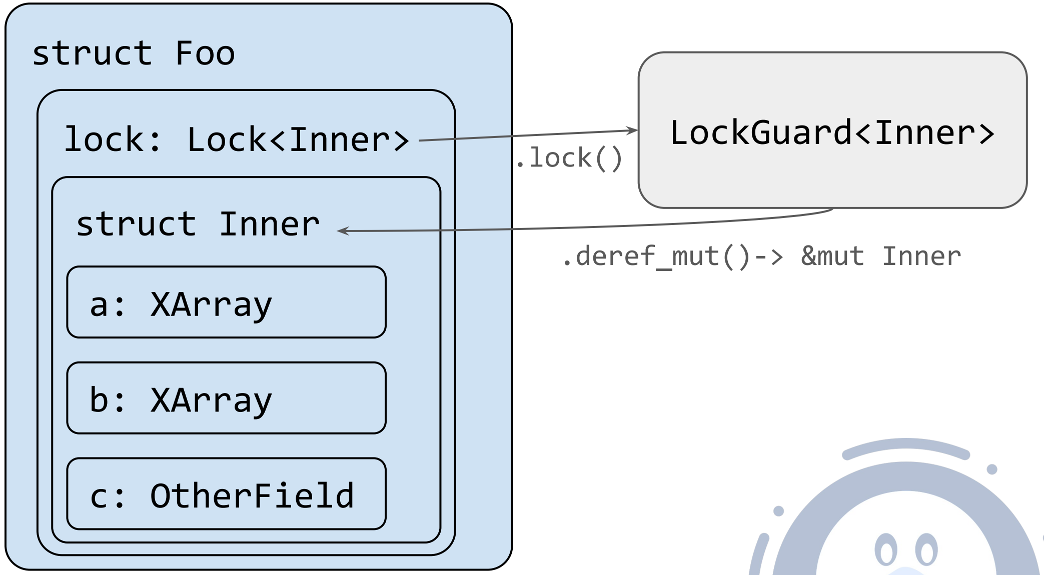
External Locking for XArray

Andreas Hindborg
Linux Plumbers Conference
December 2025

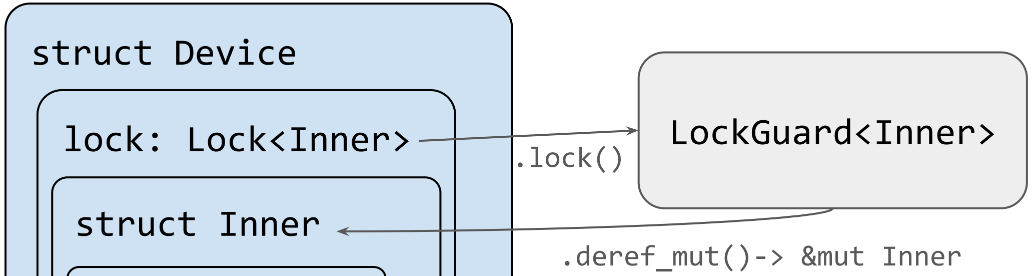
Recap

- Transactions across multiple XArrays require **external** locks.
- XArray uses **internal** locking on C side.
- We are forced to take inner lock -> **penalty**
- But we are guaranteed **exclusive access** as we hold `&mut`!
- XArray will drop internal lock to allocate on insert
  - we allocate **while holding** external lock



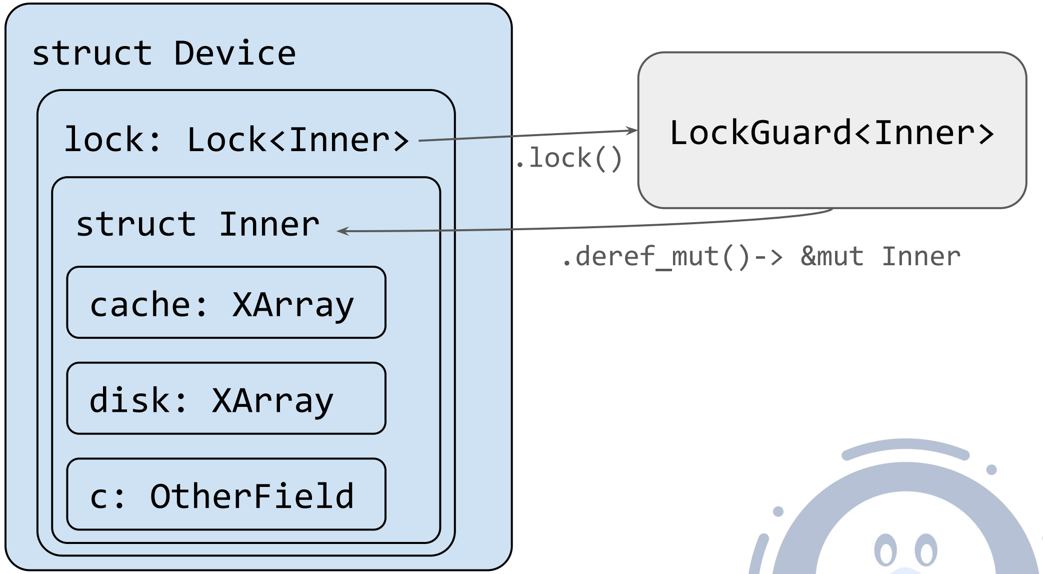
The *problem* in `rnull` (write)

- If cache is full then evict cache page according to **policy**.
- Move evicted page to same index in **other tree**.
- Index of evicted page **cannot** be known in advance.
- Reservation API **cannot** be used.



Summary: 2 distinct *problems*

1. Taking internal lock when we have `&mut` is not required
2. We cannot hold the external lock when inserting due to allocation



Proposal at Kangrejos (RfL Workshop)

- `&mut` access to `XArray` should **bypass** internal C xarray lock (skip Rust `Guard` type).
- `xarray::Guard` should be able to **drop** outer locks prior to allocating.
  - Inject an `Unlocker` token into `xarray::Guard` methods that allow `Guard` to unlock external locks.

Suggestions from Kangrejos

- Gary: Use field projections and arbitrary self types:
  - Add projection `LockGuard<Inner> -> MappedGuard<Inner, XArray>`.
  - Allow `MappedGuard<_, XArray>` to be used as `self` for `XArray`.
  - XArray can `do_unlocked(f: impl Fn())` through `MappedGuard<_, XArray>`
- Alice: Your Entry API **will not work** if you let go of the lock:
  - It **can** work, but ergonomics will **suffer**.
  - Example: VacantEntry may not be vacant when lock is **reacquired**.
    - We can **detect** this.
    - Return an **error variant** in this case and **retry** the entire operation.
    - Guaranteeing forward progress is up to the caller.

Another approach: Preloading API

- Similar to reservation API but we don't need to know the key in advance.
- API available in C for radix_tree but not for xarray.
- In advance, allocate **upper bound** XArray tree nodes in order to insert N leaf nodes.
- Only need to know **number** of leaf nodes to insert.
- We may allocate **too many** nodes, but cost is **amortized** over time.
- Hold on to allocated nodes somewhere (percpu).

Preloading: Benefits

- Better **control** over when we drop locks.
- More simple code due to no retry paths.
- We can do all our allocation at once
    - No need to drop locks **for each tree** we want to insert in.
    - We just need to know how many **leaf nodes** we want to insert.
- Patches on list, please take a look.

Pending tasks

- We still have to take XArray **internal** lock.
- C xarray code checks that xarray lock is held:
  - `lockdep_is_held(&xa->xa_lock)`
  - We don’t need this in rust when we have &mut access to an XArray
- Possible solutions
  - Ask C xarray maintainers to **remove** the check when called from Rust
  - **Rewrite** XArray internals in Rust instead of calling into C library?
    - We are prototyping this on the side
  - New option: **Lie** to lockdep?

Effect of skipping XArray internal lock

```
let start: kernel::time::Instant<kernel::time::Monotonic> =
kernel::time::Instant:: now();

let done = Arc::new(kernel::sync::atomic::Atomic:: new(0u32), GFP_KERNEL)?;

for _ in 0..4 {
   let lock = module.lock. clone();
   let done = done.clone();
   kernel::thread:: kthrad_run(KBox::new(move || {
       for _ in 0..1_000_000 {
           let outer = lock.lock();
           let _x1 = outer.x1. lock();
           let _x2 = outer.x2. lock();
           for _ in 0..100 {
               core::hint:: spin_loop();
           }
       }
       done. add(1, Relaxed);
   }, GFP_KERNEL)?);
}

while done.load(Relaxed) != 4 {
   core::hint:: spin_loop();
}

let end = kernel::time::Instant:: now();
let elapsed = end - start;
pr_info!("Locking benchmark elapsed: {elapsed}\n" );
```

Difference: 4.3(±1.4)%
11 samples, T-distribution, P95
Linux in qemu on my laptop

Thoughts?

- Entry API and Preload API is **on list**