Paul E. McKenney, Meta Platforms

Linux Foundation Live Mentorship Series, February 23, 2022

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]

[1] https://www.linuxfoundation.org/webinars/unraveling-rcu-usage-mysteries/

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

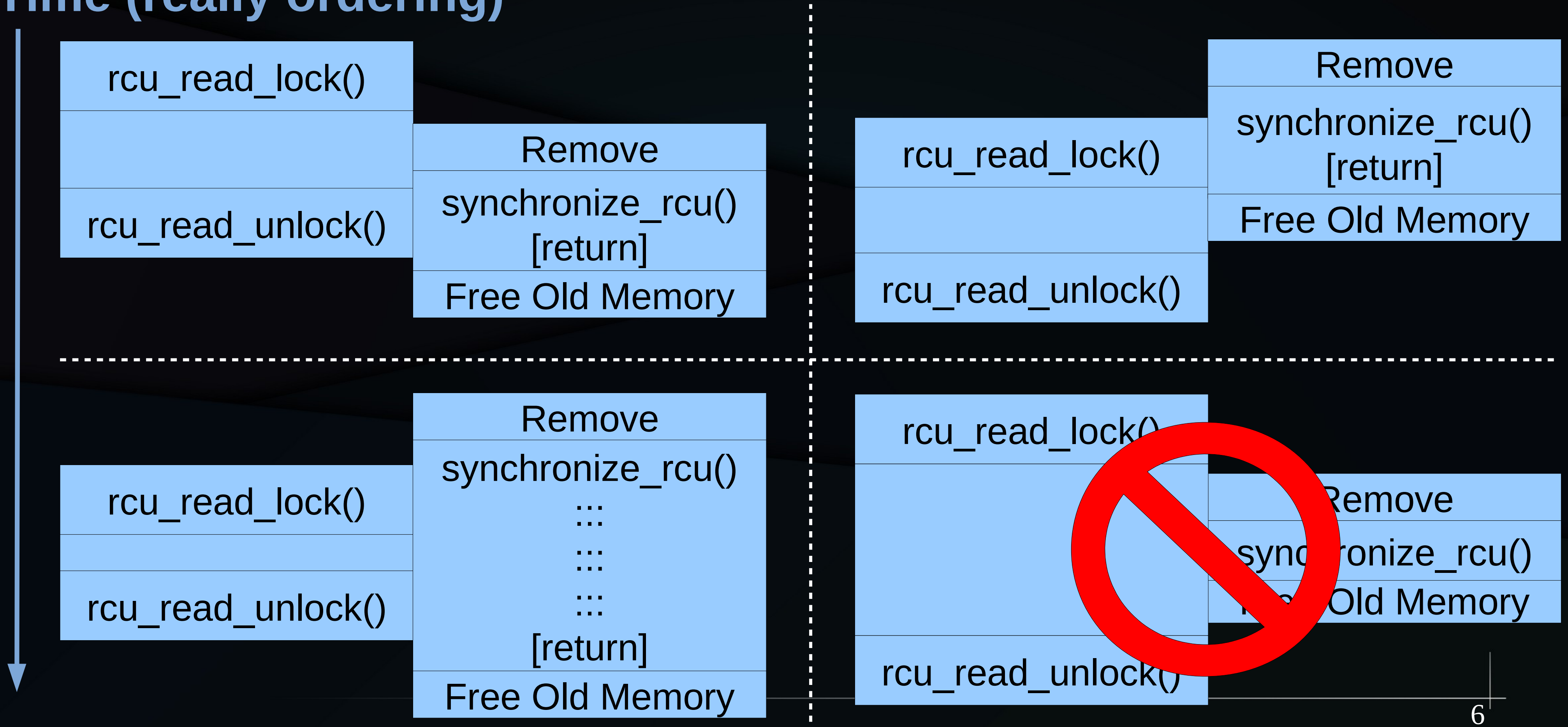
Core RCU API: Temporal vs. Spatial

- `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

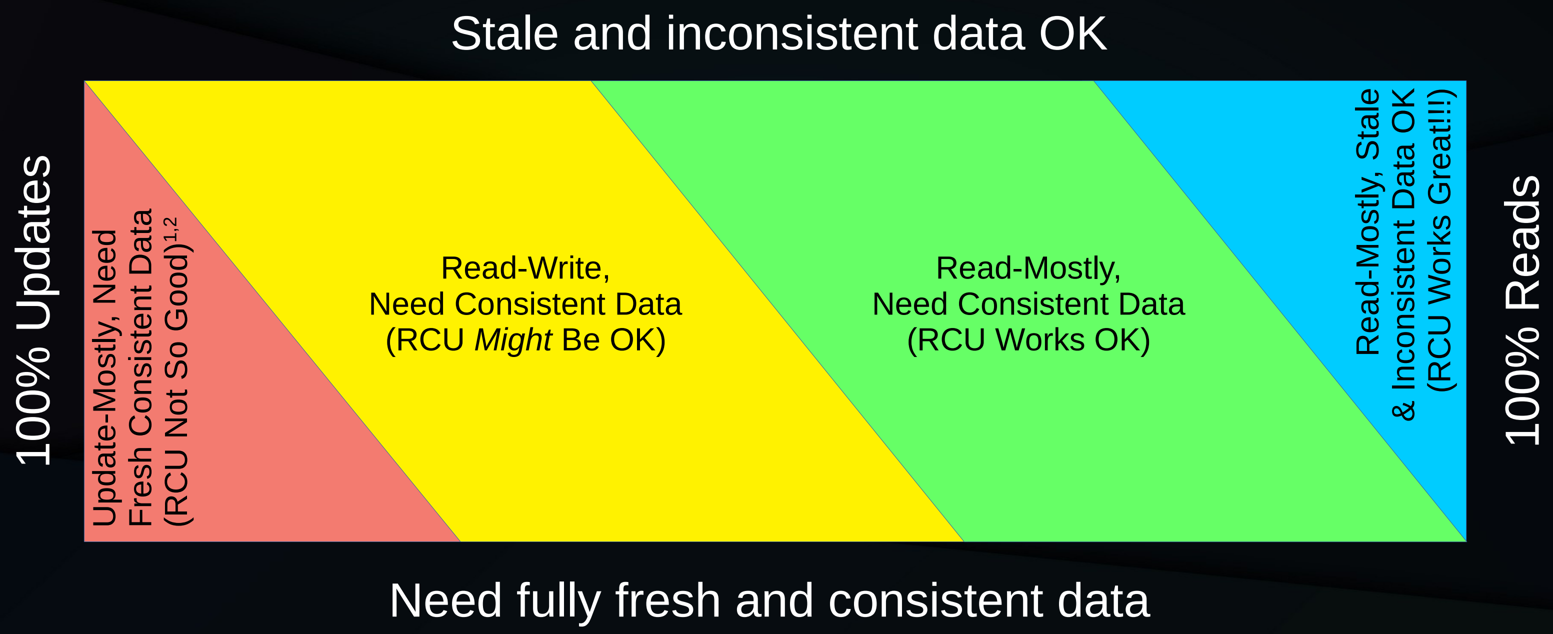
For the full Linux-kernel RCU API as of January 2019: https://lwn.net/Articles/777036/

RCU Semantics (Graphical)

Time (really ordering)



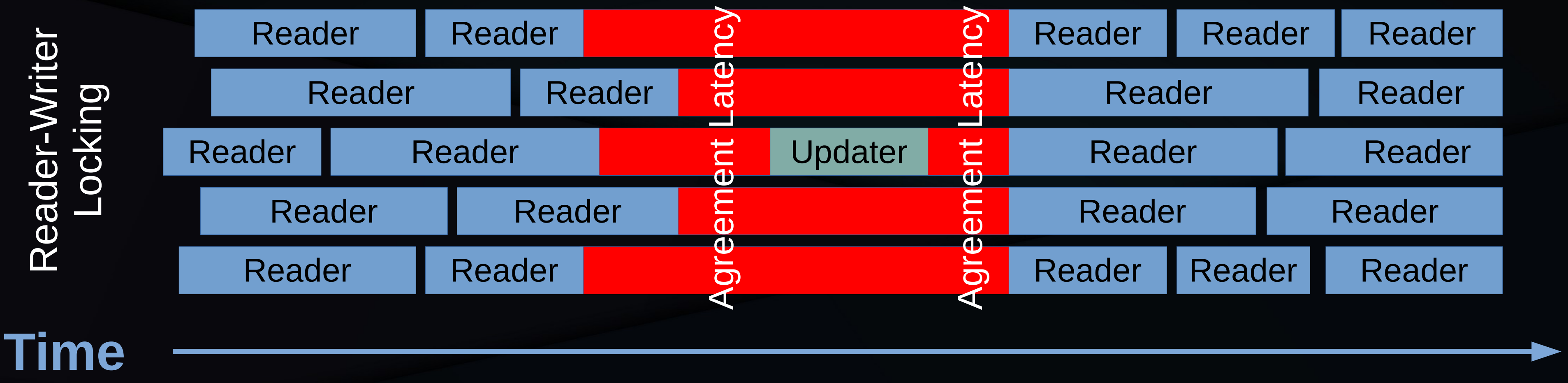
RCU Semantics (Restrictions)



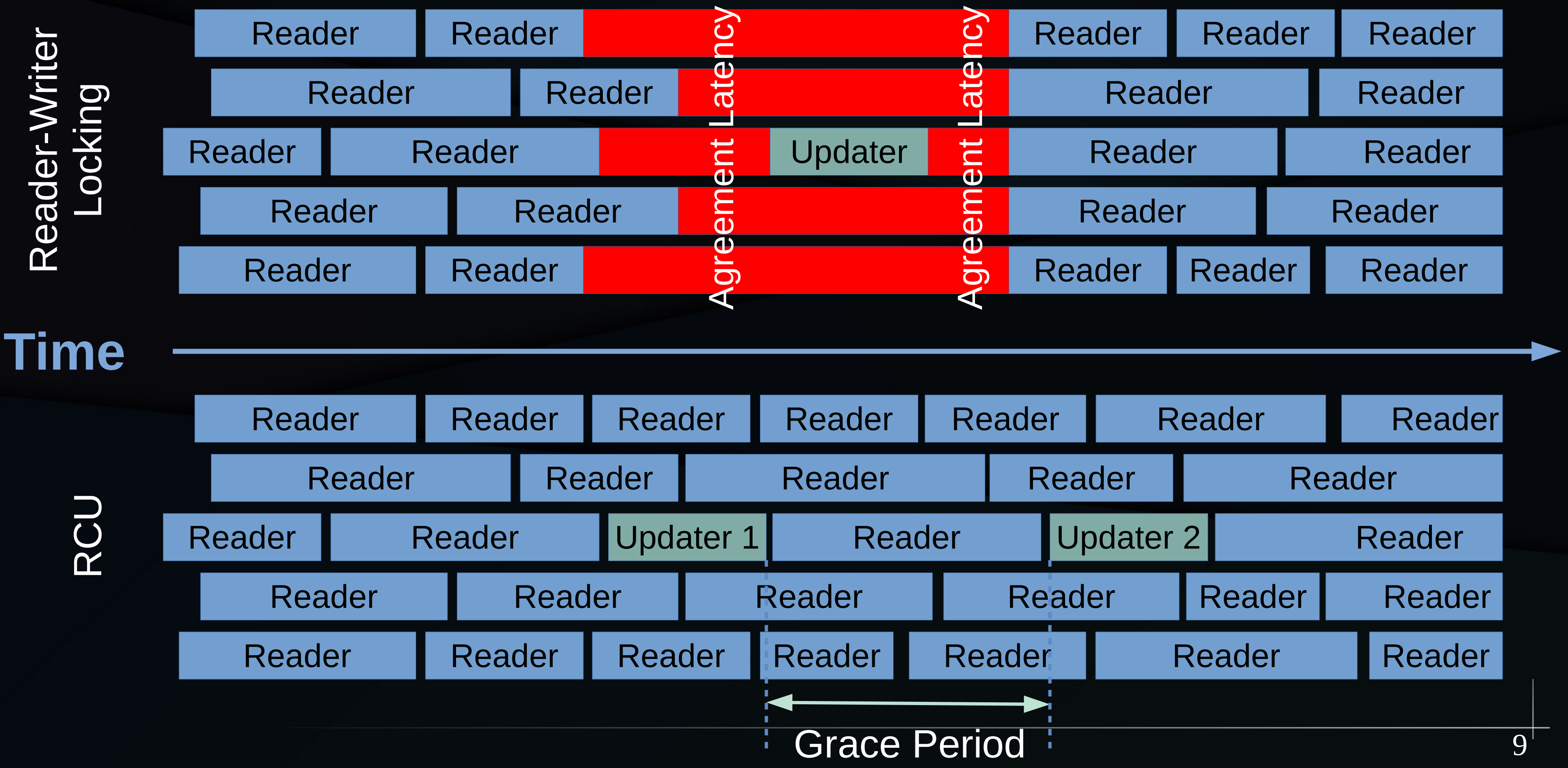
1. RCU provides ABA protection for update-friendly mechanisms (light-weight garbage collector)
2. RCU provides bounded wait-free read-side primitives for real-time use

And RCU is most frequently used for linked data structures.

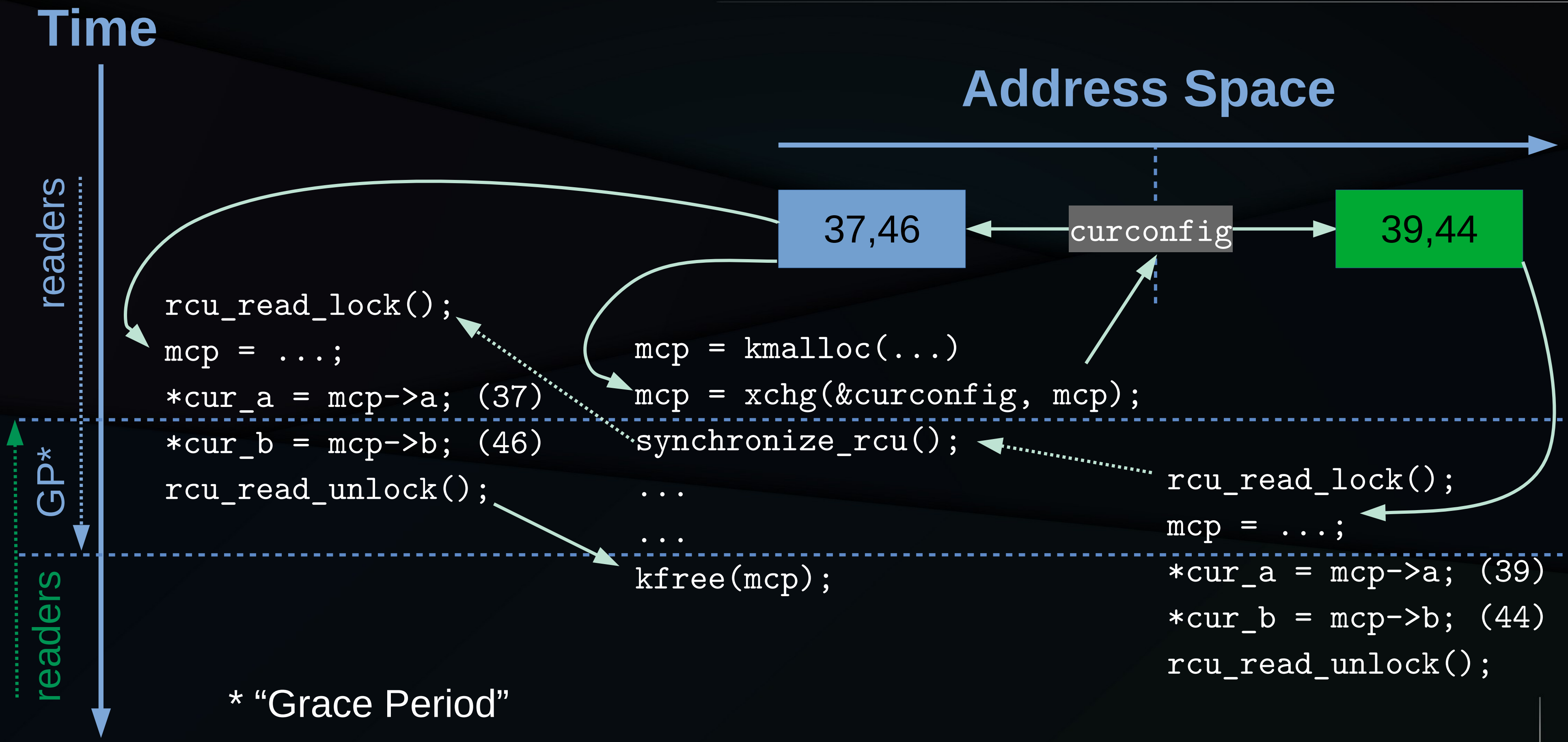
Cost of Global Agreement



RCU vs. Cost of Global Agreement

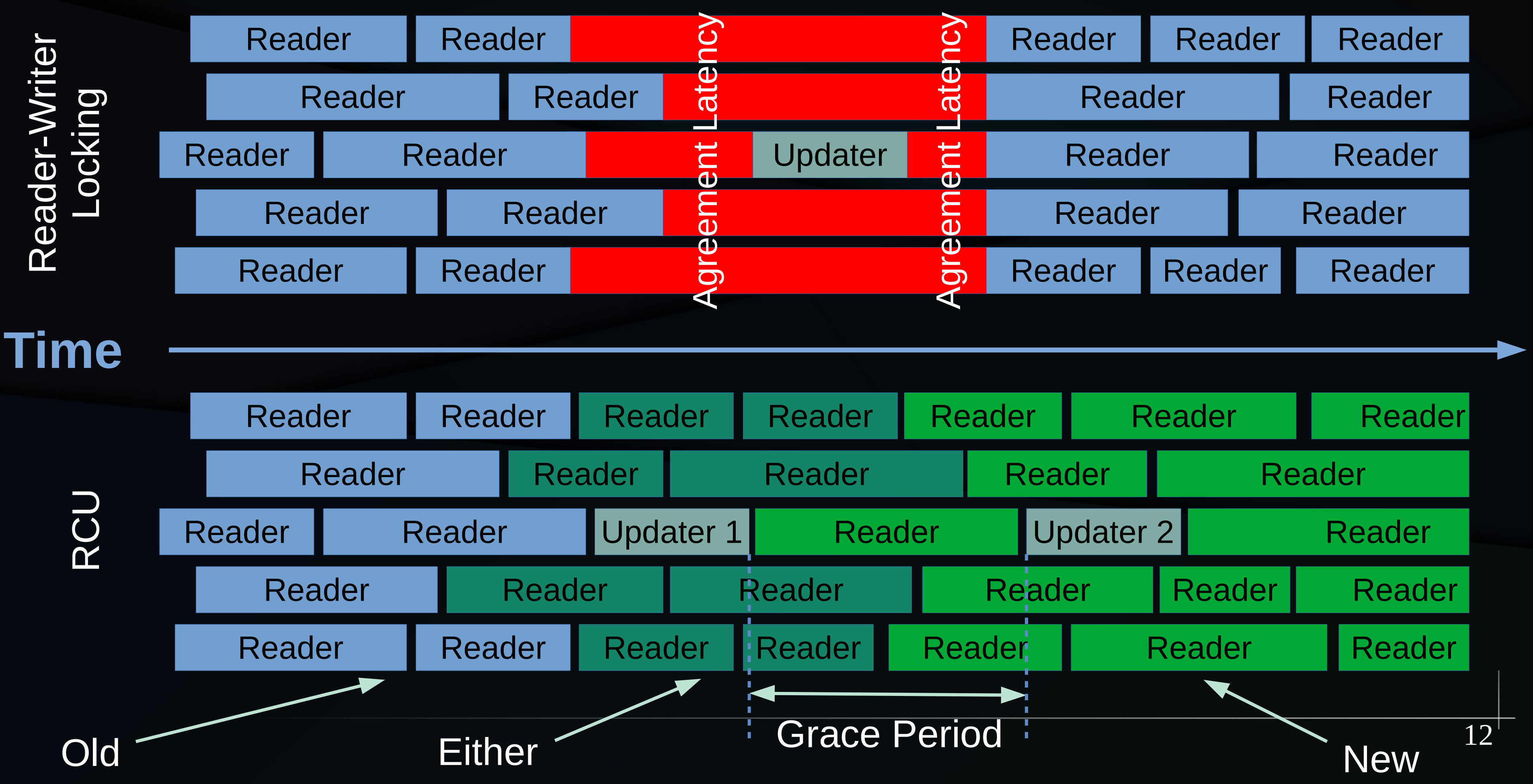


RCU Semantics (Spatio-Temporal)



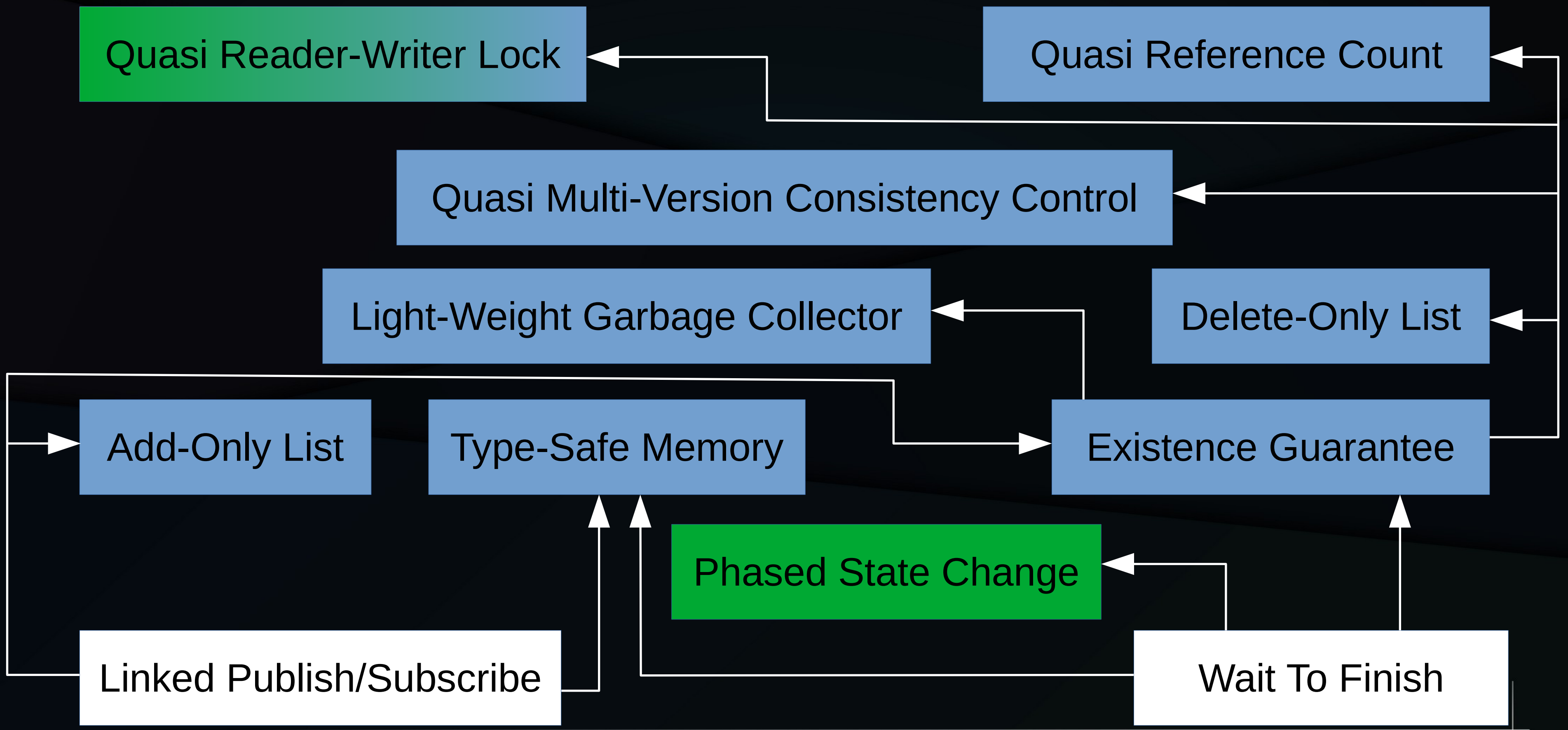
First space/time articulation for RCU (to the best of my knowledge): Jonathan Walpole and his students Josh Triplett and Phil Howard

RCU Spatio-Temporal Values



You Are Here

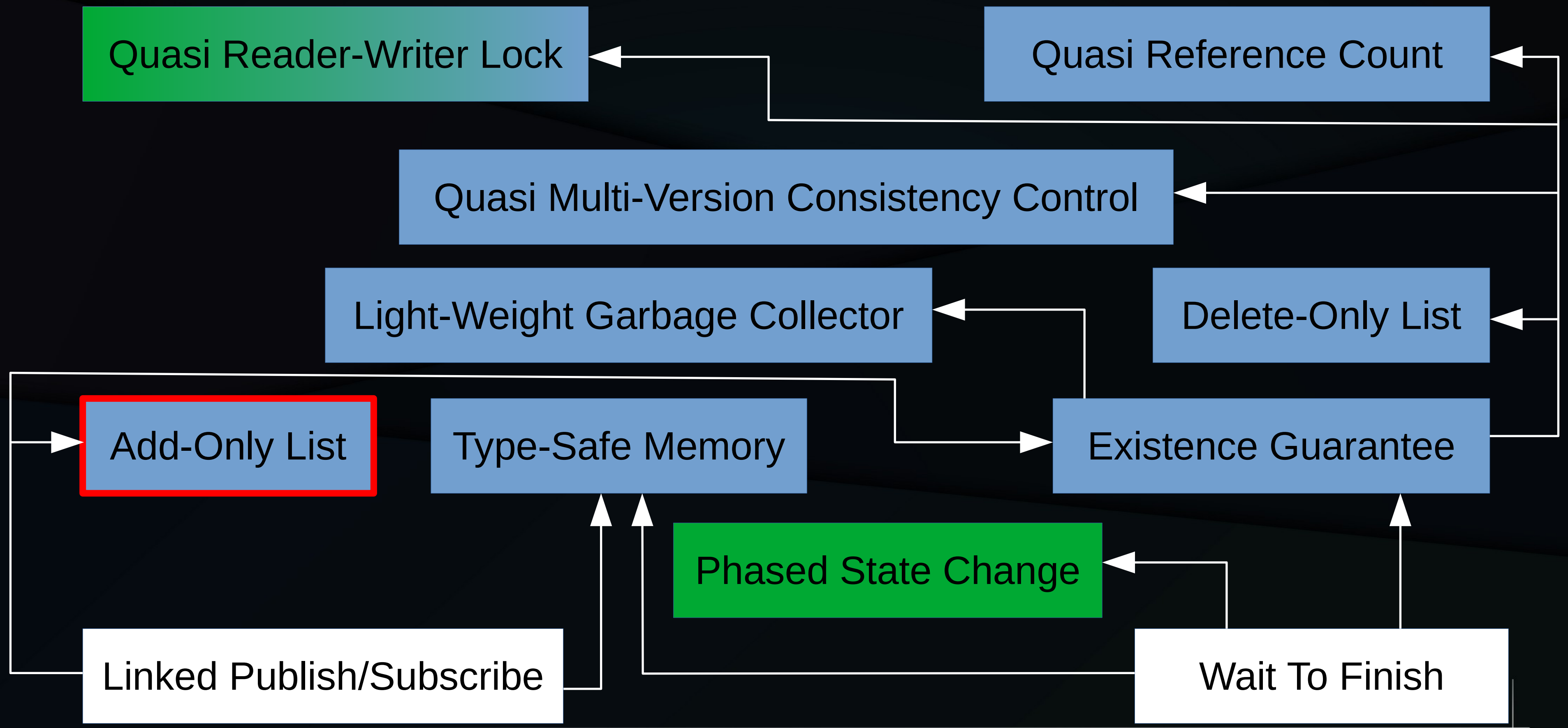
You Are Here



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
Phased State Change
Linked Publish/Subscribe
Wait To Finish

Add-Only List

You Are Here: Add-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
~~rcu_read_lock();~~
list_for_each_entry_rcu(p, &rl, nxt, true)
    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);~~
```

Resulting Code For Add-Only List

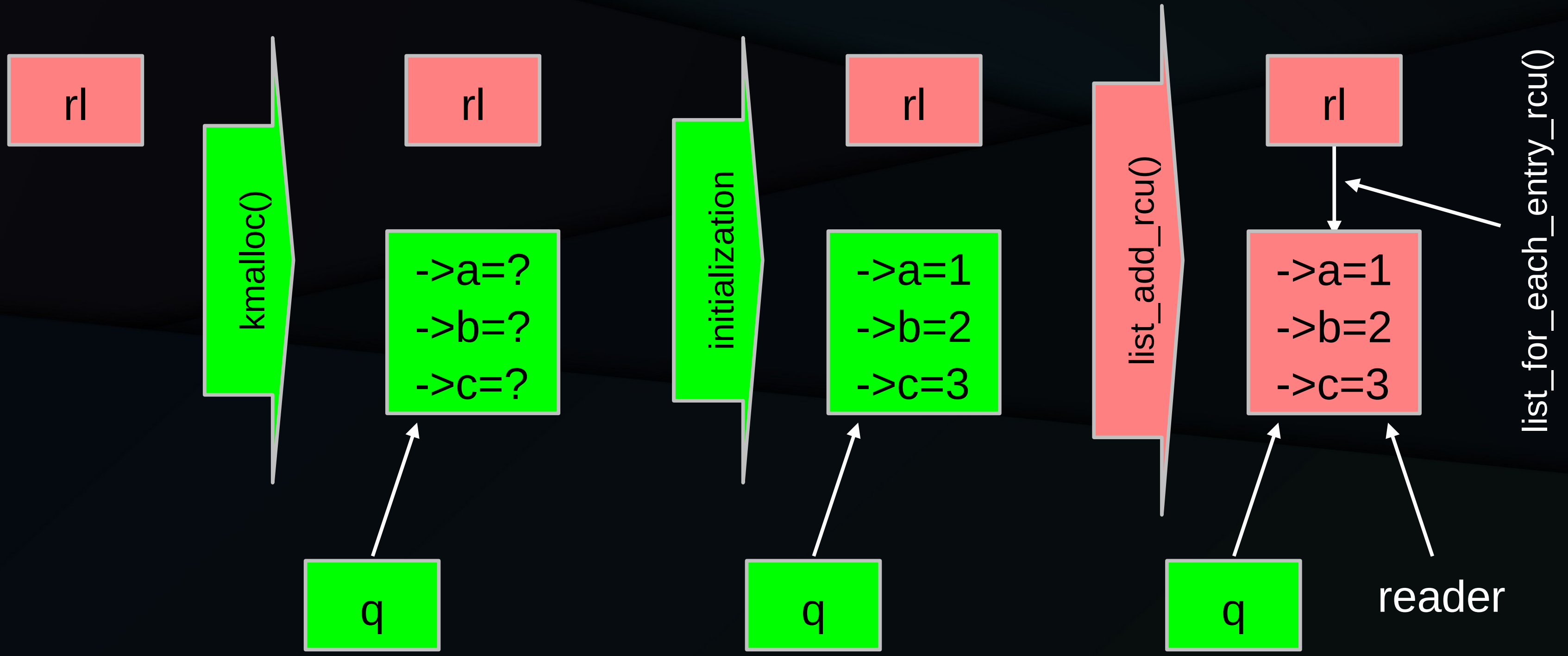
```
// Reader
list_for_each_entry_rcu(p, &rl, nxt, true)
    do_something(p);
```

```
// Updater
spin_lock(&ml);
list_add_rcu(&q->nxt, &rl);
spin_unlock(&ml);
```

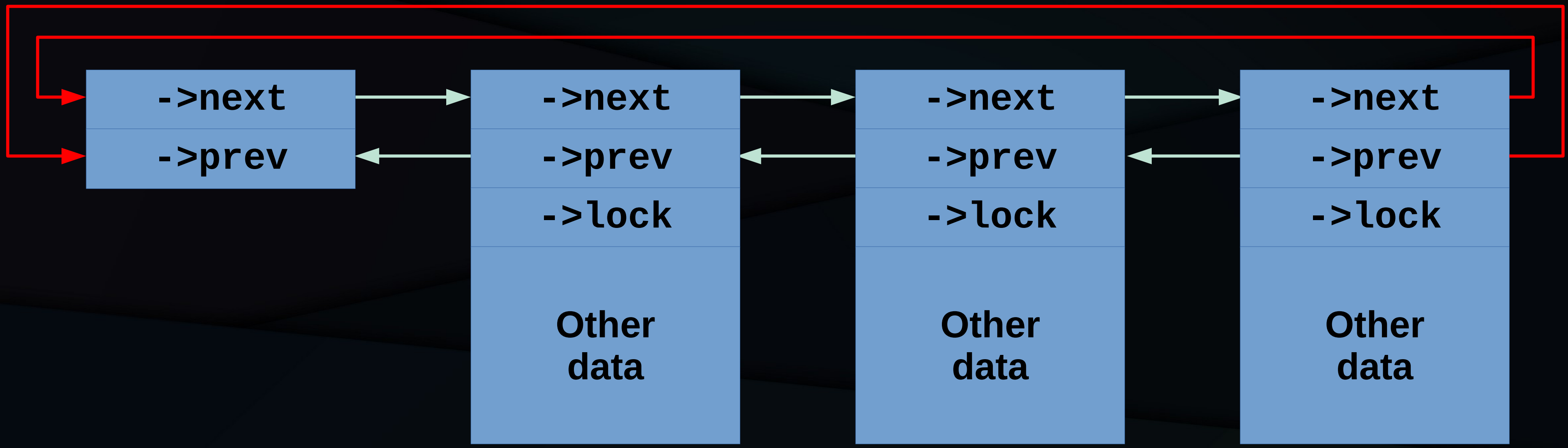
Operation of Add-Only List

Key: Dangerous for updates: all readers can access

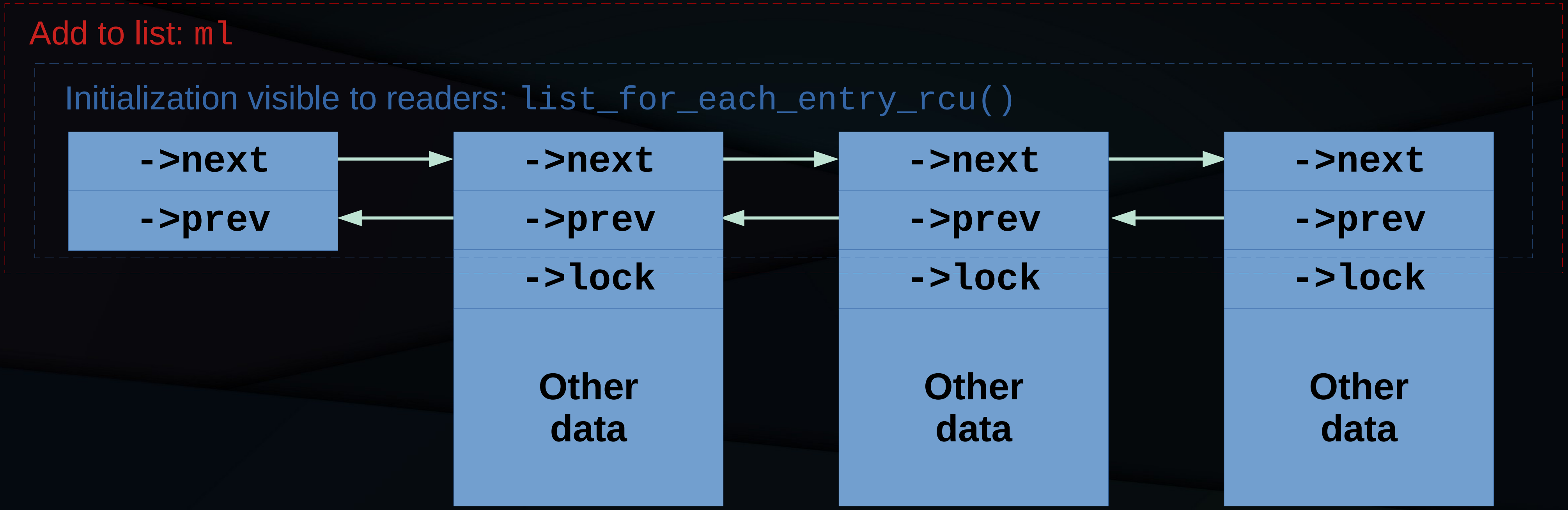
Safe for updates: inaccessible to all readers



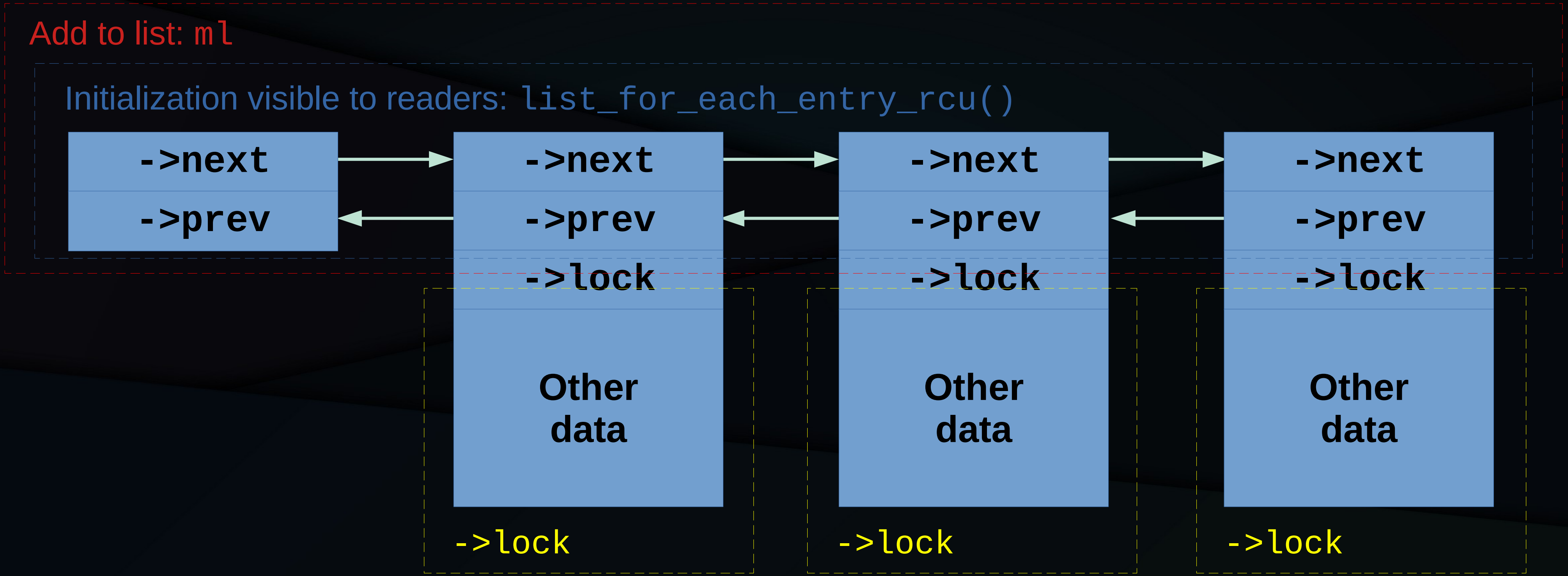
Synchronization Responsibilities



Synchronization Responsibilities



Synchronization Responsibilities



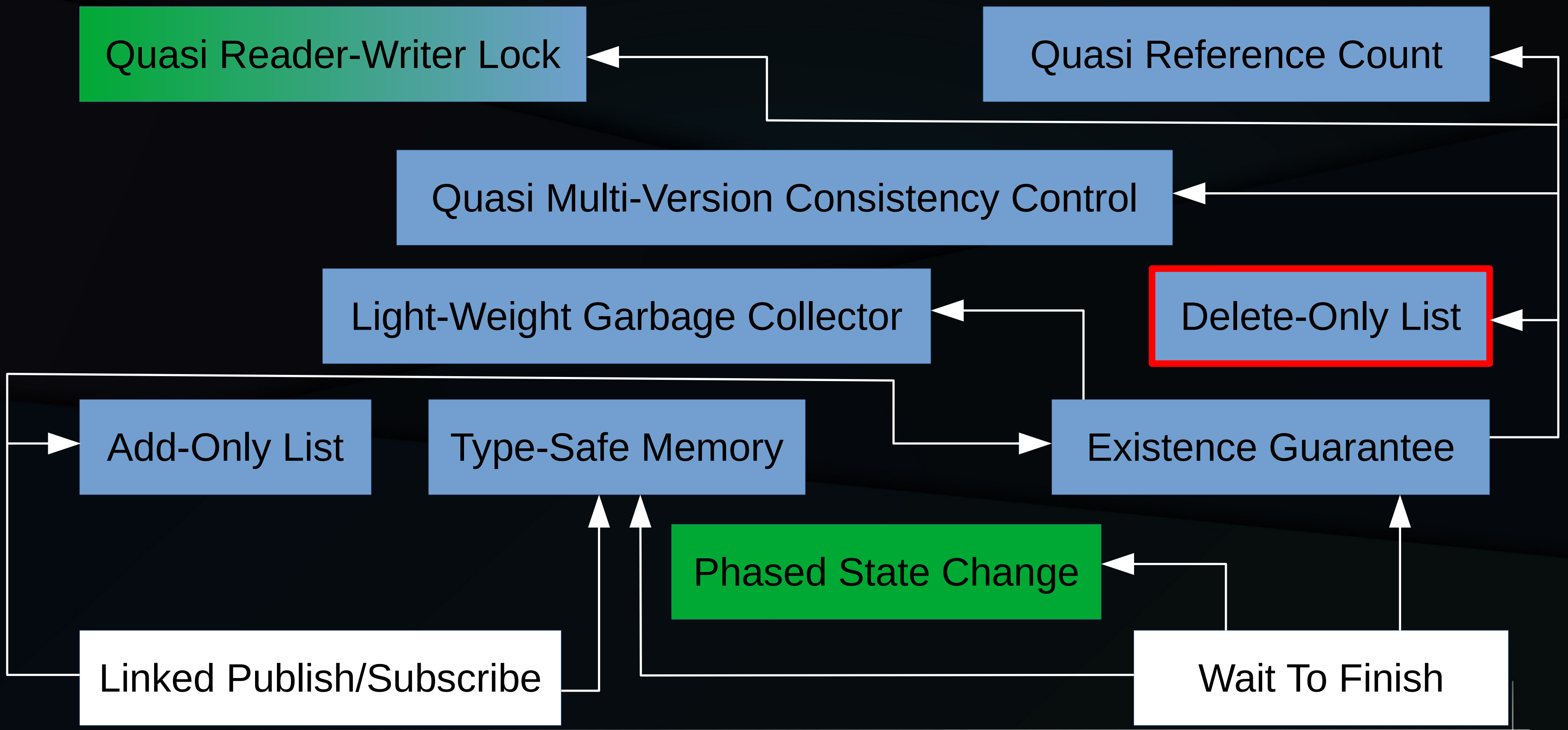
For example, if some of that “other data” is mutable.

RCU to Add-Only List

- Add to publish/subscribe for linked structure:
  - Nothing at all!!!

Delete-Only List

You Are Here: Delete-Only List



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);
```

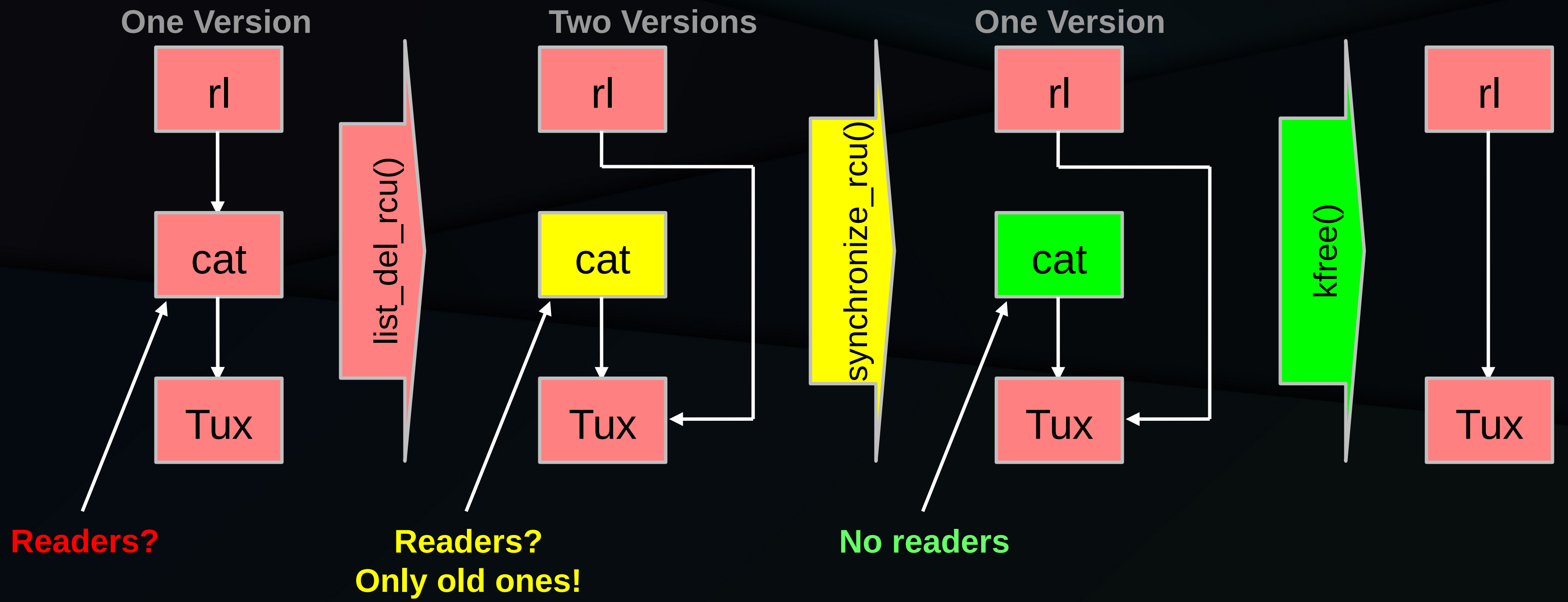
Why? Maybe you have a system that can remove failing devices, but not add new ones.

Resulting Code For Delete-Only List

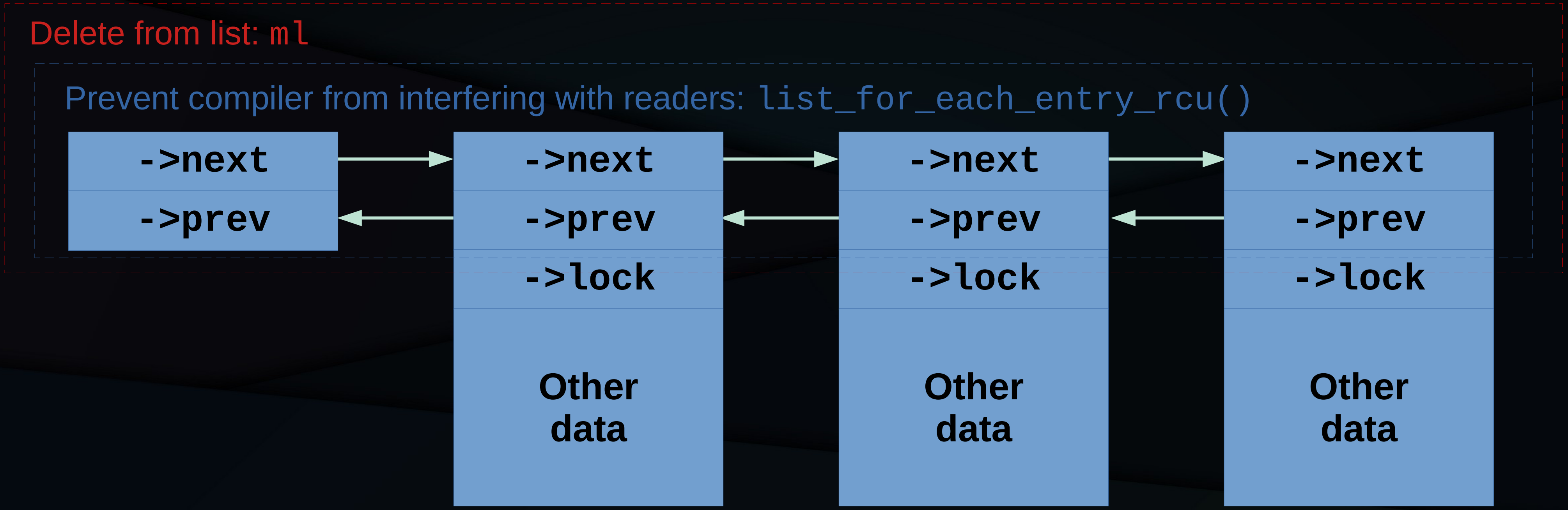
```
// Reader
rcu_read_lock();
list_for_each_entry_rcu(p, &rl, nxt) // Could use READ_ONCE()
    do_something(p);
rcu_read_unlock();

// 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

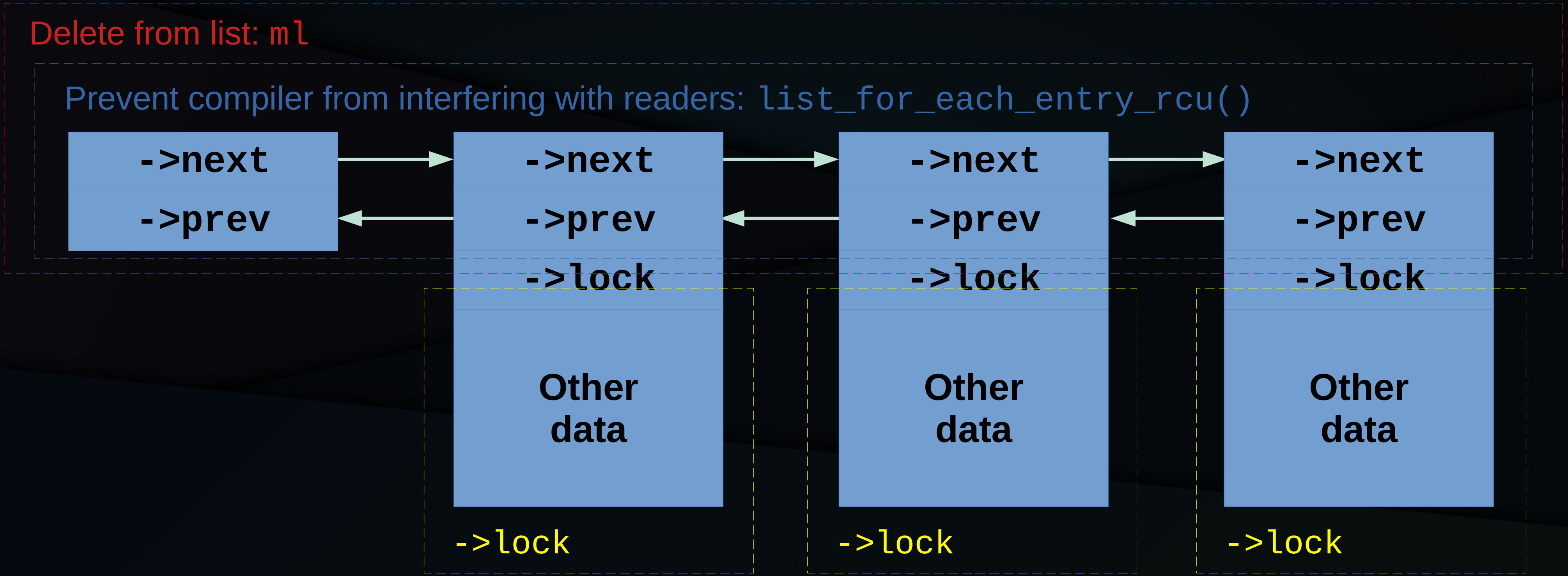
Key: Still dangerous for updates: pre-existing readers can access



Synchronization Responsibilities



Synchronization Responsibilities



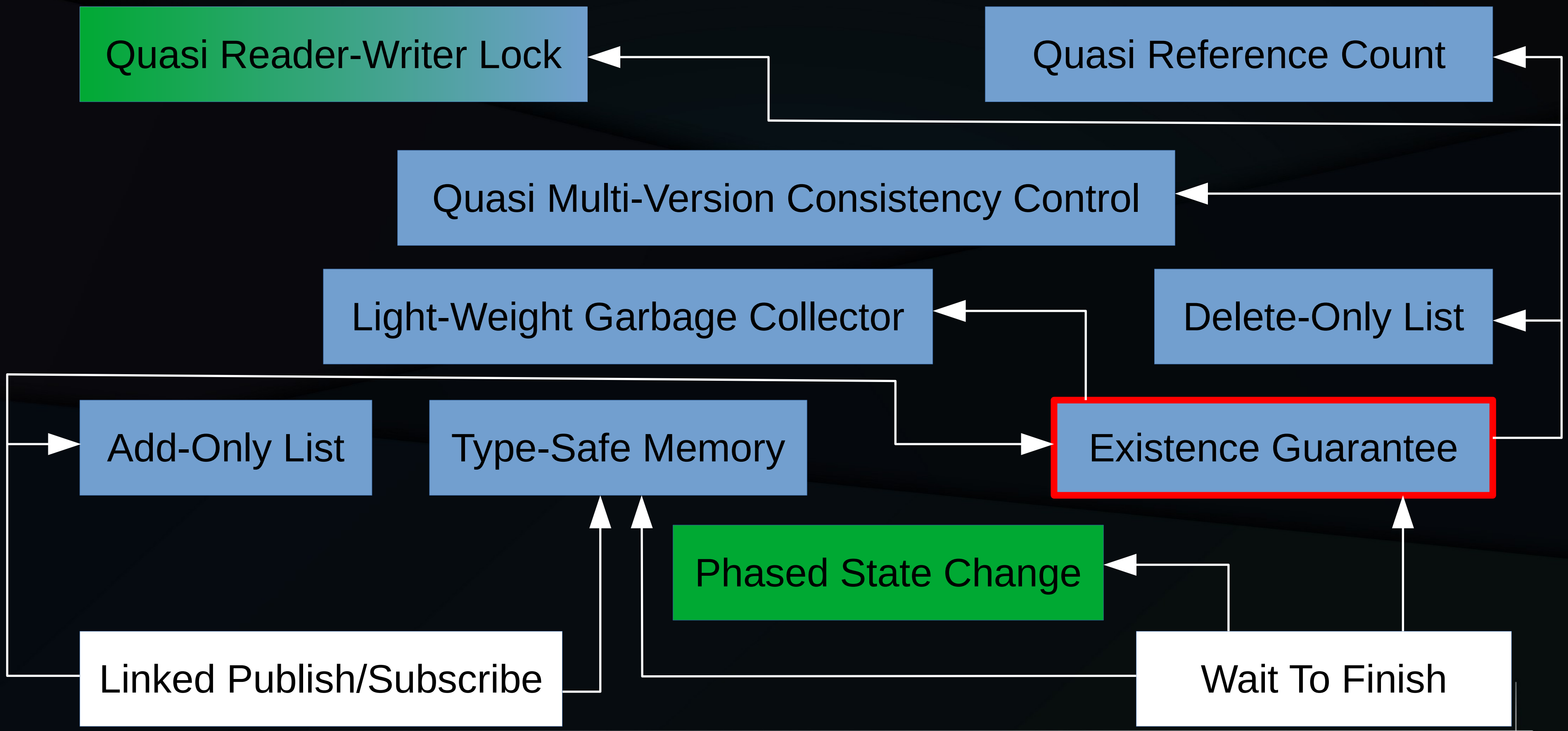
For example, if some of that “other data” is mutable.

RCU to Delete-Only List

- **Remove** from existence guarantee
  - Publish/subscribe for linked structure

Existence Guarantee

You Are Here: Existence Guarantee



Code For Existence Guarantee (Lock)

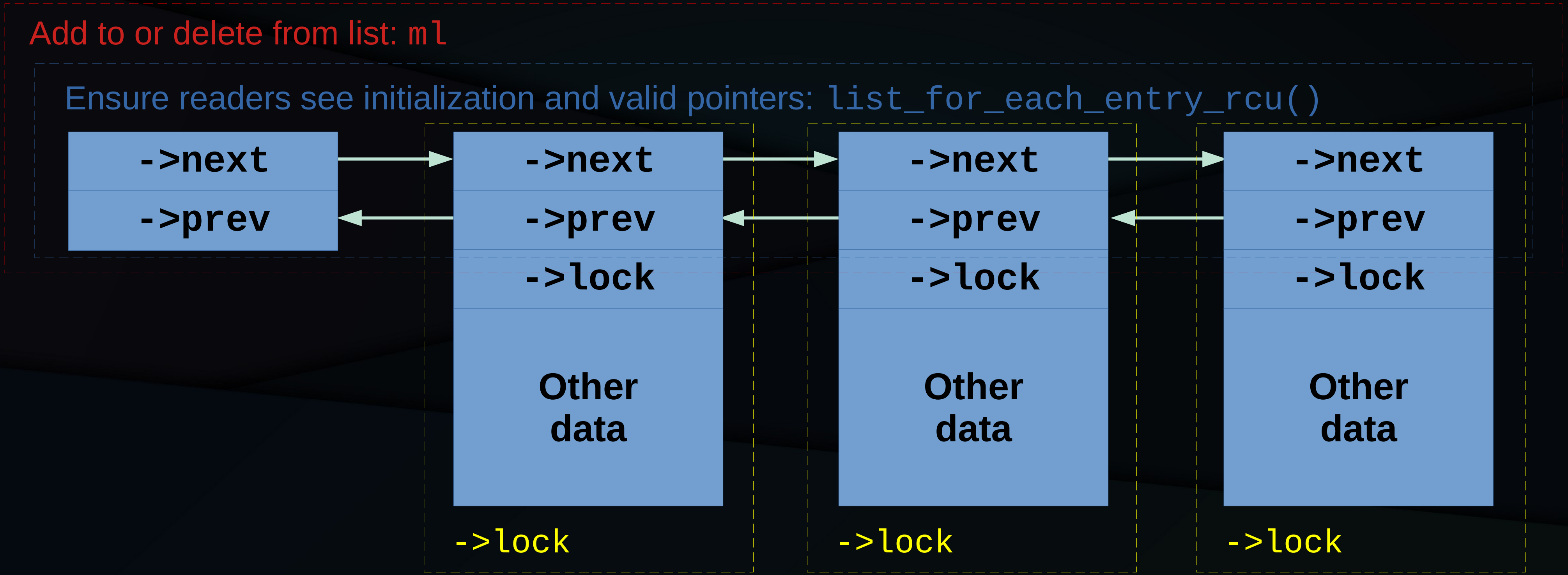
```
// Reader-then-updater
rcu_read_lock();
q = NULL;
list_for_each_entry_rcu(p, &rl, nxt)
    if (p->key == key) {
        q = p;
        spin_lock(&q->lock);  // RCU provides existence guarantee
        break;
    }
rcu_read_unlock();
if (q) {
    if (!p->deleted)
        do_some_update(p);  // Lock protects *p
    spin_unlock(&q->lock);
}
```

This could be used to implement the aforementioned per-node locking.

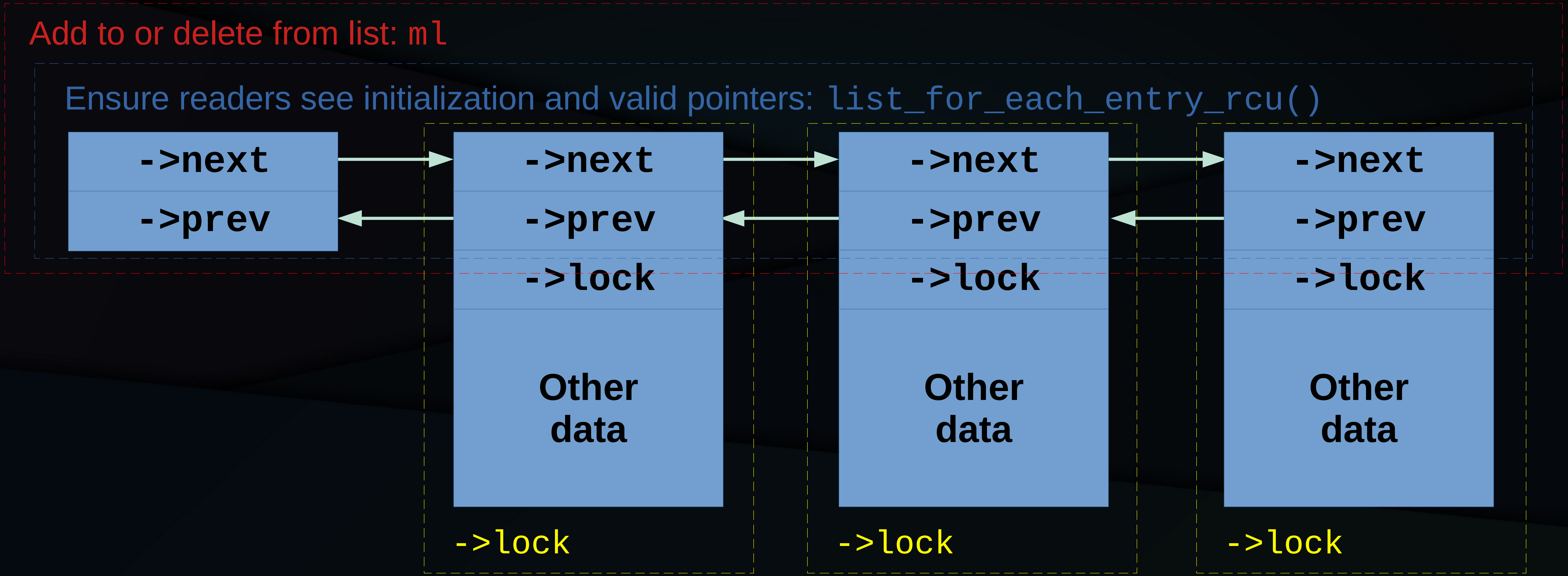
Code For Existence Guarantee (Lock)

```
// 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



Synchronization Responsibilities



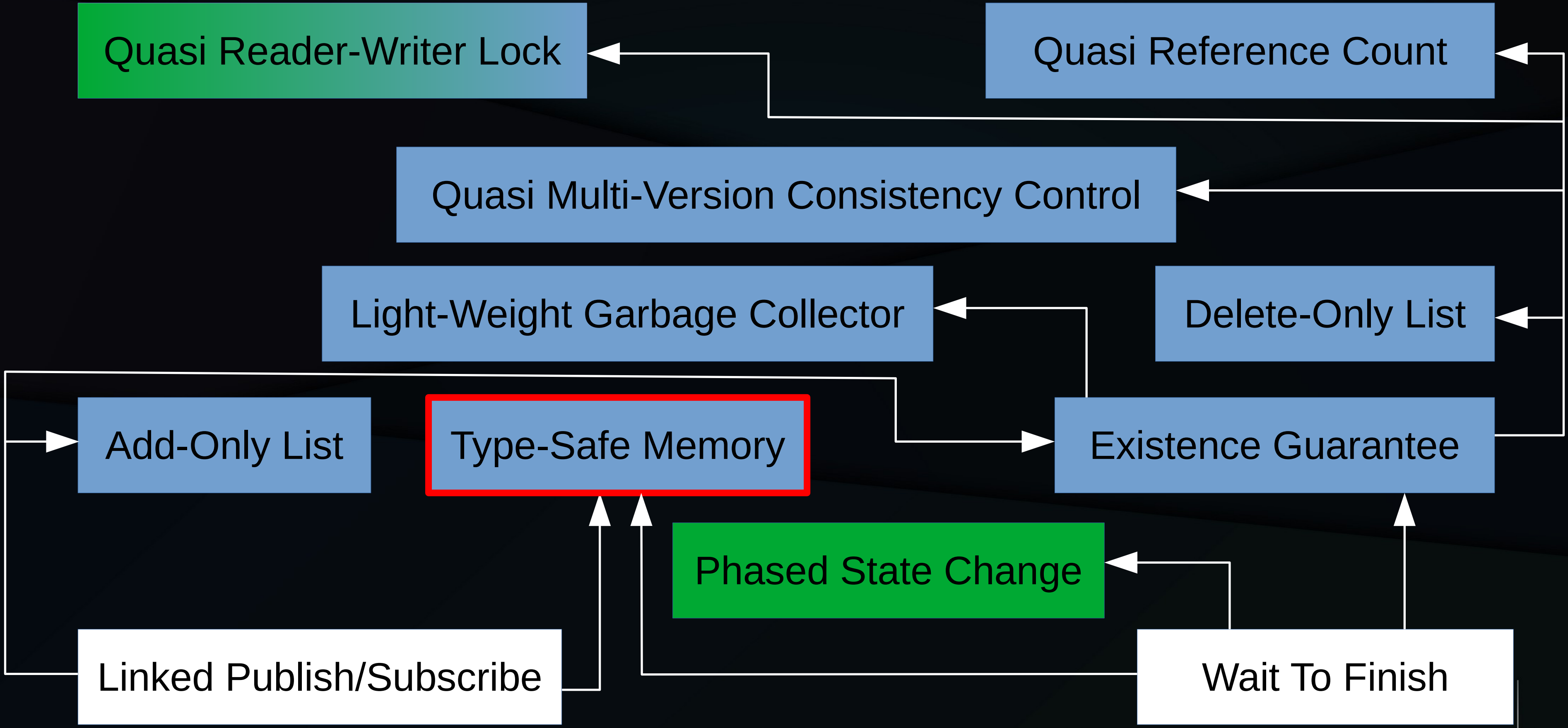
The ->lock protects “other data” and prevents the corresponding node from being removed.

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

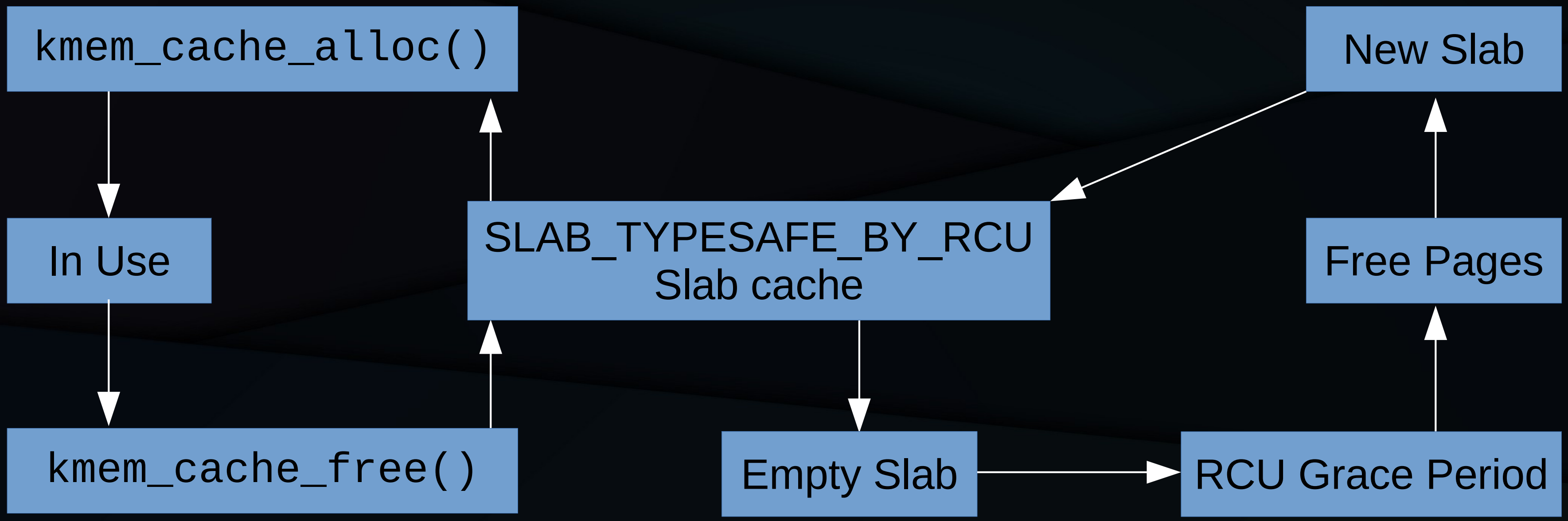
You Are Here: Type-Safe Memory



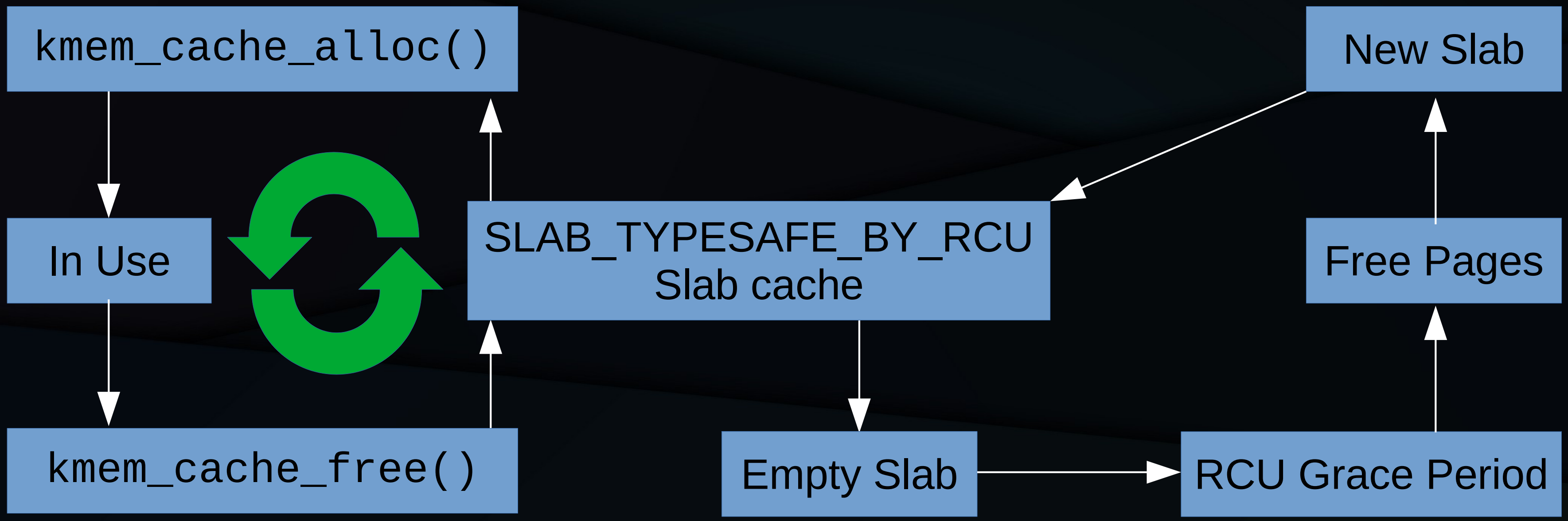
Type-Safe Memory (TSM)

- 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

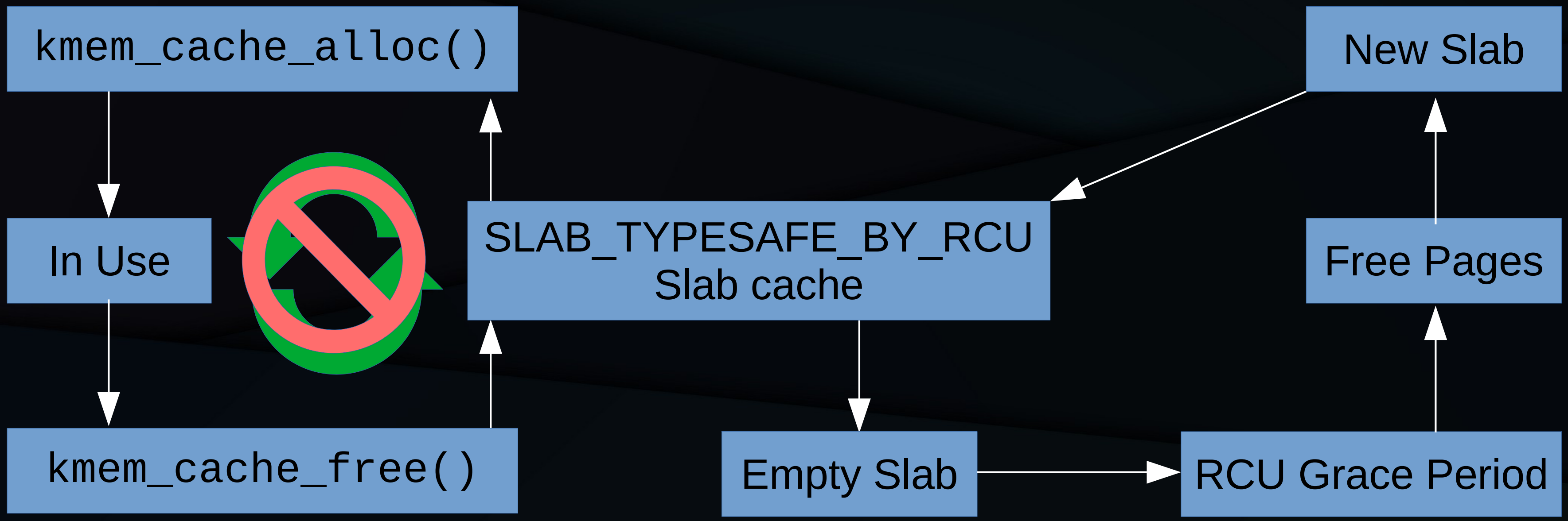
TSM State Diagram



TSM State Diagram



TSM State Diagram



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`.

Structure and Cache

```
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;
}
```

Reader Tries To Obtain Reference

```
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;
}
```

Reader/Remover Releases Reference

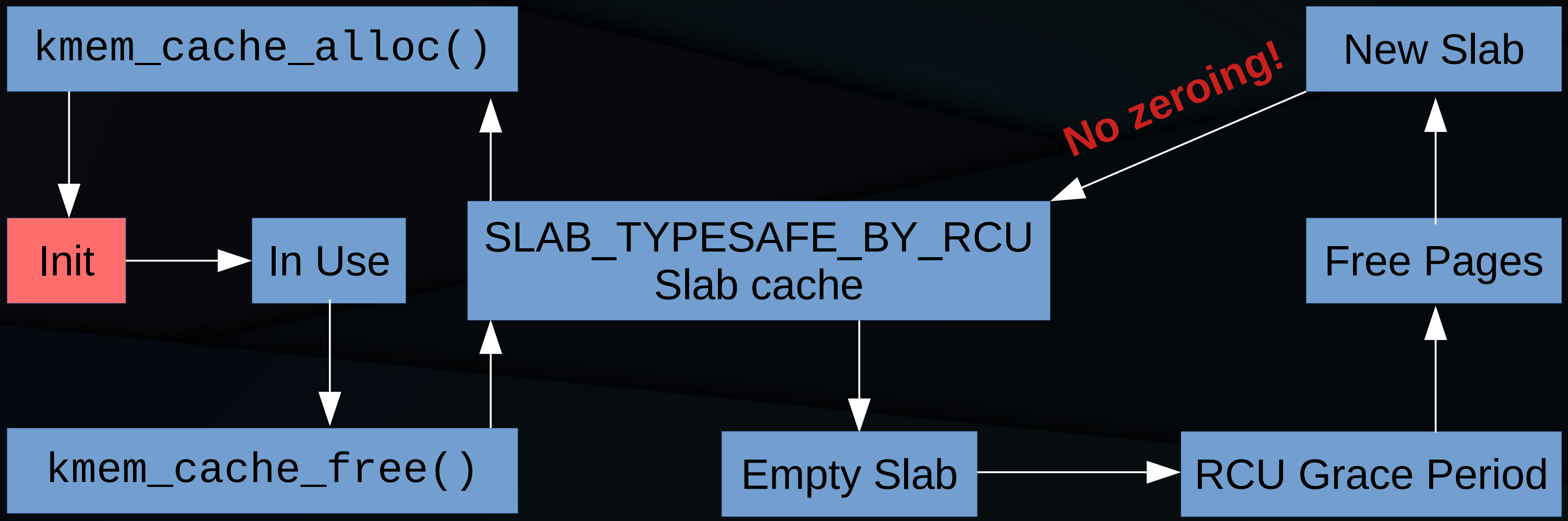
```
static void foo_put(struct foo *p)
{
	if (atomic_dec_and_test(&p->ref)) {
		// Reader attempting to obtain reference will now fail.
		kmem_cache_free(foo_cache, p);
	}
}
```

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

TSM State Diagram Redux



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!

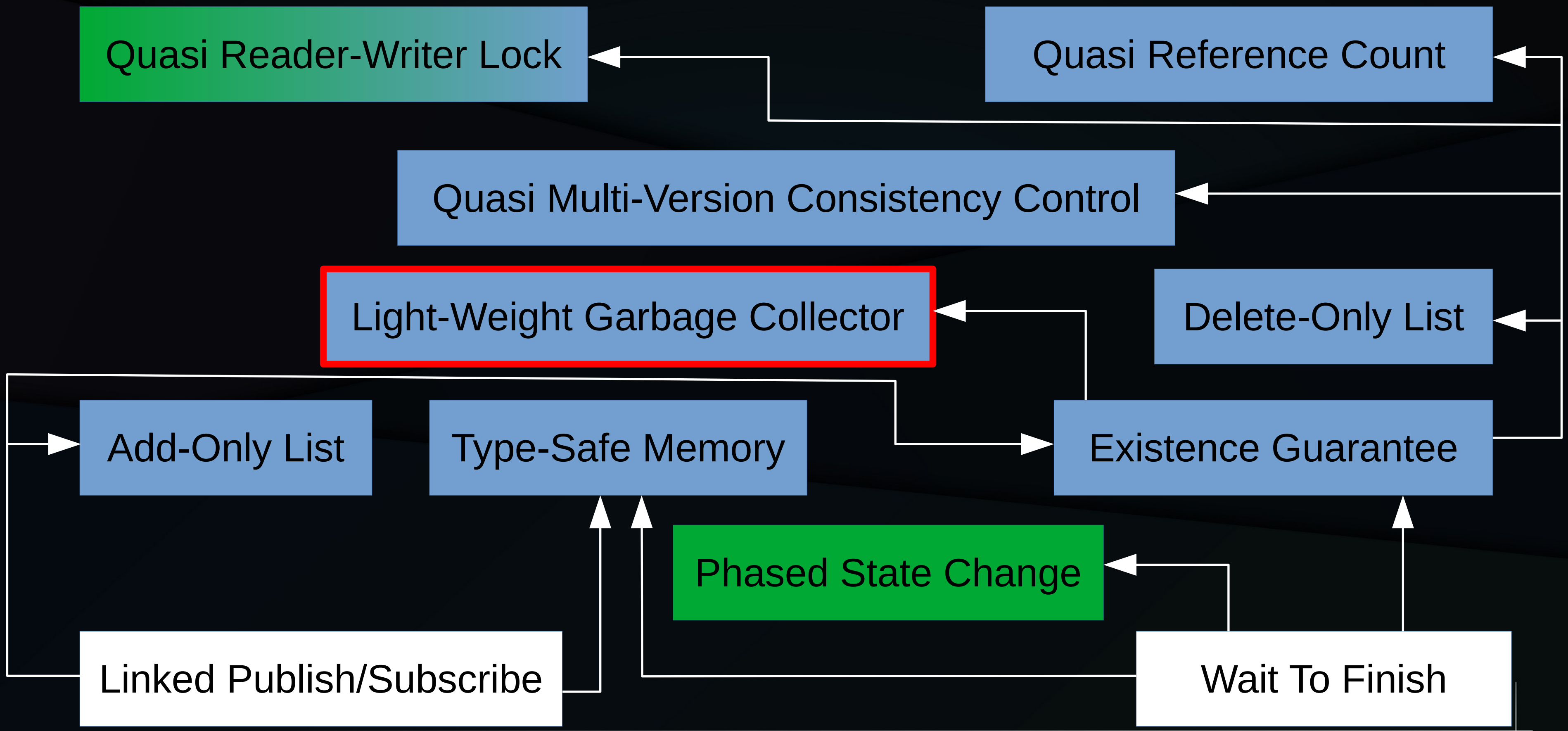
[1] https://lore.kernel.org/lkml/20220209165742.5659-1-quic_qiancai@quicinc.com/ Kudos to Jan Kara

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



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)

RCU: Lightweight GC for NBS (Code)

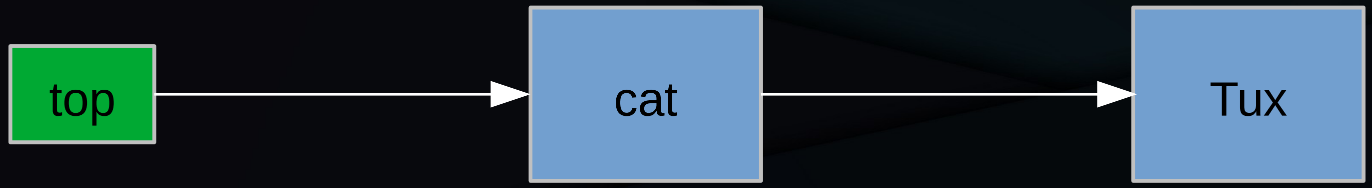
```
struct node_t* top;

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);
}
```

RCU: Lightweight GC for NBS (Code)

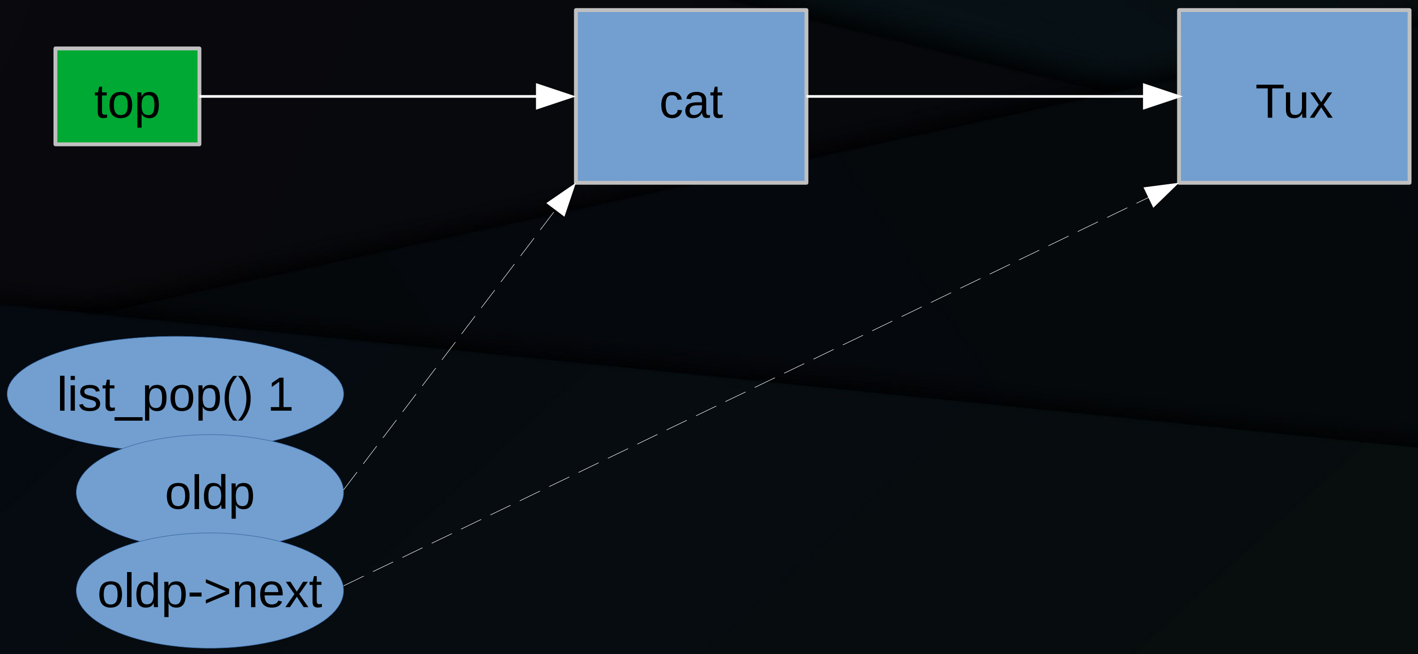
```
struct node_t *list_pop(void)
{
    struct node_t *oldp;
    struct node_t *p;

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

Why is this buggy?

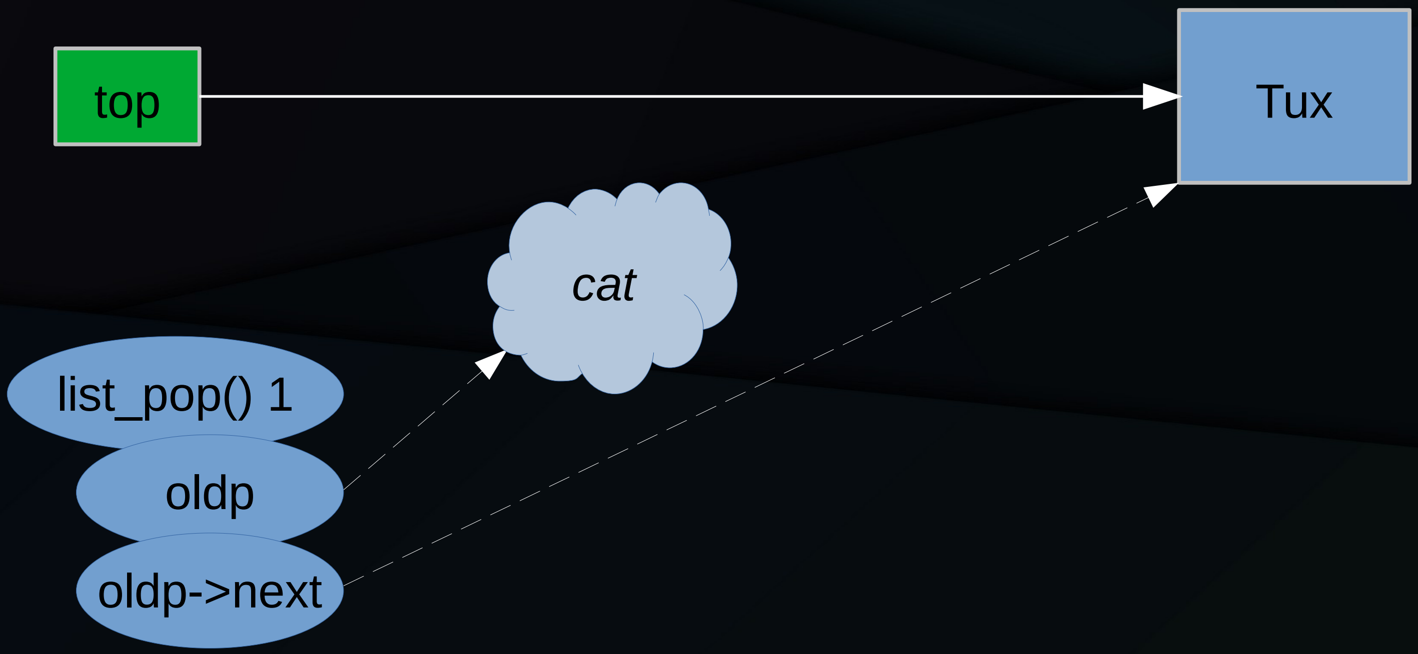
Initial State



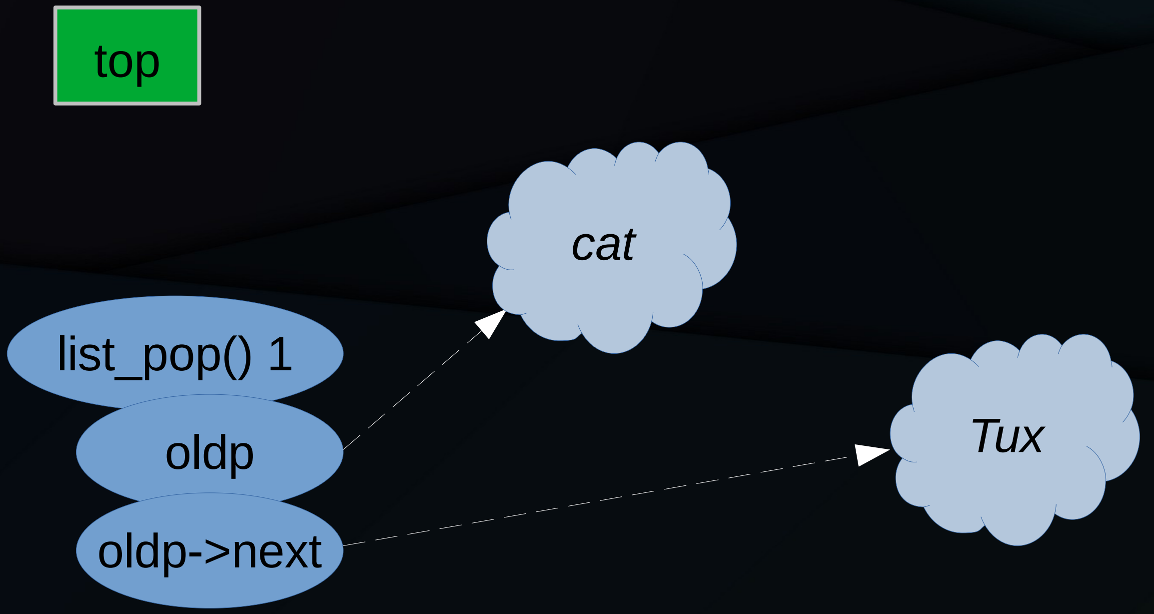
First `list_pop()` is Preempted



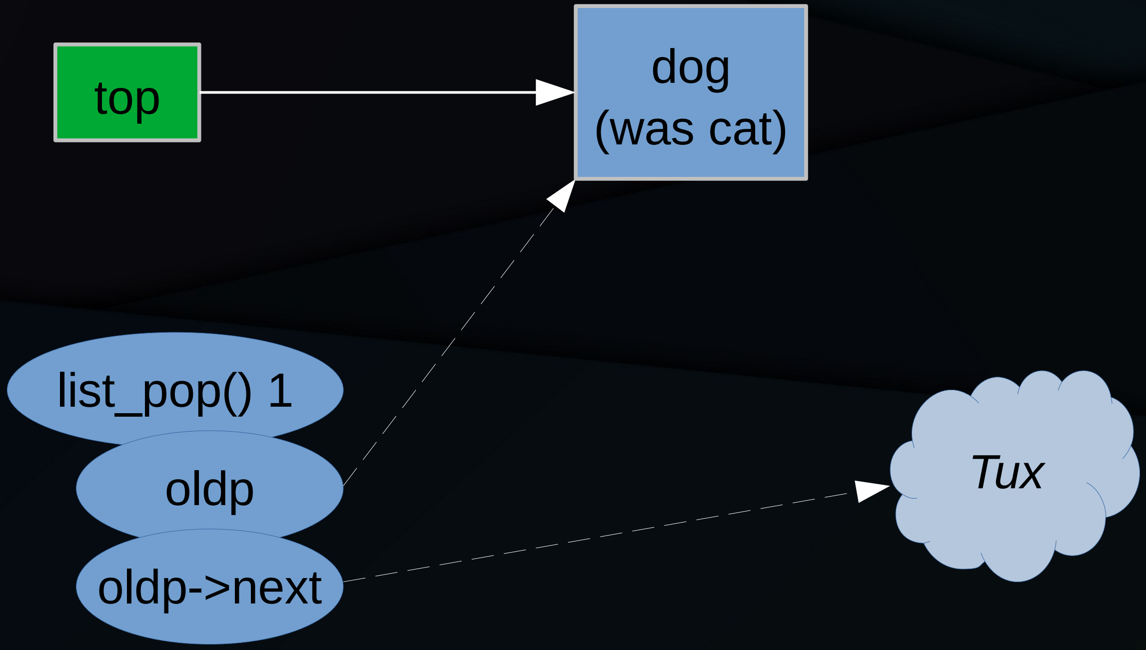
Second list_pop()



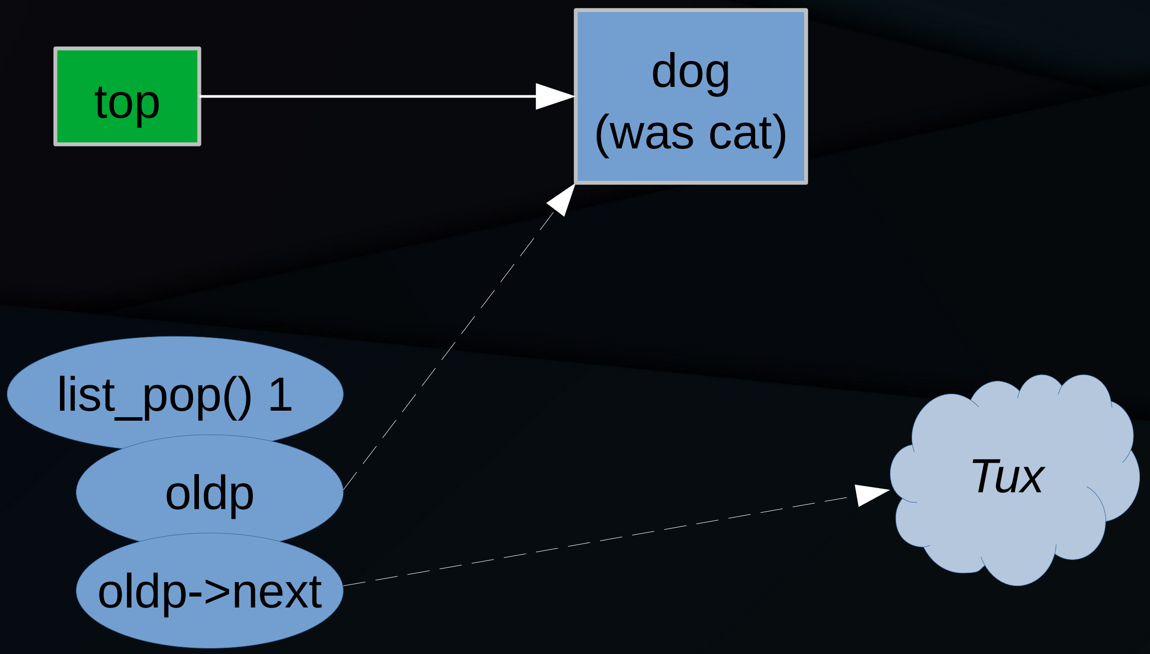
Third `list_pop()`



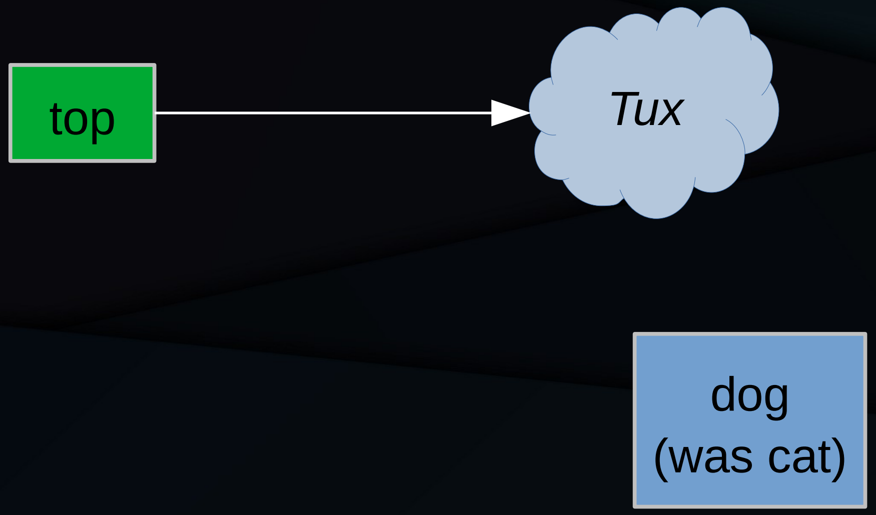
list_push(dog)



First `list_pop()` Resumes

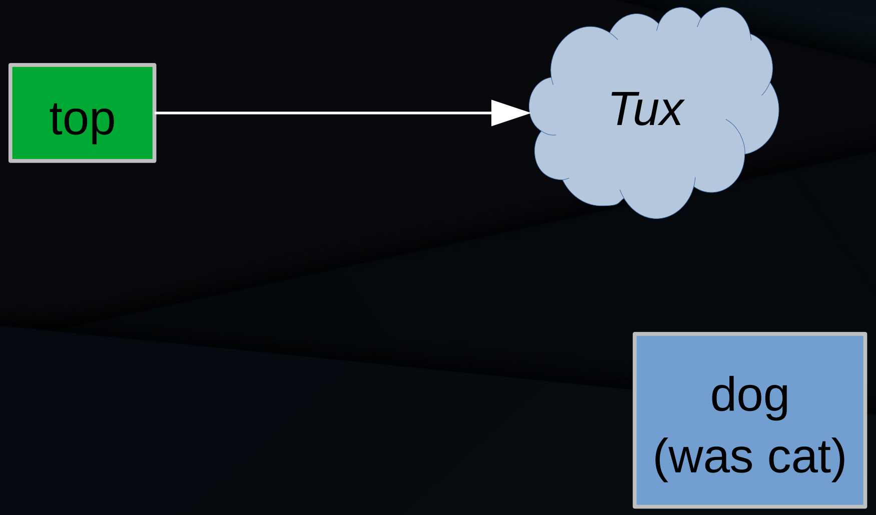


First `list_pop()` Completes



This is the dreaded ABA problem!

First `list_pop()` Completes



This is the dreaded ABA problem! Prevent this by preventing reallocation of cat...

RCU: Lightweight GC for NBS (Code)

```
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;
}
```

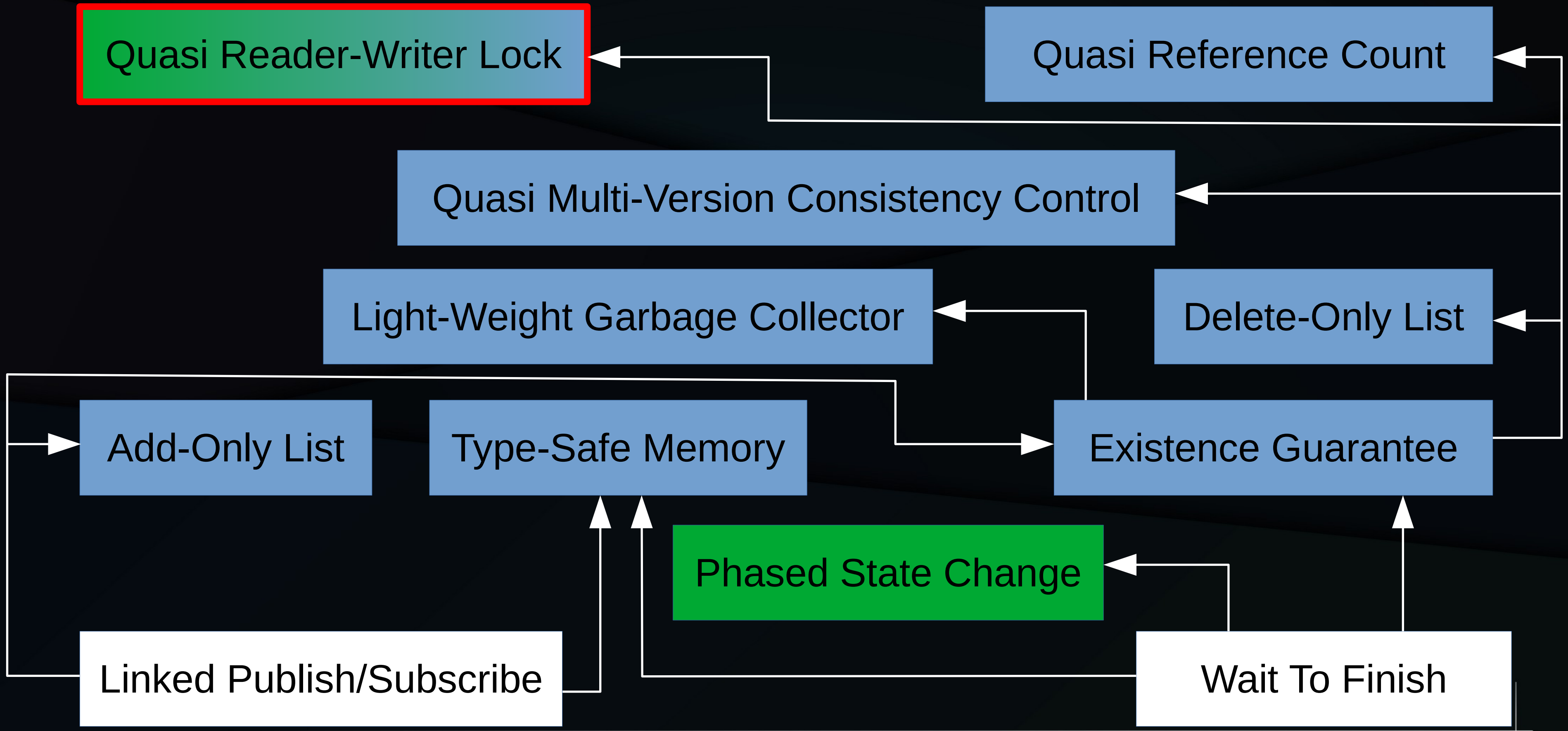
Also need to deferred-free nodes popped from the stack.

RCU to Light-Weight GC

- Add to type-safe memory:
  - Non-blocking synchronization

Quasi Reader-Writer Lock (Redux)

Quasi Reader-Writer Lock (Redux)



Read-To-Write Upgrade

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-Traversal

```
// 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();
```

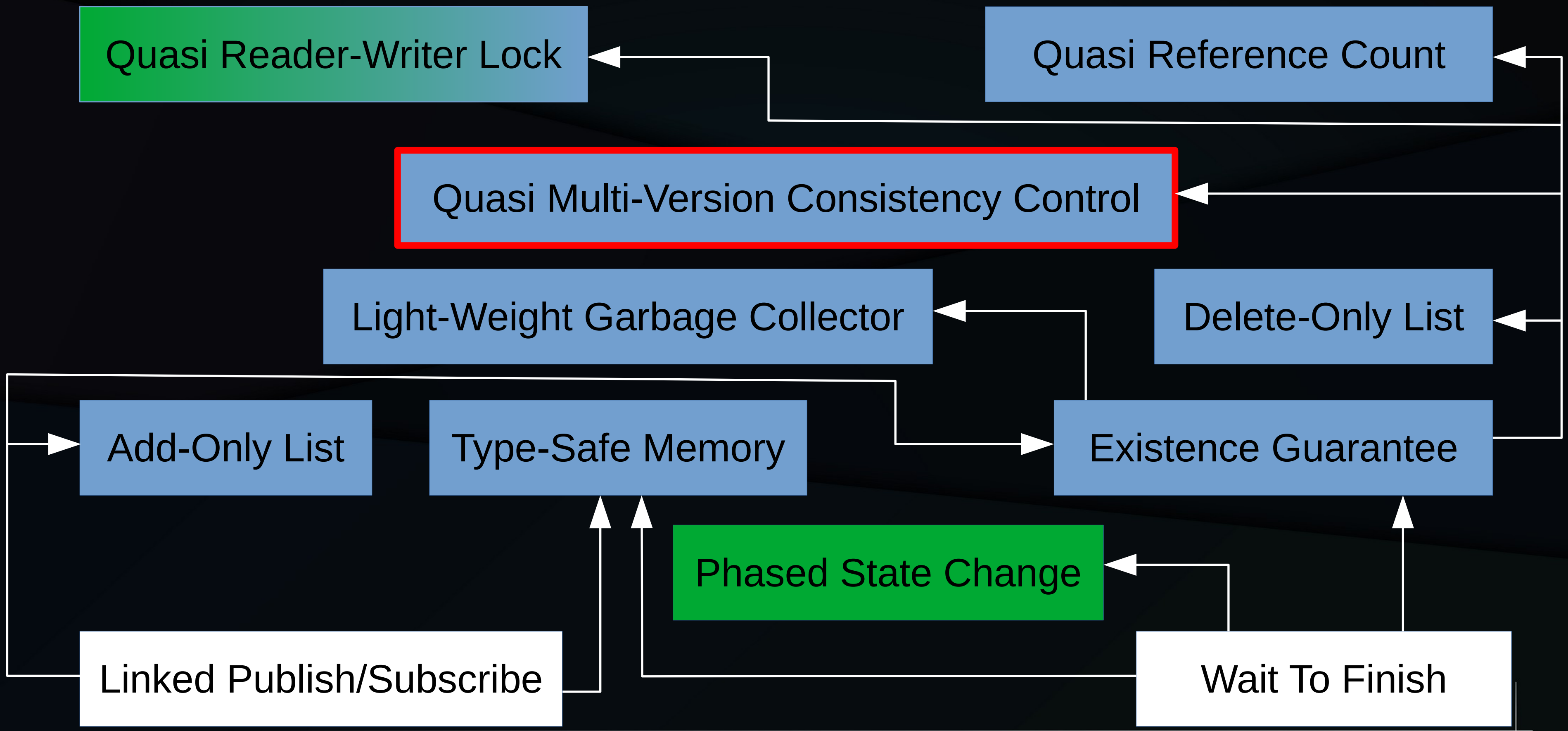
RCU to Quasi Reader-Writer Lock

- 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

You Are Here: Quasi MVCC



Pathname-Lookup Use Case

Hide

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`”

Pathname Lookup and Renames

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`”

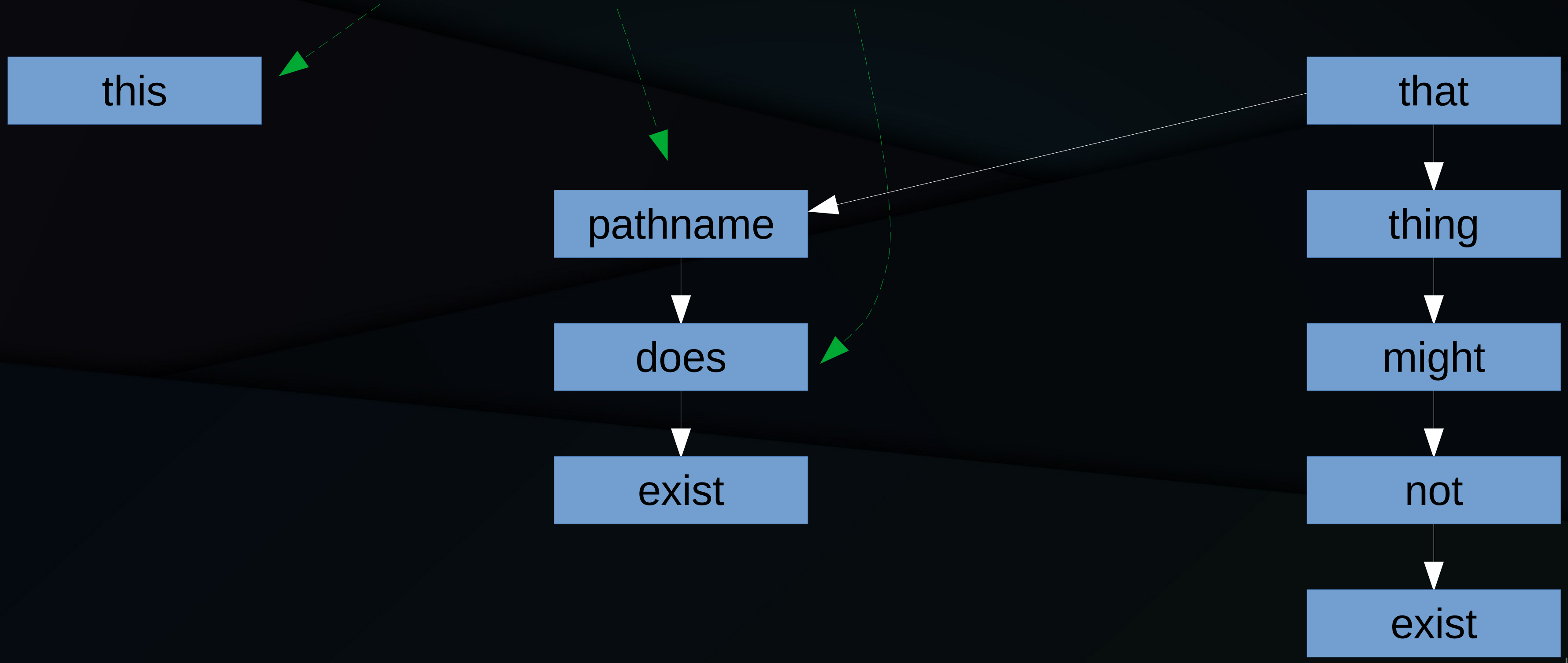
Pathname Lookup and Renames

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

Meanwhile: “`mv /this/pathname /that`”

Pathname Lookup and Renames

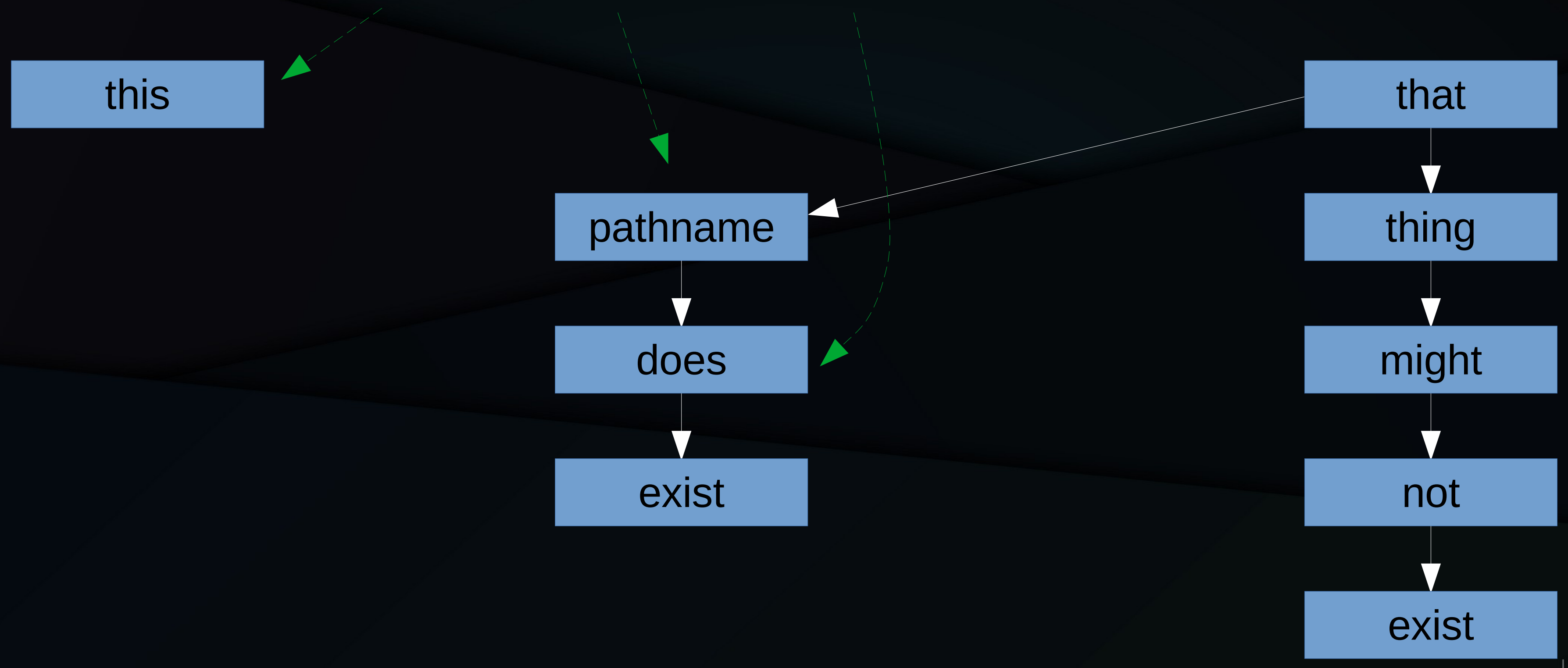
Looking up: “`/this/pathname/does/not/exist`”



Done: “`mv /this/pathname /that`”

Pathname Lookup and Renames

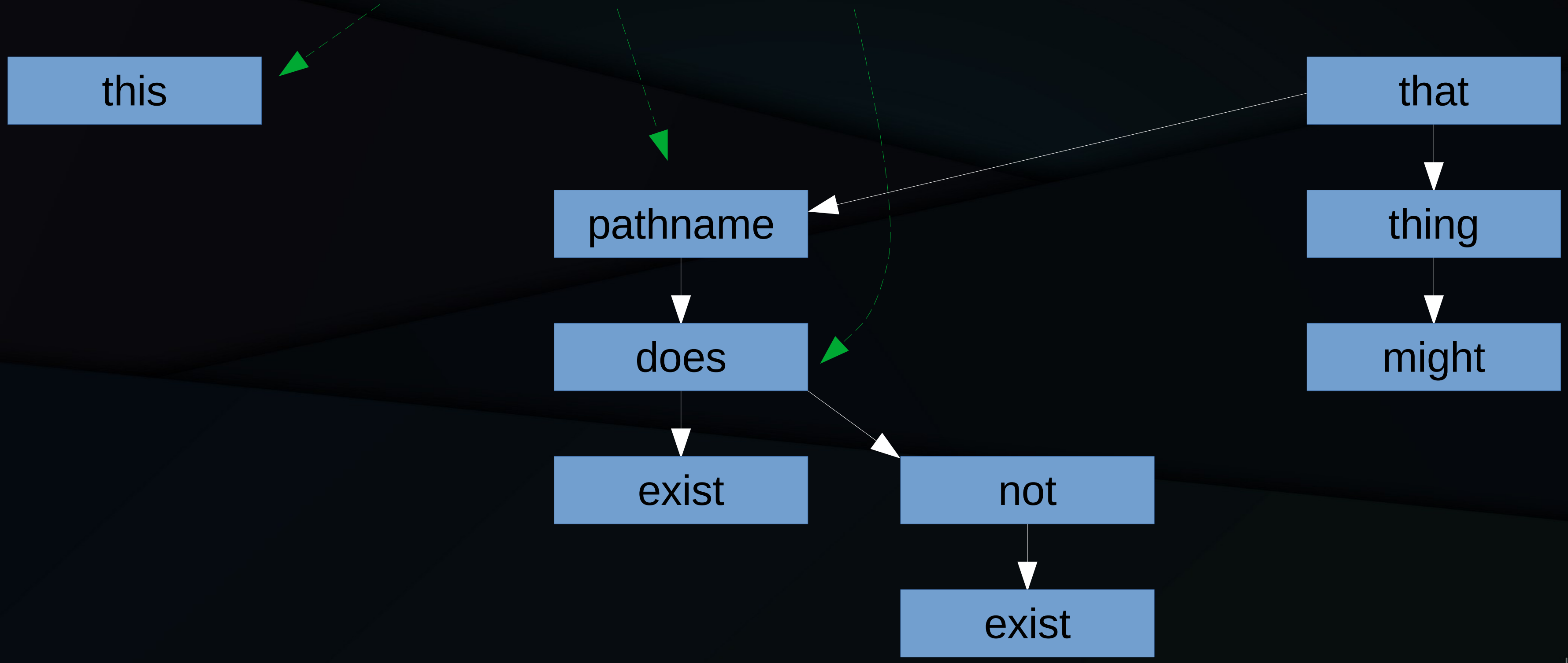
Looking up: “`/this/pathname/does/not/exist`”



Meanwhile: “`mv /that/thing/might/not /that/pathname/does`”

Pathname Lookup and Renames

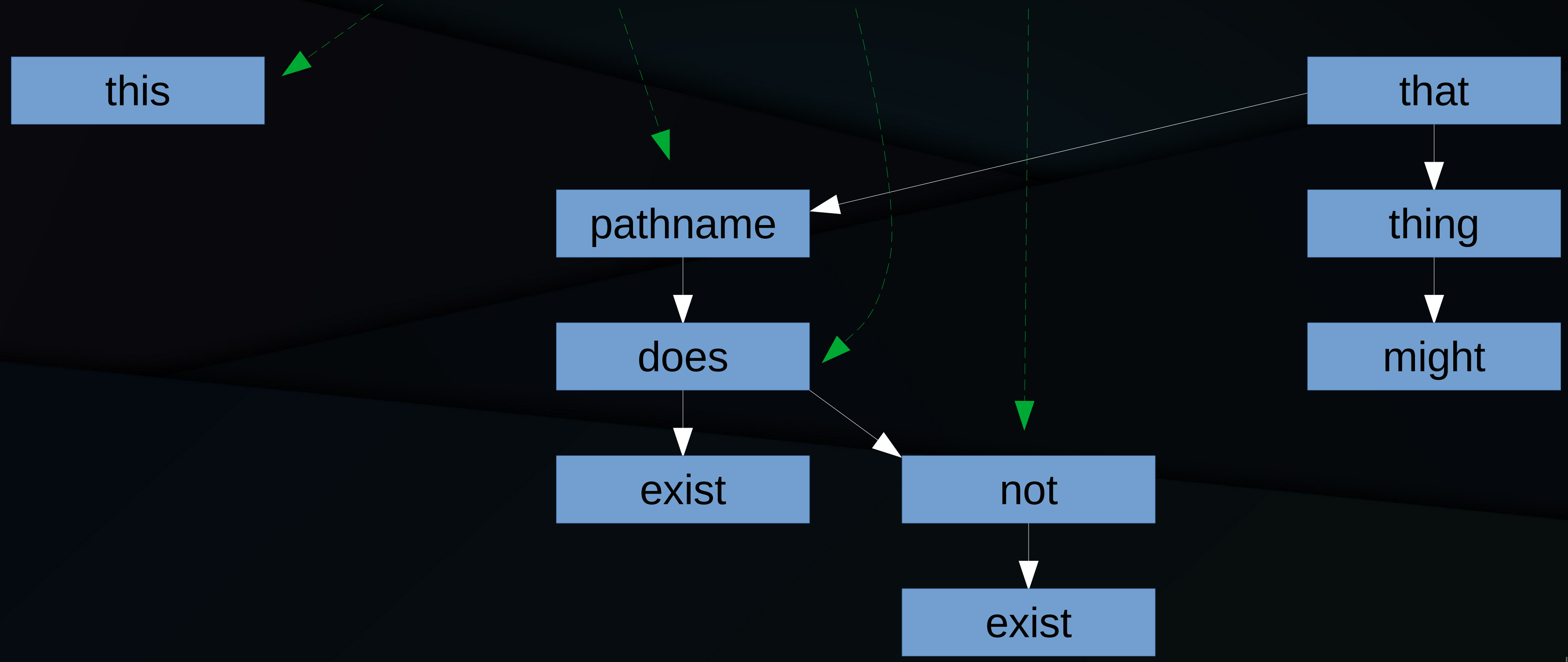
Looking up: “`/this/pathname/does/not/exist`”



Done: “`mv /that/thing/might/not /that/pathname/does`”

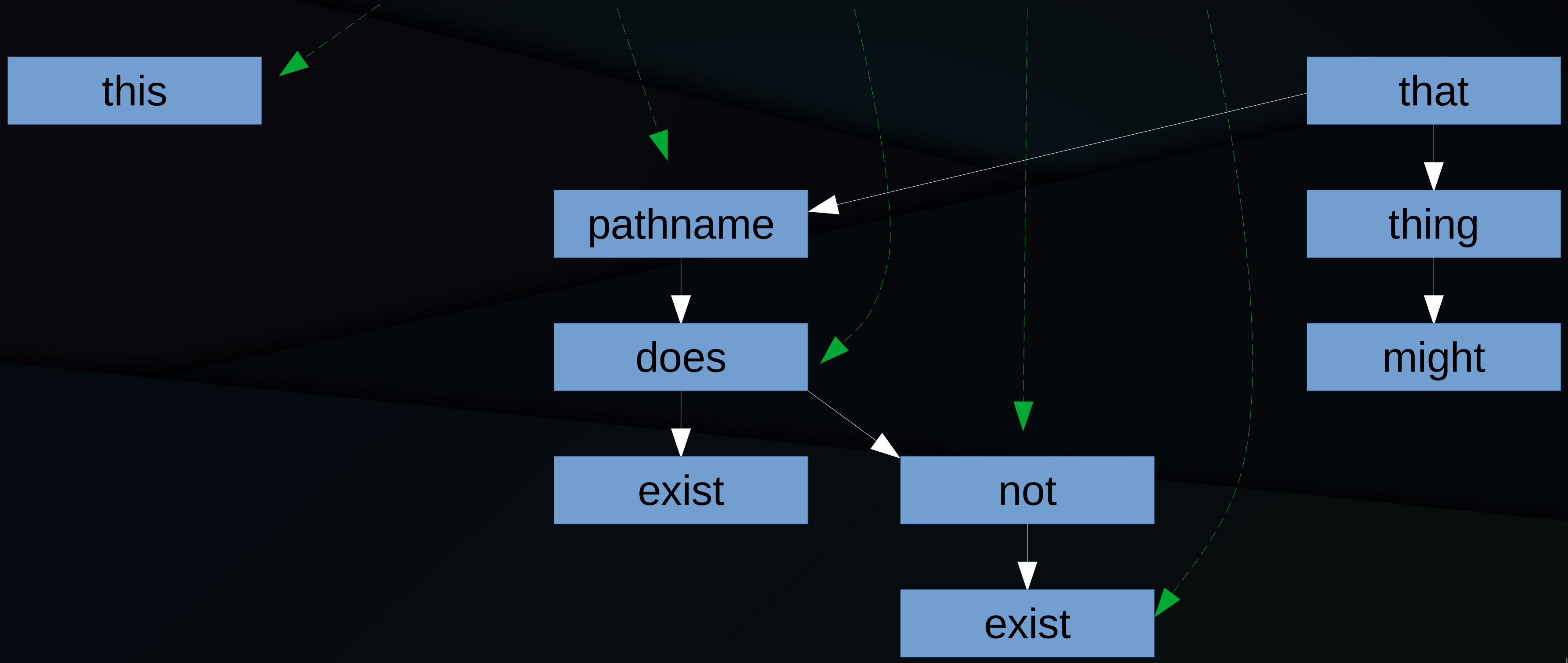
Pathname Lookup and Renames

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



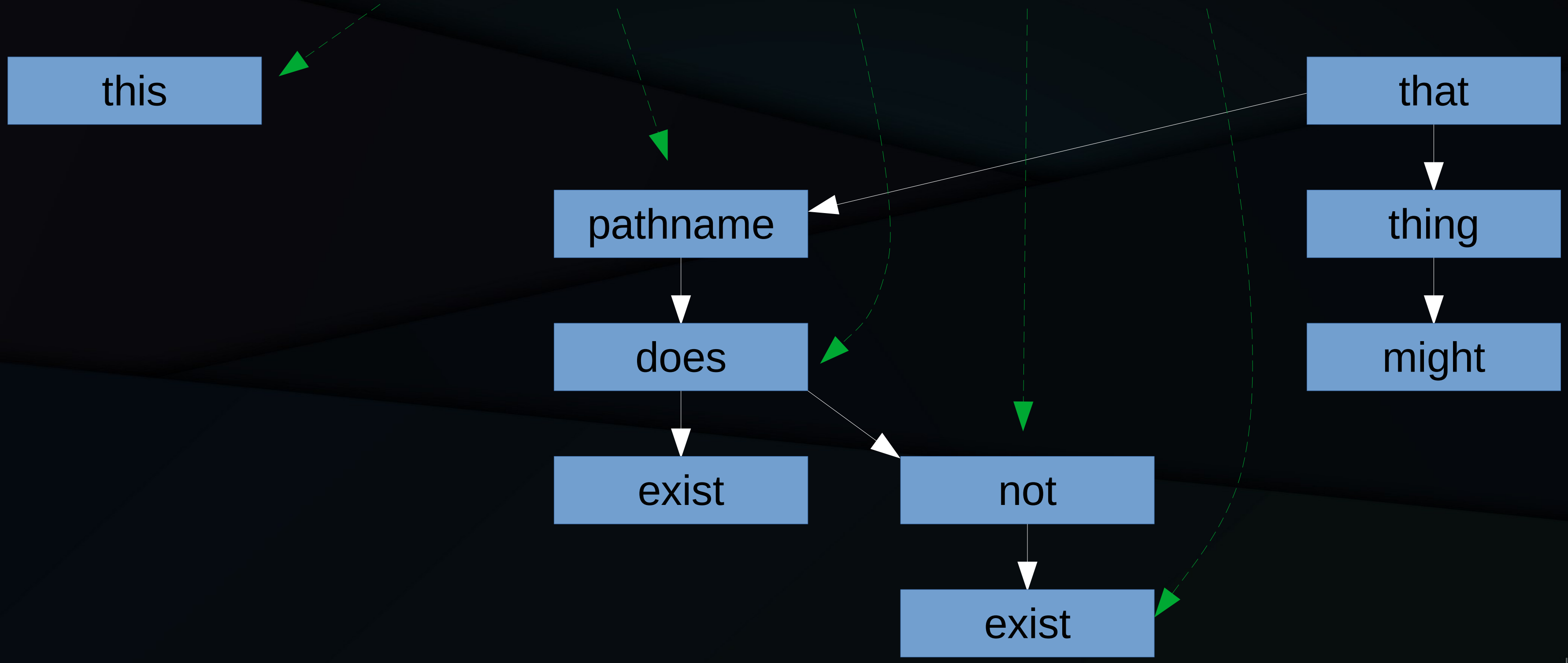
Pathname Lookup and Renames

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

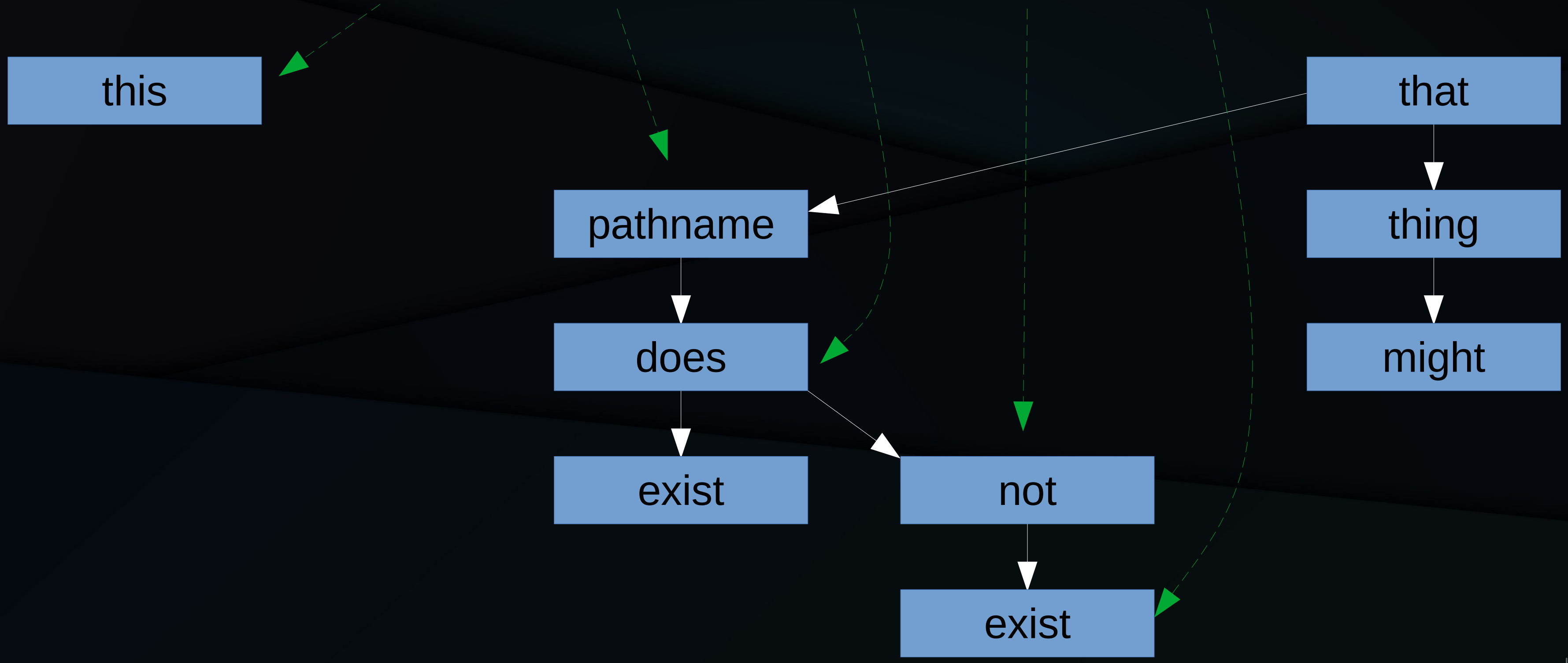
Brutally Simplified Pathwalk Code

```
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

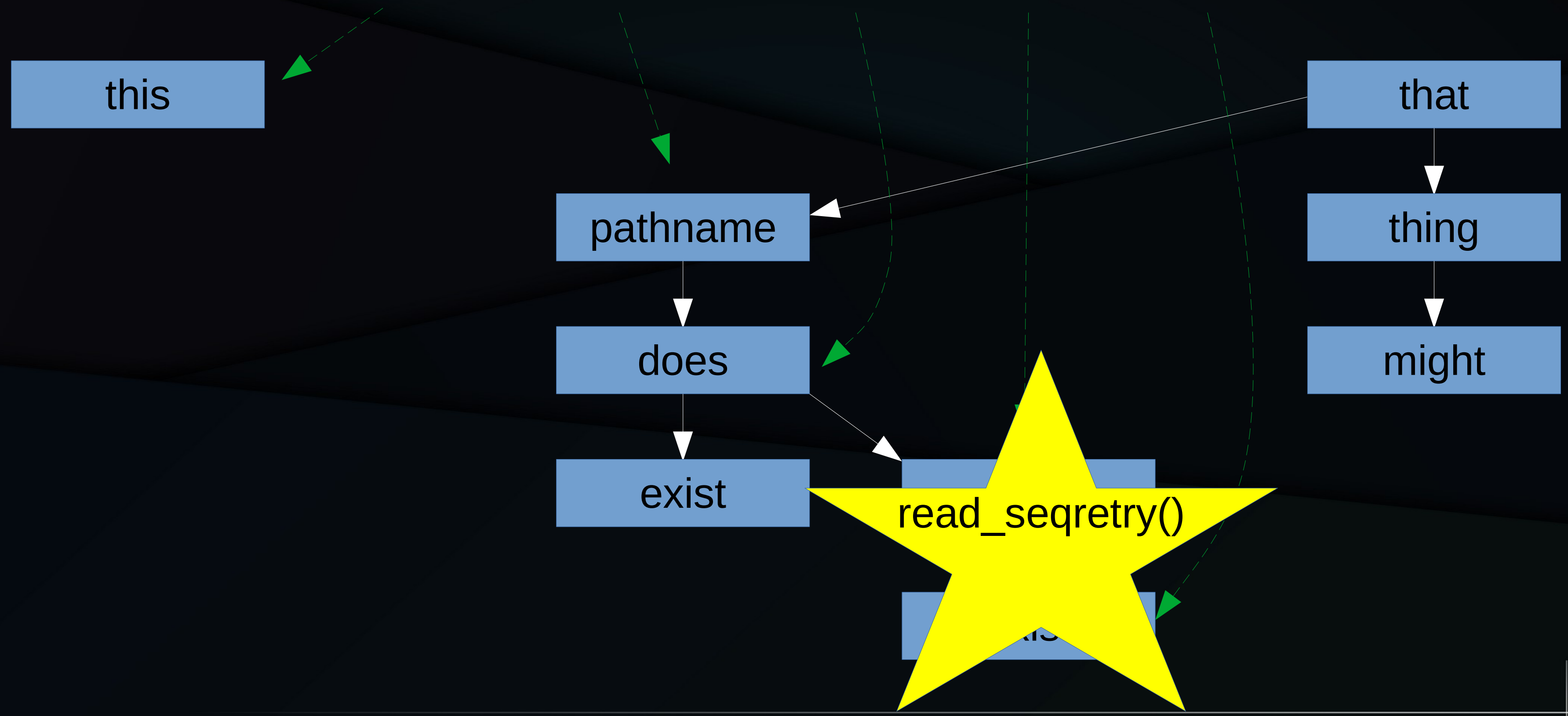
Looking up: “`/this/pathname/does/not/exist`”



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

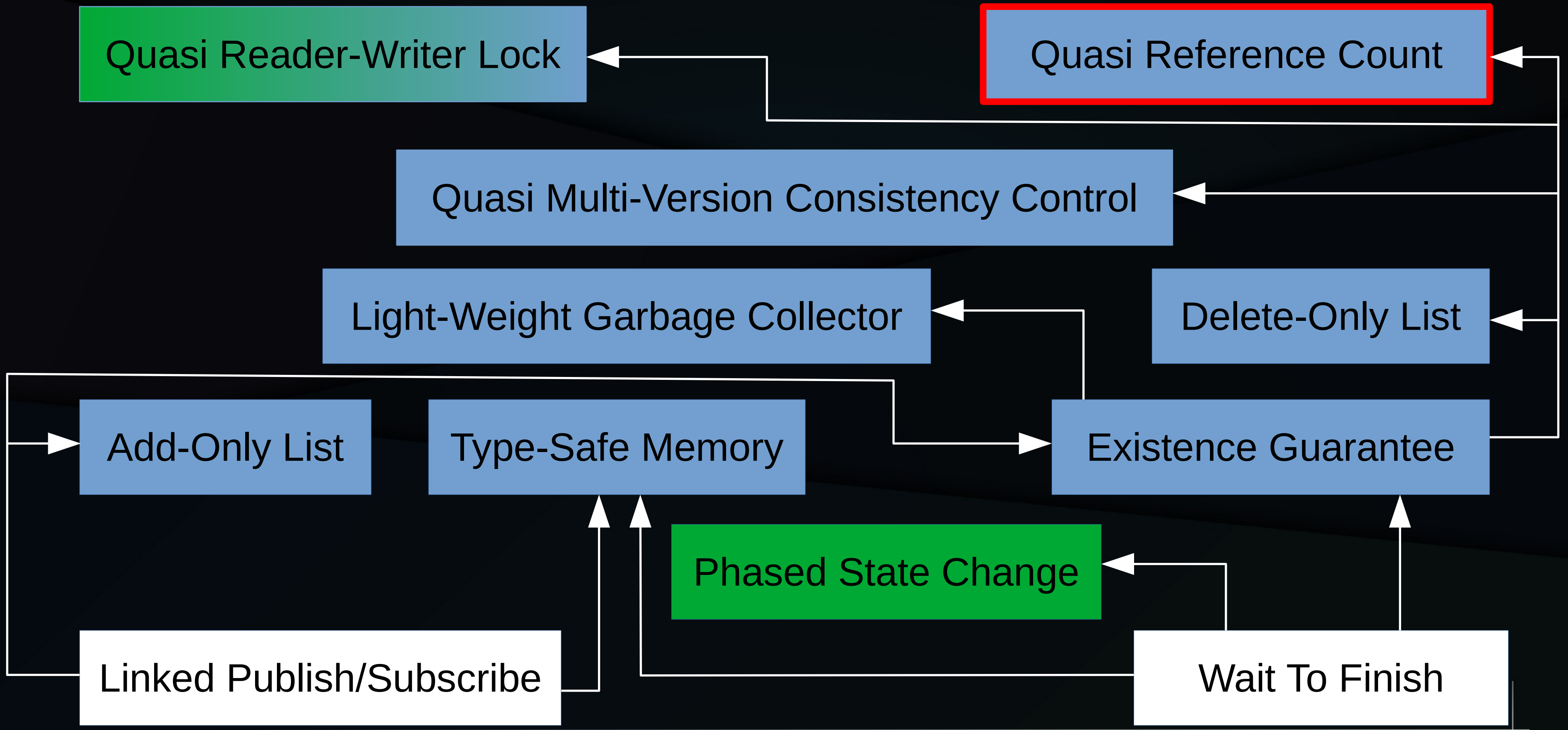
Kudos: Manish Soni, Nick Piggin, Al Viro. Christoph Hellwig. Neil Brown.

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

You Are Here: 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)

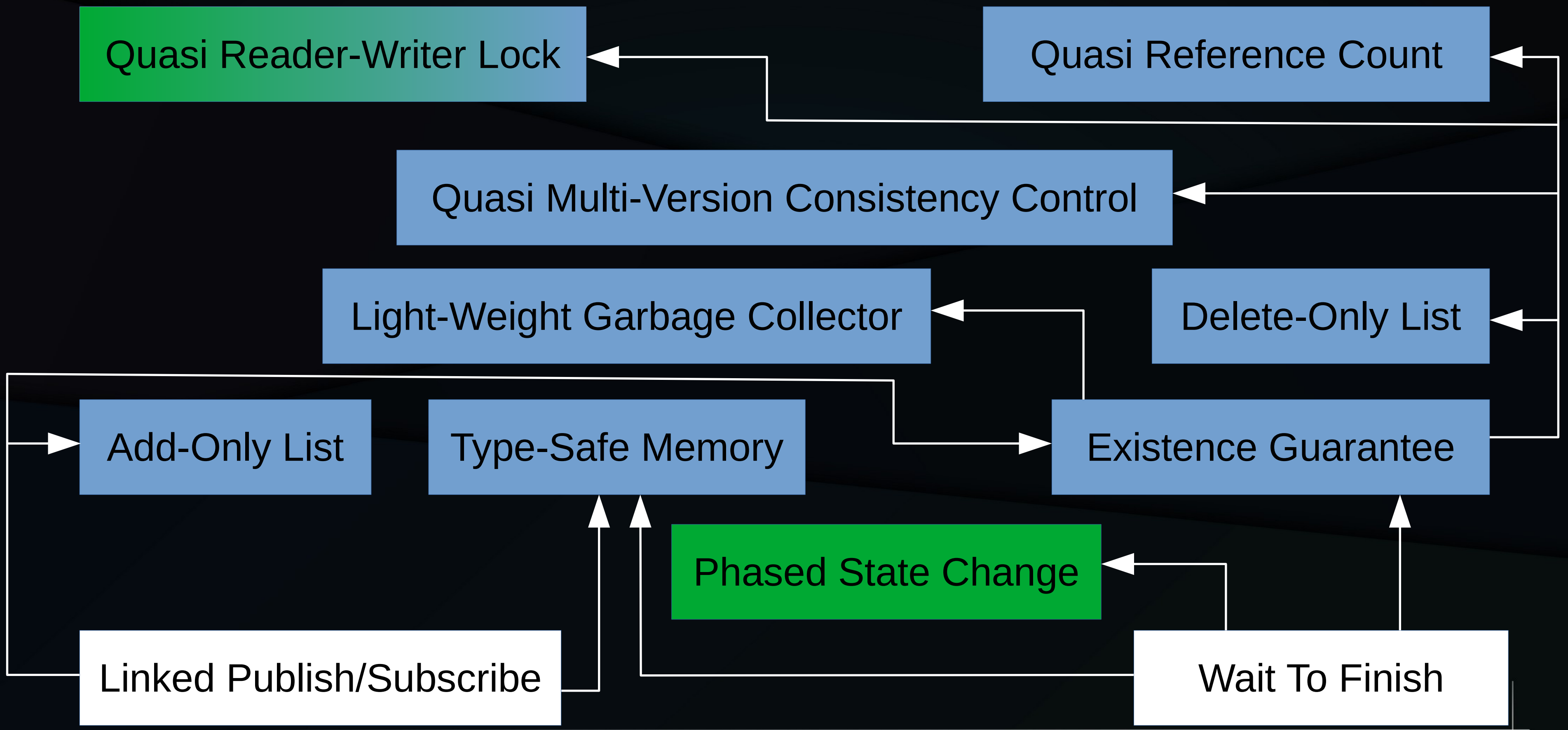
- You have already seen it!
  - Many of the earlier examples can be interpreted as quasi reference counting
- How can the same code be existence locking, quasi reader-writer locking, … ???
- What does `atomic_inc()` do?
  - Lots of things!!! Just like RCU!

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

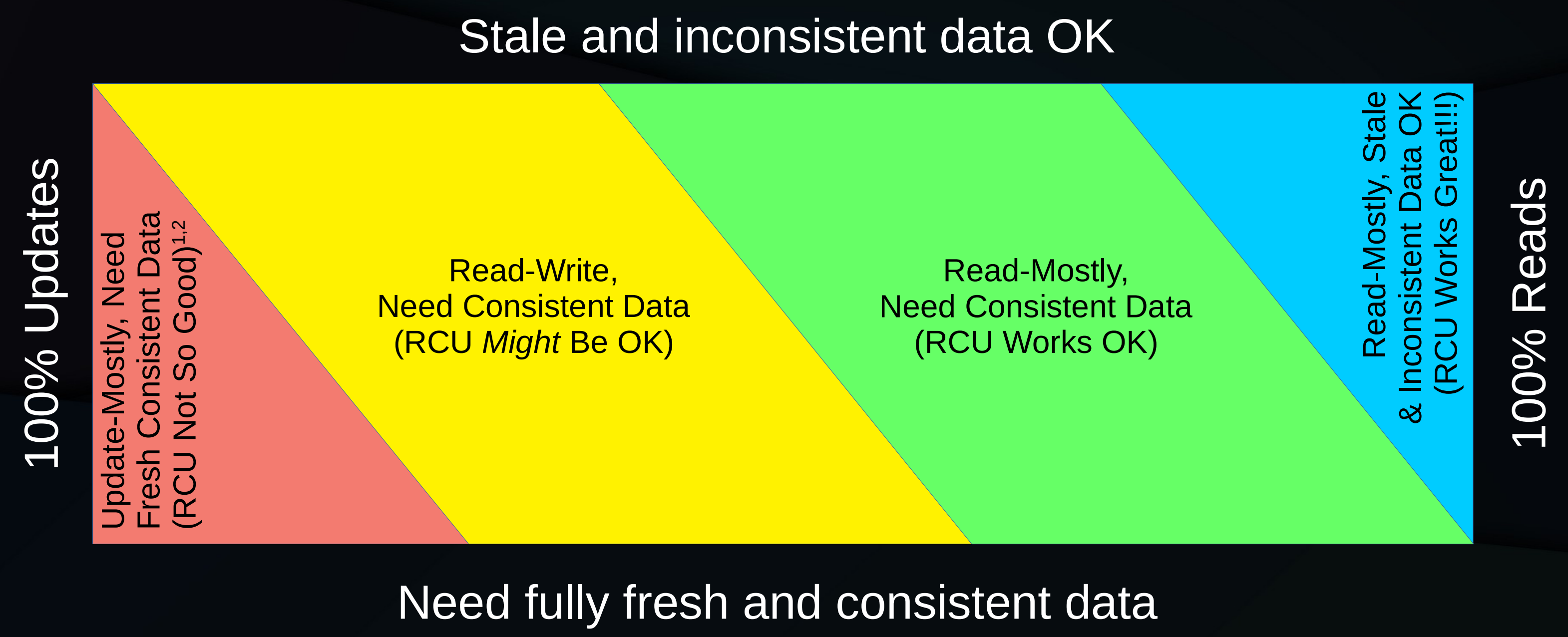
You Are Here

You Are Here



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

RCU Area of Applicability (Redux)



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

Summary

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

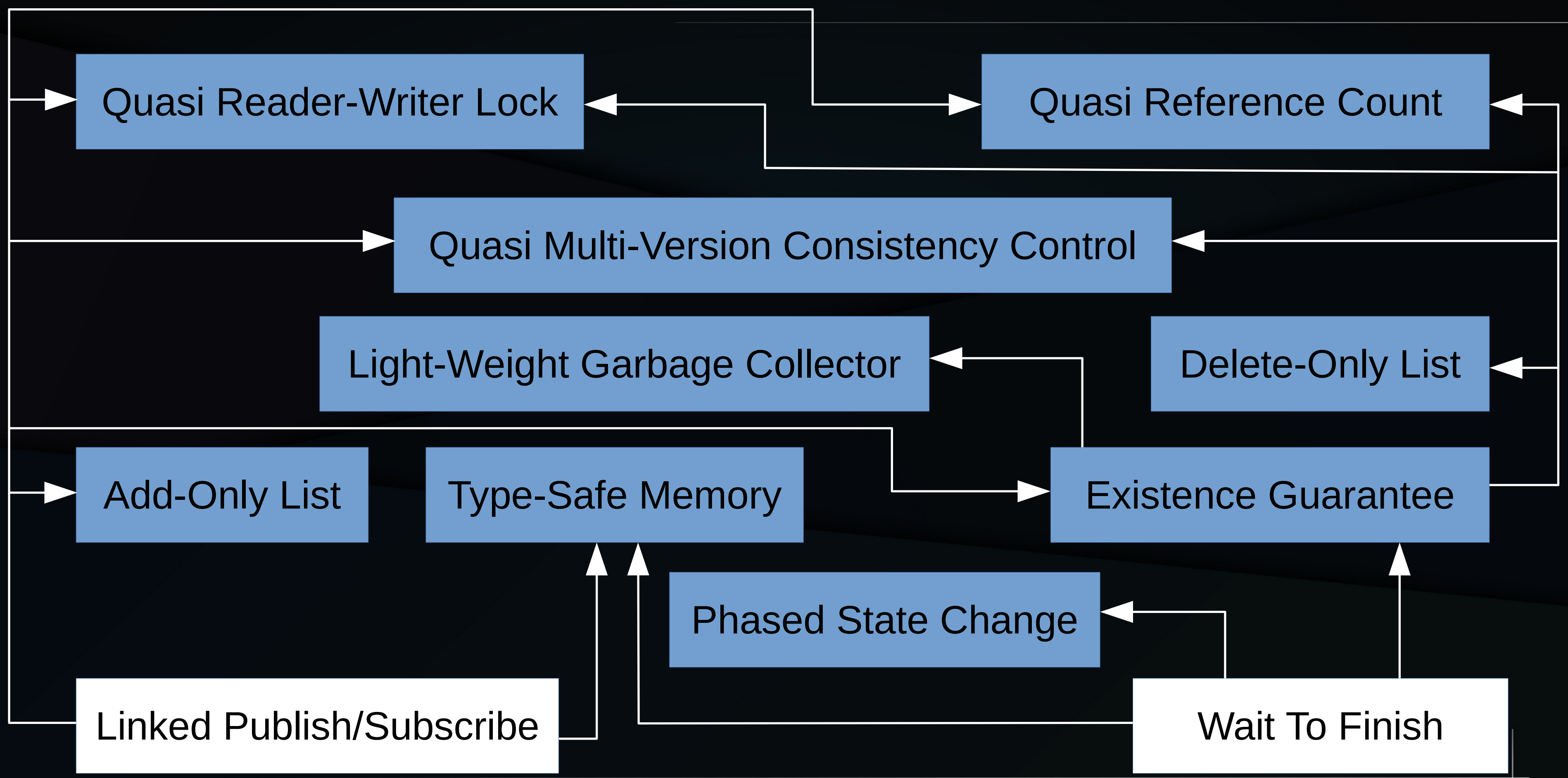
Summary

- 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

- Part 1: https://www.linuxfoundation.org/webinars/unraveling-rcu-usage-mysteries/
- “RCU Usage In the Linux Kernel: One Decade Later”:
  - http://www.rdrop.com/~paulmck/techreports/survey.2012.09.17a.pdf
  - http://www.rdrop.com/~paulmck/techreports/RCUUsage.2013.02.24a.pdf
  - 2020 update: https://dl.acm.org/doi/10.1145/3421473.3421481
- “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
- “Is Parallel Programming Hard, And, If So, What Can You Do About It?”, “Deferred Processing” chapter: https://mirrors.edge.kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html
- Folly-library RCU implementation (also C-language user-space RCU)
- Large piles of information: http://www.rdrop.com/~paulmck/RCU/