

Tackling Isolation Issues Due to Memory Pressure

When one cgroup's memory pressure stalls everyone: lock holders stuck in reclaim

Shakeel Butt Meta

LPC 2026 · Kernel Memory Management MC · Prague, 5 October 2026

My neighbor ran out of memory. Why did I stall?

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

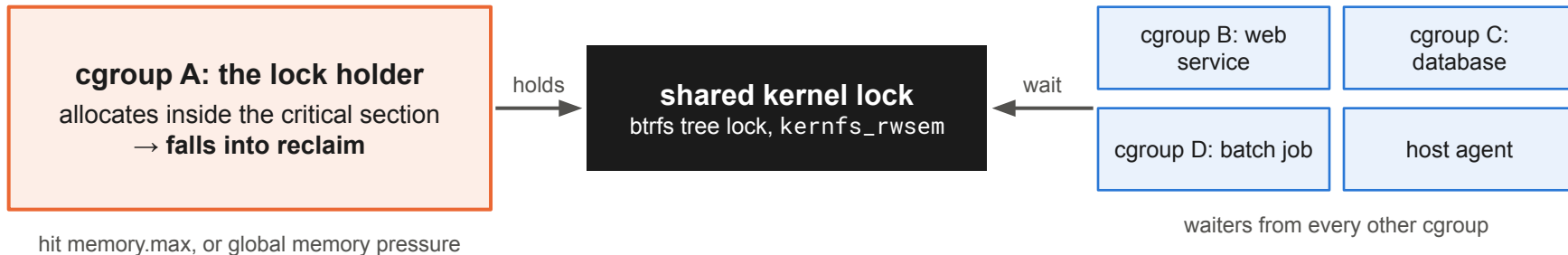
Kernel Locks

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

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

What else?

One lock, every tenant waits



- We isolate workloads with memory limits, CPU limits and cpusets. **But they still share kernel locks.**
- Cgroup limits know nothing about these locks. **A lock is only as fast as its holder:** if the holder stalls in reclaim, one cgroup's memory pressure has leaked to every workload on the host.
- Overcommit makes it worse: less free memory means more allocations reclaim, so more holders stall.
- **A single allocation while holding shared lock can become a source of isolation issue.**

How we measure it



[reclaim_lock_monitor.bt](#)

- bpftrace, on a rotating sample of hosts.
- Hooks lock and unlock of mutexes, rwsems and percpu-rwsems (write), and the reclaim begin/end tracepoints.
- At unlock, if a task is waiting and the hold spanned reclaim: one report per held lock.
- Each report: global or memcg reclaim, time held, time in reclaim, where the lock was taken, where reclaim started.

Page allocation stall warning

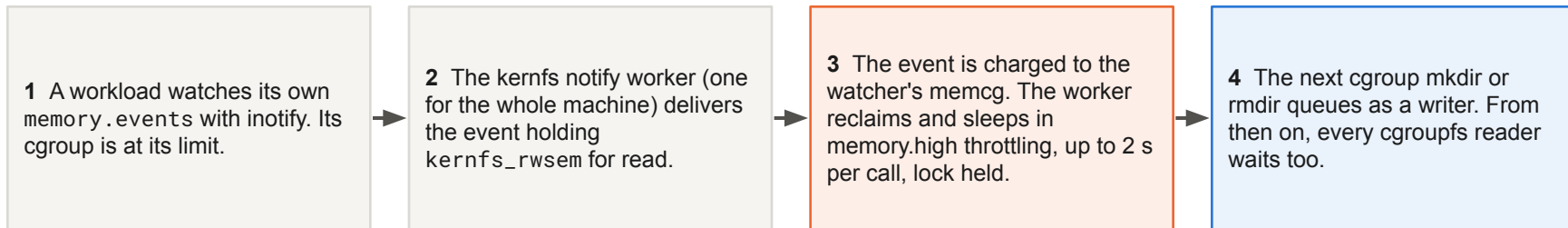
- A local patch: warns once an allocation has been stuck for 10 s, at most once every 10 s per host.
- Upstream had a similar warning until v4.15.
- The one source that sees long stalls, even when nobody is waiting.

Also

- A waiter-side lock contention profiler (like perf lock contention), and hung task warnings (120 s).

Limits: sampled hosts and rate-limited warnings, so every count is a lower bound: compare shares. Sleeping locks only. Most hosts run 6.16-based kernels. The monitor sees short holds (longest finished hold: 5.8 s); long stalls show up only in the warnings.

When memcg does hit a shared lock: kernfs notify



- **In the fleet:** kernfs locks are 0.015% of memcg-reclaim holds but 4.3% of memcg held time: about 220 ms per hold, and 125 holds of 1 s or more. Rare, but each one is long.
- **Reproducer:** an rmdir of one cgroup took 24.6 s.
- **Under global pressure** the same worker gets stuck in the page allocator instead: 3,278 stall warnings a day, spread across the fleet; median 28 s, longest 24 minutes.
- **It nests.** A cgroupfs mkdir or rmdir takes the parent directory's `i_rwsem`, then `cgroup_mutex`, then `kernfs_rwsem`. In 81.8% of hung task reports blocked on `cgroup_mutex`, the owner was stuck in `kernfs_add_one()` or `kernfs_remove_by_name_ns()`. One chain: systemd (pid 1) blocked for 643 s.

Fixing one lock at a time: 18 kernfs patches

What we did

Aug-Sep; all 18 patches are in linux-next

- dentry revalidation without kernfs_rwsem
- no kernfs_rwsem across notify delivery
- open node allocated outside the open-file mutex
- readdir without kernfs_rwsem across `dir_emit()`, plus 3 patches to keep resume safe
- less work under the write lock
- 3 bug fixes, and selftests

Why it doesn't scale

- **18 patches for one lock family.** About 150 locks have holders that go off the CPU; 40 are held across global reclaim.
- **Each fix adds complexity and still leaves cases:** `readdir` needed 3 more patches; the `btrfs` fix still leaves copy-on-write under the lock.
- **Paths stay exposed:** `kernfs_iop_lookup()` allocates an inode under `kernfs_rwsem`; `notify` still holds `kernfs_supers_rwsem` across the same allocation (2.06 s, with the fix); one notify worker for the whole machine (one run: 96 s).
- **New allocations under shared locks keep being added,** and nothing catches them.

Some allocations can't move out of the lock: they need the lock to know what to allocate, they are `__GFP_NOFAIL`, or a user page fault happens inside the critical section.

The lock and the stall belong to different subsystems

Who owns the lock

btrfs · kernfs · networking · cgroup

*MM doesn't know the task
holds a lock others wait for*

Who decides the stall

MM: reclaim it, throttle it

- **The CPU side already solved this once.** Since v6.18, a task out of cpu.max quota is throttled at return to user space, not inside a critical section. It works for every task, without knowing about locks.
- **MM can't do that for every task.** Deferring reclaim for everybody makes global pressure worse and can empty the reserves. So MM needs to know which tasks hold such locks.
- **Proxy execution doesn't cover reclaim:**
 - it only follows mutexes, and 95% of the reclaim cases are rwsems;
 - it does nothing for an owner that is asleep (I/O, memory.high throttling);
 - a holder doing direct reclaim is already running: more CPU doesn't make reclaim finish sooner.

Proposed direction: keep lock holders out of reclaim

Goal: a task that holds one of a few chosen shared locks does not reclaim, and is not throttled, while it holds the lock. The cost is still paid, just not under the lock.

1. **Maintain memory reserves**
2. **Allow lock holders dip into reserves**
3. **Take firm actions when reserves are low**

Proposed direction: keep lock holders out of reclaim

Goal: a task that holds one of a few chosen shared locks does not reclaim, and is not throttled, while it holds the lock. The cost is still paid, just not under the lock.

1. **Maintain memory reserves**
 - Concurrent async reclaimers (with cpu charge to reserve consumers)
 - Increase concurrency of reclaimers as reserves decrease
2. **Allow lock holders dip into reserves**
 - Track holders of opted-in locks
3. **Take firm actions when reserves are low**
 - Throttle non-lock holding allocators/chargers
 - Throttle lock holding allocators/chargers on unlock
 - More aggressive oom-killing

Async reclaimers (low)

Throttle non-lock holders
(min)

More drastic actions

Open Questions

1. Is it OK for MM to treat a task differently because it holds certain locks? Who picks the locks: the subsystem that owns each lock (opt-in), or MM?
2. How much can a lock holder borrow or dip into reserves?
 - a. **Global:** a share of the min reserve, like `__GFP_HIGH`?
 - b. **Memcg:** a fixed debt, or a fraction of the limit?
3. When the bound is used up, is normal reclaim the right fallback?
4. 80% of the btrfs tree-lock holders in global reclaim are kernel threads in the root cgroup.
 - a. Should their borrowing be charged to the cgroups they work for?
5. Do other fleets see this? On which locks?