Unraveling RCU-Usage Mysteries

(Additional Use Cases)
RCU Usage: Overview

- Quick Review
- You Are Here
- Use Cases:
  - Add-only list, delete-only list, existence guarantee, type-safe memory, light-weight garbage collector, quasi reader-writer lock redux, quasi multi-version concurrency control, and quasi reference count
Quick Review [1]

Quick Review

• Global agreement is expensive
  – Finite speed of light and non-zero-sized atoms...

• So use both spatial & temporal synchronization

• RCU is one way to do this
  – Hazard pointers provide another way
Core RCU API: Temporal vs. Spatial

- `rcu_read_lock()`: Begin reader
- `rcu_read_unlock()`: End reader
- `synchronize_rcu()`: Wait for pre-existing readers
- `call_rcu()`: Invoke function after pre-existing readers complete
- `rcu_dereference()`: Load RCU-protected pointer
- `rcu_dereference_protected()`: Ditto, but update-side locked
- `rcu_assign_pointer()`: Update RCU-protected pointer

For the full Linux-kernel RCU API as of January 2019: https://lwn.net/Articles/777036/
RCU Semantics (Graphical)

Time (really ordering)

rcu_read_lock()

rcu_read_unlock()

synchronize_rcu()

[return]

Free Old Memory

Remove

rcu_read_lock()

synchronize_rcu()

[return]

Free Old Memory

rcu_read_lock()

rcu_read_unlock()

:::

:::

:::

[return]

Free Old Memory

Remove

rcu_read_lock()

synchronize_rcu()

Free Old Memory

rcu_read_unlock()
1. RCU provides ABA protection for update-friendly mechanisms (light-weight garbage collector).
2. RCU provides bounded wait-free read-side primitives for real-time use.

And RCU is most frequently used for linked data structures.
Cost of Global Agreement

Reader-Writer Locking

Time

Reader  Reader  Reader  Reader  Reader  Reader  Reader

Reader  Reader  Reader  Reader  Reader  Reader  Reader

Reader  Reader  Reader  Reader  Reader  Reader  Reader

Reader  Reader  Reader  Reader  Reader  Reader  Reader
RCU vs. Cost of Global Agreement

- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
- Reader
First space/time articulation for RCU (to the best of my knowledge): Jonathan Walpole and his students Josh Triplett and Phil Howard
RCU Spatio-Temporal Values

Reader-Writer Locking

Time

RCU

Old Either Grace Period New

12

Updater 1

Updater 2
You Are Here

- Quasi Reader-Writer Lock
- Quasi Reference Count
- Quasi Multi-Version Consistency Control
- Light-Weight Garbage Collector
- Delete-Only List
- Add-Only List
- Type-Safe Memory
- Existence Guarantee
- Linked Publish/Subscribe
- Phased State Change
- Wait To Finish
Add-Only List
You Are Here: Add-Only List

Quasi Reader-Writer Lock

Quasi Multi-Version Consistency Control

Light-Weight Garbage Collector

Add-Only List

Type-Safe Memory

Existence Guarantee

Phased State Change

Linked Publish/Subscribe

Wait To Finish

Quasi Reference Count

Delete-Only List
First, Add/Delete List

// Reader
rcu_read_lock();
list_for_each_entry_rcu(p, &rl, nxt)
    do_something(p);
rcu_read_unlock();

// Updater
spin_lock(&ml);
p = list_first_entry(&rl, struct foo, nxt);
list_del_rcu(&p->nxt);
list_add_rcu(&q->nxt, &rl);
spin_unlock(&ml);
synchronize_rcu();
kfree(p);
Remove Code For Add-Only List

// Reader
cpu_read_lock();
list_for_each_entry_rcu(p, &rl, nxt, true)
    do_something(p);
cpu_read_unlock();

// Updater
spin_lock(&ml);
p = list_first_entry(&rl, struct foo, nxt);
list_del_rcu(&p->nxt);
list_add_rcu(&q->nxt, &rl);
spinUnlock(&ml);
synchronize_rcu();
kfree(p);
Resulting Code For Add-Only List

// Reader
list_for_each_entry_rcu(p, &rl, nxt, true)
   do_something(p);

// Updater
spin_lock(&ml);
list_add_rcu(&q->nxt, &rl);
spin_unlock(&ml);
Operation of Add-Only List

Key:
- Dangerous for updates: all readers can access
- Safe for updates: inaccessible to all readers

### Operation Steps

1. **Allocate Memory** (`malloc`)
   - `rl` is allocated.
   - `q` is initialized:
     - `->a=?`
     - `->b=?`
     - `->c=?`

2. **Initialize List**
   - `rl` is initialized:
     - `->a=1`
     - `->b=2`
     - `->c=3`

3. **List Add**
   - Call `list_add_rcu`
   - `rl` is added to the list:
     - `->a=1`
     - `->b=2`
     - `->c=3`

4. **Remove Reader**
   - Remove the last reader (sorted by addition order):
     - Call `list_for_each_entry_rcu`
     - Remove the `rl`:

- **Update Reader**
  - Update the last reader:
    - `->a=1`
    - `->b=2`
    - `->c=3`
Synchronization Responsibilities
Synchronization Responsibilities

Initialization visible to readers: `list_for_each_entry_rcu()`

```c
- >next
- >prev
- >lock
```

Other data

Add to list: `ml`
Synchronization Responsibilities

Initialization visible to readers: `list_for_each_entry_rcu()`

For example, if some of that “other data” is mutable.
RCU to Add-Only List

- Add to publish/subscribe for linked structure:
  - Nothing at all!!!
Delete-Only List
You Are Here: Delete-Only List

- Quasi Reader-Writer Lock
- Quasi Reference Count
- Quasi Multi-Version Consistency Control
- Light-Weight Garbage Collector
- Delete-Only List
- Add-Only List
- Type-Safe Memory
- Existence Guarantee
- Linked Publish/Subscribe
- Phased State Change
- Wait To Finish
Again, Start With Add/Delete List

// Reader
rcu_read_lock();
list_for_each_entry_rcu(p, &rl, nxt)
    do_something(p);
rcu_read_unlock();

// Updater
spin_lock(&ml);
p = list_first_entry(&rl, struct foo, nxt);
list_del_rcu(&p->nxt);
list_add_rcu(&q->nxt, &rl);
spin_unlock(&ml);
synchronize_rcu();
kfree(p);
Remove Code For Delete-Only List

// Reader
rcu_read_lock();
list_for_each_entry_rcu(p, &rl, nxt) // Could use READ_ONCE()
do_something(p);
rcu_read_unlock();

// Updater
spin_lock(&ml);
p = list_first_entry(&rl, struct foo, nxt);
list_del_rcu(&p->nxt);
list_add_rcu(&q->nxt, &rl);
spin_unlock(&ml);
synchronize_rcu();
kfree(p);

Why? Maybe you have a system that can remove failing devices, but not add new ones.
Resulting Code For Delete-Only List

// Reader
rcu_read_lock();
list_for_each_entry_rcu(p, &rl, nxt) // Could use READ_ONCE()
do_something(p);
rcu_read_unlock();

// Updater
spin_lock(&ml);
p = list_first_entry(&rl, struct foo, nxt);
list_del_rcu(&p->nxt);
spin_unlock(&ml);
synchronize_rcu();
kfree(p);
Operation of Delete-Only List

Key: ✤ Still dangerous for updates: pre-existing readers can access

One Version
- rl
- cat
- Tux

Two Versions
- rl
- list_del_rcu()
- cat
- Tux

One Version
- rl
- synchronize_rcu()
- cat
- Tux

kfree()
- rl
- Tux

Readers?
- Only old ones!

No readers
Synchronization Responsibilities

Delete from list: m

Prevent compiler from interfering with readers: list_for_each_entry_rcu()

- next
- prev
- lock

Other data

Other data

Other data
Synchronization Responsibilities

Delete from list: ml

Prevent compiler from interfering with readers: list_for_each_entry_rcu()

For example, if some of that "other data" is mutable.
RCU to Delete-Only List

- **Remove** from existence guarantee
  - Publish/subscribe for linked structure
Existence Guarantee
// Reader-then-updater
rcu_read_lock();
q = NULL;
list_for_each_entry_rcu(p, &rl, nxt)
  if (p->key == key) {
    q = p;
    spin_lock(&q->lock);  // RCU provides existence guarantee
    break;
  }
rcu_read_unlock();
if (q) {
  if (!p->deleted)
    do_some_update(p);  // Lock protects *p
  spin_unlock(&q->lock);
}
Code For Existence Guarantee (Lock)

// Updater: List mutation
spin_lock(&ml);
p = list_first_entry(&rl, struct foo, nxt);
spin_lock(&p->lock);
p->deleted = true;
list_del_rcu(&p->nxt);
spin_unlock(&p->lock);
list_add_rcu(&q->nxt, &rl);
spin_unlock(&ml);
synchronize_rcu();
kfree(p);
Synchronization Responsibilities

Ensure readers see initialization and valid pointers: list_for_each_entry_rcu()

Add to or delete from list: ml

- next
- prev

- next
- prev
- lock

Other data
- lock

- next
- prev
- lock

Other data
- lock

- next
- prev
- lock

Other data
- lock
Synchronization Responsibilities

Add to or delete from list: \texttt{ml}

Ensure readers see initialization and valid pointers: \texttt{list\_for\_each\_entry\_rcu()}

The ->lock protects “other data” and prevents the corresponding node from being removed.
RCU to Existence Guarantee

- Add to the combination of wait-for-readers and publish/subscribe for linked structure:
  - Heap allocator
  - Deferred reclamation
Type-Safe Memory
You Are Here: Type-Safe Memory

Quasi Reader-Writer Lock ➔ Quasi Multi-Version Consistency Control ➔ Light-Weight Garbage Collector ➔ Add-Only List ➔ Linked Publish/Subscribe

Quasi Reference Count ➔ Quasi Multi-Version Consistency Control ➔ Light-Weight Garbage Collector ➔ Add-Only List ➔ Linked Publish/Subscribe

Add-Only List ➔ Type-Safe Memory ➔ Phased State Change ➔ Existence Guarantee ➔ Wait To Finish

Delete-Only List ➔ Quasi Multi-Version Consistency Control ➔ Light-Weight Garbage Collector ➔ Add-Only List ➔ Linked Publish/Subscribe

Existence Guarantee ➔ Phased State Change ➔ Linked Publish/Subscribe

Wait To Finish ➔ Phased State Change ➔ Linked Publish/Subscribe
Type-Safe Memory (TSM)

- Can be freed and reallocated, but its type will not change: SLAB_TYPESAFE_BY_RCU
  - Approximation of “real” TSM
- Provides better cache locality because memory can be freed and reallocated immediately
  - No need to wait for a grace period
- But readers need a validation step
kmem_cache_alloc() -> In Use

kmem_cache_free() -> Empty Slab

SLAB_TYPESAFE_BY_RCU Slab cache

New Slab

Free Pages

RCU Grace Period
TSM State Diagram

In Use

kmem_cache_alloc()

SLAB_TYPESAFE_BY_RCU
Slab cache

Empty Slab

RCU Grace Period

Free Pages

New Slab

kmem_cache_free()
Most types of readers need to stop the churn!
TSM Readers Stopping the Churn

- Use a reference counter
- Avoid freed items: `atomic_add_unless()`
- Avoid reallocated items: Recheck key

Working code available at typesafe.2022.02.22a in -rcu tree kernel/rcu/typesafe.c.
struct foo {
    struct list_head lh;
    atomic_t ref;
    int key;
};

static struct kmem_cache *foo_cache;

// Create kmem_cache
foo_cache = kmem_cache_create("foo", sizeof(struct foo),
    sizeof(void *), SLAB_TYPESAFE_BY_RCU, NULL);

// Destroy kmem_cache, which finds your memory leaks! ;-)
kmem_cache_destroy(foo_cache);
Allocate and Initialize

static struct foo *foo_alloc(int key) {
    struct foo *p;

    p = kmem_cache_alloc(foo_cache, GFP_KERNEL);
    if (!p) return NULL;
    p->key = key;
    atomic_set_release(&p->ref, 1); // Implicit ref for data structure
    return p;
}
static struct foo *foo_get_key(int key)
{
    struct foo *p;

    rcu_read_lock();
    p = foo_lookup(key);
    if (!p) {
    } else if (!atomic_add_unless(&p->ref, 1, 0)) {
        p = NULL;
    } else if (p->key != key) {
        foo_put(p);
        p = NULL;
    }
    rcu_read_unlock();
    return p;
}
static void foo_put(struct foo *p) {
    if (atomic_dec_and_test(&p->ref)) {
        // Reader attempting to obtain reference will now fail.
        kmem_cache_free(foo_cache, p);
    }
}
Why Not Just Use Locking???
Why Not Just Use Locking???

- One, `kmem_cache_alloc()` sometimes returns uninitialized memory
  - So initialization cannot tell whether or not to invoke `spin_lock_init()`
- Two, `kmem_cache_zalloc()` clobbers lock
Without `kmem_cache_zalloc()`, "Init" cannot detect allocation from new slab!!!
Do Readers Really Need Atomics???
Do Readers Really Need Atomics???

• Strangely enough, not always!
  - But note that the atomics are per-object, not global

• The lifetime of the typesafe item might be known to be longer than some other object
  - Then a reference to that object stabilizes the item
  - The ext4 filesystem relies on this, to my surprise [1]
  - And thus no atomics for reader validation!

[1] https://lore.kernel.org/lkml/20220209165742.5659-1-quic_qiancai@quicinc.com/ Kudos to Jan Kara
RCU to Type-Safe Memory

• Add to the combination of wait-for-readers and publish/subscribe for linked structure:
  - Slab allocator
  - Deferred slab reclamation
Light-Weight Garbage Collector
You Are Here: Light-Weight GC

- Quasi Reader-Writer Lock
- Quasi Reference Count
- Quasi Multi-Version Consistency Control
- Light-Weight Garbage Collector
- Delete-Only List
- Existence Guarantee
- Add-Only List
- Type-Safe Memory
- Existence Guarantee
- Phased State Change
- Linked Publish/Subscribe
- Wait To Finish
RCU: Lightweight GC for NBS

- Many non-blocking algorithms subject to ABA
  - Where reallocated memory causes failure
  - Example: FIFO single-element push/pop
  - (Single-element push with full-stack pop tolerates ABA-style reallocation)
struct node_t* top;

void list_push(value_t v)
{
    struct node_t *newnode = malloc(sizeof(*newnode));
    struct node_t *oldtop;

    newnode->val = v;
    oldtop = READ_ONCE(top);
    do {
        newnode->next = oldtop;
        oldtop = cmpxchg(&top, newnode->next, newnode);
    } while (newnode->next != oldtop);
}
struct node_t *list_pop(void)
{
    struct node_t *oldp;
    struct node_t *p;

    p = READ_ONCE(top);
    do {
        if (!p)
            return NULL;
        oldp = p;
    } while (p = cmpxchg(&top, oldp, READ_ONCE(oldp->next)));

    return oldp;
}
Initial State

top → cat → Tux
First `list_pop()` is Preempted

Diagram:
- `top`
- `cat`
- `Tux`
- `list_pop() 1`
- `oldp`
- `oldp->next`
Second `list_pop()`
Third `list_pop()`

```
list_pop() 1
oldp
oldp->next
```

Diagram:
- `top`
- `list_pop() 1`
  - `oldp`
  - `oldp->next`
- `cat`
- `Tux`
list_push(dog)
First `list_pop()` Resumes

- **top**
- **dog (was cat)**
- `list_pop() 1`
- `oldp`
- `oldp->next`
- `Tux`
First `list_pop()` Completes

This is the dreaded ABA problem!
First `list_pop()` Completes

This is the dreaded ABA problem! Prevent this by preventing reallocation of cat...
struct node_t *list_pop(void)
{
    struct node_t *oldp;
    struct node_t *p;

    rcu_read_lock();
    p = READ_ONCE(top);
    do {
        if (!p) {
            rcu_read_unlock();
            return NULL;
        }
        oldp = p;
    } while (p = cmpxchg(&top, oldp, READ_ONCE(oldp->next)));
    rcu_read_unlock();
    return oldp;
}

Also need to deferred-free nodes popped from the stack.
RCU to Light-Weight GC

• Add to type-safe memory:
  – Non-blocking synchronization
Quasi Reader-Writer Lock (Redux)
Quasi Reader-Writer Lock (Redux)

Quasi Reader-Writer Lock

Quasi Multi-Version Consistency Control

Light-Weight Garbage Collector

Add-Only List

Type-Safe Memory

Existence Guarantee

Phased State Change

Linked Publish/Subscribe

Quasi Reference Count

Delete-Only List

Wait To Finish
Read-To-Write Upgrade
While traversing list, reader sees need to add or delete a list item

This self-deadlocks with reader-writer locking
  - Deadlocks with special reader-to-writer upgrade primitives, unless they are conditional
    • In which case, reader must handle upgrade failure

What about RCU?
Yet Again, Start With Add/Delete List

// Reader
rcu_read_lock();
list_for_each_entry_rcu(p, &rl, nxt)
  do_something(p);
rcu_read_unlock();

// Updater
spin_lock(&ml);
p = list_first_entry(&rl, struct foo, nxt);
list_del_rcu(&p->nxt);
list_add_rcu(&q->nxt, &rl);
spin_unlock(&ml);
synchronize_rcu();
kfree(p);
Add Locked Deletion Mid-Traversals

// Reader
rcu_read_lock();
list_for_each_entry_rcu(p, &rl, nxt)
    if (p->need_delete) {
        spin_lock(&ml);  // No deadlock with rcu_read_lock()
        if (p->need_delete) {
            p->need_delete = false;
            list_del_rcu(p);  // Leaves list_head ->next pointer alone
            kfree_rcu(p, rh);
        }
        spin_unlock(&ml);
    }
rcu_read_unlock();

// Updater unchanged
Ignore Deleted Item
Ignore Deleted Item

- In some cases, doing something with an already-deleted item is unacceptable
  - Poster child: System V IPC
  - Can’t allow sending a message on deleted mq!

- How can RCU accommodate this situation?
This Time, Start With List Deletion

// Reader
rcu_read_lock();
list_for_each_entry_rcu(p, &rl, nxt)
    do_something(p);
rcu_read_unlock();

// Deleter
spin_lock(&ml);
p = list_first_entry(&rl, struct foo, nxt);
list_del_rcu(&p->nxt);
spin_unlock(&ml);
synchronize_rcu();
kfree(p);
Modifications To Deletion

// Deleter
spin_lock(&ml);
p = list_first_entry(&rl, struct foo, nxt);
spin_lock(&p->lock);
p->deleted = true;
list_del_rcu(&p->nxt);
spin_lock(&p->lock);
spin_unlock(&ml);
synchronize_rcu();
kfree(p);
Modifications To Reader

// Reader
rcu_read_lock();
list_for_each_entry_rcu(p, &rl, nxt) {
    spin_lock(&p->lock); // Lock item, not search structure
    if (!p->deleted)
        do_something(p);
    spin_lock(&p->lock);
}
rcu_read_unlock();
RCU to Quasi Reader-Writer Lock

- Add to existence guarantee:
  - RCU readers as read-held reader-writer lock
  - Spatial as well as temporal synchronization
  - (Optional) Read-to-write upgrade
  - (Optional) Bridge to per-object lock or reference
  - (Optional) Ignore deleted objects

Much of this was covered in the December 7th talk.
Quasi MV Consistency Control
Pathname-Lookup Use Case

• Given a pathname, find corresponding inode
  - Traverse in-memory directory-entry cache
  - Do this locklessly, but if something bad happens, fall back to more heavily synchronized traversal
  - “Something bad” might be a path segment not in the directory-entry cache
  - Or...

Neil Brown LWN series: https://lwn.net/Articles/649115/ https://lwn.net/Articles/649729/ https://lwn.net/Articles/650786/
Pathname Lookup and Renames

Looking up: "/this/pathname/does/not/exist"

- this
- pathname
- does
- exist

- that
- thing
- might
- not
- exist
Looking up: "/this/pathname/does/not/exist"

- this
- pathname
- does
- exist

- that
- thing
- might
- not
- exist
Looking up: “/this/pathname/does/not/exist”
Pathname Lookup and Renames

Looking up: “/this/pathname/does/not/exist”
Pathname Lookup and Renames

Looking up: “/this/pathname/does/not/exist”

Meanwhile: “mv /this/pathname /that”
Pathname Lookup and Renames

Looking up: “/this/pathname/does/not/exist”

Done: “mv /this/pathname /that”
Pathname Lookup and Renames

Looking up: “/this/pathname/does/not/exist”

Meanwhile: “mv /that/thing/might/not /that/pathname/does”
Looking up: “/this/pathname/does/not/exist”

Done: “mv /that/thing/might/not /that/pathname/does”
Pathname Lookup and Renames

Looking up: “/this/pathname/does/not/exist”

- this
- pathname
- does
- exist
- not
- exist
- that
- thing
- might
Pathname Lookup and Renames

Looking up: “/this/pathname/does/not/exist”
Pathname Lookup and Renames

Looking up: “/this/pathname/does/not/exist”

We have looked up a pathname that never existed!!!
How to Avoid This Race Condition?
How to Avoid This Race Condition?

- Use sequence locking in conjunction with RCU
  - RCU makes the lockless traversal safe
  - Sequence locking detects renames
Sequence-Locking Core API

- `read_seqbegin()`: Start reader
- `read_seqretry()`: End reader and check for retry
  - An overlapping seqlock writer will force a retry
- `write_seqlock()`: Start writer
- `write_sequnlock()`: End writer
  - Renames are seqcount writers
Brutally Simplified Pathwalk Code

```c
seq = read_seqbegin(&rename_lock);
rcu_read_lock();

// Traverse the directory-entry cache

if (read_seqretry(&rename_lock, seq))
    goto rename_retry;

rcu_read_unlock(); // Success!
```

Neil Brown LWN series: https://lwn.net/Articles/649115/ https://lwn.net/Articles/649729/ https://lwn.net/Articles/650786/
Pathname Lookup and Renames

Looking up: “/this/pathname/does/not/exist”

We did two renames during the pathname lookup, so ...
Pathname Lookup and Renames

Looking up: “/this/pathname/does/not/exist”

... those renames invalidate the pathname lookup!!!
Restore Consistency To RCU Readers

- RCU makes traversal safe
- Seqlock rejects inconsistent traversals
- This simply identifies a version
  - More complex schemes can allow concurrent traversals of different versions

RCU to Quasi MVCC

• Add to existence guarantee:
  – Readers include some sort of snapshot operation
  – Constraints on readers and writers:
    • Single object,
    • Sequence locks,
    • Version number(s),
    • Issaquah challenge, ...
Quasi Reference Count
Quasi Reference Count

- **Per-item reference count:**
  - `rcu_dereference()` obtains reference limited to the enclosing RCU read-side critical section

- **Bulk reference count:**
  - `rcu_read_lock()` obtains reference on all RCU-protected objects in the system, again limited to the enclosing RCU read-side critical section
Quasi Reference Count (Code)

- You have already seen it!
  - Many of the earlier examples can be interpreted as quasi reference counting
Quasi Reference Count (Code)

- You have already seen it!
  - Many of the earlier examples can be interpreted as quasi reference counting

- How can the same code be existence locking, quasi reader-writer locking, ... ???
Quasi Reference Count (Code)

• You have already seen it!
  - Many of the earlier examples can be interpreted as quasi reference counting

• How can the same code be existence locking, quasi reader-writer locking, ... ???

• What does atomic_inc() do?
Quasi Reference Count (Code)

• You have already seen it!
  – Many of the earlier examples can be interpreted as quasi reference counting

• How can the same code be existence locking, quasi reader-writer locking, ... ???

• What does atomic_inc() do?
  – Lots of things!!!
Quasi Reference Count (Code)

• You have already seen it!
  – Many of the earlier examples can be interpreted as quasi reference counting

• How can the same code be existence locking, quasi reader-writer locking, … ???

• What does atomic_inc() do?
  – Lots of things!!! Just like RCU!
RCU to Quasi Reference Count

• Add to existence guarantee:
  – RCU readers as individual or bulk unconditional reference-count acquisitions
  – (Optional) Bridge to per-object lock or reference
You Are Here
You Are Here

Quasi Reader-Writer Lock

Quasi Multi-Version Consistency Control

Light-Weight Garbage Collector

Add-Only List

Type-Safe Memory

Existence Guarantee

Phased State Change

Linked Publish/Subscribe

Quasi Reference Count

Delete-Only List

Wait To Finish
1. RCU provides ABA protection for update-friendly mechanisms
2. RCU provides bounded wait-free read-side primitives for real-time use
Summary

- RCU synchronizes in space as well as time
  - But the time and space aspects are deeply intertwined
  - Enables near-zero-cost read-side synchronization

- Several additional example RCU use cases:
  - Add-only list, delete-only list, existence guarantee, type-safe memory, light-weight garbage collector, quasi reader-writer lock redux, quasi multi-version concurrency control, and quasi reference count
Summary

• RCU synchronizes in space as well as time
  – But the time and space aspects are deeply intertwined
  – Enables near-zero-cost read-side synchronization

• Several additional example RCU use cases:
  – Add-only list, delete-only list, existence guarantee, type-safe memory,
    light-weight garbage collector, quasi reader-writer lock redux, quasi
    multi-version concurrency control, and quasi reference count

• RCU’s dirty little secret:
  – RCU is dead simple
Summary

• RCU synchronizes in space as well as time
  - But the time and space aspects are deeply intertwined
  - Enables near-zero-cost read-side synchronization

• Several additional example RCU use cases:
  - Add-only list, delete-only list, existence guarantee, type-safe memory, light-weight garbage collector, quasi reader-writer lock redux, quasi multi-version concurrency control, and quasi reference count

• RCU’s dirty little secret:
  - RCU is dead simple, but in order to make good used of it, you must change the way that you think about your problem
Summary

- “I hear and I forget.”
- “I see and I remember.”
- “I do and I understand.”
- To really understand RCU, play with it.
We Are Here And Done!!!
For More Information

- “RCU Usage In the Linux Kernel: One Decade Later”:
- “Structured Deferral: Synchronization via Procrastination”: http://doi.acm.org/10.1145/2488364.2488549
- Linux-kernel RCU API, 2019 Edition: https://lwn.net/Articles/777036/
- “Stupid RCU Tricks: So you want to torture RCU?”: https://paulmck.livejournal.com/61432.html
- Documentation/RCU/* in kernel source
- Folly-library RCU implementation (also C-language user-space RCU)
- Large piles of information: http://www.rdrop.com/~paulmck/RCU/