Kernel Locking Engineering

Daniel Vetter, Intel SATG

ELCE 2023, Prague

Why?

- 10+ years of kernel maintainering in graphics
- lots of drivers, lots of locking rearchitecting
- unfortunately lots of bad examples

Also as Articles ...

- Locking Engineering Principles: https://blog.ffwll.ch/2022/07/locking-engineering.html
- Locking Engineering Hierachy: https://blog.ffwll.ch/2022/08/locking-hierarchy.html

Priorities in Locking Engineering

1. Make it Simple
2. Make it Correct
3. Make it Fast

Make it Correct

- design for lockdep, never against it
- avoid fancy lockdep annotations, simplify instead
- prime locking order when CONFIG_LOCKDEP
- **`might_lock(), might_sleep(), might_alloc(), lockdep_assert_held()`**

Use Correct Code

- don't invent locking/concurrency primitives
- pick the simplest possible locking design
- pick the most powerful primitive, e.g. **`flush_work()`** over completions/waitqueues

Make it Fast

- do you really need faster?
- real workloads, not microbenchmarks
- better architecture is better: Vulkan gpu model, io_uring, ...

Pinciple: Protect Data, not Code

- scales much better in review and testing
- no (subsystem) BKL!
- lockdep encourages protecting code
- beware antipatterns like **`kref_put_mutex()`**

Locking Engineering Hierarchy

Level 0: No Locking

Level 1: Big Dumb Lock

Level 2: Fine-grained Locking

Level 2.5: … because Performance

Level 3: Lockless Tricks

Level 0: No Locking

- Pattern: Immutable State: 1. construct 2. publish
- Pattern: Single Owner: **`queue_work(), completion`**
- Pattern: Reference Counting: **`struct kref`**
- Rust excels at ownership

Level 2: Fine-grained Locking

- Pattern: Object tracking lists
- Pattern: Interrupt Handler
- Pattern: Async processing
- Pattern: Weak references
- … becaus of performance reasons

Locking Antipattern: Object Lifetime vs Data Consistency

- … holding a lock to keep an object alive
- **`kref_put_mutex()`** instead of **`kref_get_unless_zero()`**
- **`flush_work()`** while holding locks
- lockdep does not understand cross-release!
- therefore use most specific existing primitive

Level 3: Lockless Tricks

- Antipattern: RCU
- Antipattern: Atomics
- beware LKMM vs C++ and atomics without **`atomic_`**
- Antipattern: **`preempt/local_irq/bh_disable()`**
- **`local_lock`** as good replacement
- Antipattern: Memory Barriers

Summary/Questions

Principles:

- 1. Dumb 2. Correct 3. Fast
- Protect Data, not Code

Hierarchy

1. No locking
2. Big Dumb Lock
3. Fine-grained Locking
4. Lockless Tricks