Paul E. McKenney, Facebook

Linux Foundation Live Mentorship Series, December 7, 2021

Unraveling RCU-Usage Mysteries

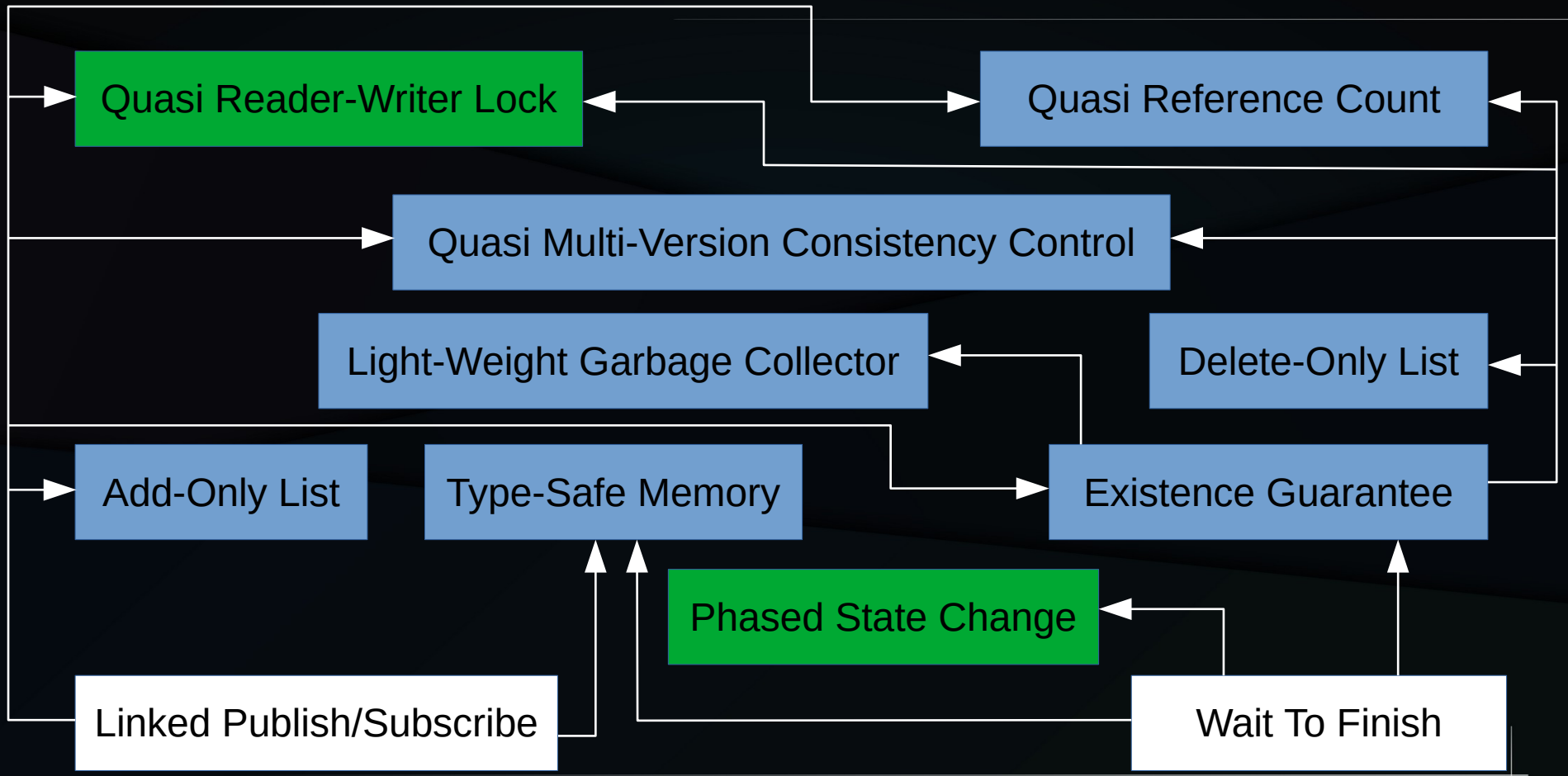
(Fundamentals)

RCU Usage: Overview

- You Are Here
- Example Multithreaded Use Case
- Reader-Writer Locked Use Case
- Read-Copy Update (RCU) Use Case
- RCU Usage in Linux Kernel
- Debugging Linux-Kernel RCU Code

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

Example Multithreaded Use Case

Hide

Example Multithreaded Use Case

- Configuration information in variables `a` and `b`:
  - `int a, b; // Current configuration values`
  - Infrequently updated based on external inputs
  - Given reader access needs consistent values
- Reading “Oldish” values OK, ***if*** consistent
- Very frequent reader access to `a` and `b`

Reader-Writer Locked Use Case

Reader-Writer Locked Use Case

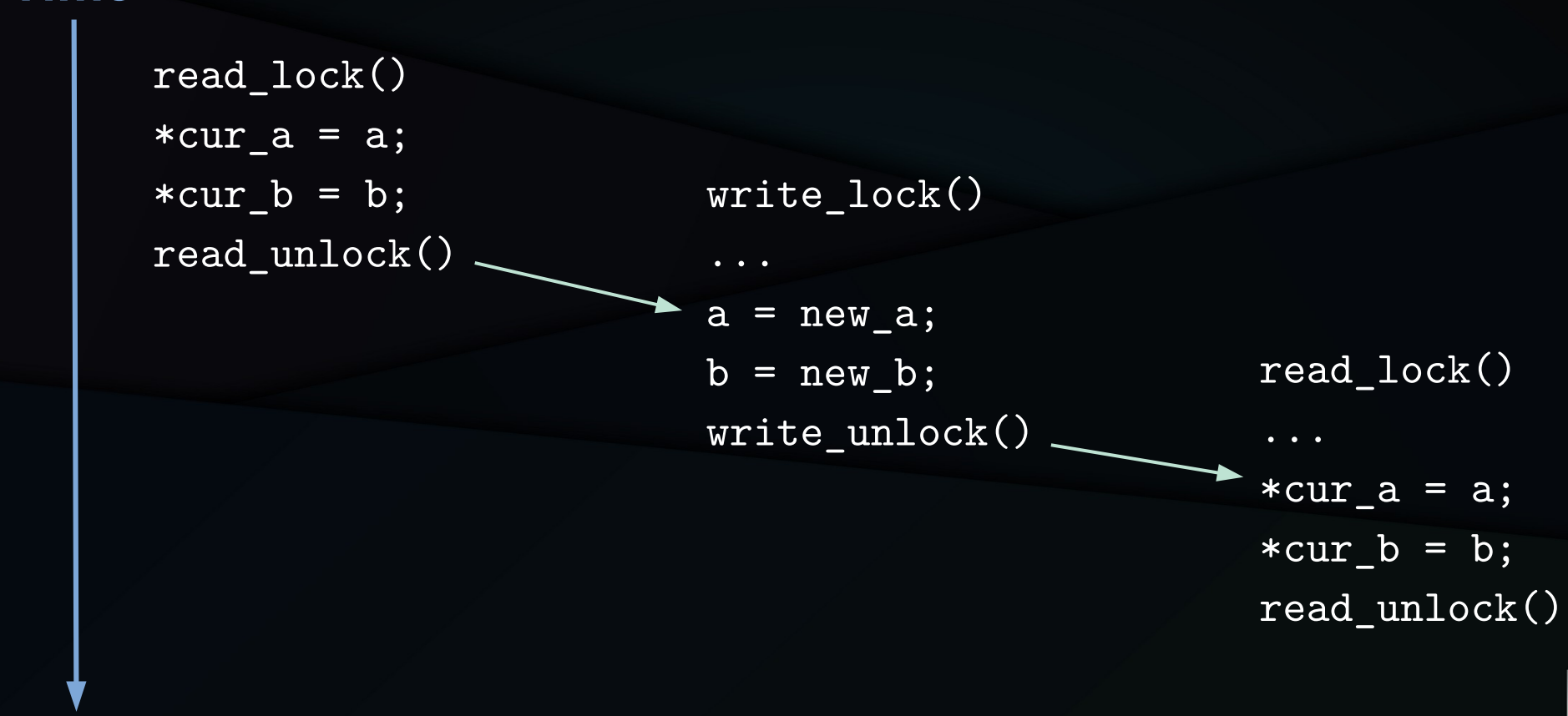
```
DEFINE_RWLOCK(myrwlock);
int a, b; // Current configuration values

void get(int *cur_a, int *cur_b)
{
    read_lock(&myrwlock);
    *cur_a = a;
    *cur_b = b;
    read_unlock(&myrwlock);
}
```

Reader-Writer Locked Use Case

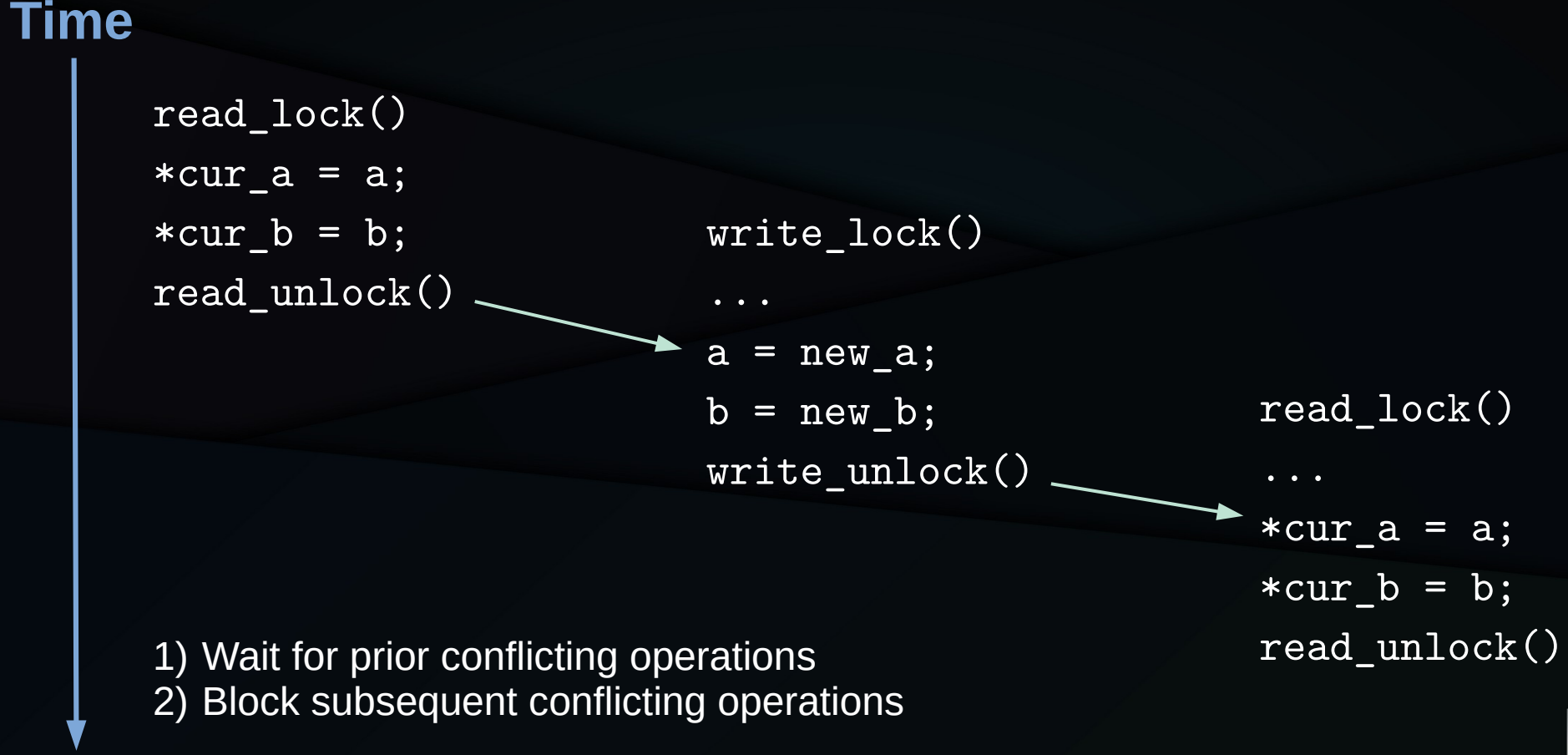
```
void set(int new_a, int new_b)
{
    write_lock(&myrwlock);
    a = new_a;
    b = new_b;
    write_unlock(&myrwlock);
}
```

Reader-Writer Lock Semantics

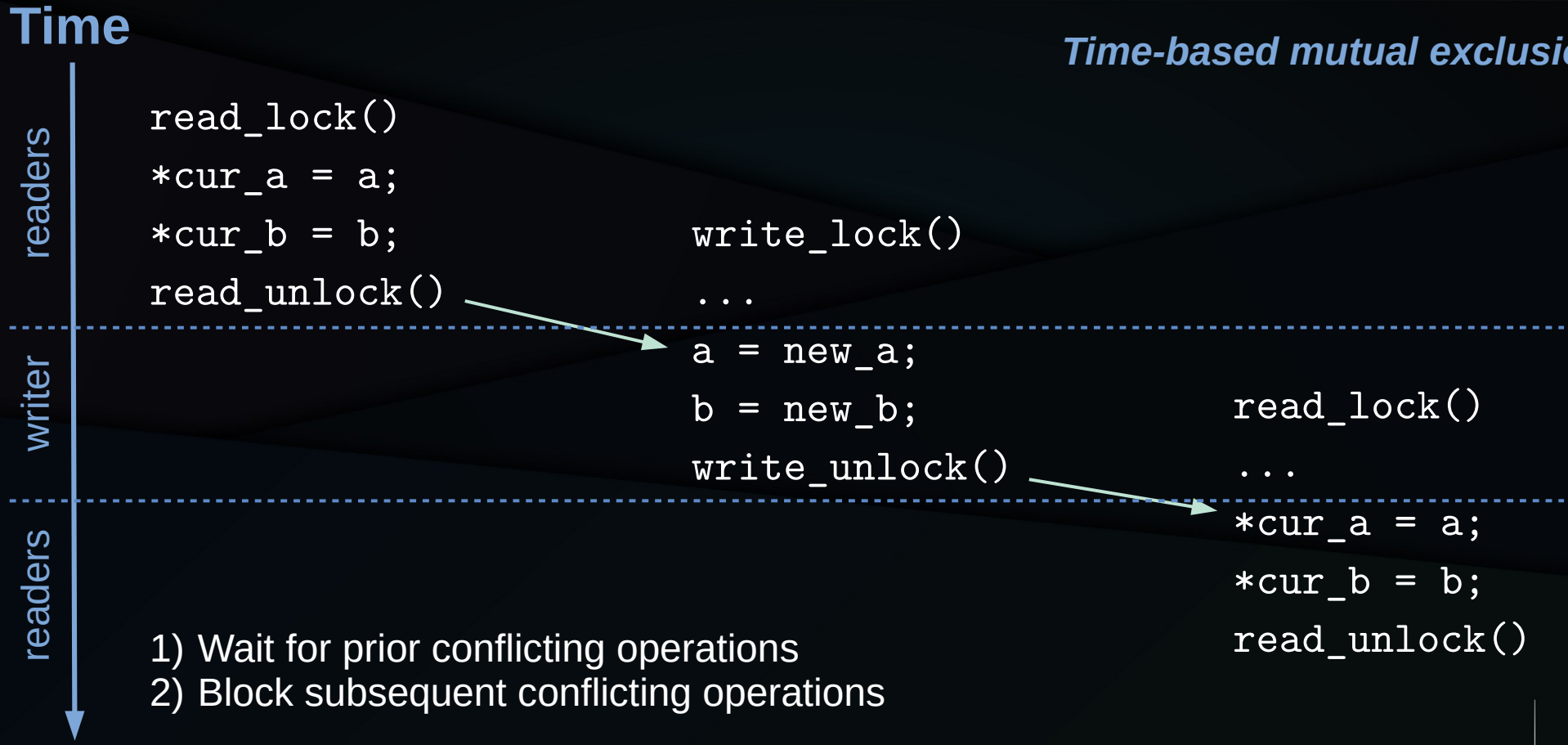
Reader-Writer Lock Semantics



Reader-Writer Lock Semantics



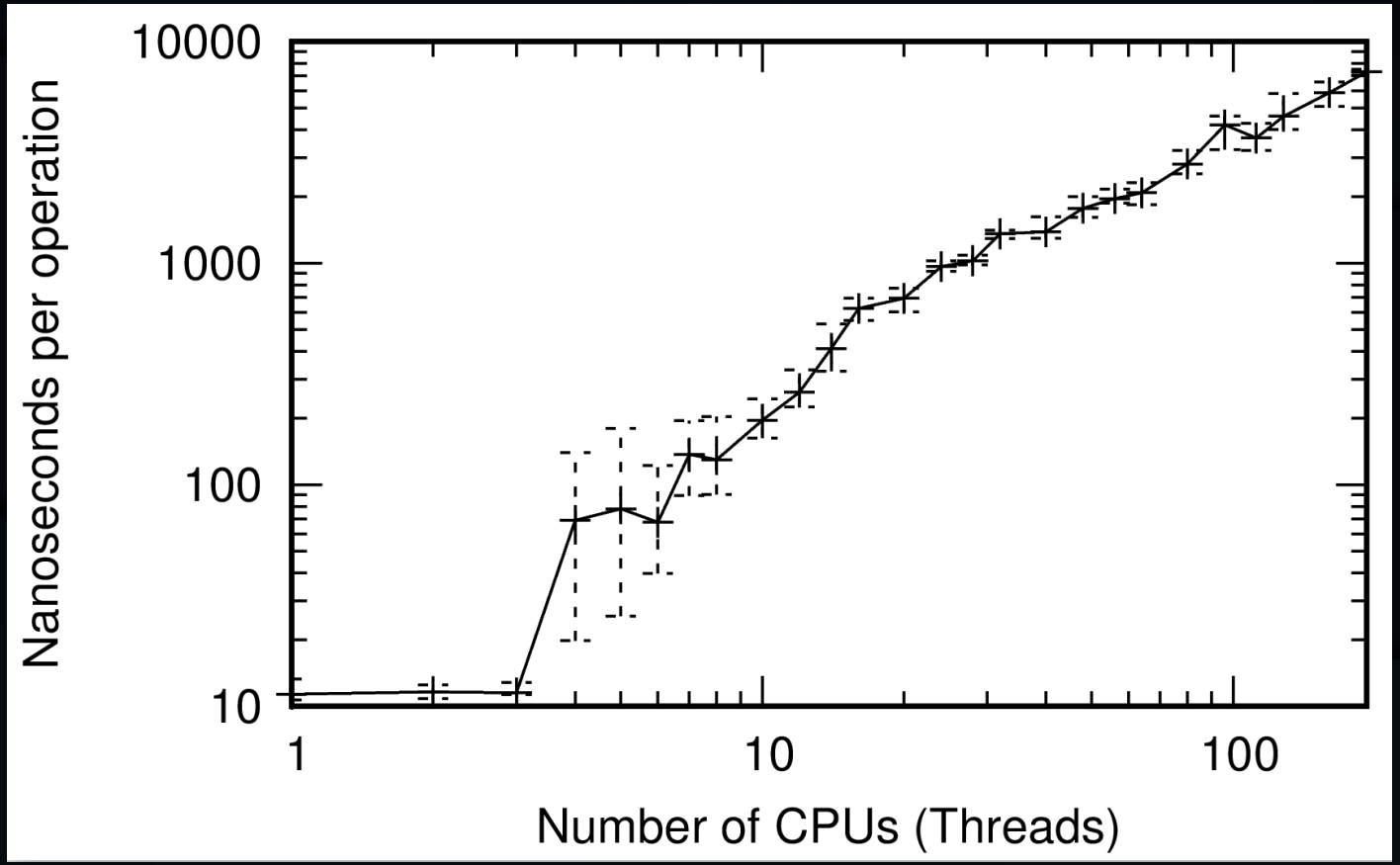
Reader-Writer Lock Semantics



Time-based mutual exclusion

Reader-Writer Lock Performance

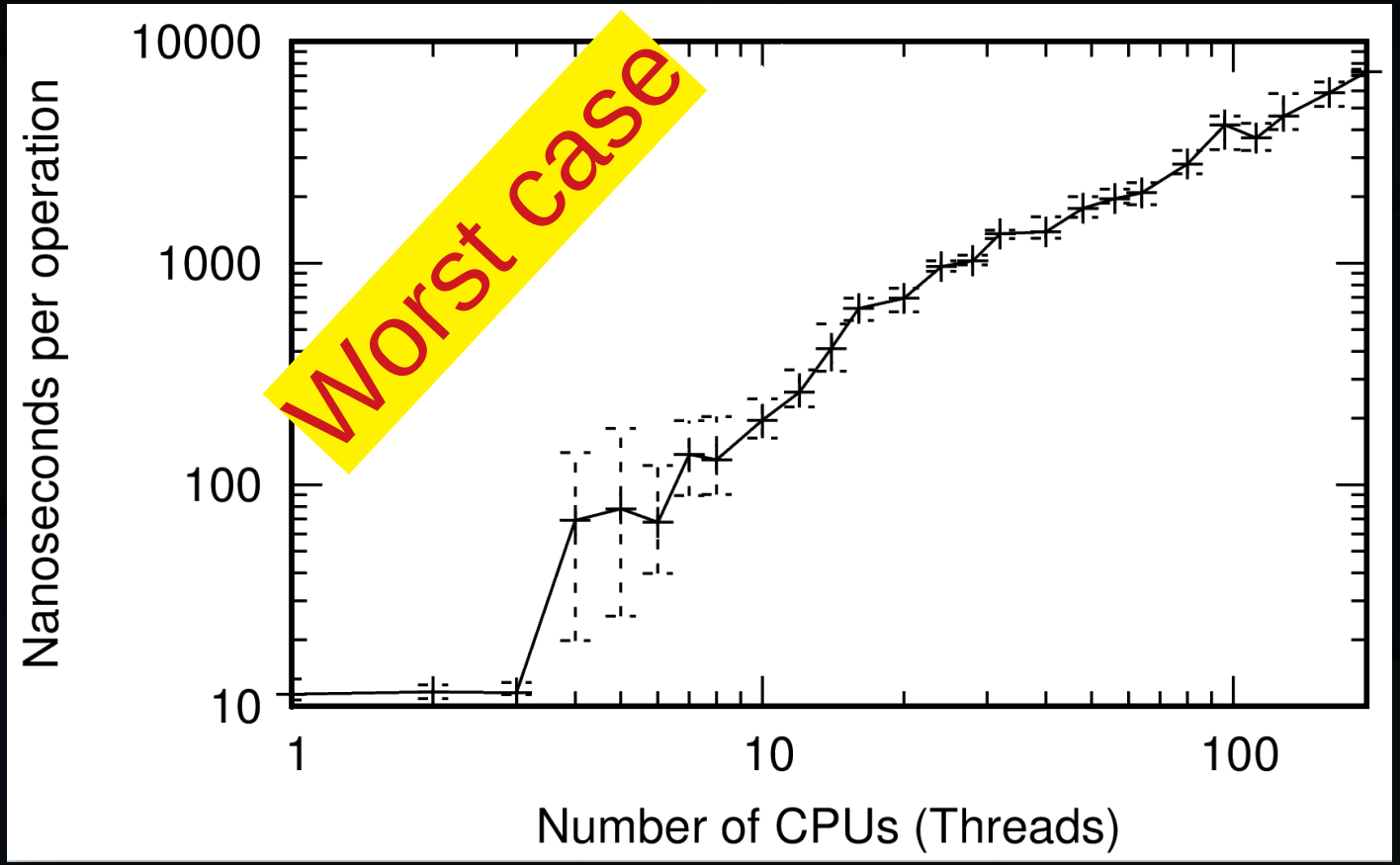
Reader-Writer Lock Performance



Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz

perfbook CodeSamples/defer/data/rcuscale.hps.2020.05.28a/rwlockperf.eps

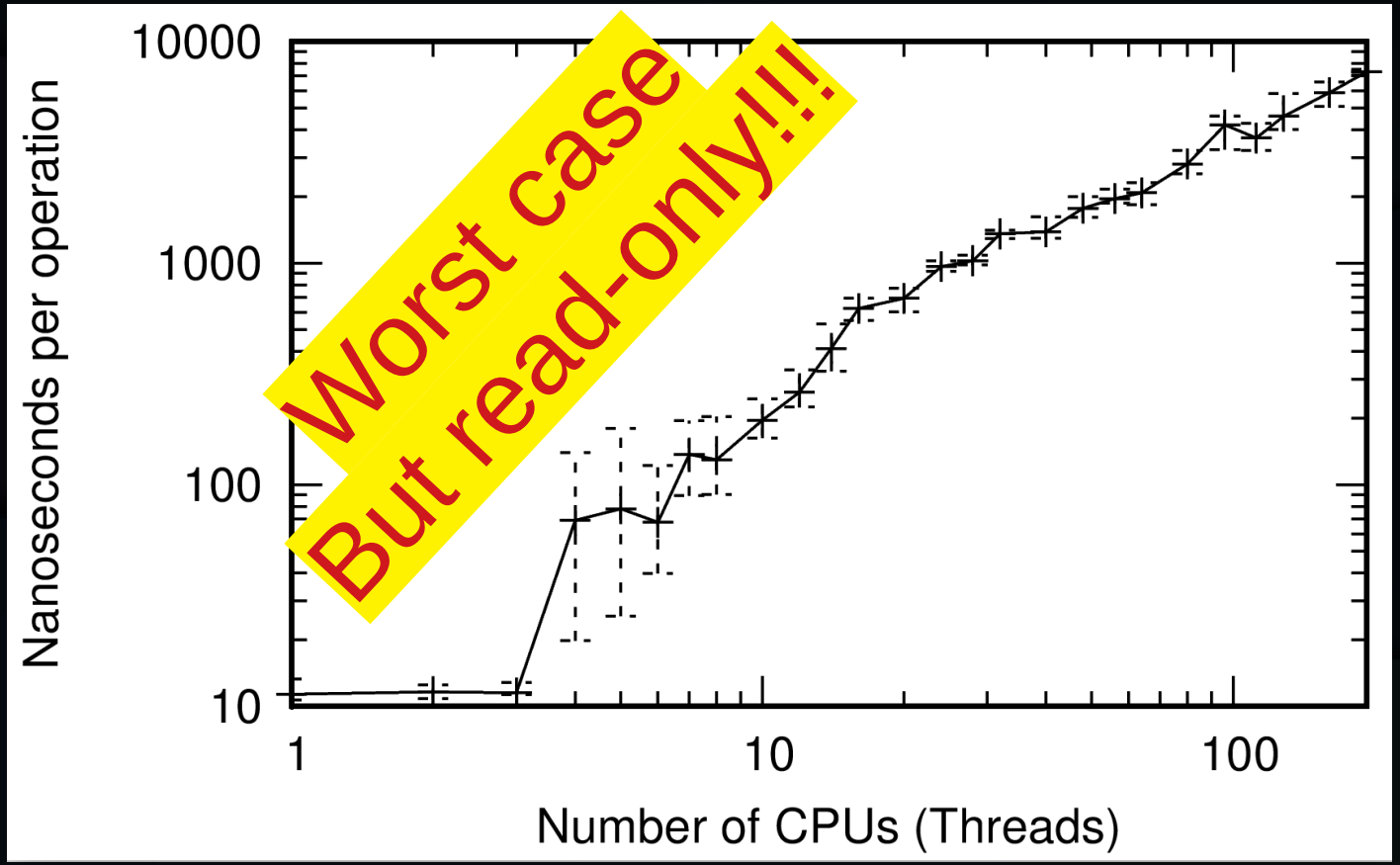
Reader-Writer Lock Performance



Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz

perfbook CodeSamples/defer/data/rcuscale.hps.2020.05.28a/rwlockperf.eps

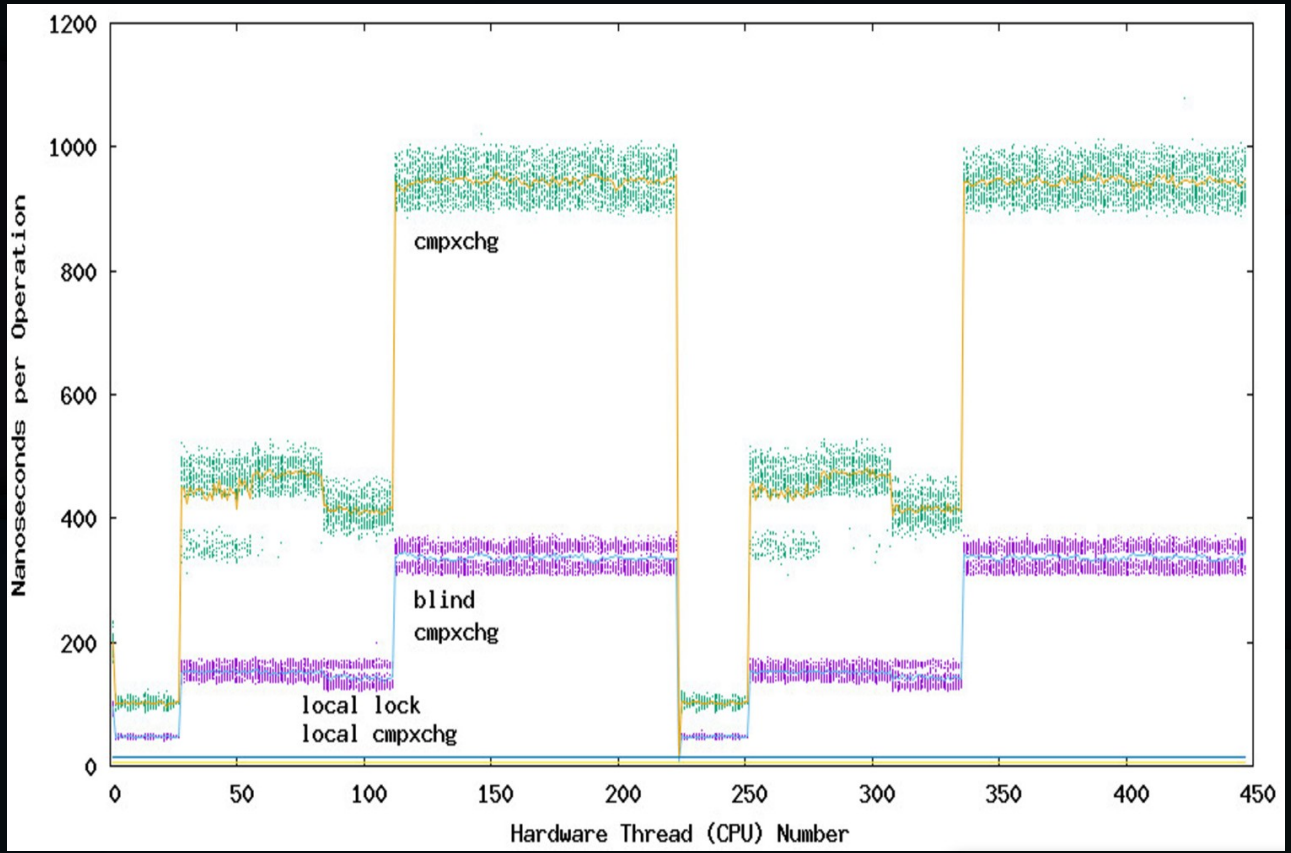
Reader-Writer Lock Performance



Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz

perfbook CodeSamples/defer/data/rcuscale.hps.2020.05.28a/rwlockperf.eps

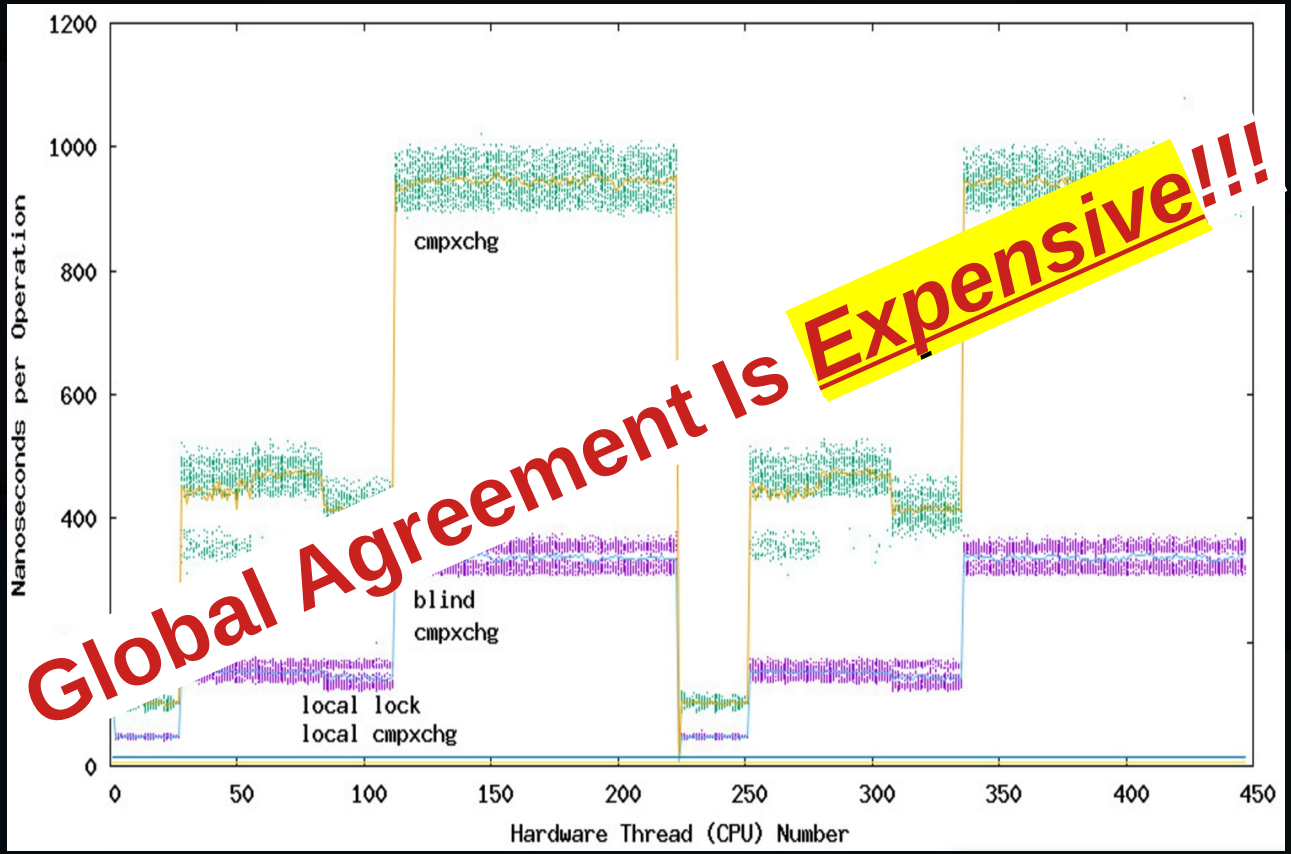
Not Just Contention: HW Latency



Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz

perfbook CodeSamples/cpu/data/hps.2020.03.04b/cachetorture-latency-scatter.jpg

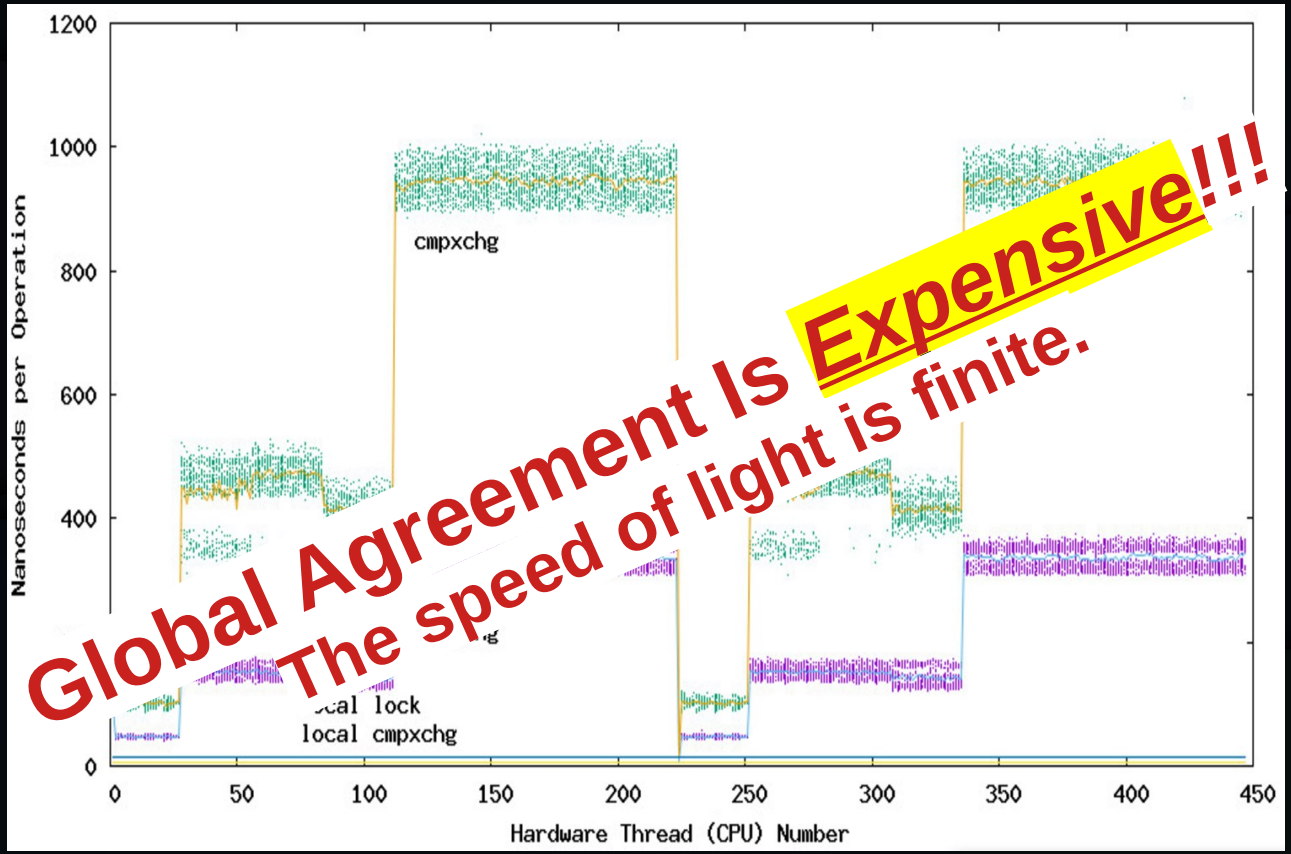
Not Just Contention: HW Latency



Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz

perfbook CodeSamples/cpu/data/hps.2020.03.04b/cachetorture-latency-scatter.jpg

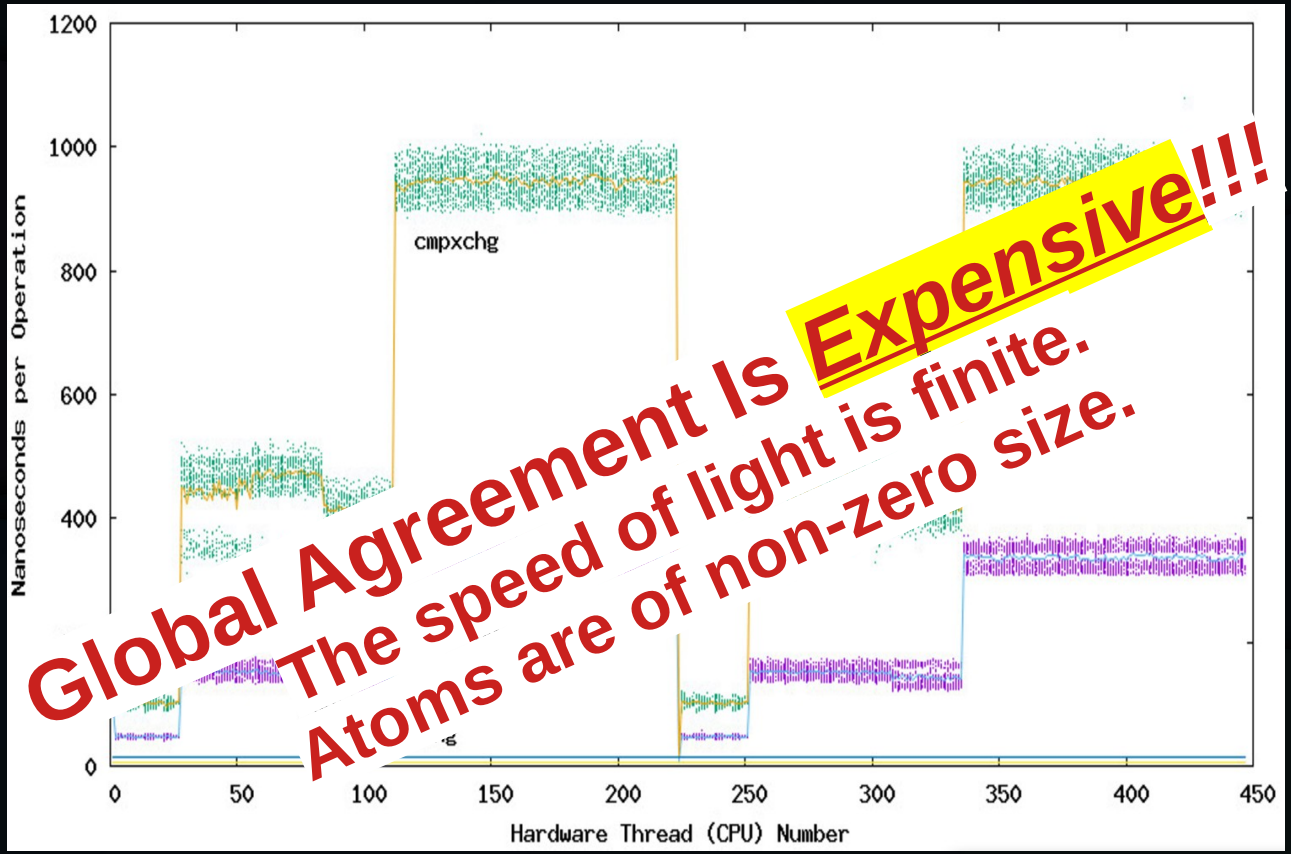
Not Just Contention: HW Latency



Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz

perfbook CodeSamples/cpu/data/hps.2020.03.04b/cachetorture-latency-scatter.jpg

Not Just Contention: HW Latency



Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz

perfbook CodeSamples/cpu/data/hps.2020.03.04b/cachetorture-latency-scatter.jpg

Respect the Laws of Physics!!!

- Following the footsteps of Admiral Hopper:
  - Light goes 11.803 inches/ns in a vacuum
    - Or, if you prefer, 29.979 cm/ns
  - But over and back: 5.9015 in/ns
  - But 4GHz: 1.4754 in/ns
  - But Cu: ~0.5 in/ns, or Si transistors: ~0.05 in/ns
  - Plus other slowdowns: prototols, electronics, ...

But Laws Have Loopholes...

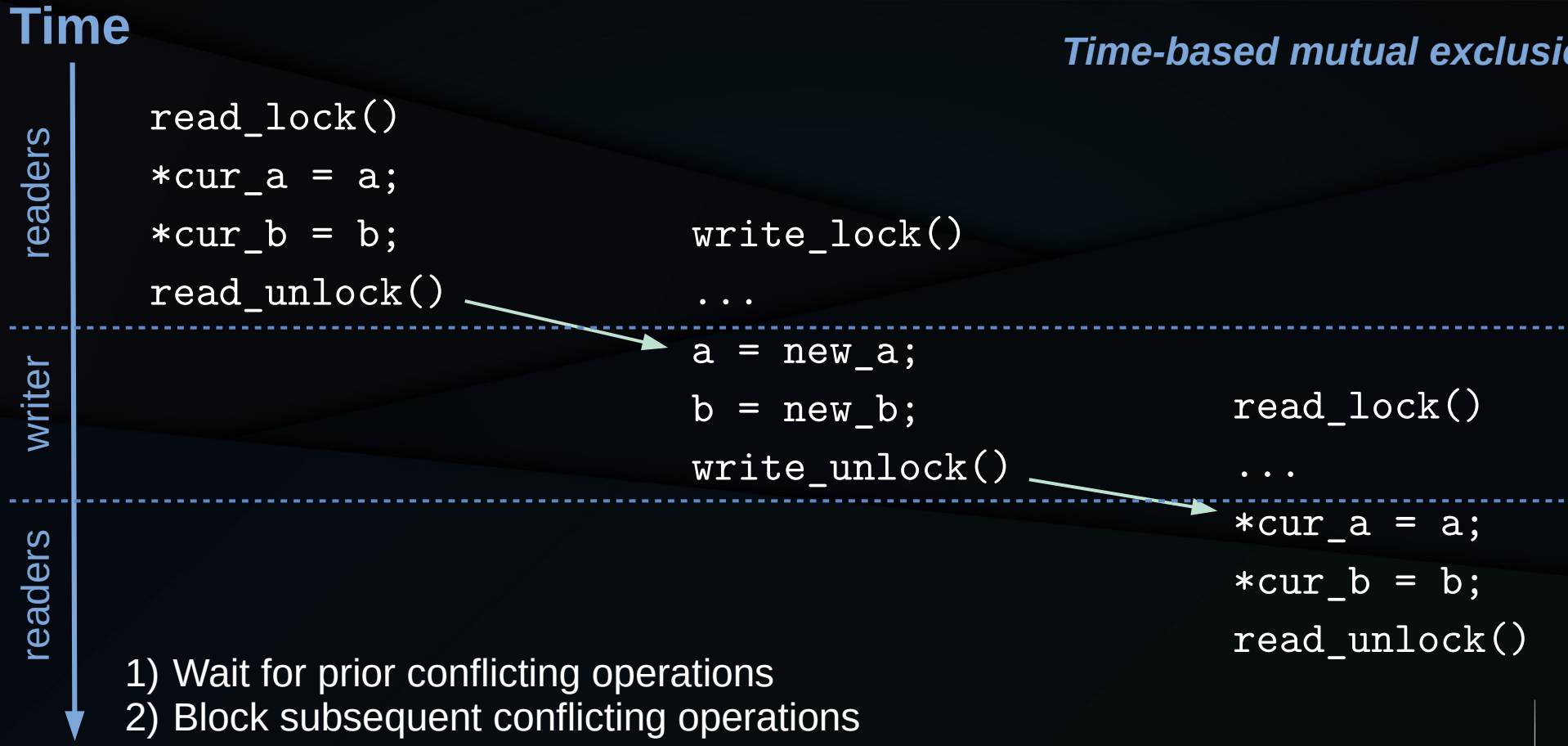
- 3D integration for smaller systems
- Systolic arrays for unidirectional data flow
  - Obtain similar effects with modern accelerators
- New semiconductor technologies
  - Too bad about the need to hand-place atoms...

But Laws Have Loopholes...

- 3D integration for smaller systems
- Systolic arrays for unidirectional data flow
  - Obtain similar effects with modern accelerators
- New semiconductor technologies
  - Too bad about the need to hand-place atoms...
- **Read-only replication**

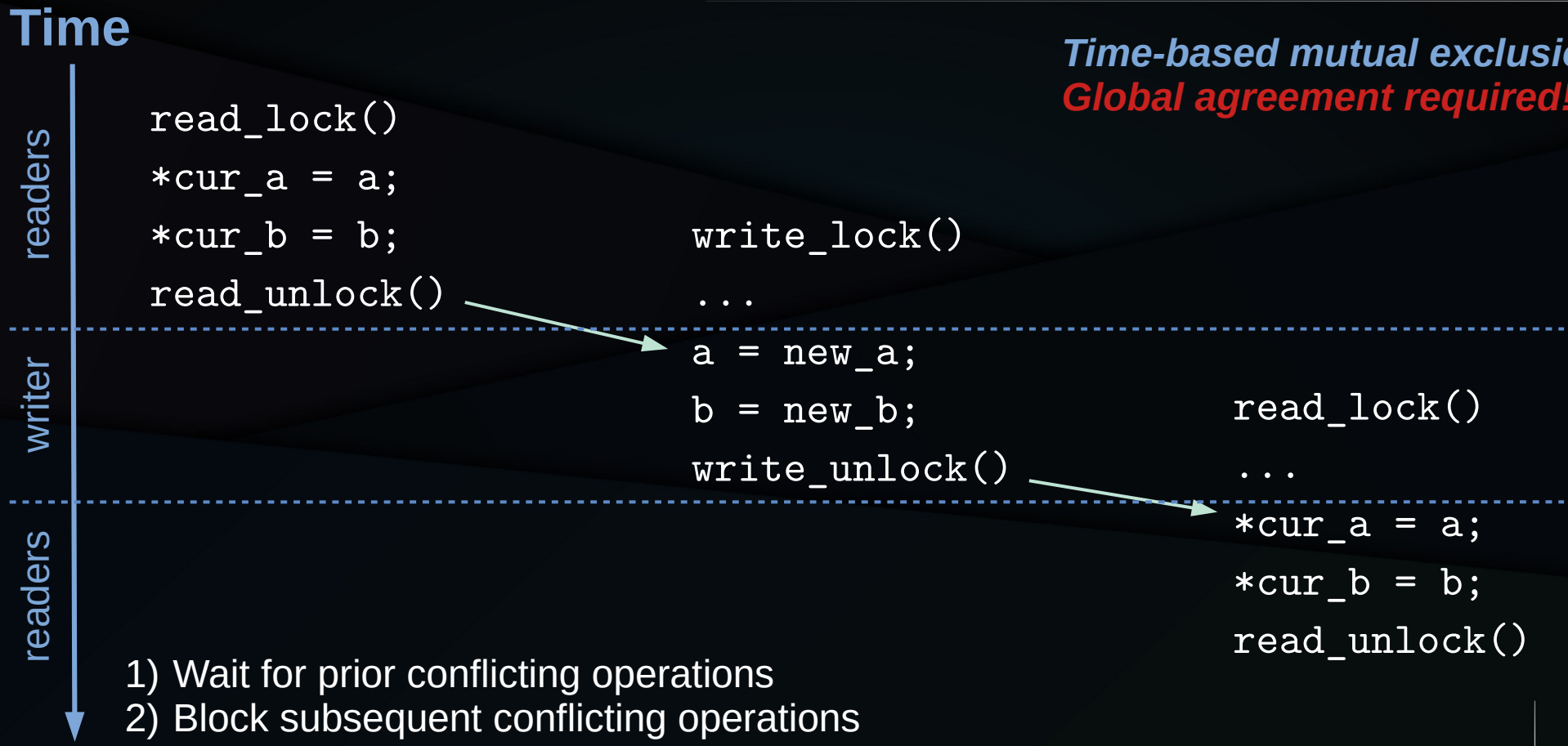
Lighter Weight Semantics

Reader-Writer Lock Semantics (Redux)



Time-based mutual exclusion

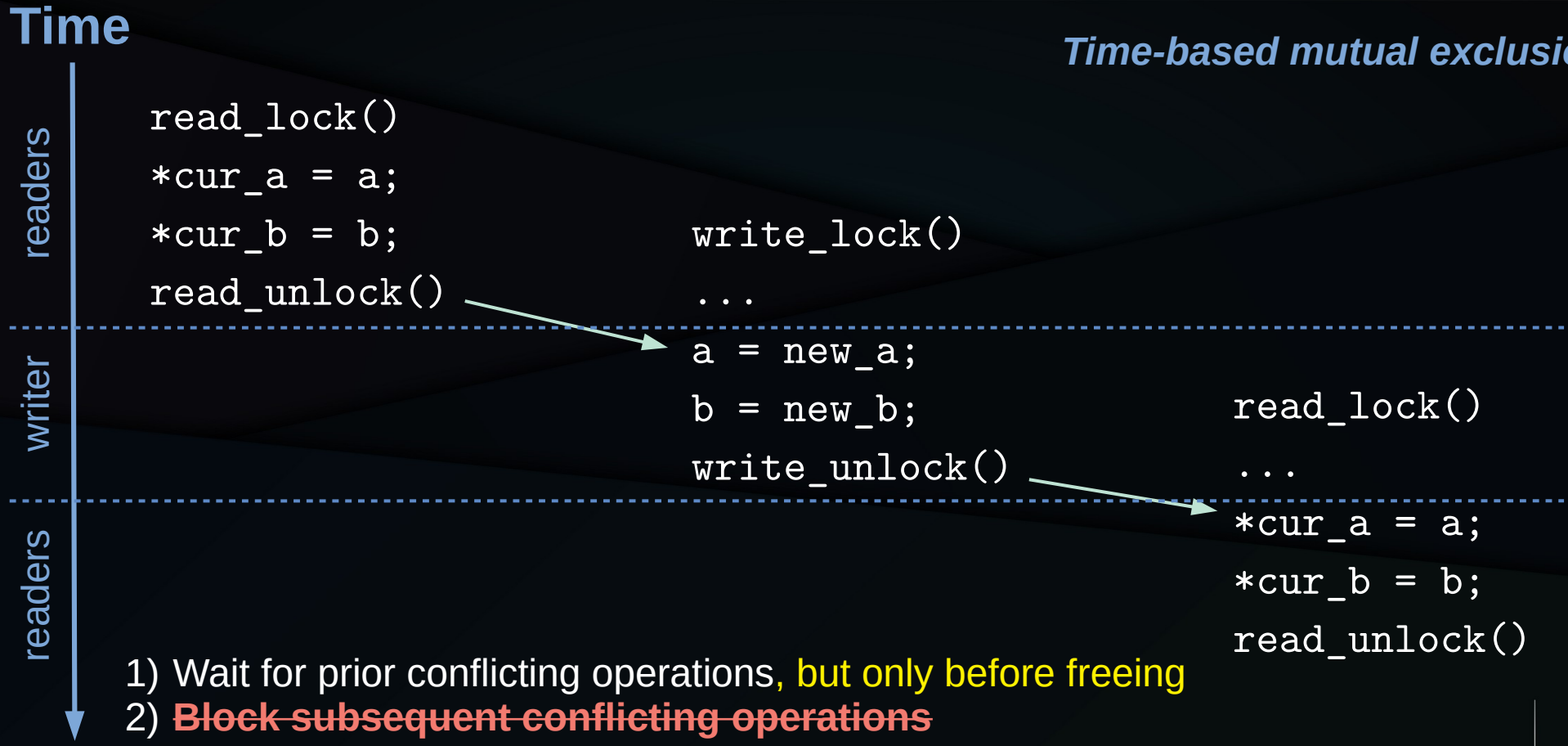
Reader-Writer Lock Semantics (Redux)



Time-based mutual exclusion:
Global agreement required!!!

1) Wait for prior conflicting operations
2) Block subsequent conflicting operations

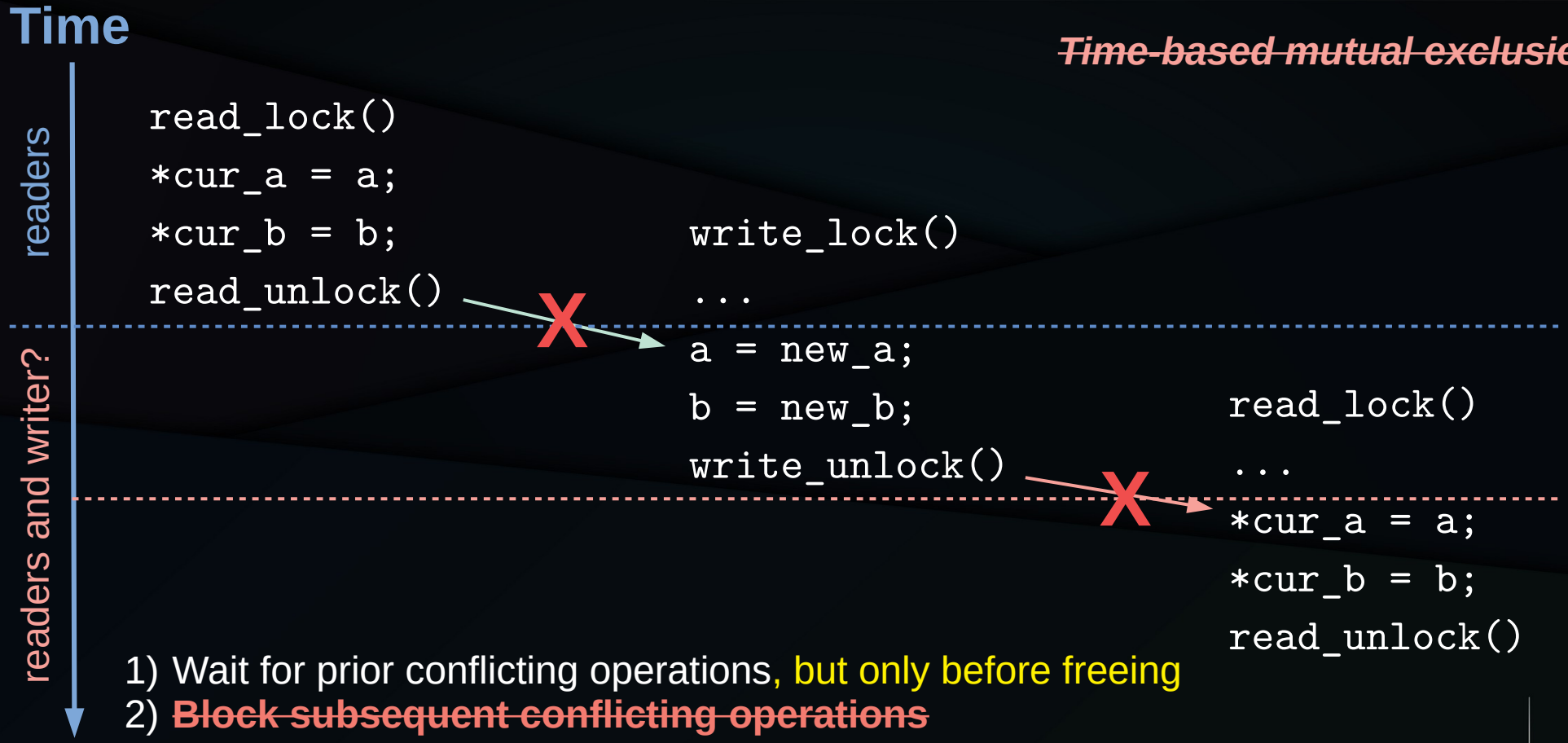
Lighter Weight Semantics



Time-based mutual exclusion

1) Wait for prior conflicting operations, but only before freeing
2) ~~Block subsequent conflicting operations~~

Lighter Weight Semantics???



~~*Time-based mutual exclusion*~~

1) Wait for prior conflicting operations, but only before freeing
2) ~~**Block subsequent conflicting operations**~~

Core RCU API

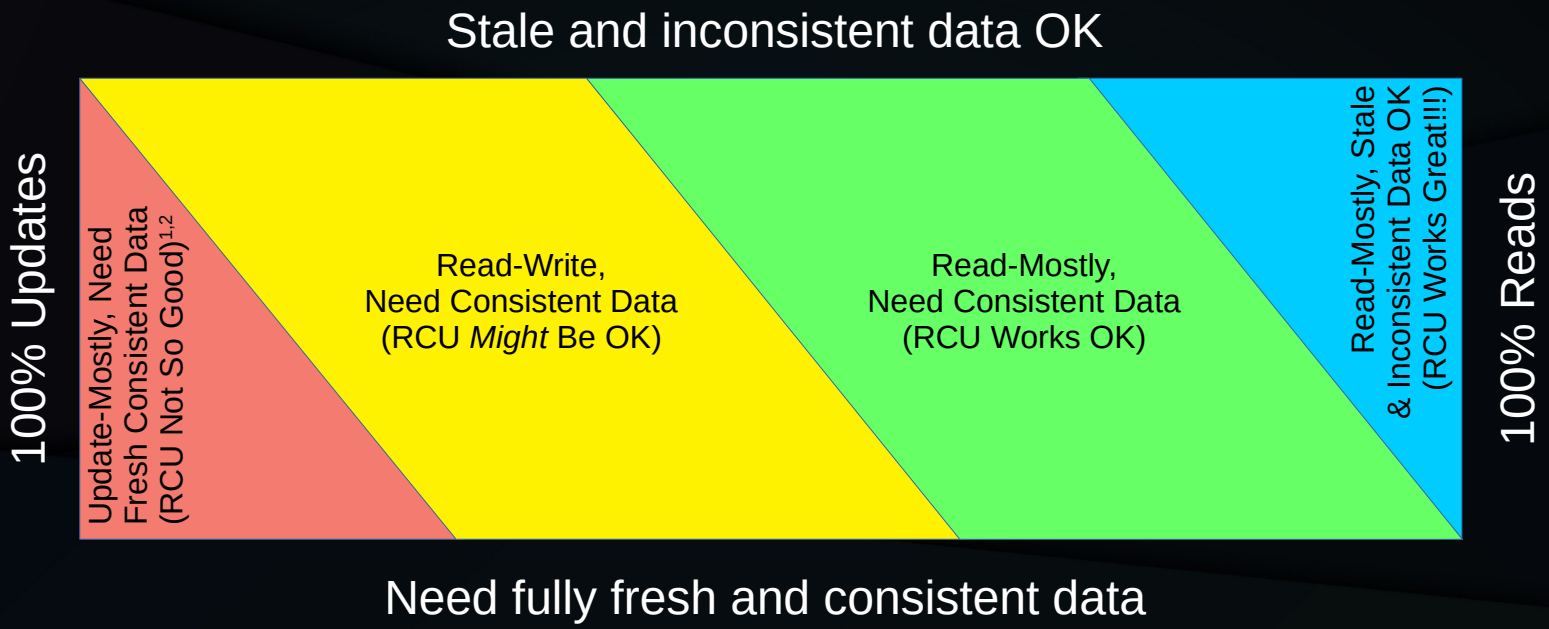
- `rcu_read_lock()`: Begin reader, sort of like read_lock()
- `rcu_read_unlock()`: End reader, sort of like read_unlock()
- `synchronize_rcu()`: Wait for pre-existing readers
  - Sort of like write_lock() immediately followed by write_unlock()
- `call_rcu()`: Invoke function after pre-existing readers 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 (English)

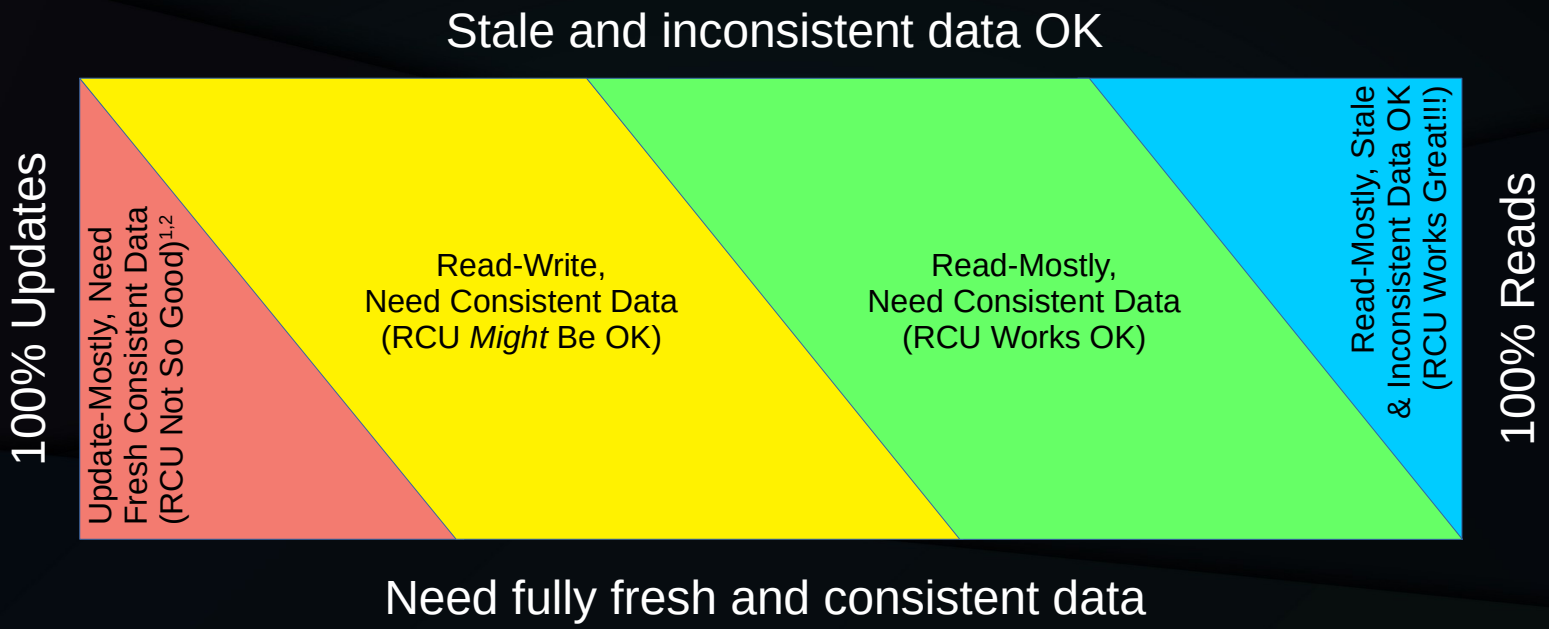
- Semantics weakened from reader-writer locking:
  1) Writers wait for read-holders, but only before freeing
  2) ~~Readers wait for the writer-holders~~
- Compensate for weak temporal semantics by adding *restrictions and spatial semantics* to RCU use cases
- Restated for RCU:
  - If synchronize_rcu() cannot prove that it started before a given rcu_read_lock(), it must not return to its caller until the matching rcu_read_unlock() completes

RCU Semantics (Restrictions)



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

RCU Semantics (Restrictions)



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

And RCU is most frequently used for linked data structures.

RCU Semantics (Show Me The Code)

Example Application (Redux)

- Configuration information in variables `a` and `b`:
  - `int a, b; // Current configuration values`
  - Infrequently updated based on external inputs
  - Given reader access needs consistent values
- Reading “Oldish” values OK, ***if*** consistent
- Very frequent reader access to `a` and `b`

Design of RCU Use Case

- Put a & b into a structure to obtain consistency
  - “All problems in computer science can be solved by another level of indirection.”

    David Wheeler
- Update: Allocate new structure & update pointer
- Free old structure “when it is safe to do so”

Core RCU API (Redux)

- `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/

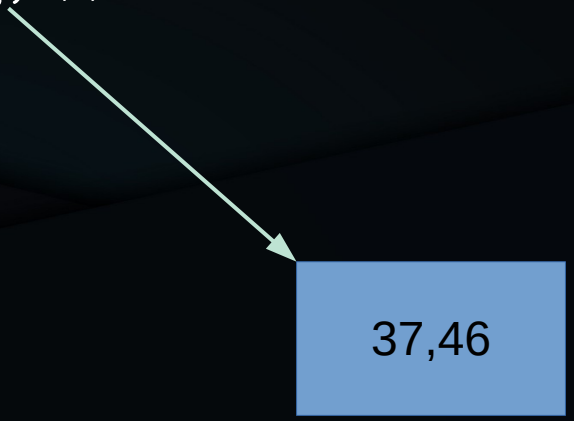
RCU Use Case: Reader

RCU Use Case: Reader

```
struct myconfig { int a, b; } *curconfig; // Initialized

void get(int *cur_a, int *cur_b)
{
    struct myconfig *mcp;

    rcu_read_lock();
    mcp = rcu_dereference(curconfig);
    *cur_a = mcp->a;
    *cur_b = mcp->b;
    rcu_read_unlock();
}
```

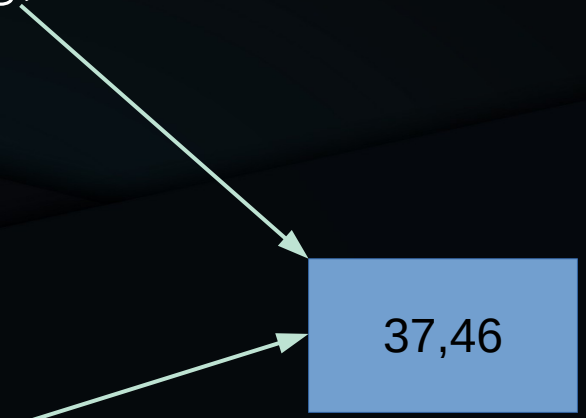
RCU Use Case: Reader

```
struct myconfig { int a, b; } *curconfig; // Initialized

void get(int *cur_a, int *cur_b)
{
    struct myconfig *mcp;

    rcu_read_lock();
    mcp = rcu_dereference(curconfig);
    *cur_a = mcp->a;
    *cur_b = mcp->b;
    rcu_read_unlock();
}
```



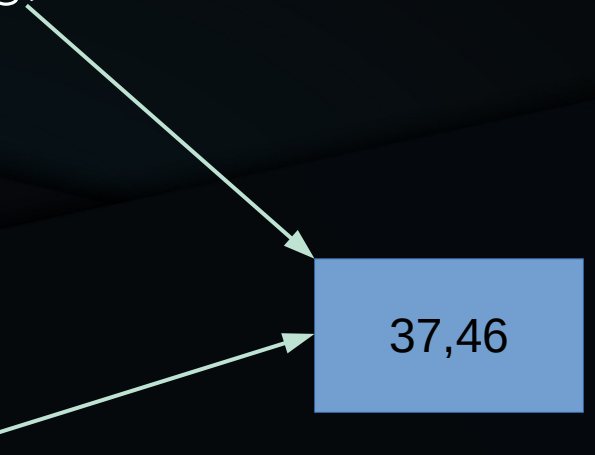
RCU Use Case: Reader

```
struct myconfig { int a, b; } *curconfig; // Initialized

void get(int *cur_a, int *cur_b)
{
    struct myconfig *mcp;

    rcu_read_lock();
    mcp = rcu_dereference(curconfig);
    *cur_a = mcp->a;
    *cur_b = mcp->b;
    rcu_read_unlock();
}
```



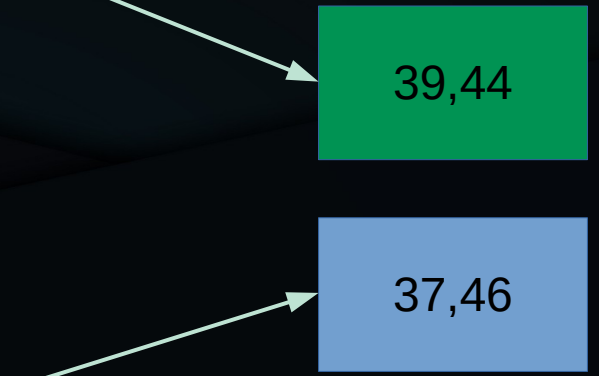
RCU Use Case: Reader

```
struct myconfig { int a, b; } *curconfig; // Initialized

void get(int *cur_a, int *cur_b)
{
    struct myconfig *mcp;

    rcu_read_lock();
    mcp = rcu_dereference(curconfig);
    *cur_a = mcp->a; (37)
    *cur_b = mcp->b;
    rcu_read_unlock();
}
```



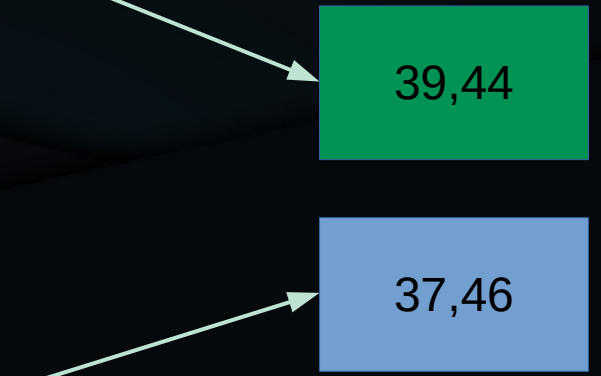
RCU Use Case: Reader

```
struct myconfig { int a, b; } *curconfig; // Initialized

void get(int *cur_a, int *cur_b)
{
    struct myconfig *mcp;

    rcu_read_lock();
    mcp = rcu_dereference(curconfig);
    *cur_a = mcp->a; (37)
    *cur_b = mcp->b;
    rcu_read_unlock();
}
```



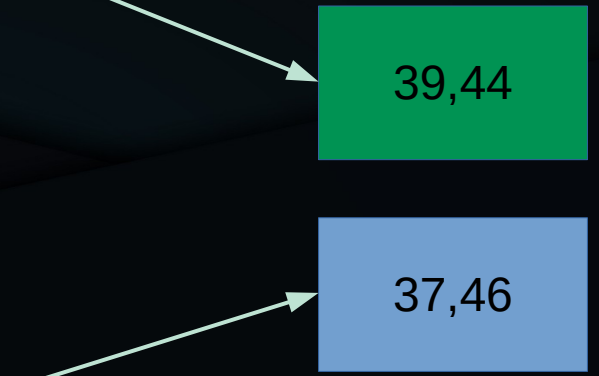
RCU Use Case: Reader

```
struct myconfig { int a, b; } *curconfig; // Updated

void get(int *cur_a, int *cur_b)
{
    struct myconfig *mcp;

    rcu_read_lock();
    mcp = rcu_dereference(curconfig);
    *cur_a = mcp->a; (37)
    *cur_b = mcp->b; (46)
    rcu_read_unlock();
}
```



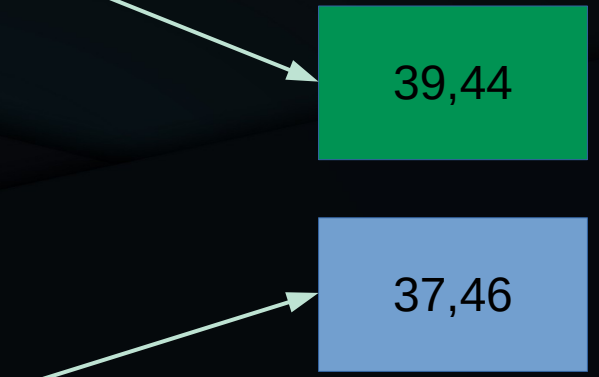
RCU Use Case: Reader

```
struct myconfig { int a, b; } *curconfig; // Updated

void get(int *cur_a, int *cur_b)
{
    struct myconfig *mcp;

    rcu_read_lock();
    mcp = rcu_dereference(curconfig);
    *cur_a = mcp->a; (37)
    *cur_b = mcp->b; (46)
    rcu_read_unlock();
}
```



RCU Use Case: Reader

```
struct myconfig { int a, b; } *curconfig; // Updated

void get(int *cur_a, int *cur_b)
{
    struct myconfig *mcp;

    rcu_read_lock();
    mcp = rcu_dereference(curconfig);
    *cur_a = mcp->a; (37)
    *cur_b = mcp->b; (46)
    rcu_read_unlock();
}
```



Despite change, got consistent values!

RCU Use Case: Writer

RCU Use Case: Typical Writer

```
DEFINE_SPINLOCK(mylock);

void set(int cur_a, int cur_b)
{
    struct myconfig *mcp = kmalloc(...);
    struct myconfig *oldmcp;

    BUG_ON(!mcp);
    mcp->a = cur_a;
    mcp->b = cur_b;
    spin_lock(&mylock);
    oldmcp = rcu_dereference_protected(curconfig, lockdep_is_held(&mylock));
    rcu_assign_pointer(curconfig, mcp);
    spin_unlock(&mylock);
    synchronize_rcu();
    kfree(oldmcp);
}
```

RCU Use Case: Typical Writer

```
DEFINE_SPINLOCK(mylock); // RCU doesn't care how writers synchronize!!!

void set(int cur_a, int cur_b)
{
    struct myconfig *mcp = kmalloc(...);
    struct myconfig *oldmcp;

    BUG_ON(!mcp);
    mcp->a = cur_a;
    mcp->b = cur_b;
    spin_lock(&mylock);
    oldmcp = rcu_dereference_protected(curconfig, lockdep_is_held(&mylock));
    rcu_assign_pointer(curconfig, mcp);
    spin_unlock(&mylock);
    synchronize_rcu();
    kfree(oldmcp);
}
```

RCU Use Case: Typical Writer

```
DEFINE_SPINLOCK(mylock); // RCU doesn't care how writers synchronize!!!

void set(int cur_a, int cur_b)
{
    struct myconfig *mcp = kmalloc(...);
    struct myconfig *oldmcp;

    BUG_ON(!mcp);
    mcp->a = cur_a;
    mcp->b = cur_b;
    spin_lock(&mylock);
    oldmcp = rcu_dereference_protected(curconfig, lockdep_is_held(&mylock));
    rcu_assign_pointer(curconfig, mcp);
    spin_unlock(&mylock);
    synchronize_rcu();
    kfree(oldmcp);
}
```

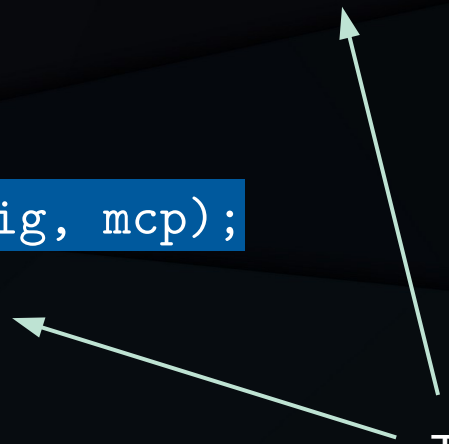
Need writer and two readers on single slide!!!

RCU Use Case: Atomic Writer

```
void set(int cur_a, int cur_b)
{
    struct myconfig *mcp = kmalloc(...);

    mcp->a = cur_a;
    mcp->b = cur_b;
    mcp = xchg(&curconfig, mcp);
    synchronize_rcu();
    kfree(mcp);
}
```

RCU Use Case: Atomic Writer

```
void set(int cur_a, int cur_b)
{
    struct myconfig *mcp = kmalloc(...);

    mcp->a = cur_a;
    mcp->b = cur_b;
    mcp = xchg(&curconfig, mcp);
    synchronize_rcu();
    kfree(mcp);
}
```



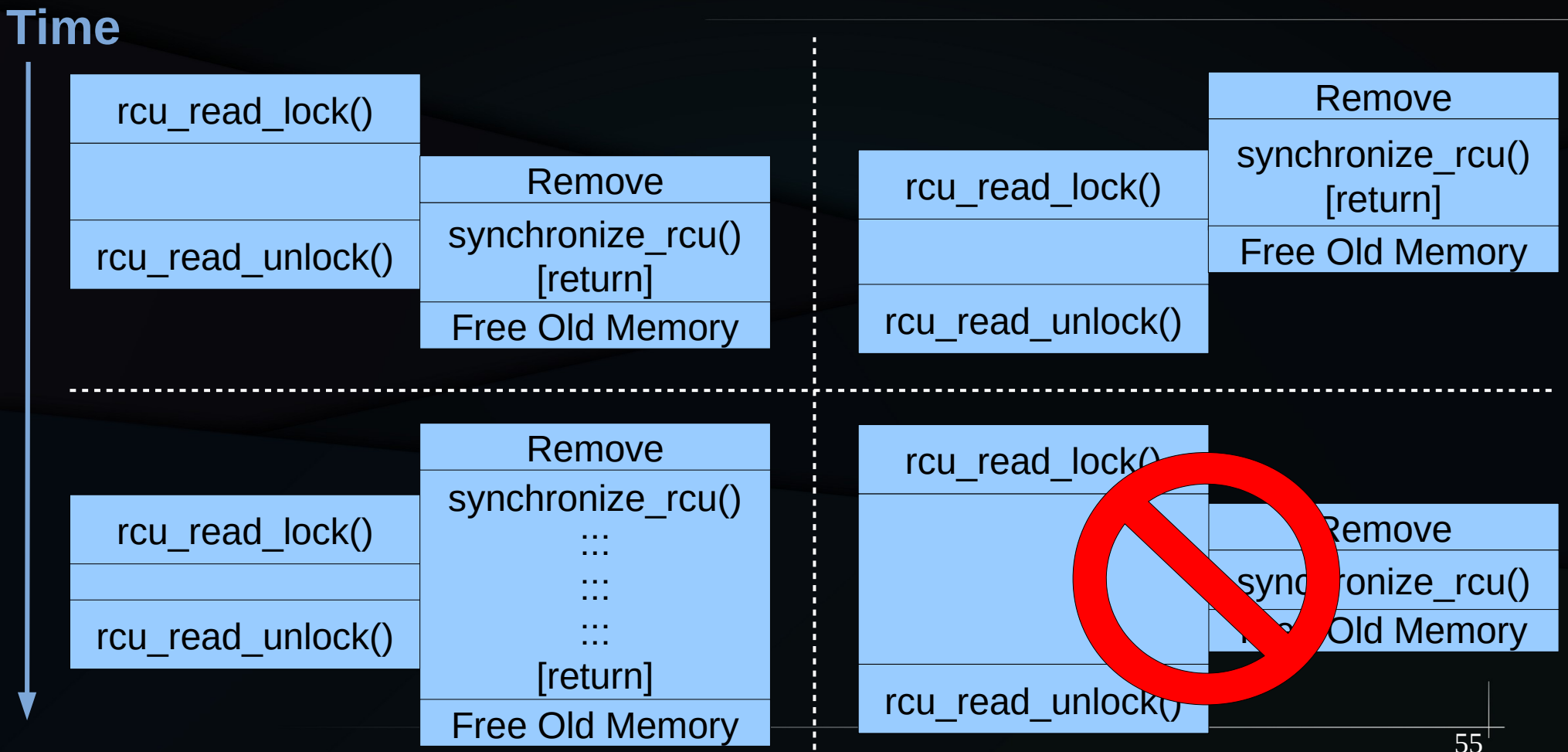
These will represent one writer

RCU Semantics

RCU Semantics (English, Redux)

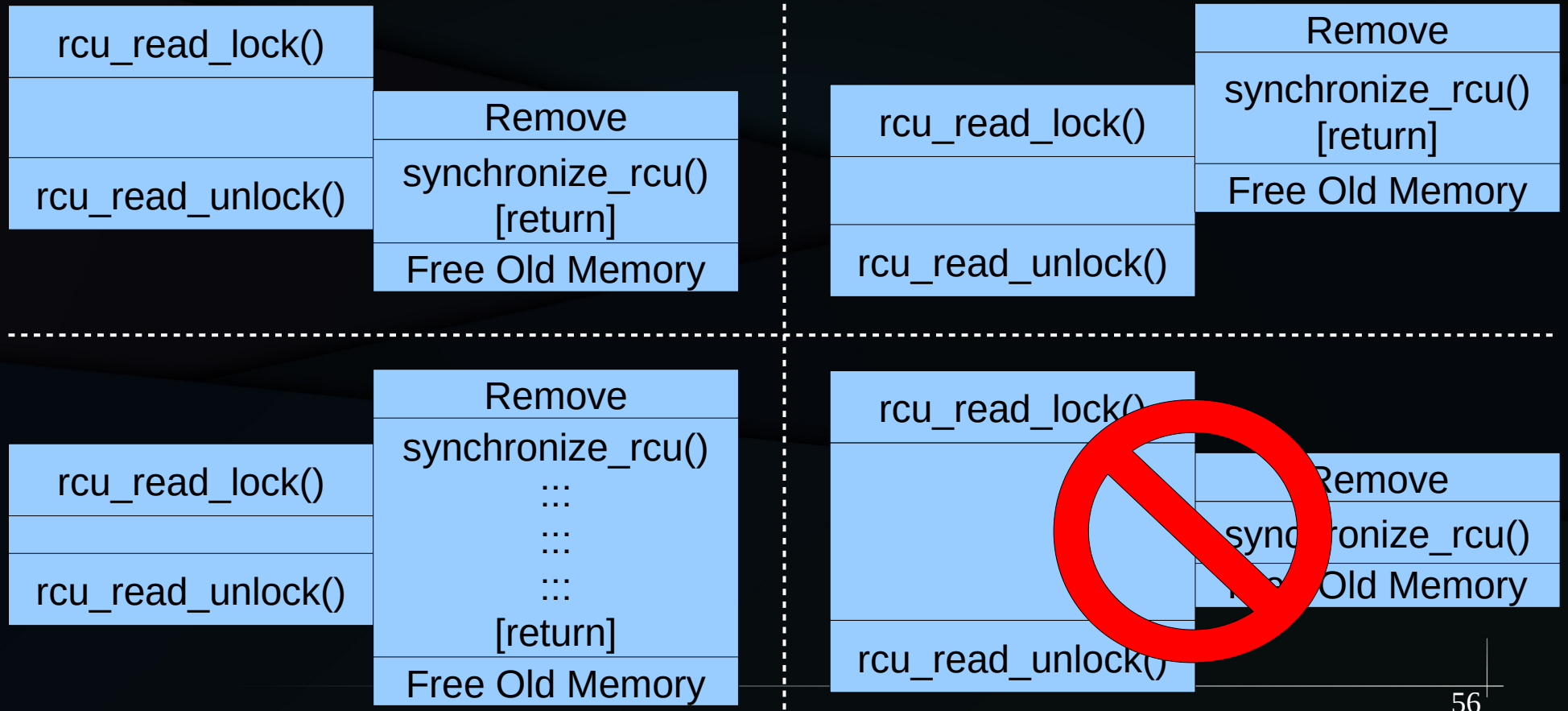
- Semantics weakened from reader-writer locking:
  1) Writers wait for read-holders, but only before freeing
  2) ~~Readers wait for the writer-holders~~
- Compensate for weak temporal semantics by adding *restrictions and spatial semantics* to RCU use cases
- Restated for RCU:
  - If synchronize_rcu() cannot prove that it started before a given rcu_read_lock(), it must not return to its caller until the matching rcu_read_unlock() completes

RCU Semantics (Graphical)



RCU Semantics (Graphical)

Time (really ordering)



RCU Semantics (Exercise The Code)


```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

```
mcp = kmalloc(...)
mcp = xchg(&curconfig, mcp);
synchronize_rcu();
kfree(mcp);
```

```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

RCU Semantics (Exercise The Code)


```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

```
mcp = kmalloc(...)
mcp = xchg(&curconfig, mcp);
synchronize_rcu();
kfree(mcp);
```

```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

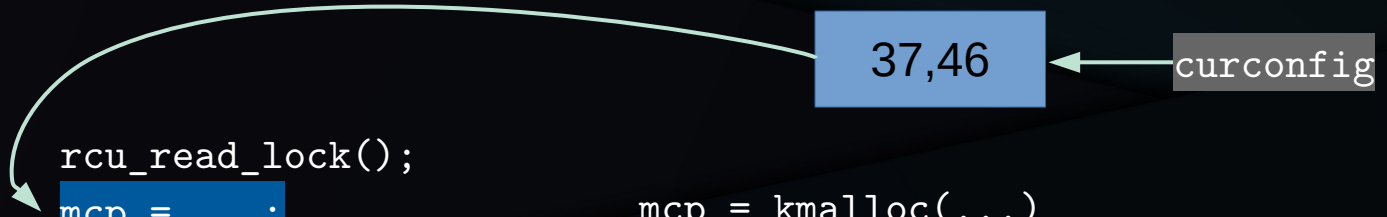
RCU Semantics (Exercise The Code)


```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

```
mcp = kmalloc(...)
mcp = xchg(&curconfig, mcp);
synchronize_rcu();
kfree(mcp);
```

```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

RCU Semantics (Exercise The Code)

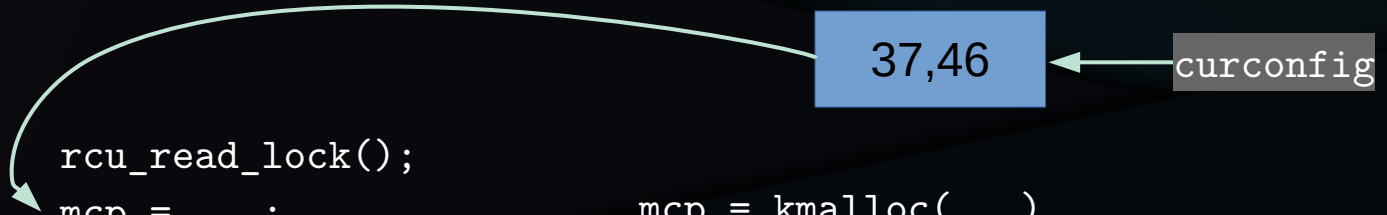



```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

```
mcp = kmalloc(...)
mcp = xchg(&curconfig, mcp);
synchronize_rcu();
kfree(mcp);
```

```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

RCU Semantics (Exercise The Code)

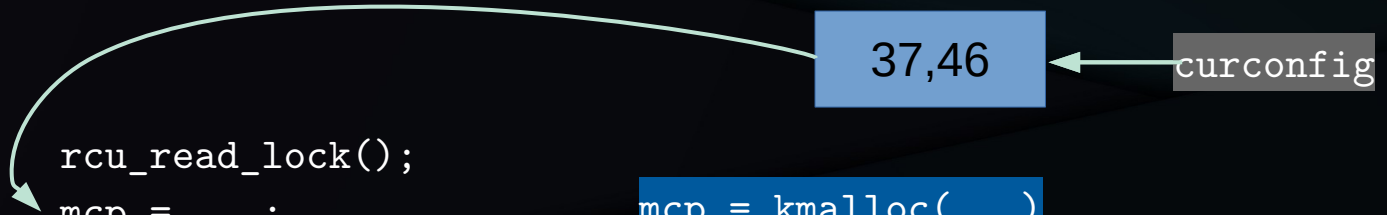



```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a; (37)
*cur_b = mcp->b;
rcu_read_unlock();
```

```
mcp = kmalloc(...)
mcp = xchg(&curconfig, mcp);
synchronize_rcu();
kfree(mcp);
```

```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

RCU Semantics (Exercise The Code)




```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;  (37)
*cur_b = mcp->b;
rcu_read_unlock();
```

```
mcp = kmalloc(...)
mcp = xchg(&curconfig, mcp);
synchronize_rcu();
kfree(mcp);
```

```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

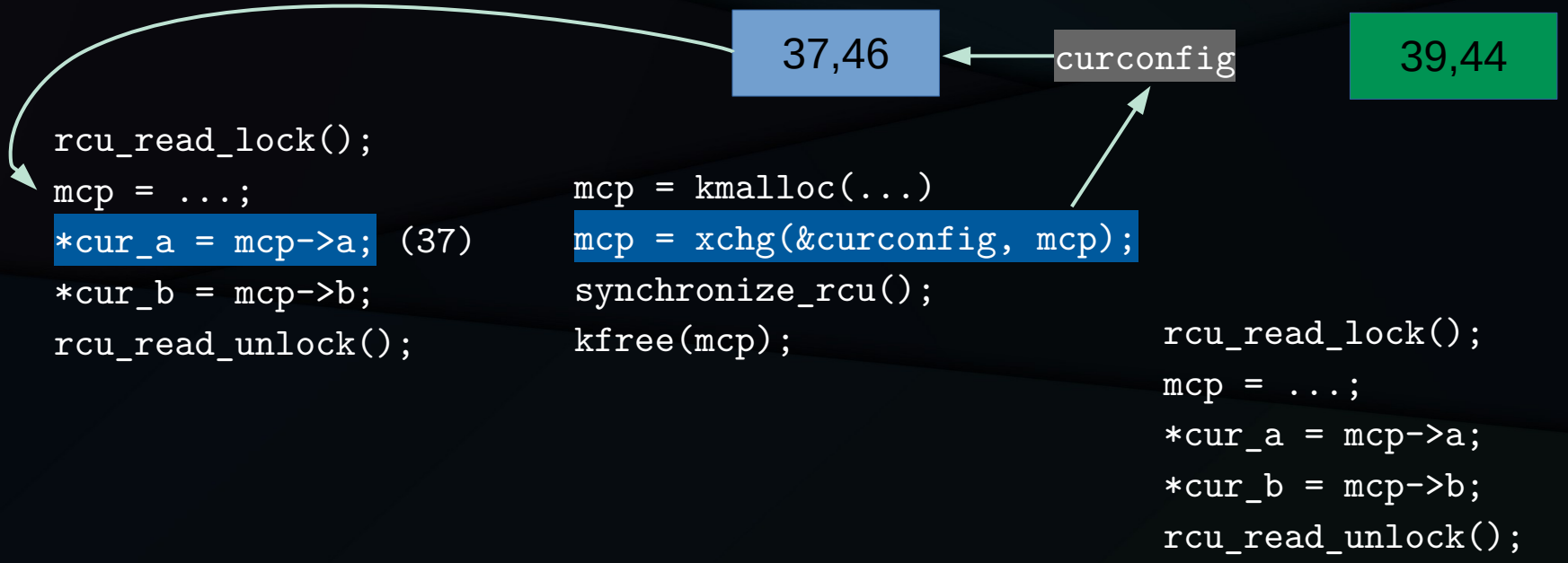
RCU Semantics (Exercise The Code)


```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a; (37)
*cur_b = mcp->b;
rcu_read_unlock();
```

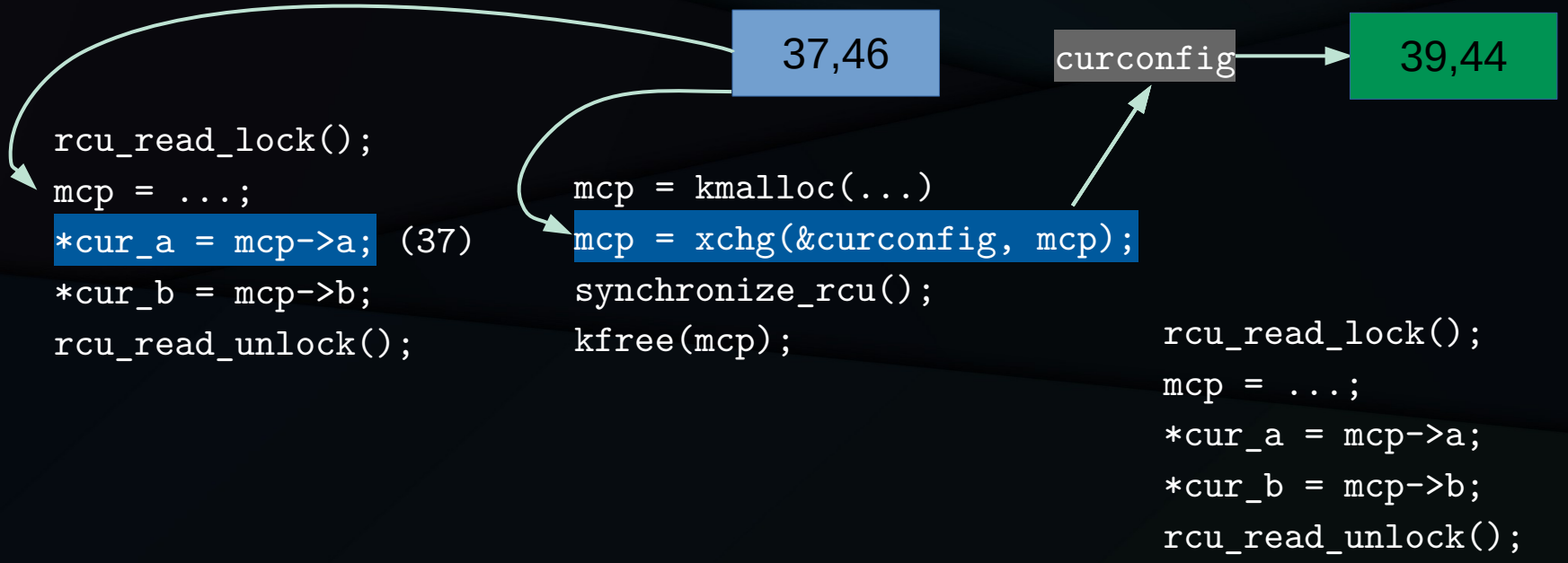
```
mcp = kmalloc(...)
mcp = xchg(&curconfig, mcp);
synchronize_rcu();
kfree(mcp);
```

```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

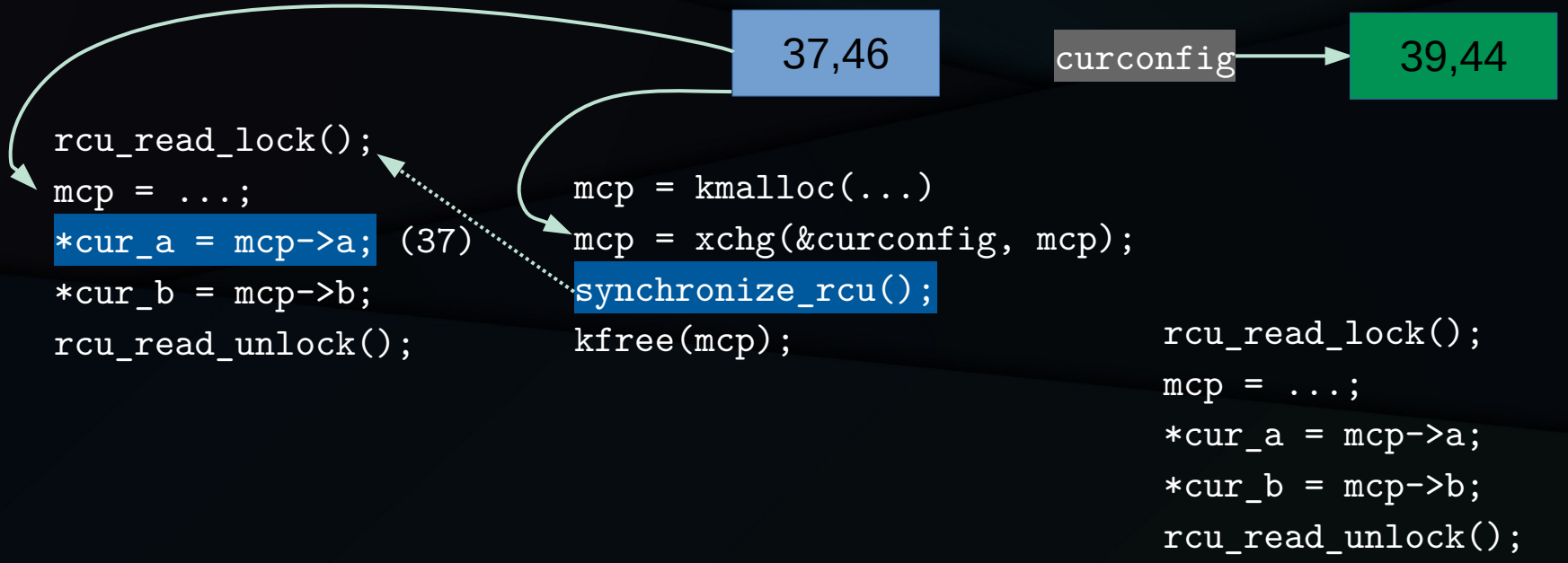
RCU Semantics (Exercise The Code)



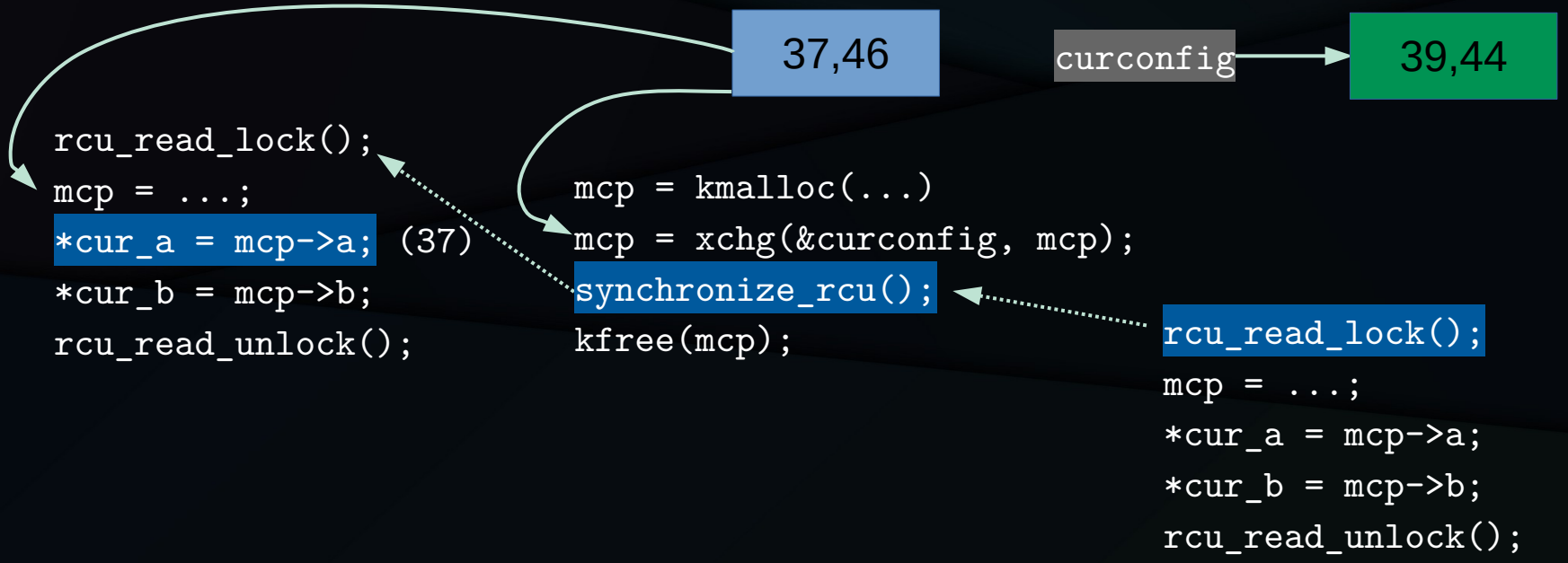
RCU Semantics (Exercise The Code)



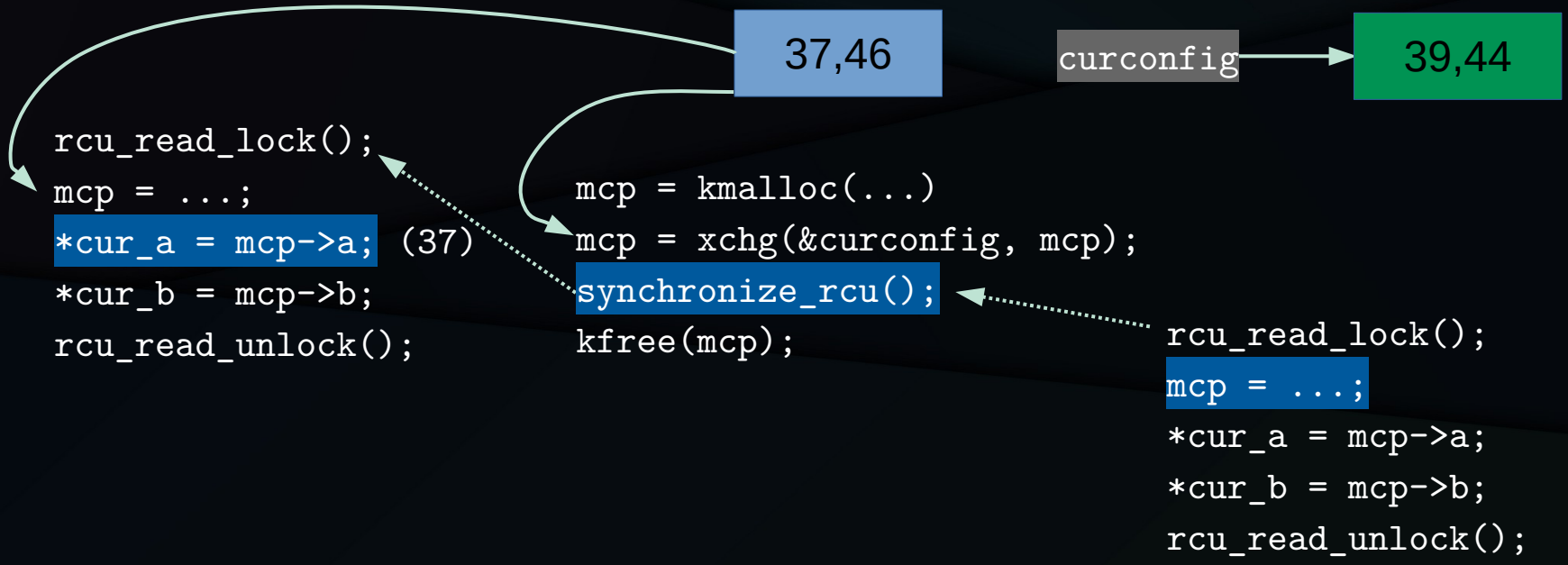
RCU Semantics (Exercise The Code)



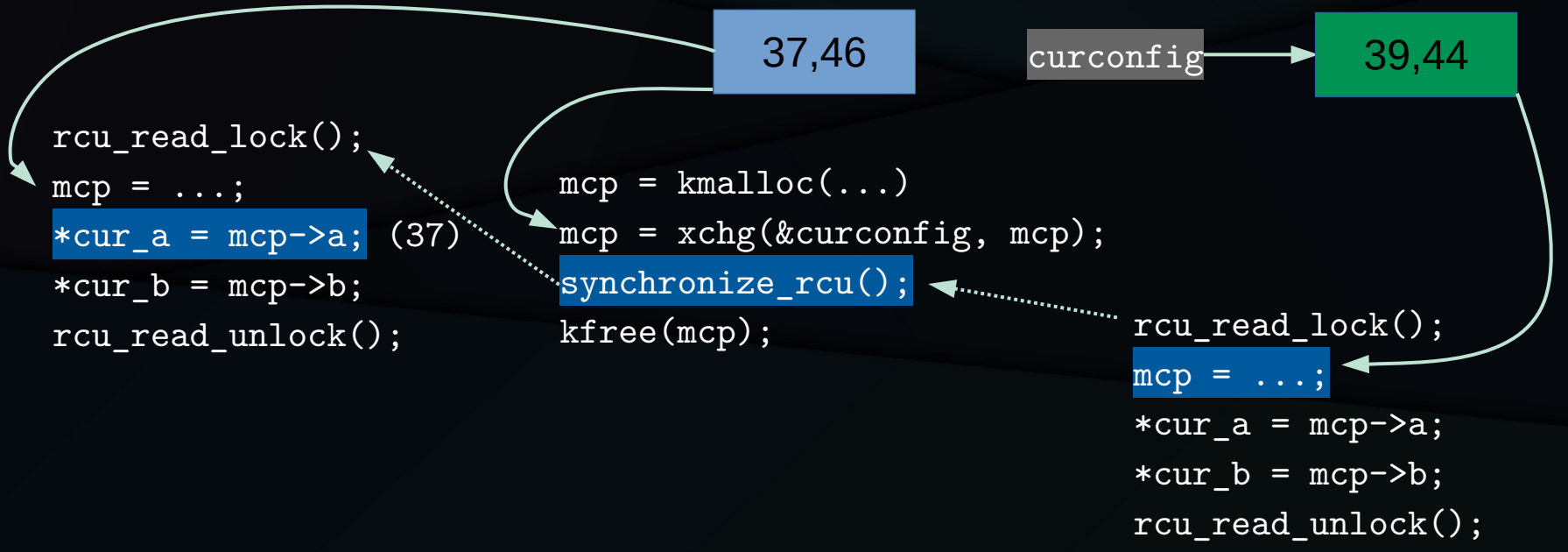
RCU Semantics (Exercise The Code)



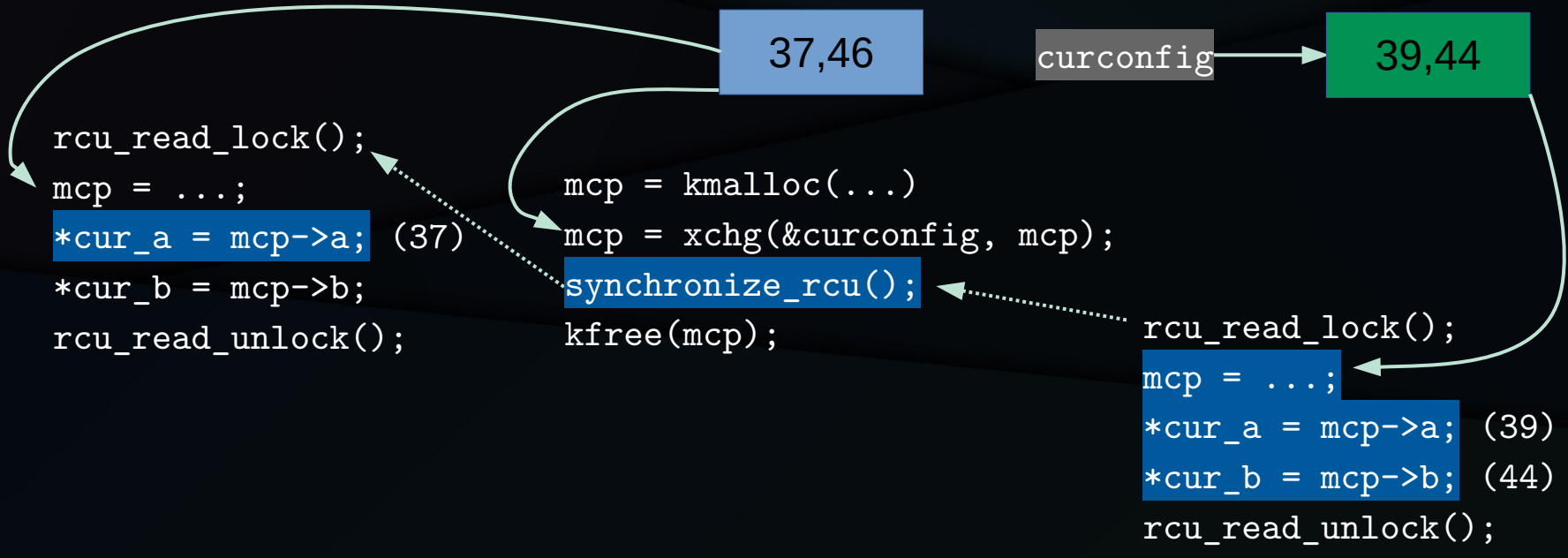
RCU Semantics (Exercise The Code)



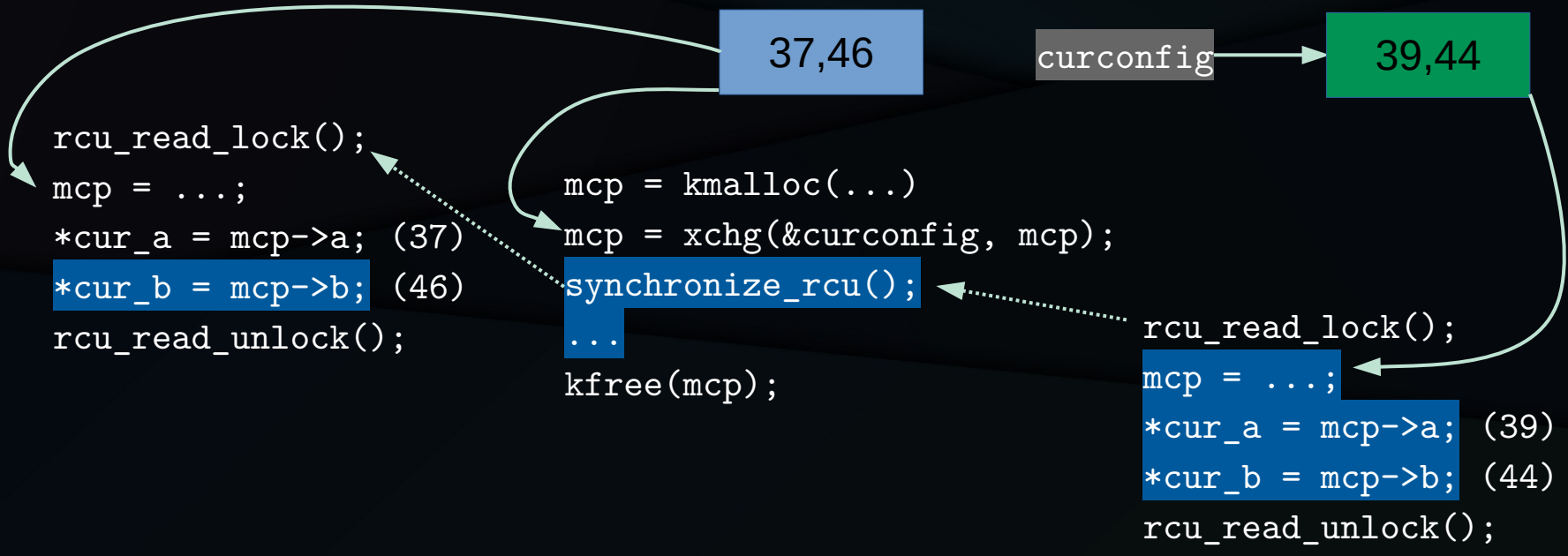
RCU Semantics (Exercise The Code)



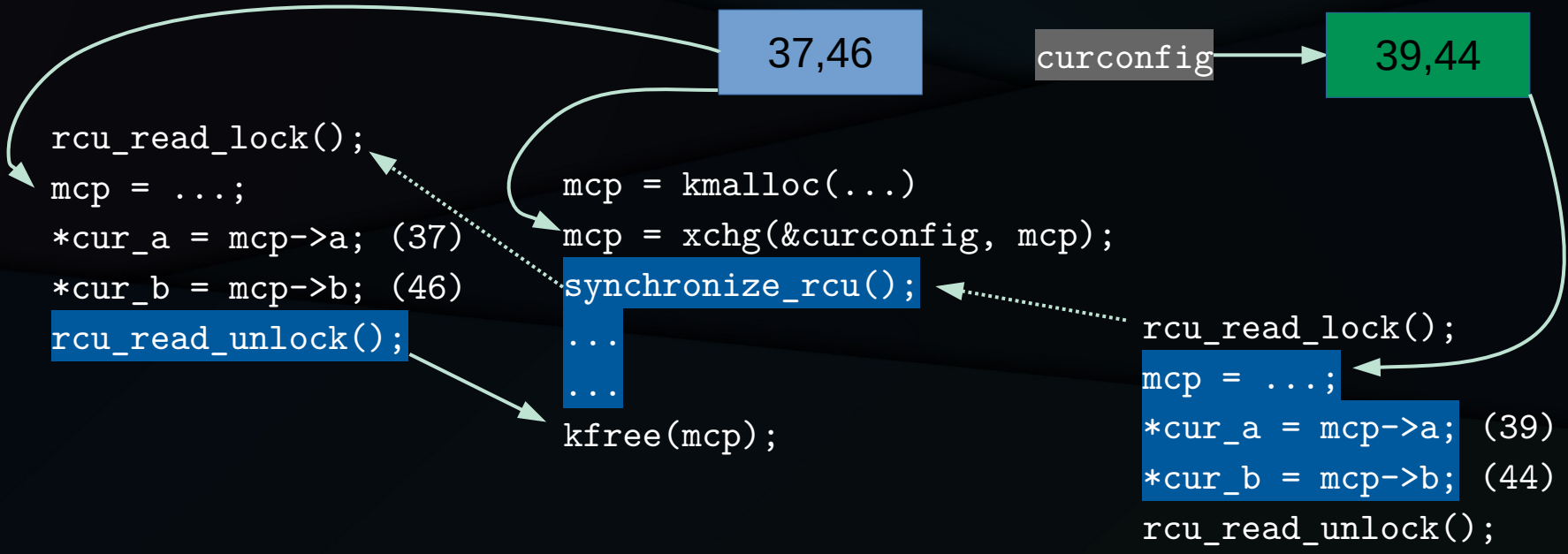
RCU Semantics (Exercise The Code)



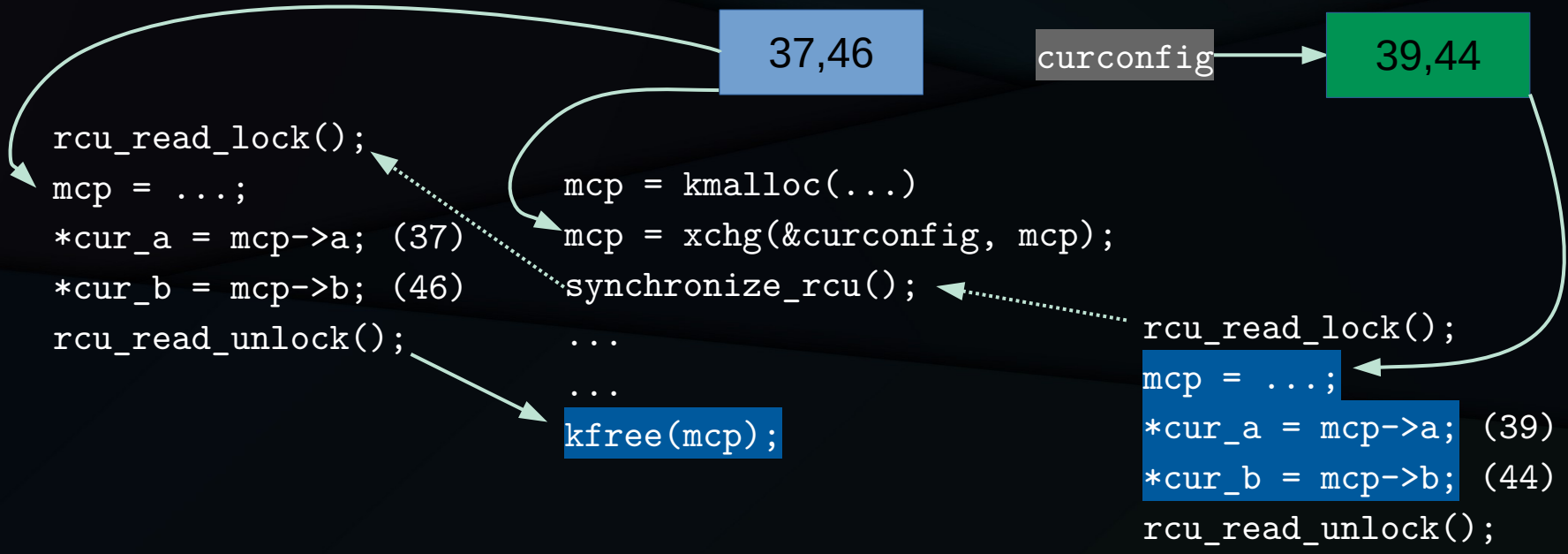
RCU Semantics (Exercise The Code)



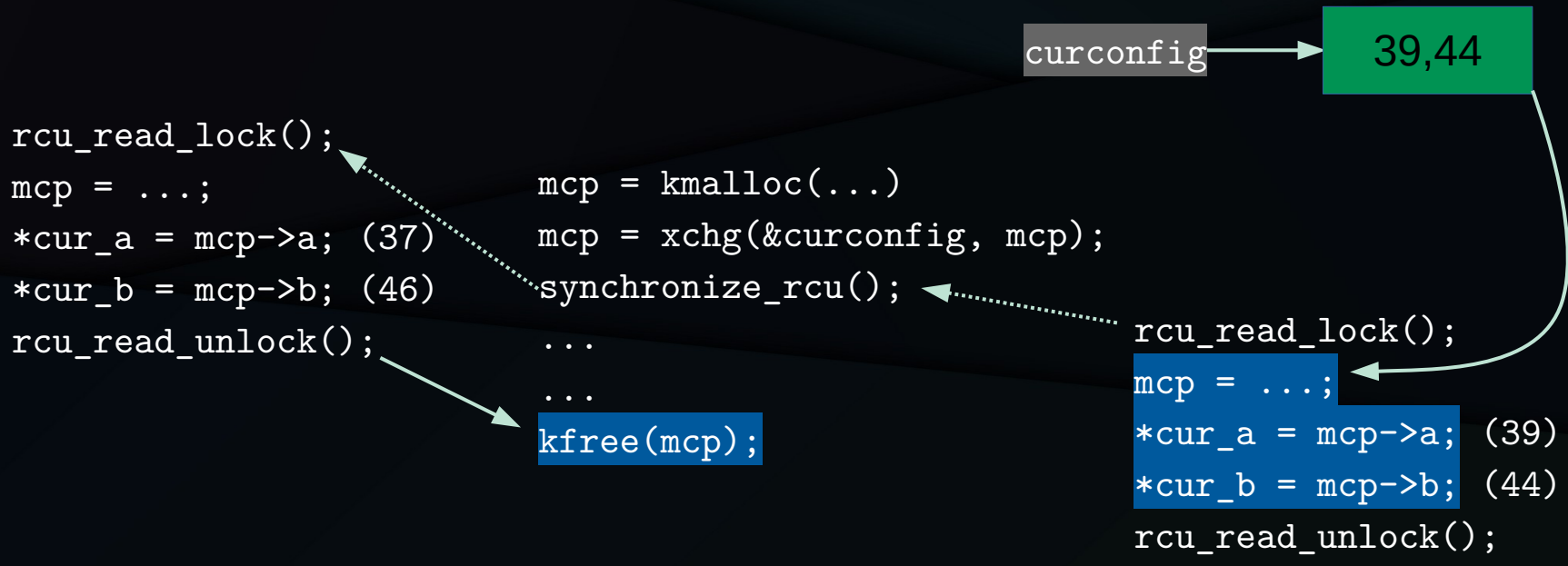
RCU Semantics (Exercise The Code)



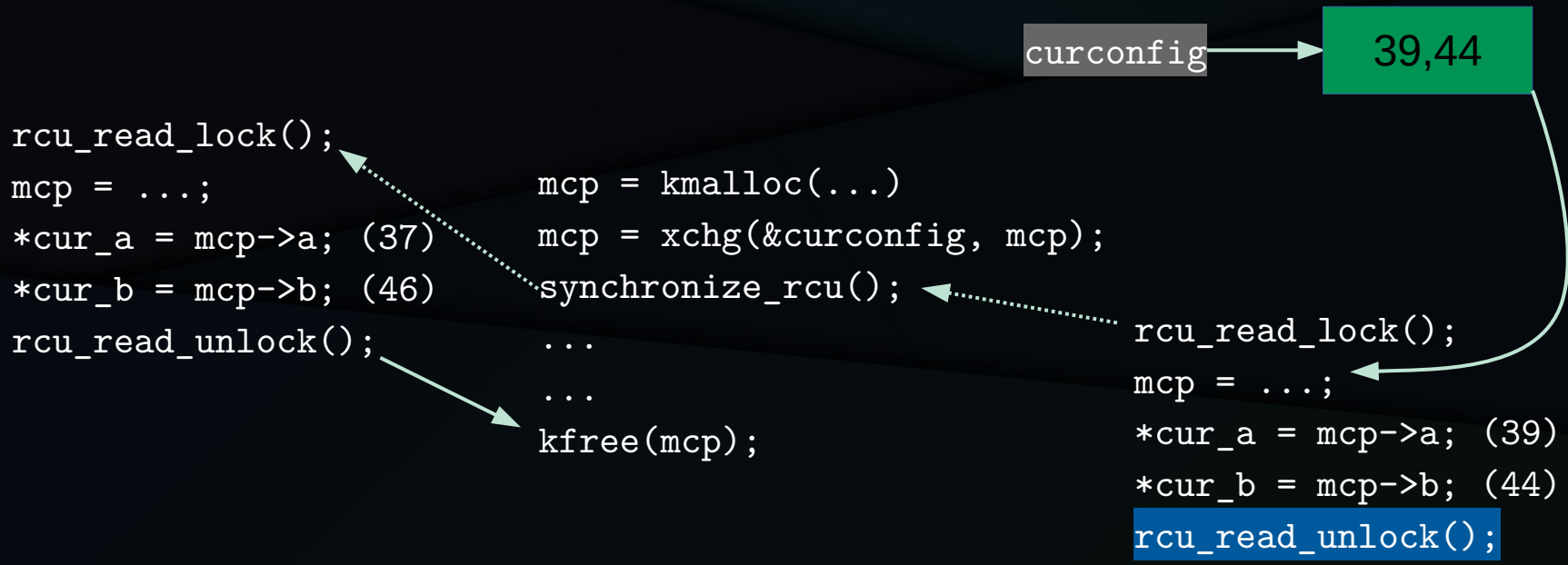
RCU Semantics (Exercise The Code)



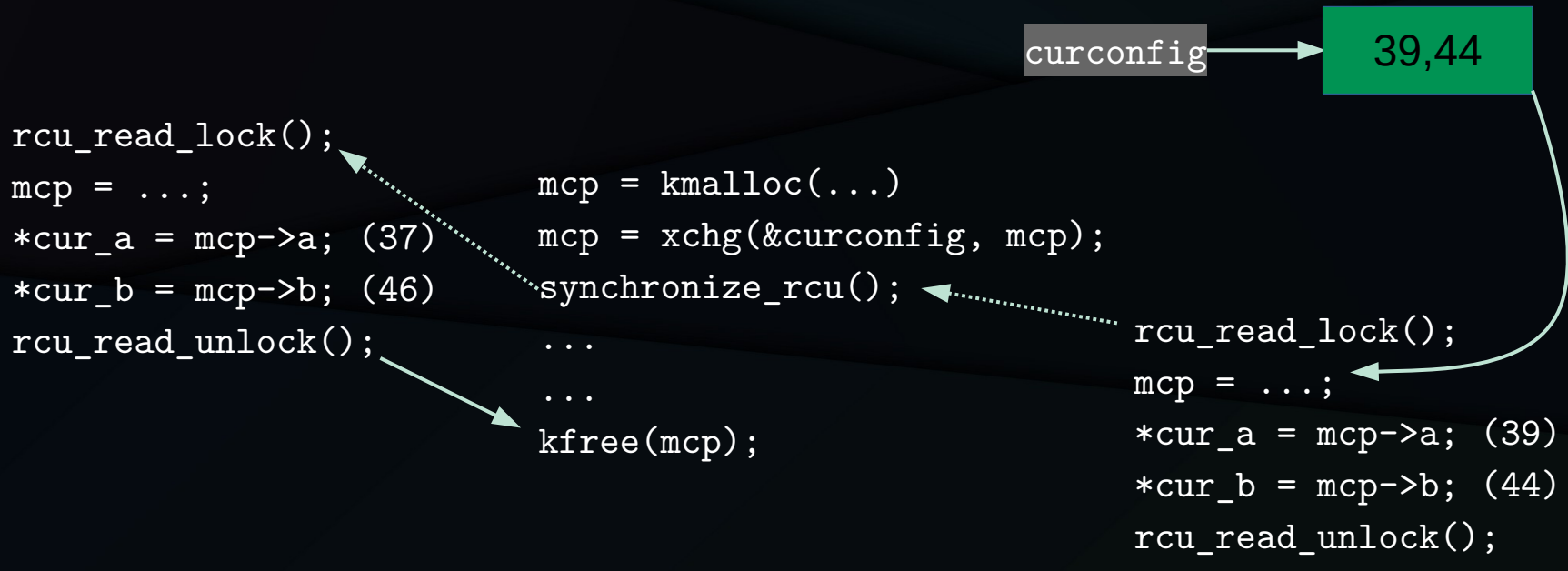
RCU Semantics (Exercise The Code)



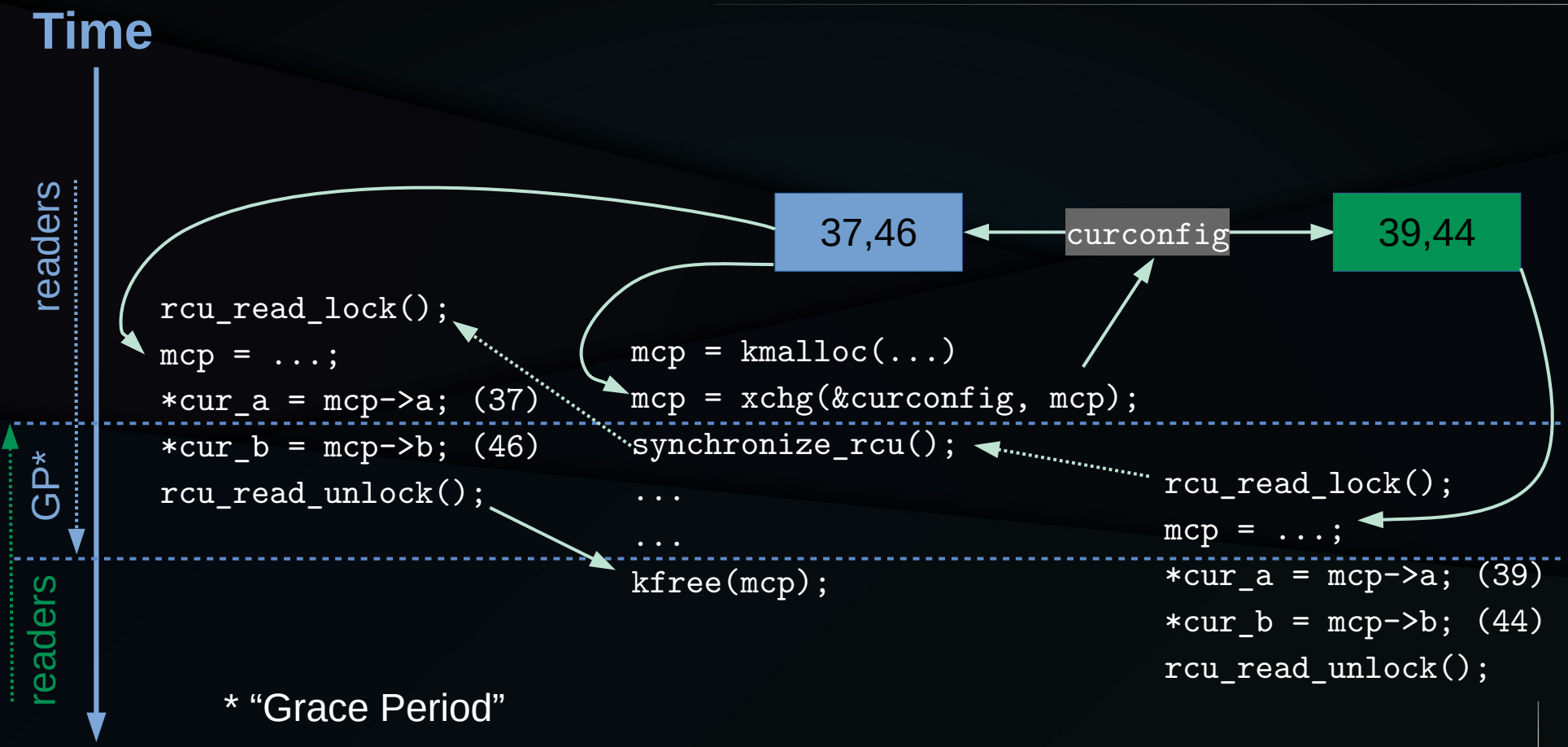
RCU Semantics (Exercise The Code)



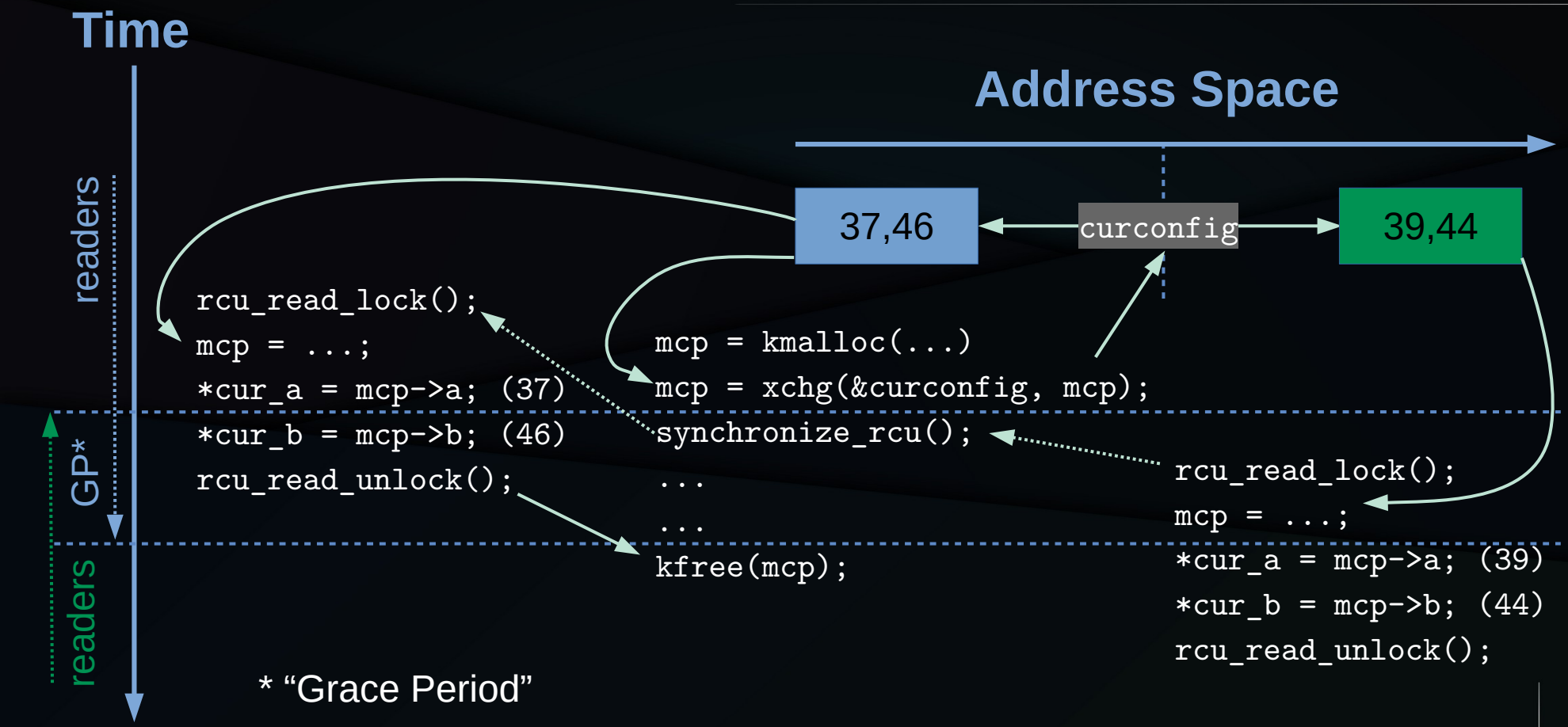
RCU Semantics (Exercise The Code)



RCU Semantics: (Temporal)



RCU Semantics: (Temporal & Spatial)



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

Space/Time Synchronization

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
  - With `xchg()` standing in for these last two

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

Space/Time Synchronization Outline

- Readers:
  - Time via rcu_read_lock() and rcu_read_unlock()
  - Space via rcu_dereference() and friends
- Updates are split into reader-visible and not
  - Add: Initialize, then use rcu_assign_pointer()
  - Delete: Remove, wait for grace period, then free
- Existence guarantees (as opposed to ownership)

Reader Space/Time Synchronization

```
void get(int *cur_a, int *cur_b)
{
    struct myconfig *mcp;

    rcu_read_lock();
    mcp = rcu_dereference(curconfig);
    *cur_a = mcp->a; (37)
    *cur_b = mcp->b; (46)
    rcu_read_unlock();
}
```

Temporal synchronization 1

Spatial synchronization

Existence guarantee

Temporal synchronization 2

Reader Space/Time Synchronization

```
void get(int *cur_a, int *cur_b)
{
    struct myconfig *mcp;

    rcu_read_lock();
    mcp = rcu_dereference(curconfig);
    *cur_a = mcp->a; (37)
    *cur_b = mcp->b; (46)
    rcu_read_unlock();
}
```

Temporal synchronization 1
(Start read-side critical section)

Spatial synchronization
(Get current version)

Existence guarantee

Temporal synchronization 2
(End read-side critical section)

Updater Space/Time Synchronization

```
void set(int *cur_a, int *cur_b)
{
    struct myconfig *mcp = kmalloc(...);

    mcp->a = a;
    mcp->b = b;
    mcp = xchg(&curconfig, mcp);
    synchronize_rcu();
    kfree(mcp);
}
```

Spatial synchronization

Temporal synchronization

Updater Space/Time Synchronization

```
void set(int *cur_a, int *cur_b)
{
    struct myconfig *mcp = kmalloc(...);

    mcp->a = a;
    mcp->b = b;
    mcp = xchg(&curconfig, mcp);
    synchronize_rcu();
    kfree(mcp);
}
```

Spatial synchronization
(Make old data inaccessible and new data accessible to future readers)

Temporal synchronization
(Wait for pre-existing readers)

RCU: Exploiting Both Temporal and Spatial Synchronization for Decades!

Paolo Neo, Public Domain

Who Does Spatial Synchronization?!?

Who Does Spatial Synchronization?!?

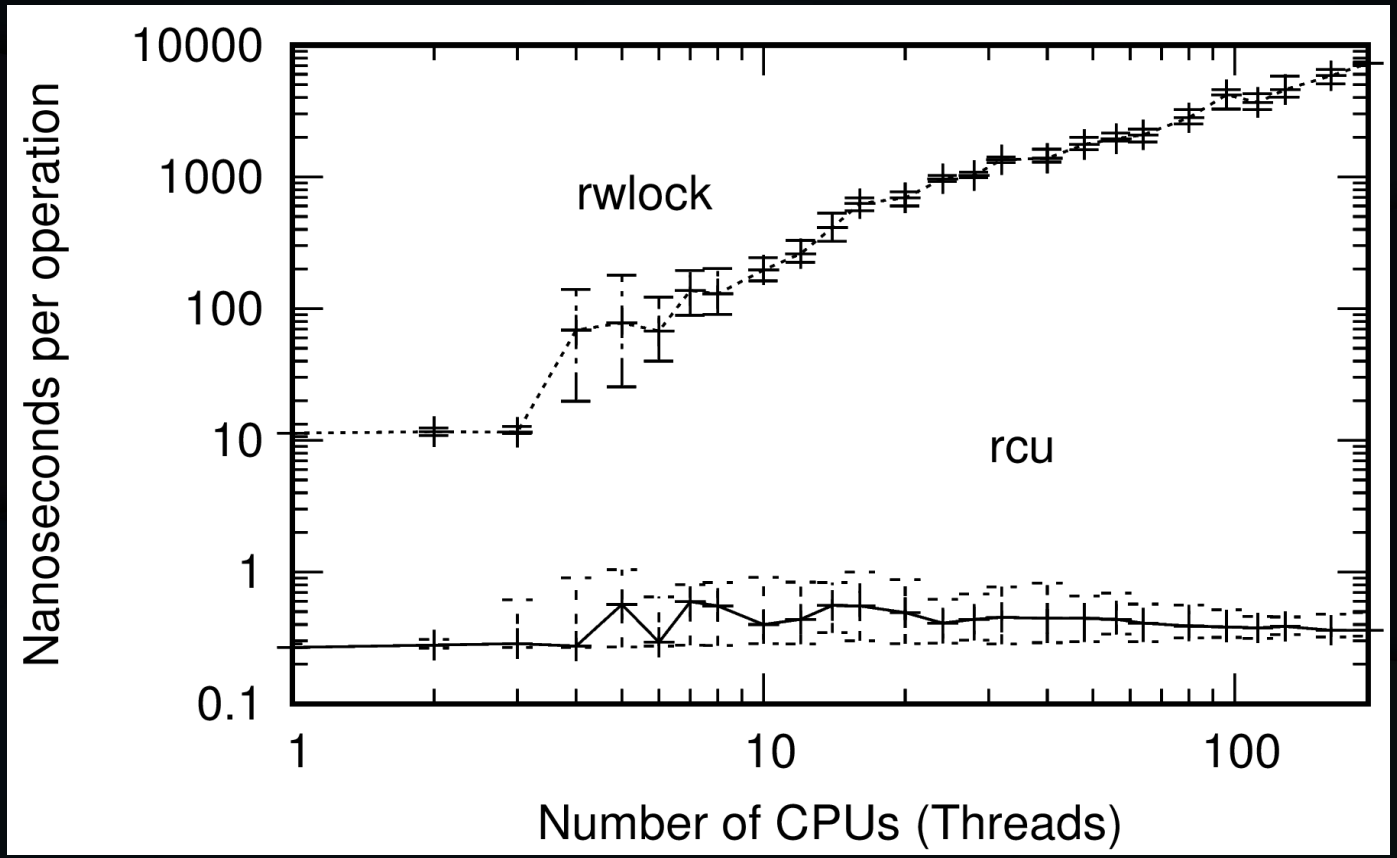
- Per-task stack locations
- Per-CPU/-thread variables
- Hash tables with per-bucket locks
  - And sharding in general (the “data locking” of old)
- Hazard pointers & other deferred reclamation

Who Does Spatial Synchronization?!?

- Per-task stack locations
- Per-CPU/-thread variables
- Hash tables with per-bucket locks
  - And sharding in general (the “data locking” of old)
- Hazard pointers & other deferred reclamation
- **In short, pretty much everybody!!!**

Performance: rwlock vs RCU

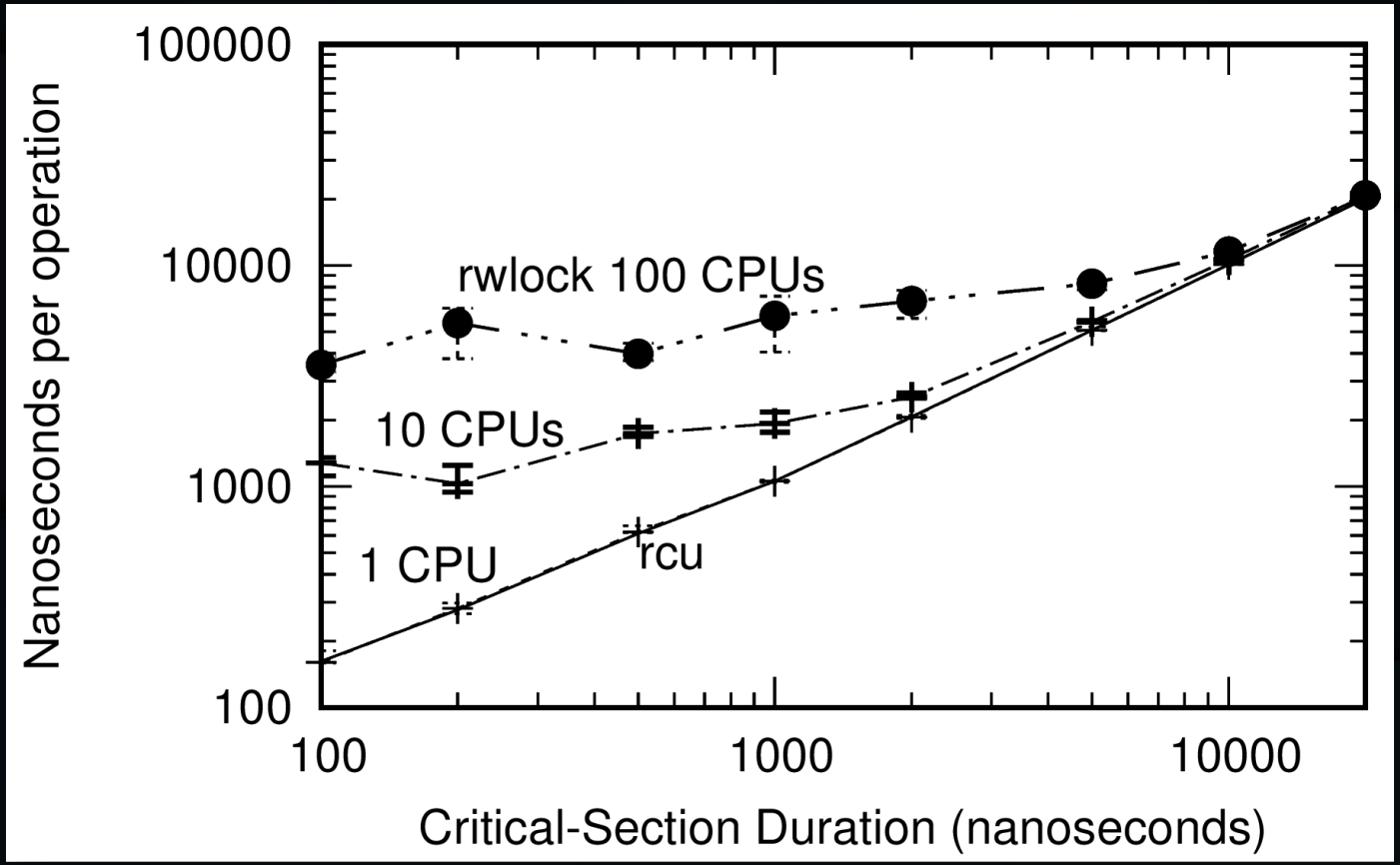
Scalability for Empty Critical Sections



Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz

perfbook CodeSamples/defer/data/rcuscale.hps.2020.05.28a/rwlockperf.eps

… And Non-Empty Critical Sections



Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz

perfbook CodeSamples/defer/data/rcuscale.hps.2020.05.28a/rwlockperf.eps

RCU Insert, Delete, and Traversal

RCU Insert, Delete, and Search

- Atomic replacement is unusual
- More typical: Separate insertion and deletion
  - `list_add_rcu()`
  - `list_del_rcu()`
  - `list_for_each_entry_rcu()`
- And asynchronous post-grace-period kfree():
  - `kfree_rcu()`

RCU to Quasi Reader-Writer Lock

RCU to Quasi Reader-Writer Lock

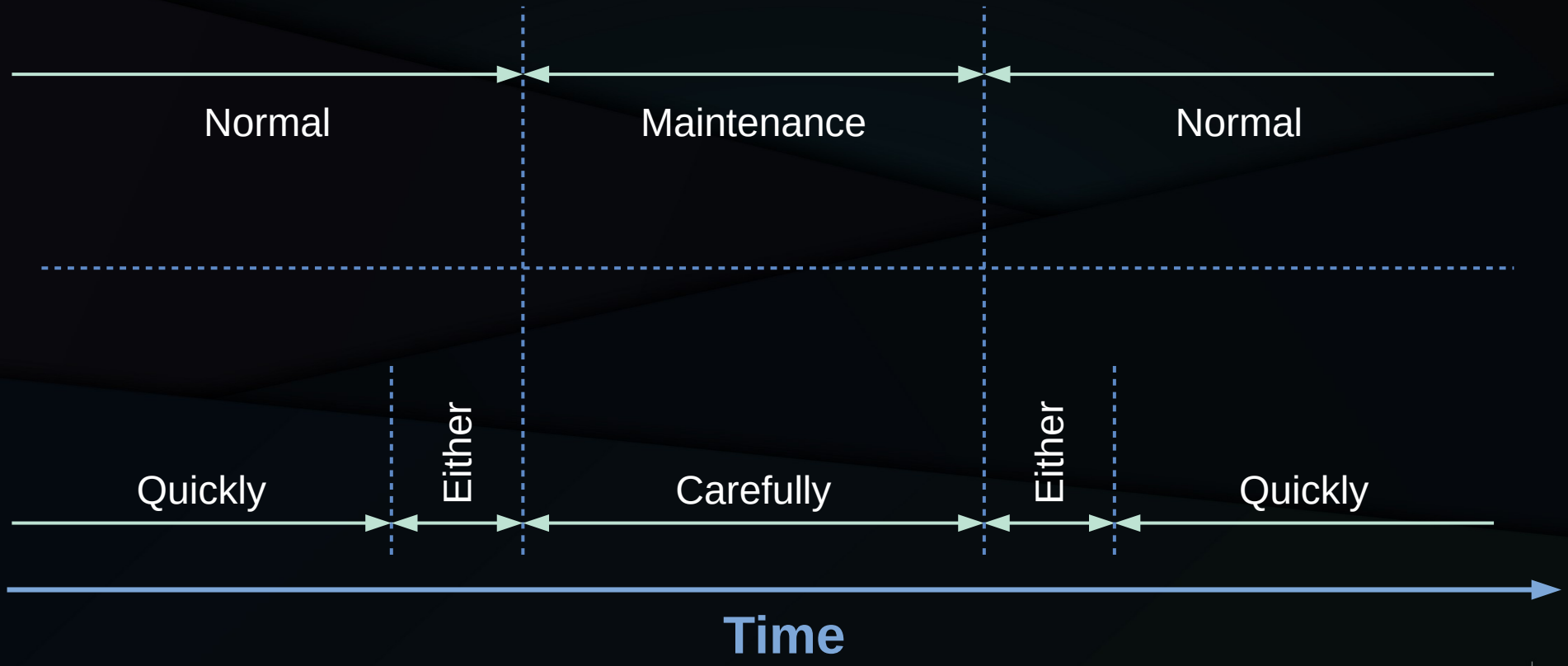
- Add to publish/subscribe and wait-to-finish:
  - Heap-allocated linked structure
  - Deferred reclamation
  - RCU readers as read-held reader-writer lock
  - Both spatial and temporal synchronization

Example: Phased State Change

Example: Phased State Change

- Multithreaded application
  - Common-case operation must be fast
  - But care is required during maintenance
- Use flag to indicate that care is required
  - But how to reliably synchronize?
  - OK to be careful just before/after maintenance

Phased State Change (Graphical)



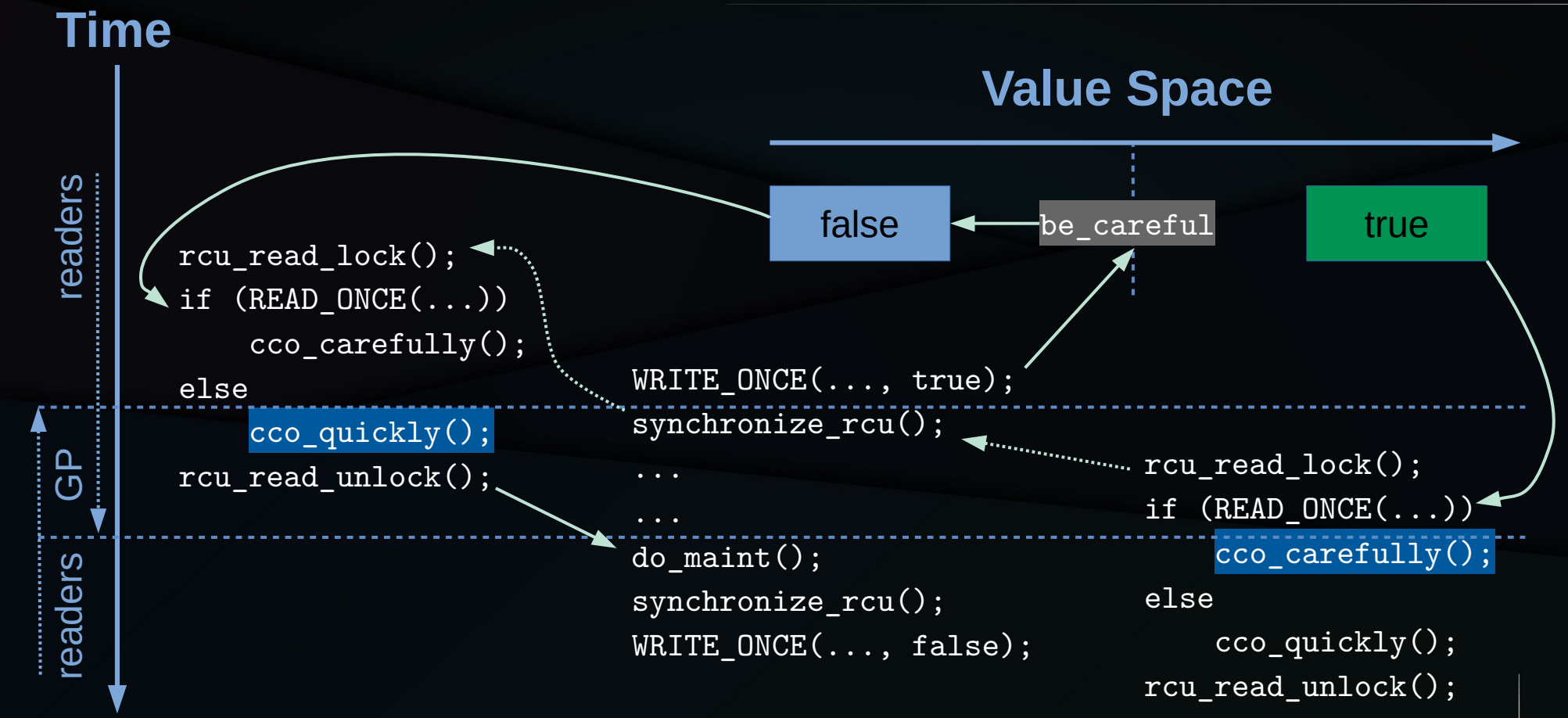
Common-Case Operation

```
bool be_careful;

void cco(void)
{
    rcu_read_lock();
    if (READ_ONCE(be_careful))
        cco_carefully();
    else
        cco_quickly();
    rcu_read_unlock();
}
```

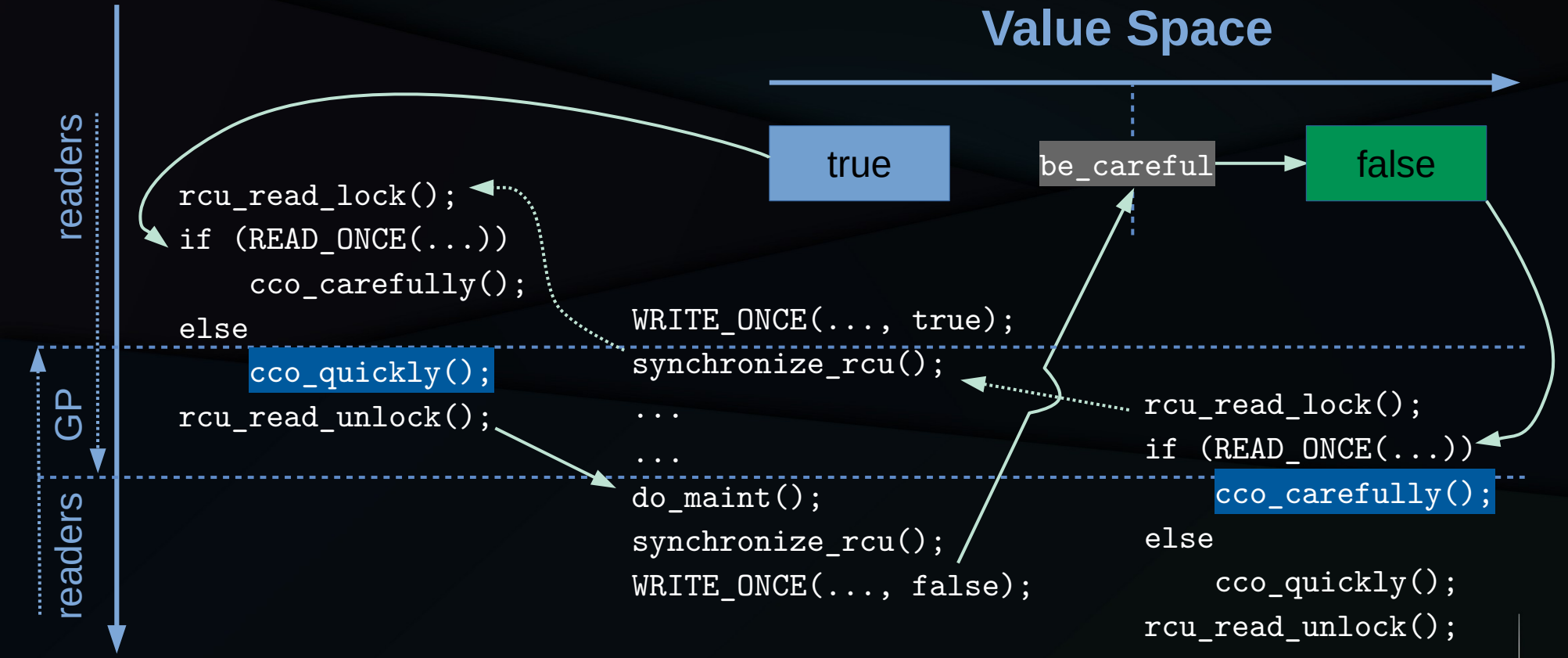
Maintenance Operation

```
void maint(void)
{
    WRITE_ONCE(be_careful, true);
    synchronize_rcu();
    do_maint();
    synchronize_rcu(); // Why is this needed?
    WRITE_ONCE(be_careful, false);
}
```

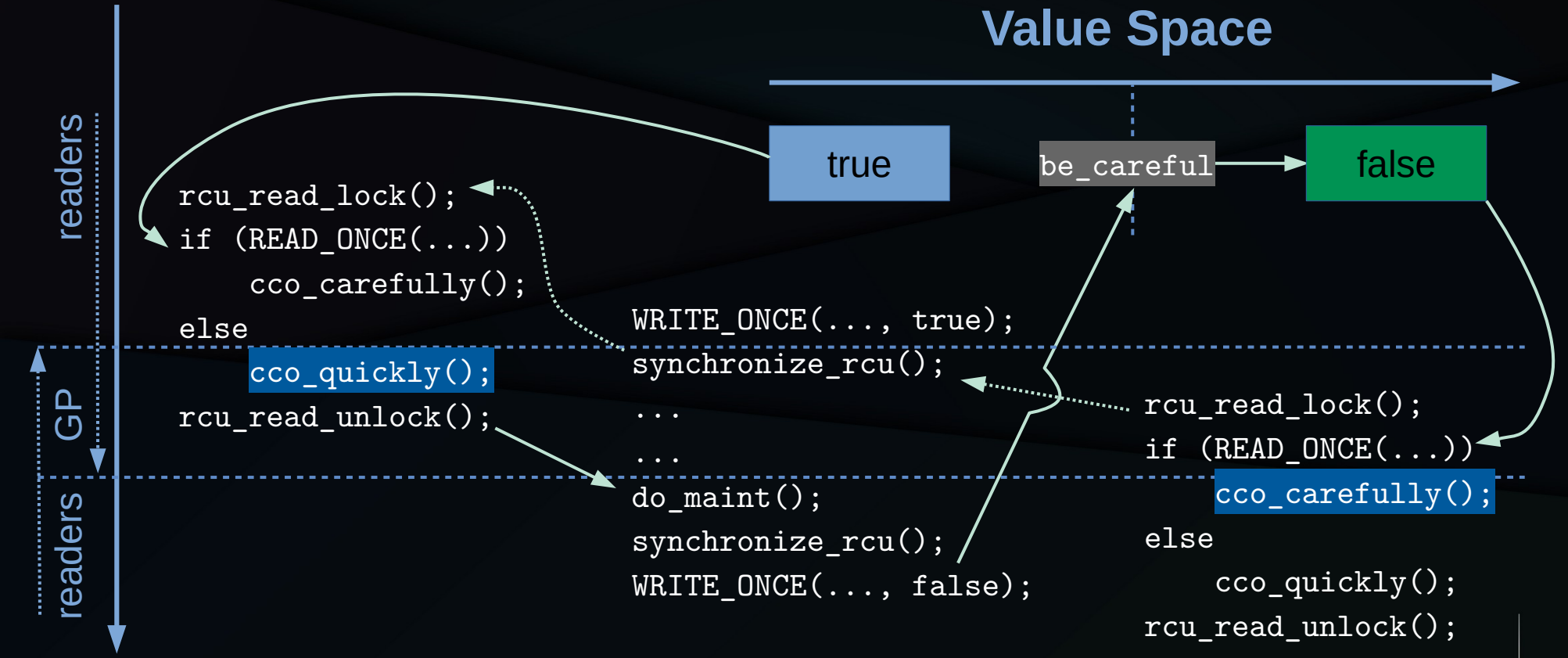
Phased State Change: Time & Space



Phased State Change: Time & Space



Phased State Change: Time & Space



Alternatives for maintenance-end synchronization?

RCU to Phased State Change

RCU to Phased State Change

- Add to wait-to-finish:
  - Checked state variable

Linux Kernel Usage?

Linux Kernel Usage?

- Example RCU-protected data structures:
  - task_struct: parent, cred, sighand, cgroups
  - cgroup: subsys
  - fdtable: fd
  - files_struct: fdt, fd_array
  - inode, super_block: [is]_fsnotify_marks

Linux Kernel Usage?

- include/linux/rcu_sync.h:
  - rcu_sync_init(), rcu_sync_enter_start(), rcu_sync_enter(), rcu_sync_exit()
- Used by per-CPU reader-writer semaphores
  - Light-weight readers in the absence of writers

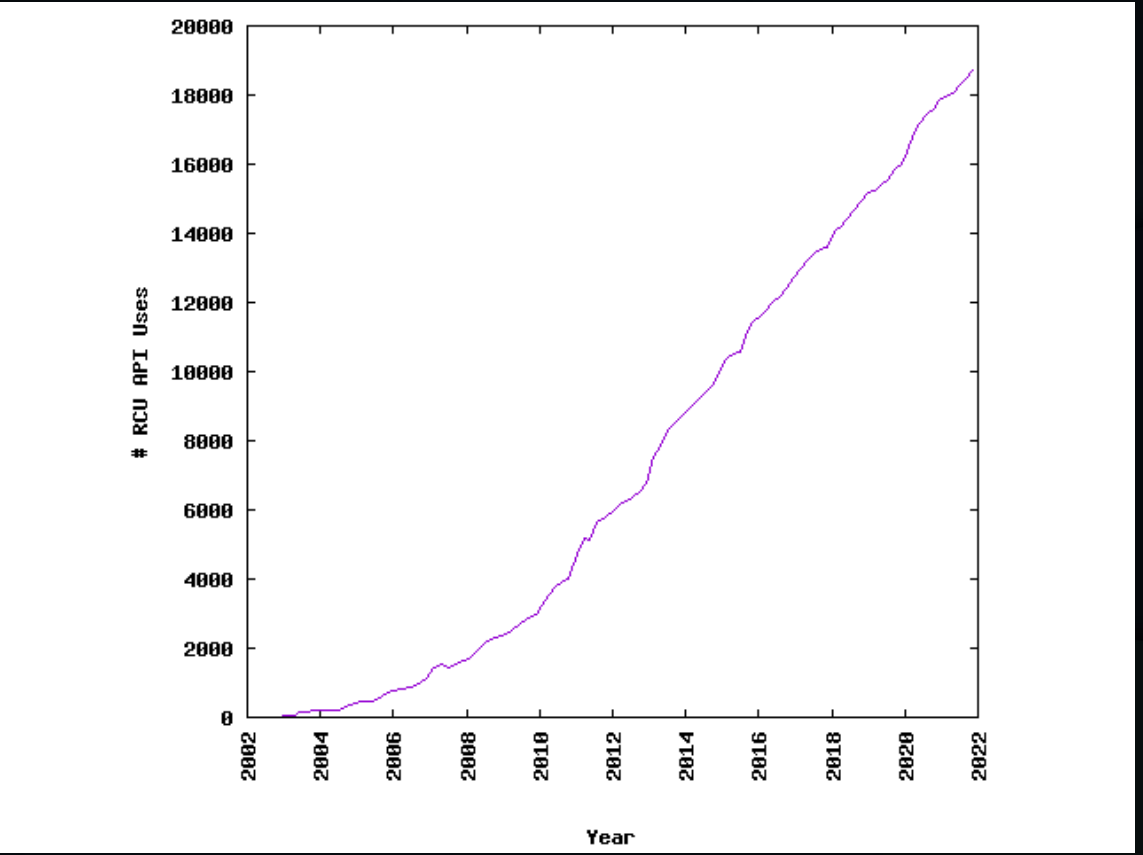
Overall RCU Usage in v5.15?

Overall RCU Usage in v5.15?

Subsystem	Calls to RCU APIs	Lines of Code *	Uses/KLoc
ipc	89	9,649	9.22
virt	65	8,465	7.68
net	7,411	1,192,884	6.21
security	595	104,460	5.70
kernel	1,689	406,868	4.15
mm	319	160,825	1.98
...	...	...	...
drivers	5,328	19,275,468	0.28
...	...	...	...
Total	18,710	27,557,130	0.68

* Only code in .h and .c files is counted, but all lines in such files are counted.

Overall RCU Usage Over Time?



http://www.rdrop.com/~paulmck/RCU/linuxusage.html

Debugging Linux-Kernel RCU Code

Debugging Linux-Kernel RCU Code

- Assertions: lockdep, sparse, KCSAN
- Prototype with userspace RCU
  - Use a real debugger!!!
- RCU CPU stall warnings

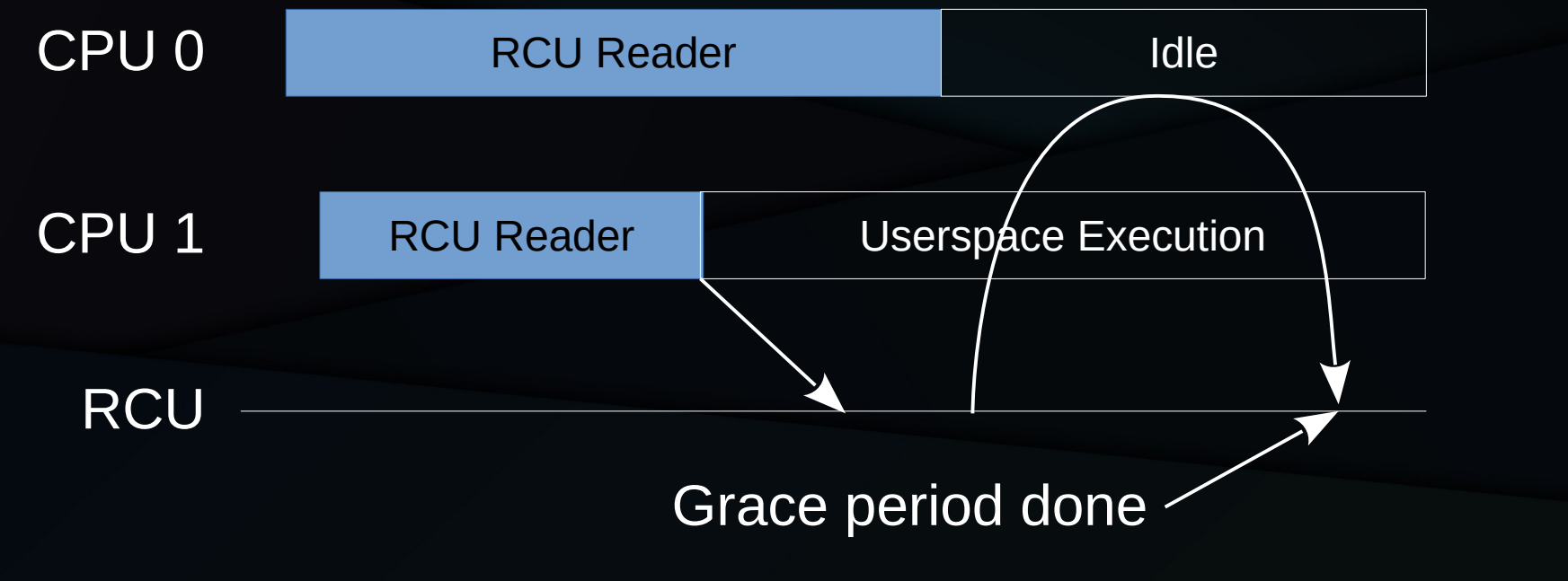
RCU lockdep Assertions

- Code really an RCU reader?
  - `rcu_read_lock_held()`
  - `rcu_read_lock_bh_held()`
  - `rcu_read_lock_sched_held()`
  - `rcu_read_lock_any_held()`
- Code an RCU reader or protected by specified lock(s)?
  - `rcu_dereference_check()`
  - `rcu_dereference_bh_check()`
  - `rcu_dereference_sched_check()`

RCU sparse and KCSAN Assertions

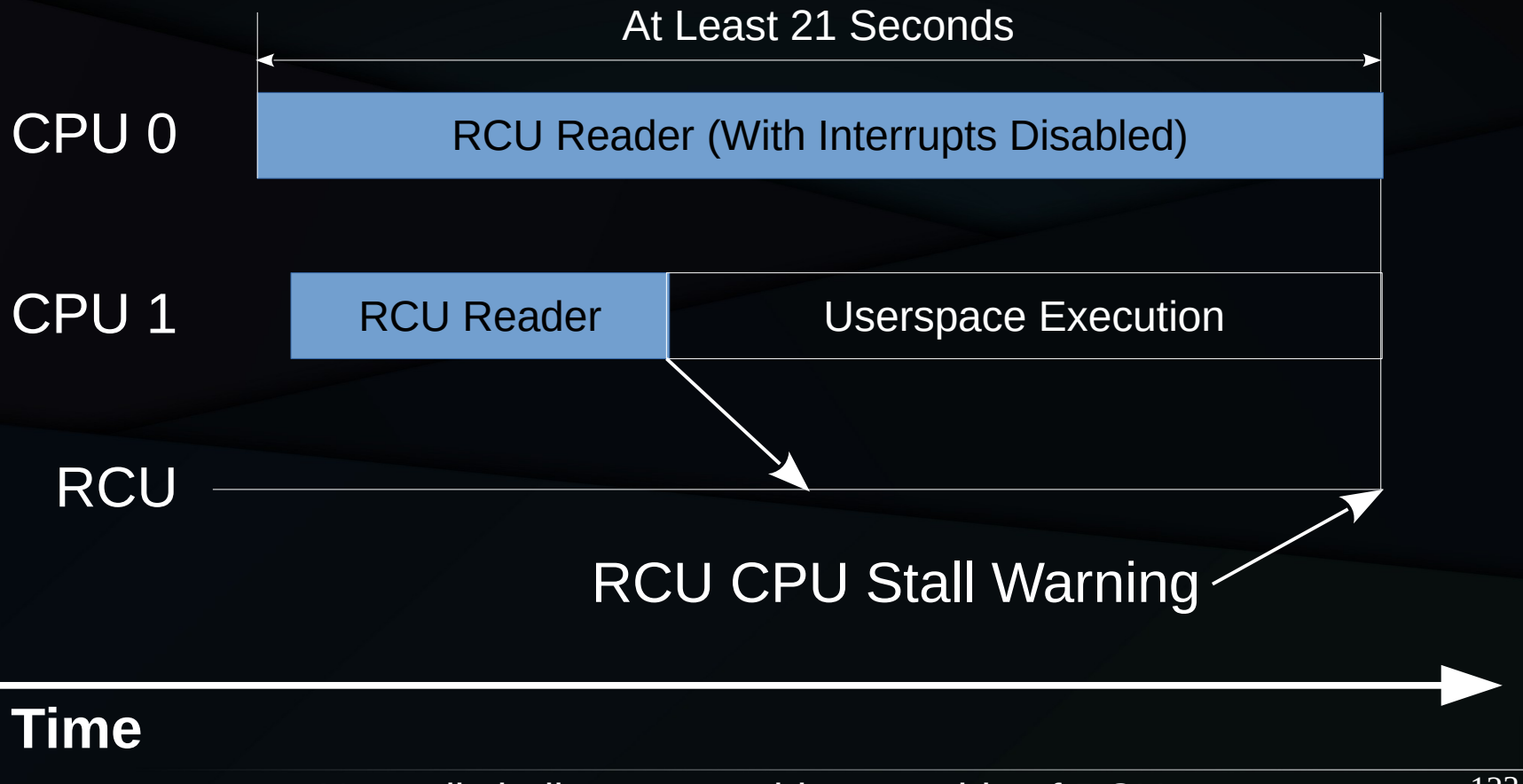
- Mark a pointer as an RCU-protected pointer with `__rcu`:
  - `void __rcu *map`
- KCSAN assertions (not just for RCU!)
  - `ASSERT_EXCLUSIVE_BITS()`
  - `ASSERT_EXCLUSIVE_ACCESS()`
  - `ASSERT_EXCLUSIVE_ACCESS_SCOPED()`
  - `ASSERT_EXCLUSIVE_WRITER()`
  - `ASSERT_EXCLUSIVE_WRITER_SCOPED()`

RCU With No Stall Warning



More complexity for CONFIG_PREEMPT=y kernels

RCU CPU Stall Warning



Normally indicates a problem outside of RCU

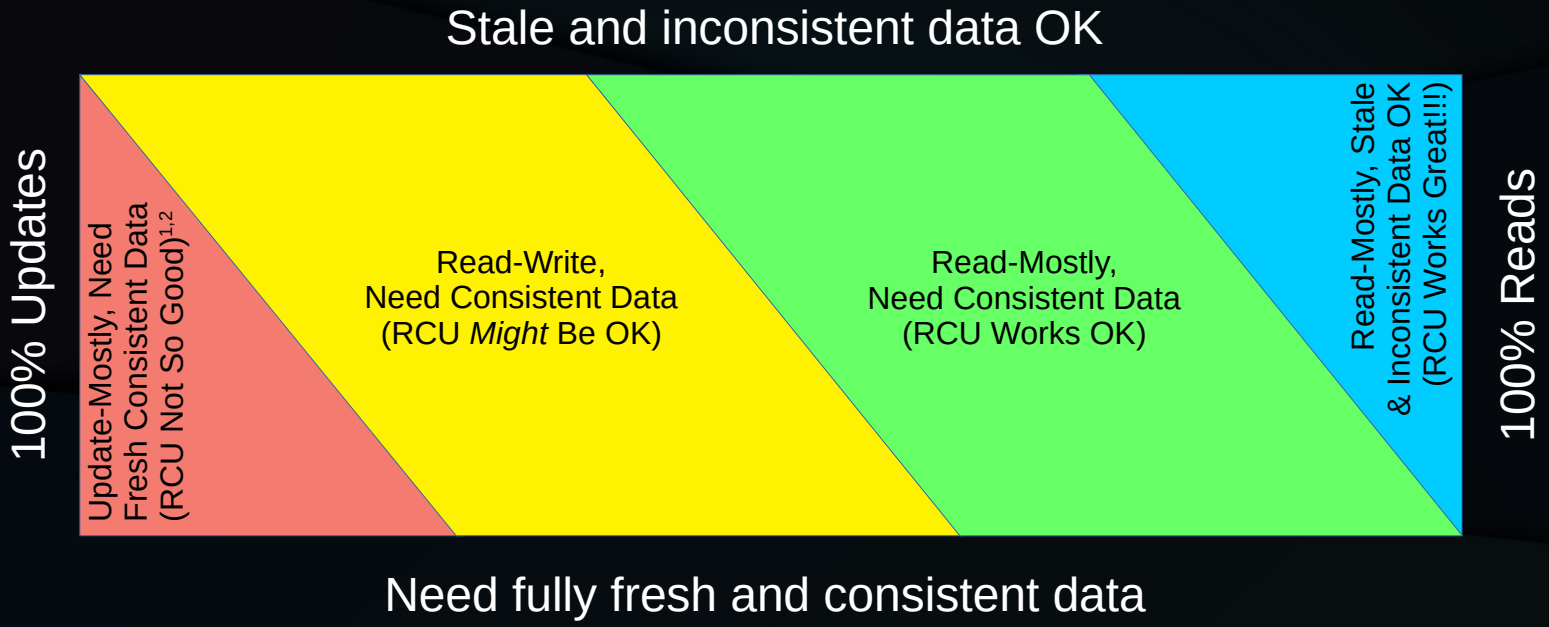
But I Don’t Want Stall Warnings!!!

- Boot with rcupdate.rcu_cpu_stall_suppress=1
- Boot with rcupdate.rcu_cpu_stall_timeout=NN (in seconds)
  - Or build with CONFIG_RCU_CPU_STALL_TIMEOUT=NN
  - 3 <= N <= 300, in seconds
- Why would you want to suppress CPU stall warnings?
  - Slow embedded system tested on faster development system
    - Increase timeout on slow embedded system (or decrease during test)
  - Throughput-based rip-and-replace cloud-computing environment
  - Embedded production environment where console output is ignored
    - Suppress warnings entirely
- But if response time matters, you do care about CPU stalls
  - Especially during development and testing: RCU CPU stall warnings find real bugs

Many Causes of RCU CPU Stalls

- For more information:
  - 2018 linux.conf.au presentation
    - Video: https://www.youtube.com/watch?v=23_GOr8Sz-E
    - Slides: https://static.sched.com/hosted_files/ossna2017/2b/stallwarning.2017.09.08a.pdf
  - Documentation in Linux-kernel source tree
    - Documentation/RCU/stallwarn.rst

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
- Two example RCU use cases:
  - Quasi reader-writer lock (consistent config)
  - Phased state change (maintenance mode)
- RCU's dirty little secret

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
- Two example RCU use cases:
  - Quasi reader-writer lock (consistent config)
  - Phased state change (maintenance mode)
- RCU's dirty little secret:
  - RCU is dead simple

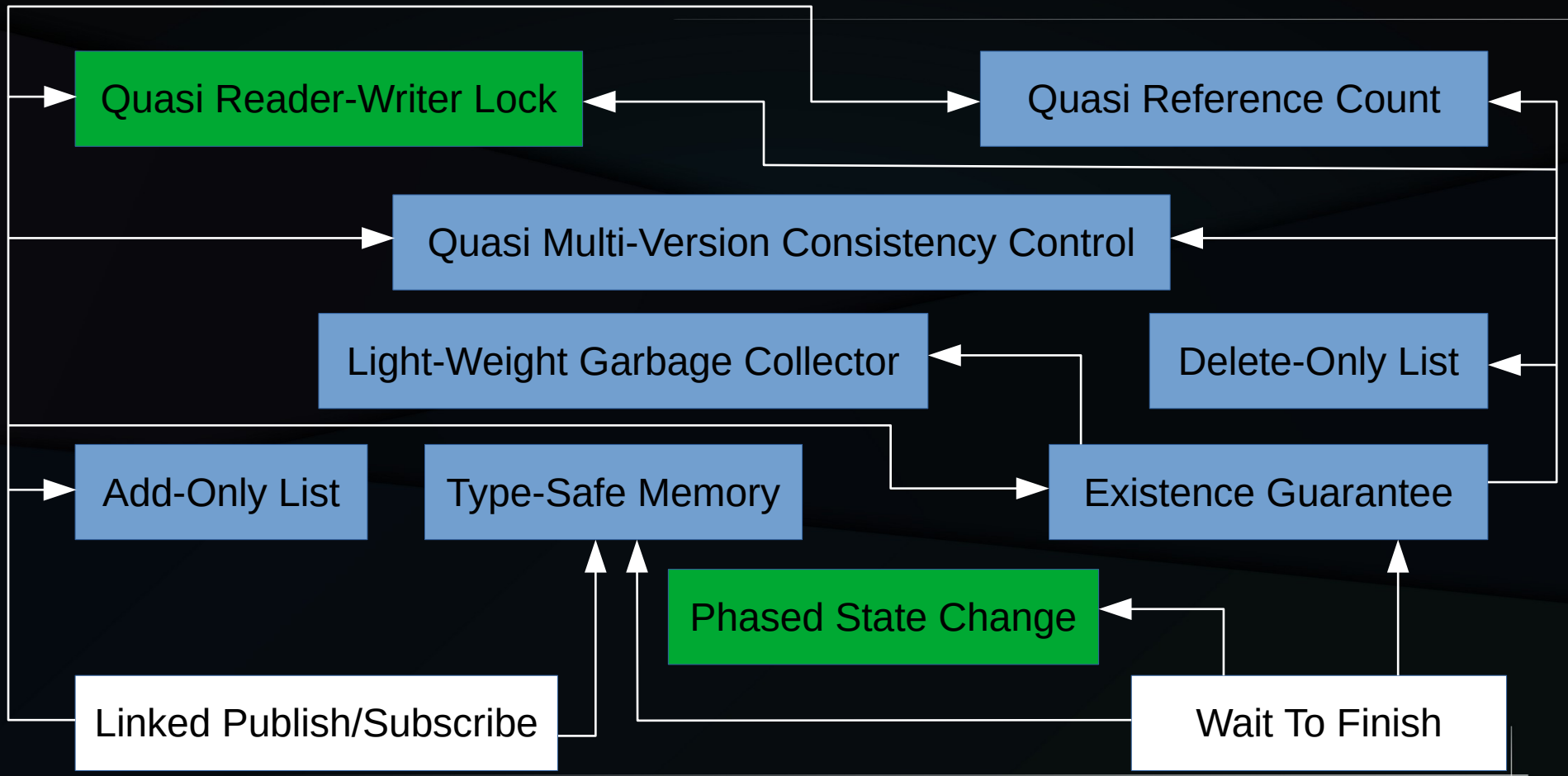
Summary

- RCU synchronizes in space as well as time
  - But the time and space aspects are deeply intertwined
  - Enables near-zero-cost read-side synchronization
- Two example RCU use cases:
  - Quasi reader-writer lock (consistent config)
  - Phased state change (maintenance mode)
- 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.

You Are Here; More in February



For More Information

- “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/