Paul E. McKenney, Facebook

Northern Arizona University NAU 499/599, March 23, 2021

Does RCU Really Work?

And if so, how would we know?

© 2021 Facebook Corporation

RCU Usage: Overview

- What is RCU supposed to do?
- Is there a real problem?
- What is currently being done?
- Formal verification and Linux-kernel regression tests?
- Formal verification and memory models?
- Could better things happen???
- Summary

What is RCU Supposed to Do?

Hide

What is RCU Supposed to Do?

- Example multithreaded rwlock use case
- RCU semantics: English, usage restrictions, “show me the code”, graphical, memory ordering, “exercise the code” and temporal/spatial.
- RCU performance
- RCU usage

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

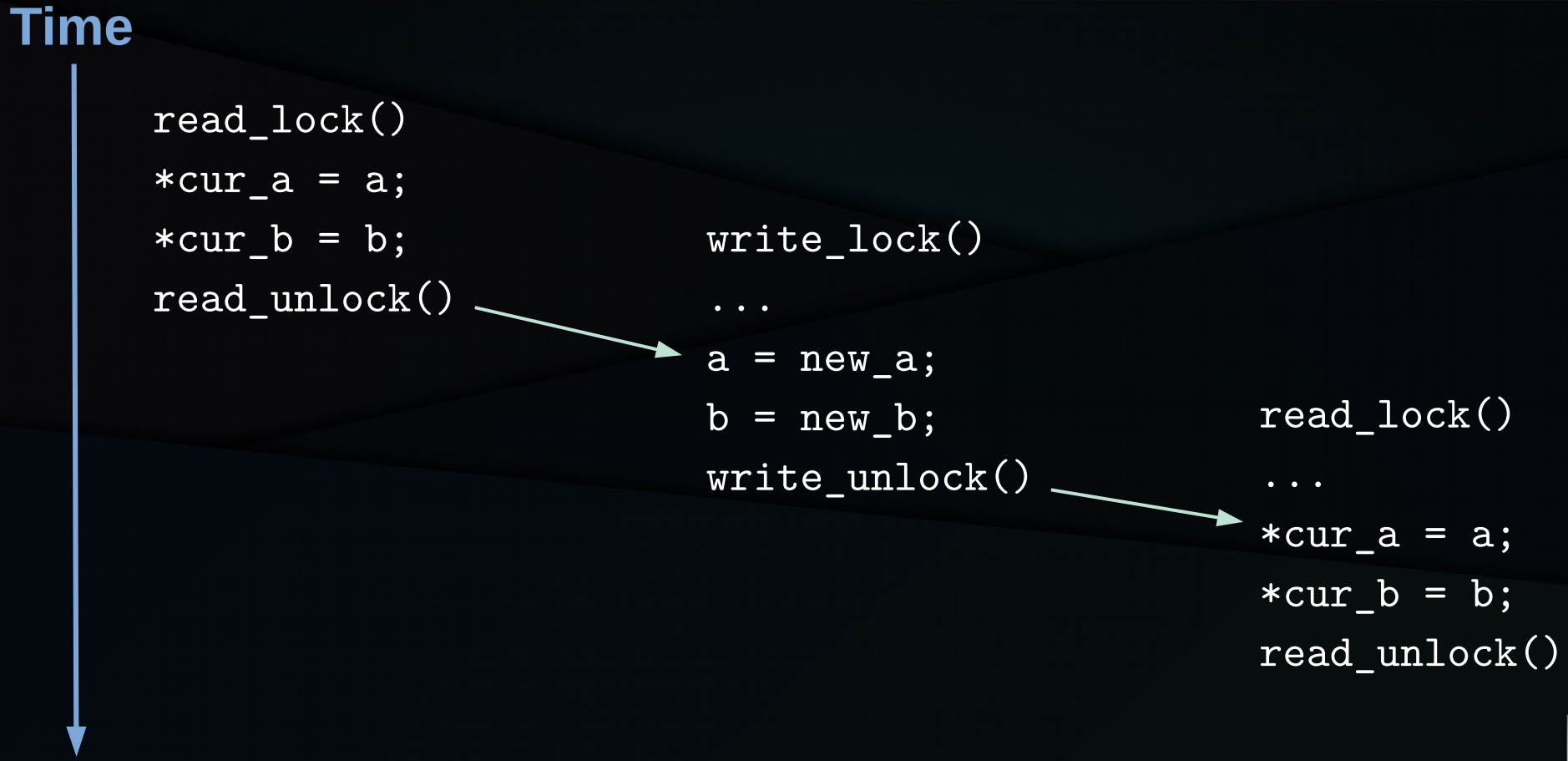
```
DEFINE_RWLOCK(myrwlock);
int a, b; // Current configuration values

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

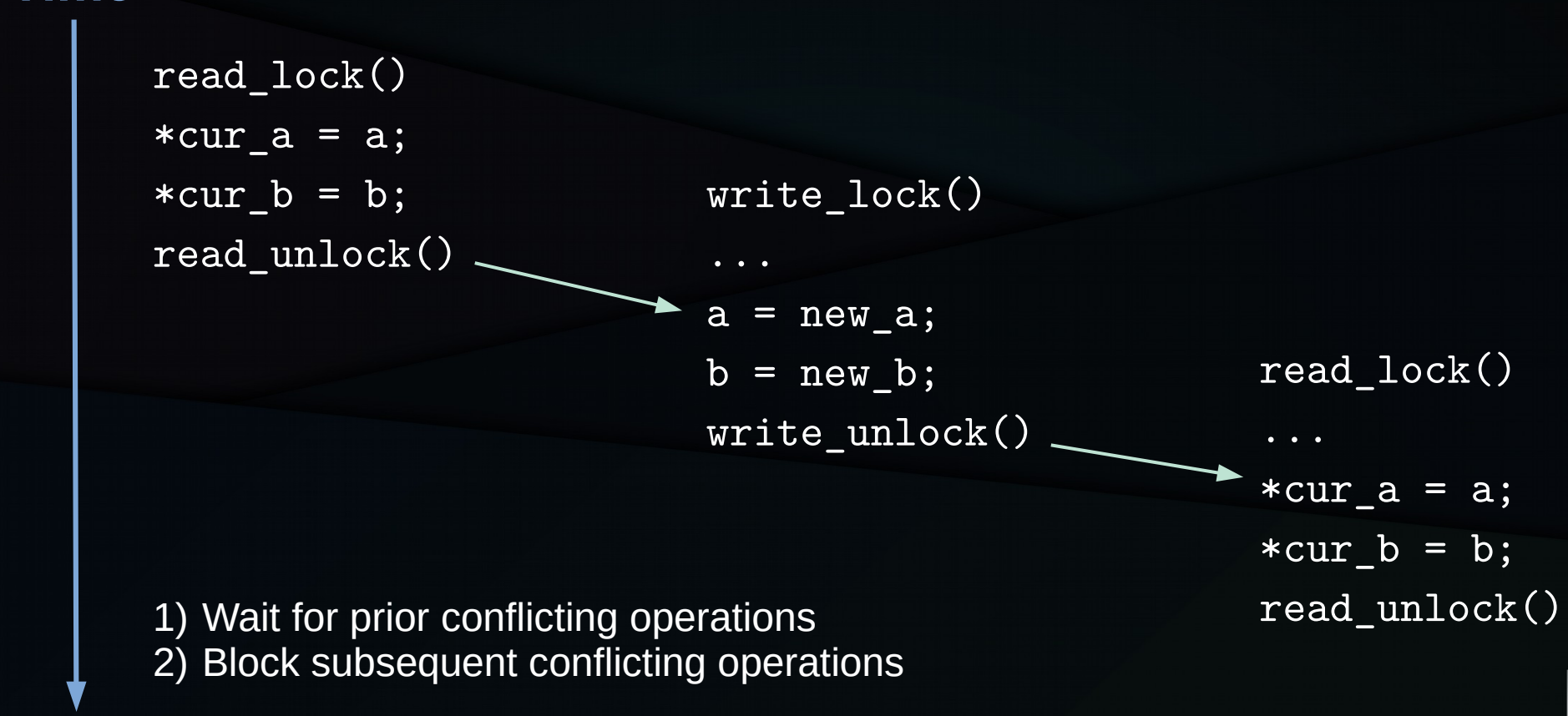
```
void set(int new_a, int new_b)
{
    write_lock(&myrwlock);
    a = new_a;
    b = new_b;
    write_unlock(&myrwlock);
}
```

Reader-Writer Lock Semantics

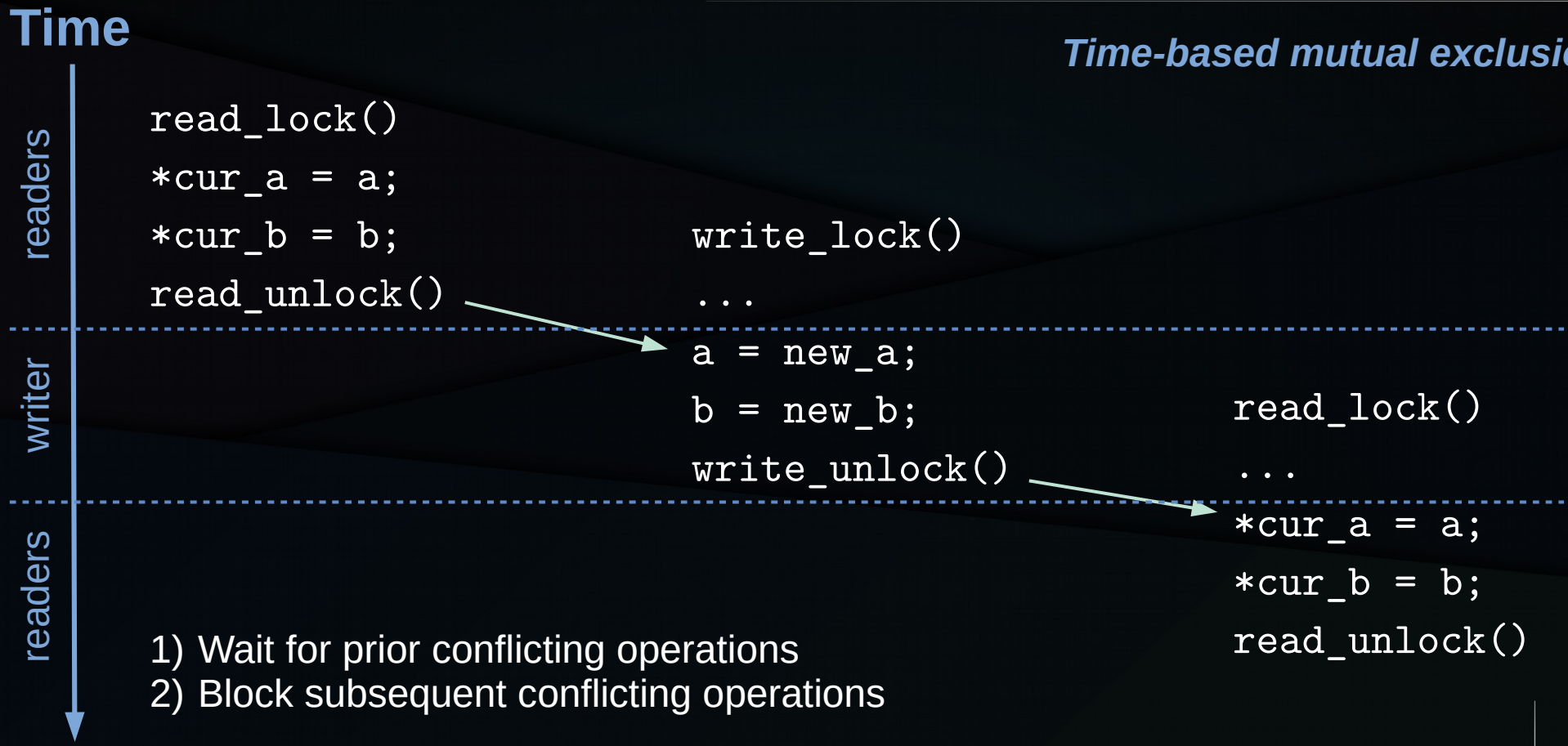
Reader-Writer Lock Semantics



Reader-Writer Lock Semantics



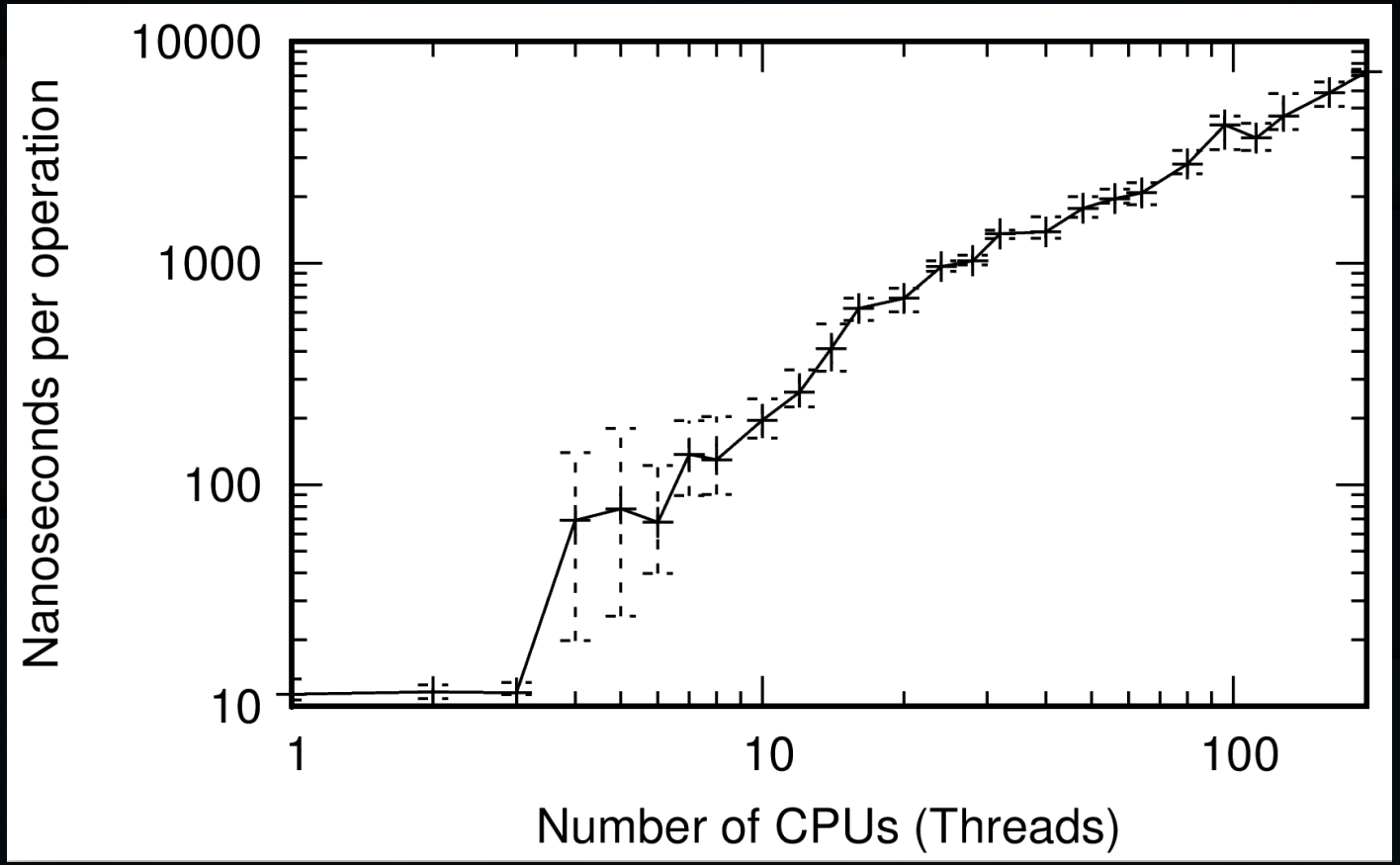
Reader-Writer Lock Semantics



Time-based mutual exclusion

Reader-Writer Lock Performance

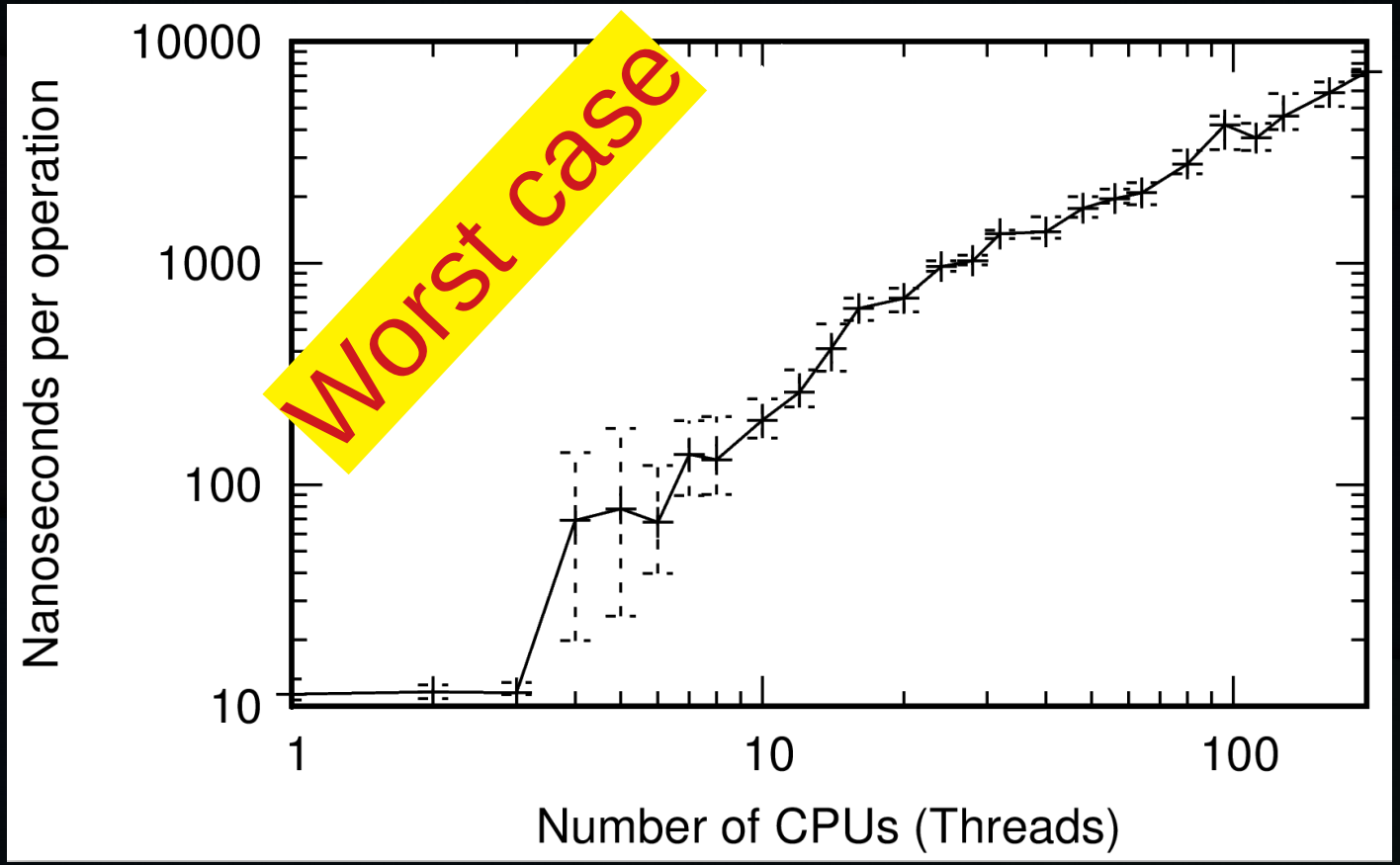
Reader-Writer Lock Performance



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

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

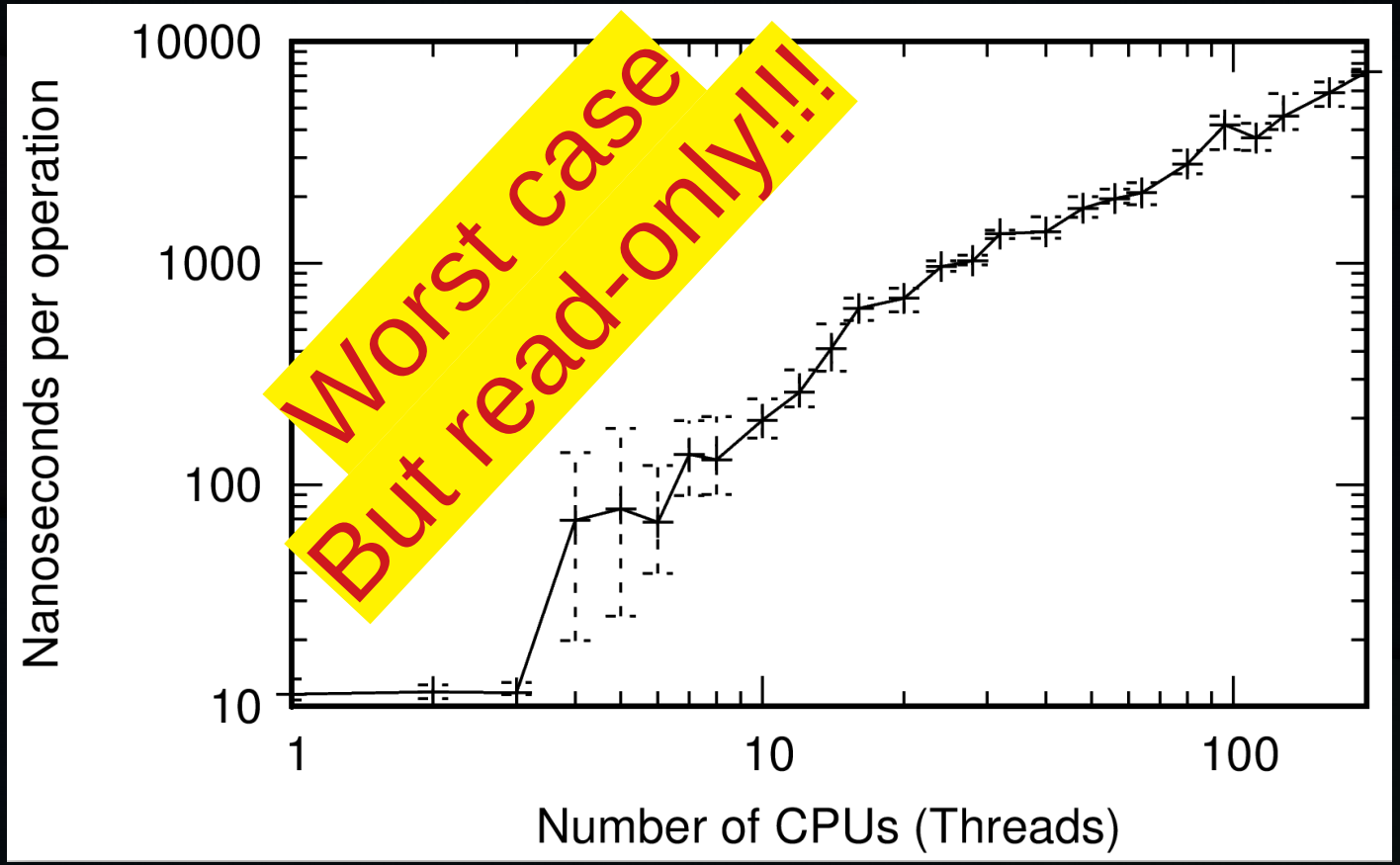
Reader-Writer Lock Performance



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

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

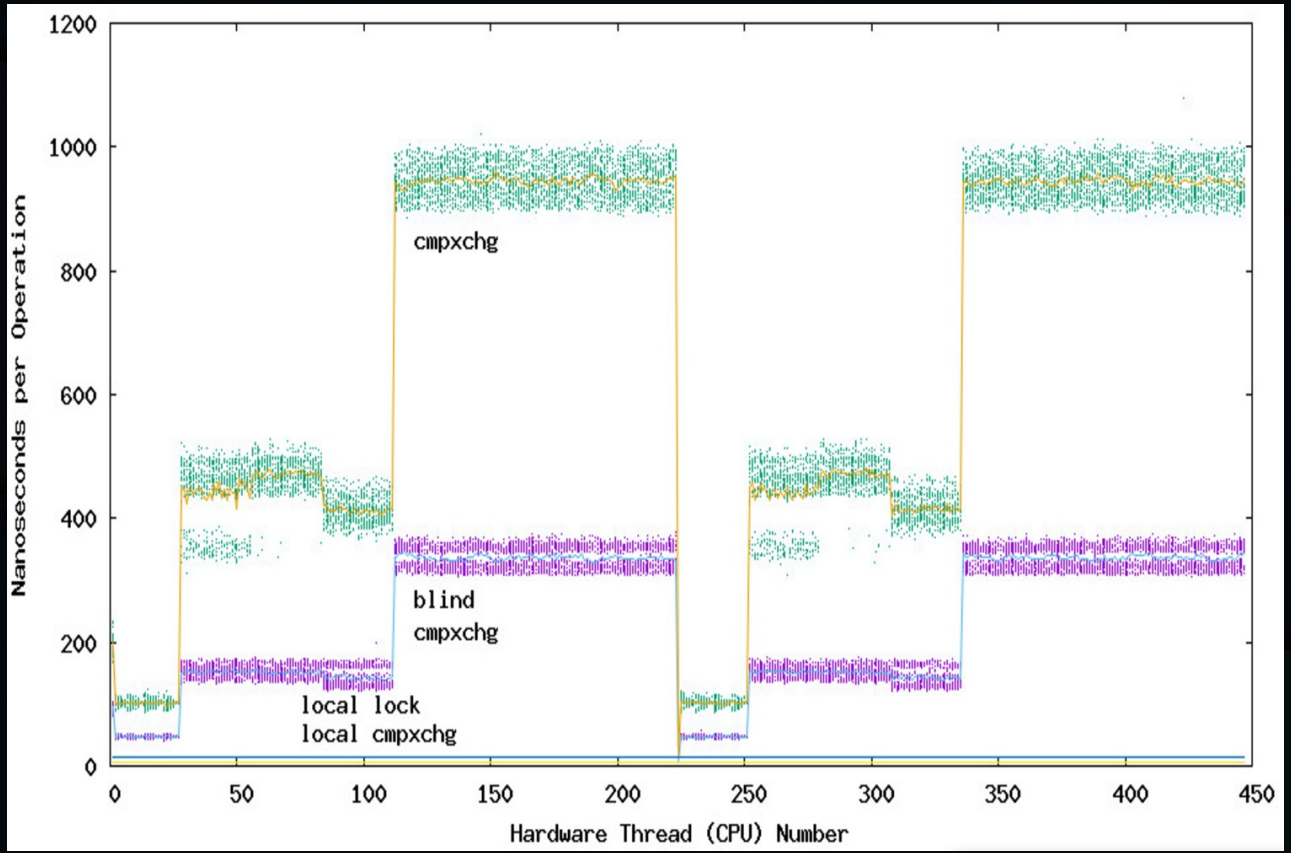
Reader-Writer Lock Performance



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

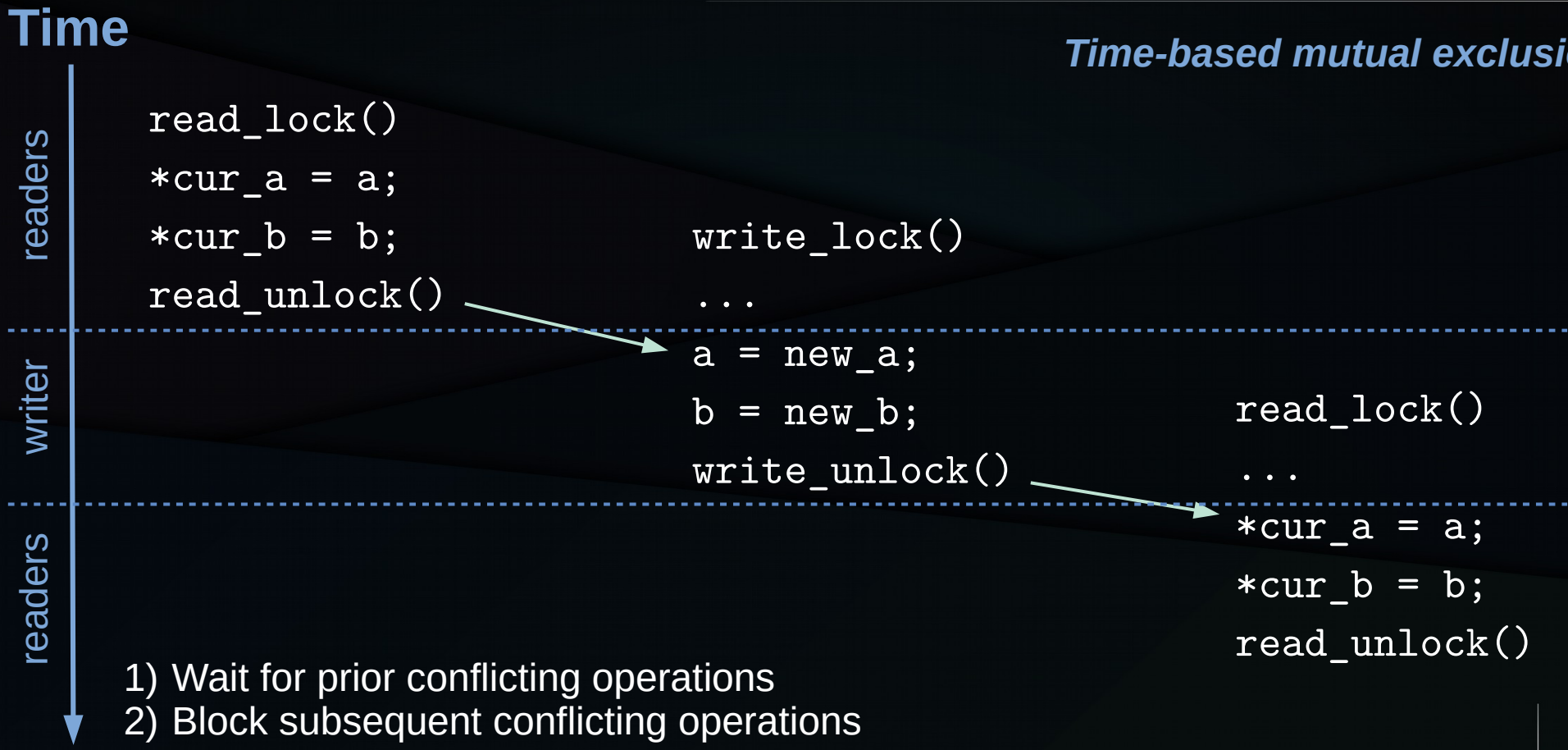


Intel(R) Xeon(R) Platinum 8176 CPU @ 2.10GHz

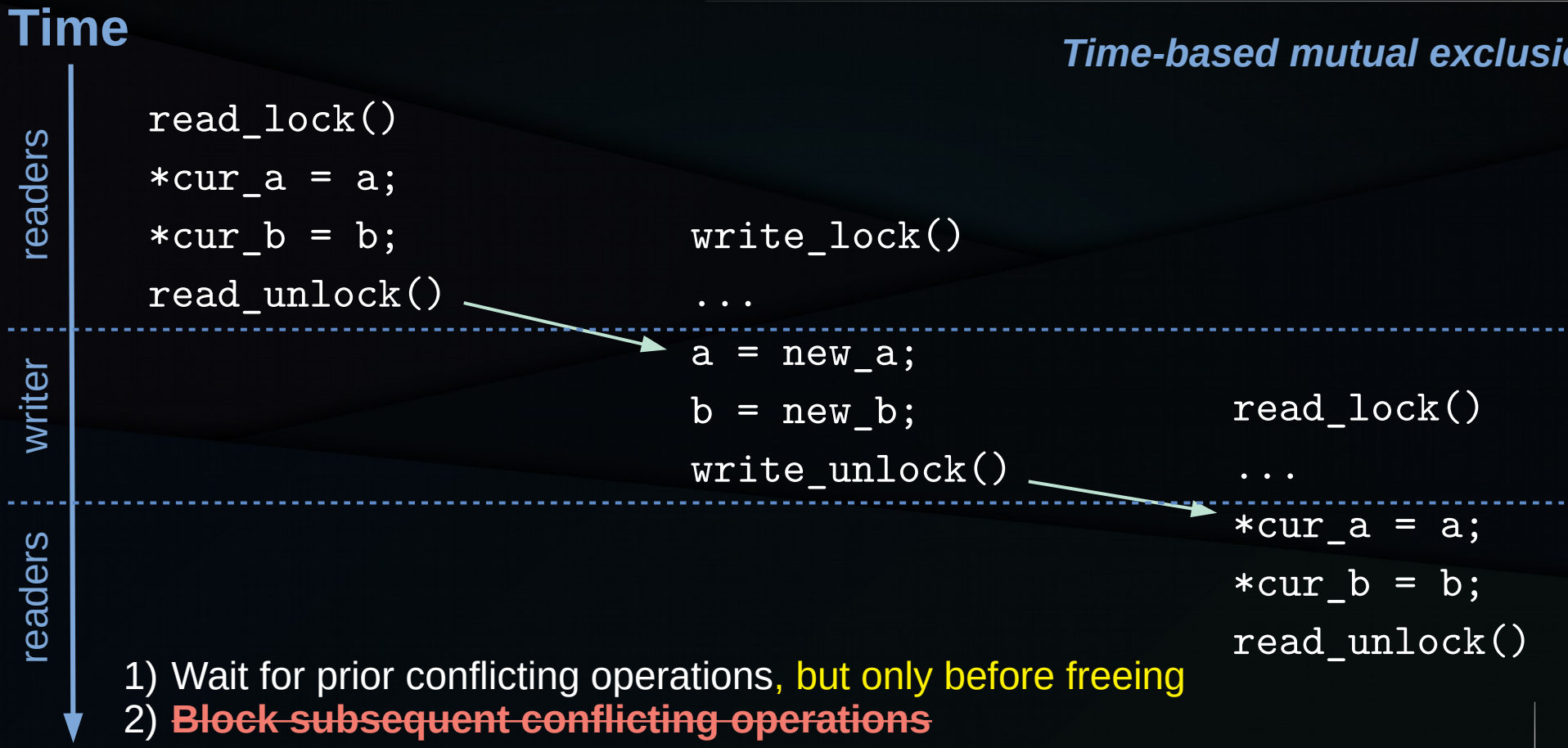
perfbook CodeSamples/cpu/data/hps.2020.03.04b/cachetorture-latency-scatter.jpg

Lighter Weight Semantics

Lighter Weight Semantics

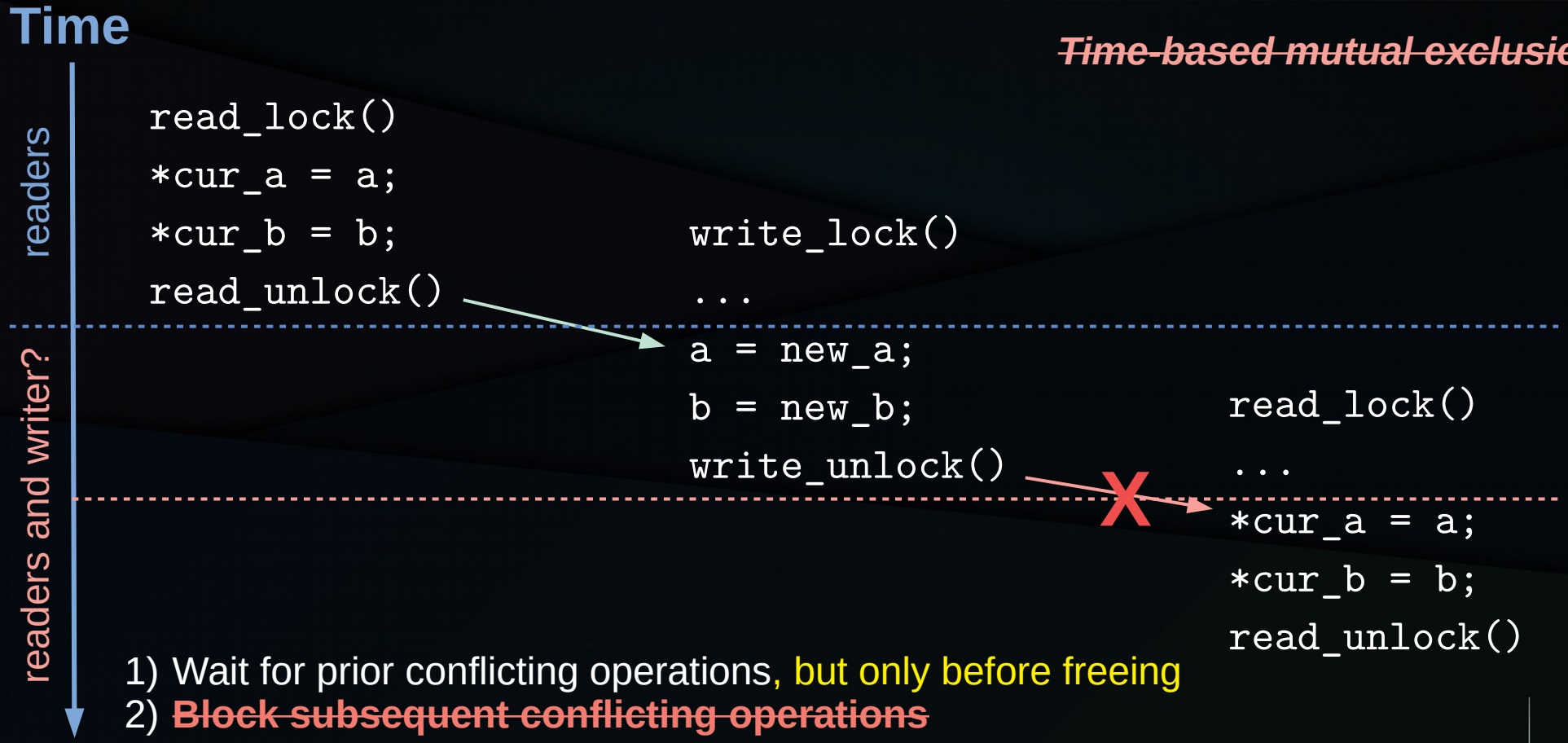


Lighter Weight Semantics



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

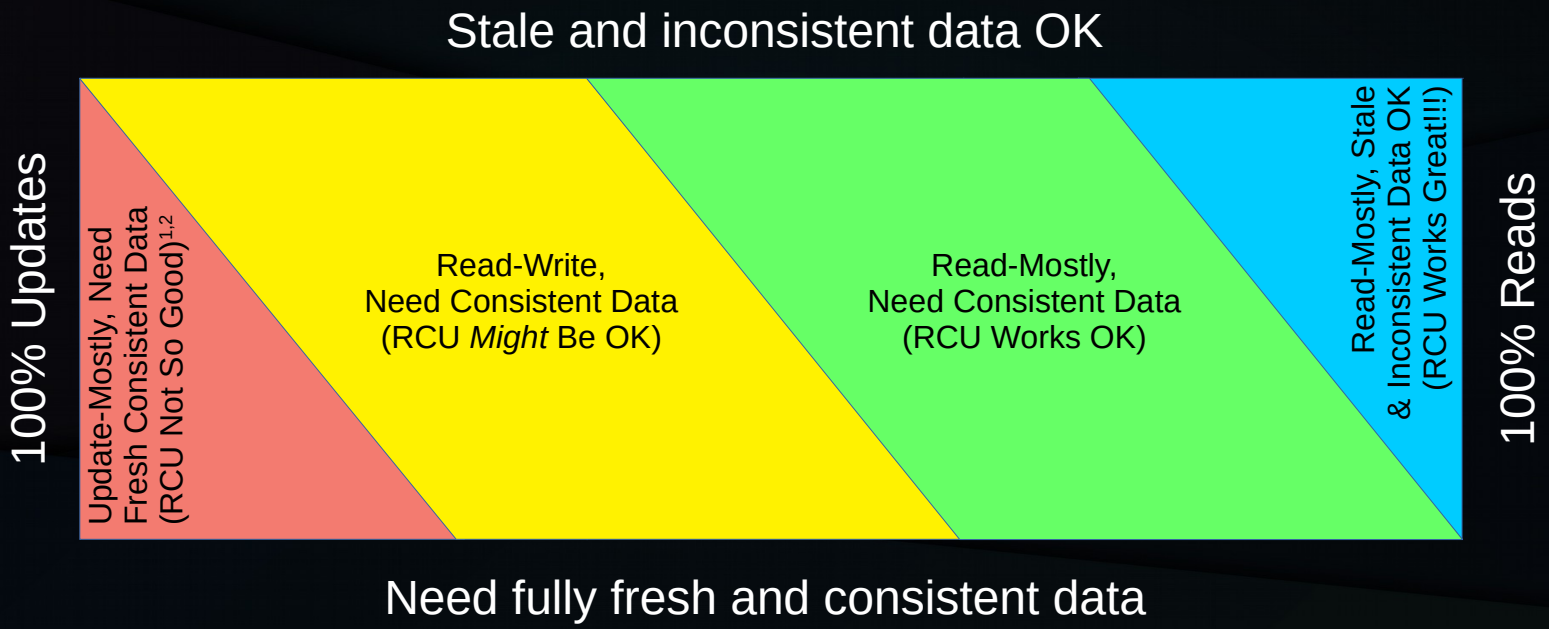
Lighter Weight Semantics???



RCU Semantics (English)

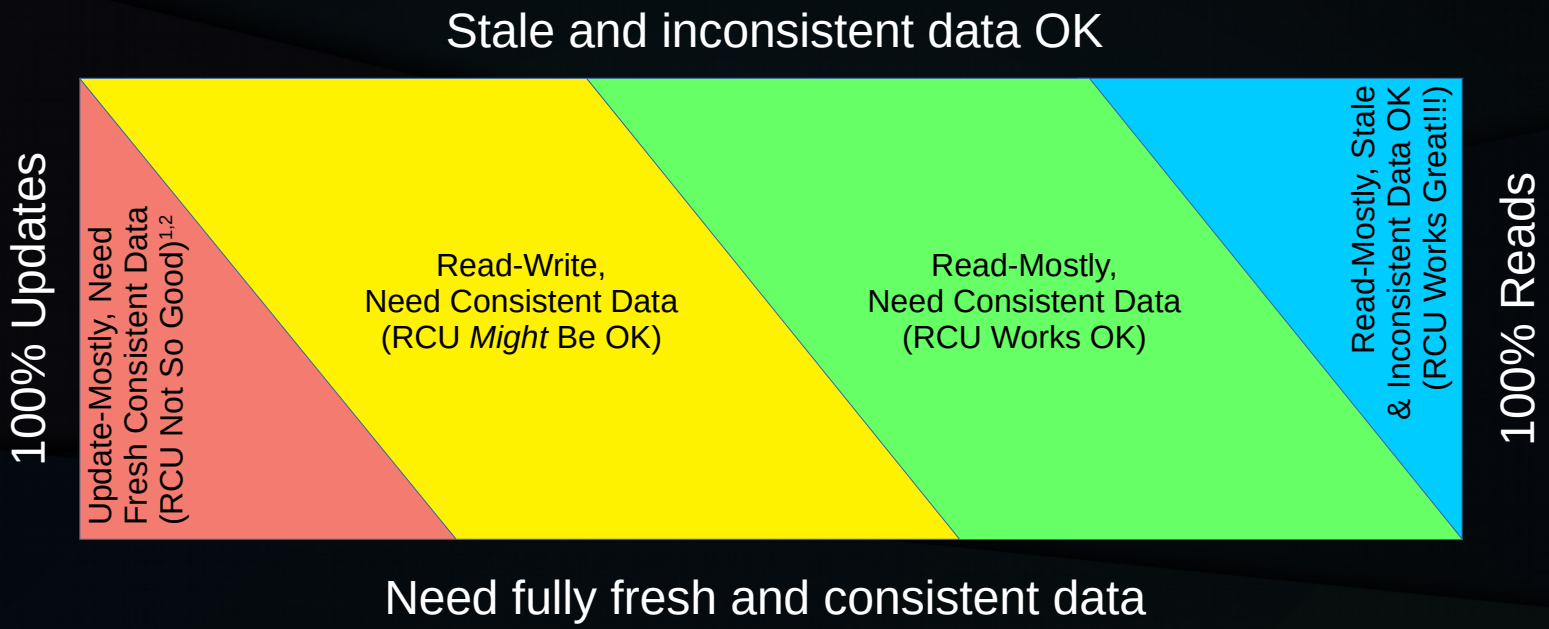
- Semantics weakened from reader-writer locking:
  1) Writers wait for read-holders, but only before freeing
  2) ~~Readers wait for the writer-holders~~
- Compensate for weak temporal semantics by adding *restrictions and spatial semantics* to RCU use cases
- Restated for RCU:
  - If synchronize_rcu()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

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)

nothin
g

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: Create new structure & update pointer
- Free old structure “when it is safe to do so”

Core RCU API

- `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/

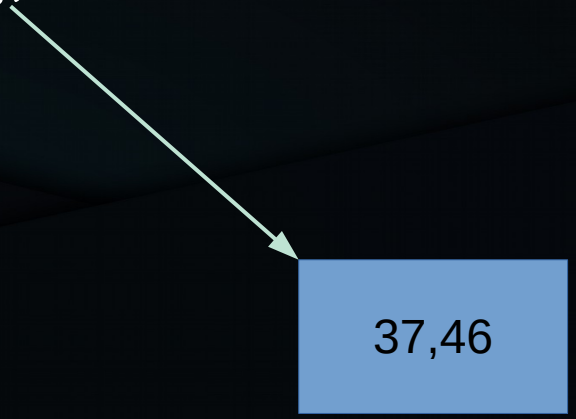
RCU Use Case: Reader

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();
}
```

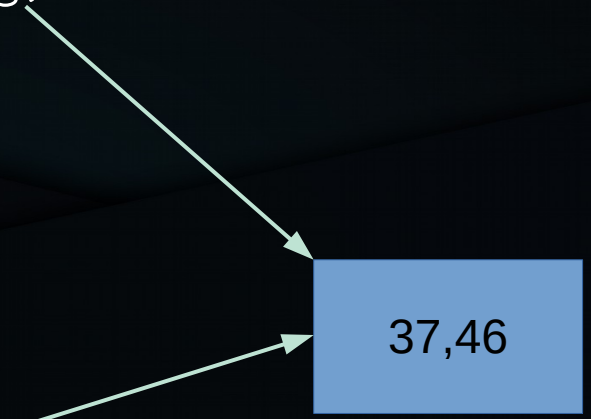
RCU Use Case: Reader

```
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();
}
```



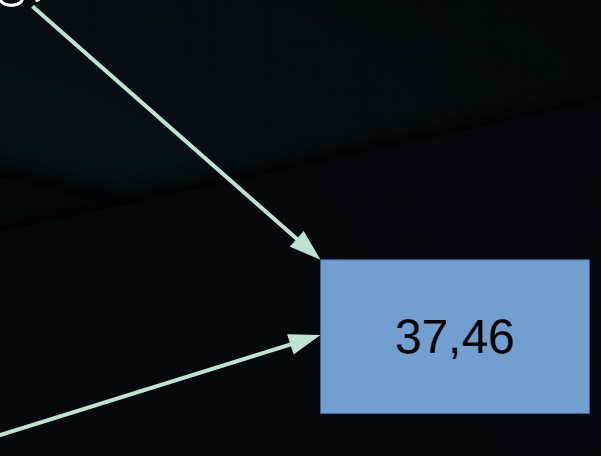
RCU Use Case: Reader

```
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();
}
```

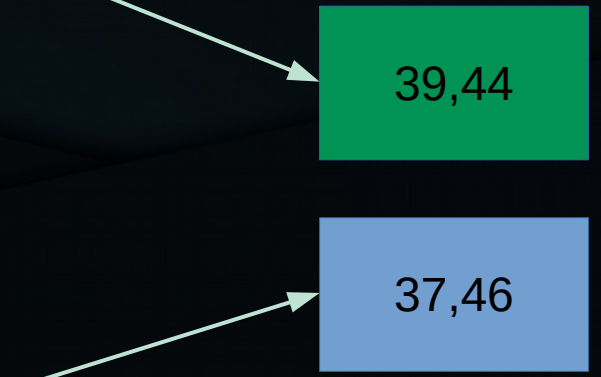


RCU Use Case: Reader

```
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 new_a, int new_b)
{
    struct myconfig *mcp = kmalloc(...);
    struct myconfig *oldmcp;

    BUG_ON(!mcp);
    mcp->a = new_a;
    mcp->b = new_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 new_a, int new_b)
{
    struct myconfig *mcp = kmalloc(...);
    struct myconfig *oldmcp;

    BUG_ON(!mcp);
    mcp->a = new_a;
    mcp->b = new_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 new_a, int new_b)
{
    struct myconfig *mcp = kmalloc(...);
    struct myconfig *oldmcp;

    BUG_ON(!mcp);
    mcp->a = new_a;
    mcp->b = new_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 new_a, int new_b)
{
    struct myconfig *mcp = kmalloc(...);

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

RCU Use Case: Atomic Writer

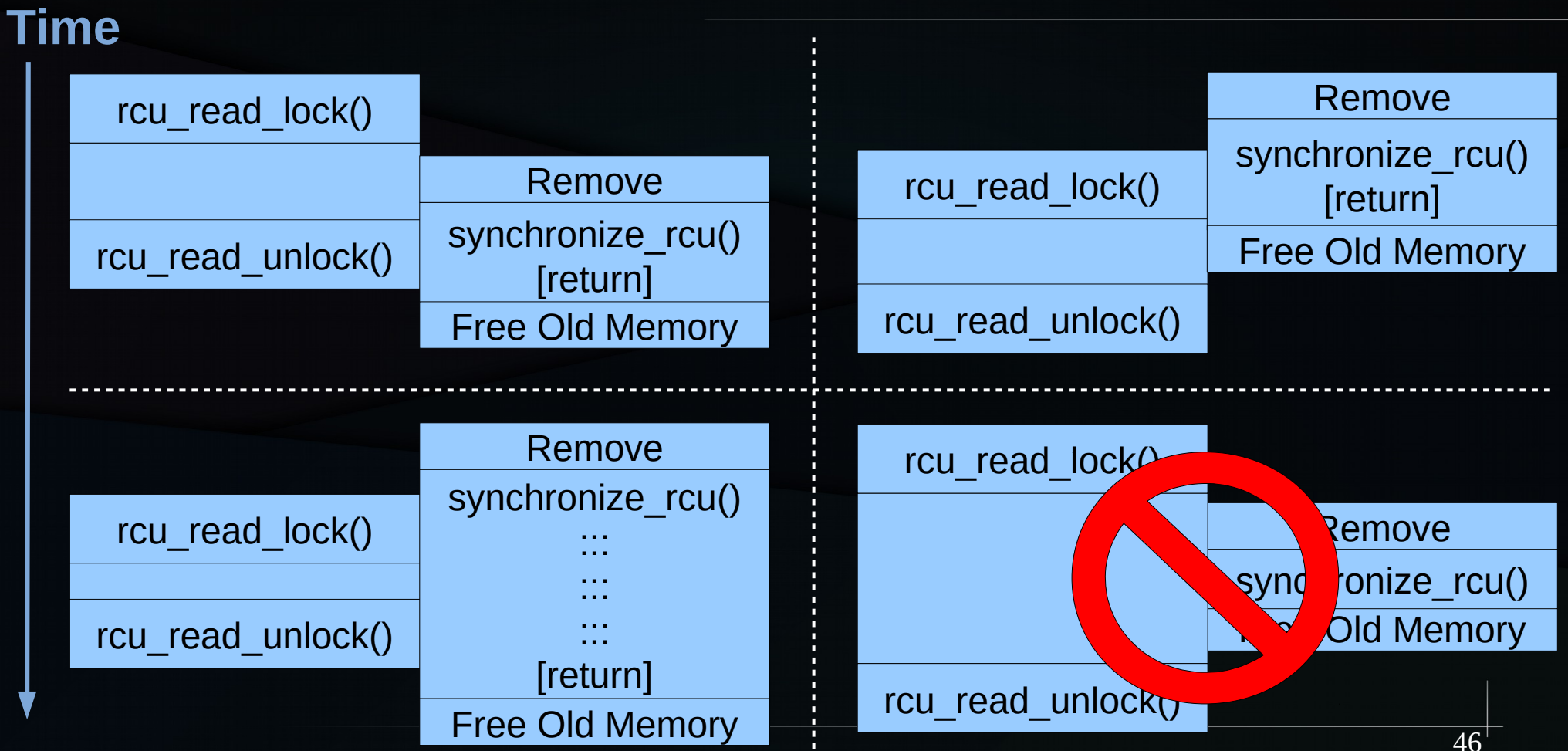
```
void set(int new_a, int new_b)
{
    struct myconfig *mcp = kmalloc(...);

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

RCU Semantics

RCU Semantics (English, Redux)

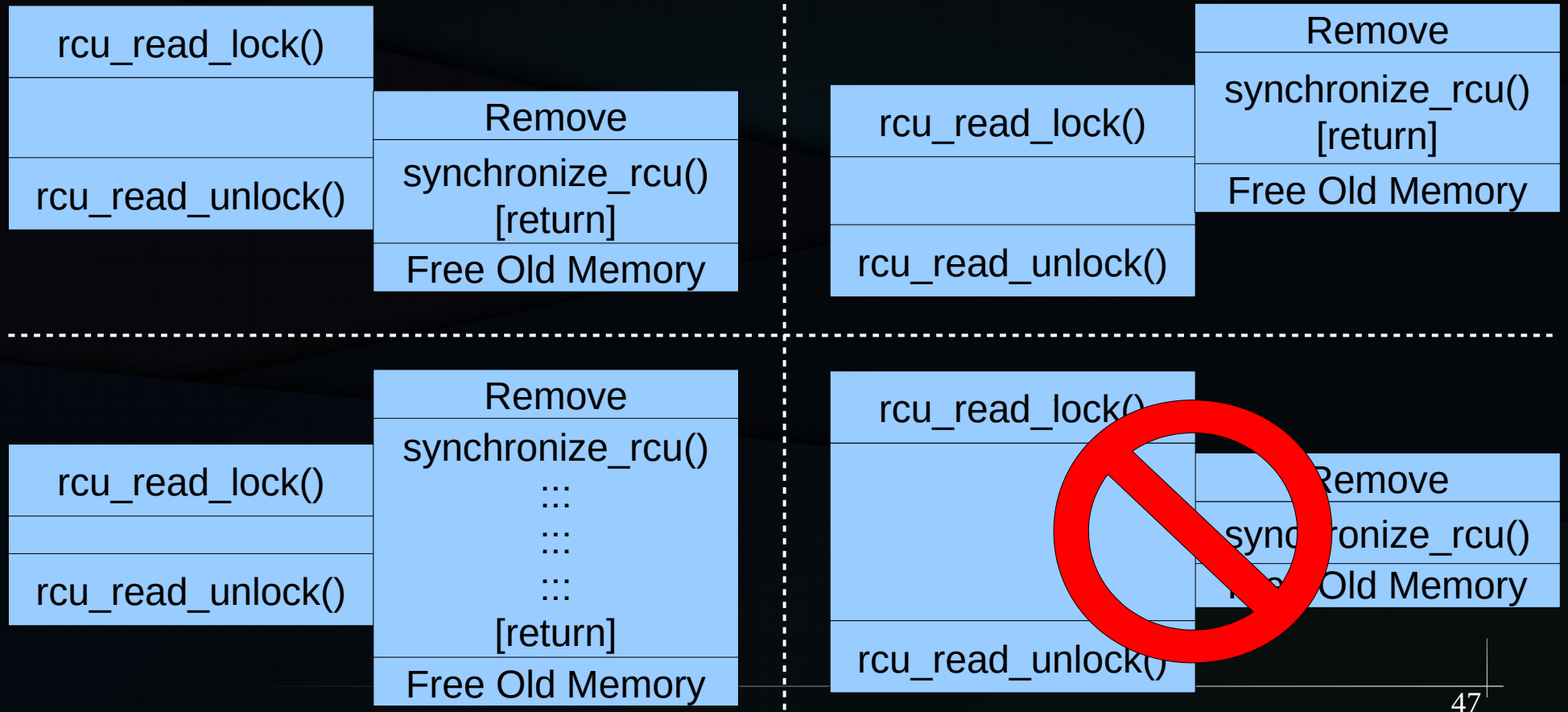
- Semantics weakened from reader-writer locking:
  1) Writers wait for read-holders, but only before freeing
  2) ~~Readers wait for the writer-holders~~
- Compensate for weak temporal semantics by adding *restrictions and spatial semantics* to RCU use cases
- Restated for RCU:
  - If synchronize_rcu()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

RCU Semantics (Graphical)

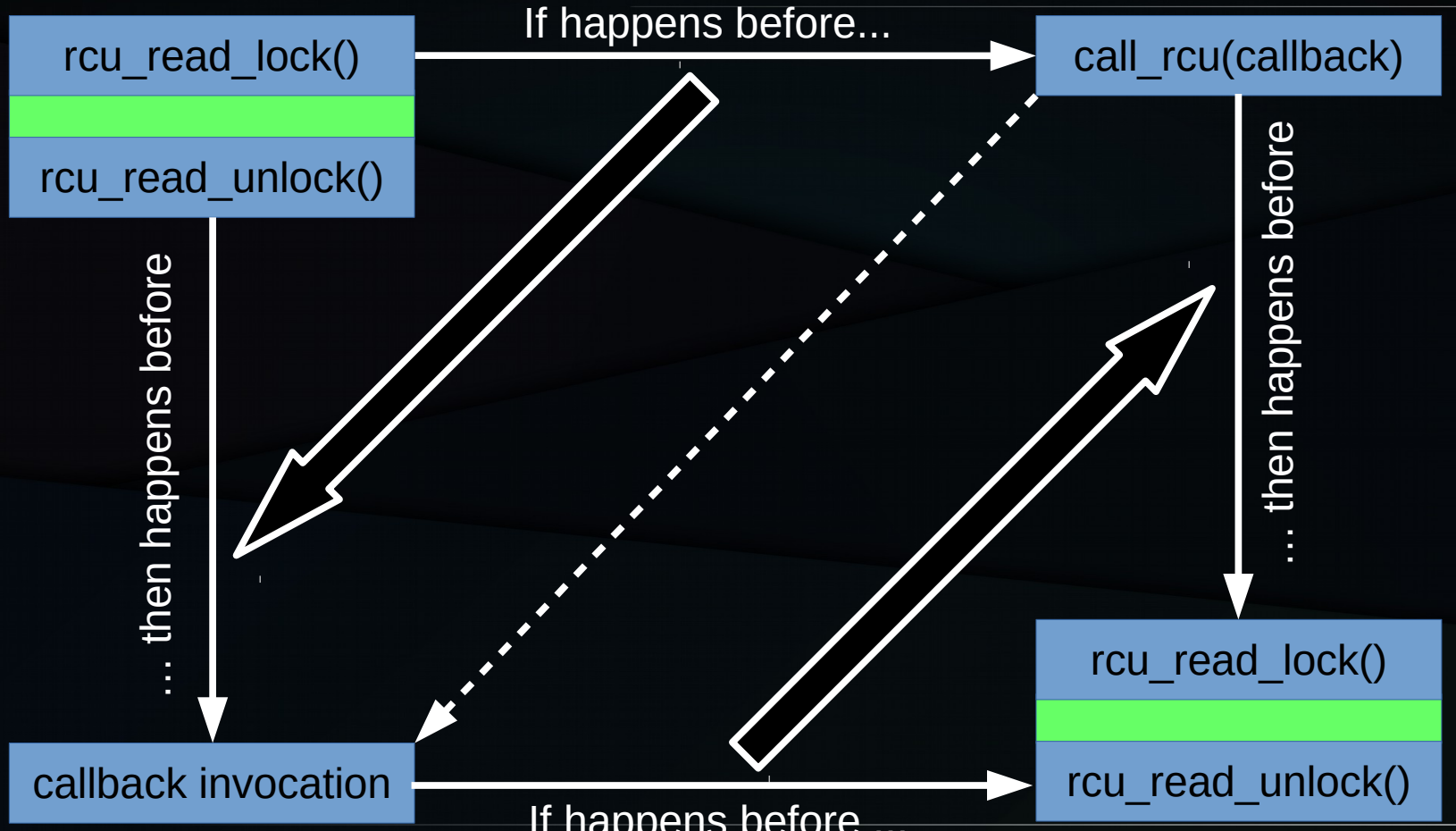


RCU Semantics (Graphical)

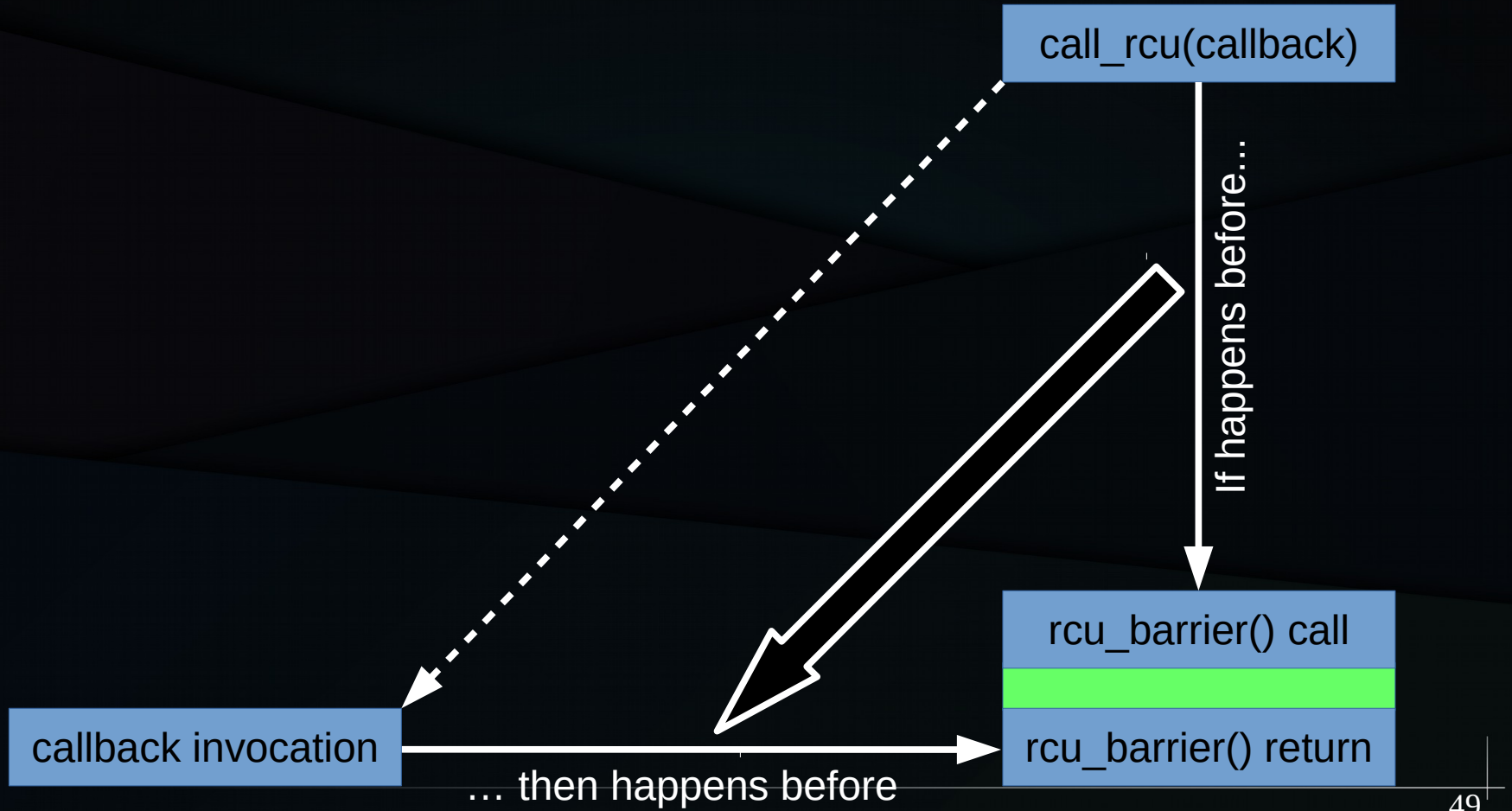
Time (really ordering)



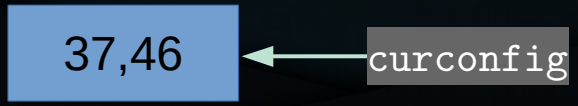
RCU Semantics (Memory Ordering)



RCU Semantics (Memory Ordering)



RCU Semantics (Exercise The Code)

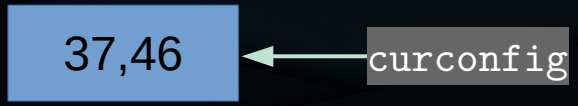



```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

```
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)

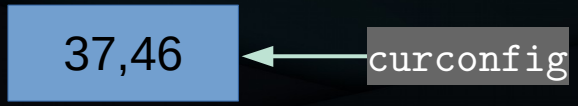



```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

```
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)

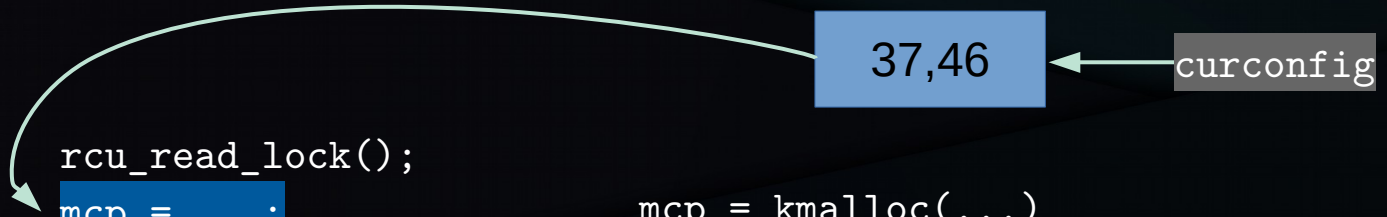



```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

```
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)

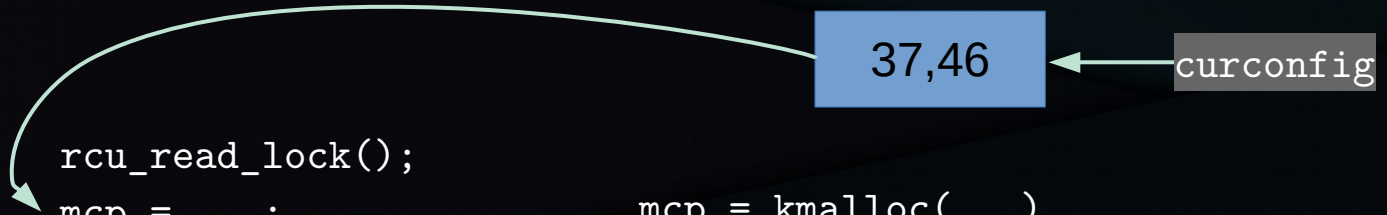



```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a;
*cur_b = mcp->b;
rcu_read_unlock();
```

```
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)

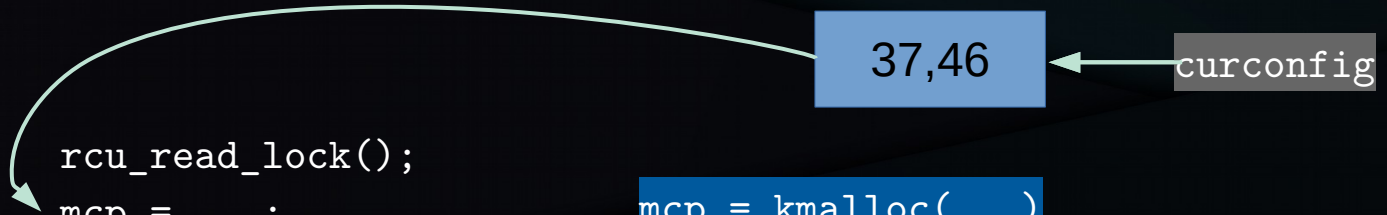



```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a; (37)
*cur_b = mcp->b;
rcu_read_unlock();
```

```
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)

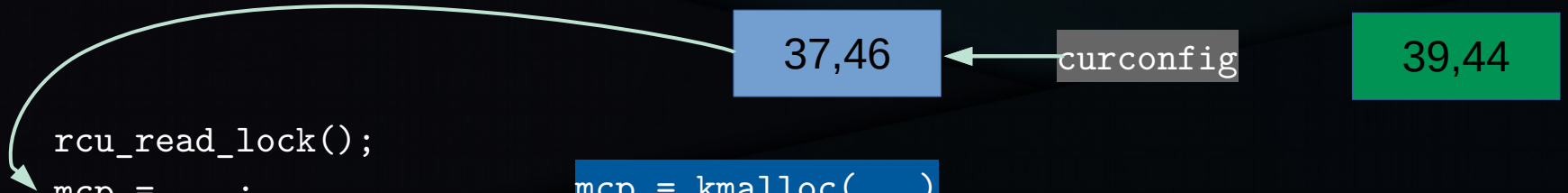



```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a; (37)
*cur_b = mcp->b;
rcu_read_unlock();
```

```
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)

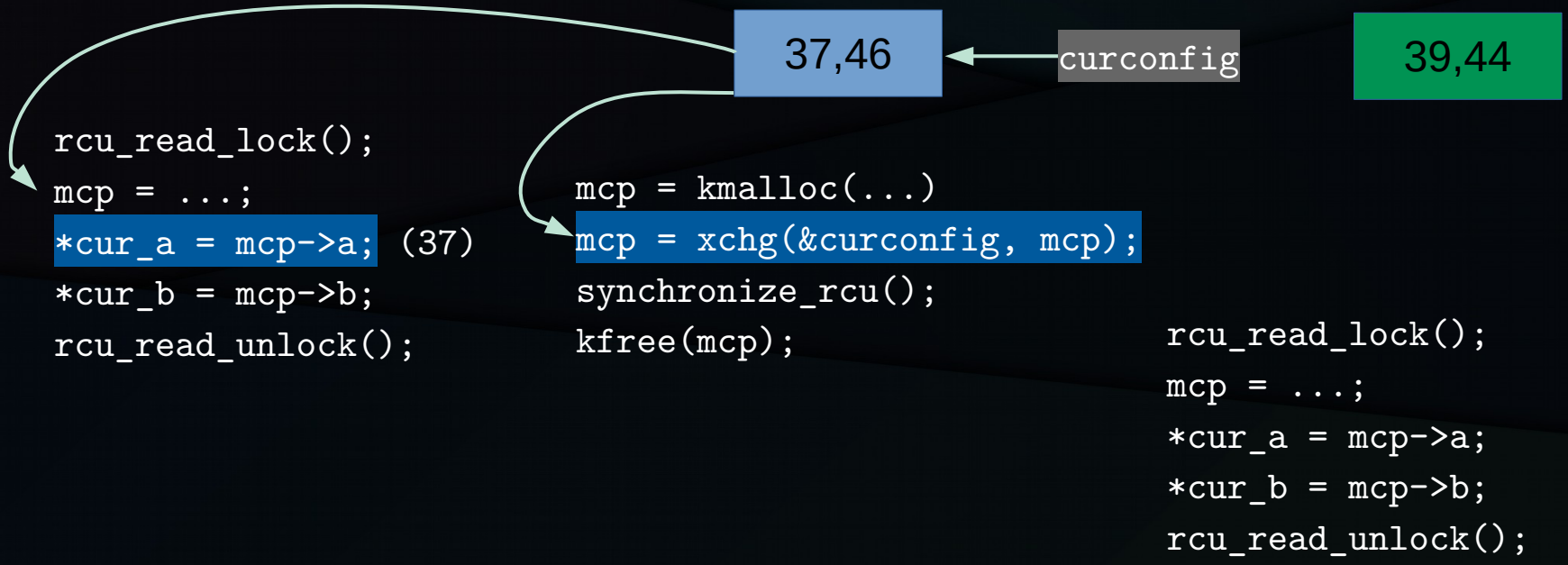



```
rcu_read_lock();
mcp = ...;
*cur_a = mcp->a; (37)
*cur_b = mcp->b;
rcu_read_unlock();
```

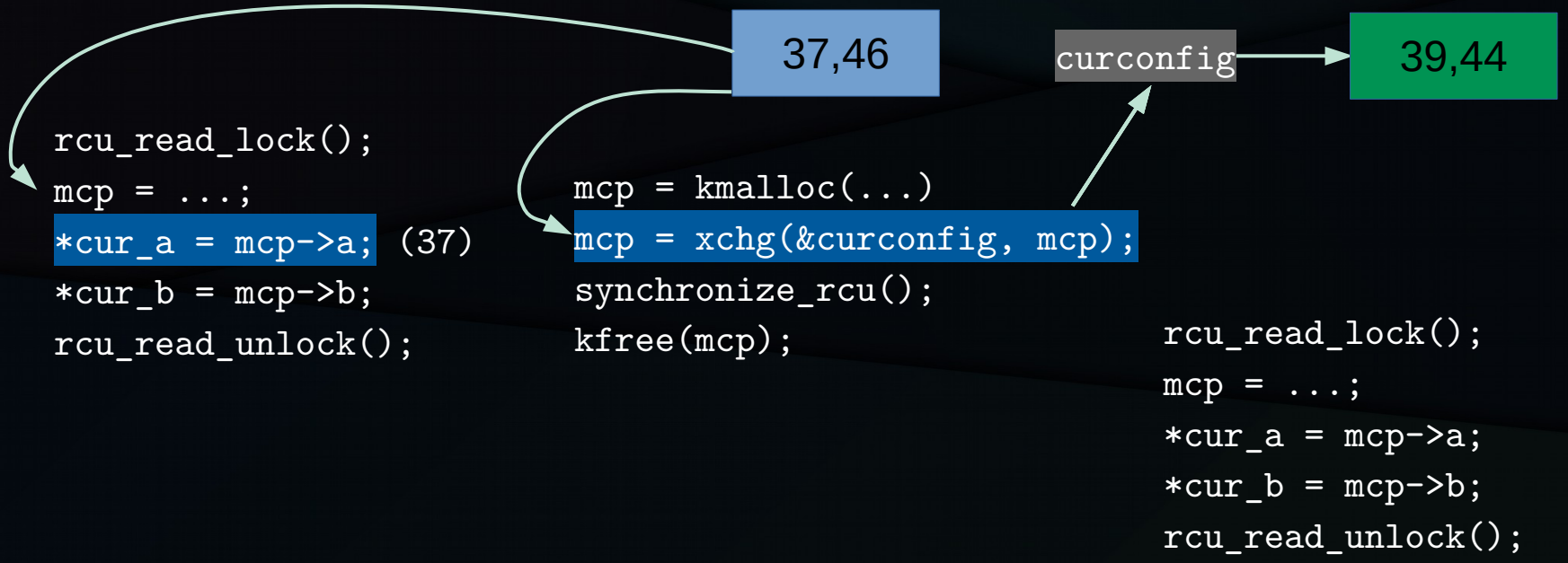
```
mcp = kmalloc(...)
mcp = xchg(&curconfig, mcp);
synchronize_rcu();
kfree(mcp);
```

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

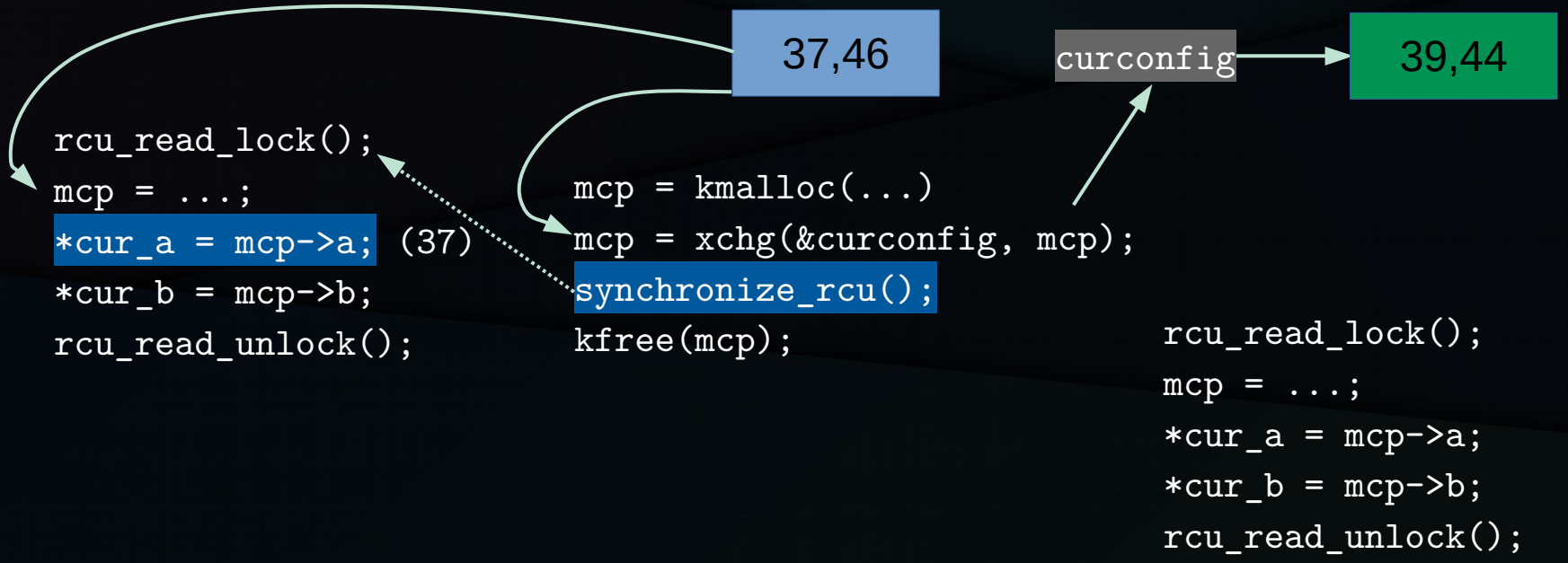
RCU Semantics (Exercise The Code)



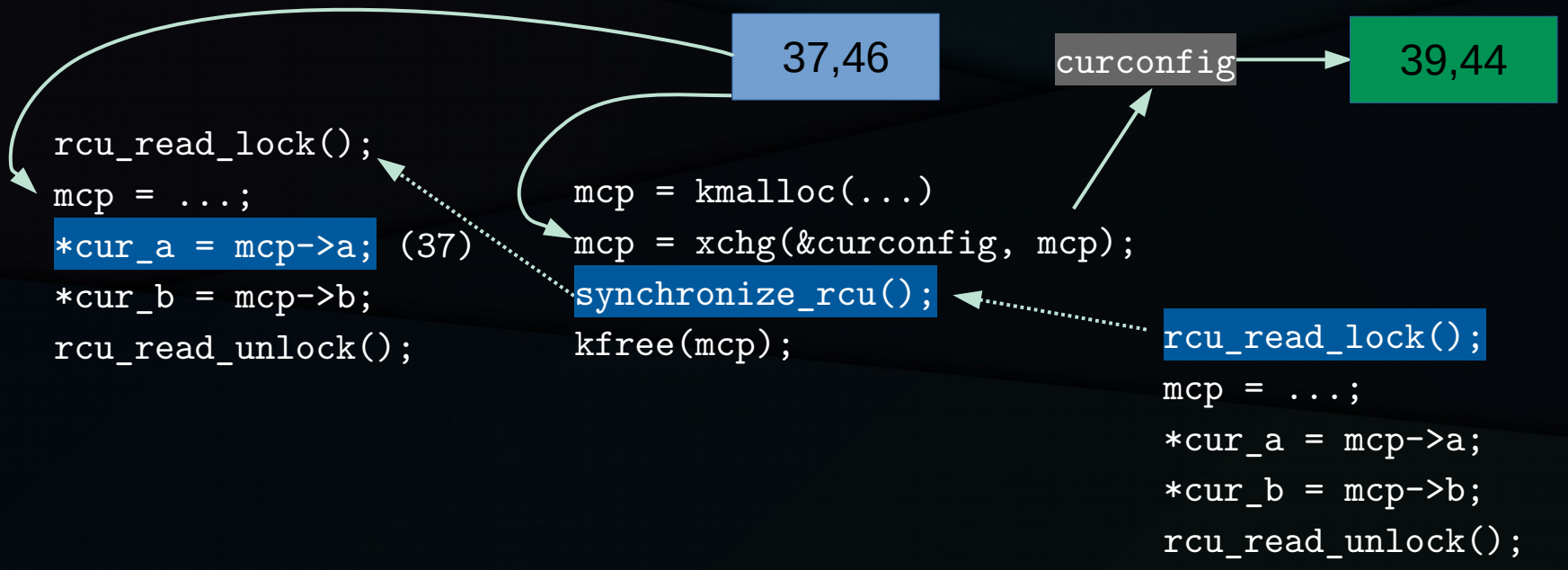
RCU Semantics (Exercise The Code)



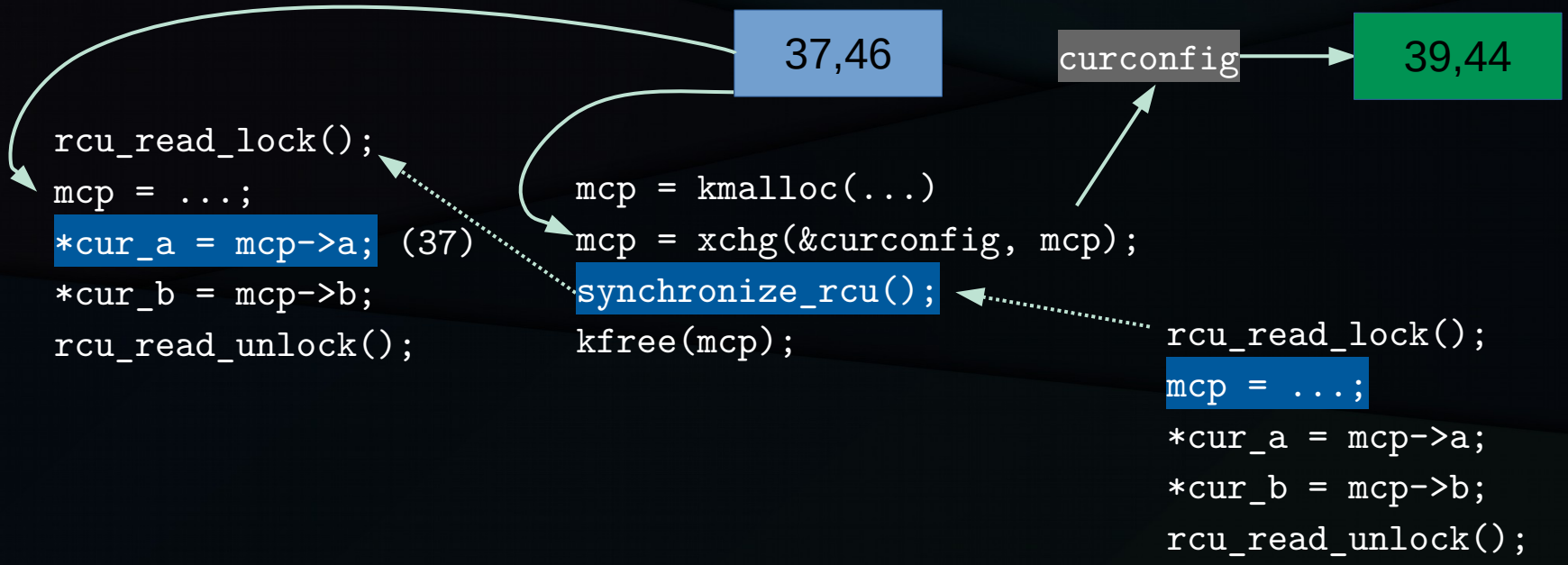
RCU Semantics (Exercise The Code)



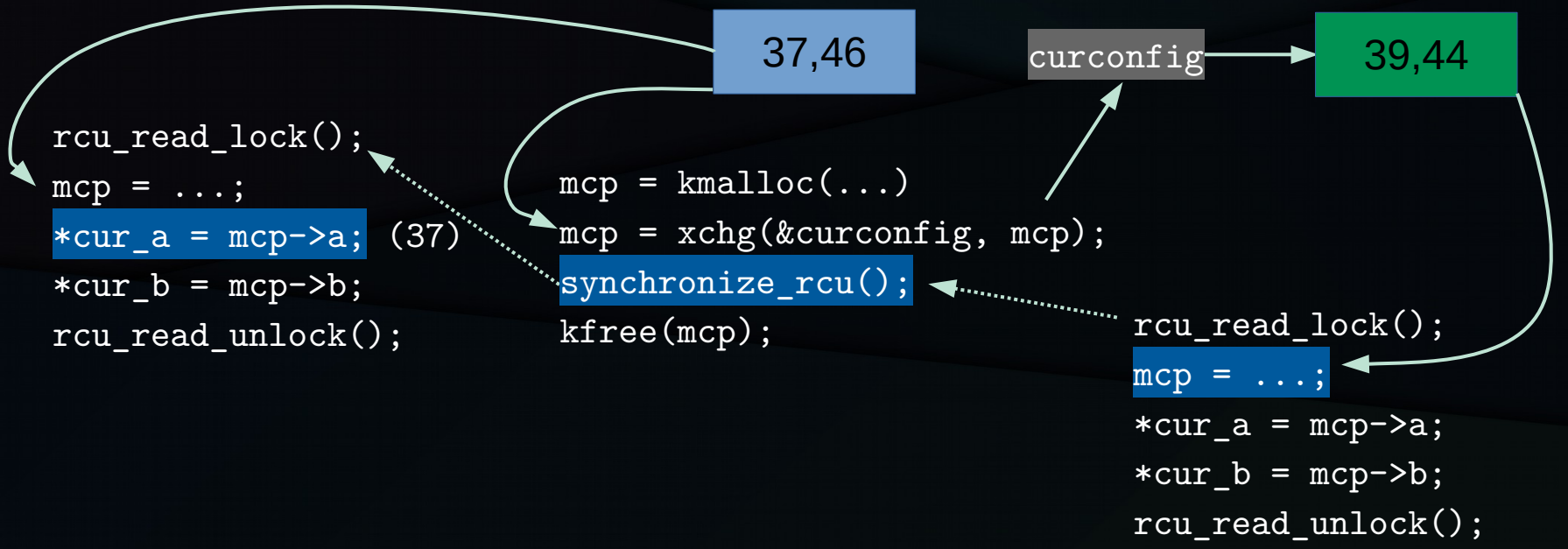
RCU Semantics (Exercise The Code)



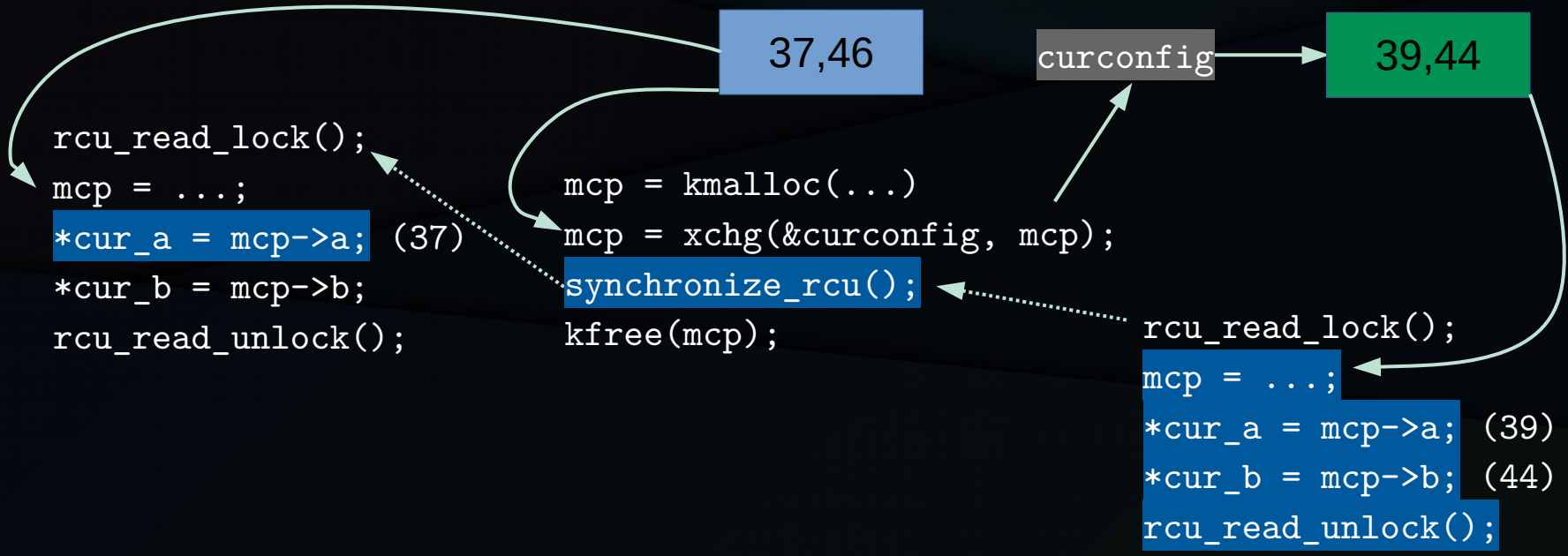
RCU Semantics (Exercise The Code)



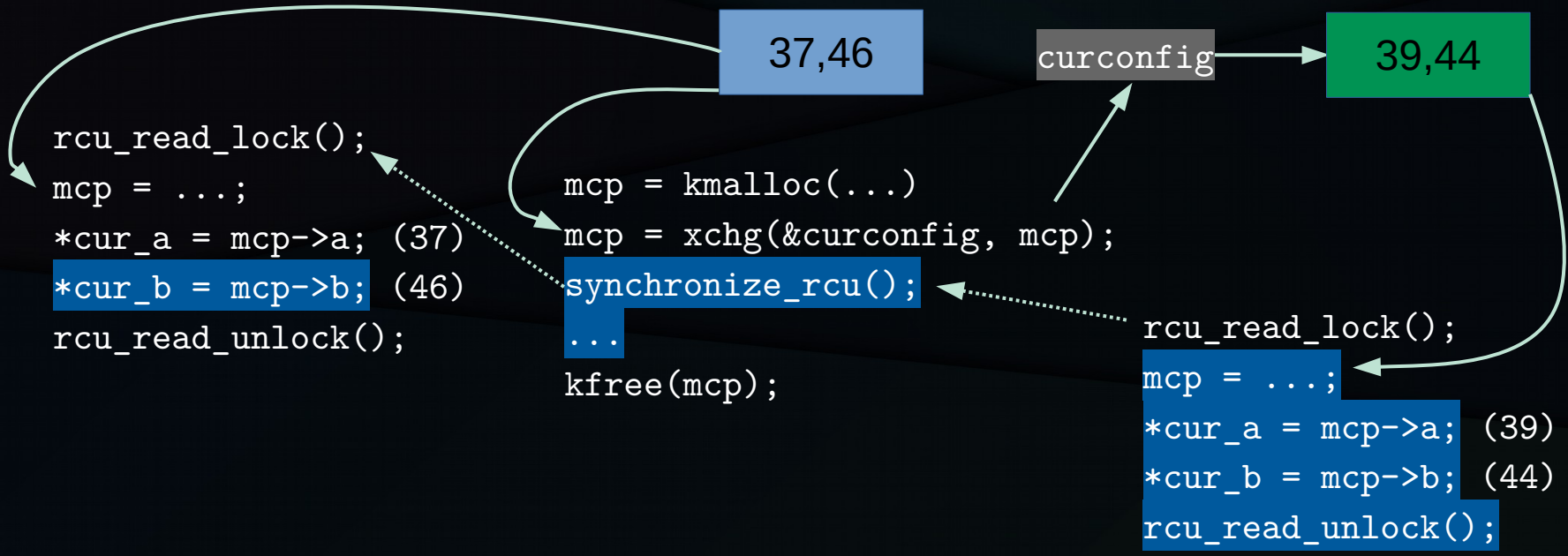
RCU Semantics (Exercise The Code)



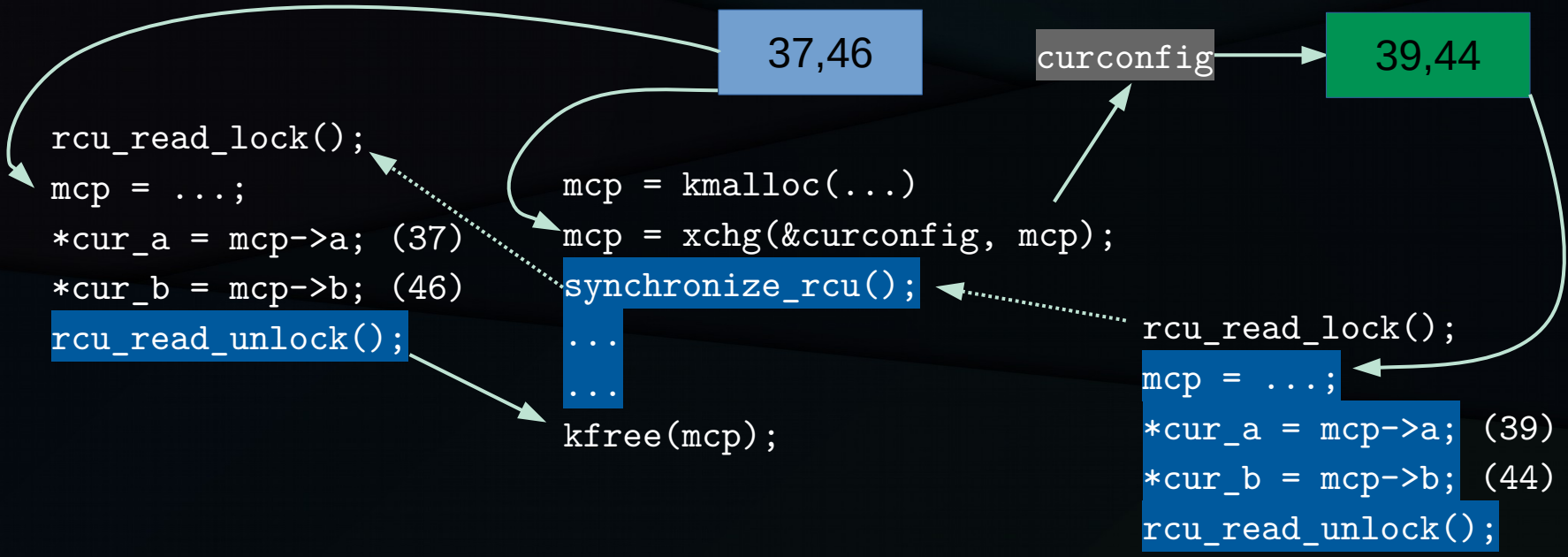
RCU Semantics (Exercise The Code)



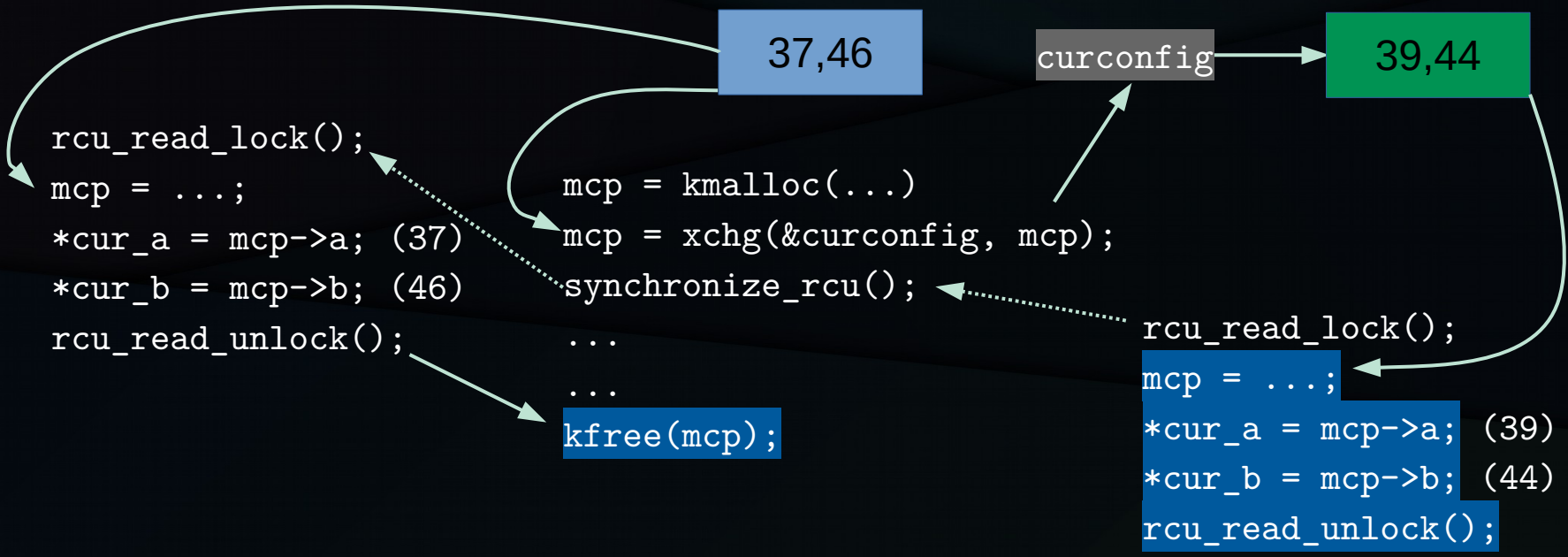
RCU Semantics (Exercise The Code)



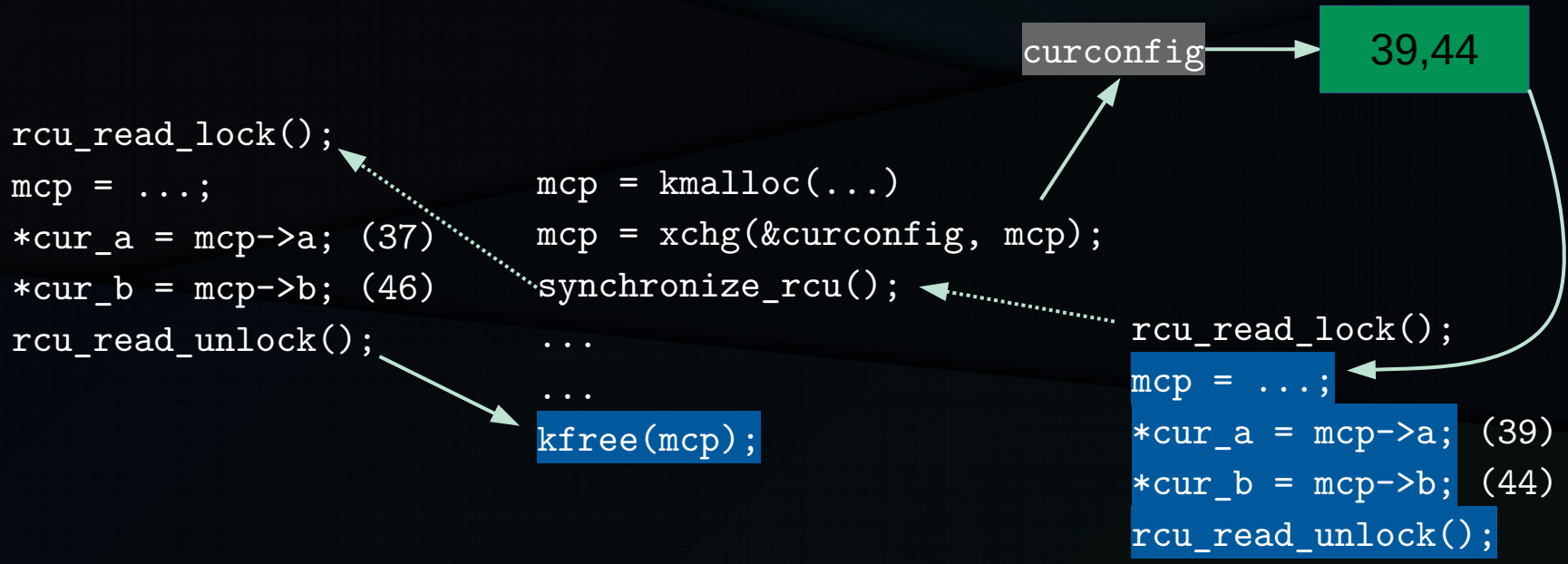
RCU Semantics (Exercise The Code)



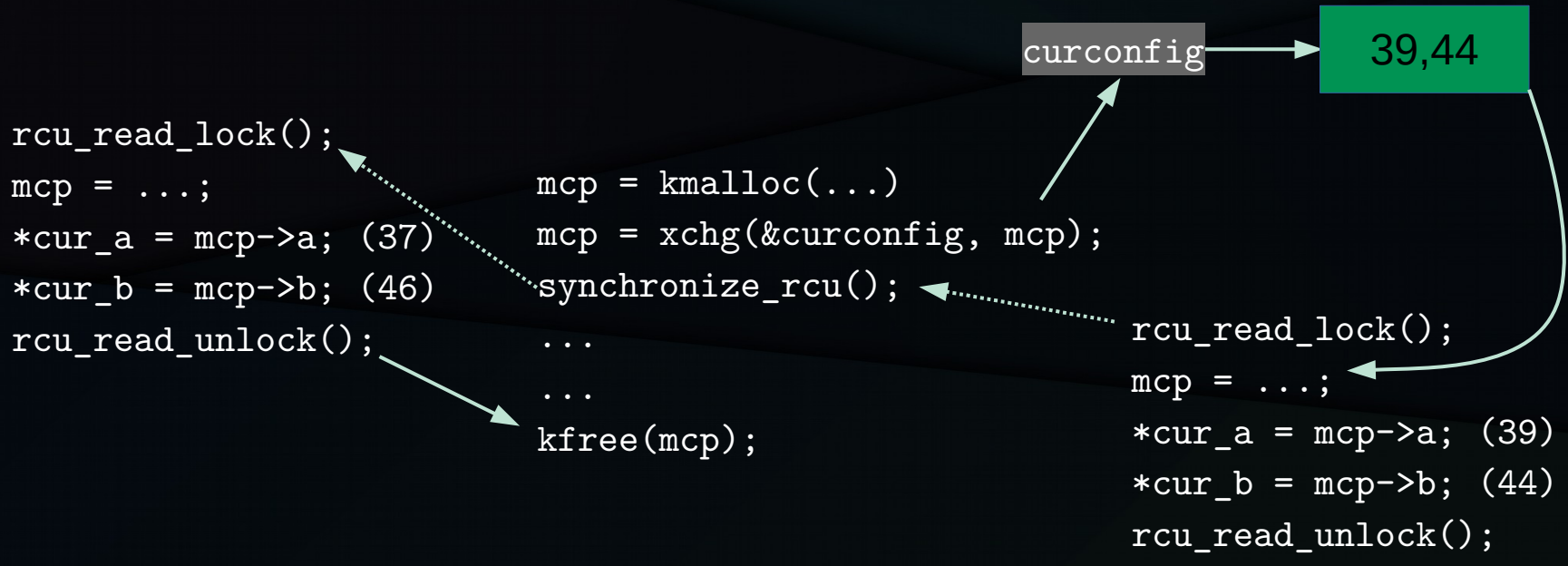
RCU Semantics (Exercise The Code)



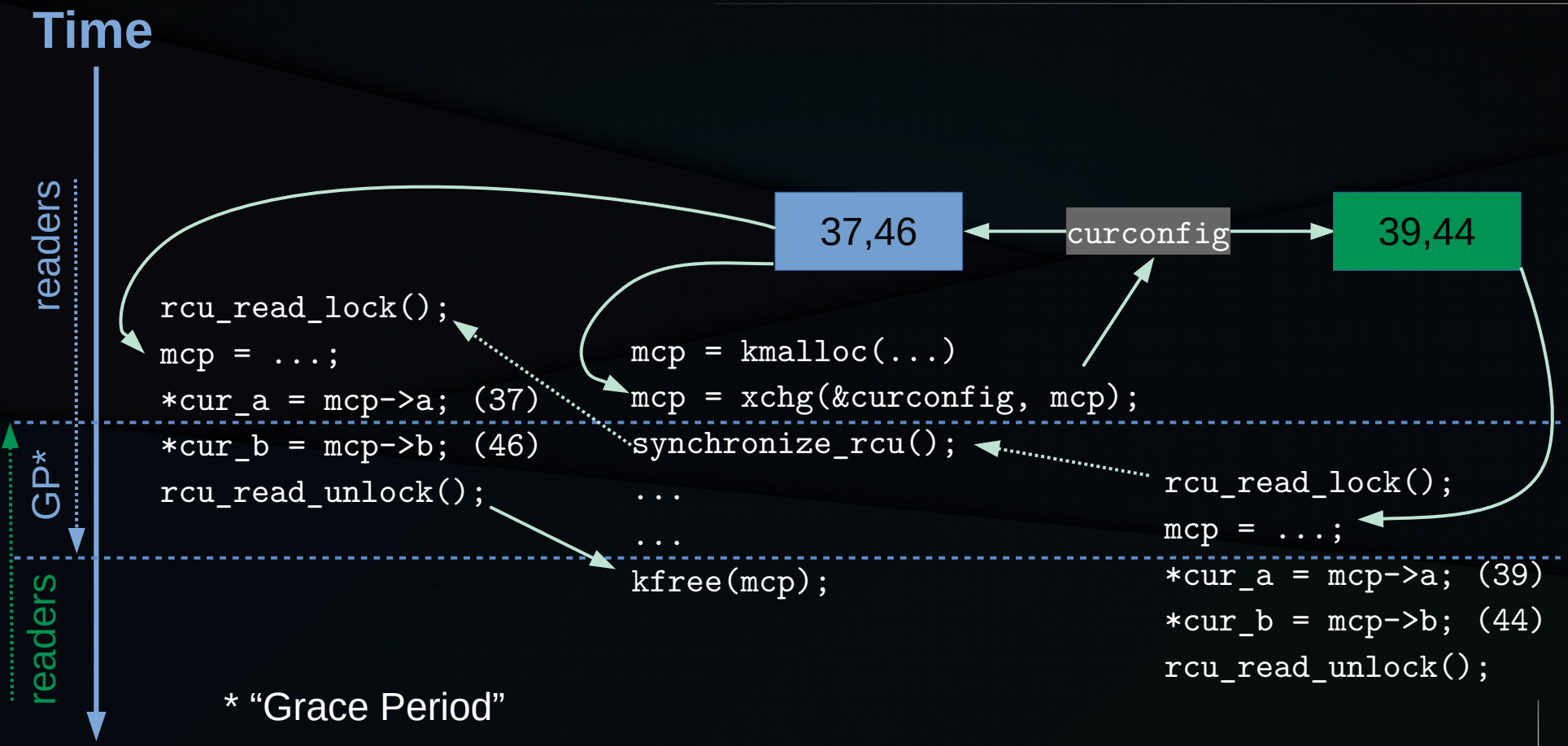
RCU Semantics (Exercise The Code)



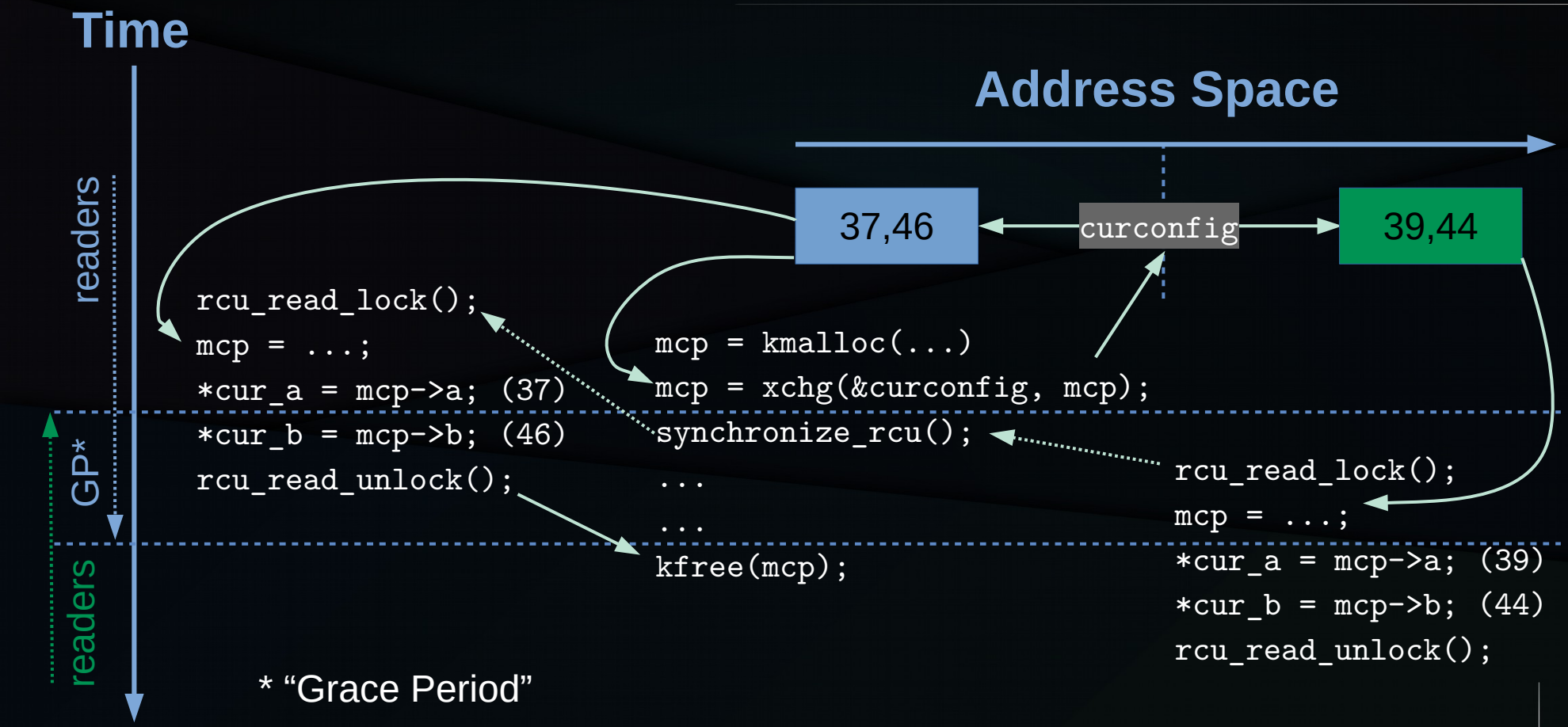
RCU Semantics (Exercise The Code)



RCU Semantics: (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

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)

Updater 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

Temporal synchronization 2

Updater 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)

Temporal synchronization 2
(End read-side critical section)

RCU: Exploiting Both Temporal and Spatial Synchronization for Decades!

Paolo Neo, Public Domain

Who Does Spatial Synchronization?!?

Who Does Spatial Synchronization?!?

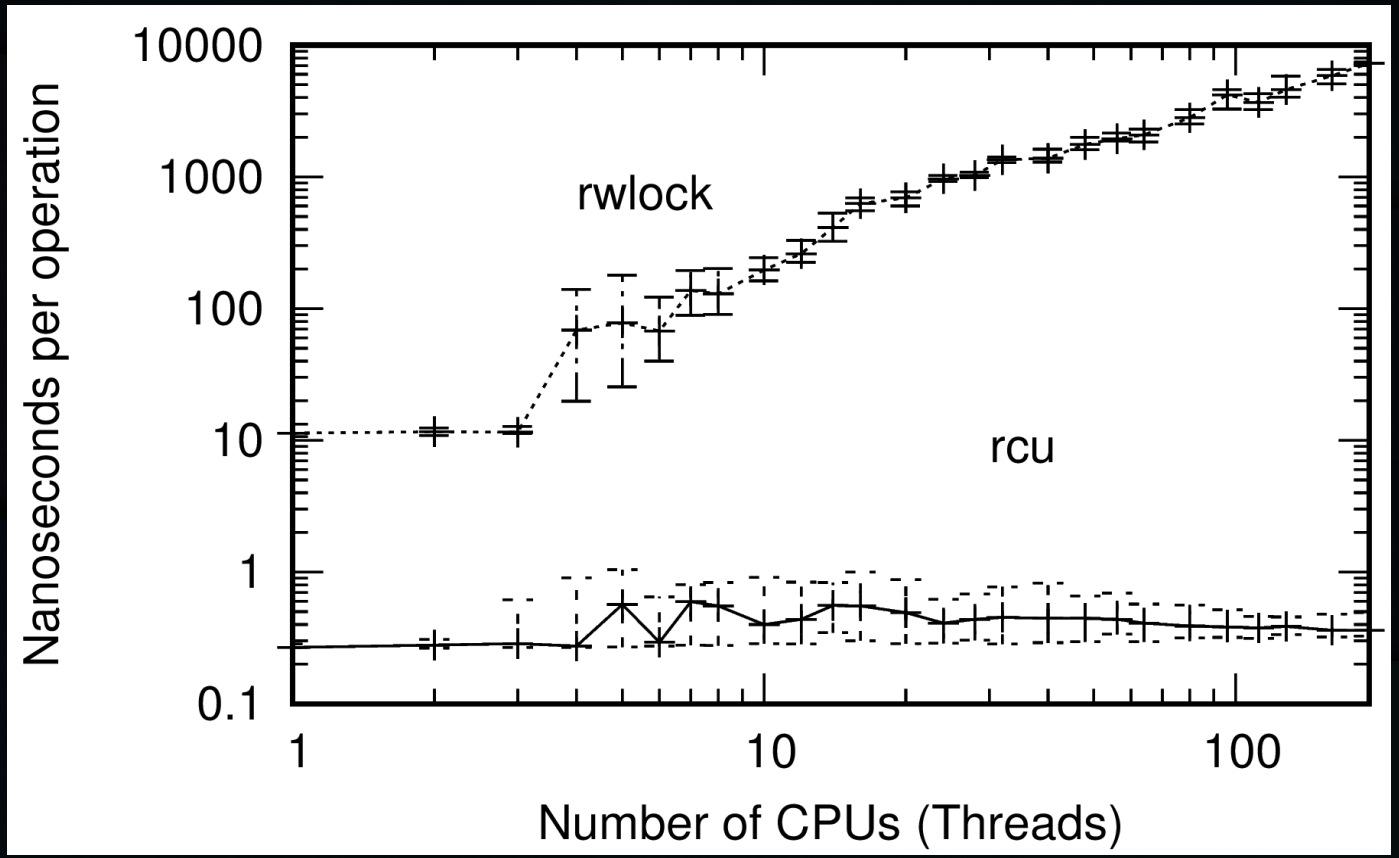
- Per-task stack locations
- Per-CPU/-thread variables
- Hash tables with per-bucket locks
  - And sharding in general (the “data locking” of old)
- Hazard pointers & other deferred reclamation

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

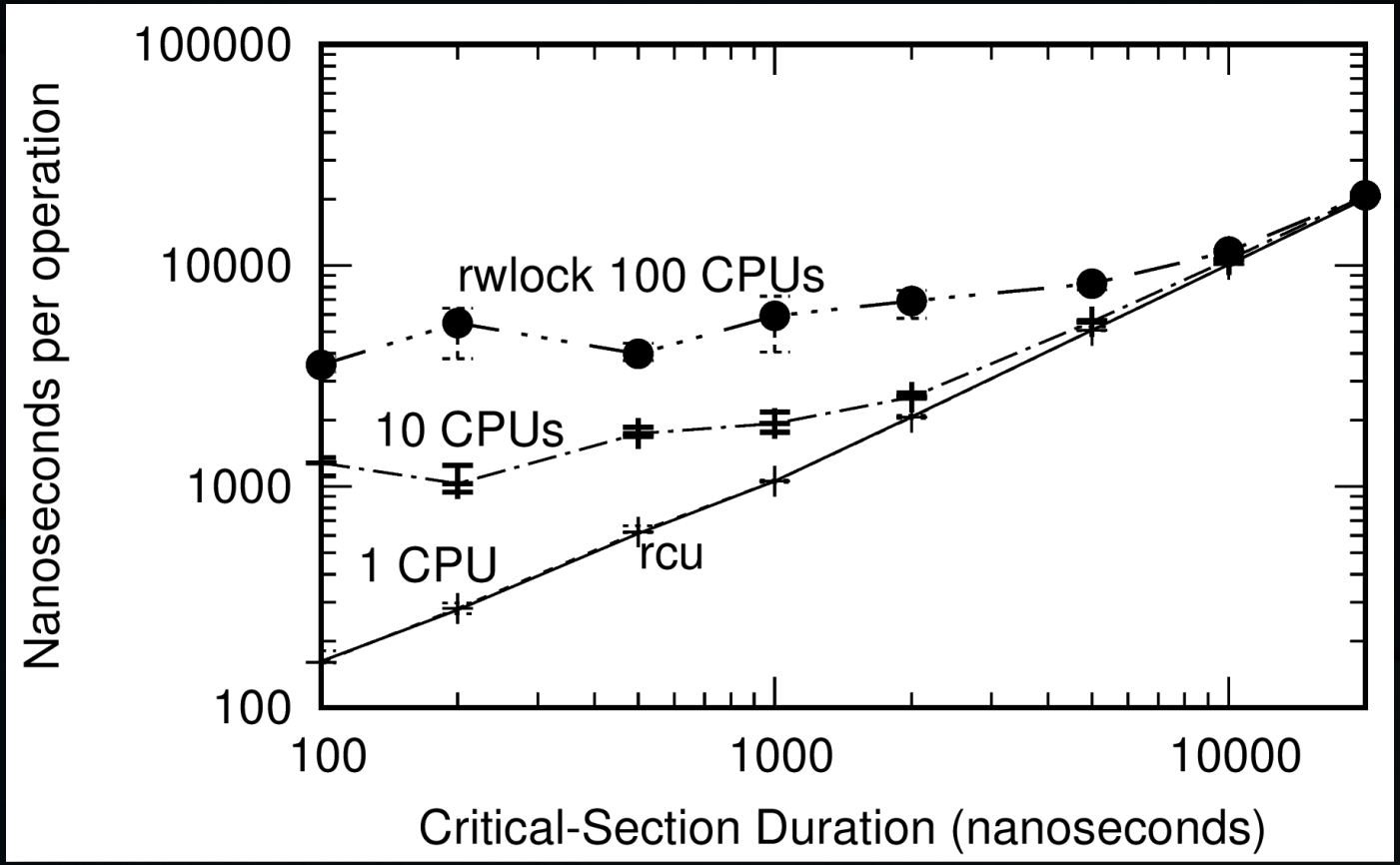
Scalability for Empty Critical Sections



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

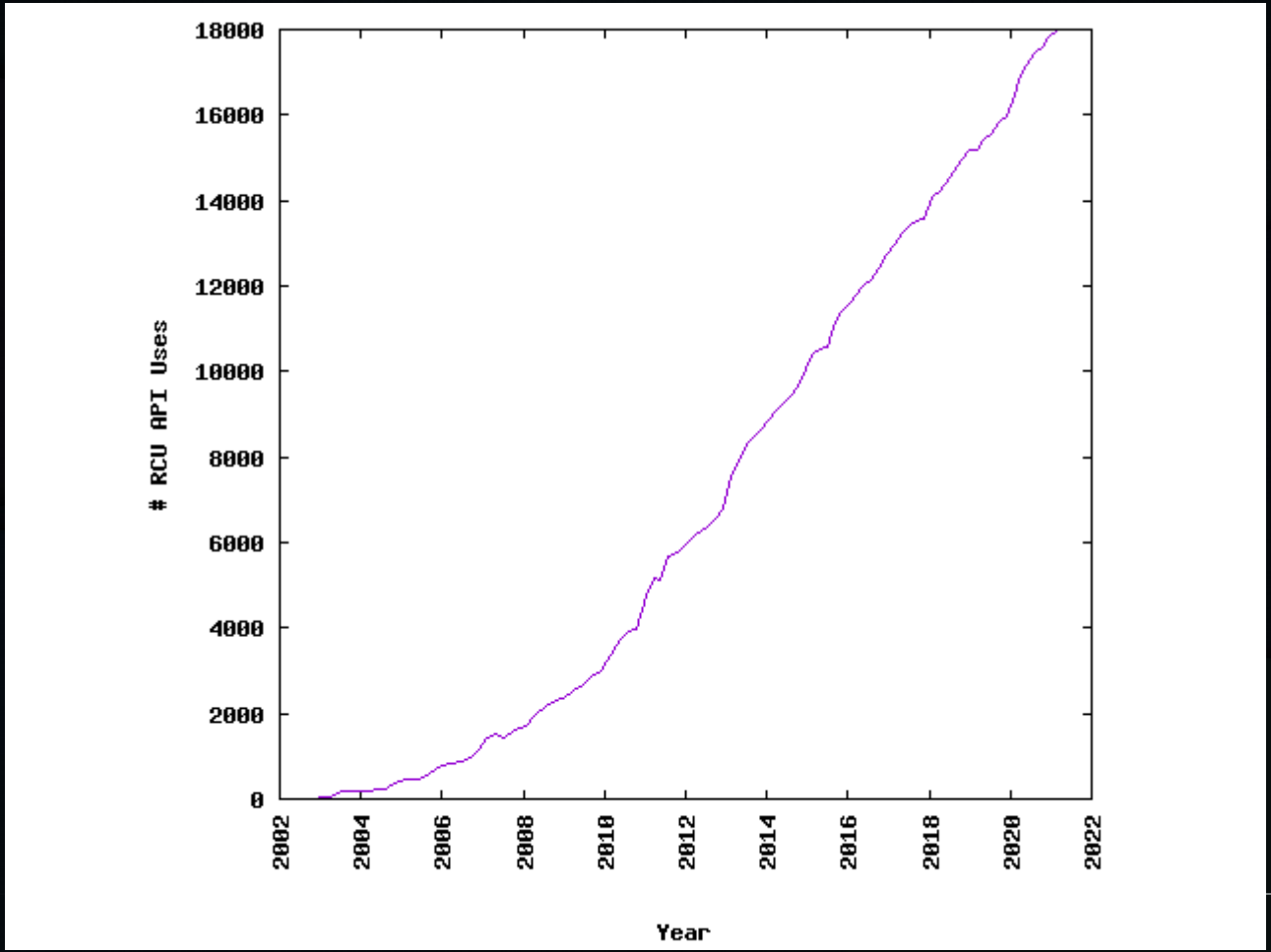
Linux Kernel Usage?

Overall RCU Usage in v5.6?

Subsystem	Calls to RCU APIs	Lines of Code	Uses/KLoc
ipc	91	9,550	9.53
net	6,959	1,116,949	6.23
security	449	99,352	4.52
kernel	1,407	361,593	3.89
virt	85	26,624	3.19
block	126	58,148	2.17
...	...	...	...
drivers	4,806	16,928,229	0.28
...	...	...	...

"RCU Usage In the Linux Kernel: Eighteen Years Later", https://dl.acm.org/doi/10.1145/3421473.3421481

RCU Usage Trend in Linux Kernel



Is There a Real Problem?

Is There a Real Problem?

- RCU Semantics Viewpoint
  - “Show me the textbook implementation!!!”
- Software-Engineering Viewpoint
- Installed-Base Viewpoint
- Software-Stack Depth Viewpoint
- Natural-Selection Viewpoint

RCU Semantics Viewpoint

- RCU has simple semantics:
  - RCU grace period must wait for all pre-existing RCU readers
- Textbook read-side primitives:

```
#define rcu_read_lock()
#define rcu_read_unlock()
#define rcu_dereference(p) \
({ \
        typeof(*p) *__p1 = READ_ONCE(p); \
        __p1; \
})
#define rcu_assign_pointer(p, v) smp_store_release((p), (v))
void synchronize_rcu(void)
{
        int cpu;

        for_each_online_cpu(cpu)
                run_on(cpu);
}
```

Here is Your Elegant Synchronization Mechanism:

Photo by "Golden Trvs Gol twister", CC by SA 3.0

Here is Your Elegant Synchronization Mechanism Equipped to Survive in The Linux Kernel:

Photo by Луц Фишер-Лампрехт, CC by SA 3.0

A Few Linux-Kernel Issues...

- Systems with 1000s of CPUs
- Sub-20-microsecond real-time response requirements
- CPUs can come and go (“CPU hotplug”)
- If you disturb idle CPUs. you enrage low-power embedded folks
- Forward progress requirements: callbacks, network DoS attacks
- RCU grace periods must provide extremely strong ordering
- RCU uses the scheduler, and the scheduler uses RCU
- Firmware sometimes lies about the number and age of CPUs
- RCU must work during early boot, even before RCU initialization
- Preemption can happen, even when interrupts are disabled (vCPUs!)
- RCU should identify errors in client code (maintainer self-defense!)

Two Definitions and a Consequence

Two Definitions and a Consequence

- A non-trivial software system contains at least one bug
- A reliable software system contains no known bugs

Two Definitions and a Consequence

- A non-trivial software system contains at least one bug
- A reliable software system contains no known bugs

- Therefore, any non-trivial reliable software system contains at least one bug that you don't know about

Two Definitions and a Consequence

- A non-trivial software system contains at least one bug
- A reliable software system contains no known bugs

- Therefore, any non-trivial reliable software system contains at least one bug that you don't know about
- I assert that Linux-kernel RCU is both non-trivial and reliable, thus contains at least one bug that I don't (yet) know about

Two Definitions and a Consequence

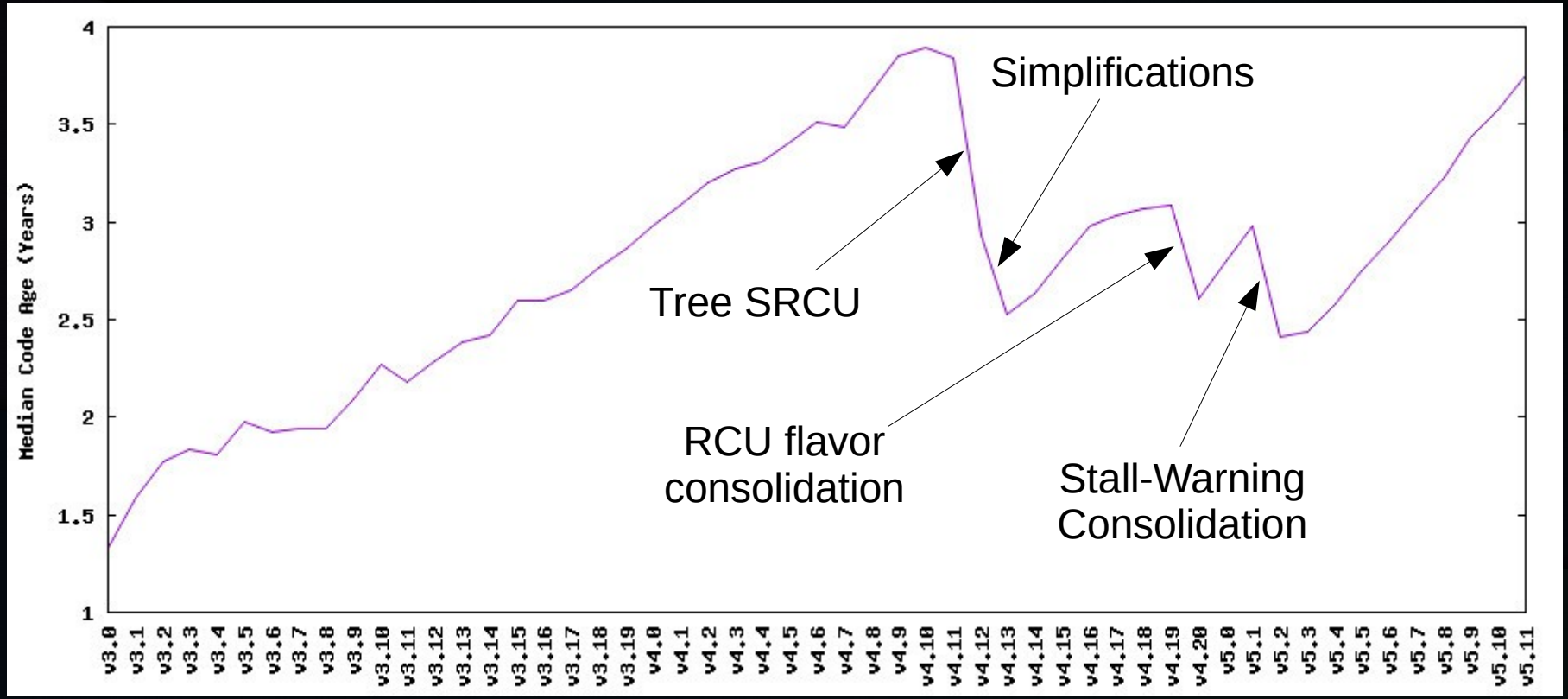
- A non-trivial software system contains at least one bug
- A reliable software system contains no known bugs

- Therefore, any non-trivial reliable software system contains at least one bug that you don't know about
- I assert that Linux-kernel RCU is both non-trivial and reliable, thus contains at least one bug that I don't (yet) know about
  - But how many bugs? Analyze from a software-engineering viewpoint...

Software-Engineering Viewpoint

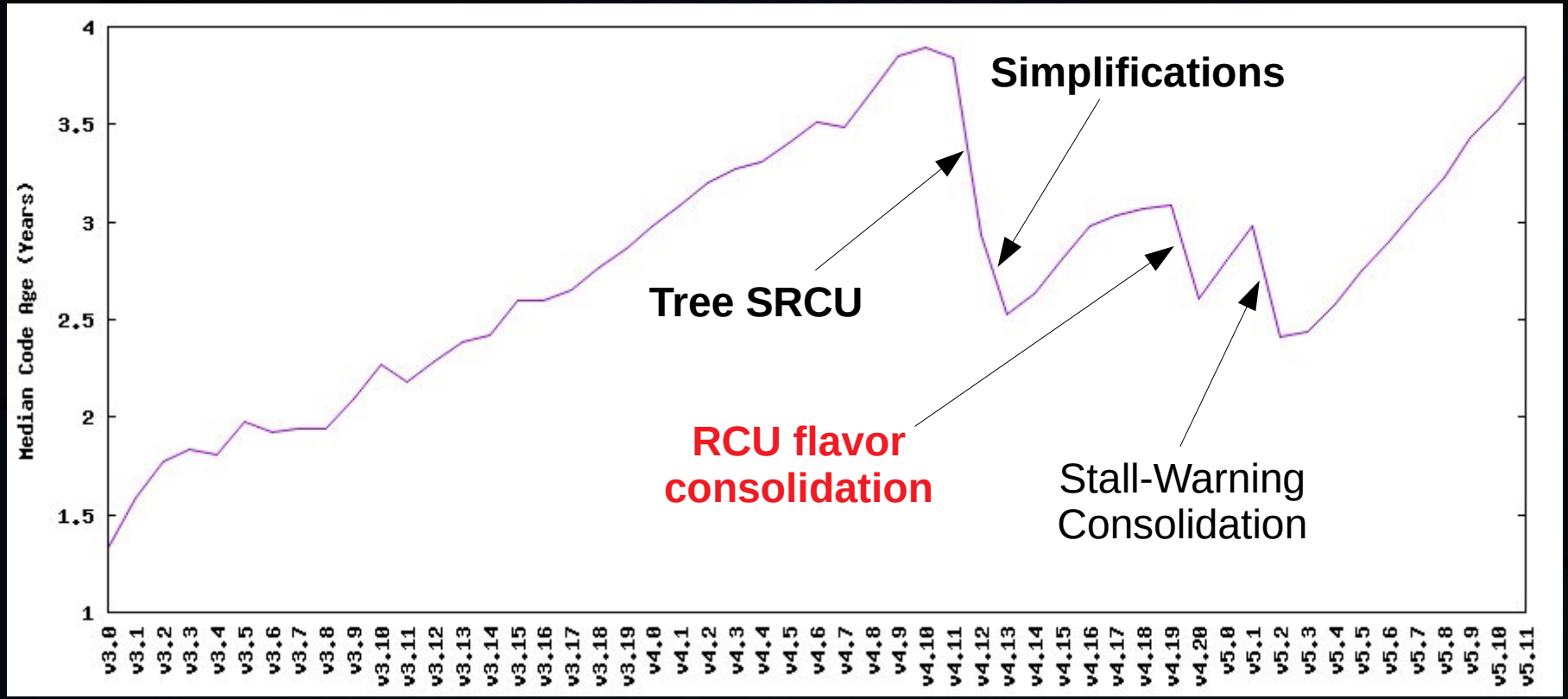
- RCU contains 17,682 LoC (including comments, etc.)
- 1-3 bugs/KLoC for production-quality code: ***18-53 bugs***
  - Best case I have seen: 0.04 bugs/KLoC for safety-critical code
    - Extreme code-style restrictions, single-threaded, formal methods, …
    - And still way more than zero bugs!!! :-)
- What is the median age of Linux-kernel RCU code?
  - Because young code tends to be buggier than old code!

Median Age of RCU Code



~/paper/RCU/intro/LinuxPrimHist

Median Age of RCU Code, Assessed



~/paper/RCU/intro/LinuxPrimHist

Linux-Kernel Maintainer Viewpoint

- Greg Kroah-Hartman: “I need to go rebase 500 commits.”
- Paul E. McKenney: “Rebasing 500 commits would kill me!”
- Greg Kroah-Hartman: “500 RCU commits would kill the kernel!”

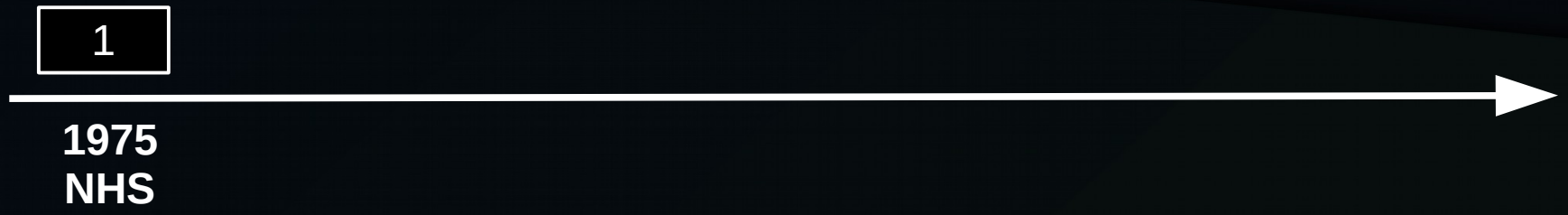
Software-Engineering Viewpoint

- RCU contains 17,682 LoC (including comments, etc.)
- 1-3 bugs/KLoC for production-quality code: ***18-53 bugs***
  - Best case I have seen: 0.04 bugs/KLoC for safety-critical code
    - Extreme code-style restrictions, single-threaded, formal methods, …
    - And still way more than zero bugs!!! :-)
- Median age of an RCU LoC is less than four years
  - And young code tends to be buggier than old code!
- We should therefore expect a few tens more bugs!!!

Installed-Base Viewpoint

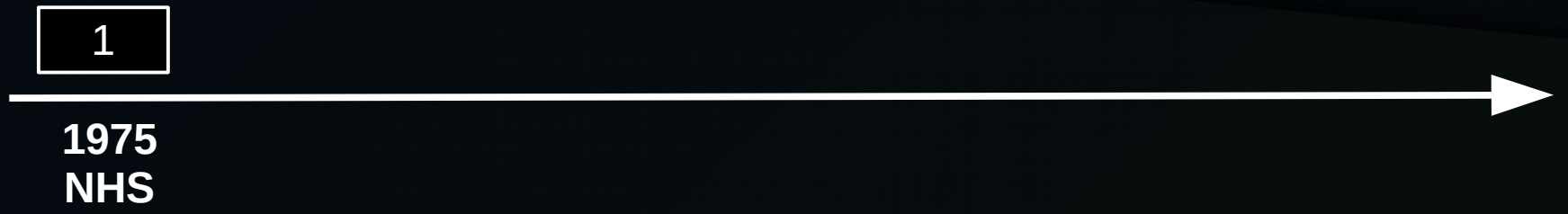
Installed-Base Viewpoint

Million-Year Bug? Once In a Million Years!!!



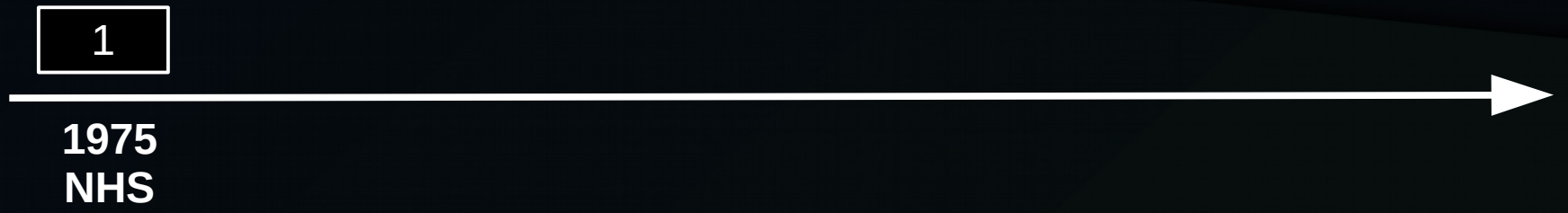
Installed-Base Viewpoint

Million-Year Bug? Once In a Million Years!!!
Murphy is a nice guy: Everything that can happen, will...



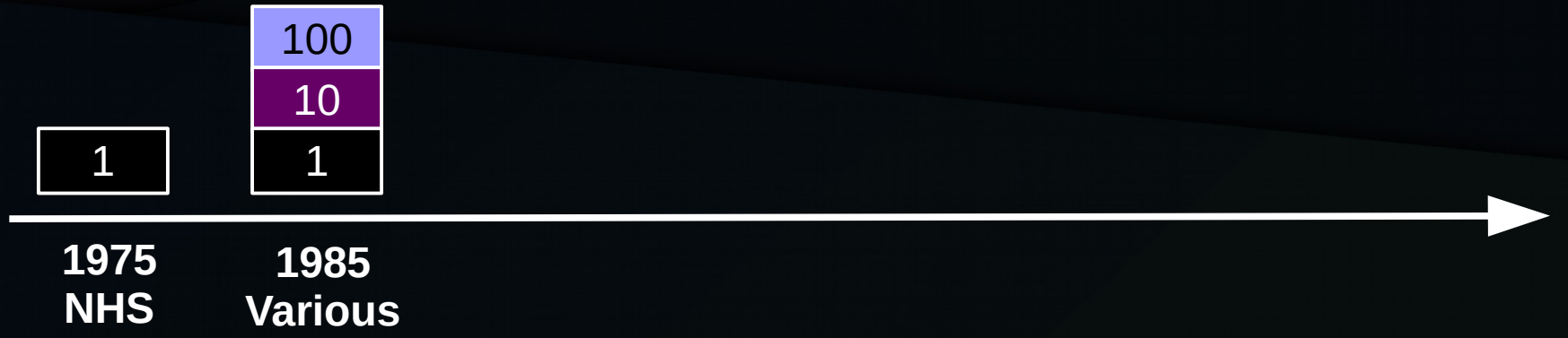
Installed-Base Viewpoint

Million-Year Bug? Once In a Million Years!!!
Murphy is a nice guy: Everything that can happen, will...
...maybe in geologic time



Installed-Base Viewpoint

Million-Year Bug? Once in Ten Millennia

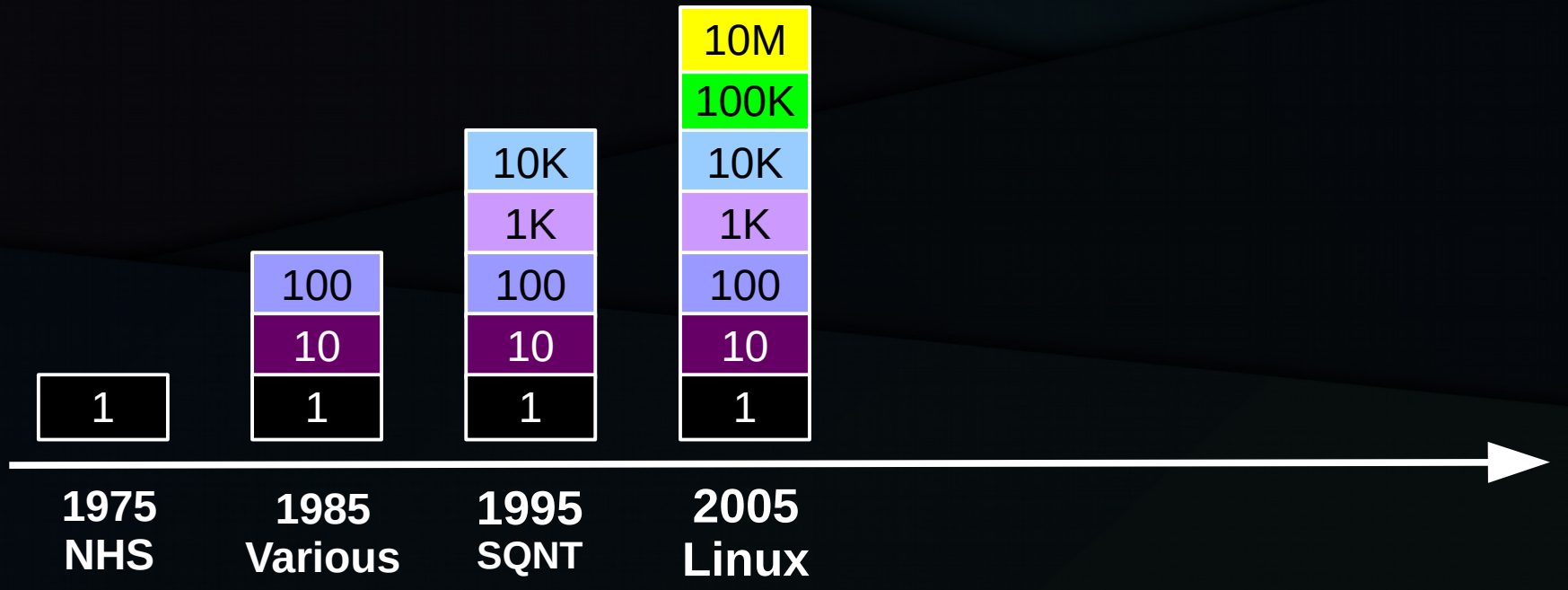


Installed-Base Viewpoint

Million-Year Bug? Once per Century

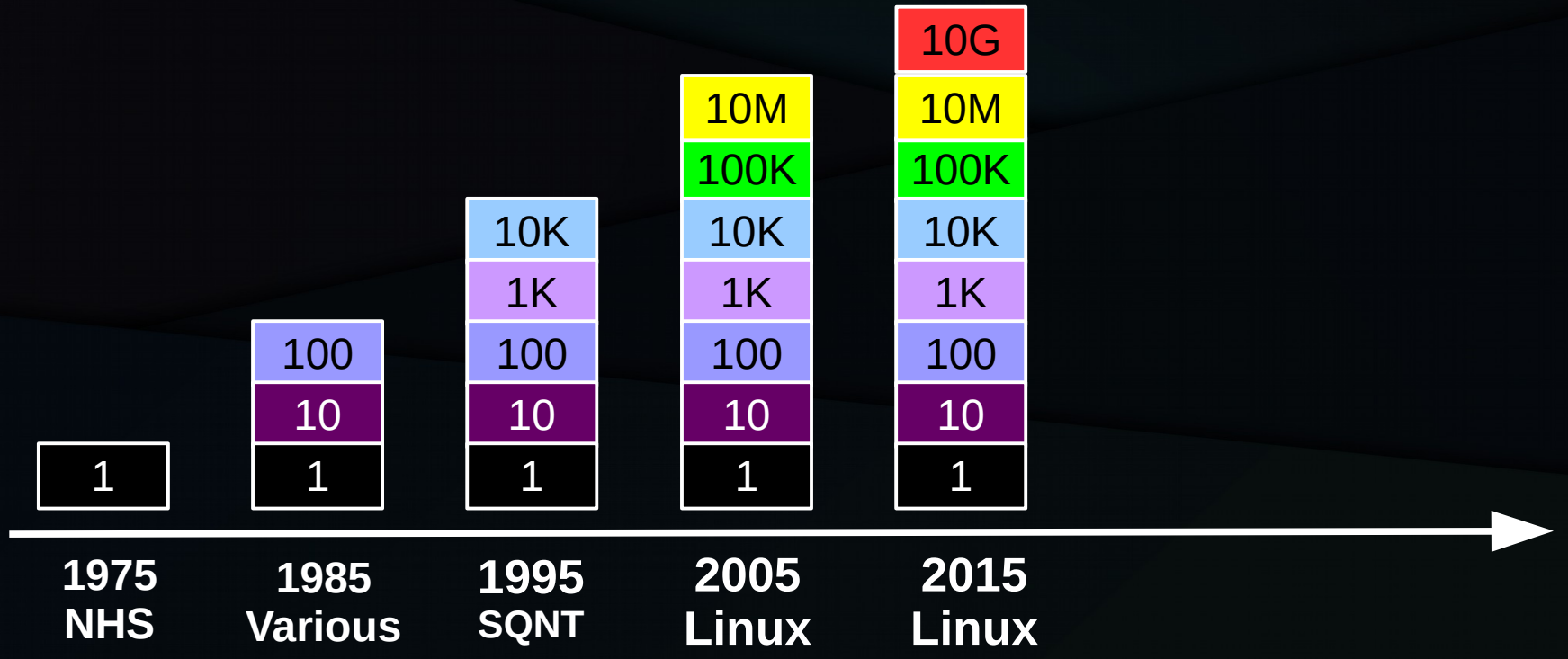
Installed-Base Viewpoint

Million-Year Bug? Once a Month



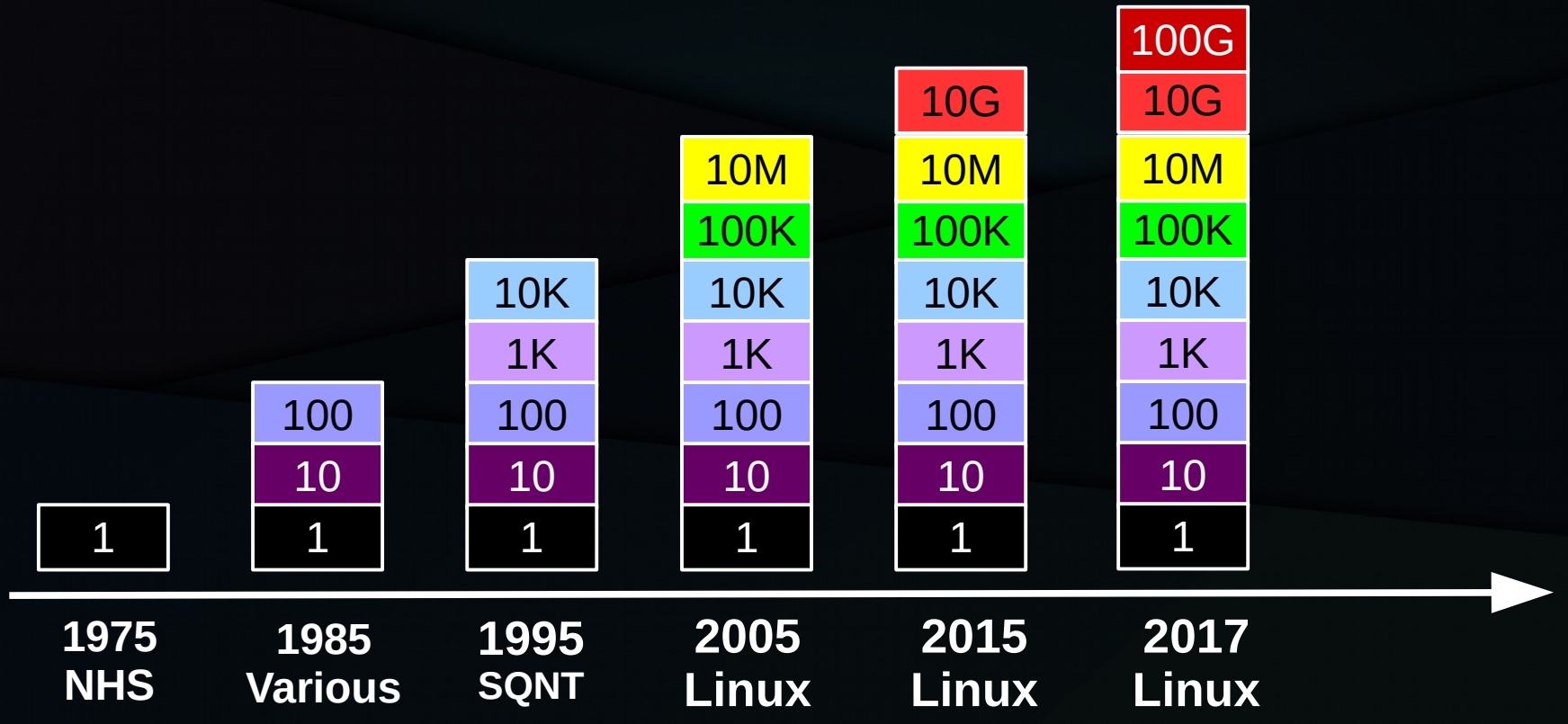
Installed-Base Viewpoint

Million-Year Bug? Several Times per *Day*



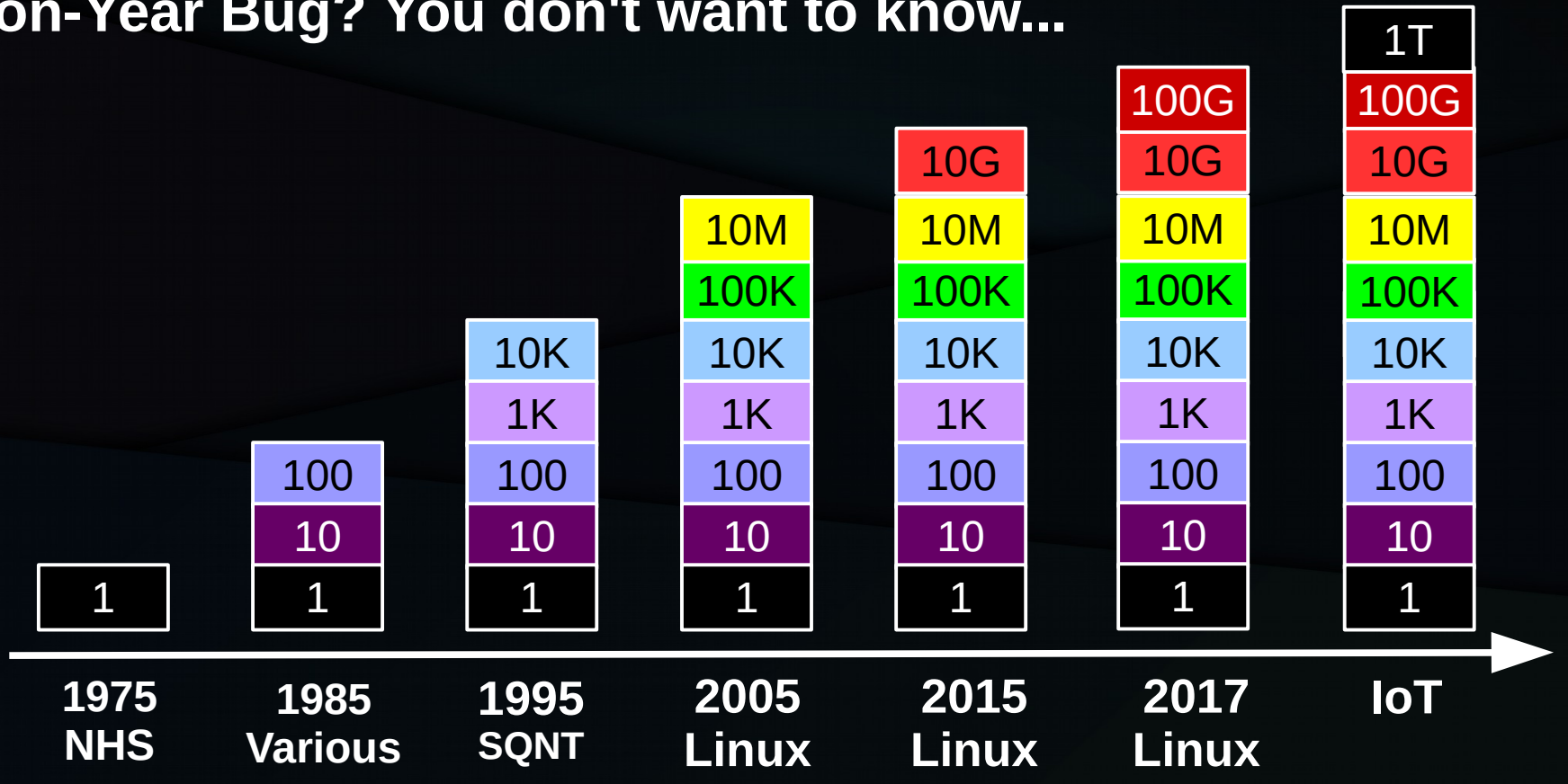
Installed-Base Viewpoint

Million-Year Bug? Several Times per *Hour*



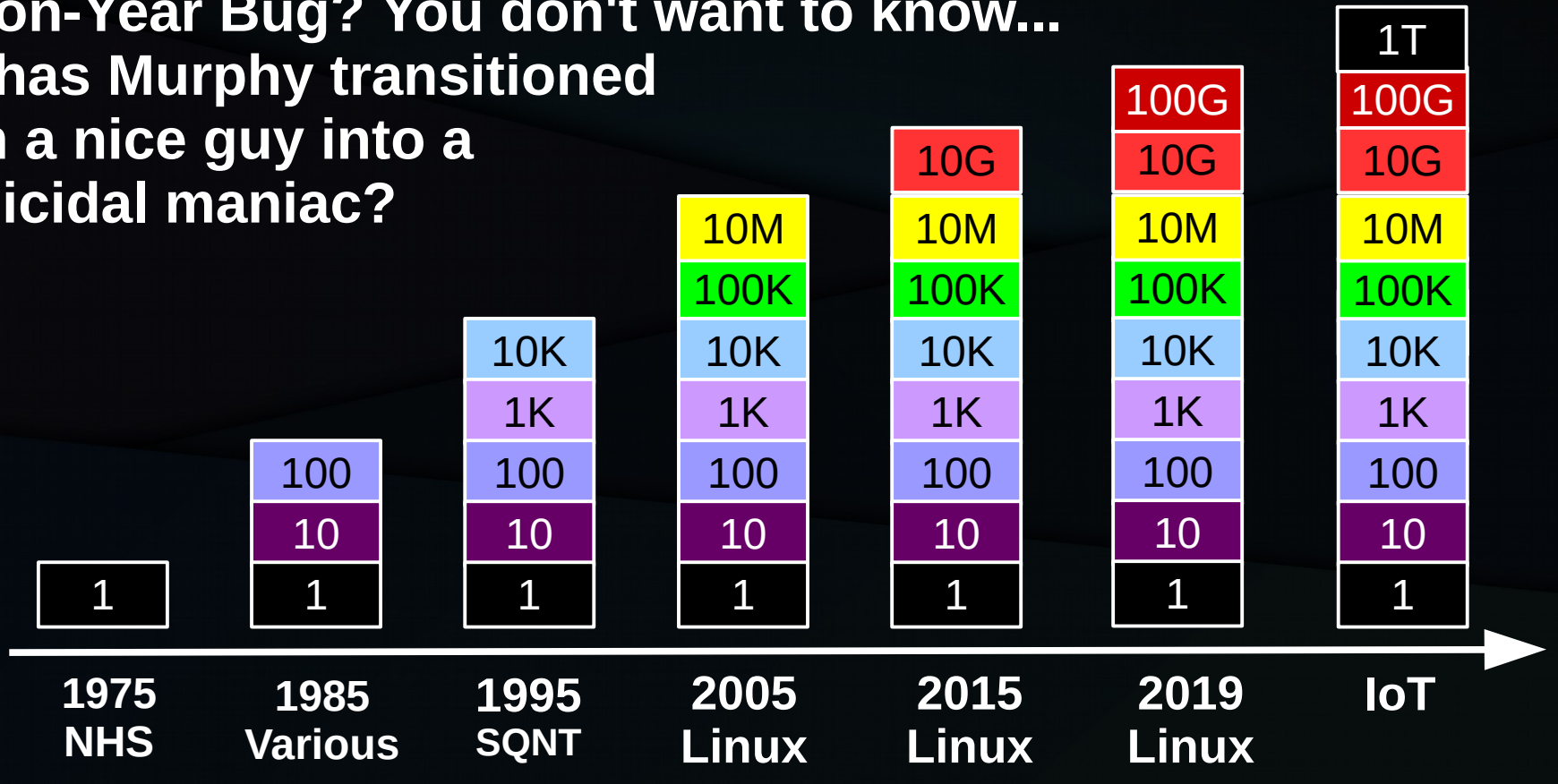
Installed-Base Viewpoint

Million-Year Bug? You don't want to know...



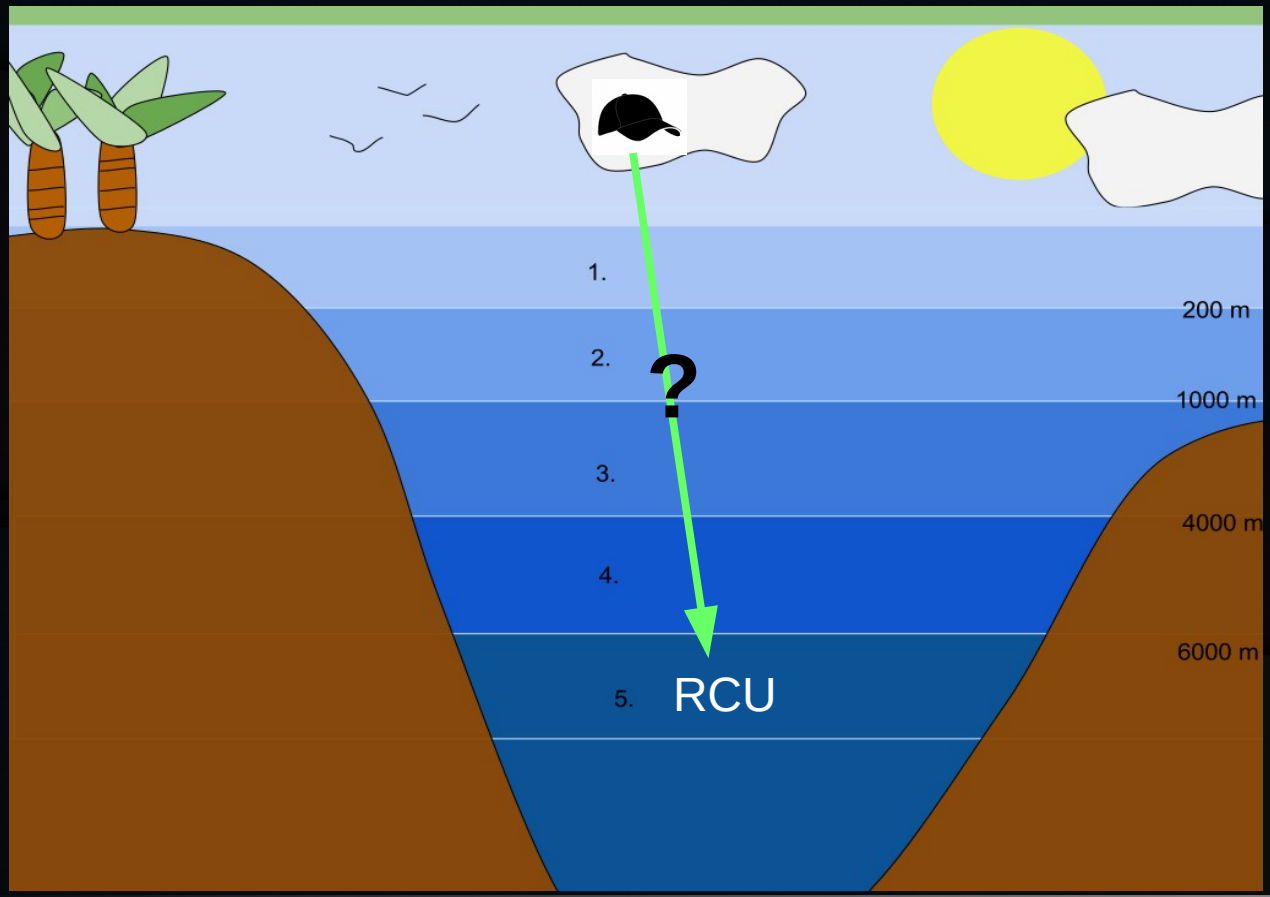
Installed-Base Viewpoint

Million-Year Bug? You don't want to know...
But has Murphy transitioned from a nice guy into a homicidal maniac?



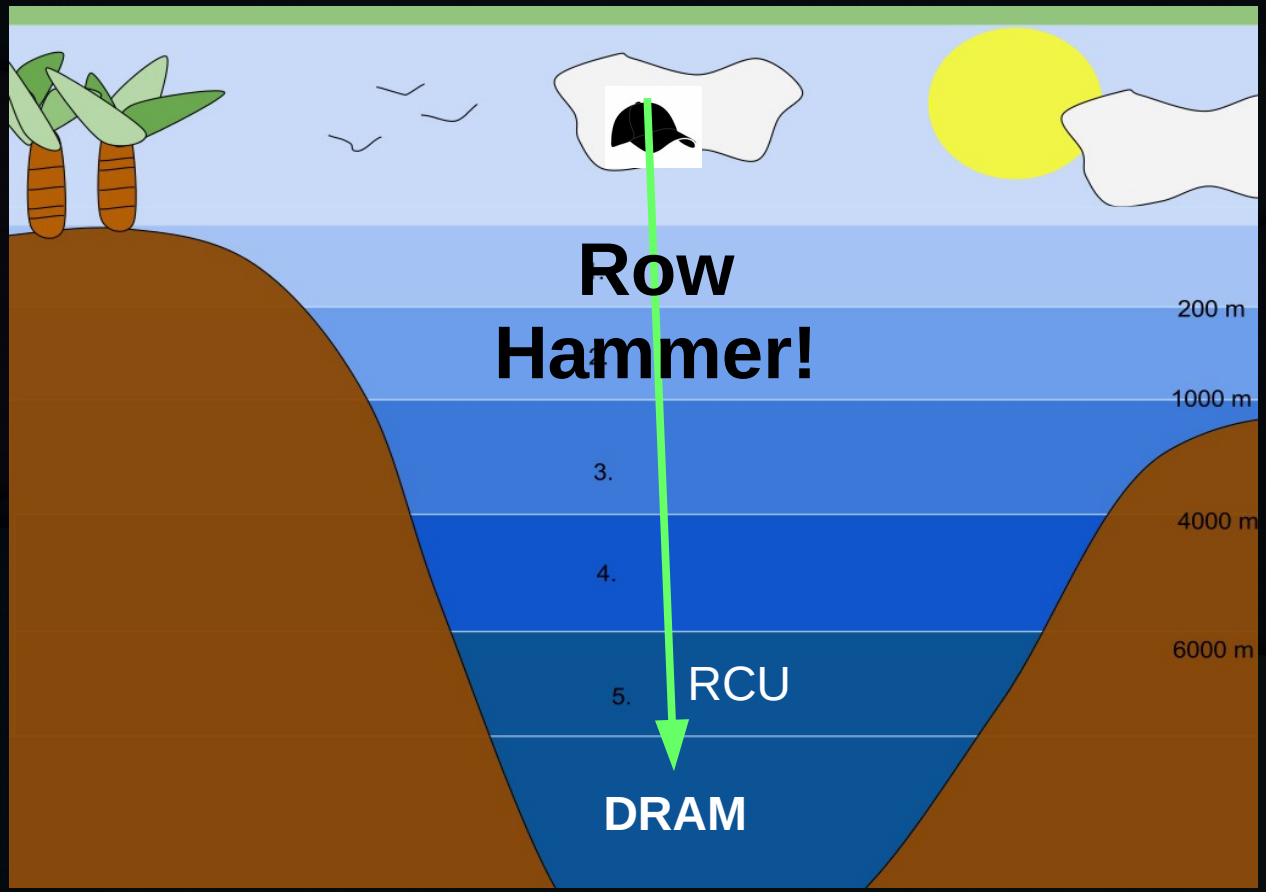
Software-Stack Depth Viewpoint

Software-Stack Depth Viewpoint???



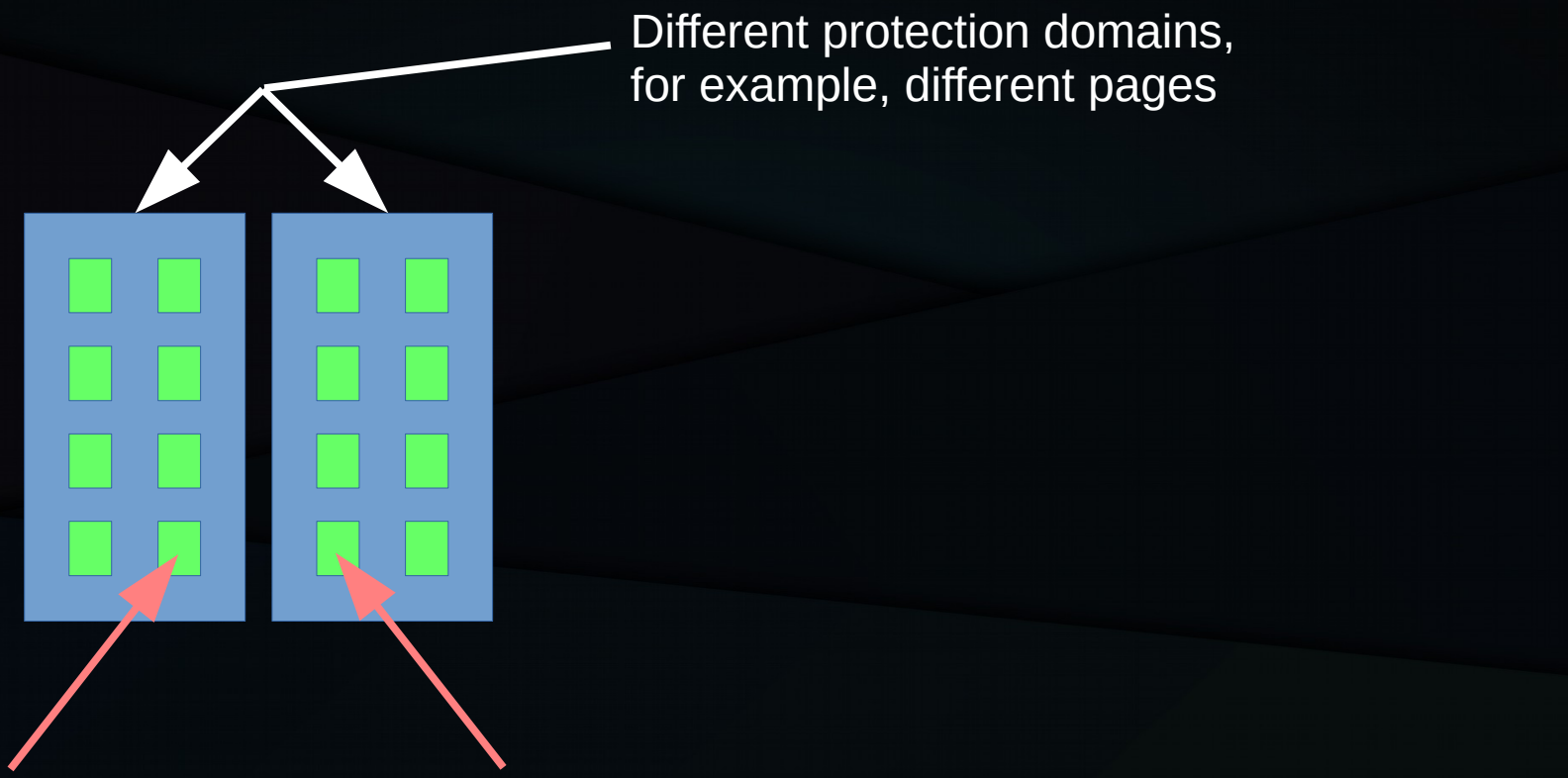
Credit: Awakening Conscience, licensed under the Creative Commons Attribution-Share Alike 4.0 International license

Software-Stack Depth Viewpoint!!!



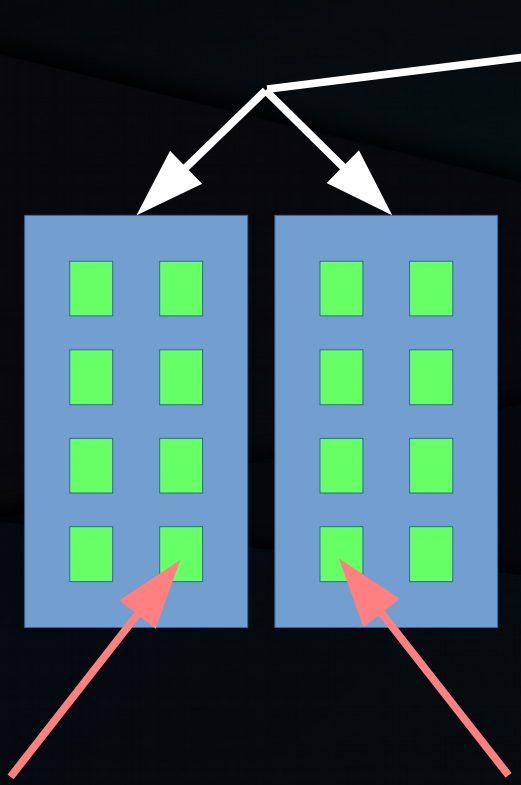
Credit: Awakening Conscience, licensed under the Creative Commons Attribution-Share Alike 4.0 International license

Obligatory Row Hammer Diagram



http://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf

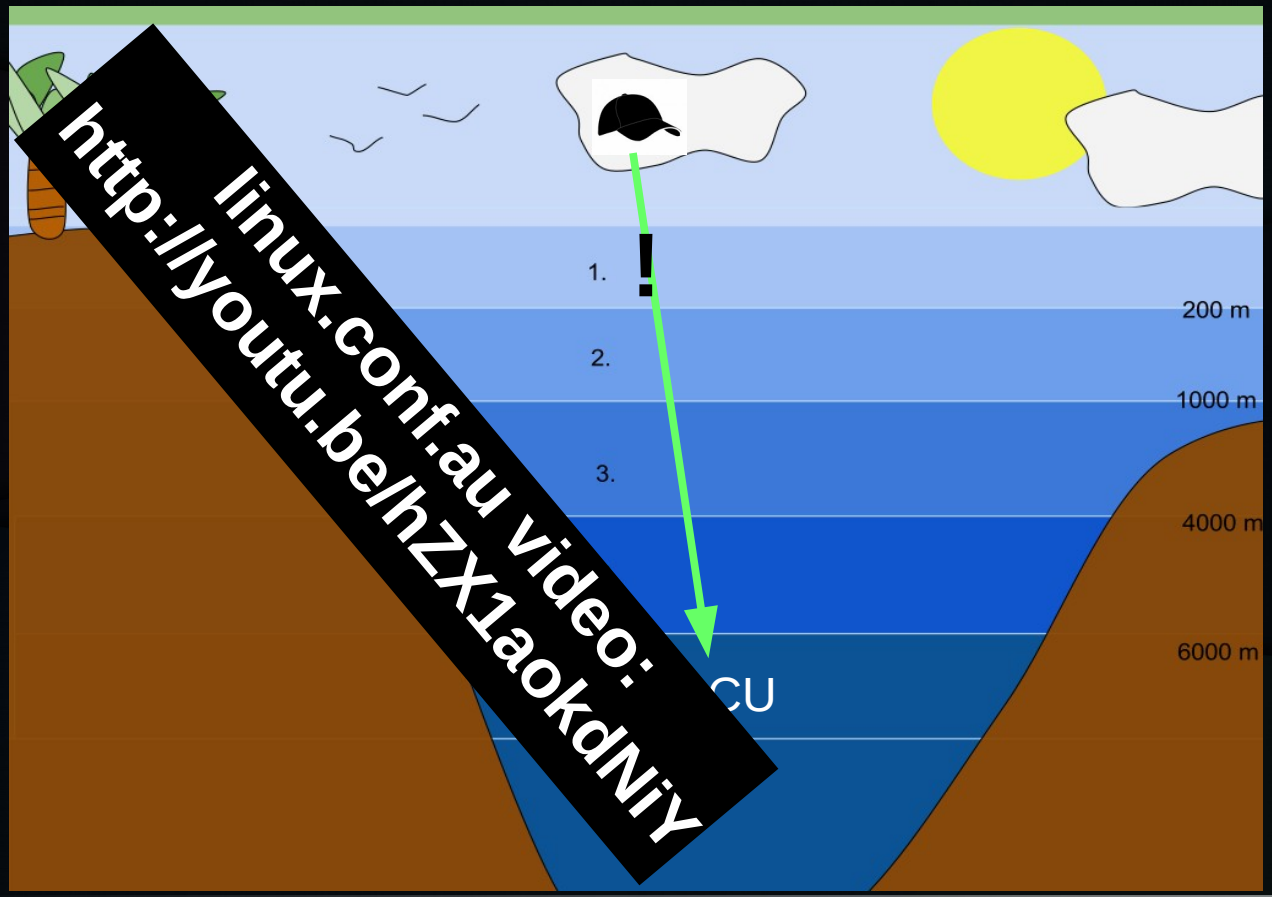
Obligatory Row Hammer Diagram



Can’t abstract our way out of this!!!

http://users.ece.cmu.edu/~yoonguk/papers/kim-isca14.pdf

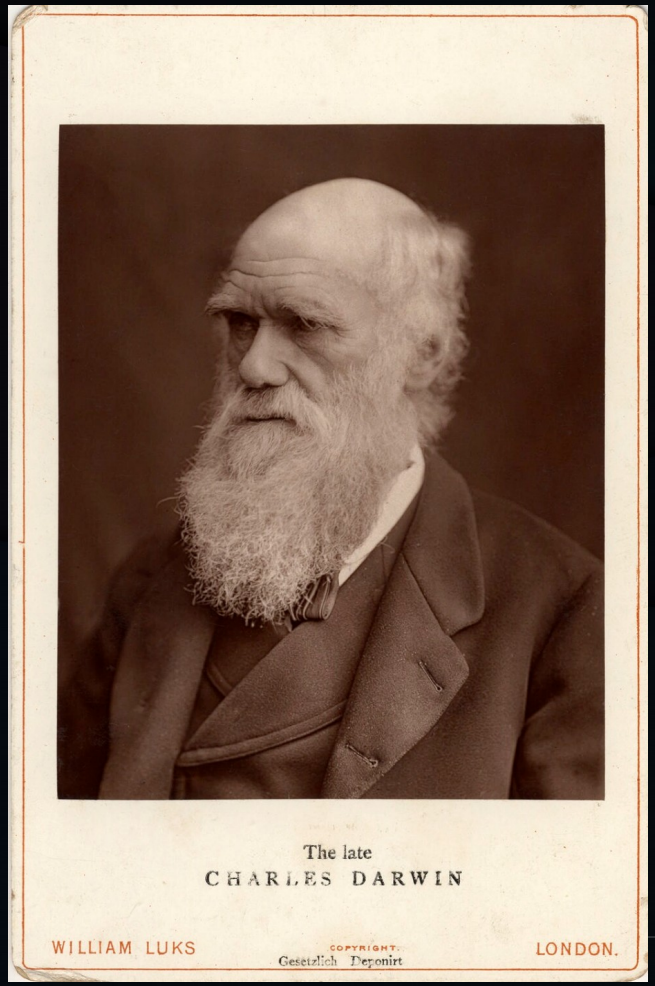
Not Just a Theoretical Possibility...



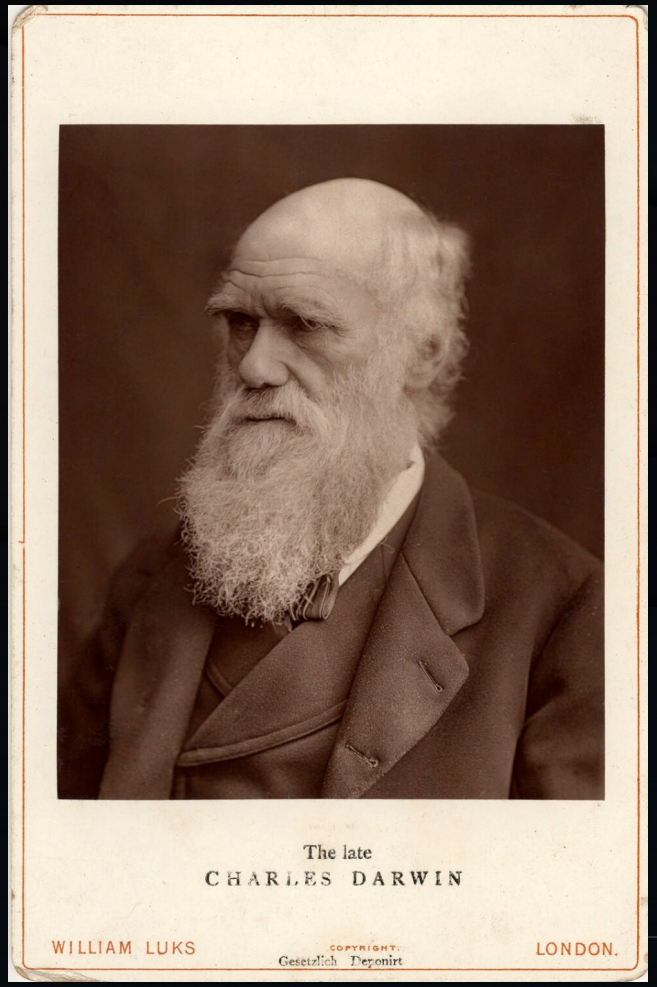
Credit: Awakening Conscience, licensed under the Creative Commons Attribution-Share Alike 4.0 International license

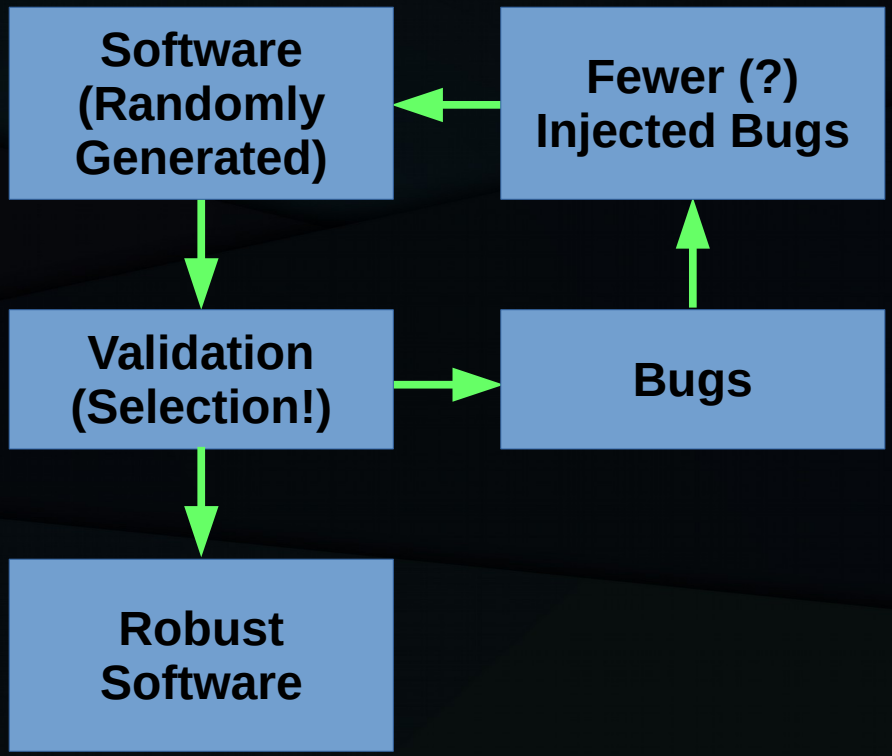
Natural Selection Viewpoint

Natural Selection Viewpoint

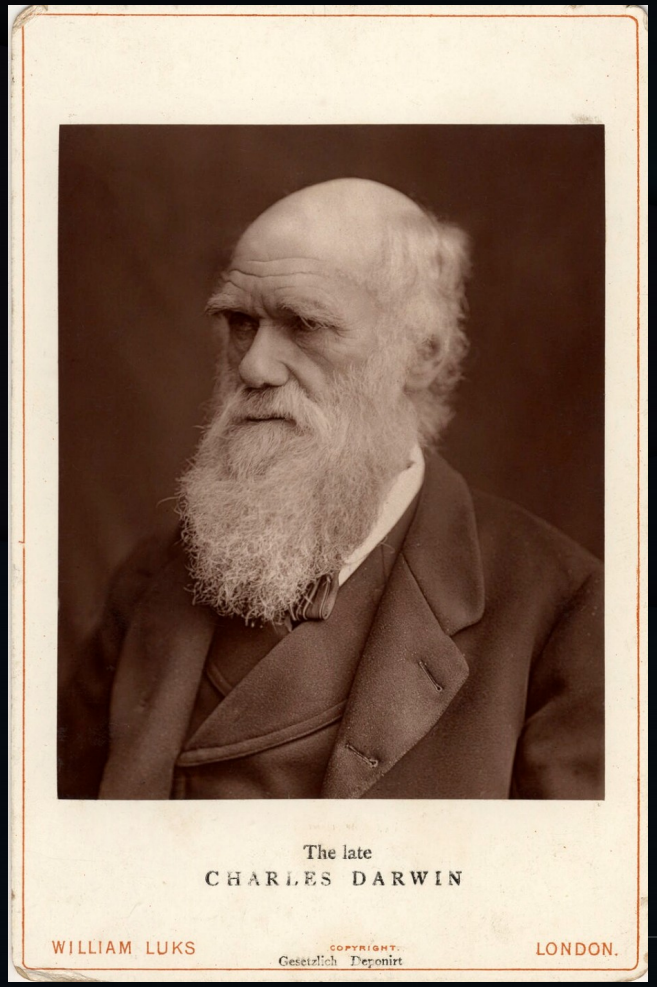


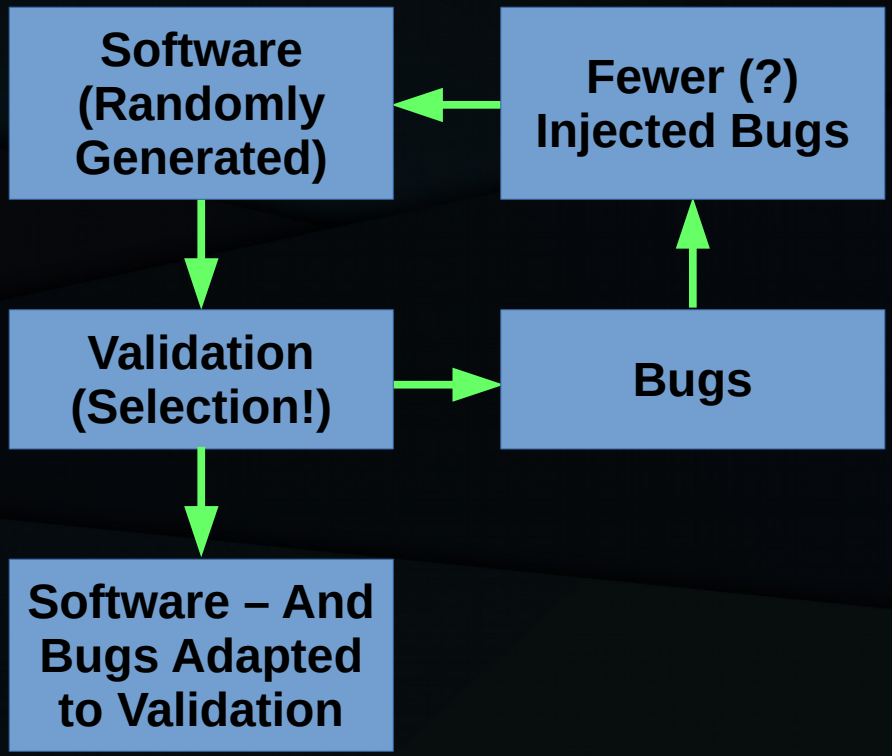
Natural Selection: Not Just Lifeforms



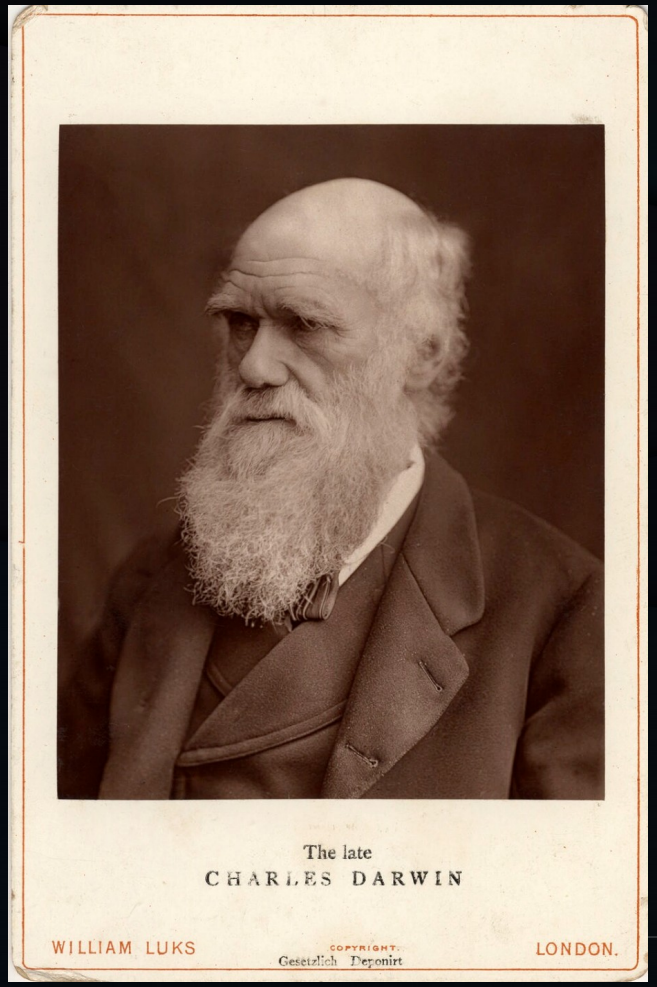


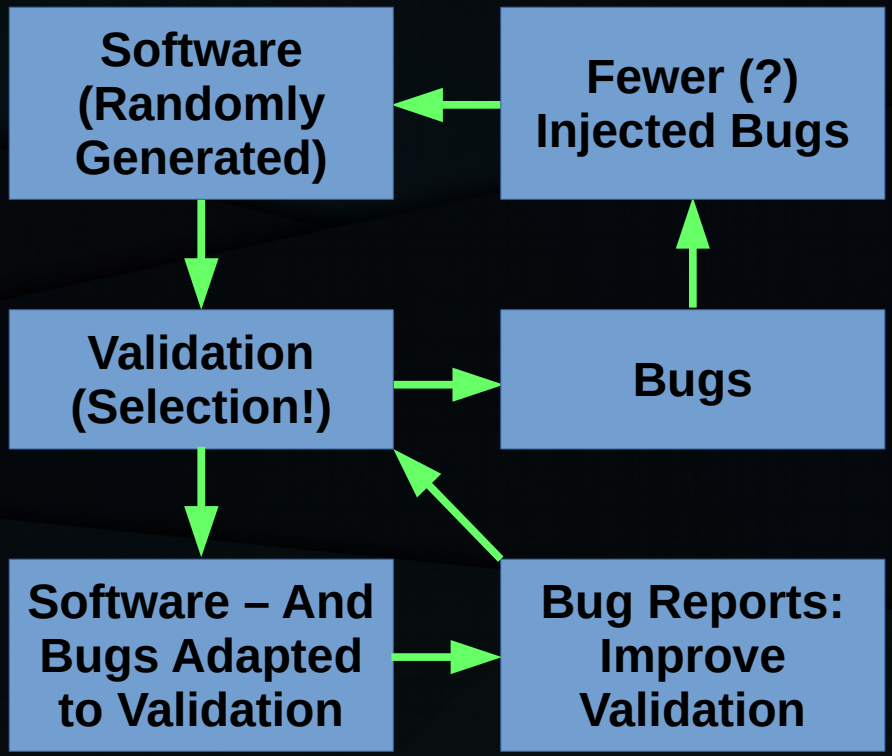
Natural Selection: Not Just Lifeforms





Natural Selection: Not Just Lifeforms





Natural Selection: Not Just Lifeforms

- There are bugs in there somewhere!!!
  - A failing test is usually a bug in the code under test
  - A passing test is a bug in the test
- The price of robust software is eternal bugs!
  - And eternal test development
- But no zero-day bugs in Linux-kernel RCU!!!

What Is Currently Being Done?

What Is Currently Being Done?

- Stress-test suite: “rcutorture”
  - http://lwn.net/Articles/154107/, http://lwn.net/Articles/622404/
- “Intelligent fuzz testing”: “trinity”, syzkaller
  - http://codemonkey.org.uk/projects/trinity/, https://github.com/google/syzkaller
- Test suites including static analysis: “0-day test robot”, “hulk robot”, ...
  - https://lwn.net/Articles/514278/
- Integration testing: “linux-next tree”
  - https://lwn.net/Articles/571980/
- Kernel Sanitizers
  - https://github.com/google/ktsan, https://github.com/google/kasan, https://github.com/google/ktsan/wiki/KCSAN
- Lock dependency checker (lockdep)
- But it does appear that more is needed...

RCU Validation Options

- Other failures mask RCU's, including HW
  - But Linux used in safety-critical systems
- More CPUs in tests (4,000 last weekend)
- Force rare critical operations more frequently
- Tests targeted to possible race conditions
- Formal verification for regression tests?

Formal Verification & Regression Tests?

Formal Verification & Regression Tests?

- “To err is human”
- The Linux kernel supports 25 CPU architectures (was 31)
- New Linux-kernel release every 2-3 months
- Finite available hardware and personnel
- Bugs are known to exist: What else can you tell us?
- Adding scaffolding/specifications adds bugs: Breakeven?
- Fixing bugs with some probability adds bugs: Breakeven?

Formal Verification & Regression Tests?

- “To err is human”
- The Linux kernel supports 25 CPU architectures (was 31)
- New Linux-kernel release every 2-3 months
- Finite available hardware and personnel
- Bugs are known to exist: What else can you tell us?
- Adding scaffolding/specifications adds bugs: Breakeven?
- Fixing bugs with some probability adds bugs: Breakeven?
- ***Other validation techniques: Investment tradeoffs?***

Formal Verification & Regression Tests

- Automatic translation or no translation
  - Automatically discard irrelevant code
  - Manual translation: Opportunity for human error!
- Correctly mode environment
  - Including memory model
  - QRCU benchmark: An excellent cautionary tale
- Reasonable memory & CPU overhead
  - Bugs located in practice as well as in theory
  - Linux-kernel RCU is 17KLoC (plus 8KLoC tests) and release cycles are short
- Map to source line(s) containing the bug
  - “Something is wrong somewhere” is not helpful
  - One bug reported thus far this week!!!
- Modest input outside of code under test
  - Glean the specification from the source code itself (empirical/incomplete spec!)
  - Specifications are large bodies of software and can therefore have their own bugs
- Find relevant bugs
  - Low false-positive rate, weight towards likelihood of occurrence (fixes create bugs!)

Many of these are at best acquired tastes for FV researchers

A Few Formal Verification Tools

- Promela/spin, TLA
- PPCMEM, ARMMEM, RMEM (new!)
- Herd
- Linux-kernel memory model (LKMM)
- CBMC
- Nidhugg

Promela/spin & TLA: Design-Time Verification

- 1993: Shared-disk/network election algorithm (pre-Linux)
  - Single-point-of failure bug in specification: Fixed during coding
  - But fix had bug that propagated to field: Cluster partition
  - **Conclusion**: Formal verification is trickier than expected!!!
- 2007: “Quick” RCU (QRCU) – fast updaters
  - http://lwn.net/Articles/243851/, but never accepted into Linux kernel
- 2008: RCU idle-detection energy-efficiency logic
  - http://lwn.net/Articles/279077/
  - Verified, but much simpler approach found two years later
  - **Hypothesis**: Need for formal verification: Symptom of too-complex design?

Promela/spin & TLA: Design-Time Verification

- 2012: Verify userspace RCU, emulating weak memory
  - Two independent models (Desnoyers and myself), bug injection
- 2014: NMIs can nest!!! Affects energy-efficiency logic
  - Verified, and no simpler approach apparent thus far!!!
  - Note: Excellent example of empirical specification (AKA “incomplete specification”)
- 2018: TLA & queued spinlock (Catalin Marinas and Will Deacon)
  - Liveness: https://linuxplumbersconf.org/event/2/contributions/60/

https://lamport.azurewebsites.net/tla/tla.html

PPCMEM, ARMMEM, RMEM, and Herd

- Verified suspected bug in Power Linux atomic primitives
- Found bug in Power Linux spin_unlock_wait()
- Verified ordering properties of locking primitives
- Excellent memory-ordering teaching tools
  - Starting to be used more widely within IBM as a design-time tool
  - And within the wider Linux-kernel community
- PPCMEM: (http://lwn.net/Articles/470681/)
  - Accurate but slow
- Herd: (http://lwn.net/Articles/608550/)
  - Faster, but still not able to handle 10,000-line programs

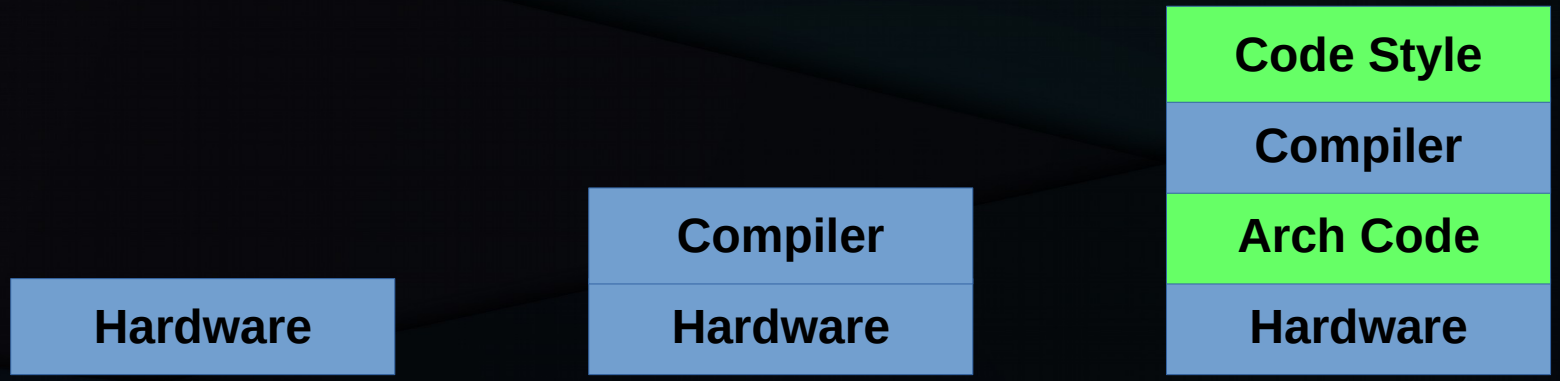
Alglave, Maranget, Pawan, Sarkar, Sewell, Williams, Nardelli:
“PPCMEM/ARMMEM: A Tool for Exploring the POWER and ARM Memory Models”
Alglave, Maranget, and Tautschnig: “Herding Cats: Modelling, Simulation, Testing, and Data-mining for Weak Memory”

Linux-Kernel Memory Model (LKMM)

- Set of .bell and .cat files processed by herd
  - Quasi-C-language input, otherwise like assembly-language tests
  - Very roughly: Intersection of guarantees from CPUs supporting Linux
- Handles READ_ONCE(), WRITE_ONCE(), barriers, atomic operations, locking, RCU, SRCU
  - Accepted into Linux kernel in 2018: tools/memory-model
  - Helped shape RISC-V architecture-specific code
  - Helped abolish spin_unlock_wait(): No agreement on semantics :-)
- Limitations similar to those of herd's assembly features:
  - Small size, exponential complexity, no structs/arrays, …
  - Also does not yet handle plain C-language accesses, seqlock, …
- LKMM is nevertheless useful to Linux kernel hackers

https://lwn.net/Articles/718628/, https://lwn.net/Articles/720550/, http://diy.inria.fr/linux/ (ASPLOS 2018 extended), https://www.youtube.com/watch?v=ULFytshTvIY (linux.conf.au 2017)

Progression of Memory Models



Hardware Memory Model

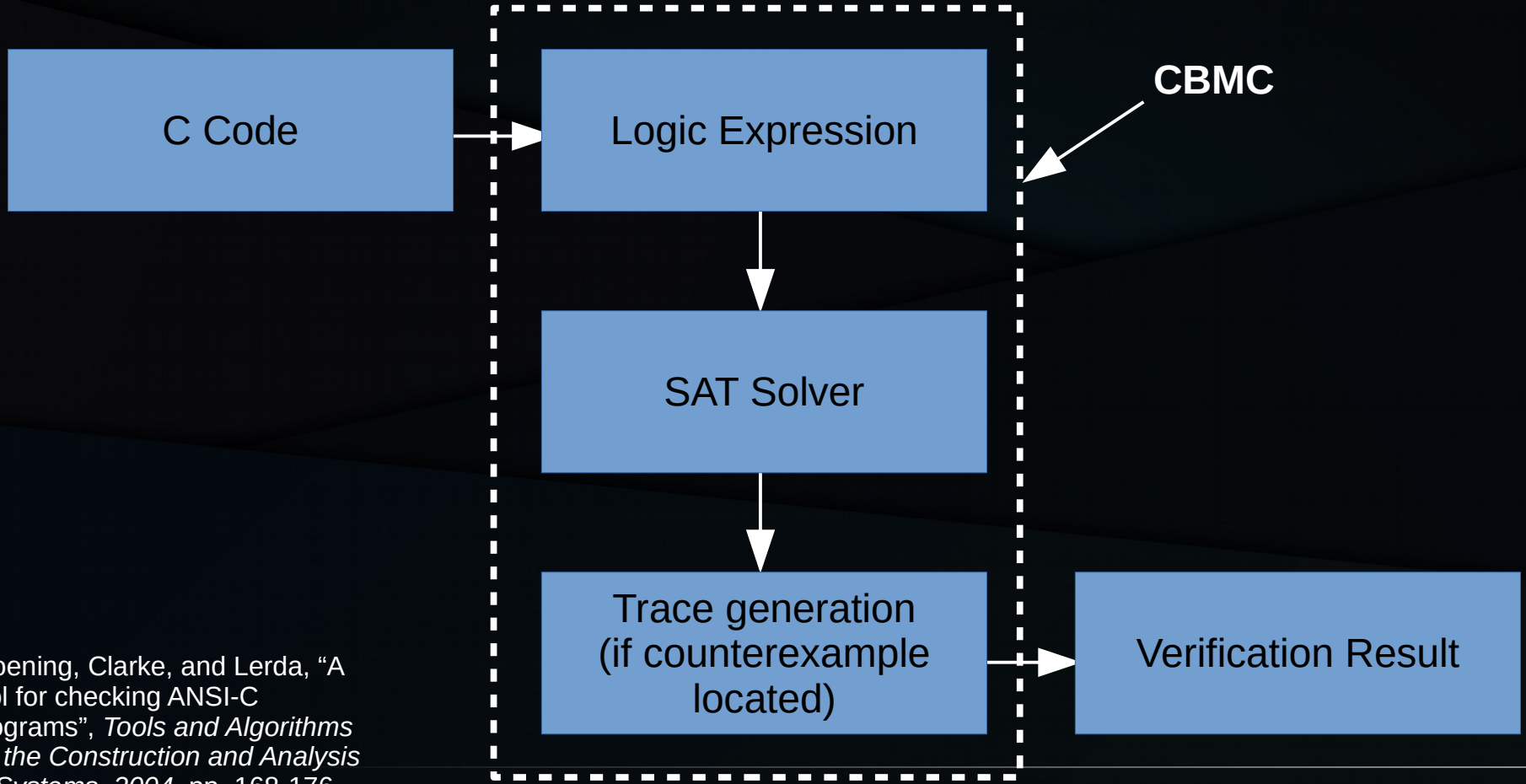
Language Memory Model

Linux-Kernel Memory Model

LKMM Points of Interest

- Loads and stores
- Atomic read-modify-write operations
- Data-race detection
- Locking, RCU, and SRCU modeled directly
- Moral equivalent of full state-space search

CBMC (Very) Rough Schematic



Kroening, Clarke, and Lerda, “A tool for checking ANSI-C programs”, *Tools and Algorithms for the Construction and Analysis of Systems, 2004,* pp. 168-176.
https://github.com/diffblue/cbmc

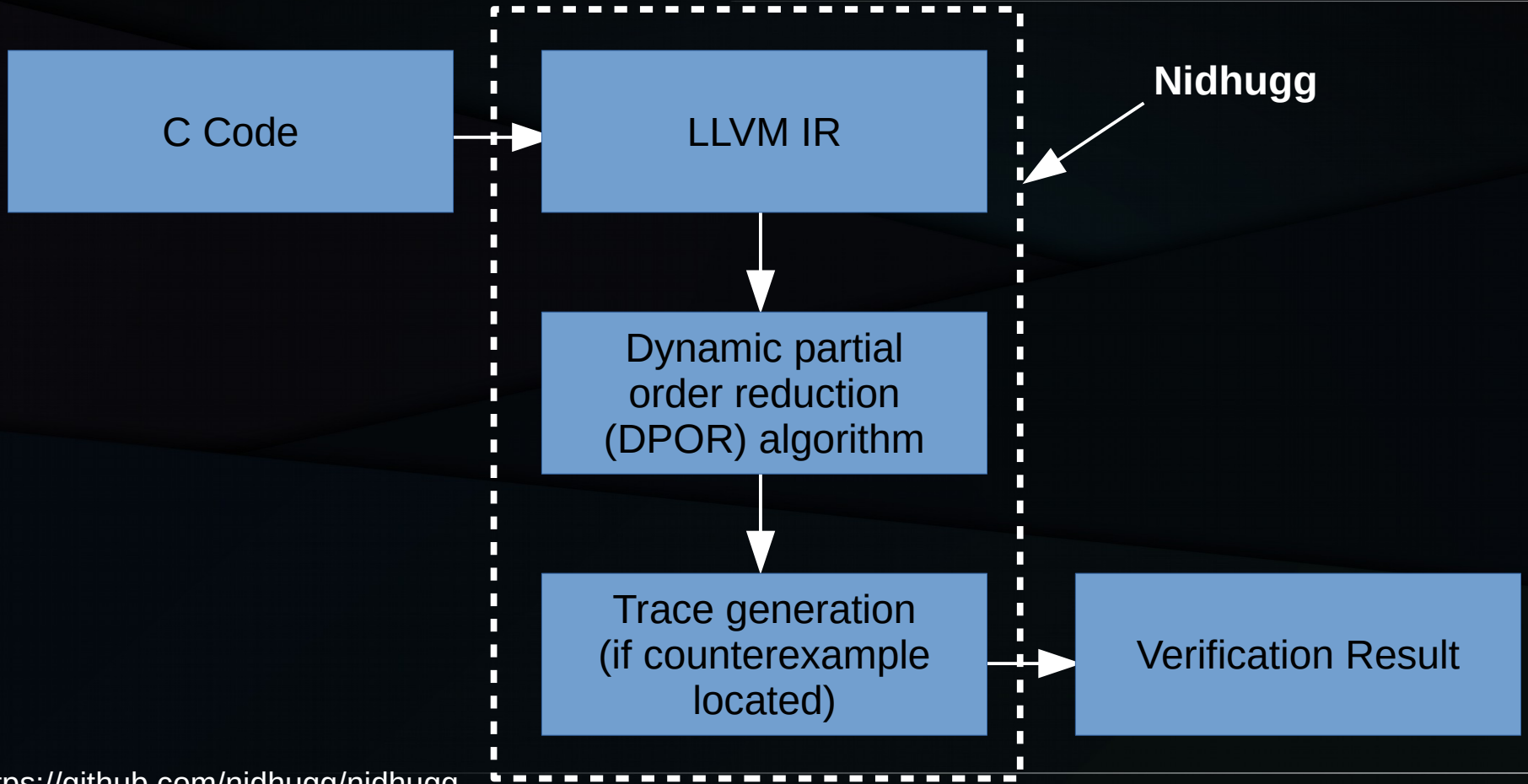
C Bounded Model Checker (CBMC)

- Nascent concurrency and weak-memory functionality
- Valuable property: “Just enough specification”
  - Assertions in code act as specifications!
  - Can provide additional specifications in “verification driver” code
- Verified rcu_dereference() and rcu_assign_pointer()
  - Alglave et al.: https://dl.acm.org/citation.cfm?id=2526873
- I used CBMC to verify Tiny RCU
  - But when I say “Tiny”, I really do mean tiny!!!
- Substantial portion of Tree RCU verified as tour de force
  - Lihao Liang, Oxford, et al.: https://arxiv.org/abs/1610.03052
- Linux-kernel SRCU verified on more routine basis, for awhile, anyway
  - Lance Roy: https://www.spinics.net/lists/kernel/msg2421833.html

C Bounded Model Checker (CBMC)

- Formal verification of Linux-kernel RCU?
  - Sure, I can also write printf(“VERIFIED\n”);
- I therefore maintain bug-injected RCU versions
  - https://paulmck.livejournal.com/46993.html
- How did CBMC do? Only 2 failures out of 30.
  - Interrupt over-approximation, memory exhaustion
  - Up to 90.4M SAT variables, 75GB, ~70 CPU hours
    - Ran on 64-bit 2.4GHz Xeon, 12 cores & 96GB memory
- CBMC is promising, especially if SAT progress continues

Nidhugg (Very) Rough Schematic



https://github.com/nidhugg/nidhugg
https://link.springer.com/chapter/10.1007/978-3-662-46681-0_28

Nidhugg: Stateless Model Checker

- Good concurrency, nascent weak-memory functionality
  - Uses Clang/LLVM, emits LLVM-IR, then analyzes it
- Like CBMC, “Just enough specification”
  - Assertions in code act as specifications!
  - Can provide additional specifications in “verification driver” code
- And also substantial portion of Tree RCU verified
  - Kokologiannakis et al., NTUA: https://doi.org/10.1145/3092282.3092287
- Tentative conclusions comparing to CBMC:
  - Less capable than CBMC (CBMC handles data non-determinism)
  - More scalable than CBMC (Nidhugg analyzes more code faster)
  - But neither found a Linux-kernel bug I didn't already know about
  - Future work includes more detailed comparison
    - And hopefully finding bugs that I don't already know about!

Scorecard For Linux-Kernel C Code (Incomplete)

	Promela	PPCMEM	Herd	LKMM	CBMC	Nidhugg
(1) Automated						
(2) Handle env.	(MM)		(MM)		(MM)	(MM)
(3) Low overhead					SAT?	
(4) Map to source						
(5) Modest input						
(6) Relevant bugs	???	???	???	???	???	???
Paul's first use	1993	2011	2014	2015	2015	2017

Promela/TLA MM: Only SC: Weak memory must be implemented in model
Herd MM: Some PowerPC and ARM corner-case issues
CBMC MM: SC, TSO, and PSO (Want LKMM!)
Nidhugg MM: Only SC, TSO, and nascent Power (Want LKMM!)
Note: All handle concurrency! (Promela has done so for 30 years!!!)

Scorecard For Linux-Kernel C Code

	Promela	PPCMEM	Herd	LKMM	CBMC	Nidhugg	Test
(1) Automated							
(2) Handle env.	(MM)		(MM)		(MM)	(MM)	
(3) Low overhead					SAT?		
(4) Map to source							
(5) Modest input							
(6) Relevant bugs	???	???	???	???	???	???	
Paul's first use	1993	2011	2014	2015	2015	2017	1973

Scorecard For Linux-Kernel C Code

	Promela	PPCMEM	Herd	LKMM	CBMC	Nidhugg	Test
(1) Automated							
(2) Handle env.	(MM)		(MM)		(MM)	(MM)	
(3) Low overhead					SAT?		
(4) Map to source							
(5) Modest input							
(6) Relevant bugs	???	???	???	???	???	???	
Paul's first use	1993	2011	2014	2015	2015	2017	1973

So why do anything other than testing?

Scorecard For Linux-Kernel C Code

	Promela	PPCMEM	Herd	LKMM	CBMC	Nidhugg	Test
(1) Automated							
(2) Handle env.	(MM)		(MM)		(MM)	(MM)	
(3) Low overhead					SAT?		
(4) Map to source							
(5) Modest input							
(6) Relevant bugs	???	???	???	???	???	???	
Paul's first use	1993	2011	2014	2015	2015	2017	1973

So why do anything other than testing?

- Low-probability bugs can require excessively expensive testing regimen
- Large installed base will encounter low-probability bugs
- Safety-critical applications are sensitive to low-probability bugs

More to Life Than Regression Testing!!!

	Promela	PPCMEM	Herd	LKMM	CBMC	Nidhugg	Test
(1) Automated							
(2) Handle env.	(MM)		(MM)		(MM)	(MM)	
(3) Low overhead					SAT?		
(4) Map to source							
(5) Modest input							
(6) Relevant bugs	???	???	???	???	???	???	
Paul's first use	1993	2011	2014	2015	2015	2017	1973
Design?	*	*	*	*	*	*	**
Verify bug?	*	*	*	*	*	*	***
Verify fix?	*	*	*	*	*	*	***

* Assuming no bugs in tool ** Design-time testing *** Weak form of probabilistic testing

Could Better Things Happen?

Challenges/Limitations/Future Work

- Better modeling of interrupts & kernel threads
  - For CBMC: model concurrent linked lists for call_rcu()
- Incorporate Linux-kernel memory model
  - And/or ARM, PowerPC, RISC-V, ...
- Forward progress: Detect hangs & deadlocks
  - Can already detect unconditional hangs/deadlocks
- Fully analyze unbounded looping
  - Or at least automatically derive unrolling bounds
- Larger programs: Automatic decomposition?
  - RacerD is a small but important step in this direction

Why Automatic Decomposition???

LKMM Locking: Modeling vs. Emulation					
	Modeling	Emulation			
# threads	lock.cat	CAS Filter	xchg() Filter	CAS Exists	xchg() Exists
2	0.004	0.022	0.027	0.039	0.058
3	0.041	0.743	0.968	1.653	3.203
4	0.374	59.565	74.818	151.962	500.96
5	4.905				

CAS Filter: Emulate with `cmpxchg_acquire()` and “filter” clause
xchg() Filter: Emulate with `xchg_acquire()` and “filter” clause
CAS Exists: Emulate with `cmpxchg_acquire()` and “exists” clause
xchg() Exists: Emulate with `xchg_acquire()` and “exists” clause

Why Automatic Decomposition???

- Exponential runtime is expected behavior
  - On a bad day, you instead get undecidability!!!
- Therefore, huge performance and scalability benefits from:
  - Goal: Combinatorial ***implosion***
  - Higher levels of abstraction (vertical decomposition)
    - Verify use of locking instead of both use and implementation!!!
  - Partitioning code to be verified (horizontal decomposition)
- Decomposition is common practice in hardware verification
  - And starting to appear in software, but a very long way to go
- Automation is required for use in regression test suites

Why Automatic Decomposition???

LKMM Locking: Modeling vs. Emulation					
	Modeling	Emulation			
# threads	lock.cat	CAS Filter	xchg() Filter	CAS Exists	xchg() Exists
2	0.004	0.022	0.027	0.039	0.058
3	0.041	0.743	0.968	1.653	3.203
4	0.374	59.565	74.818	151.962	500.96
5	4.905				

Verification Challenge

- Find Linux-kernel bugs that I don't already know about!
  - Find bug in rcu_preempt_offline_tasks()
    - http://paulmck.livejournal.com/37782.html
  - Find bug in RCU_NO_HZ_FULL_SYSIDLE
    - http://paulmck.livejournal.com/38016.html
  - Find bug in RCU linked-list use cases
    - http://paulmck.livejournal.com/39793.html
  - Verification Challenges 6 and 7
    - https://paulmck.livejournal.com/46993.html
    - https://paulmck.livejournal.com/50441.html
- Or find bugs in other popular open-source SW

Linux-Kernel RCU Bug Expectations

The usual influx of bugs that I don't expect at all...

Linux-Kernel RCU Bug Expectations

The usual influx of bugs that I don't expect at all...

Because Murphy Never Sleeps!!!

Case in Point From 2018

Date: S[illegible] Mar 2018 17:50:44 -0800
From: [illegible]orvalds <torvalds@linux-foundation.org>
To: [illegible]jannh@google.com>, Tejun Heo <tj@kernel.org>, Paul McKenney [illegible]@linux.vnet.ibm.com>
Cc: [illegible]se <bcrl@kvack.org>, security@kernel.org, Al Viro [illegible]linux.org.uk>
Subject: [illegible]ing bug in lookup_ioctx()
From lin[illegible] Sat Mar 3 17:54:39 2018

[Adding Al, [illegible]n and to the cc too for various reasons]

On Fri, Mar 2, [illegible]M, Jann Horn <jannh@google.com> wrote:

[. . .]

> I'm not sending a p[illegible]'m not sure whether the intent here is to
> use RCU, and if so, w[illegible]ld be RCU-sched or normal RCU.

It's meant to use regular

But then in commit a4244454df[illegible]fcount: use RCU-sched
insted of normal RCU") the perc[illegible]ere changed to use
RCU-sched.

.. and in the process apparently br[illegible]CU locking.

Tejun, Paul, please tell me why I'm wr[illegible]

Linus

http://youtu.be/hZX1aokdNiY

linux.conf.au video:

security@kernel.org

LKML

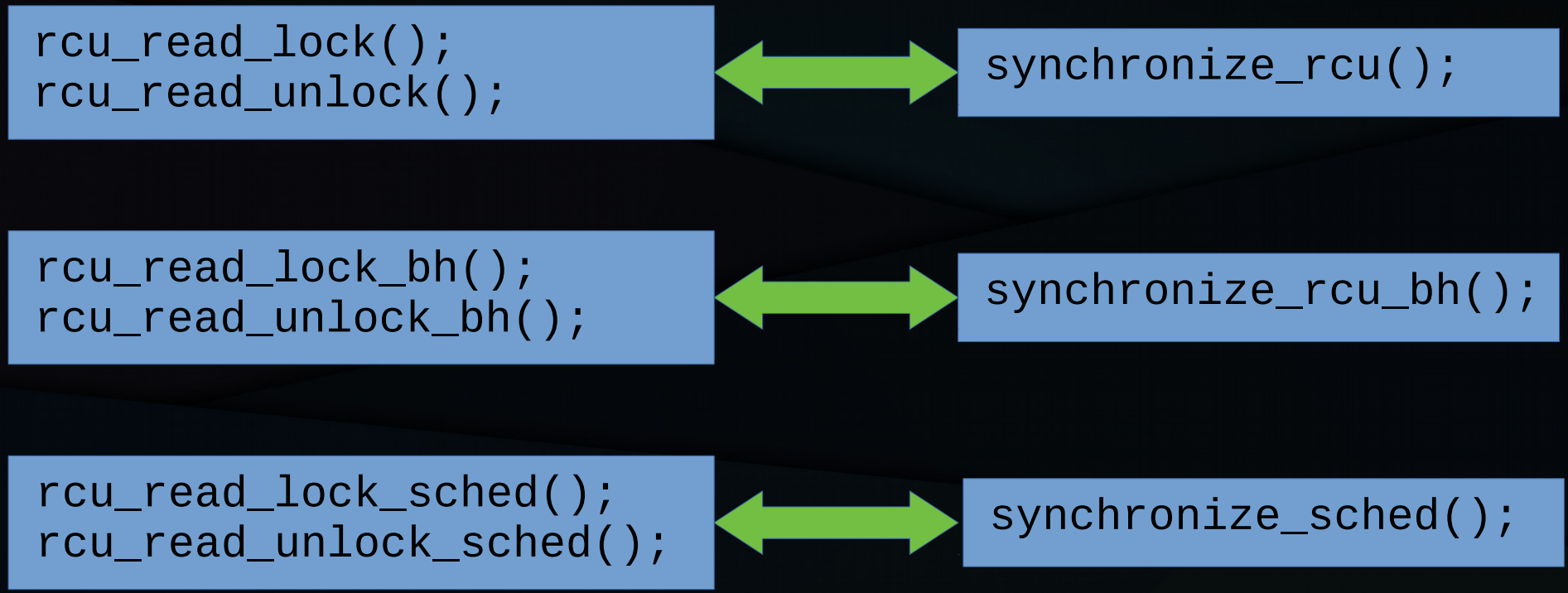
What Was The Problem???

```
void reader(void)
{
  rcu_read_lock_sched();
  /*
   * Access RCU-
   * protected data.
   */
  rcu_read_unlock_sched();
}
```

```
void updater(void)
{
  /* Remove old data. */
  synchronize_rcu();
  /* Free old data. */
}
```

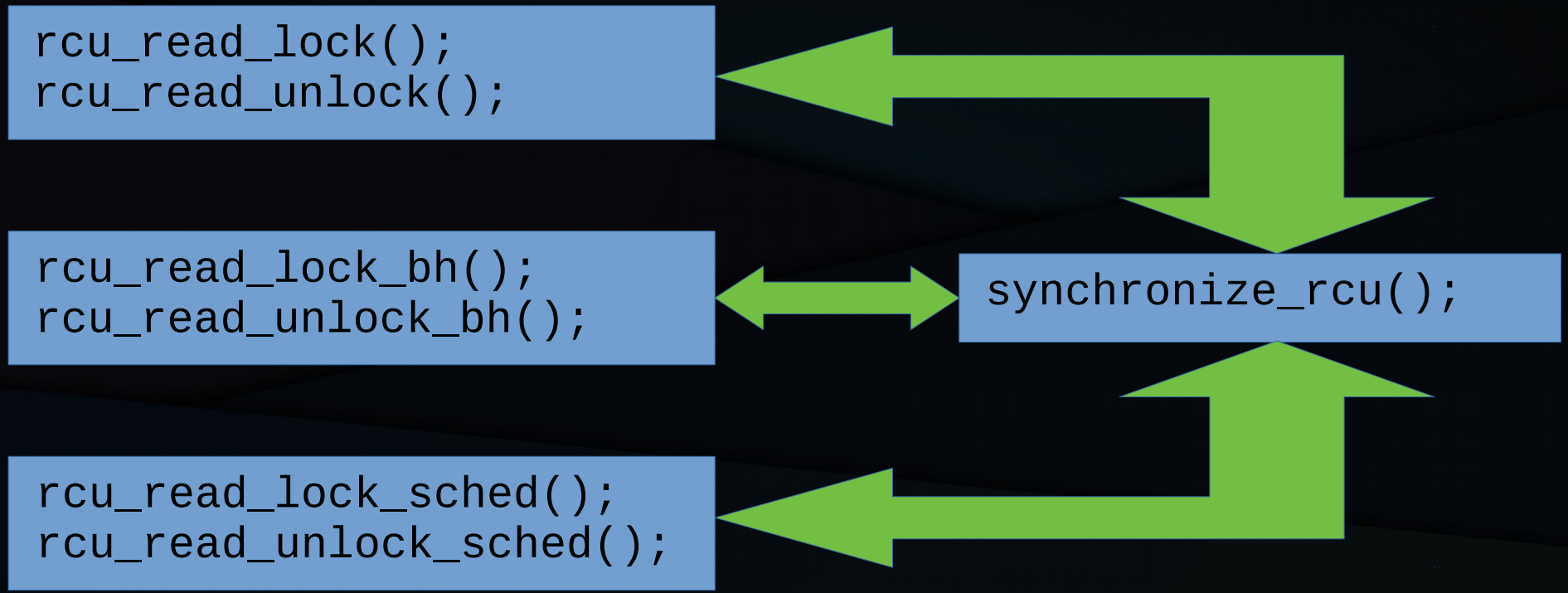
This is about as healthy for your kernel as acquiring the wrong lock!!!

Consistency Required, Which is Bad!



To err is human...
Plus userspace controls content of much kernel data!!!

Desired State (Usability & Security)



But Non-Trivial to Fix

- 8 patches to consolidate the three flavors of RCU

But Non-Trivial to Fix

- 35 patches to merge grace-period counters
- 8 patches to consolidate the three flavors of RCU

But Non-Trivial to Fix

- 3 patches to funnel-lock grace-period start
- 35 patches to merge grace-period counters
- 8 patches to consolidate the three flavors of RCU

But Non-Trivial to Fix

- 15 patches for pre-existing rcutorture failures
- 3 patches to funnel-lock grace-period start
- 35 patches to merge grace-period counters
- 8 patches to consolidate the three flavors of RCU

But Non-Trivial to Fix

- 17 patches to add debugging code
- 15 patches for pre-existing rcutorture failures
- 3 patches to funnel-lock grace-period start
- 35 patches to merge grace-period counters
- 8 patches to consolidate the three flavors of RCU

But Non-Trivial to Fix

- 42 patches to add rcutorture tests
- 17 patches to add debugging code
- 15 patches for pre-existing rcutorture failures
- 3 patches to funnel-lock grace-period start
- 35 patches to merge grace-period counters
- 8 patches to consolidate the three flavors of RCU

But Non-Trivial to Fix

- 107 patches to remove RCU-bh & RCU-sched and simplify
- 42 patches to add rcutorture tests
- 17 patches to add debugging code
- 15 patches for pre-existing rcutorture failures
- 3 patches to funnel-lock grace-period start
- 35 patches to merge grace-period counters
- 8 patches to consolidate the three flavors of RCU

But Non-Trivial to Fix

- 17 patches for drive-by optimizations
- 107 patches to remove RCU-bh & RCU-sched and simplify
- 42 patches to add rcutorture tests
- 17 patches to add debugging code
- 15 patches for pre-existing rcutorture failures
- 3 patches to funnel-lock grace-period start
- 35 patches to merge grace-period counters
- 8 patches to consolidate the three flavors of RCU

But Non-Trivial to Fix

- 17 patches for drive-by optimizations
- 107 patches to remove RCU-bh & RCU-sched and simplify
- 42 patches to add rcutorture tests
- 17 patches to add debugging code
- 15 patches for pre-existing rcutorture failures
- 3 patches to funnel-lock grace-period start
- 35 patches to merge grace-period counters
- 8 patches to consolidate the three flavors of RCU

And formal verification would not have helped to find this one...

Summary

Summary

- Making your software do exactly what you want it to is a difficult undertaking
  - And it is insufficient: You might be confused about requirements
- Ease-of-use issues can result in security holes
  - Testing and reliability statistics are subject to misuse “Black Swans”
  - On the other hand, fixing these issues can simplify your code
- RCU currently seems to be in pretty good shape
  - But recent change means opportunity for formal verification
  - And there is some risk due to lack of synchronize_sched()

Summary

- Making your software do exactly what you want it to is a difficult undertaking
  - And it is insufficient: You might be confused about requirements
- Ease-of-use issues can result in security holes
  - Testing and reliability statistics are subject to misuse “Black Swans”
  - On the other hand, fixing these issues can simplify your code
- RCU currently seems to be in pretty good shape
  - But recent change means opportunity for formal verification
  - And there is some risk due to lack of synchronize_sched()
- For most validation tasks, testing still has highest ROI
- But testing does have limitations, so additional validation help would be extremely welcome!!!

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/57769.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/