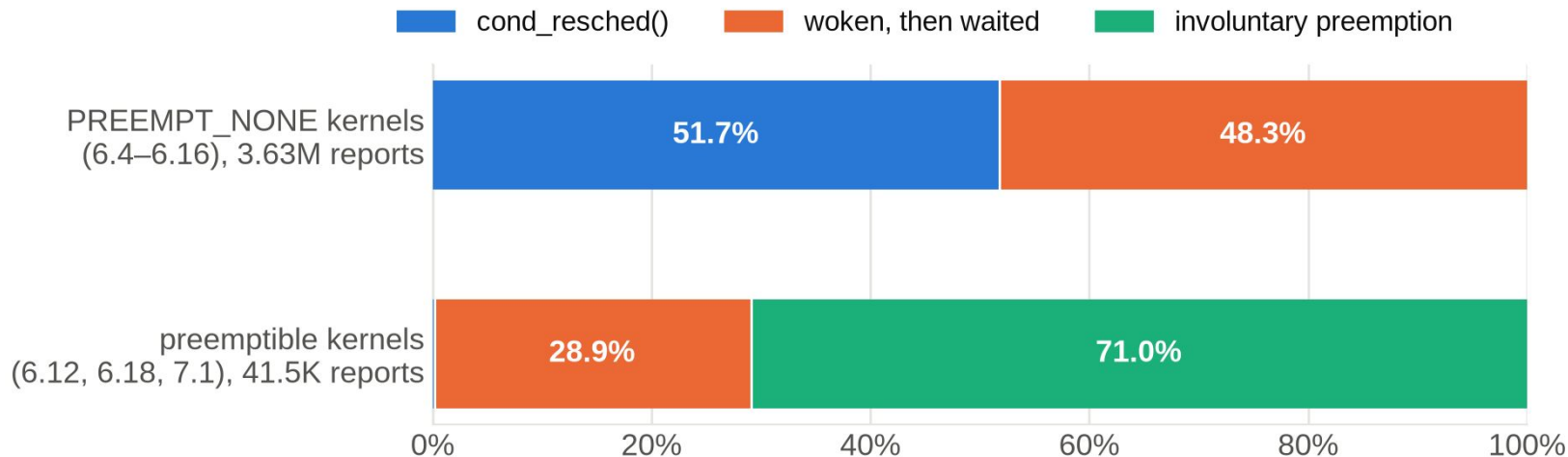
How long the holder waited, by cause (longest wait per report)



How many CPUs the holder was allowed to use

- **Mostly CPU contention, not throttling:** cond_resched() 50.6%, woken then waited 47.5%, involuntary 0.8%, cpu.max 0.3%. Most waits are under 20 ms; 6.0% of cond_resched() yields reach 100 ms.
- **Small cpusets:** 24.0% of holders that lost the CPU could use under a quarter of the machine, vs 3.0% of holders that slept (8x). The kernel pins itself too: the XFS inode-GC worker.
- cpu.max has its own shape: 44% of throttled waits last 20-100 ms, waiting for the next quota period.

The preemption model decides how it shows up



- On a preemptible kernel `cond_resched()` does nothing, so the holder is preempted inside the critical section. **Since v7.0, PREEMPT_NONE and PREEMPT_VOLUNTARY can't be selected on x86 and arm64:** our 7.1 hosts show where the fleet is going.
- Involuntary preemption waits run longer: mean 38 ms, 7.7% at 100 ms or more (wakeup waits: 14 ms, 1.1%).
- `cpu.max`: since v6.18 CFS defers throttling to the return to user space, so a holder is not throttled mid-critical-section. BPF schedulers that enforce `cpu.max` still throttle at enqueue.

One host: catching a stall with wprof

[Perfetto screenshot of a captured lock-holder stall: from David]

holder track: runnable, not running · the CPUs it may use · who ran there, with priority, cgroup and throttle state

- **wprof**: a BPF tracer that writes Perfetto traces: context switches, wakeups, off-CPU and waker stacks, task states, sched_ext events.
- Flight recorder: `wprof -F 10s, 1gb` keeps a rolling window in memory and writes it out on SIGINT.
- The trigger comes from outside: a “holder runnable > X ms with waiters” detector, a hung-task line, a PSI or SLO alarm.
- **It shows what the fleet data can't**: the waiter, and who ran instead.

Three kinds of fixes

	what	examples and status
A	shrink or split the lock	ethtool without <code>rtnl_mutex</code> : in v7.2 for drivers with a per-device lock (mlx5, bnxt, ...). <code>sysfs speed/duplex</code> : queued in <code>net-next</code> ; its commit cites a node-exporter holding RTNL on CPU-throttled hosts.
B	move the sleep out of the lock	<code>favordynmods</code> : on 6.16 it already takes the RCU grace period out of <code>cgroup_mutex</code> (off by default); v6.18 adds a per-process lock. Don't hold <code>rdtgroup_mutex</code> across the cross-CPU counter read. Unbound XFS inode-GC worker.
C	keep the holder on the CPU	CFS defers <code>cpu.max</code> throttling to the return to user space (v6.18). Proxy execution (v6.17): mutexes only, can't be built with <code>sched_ext</code> . No priority inheritance for mutex or rwsem outside <code>PREEMPT_RT</code> .

- **The cheapest fix of all: poll less.** Two agents reading the same `resctrl` counters; NIC stats read through the legacy `ioctl`; a management tool walking `btrfs` trees.
- **C helps every lock at once:** the sleeping-lock version of “don't preempt a lock holder”. Scheduler side: the proxy-execution talk, Sched/RT MC, Wednesday 10:24.

What we can't see yet, and questions

What we can't see

- **Who holds a lock, cheaply.** Every tool re-implements holder tracking, so it runs on a sample. CONFIG_TRACK_LOCK_HOLDERS (RFC, Sep 2026), lock-holder tracepoints.
- **Waits with no holder to blame:** rwsem readers have no owner; a lock handed to a waiter that can't run.
- **Waiters that don't look stuck:** since v6.15 sysfs readers wait for rtnl in S state, so hung-task and D-state tools miss them.
- **One stall end to end:** holder and waiter are seen by different tools on different hosts.
- **Throttling:** no CFS throttle tracepoint; cpu.stat counts the fair class only.
- **A trigger** for flight recorders.

Questions for the room

- Should the kernel expose a cheap “who is everyone waiting for” signal? (holder tracking; proxy execution's blocked_on chain)
- A standard event when a lock holder has been runnable too long, to trigger a flight recorder?
- One throttle event and stat for CFS and BPF schedulers?
- How do other fleets find these? Is management churn your top offender too?

Reclaim side: MM MC, today 18:00 · Proxy execution: Sched/RT MC, Wed 10:24 · Xi Wang, LPC 2024 (priority inheritance for CFS bandwidth)

Every way a holder is off the CPU

cause	reports	share	holds >= 1 s	holds >= 10 s	longest hold
RCU grace period	17,466,155	42.0%	199	1	16.4 s
block I/O	13,232,988	31.8%	3,995	30	36.9 s
nested lock	2,437,798	5.9%	1,514	27	20.6 s
timed sleep / polling	2,171,871	5.2%	99	4	30.5 s
cond_resched()	1,878,342	4.5%	1,989	24	20.4 s
woken, then waited for a CPU	1,765,882	4.2%	179	1	26.2 s
cross-CPU call	1,354,229	3.3%	12	0	2.7 s
other event	529,803	1.3%	206	0	3.3 s
io.cost / writeback	208,109	0.5%	1,034	9	21.5 s
filesystem reservation	200,082	0.5%	8,249	229	64.5 s
workqueue flush	156,007	0.4%	117	0	7.3 s
allocation / reclaim	84,910	0.2%	33	0	9.5 s
userspace filesystem	60,464	0.15%	193	4	36.3 s
involuntary preemption	29,474	0.07%	73	0	9.1 s
cpu.max throttle	10,046	0.02%	13	1	14.2 s

All 41.6M reports, 28 days. Frequency and tail rank in opposite order. The ≥ 10 s tail is mostly “still held when the script stopped” records (length-biased).

All shared locks (28 days)

lock	shared by	reports	hosts	woken, waited	cond_ resched	invol.	< 1/4 CPUs	while sleeping, held across
btrfs b-tree node locks	a filesystem	607,937	33,753	72.5%	26.7%	0.2%	0.3%	nested lock 82%, I/O 12%
rtnl_mutex	the host	483,189	24,776	99.5%	0.1%	0.1%	5.6%	NIC firmware 81%
rdtgroup_mutex	the host	320,352	10,846	100%	0%	0%	0%	cross-CPU call 100%
bdev invalidate_lock	a device	104,501	873	93.3%	5.7%	0%	98.8%	block I/O 100%
uprobes_mmap_mutex[13]	the host	83,984	16,655	1.1%	98.4%	0.5%	10.3%	block I/O, writeback
cgroup_mutex	the host	56,775	20,639	91.4%	8.3%	0.1%	9.3%	RCU grace period 100%
XFS rt bitmap lock	a filesystem	45,111	3,535	55.3%	0.6%	0%	99.8%	metadata read 100%
cgroup_threadgroup_rwsem	the host	15,783	403	99.8%	0%	0.2%	0.1%	RCU grace period 97%
zswap acomp_ctx->mutex	a CPU	13,182	268	0%	0.1%	99.9%	1.9%	(never sleeps)
btrfs trans/log locks	a filesystem	8,120	6,312	37.7%	60.4%	1.2%	1.0%	block I/O 94%
wq_pool_attach_mutex	the host	7,821	2,519	89.8%	0%	5.8%	0%	(rarely sleeps)
kernfs_rwsem	the host	5,137	3,866	94.5%	1.7%	3.2%	3.6%	nested lock 100%
bnxt hwrn_cmd_lock	a NIC	3,677	2,053	99.9%	0%	0%	0.1%	firmware poll 100%
lru_add_drain_all lock	the host	2,878	985	99.7%	0%	0%	20.9%	workqueue flush 100%
s_vfs_rename_mutex	a filesystem	1,998	831	61.4%	37.8%	0.2%	0%	block I/O 84%
text_mutex	the host	1,175	936	0%	99.9%	0.1%	0.5%	-
namespace_sem	the host	826	555	76.2%	3.6%	14.3%	4.1%	nested lock 86%

Total for all shared locks: 1,763,619 reports = 47.5% of the 3.72M. Not counted: mmap_lock (41.5%), per-file and per-inode locks.

Monitor caveats

- **Sampled hosts.** Strobelight runs the script on a rotating subset of hosts, so counts are lower bounds. Compare shares, not totals.
- **Contended holds only.** Production runs in waiters-only mode: every report had a waiter queued at unlock.
- **Sleeping locks only.** Spinlocks, rwlocks, seqlocks, rt_mutex, percpu_down_read, plain semaphores and trylocks are invisible. A lock can show up blocking on something it can't show holding.
- **arm64 stacks are one frame short.** Changes acquire-site names, not lock totals.
- **“Still held” records are length-biased.** Locks still held when the script stops are reported as “held at least this long so far”. Every wait of 5 s or more is one of these; the longest completed wait is 3.5 s.
- **Throttle is not tagged on v6.18+.** Those kernels defer cpu.max throttling to the return to user space; the script turns the tag off there, so their zero is by design.

Per kernel (28 days)

kernel	reports	hosts	throttle	woken, then waited	cond_resched()	involuntary
6.16	19,973,594	120,089	6,911	905,195	1,039,990	0
6.11	16,876,496	23,594	481	812,243	793,059	0
6.4	2,578,033	1,725	112	723	1,088	0
6.13	1,289,586	10,465	2,325	34,833	43,348	0
6.18 (VM hosts)	610,804	11,242	not tagged	450	13	8,505
7.1	133,428	2,475	not tagged	6,752	30	15,667
6.9	83,999	351	163	880	814	0
6.12 (hardened)	73,954	316	54	4,806	0	5,302

PREEMPT_NONE kernels (6.4-6.16): zero involuntary preemption. Preemptible kernels (6.12, 6.18, 7.1): cond_resched() is a no-op, so involuntary preemption takes its place.

mmap_lock: the biggest lock we set aside

- **1.54M reports, 41.5% of the CPU loss.** Per process, so it mostly stalls the holder's own threads.
- 97.8% is `cond_resched()` inside long `mprotect`, `madvise`, `munmap` and fork walks with the lock held.
- 36% of its holders may use under a quarter of the machine.
- It crosses workloads when another process reads `/proc/<pid>/` files that take the target's `mmap_lock` (monitoring agents).
- Proxy execution covers mutexes only, and `rwsem` readers have no owner to boost anyway.

Where each lock sleeps (code paths)

lock	code path at v6.16 (fleet), and what changed upstream
rtnl_mutex	dev_ioctl -> dev_ethtool: rtnl_lock() for every command, then vzalloc() and driver ops. mlx5: mlx5_access_reg -> mlx5_cmd_do -> cmd_exec -> wait_for_completion_timeout (60 s default); even the count call talks to firmware. v7.2: skipped for drivers with a per-device lock.
cgroup_mutex	__cgroup_procs_write -> cgroup_kn_lock_live(cgroup_mutex) -> cgroup_attach_lock -> percpu_down_write(&cgroup_threadgroup_rwsem) -> rcu_sync_enter -> synchronize_rcu when rcu_sync is idle. favordynmods keeps rcu_sync entered: no grace period. rmdir also drains open files under cgroup_mutex.
rdtgroup_mutex	rdtgroup_mondata_show -> rdtgroup_kn_lock_live(cpus_read_lock + rdtgroup_mutex) -> mon_event_read -> smp_call_on_cpu: queue_work_on the domain's first CPU, then wait_for_completion. IPI only if every CPU in the domain is nohz_full; before v6.9 it was always an IPI.
kernfs_rwsem	kernfs_add_one / __kernfs_remove: allocate before down_write, rbtree work under it; kernfs_drain() drops the lock before it waits. Exceptions: nested kernfs_iattr_rwsem; rename allocates; v7.1 adds ilookup()/iput() in removal.
btrfs	btrfs_commit_transaction: tree_log_mutex across metadata writeback and superblock writes with device flushes. btrfs_sync_log: the filesystem-wide log_root_tree->log_mutex across log writeback waits.
net-sysfs	speed_show: since v6.15 blocks in rtnl_lock_interruptible(), an S-state sleep that hung-task and D-state tools don't see. linux-next: per-device lock only, for mlx5-class devices.