Constructing correct RCU data structures

Josh Triplett

October 25, 2013

Locks are (relatively) easy

- Simple mental model
- Hold the lock when touching the data structure
- Make the data structure consistent before dropping the lock

Locks are (relatively) easy

- Simple mental model
- Hold the lock when touching the data structure
- Make the data structure consistent before dropping the lock
- Fine-grained locks, rwlocks: not conceptually harder
  - Note what each lock protects and what order to acquire them
  - lockdep helps

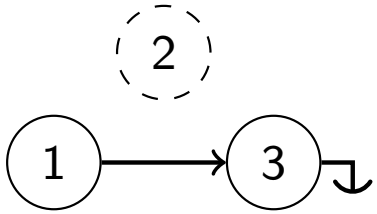
RCU data structures are mostly cargo-culted

- How do you construct new data structures and algorithms?
- How do you review new data structures and algorithms?

RCU data structures are mostly cargo-culted

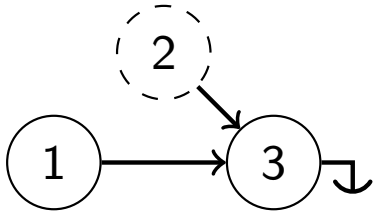
- How do you construct new data structures and algorithms?
- How do you review new data structures and algorithms?
- How much does your data structure look like a linked list?

RCU linked-list insertion



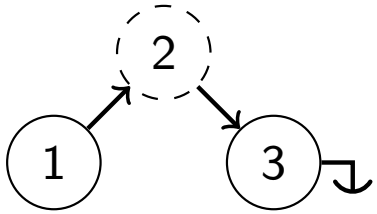
- Initial state of the list; writer wants to insert 2.

RCU linked-list insertion



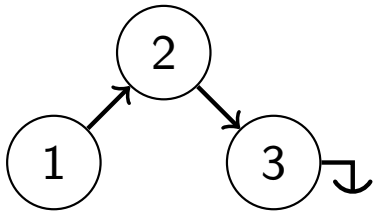
- Initial state of the list; writer wants to insert 2.
- Initialize 2's next pointer to point to 3

RCU linked-list insertion



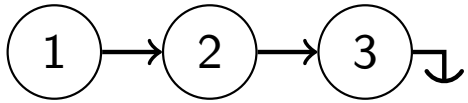
- Initial state of the list; writer wants to insert 2.
- Initialize 2's next pointer to point to 3
- rcu_assign_pointer to publish 2 to node 1's next pointer (includes an smp_wmb())

RCU linked-list insertion



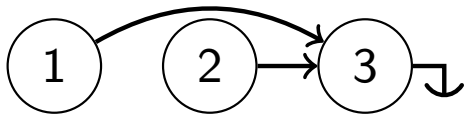
- Initial state of the list; writer wants to insert 2.
- Initialize 2's next pointer to point to 3
- rcu_assign_pointer to publish 2 to node 1's next pointer (includes an smp_wmb())
- Readers can immediately begin observing the new node

RCU linked-list removal



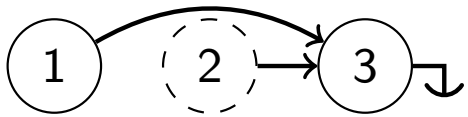
- Initial state of the list; writer wants to remove node 2

RCU linked-list removal



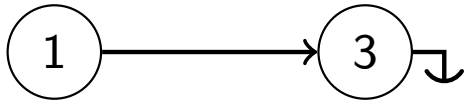
- Initial state of the list; writer wants to remove node 2
- Set 1's next pointer to 3, removing 2 from the list for all future readers

RCU linked-list removal



- Initial state of the list; writer wants to remove node 2
- Set 1's next pointer to 3, removing 2 from the list for all future readers
- synchronize_rcu() to wait for existing readers to finish

RCU linked-list removal



- Initial state of the list; writer wants to remove node 2
- Set 1's next pointer to 3, removing 2 from the list for all future readers
- synchronize_rcu() to wait for existing readers to finish
- Now no readers can hold references to 2, so the writer can safely reclaim it.

How do you generalize this?

- Readers and writers can overlap
- Loads and stores can be reordered
- Hard to reason about barrier and synchronize_rcu() placement

How do you generalize this?

- Readers and writers can overlap
- Loads and stores can be reordered
- Hard to reason about barrier and synchronize_rcu() placement
- Trying to prove a negative
- Construct ordering scenarios, insert barriers, repeat until insane

Simple mental model for new RCU data structures

- Forget about overlap, interleaving, or reordering
- Assume a reader can run atomically between any two stores.

Enforce this model via completely mechanical barrier placement.

Placing barriers

- Between a pair of loads in a reader:
  - Use rcu_dereference() for dependent reads (traversal)
  - Use smp_rmb() for independent reads

Placing barriers

- Between a pair of loads in a reader:
  - Use rcu_dereference() for dependent reads (traversal)
  - Use smp_rmb() for independent reads
- Between ordered writes in a writer, compare write order to reader traversal order:

Placing barriers

- Between a pair of loads in a reader:
  - Use rcu_dereference() for dependent reads (traversal)
  - Use smp_rmb() for independent reads
- Between ordered writes in a writer, compare write order to reader traversal order:
  - If you write in the same order a reader reads, use synchronize_rcu()
  - If you write in the opposite order a reader reads, use smp_wmb() or rcu_assign_pointer()

Examples

- Linked-list insert
- Linked-list removal
- Array resize
- Your data structure (or patch to review) here