CSE221 Lecture 15

Aronya Baksy

November 2024

1 RCU Usage in the Linux Kernel: 18 years later

1.1 Goals

- Discuss requirements, design and usage of **Read Copy Update** in Linux kernel
- Older kernel versions used **one global lock** which made multi-core performance **poor**
  - More fine-grained locking at the subsystem/submodule level, but can be optimized further based on access patterns (reader vs writer balance)

1.2 RCU Requirements

- Requirements for RCU:
  - useful forward progress for **concurrent readers**, even during updates;
  - low computation and storage overhead;
  - deterministic completion time (e.g.: Non Maskable Interrupts using spinlocks, not deterministic as they involve retries)
- Why not **read-write locks**?
  - R/w locks require overhead of space (counter to maintain count of readers and writers)
  - Atomic writes used by read-write locks are expensive, locking memory buses and enforcing serialization
  - no reading while a thread is writing (goes against RCU goal 1)
- **read-side code** is any code that accesses but not modifies the data str, **write-side code** only modifies data str (e.g. writers might contain both read-side and write-side code)

1.3 How does RCU Work

- **Idea 1**: Do not modify data structures in place, make a copy always
- **Idea 2**: Memory barriers enforce ordering to ensuer that readers do not see inconsistent data
- **Idea 3**: Grace period enforced: readers not allowed to be in an RCU critical section across context switches, and writers must wait until all readers have gone through context swtich before freeing
- Whenever a thread is modifying a shared data structure, all readers are guaranteed to see and traverse either the older or the new structure, therefore avoiding inconsistencies
- Steps taken by a thread to update a shared data structure using RCU:
  - create a new structure,
  - copy the data from the old structure into the new one, and save a pointer to the old structure,
  - modify the new, copied, structure,
  - update the global pointer to refer to the new structure,
  - sleep until the kernel determines that there are no readers left using the old structure, for example, in the Linux kernel, by using `synchronize_rcu()`
  - once awakened by the kernel, deallocate the old structure.

1.4 Linux RCU Design

1.4.1 Readers APIs

- `rcu_read_lock` and `rcu_read_unlock` are used to enter and exit read-side critical sections, enable and disable pre-emption for that reader
- `rcu_dereference(p)`: Like *p, but coordinates with `rcu_assign_pointer`. If *p returns a value assigned by `rcu_assign_pointer`, then later memory accesses (with or without `rcu_dereference`) will see any changes preceding the `rcu_assign_pointer`. Includes memory fences to ensure ordering
- RCU allows threads to wait for the completion of pre-existing RCU critical sections, but it does not provide synchronization among threads that update a data structure (**grace period**).

1.4.2 Writers APIs

- `synchronize_rcu` returns when all RCU critical sections executing at the moment of its calling are finished, wait for all context switches
- `call_rcu(callback, arg)`: wait for all the readers to finish, then run callback function
- `rcu_assign_pointer(p, x)`: "publish". Like $p = x$, but ensures that any read-side code that sees x will observe all prior assignments. Update pointer and include memory fences
- Readers use `rcu_dereference` to signal their intent to read a pointer in a RCU critical section.
- Updaters use `rcu_assign_pointer` to mutate these pointers.

1.5 Disadvantages of RCUs

- Mostly useful only in read-heavy workloads because it makes reader synchronization very light (no atomic ops, no locks, no fences) but writes are heavier (waiting for readers, copy overheads)
- Not clear how to use it for non-linear data structures (e.g. on non-linear data structures like trees)
- Readers might see stale data before update completes
- Readers cannot hold references across context switches or thread sleeps
- Relies on frequent context switches

2 An Analysis of Linux Scalability to Many Cores

2.1 Goals

- Use benchmarks on 7 system apps to show scalability bottlenecks in Linux on multicore systems
- Propose a fix called *sloppy counters* to eliminate bottlenecks from apps (3k lines code changed total)
- Claim: no need to move away from traditional OS kernel organization yet

2.2 Scalability

- **Application Level**: Scalability is limited by the *serial fraction* of a program (Amdahl's Law)
- **OS Level**: Classic considerations in parallel programming:
  - Lock on a shared data structure, more cores implies higher wait time
  - Writing to shared memory, waiting for cache coherency is more when there are more cores
  - More cores implies more misses in shared caches (applies mainly to LLC)
  - Contention over h/w resources like interconnects or DRAM bus
  - More idle cores because of insufficient number of tasks (i.e. insufficient app-level concurrency)

2.3 Techniques for Improving Scalability

- **Multicore packet processing** using h/w hash functions to direct all TCP packets for a particular connection to a fixed core, as well as kernel mods for `accept()` to listen for packets only on that core's packet queue (improves short TCP connection perf)
- **Sloppy counter** represents one logical counter as a single shared central counter and a set of per-core counts of spare references:
  - When a core increments a sloppy counter by V , it first tries to acquire a spare reference by decrementing its per-core counter by V
  - When a core decrements a sloppy counter by V , it releases these references as local spare references, incrementing its per-core counter by V
  - **Invariant**: sum of per-core counters and the number of resources in use equals the value in the shared counter. This yields:

$$C_t = C_{shared} - \sum_p c_p \tag{1}$$

  - Where C_t is true value, C_{shared} is shared value and c_p is local counter of core p.
  - *Occasionally* reconcile the central and per-core counters, for example when deciding whether an object can be de-allocated.
- optimized dentry comparisons using a **lock-free comparison** protocol (increased scalability for name lookups in the directory entry cache)
- Split some **data structures** to be **per-core** instead of global, e.g.: a per-superblock list of open files a table of mount points used during path lookup, and the pool of free packet buffers (avoids lock contention and data contention between cores).
- Optimize cache access patterns to **eliminiate false sharing** (i.e. ensure that variables updated often and read often do not share cache lines)
- Identify and eliminate **unnecessary locking**