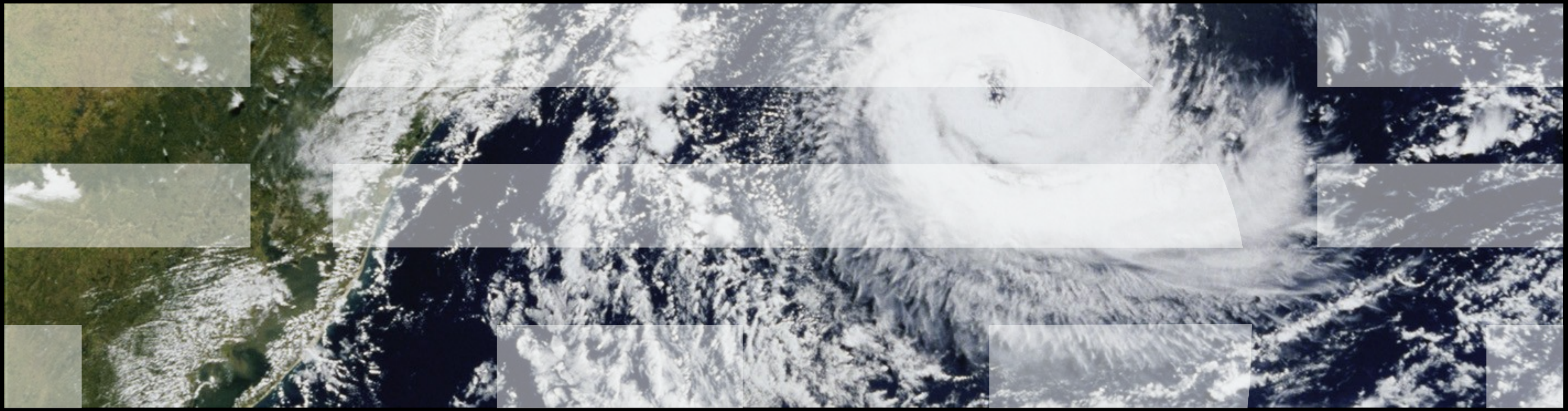
Paul E. McKenney, IBM Distinguished Engineer, Linux Technology Center
Member, IBM Academy of Technology
November 1, 2013

What Is RCU?

Guest Lecture for University of Cambridge



Overview

- Mutual Exclusion
- Example Application
- Performance of Synchronization Mechanisms
- Making Software Live With Current (and Future) Hardware
- Implementing RCU
- RCU Grace Periods: Conceptual and Graphical Views
- Performance
- RCU Area of Applicability
- Summary

Mutual Exclusion

Mutual Exclusion

- What mechanisms can enforce mutual exclusion?

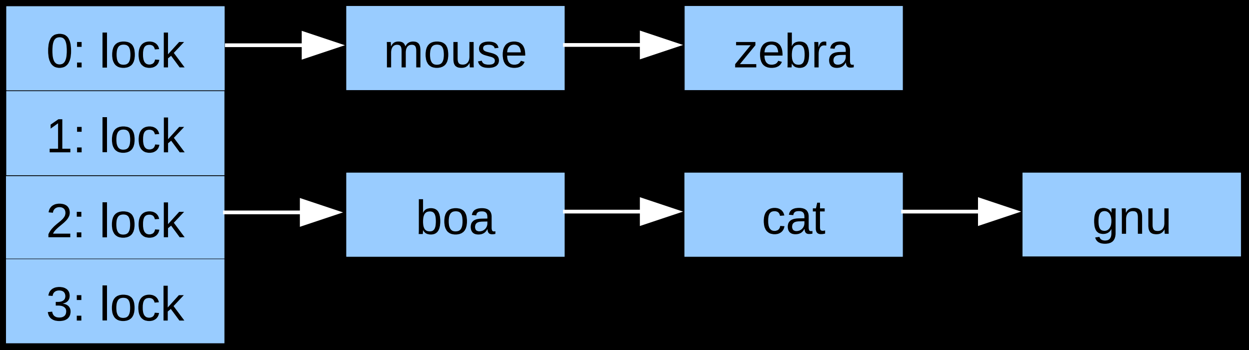
Example Application

Example Application

- Schrödinger wants to construct an in-memory database for the animals in his zoo (example in upcoming ACM Queue)
  - Births result in insertions, deaths in deletions
  - Queries from those interested in Schrödinger's animals
  - Lots of short-lived animals such as mice: High update rate
  - Great interest in Schrödinger's cat (perhaps queries from mice?)

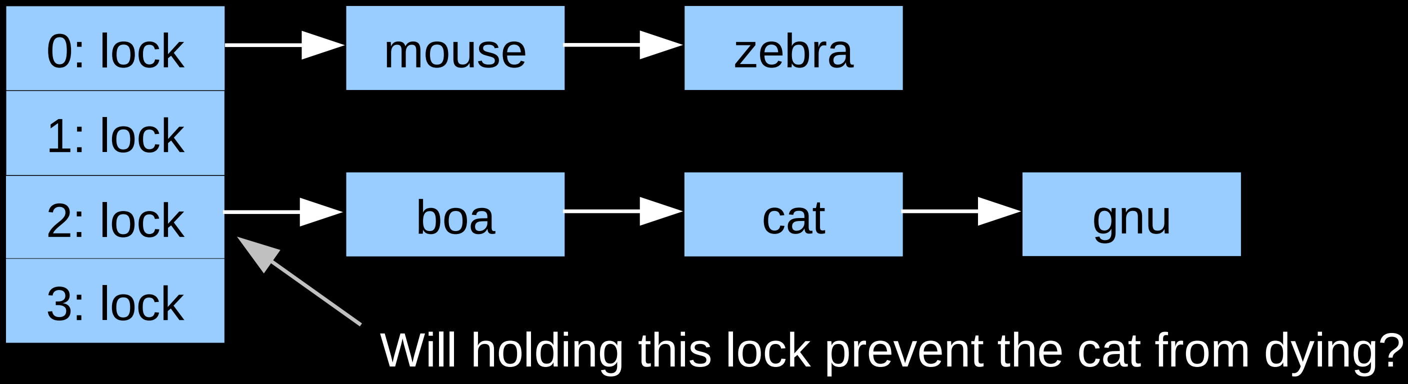
Example Application

- Schrödinger wants to construct an in-memory database for the animals in his zoo (example in upcoming ACM Queue)
  - Births result in insertions, deaths in deletions
  - Queries from those interested in Schrödinger's animals
  - Lots of short-lived animals such as mice: High update rate
  - Great interest in Schrödinger's cat (perhaps queries from mice?)
- Simple approach: chained hash table with per-bucket locking

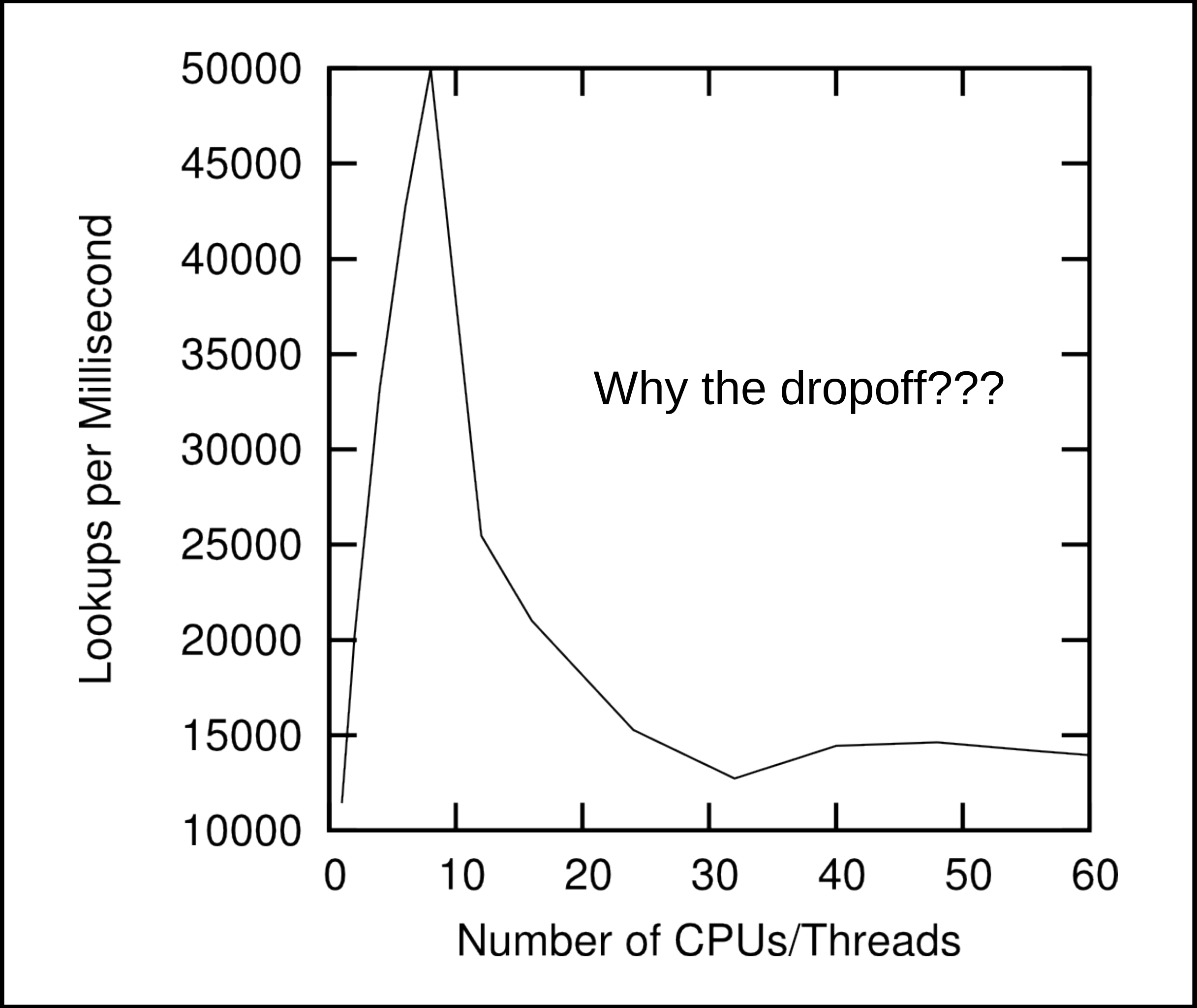


Example Application

- Schrödinger wants to construct an in-memory database for the animals in his zoo (example in upcoming ACM Queue)
  - Births result in insertions, deaths in deletions
  - Queries from those interested in Schrödinger's animals
  - Lots of short-lived animals such as mice: High update rate
  - Great interest in Schrödinger's cat (perhaps queries from mice?)
- Simple approach: chained hash table with per-bucket locking

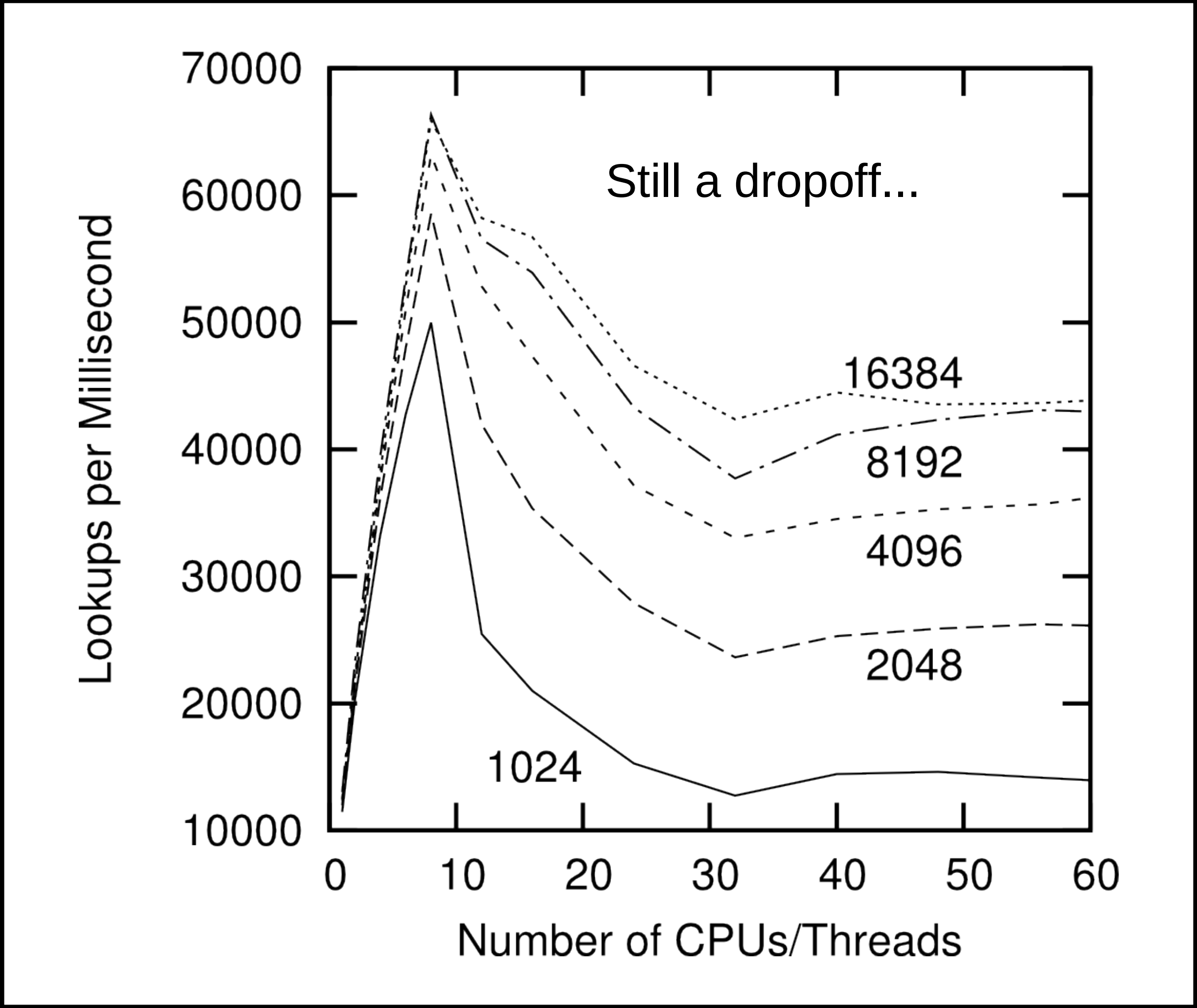


Read-Only Bucket-Locked Hash Table Performance



2GHz Intel Xeon Westmere-EX, 1024 hash buckets

Varying Number of Hash Buckets

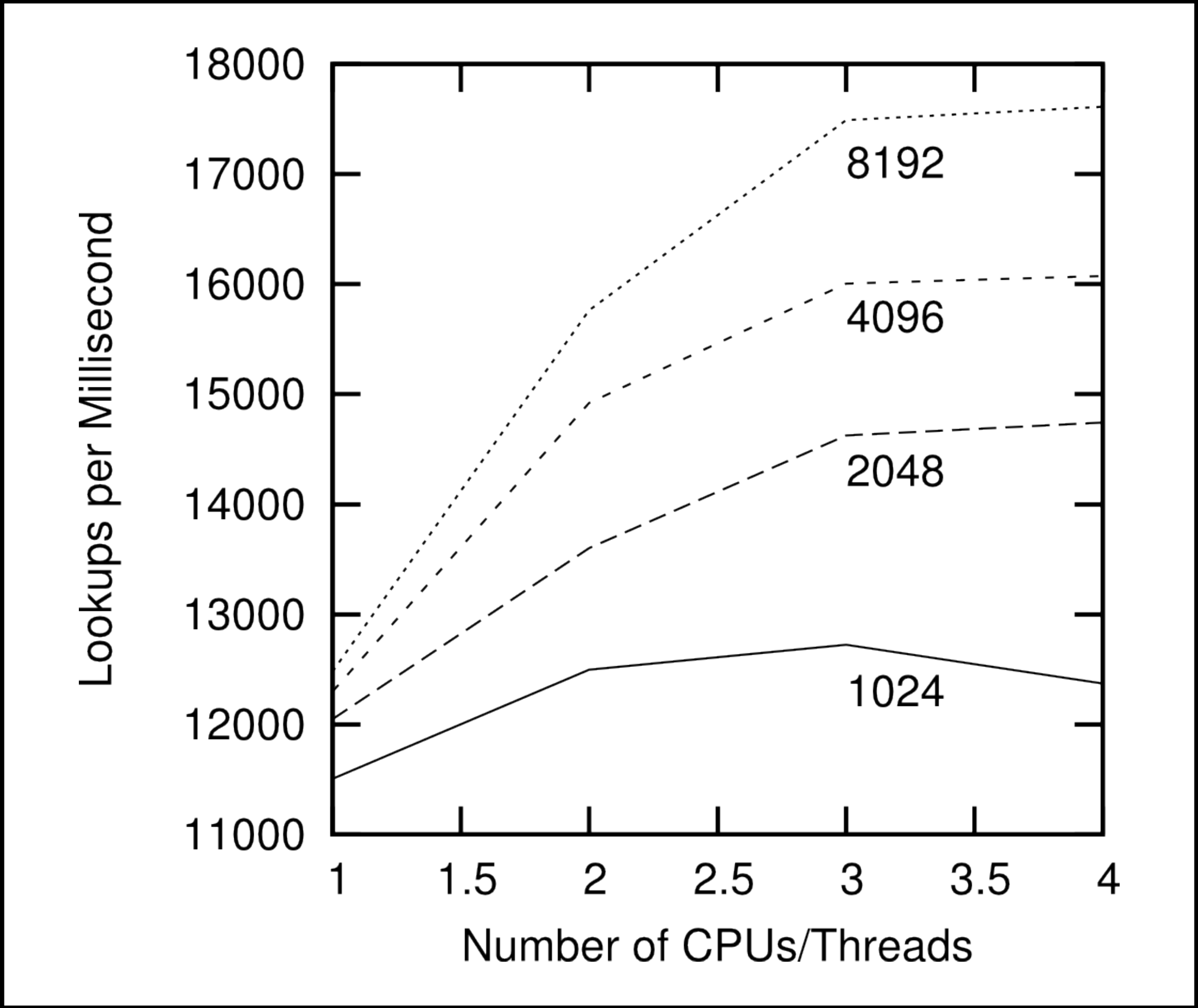


2GHz Intel Xeon Westmere-EX

NUMA Effects???

- /sys/devices/system/cpu/cpu0/cache/index0/shared_cpu_list:
  - 0,32
- /sys/devices/system/cpu/cpu0/cache/index1/shared_cpu_list:
  - 0,32
- /sys/devices/system/cpu/cpu0/cache/index2/shared_cpu_list:
  - 0,32
- /sys/devices/system/cpu/cpu0/cache/index3/shared_cpu_list:
  - **0-7,32-39**
- Two hardware threads per core, eight cores per socket
- Try using only one CPU per socket: CPUs 0, 8, 16, and 24

Bucket-Locked Hash Performance: 1 CPU/Socket



This is not the sort of scalability Schrödinger requires!!!

Performance of Synchronization Mechanisms

Performance of Synchronization Mechanisms

16-CPU 2.8GHz Intel X5550 (Nehalem) System

Operation	Cost (ns)	Ratio
Clock period	0.4	1
“Best-case” CAS	12.2	33.8
Best-case lock	25.6	71.2
Single cache miss	12.9	35.8
CAS cache miss	7.0	19.4
Single cache miss **(off-core)**	31.2	86.6
CAS cache miss **(off-core)**	31.2	86.5
Single cache miss **(off-socket)**	92.4	256.7
CAS cache miss **(off-socket)**	95.9	266.4

And these are best-case values!!! (Why?)

Why All These Low-Level Details???

Why All These Low-Level Details???

- Would you trust a bridge designed by someone who did not understand strengths of materials?
  - Or a ship designed by someone who did not understand the steel-alloy transition temperatures?
  - Or a house designed by someone who did not understand that unfinished wood rots when wet?
  - Or a car designed by someone who did not understand the corrosion properties of the metals used in the exhaust system?
  - Or a space shuttle designed by someone who did not understand the temperature limitations of O-rings?

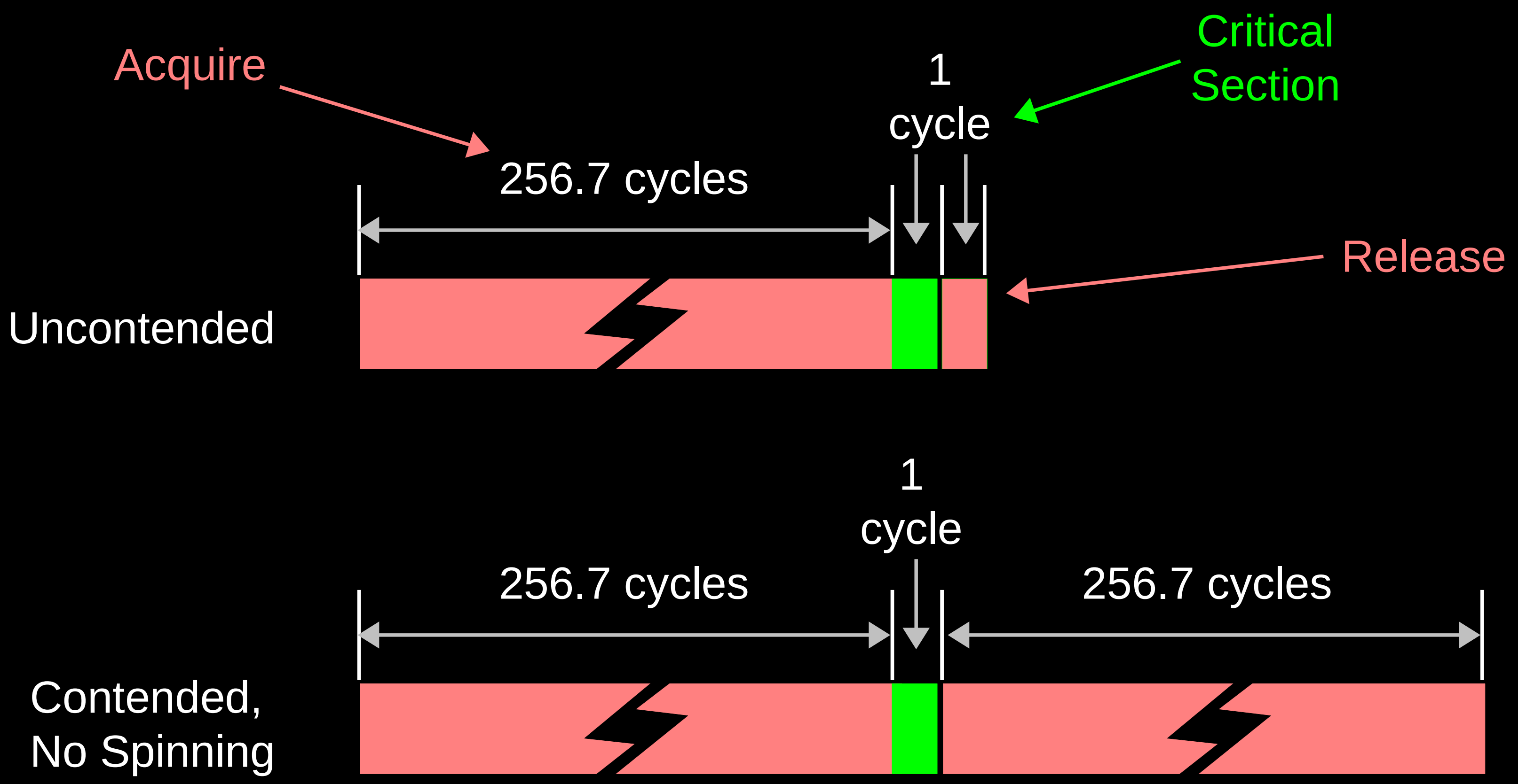
Why All These Low-Level Details???

- Would you trust a bridge designed by someone who did not understand strengths of materials?
  - Or a ship designed by someone who did not understand the steel-alloy transition temperatures?
  - Or a house designed by someone who did not understand that unfinished wood rots when wet?
  - Or a car designed by someone who did not understand the corrosion properties of the metals used in the exhaust system?
  - Or a space shuttle designed by someone who did not understand the temperature limitations of O-rings?
- So why trust algorithms from someone ignorant of the properties of the underlying hardware???

But What Do The Operation Timings Really Mean???

But What Do The Operation Timings Really Mean???

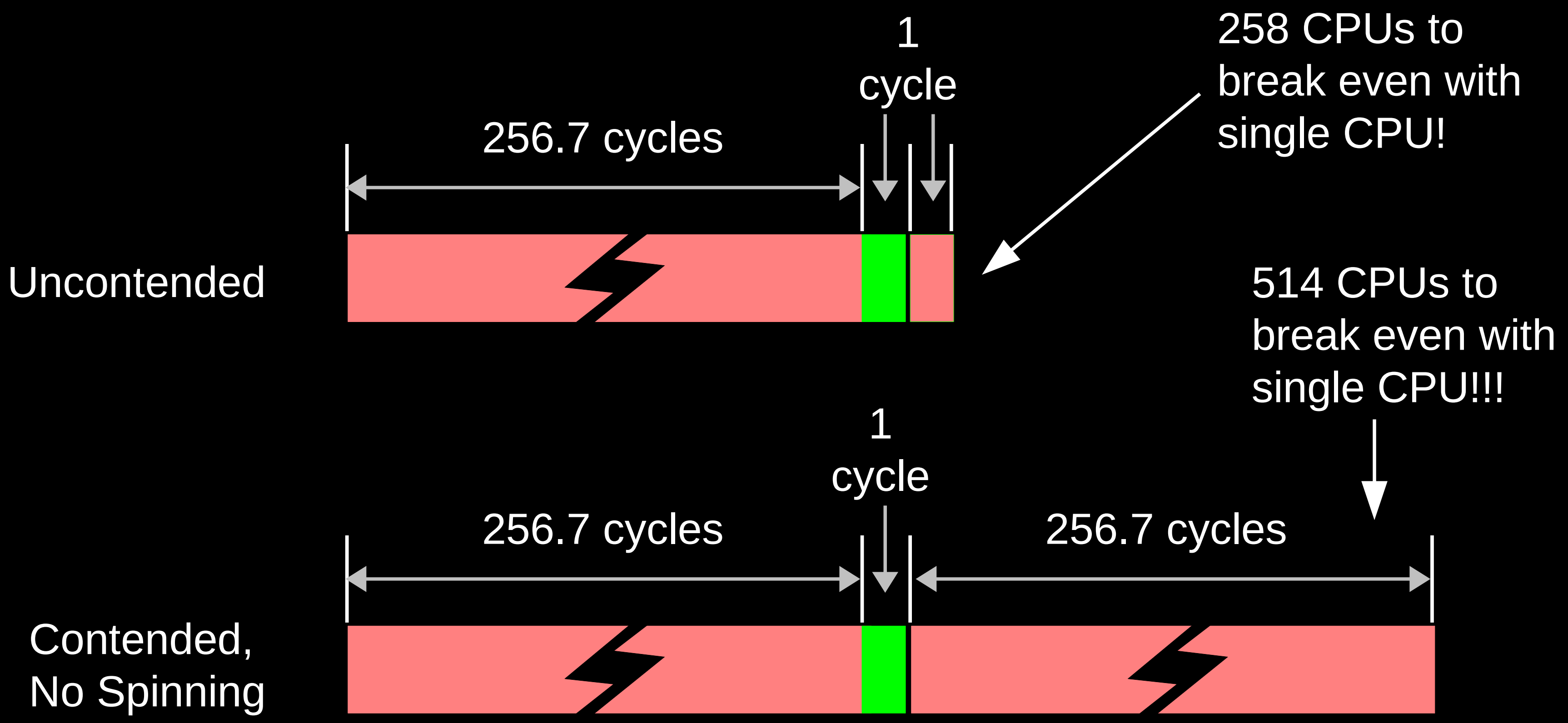
- Single-instruction critical sections protected by multiple locks



So, what *does* this mean?

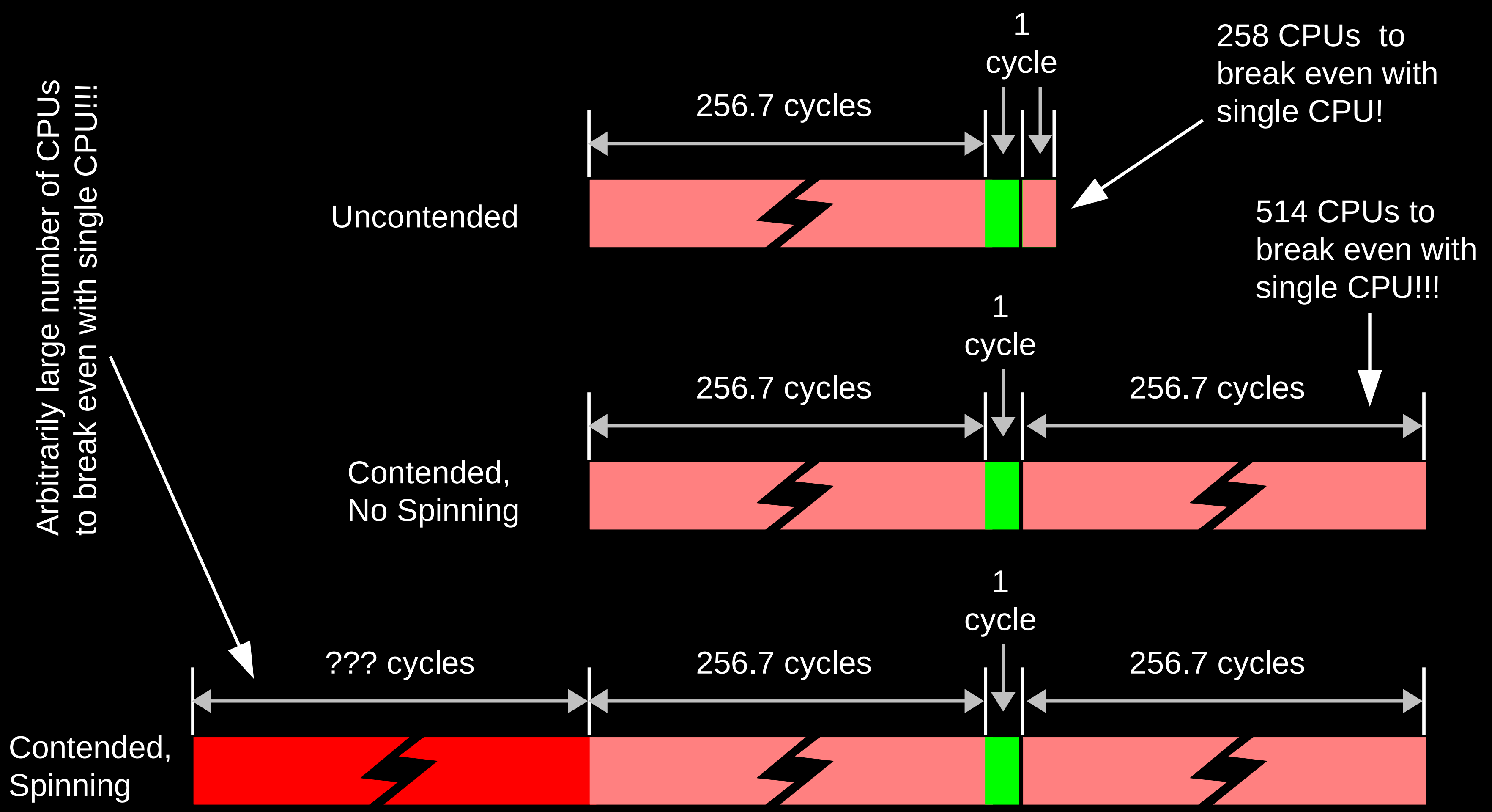
But What Do The Operation Timings Really Mean???

- Single-instruction critical sections protected by multiple locks



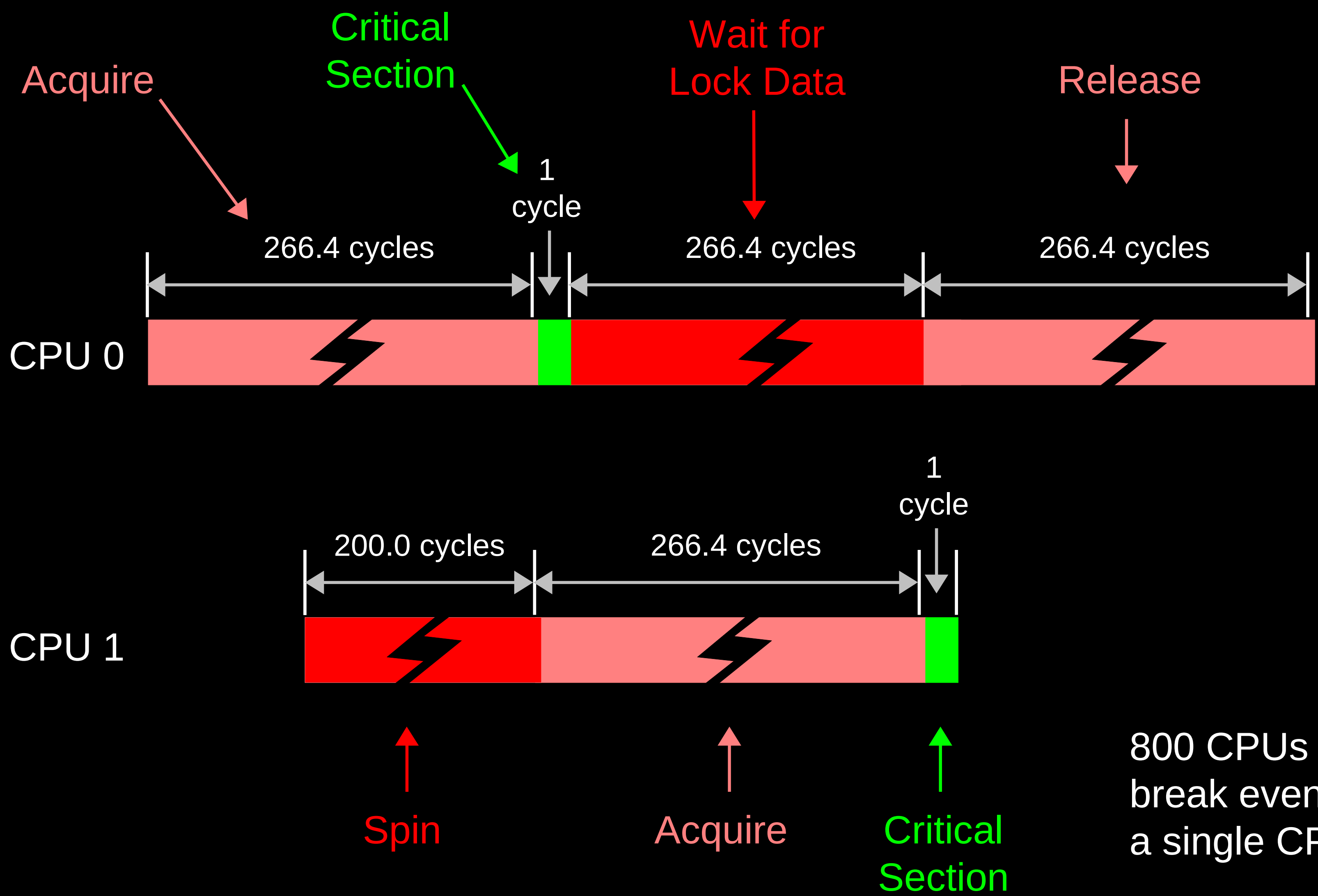
But What Do The Operation Timings Really Mean???

- Single-instruction critical sections protected by multiple locks



Reader-Writer Locks Are Even Worse!

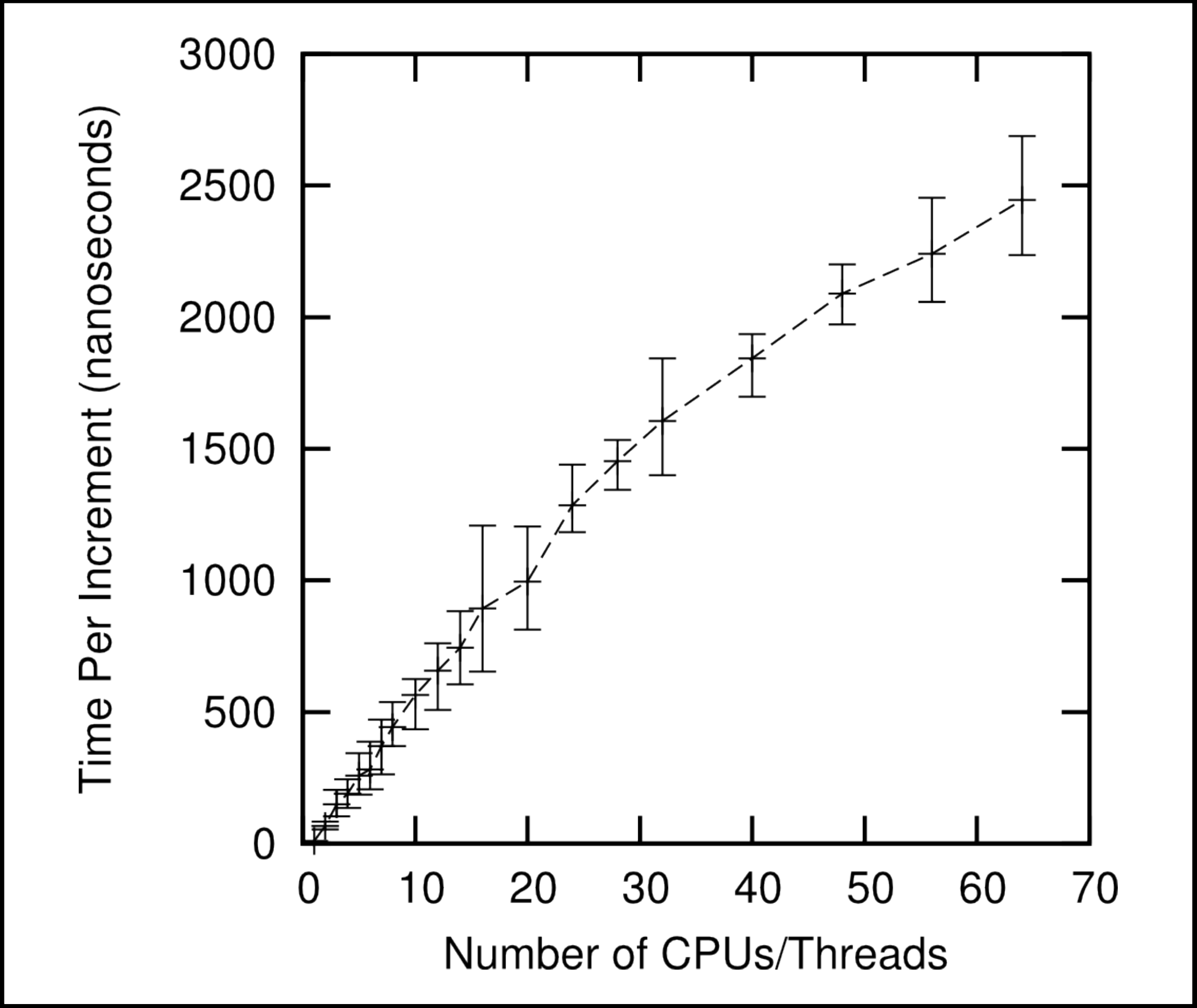
Reader-Writer Locks Are Even Worse!



800 CPUs to break even with a single CPU!!!

But What About Scaling With Atomic Operations?

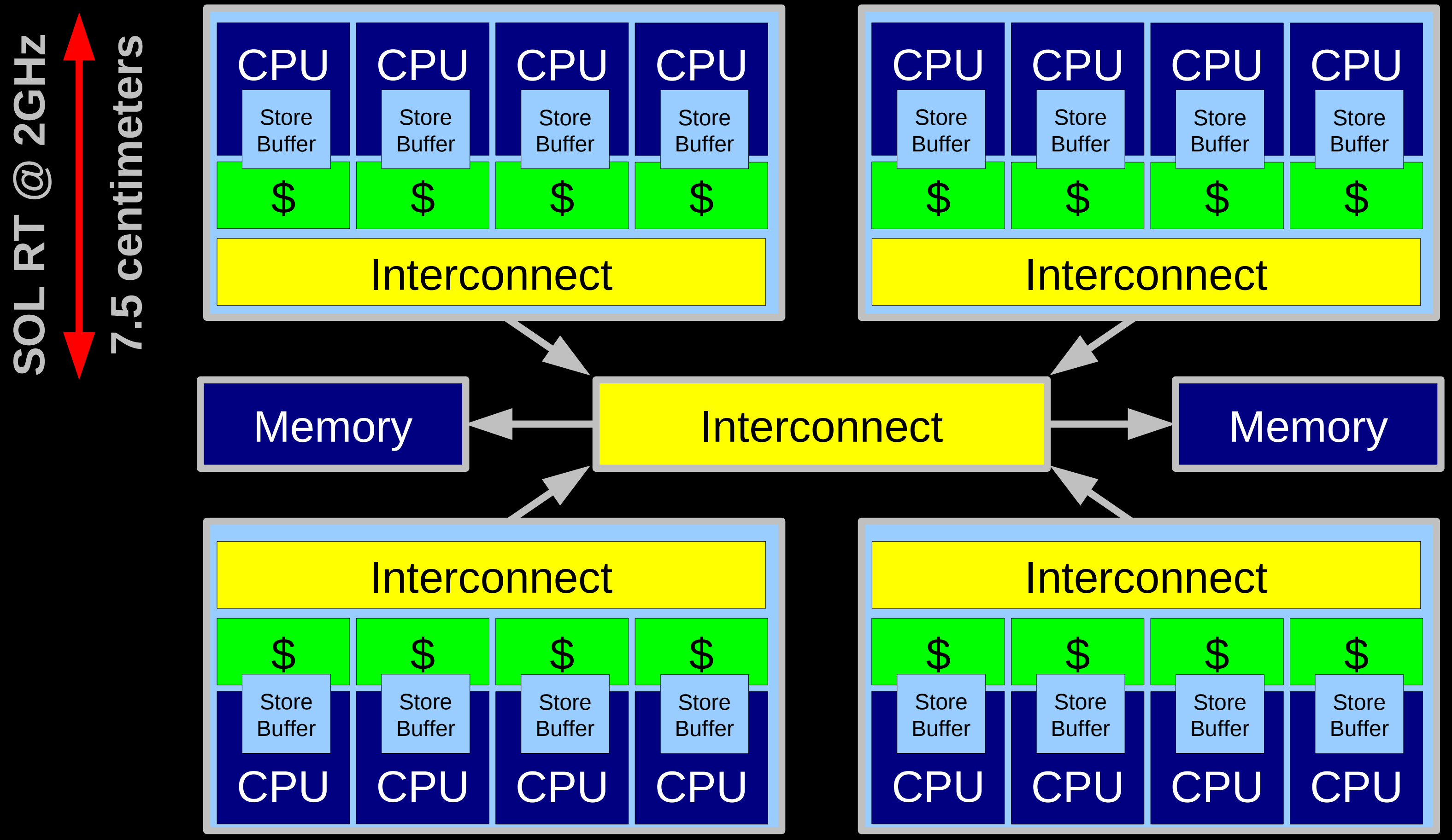
If You Think Single Atomic is Expensive, Try Lots!!!



2GHz Intel Xeon Westmere-EX

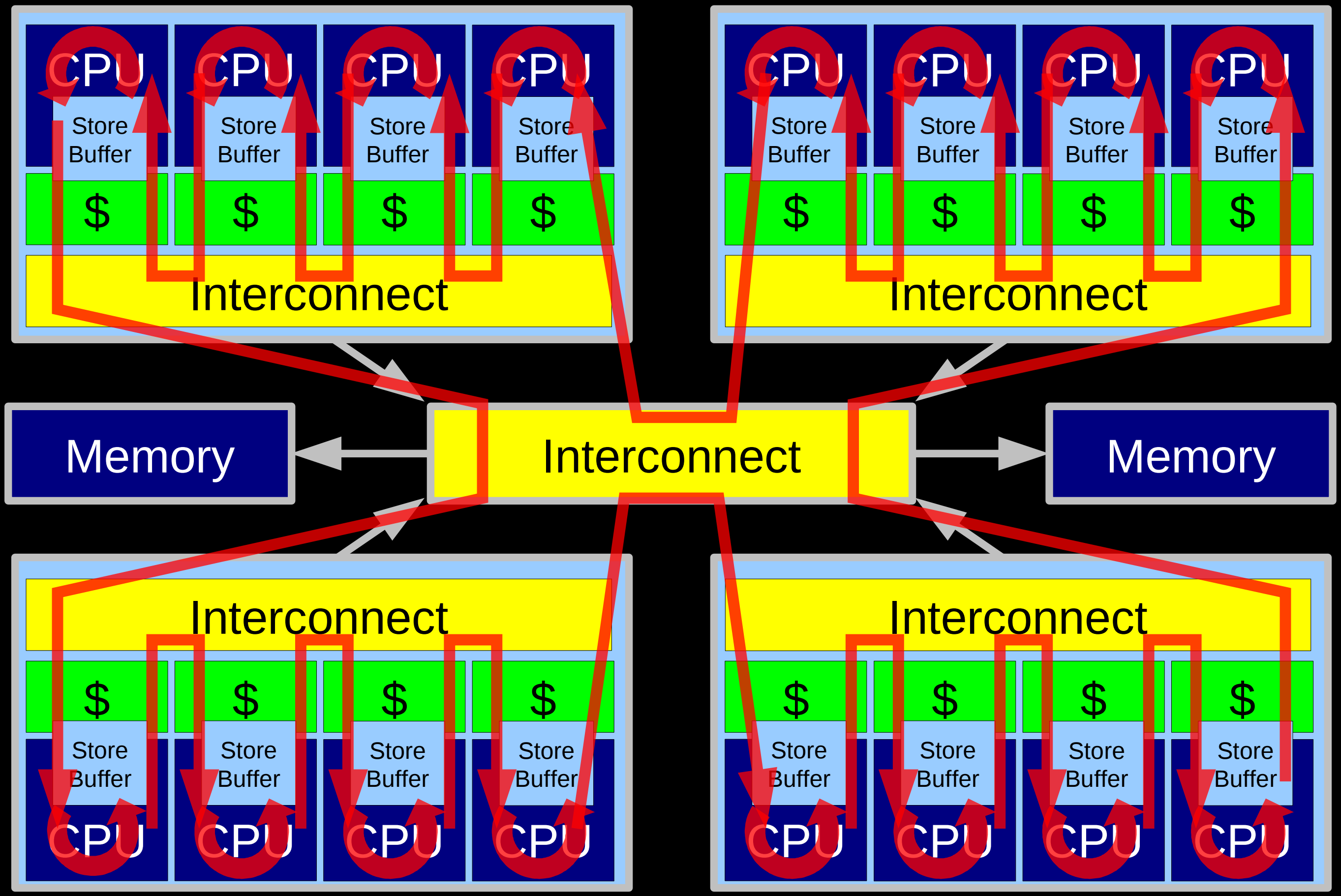
Why So Slow???

System Hardware Structure and Laws of Physics



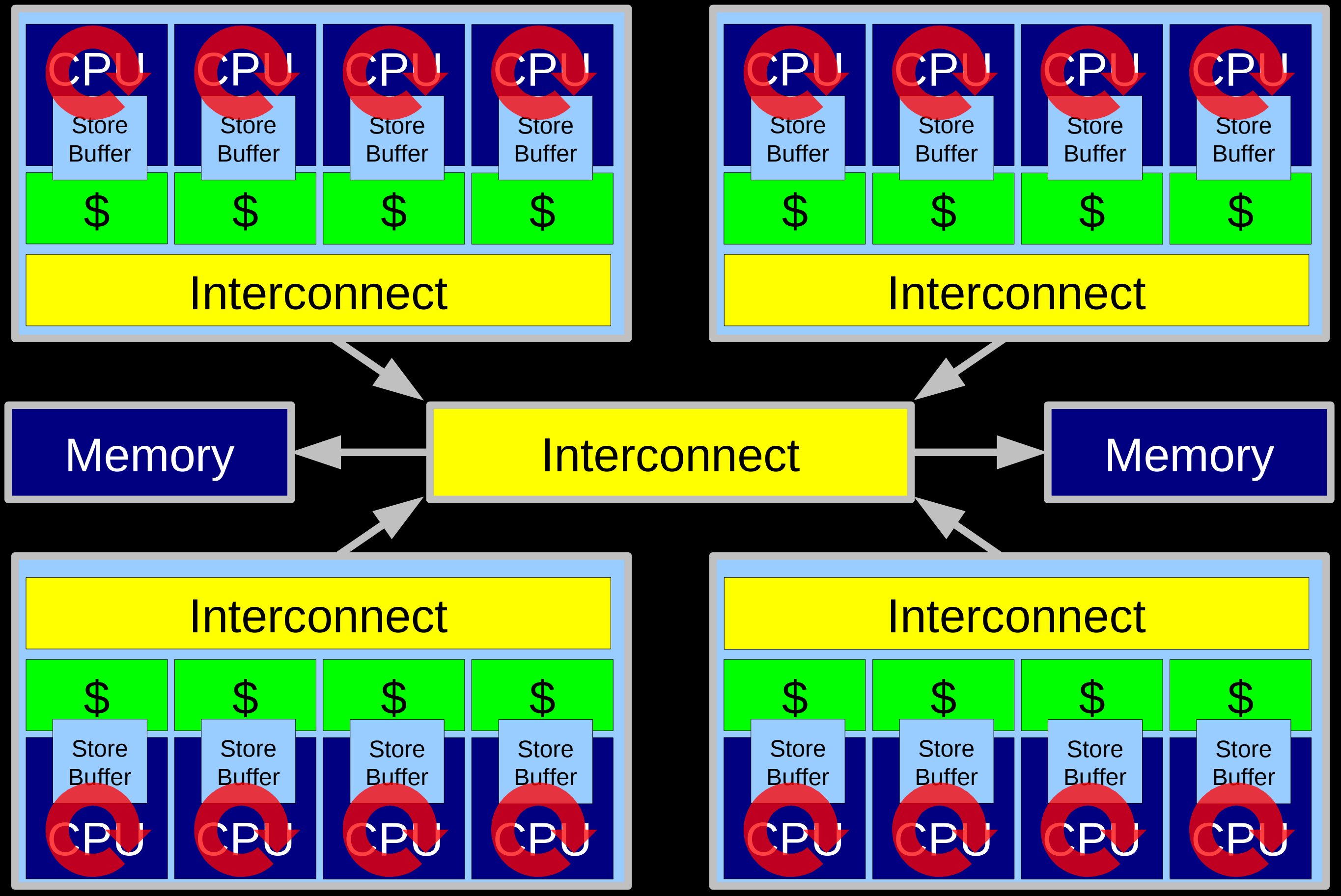
Electrons move at 0.03C to 0.3C in transistors and, so lots of waiting. 3D???

Atomic Increment of Global Variable



Lots and Lots of Latency!!!

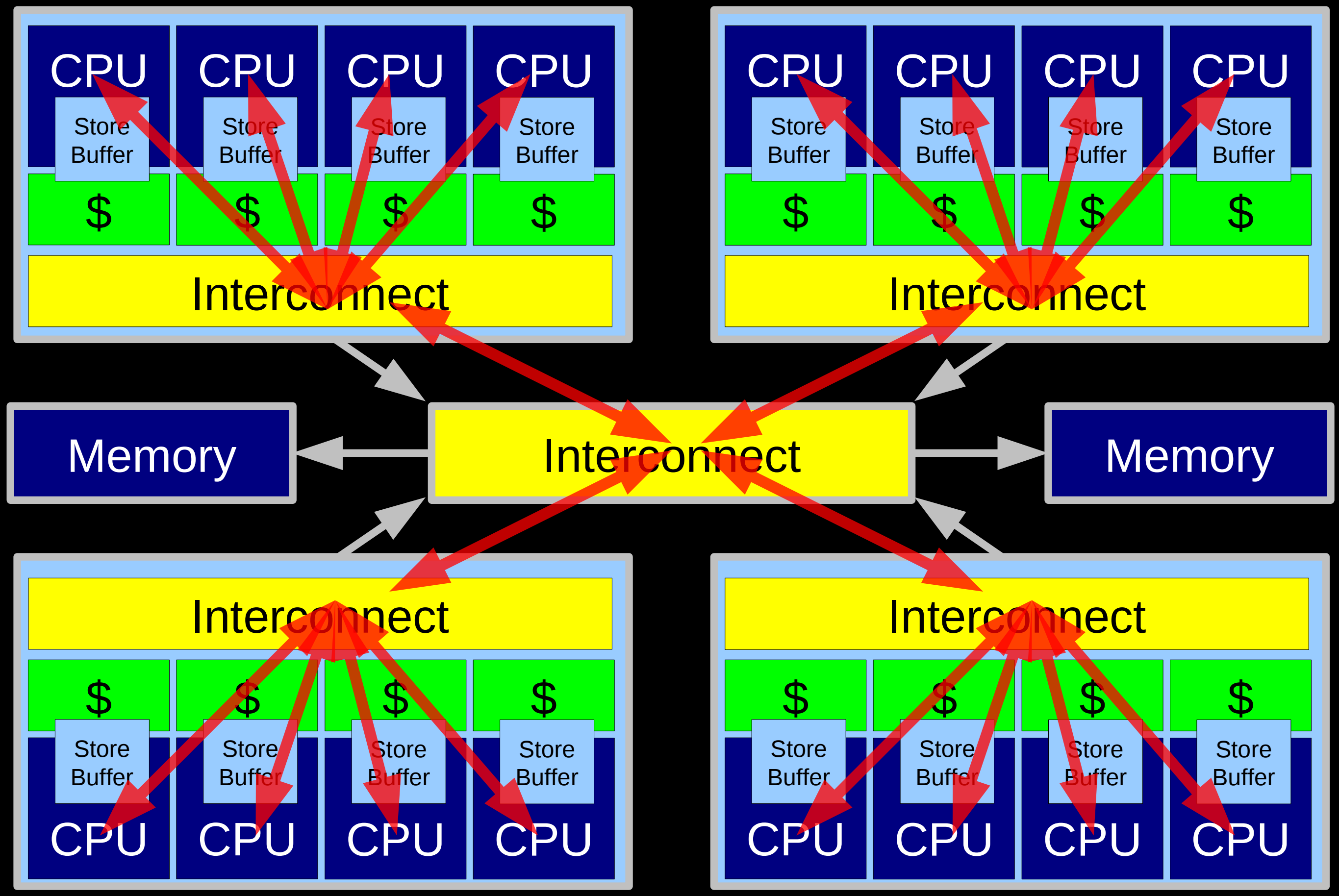
Atomic Increment of Per-CPU Counter



Little Latency, Lots of Increments at Core Clock Rate

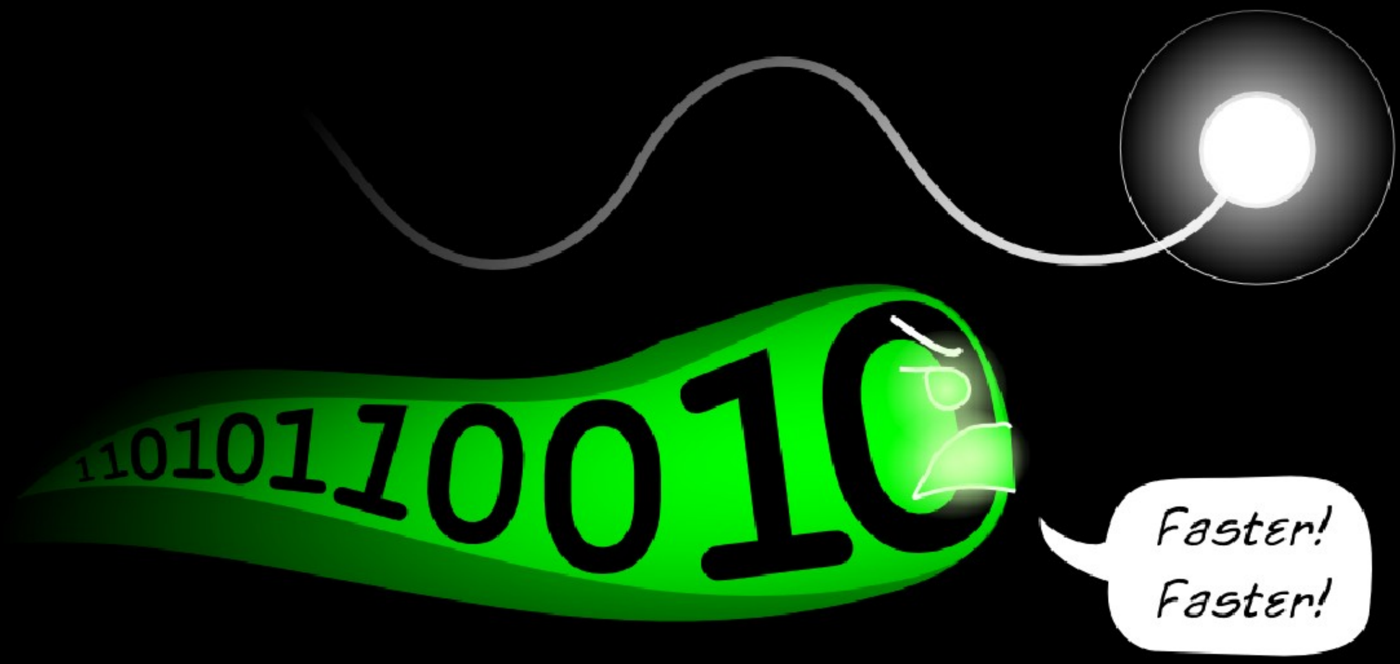
Can't The Hardware Do Better Than This???

HW-Assist Atomic Increment of Global Variable

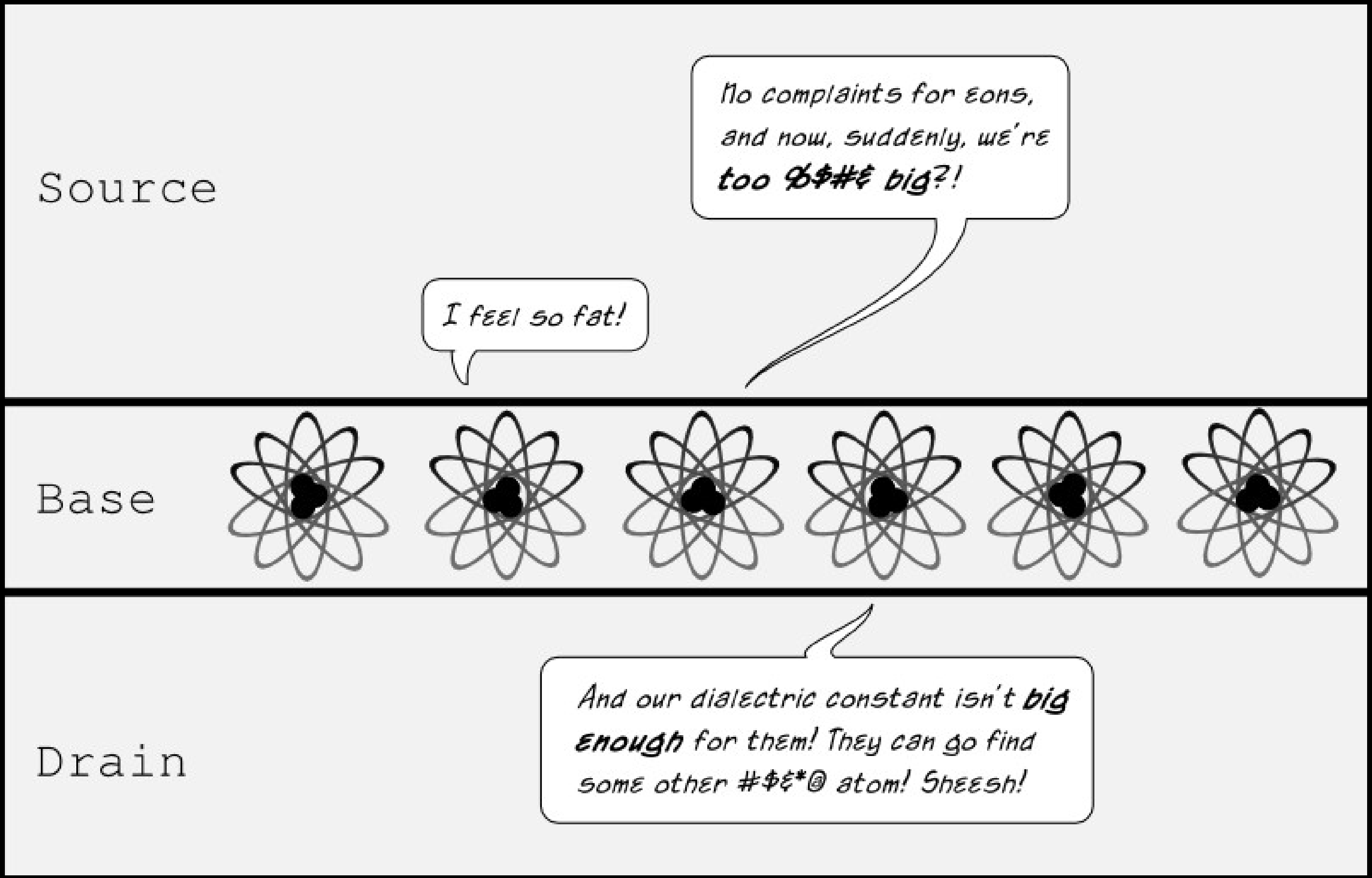


SGI systems used this approach in the 1990s, expect modern CPUs to optimize.
Still not as good as per-CPU counters.

Problem With Physics #1: Finite Speed of Light

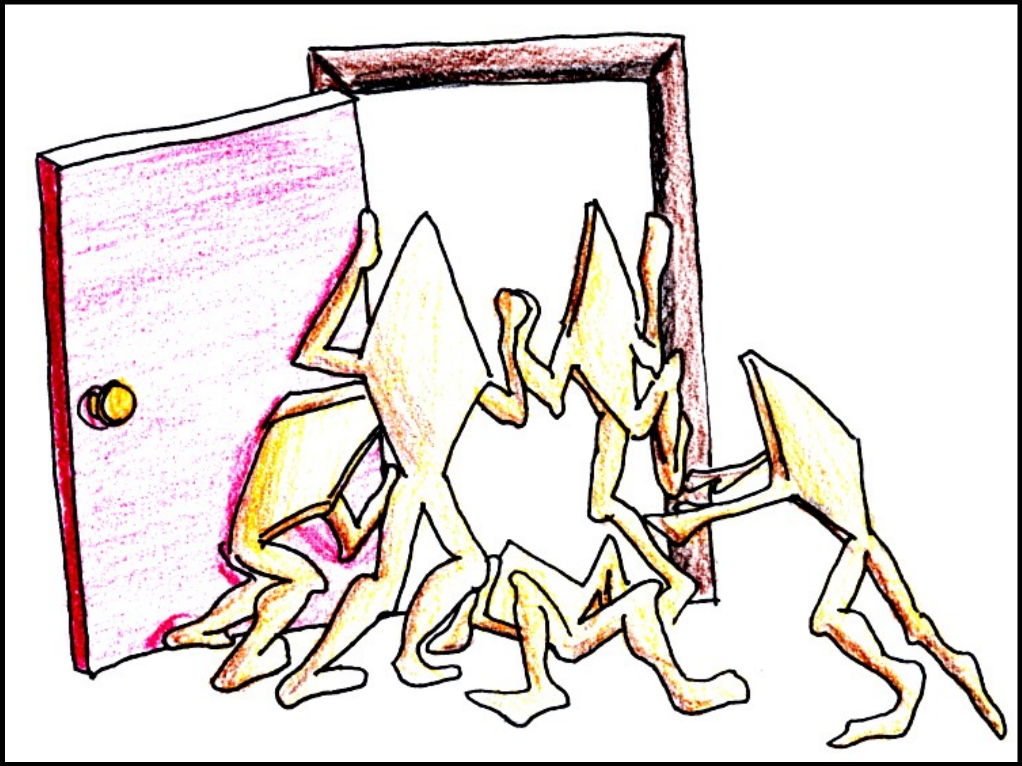


Problem With Physics #2: Atomic Nature of Matter



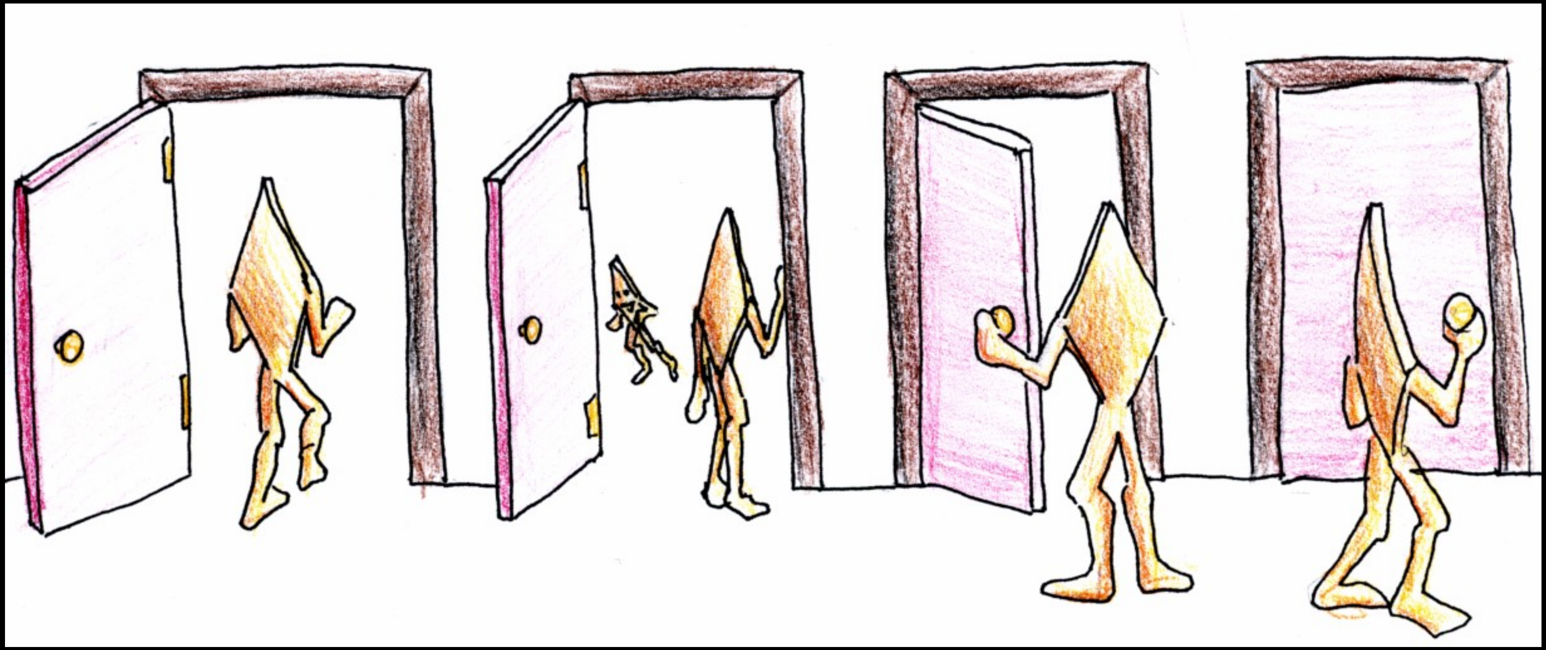
How Can Software Live With This Hardware???

Design Principle: Avoid Bottlenecks



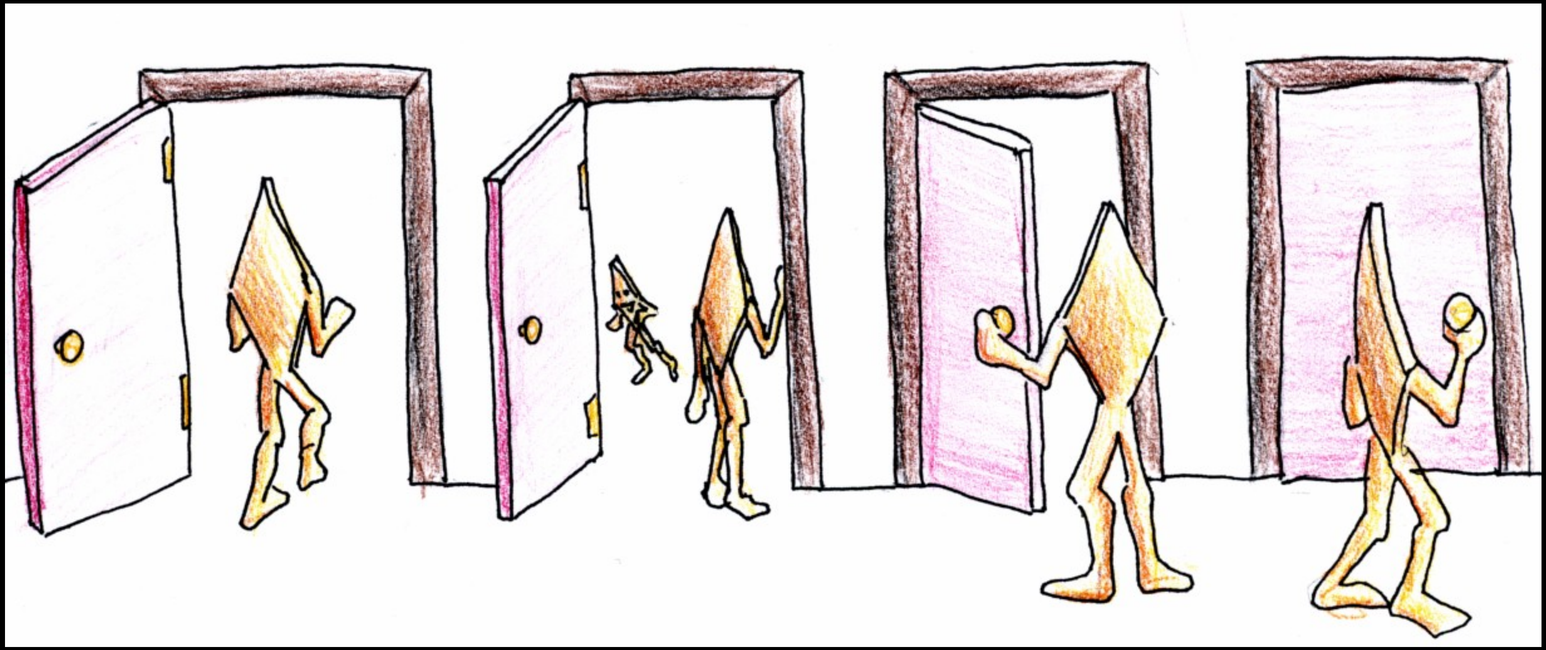
Only one of something: bad for performance and scalability.
Also typically results in high complexity.

Design Principle: Avoid Bottlenecks



Many instances of something good!
Avoiding tightly coupled interactions is an excellent way to avoid bugs.

Design Principle: Avoid Bottlenecks



Many instances of something good!
Avoiding tightly coupled interactions is an excellent way to avoid bugs.
But NUMA effects defeated this for per-bucket locking!!!

Design Principle: Avoid Expensive Operations

Need to be here! (Partitioning/RCU)

16-CPU 2.8GHz Intel X5550 (Nehalem) System

Operation	Cost (ns)	Ratio
Clock period	0.4	1
“Best-case” CAS	12.2	33.8
Best-case lock	25.6	71.2
Single cache miss	12.9	35.8
CAS cache miss	7.0	19.4
Single cache miss **(off-core)**	31.2	86.6
CAS cache miss **(off-core)**	31.2	86.5
Single cache miss **(off-socket)**	92.4	256.7
CAS cache miss **(off-socket)**	95.9	266.4

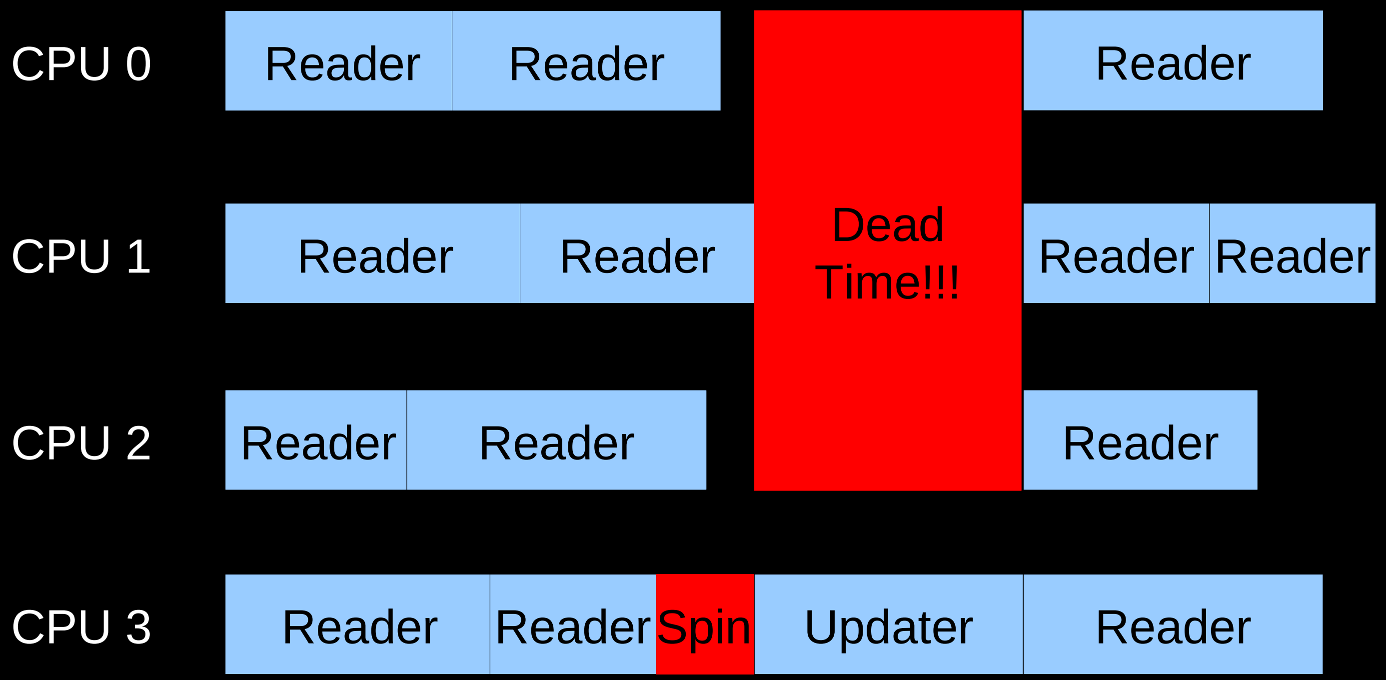
Heavily optimized reader-writer lock might get here for readers (but too bad about those poor writers...)

Typical synchronization mechanisms do this a lot

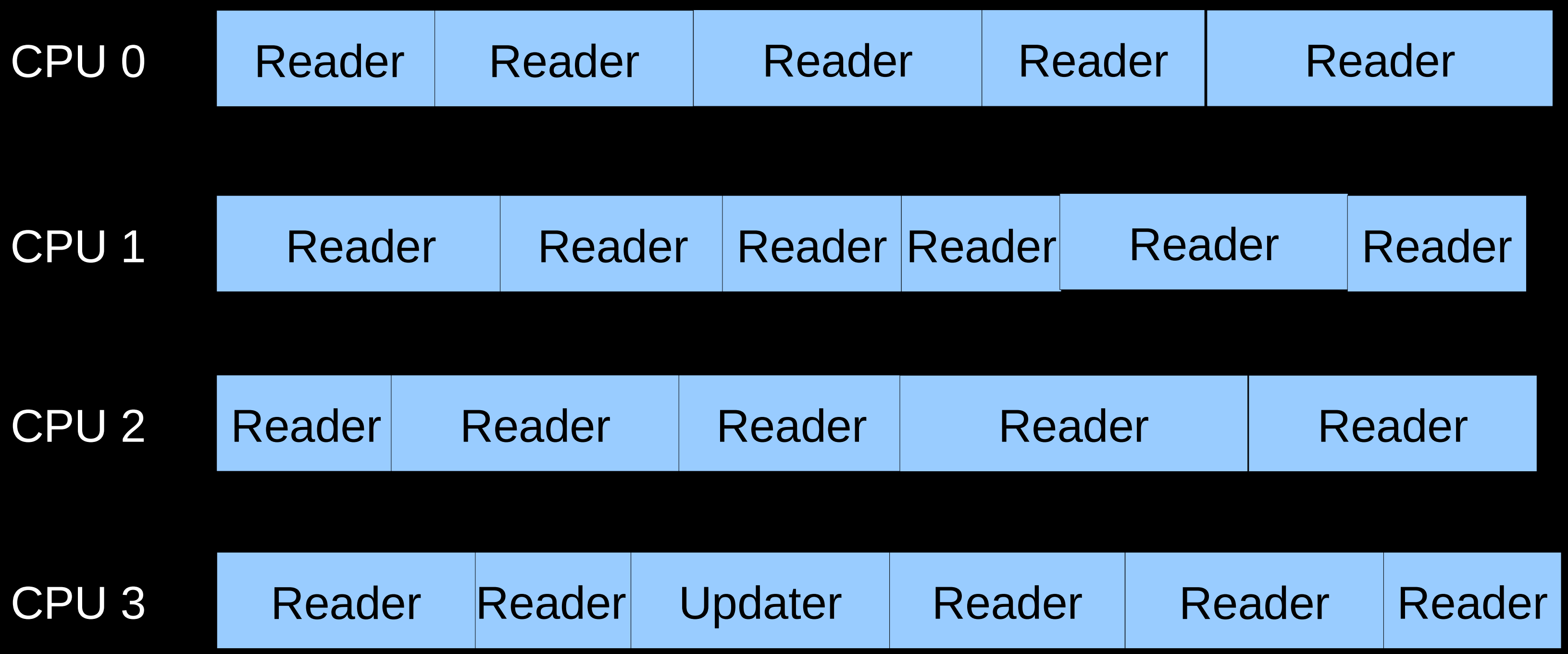
Design Principle: Get Your Money's Worth

- If synchronization is expensive, use large critical sections
- On Nehalem, off-socket CAS costs about 260 cycles
  - So instead of a single-cycle critical section, have a 26000-cycle critical section, reducing synchronization overhead to about 1%
- Of course, we also need to keep contention low, which usually means we want short critical sections
  - Resolve this by applying parallelism at as high a level as possible
  - Parallelize entire applications rather than low-level algorithms!
- This does not work for Schrödinger: The overhead of hash-table operations is too low
  - Which is precisely why we selected hash tables in the first place!!!

Design Principle: Avoid Mutual Exclusion!!!



Design Principle: Avoiding Mutual Exclusion



No Dead Time!

But How Can This Possibly Be Implemented???

Implementing Read-Copy Update (RCU)

- Lightest-weight conceivable read-side primitives

Implementing Read-Copy Update (RCU)

- Lightest-weight conceivable read-side primitives
  - /* Assume non-preemptible (run-to-block) environment. */
  - #define rcu_read_lock()
  - #define rcu_read_unlock()

Implementing Read-Copy Update (RCU)

- Lightest-weight conceivable read-side primitives
  - /* Assume non-preemptible (run-to-block) environment. */
  - #define rcu_read_lock()
  - #define rcu_read_unlock()
- Best possible performance, scalability, real-time response, wait-freedom, and energy efficiency

Implementing Read-Copy Update (RCU)

- Lightest-weight conceivable read-side primitives
  - /* Assume non-preemptible (run-to-block) environment. */
  - #define rcu_read_lock()
  - #define rcu_read_unlock()
- Best possible performance, scalability, real-time response, wait-freedom, and energy efficiency
- How can something that does not affect machine state possibly be used as a synchronization primitive???

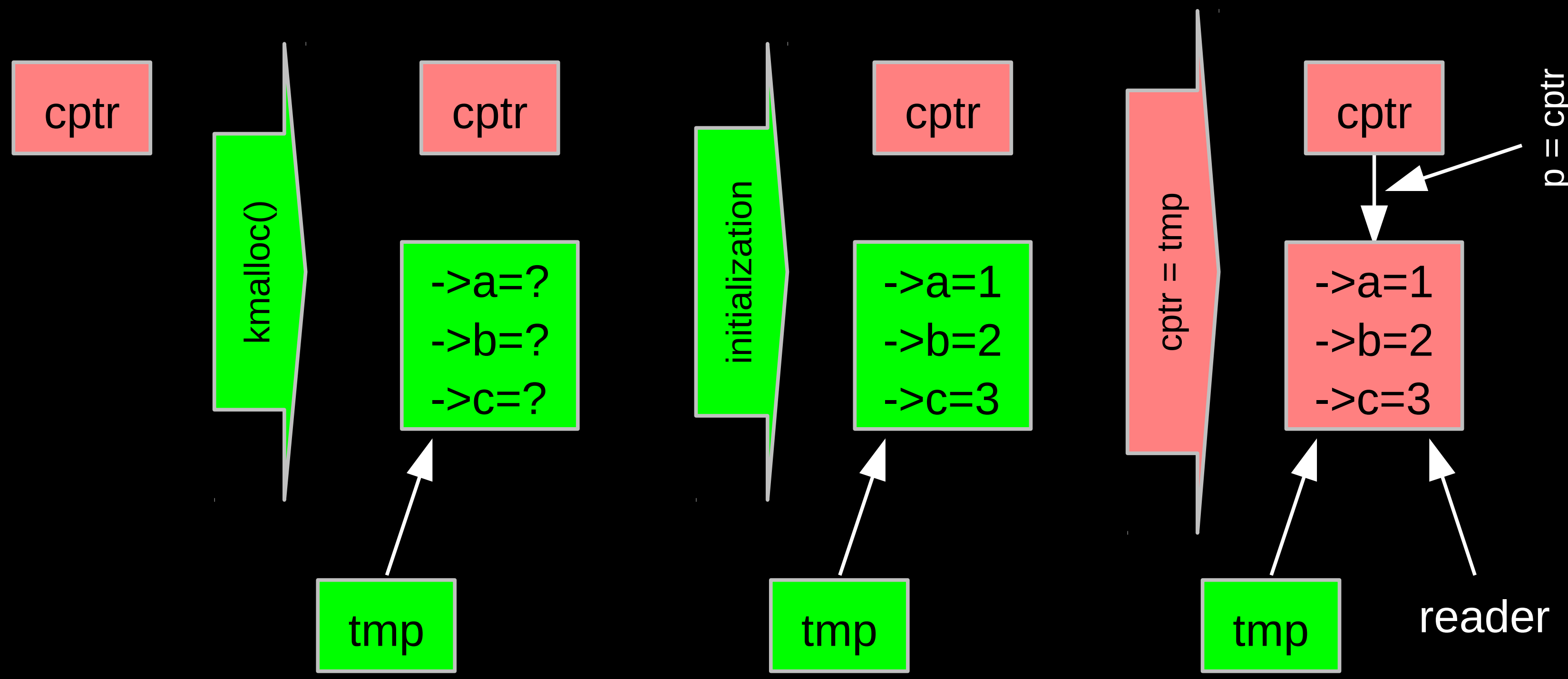
What Is RCU?

- Publishing of new data
- Subscribing to the current version of data
- Waiting for pre-existing RCU readers: Avoid disrupting readers by maintaining multiple versions of the data
  - Each *read*er continues traversing its *copy* of the data while a new *copy* might be being created concurrently by each *update*r *
    - Hence the name *read-copy update*, or RCU
  - Once all pre-existing RCU readers are done with them, old versions of the data may be discarded

* This expansion provided by Jonathan Walpole

Publication of And Subscription to New Data

Key:
- Dangerous for updates: all readers can access
- Still dangerous for updates: pre-existing readers can access (next slide)
- Safe for updates: inaccessible to all readers



Memory Ordering: Mischief From Compiler and CPU

Memory Ordering: Mischief From Compiler and CPU

- Original updater code:

```
p = malloc(sizeof(*p));
p->a = 1;
p->b = 2;
p->c = 3;
cptr = p;
```

- Original reader code:

```
p = cptr;
foo(p->a, p->b, p->c);
```

- Mischievous updater code:

```
p = malloc(sizeof(*p));
cptr = p;
p->a = 1;
p->b = 2;
p->c = 3;
```

- Mischievous reader code:

```
retry:
p = guess(cptr);
foo(p->a, p->b, p->c);
if (p != cptr)
    goto retry;
```

Memory Ordering: Mischief From Compiler and CPU

- Original updater code:

```
p = malloc(sizeof(*p));
p->a = 1;
p->b = 2;
p->c = 3;
cptr = p;
```

- Original reader code:

```
p = cptr;
foo(p->a, p->b, p->c);
```

- Mischievous updater code:

```
p = malloc(sizeof(*p));
cptr = p;
p->a = 1;
p->b = 2;
p->c = 3;
```

- Mischievous reader code:

```
retry:
p = guess(cptr);
foo(p->a, p->b, p->c);
if (p != cptr)
    goto retry;
```

But don't take *my* word for it on HW value speculation:
http://www.openvms.compaq.com/wizard/wiz_2637.html

Preventing Memory-Order Mischief

- Updater uses rcu_assign_pointer() to publish pointer:

```
#define rcu_assign_pointer(p, v) \
({ \
        smp_wmb(); /* SMP Write Memory Barrier */ \
        (p) = (v); \
})
```

- Reader uses rcu_dereference() to subscribe to pointer:

```
#define rcu_dereference(p) \
({ \
        typeof(p) _p1 = (*(volatile typeof(p)*)&(p)); \
        smp_read_barrier_depends(); \
        _p1; \
})
```

- The Linux-kernel definitions are more ornate: Debugging code

Preventing Memory-Order Mischief

- "Memory-order-mischief proof" updater code:

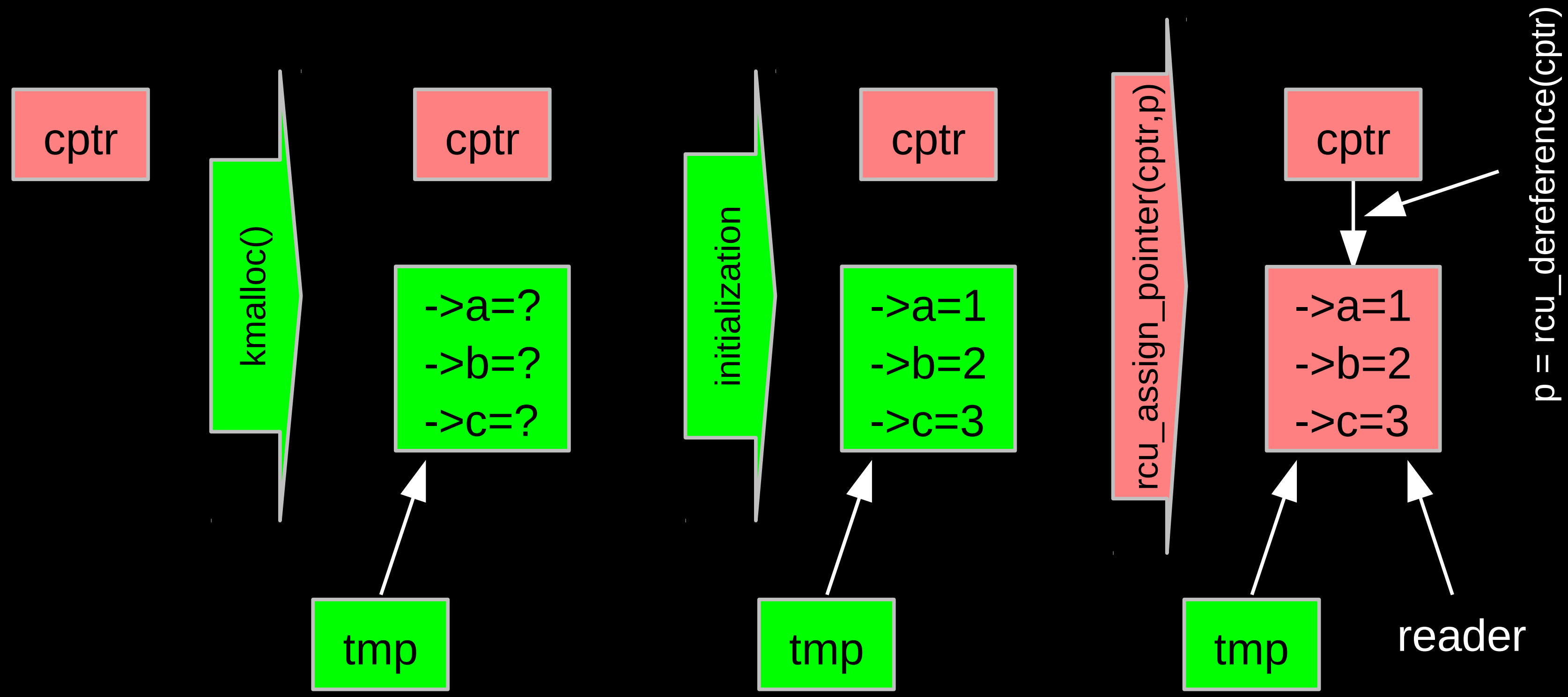
```
p = malloc(sizeof(*p));
p->a = 1;
p->b = 2;
p->c = 3;
rcu_assign_pointer(cptr, p);
```

- "Memory-order-mischief proof" reader code:

```
p = rcu_dereference(cptr);
foo(p->a, p->b, p->c);
```

Publication of And Subscription to New Data

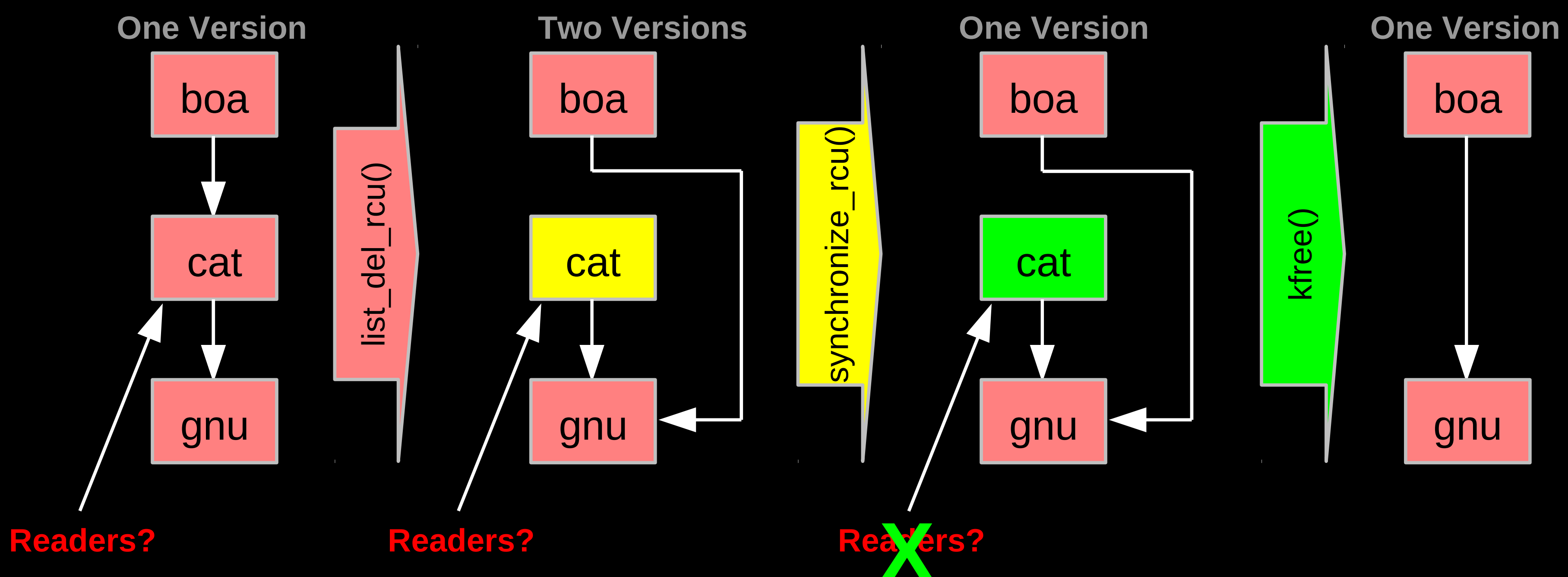
Key:
- (red) Dangerous for updates: all readers can access
- (yellow) Still dangerous for updates: pre-existing readers can access (next slide)
- (green) Safe for updates: inaccessible to all readers



But if all we do is add, we have a big memory leak!!!

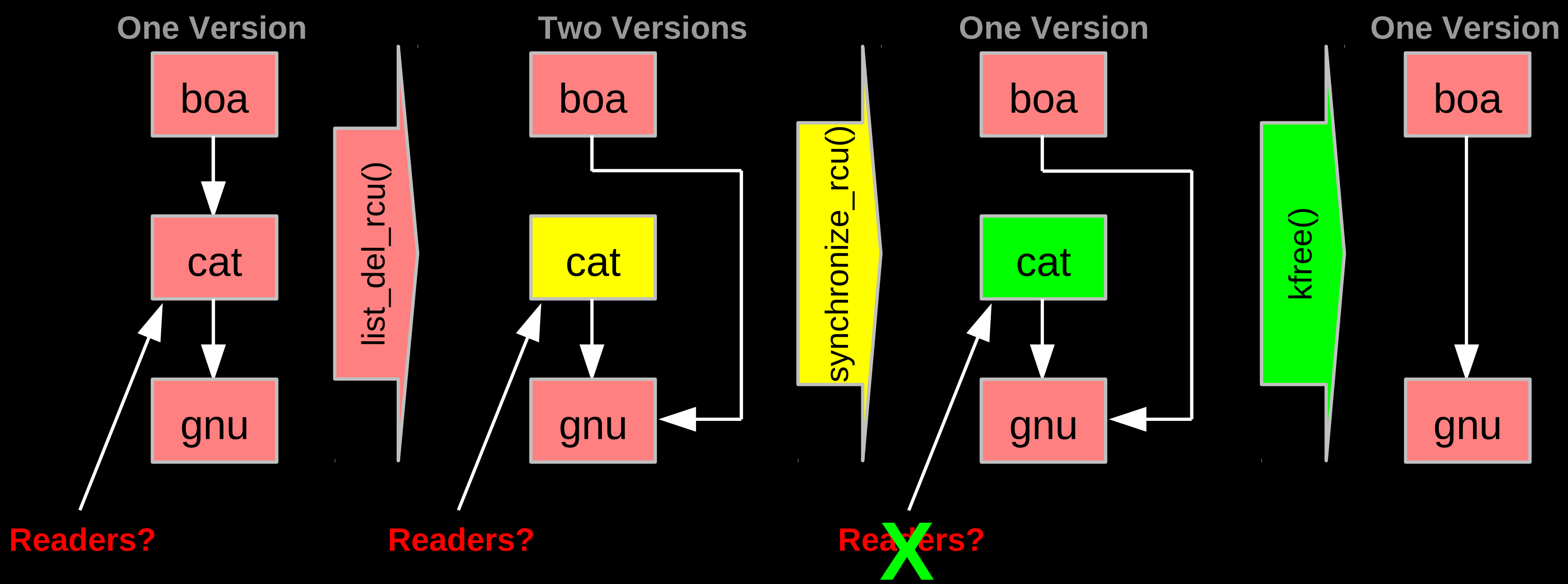
RCU Removal From Linked List

- Combines waiting for readers and multiple versions:
  – Writer removes the cat's element from the list (list_del_rcu())
  – Writer waits for all readers to finish (synchronize_rcu())
  – Writer can then free the cat's element (kfree())



RCU Removal From Linked List

- Combines waiting for readers and multiple versions:
  - Writer removes the cat's element from the list (list_del_rcu())
  - Writer waits for all readers to finish (synchronize_rcu())
  - Writer can then free the cat's element (kfree())

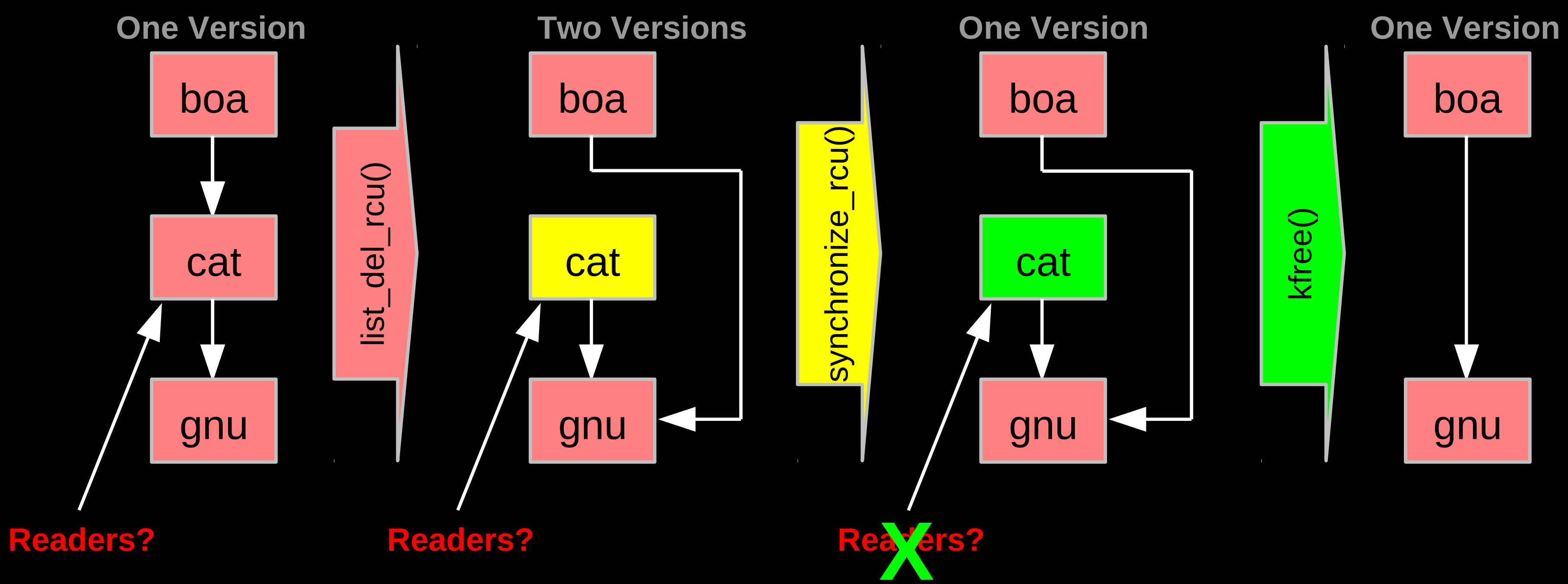


But how can software deal with two different versions simultaneously???

Two Different Versions Simultaneously???

RCU Removal From Linked List

- Combines waiting for readers and multiple versions:
  - Writer removes the cat's element from the list (list_del_rcu())
  - Writer waits for all readers to finish (synchronize_rcu())
  - Writer can then free the cat's element (kfree())



But if readers leave no trace in memory, how can we possibly tell when they are done???

How Can RCU Tell When Readers Are Done???

How Can RCU Tell When Readers Are Done???

That is, without re-introducing all of the overhead and latency inherent to other synchronization mechanisms...

But First, Some RCU Nomenclature

- *RCU read-side critical section*
  - Begins with rcu_read_lock(), ends with rcu_read_unlock(), and may contain rcu_dereference()
- *Quiescent state*
  - Any code that is not in an RCU read-side critical section
- *Extended quiescent state*
  - Quiescent state that persists for a significant time period
- *RCU grace period*
  - Time period when every thread was in at least one quiescent state

But First, Some RCU Nomenclature

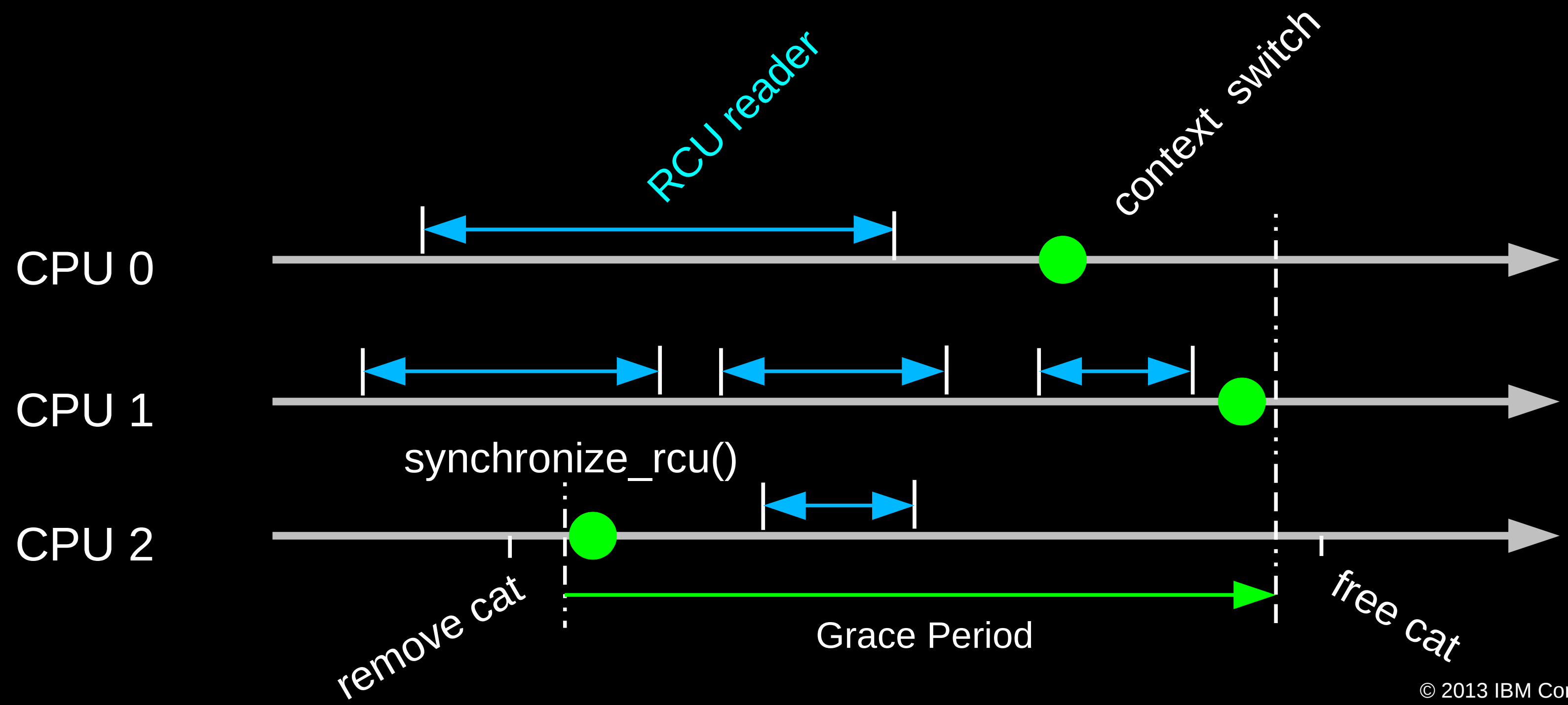
- *RCU read-side critical section*
  - Begins with rcu_read_lock(), ends with rcu_read_unlock(), and may contain rcu_dereference()
- *Quiescent state*
  - Any code that is not in an RCU read-side critical section
- *Extended quiescent state*
  - Quiescent state that persists for a significant time period
- *RCU grace period*
  - Time period when every thread was in at least one quiescent state
- OK, names are nice, but how can you possibly implement this???

Waiting for Pre-Existing Readers: QSBR

- Non-preemptive environment (CONFIG_PREEMPT=n)
  - RCU readers are not permitted to block
  - Same rule as for tasks holding spinlocks

Waiting for Pre-Existing Readers: QSBR

- Non-preemptive environment (CONFIG_PREEMPT=n)
  – RCU readers are not permitted to block
  – Same rule as for tasks holding spinlocks
- CPU context switch means all that CPU's readers are done
- *Grace period* ends after all CPUs execute a context switch



Synchronization Without Changing Machine State???

- But rcu_read_lock() does not need to change machine state
  - Instead, it acts on the developer, who must avoid blocking within RCU read-side critical sections
  - Or, more generally, avoid quiescent states within RCU read-side critical sections

Synchronization Without Changing Machine State???

- But rcu_read_lock() does not need to change machine state
  - Instead, it acts on the developer, who must avoid blocking within RCU read-side critical sections
  - Or, more generally, avoid quiescent states within RCU read-side critical sections
- RCU is therefore synchronization via social engineering

Synchronization Without Changing Machine State???

- But rcu_read_lock() does not need to change machine state
  - Instead, it acts on the developer, who must avoid blocking within RCU read-side critical sections
  - Or, more generally, avoid quiescent states within RCU read-side critical sections
- RCU is therefore synchronization via social engineering
- As do all other synchronization mechanisms:
  - “Avoid data races”
  - “Protect specified variables with the corresponding lock”
  - “Access shared variables only within transactions”

Toy Implementation of RCU: 20 Lines of Code

- Read-side primitives:

```
#define rcu_read_lock()
#define rcu_read_unlock()
#define rcu_dereference(p) \
({ \
        typeof(p) _p1 = (*(volatile typeof(p)*)&(p)); \
        smp_read_barrier_depends(); \
        _p1; \
})
```

- Update-side primitives

```
#define rcu_assign_pointer(p, v) \
({ \
        smp_wmb(); \
        (p) = (v); \
})
void synchronize_rcu(void)
{
        int cpu;

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

Toy Implementation of RCU: 20 Lines of Code, Full Read-Side Performance!!!

- Read-side primitives:

```
#define rcu_read_lock()
#define rcu_read_unlock()
#define rcu_dereference(p) \
({ \
        typeof(p) _p1 = (*(volatile typeof(p)*)&(p)); \
        smp_read_barrier_depends(); \
        _p1; \
})
```

- Update-side primitives

```
#define rcu_assign_pointer(p, v) \
({ \
        smp_wmb(); \
        (p) = (v); \
})
void synchronize_rcu(void)
{
        int cpu;

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

Only 9 of which are needed on sequentially consistent systems...
And some people still insist that RCU is complicated... ;-)

RCU Usage: Readers

- Pointer to RCU-protected object guaranteed to exist throughout RCU read-side critical section

```
rcu_read_lock(); /* Start critical section. */
p = rcu_dereference(cptr);
/* *p guaranteed to exist. */
do_something_with(p);
rcu_read_unlock(); /* End critical section. */
/* *p might be freed!!! */
```

- The rcu_read_lock(), rcu_dereference() and rcu_read_unlock() primitives are very light weight
- However, updaters must take care...

RCU Usage: Updaters

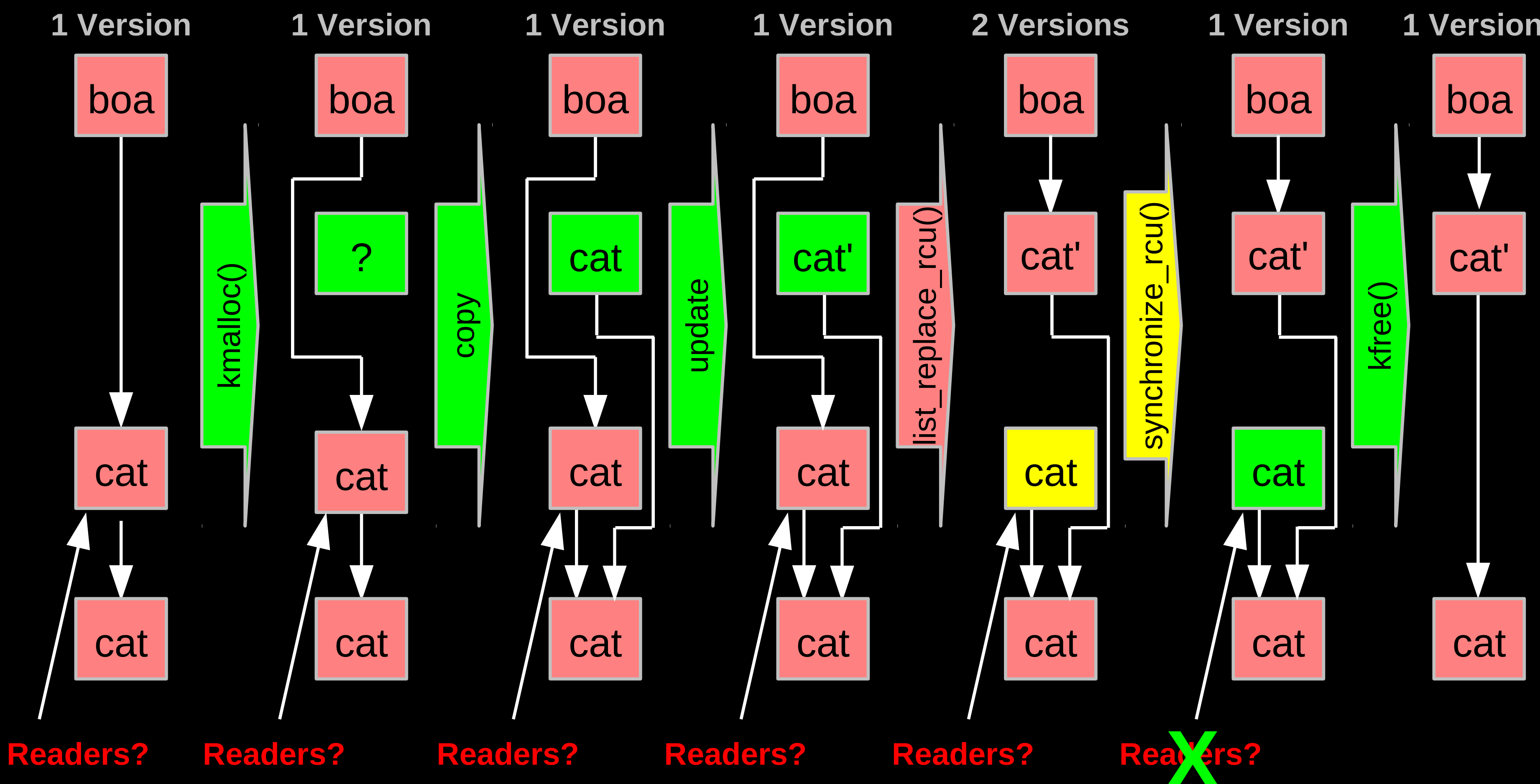
- Updaters must wait for an *RCU grace period* to elapse between making something inaccessible to readers and freeing it

```
spin_lock(&updater_lock);
q = cptr;
rcu_assign_pointer(cptr, new_p);
spin_unlock(&updater_lock);
synchronize_rcu(); /* Wait for grace period. */
kfree(q);
```

- RCU grace period waits for all pre-exiting readers to complete their RCU read-side critical sections

Complex Atomic-To-Reader Updates

RCU Replacement Of Item In Linked List

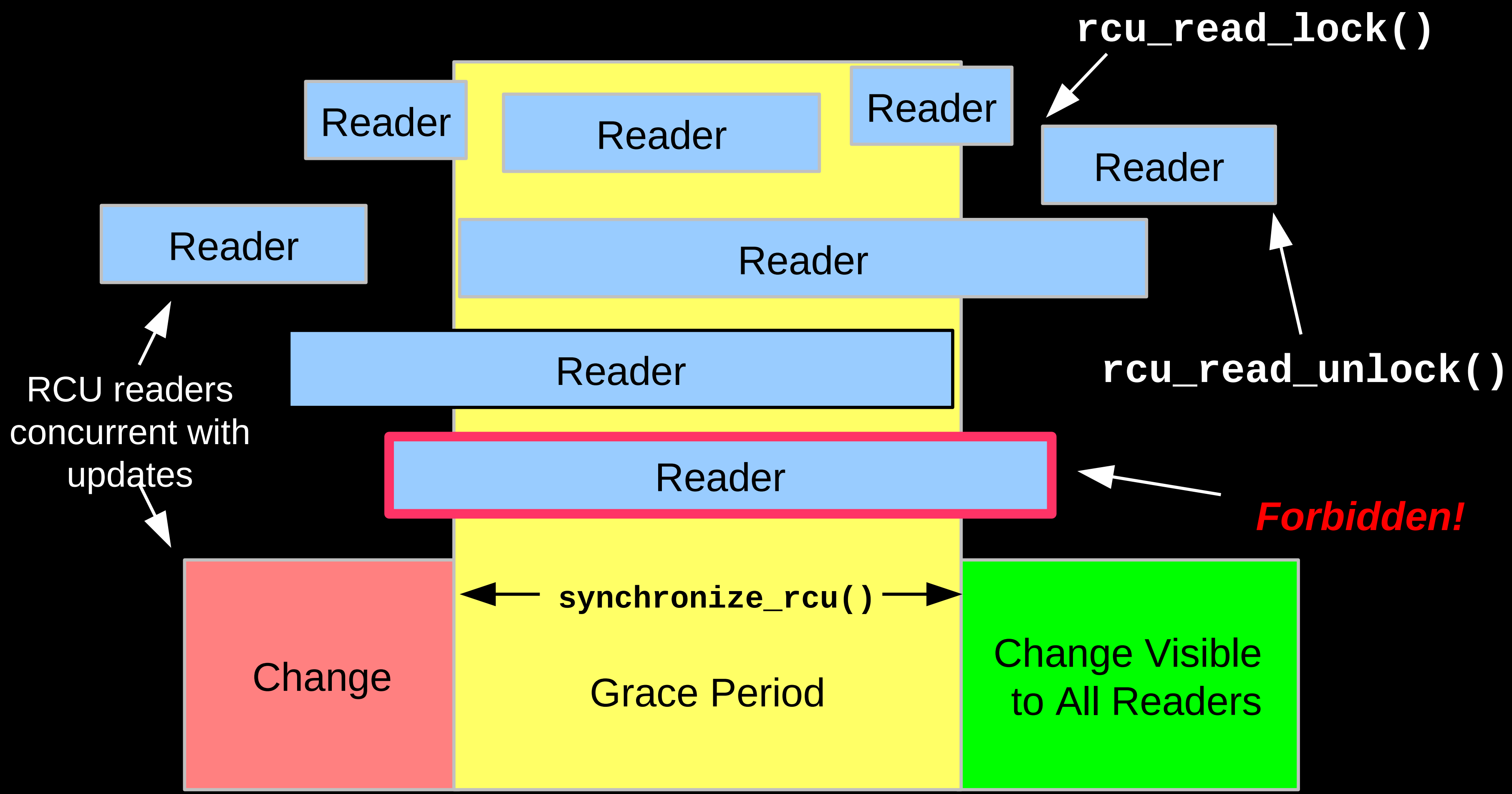


RCU Grace Periods: Conceptual and Graphical Views

RCU Grace Periods: A Conceptual View

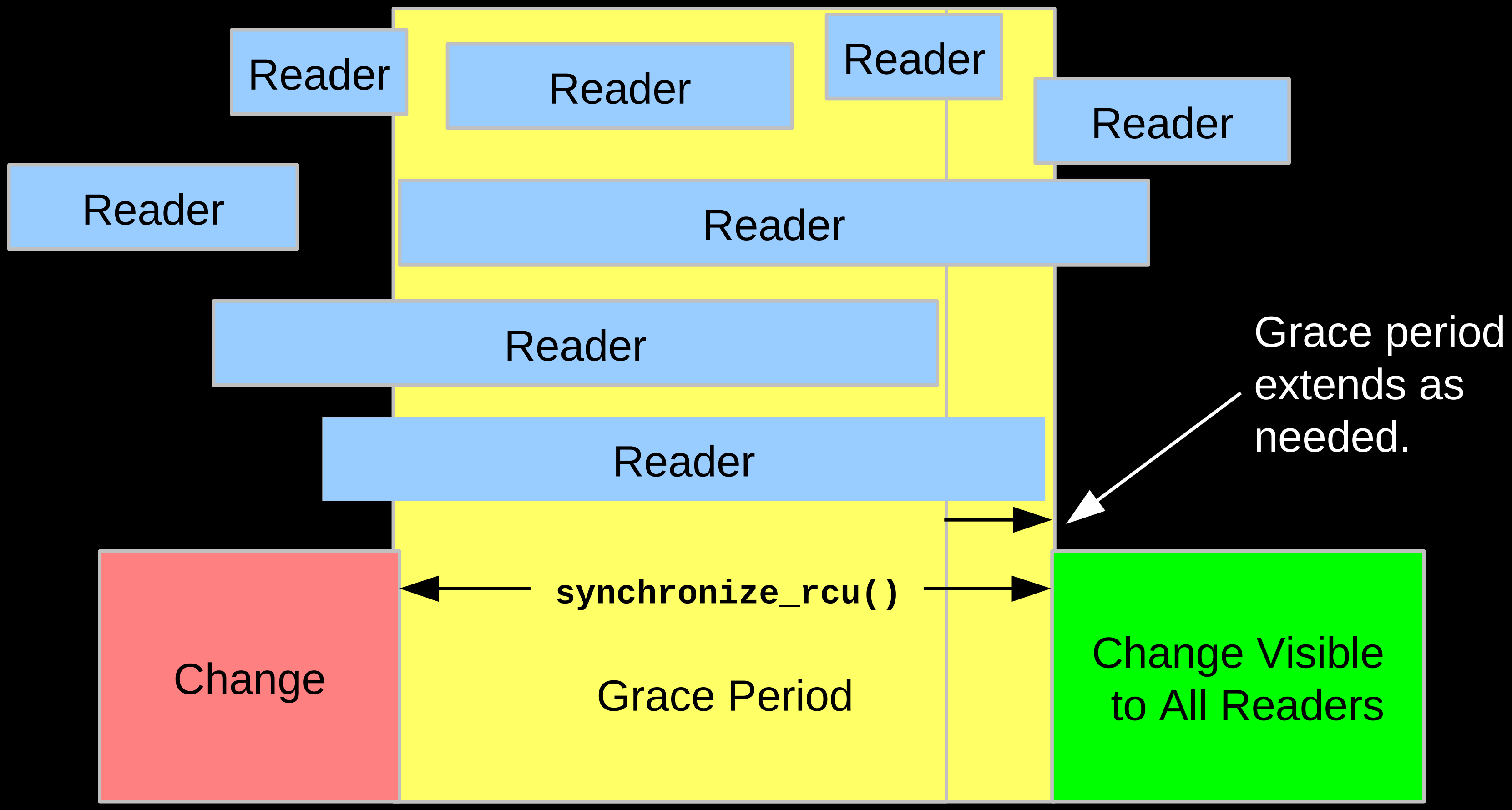
- *RCU read-side critical section*
  - Begins with rcu_read_lock(), ends with rcu_read_unlock(), and may contain rcu_dereference()
- *Quiescent state*
  - Any code that is not in an RCU read-side critical section
- *Extended quiescent state*
  - Quiescent state that persists for a significant time period
- *RCU grace period*
  - Time period when every thread is in at least one quiescent state
  - Ends when all pre-existing readers complete
  - Guaranteed to complete in finite time iff all RCU read-side critical sections are of finite duration
- But what happens if you try to extend an RCU read-side critical section across a grace period?

RCU Grace Periods: A Graphical View



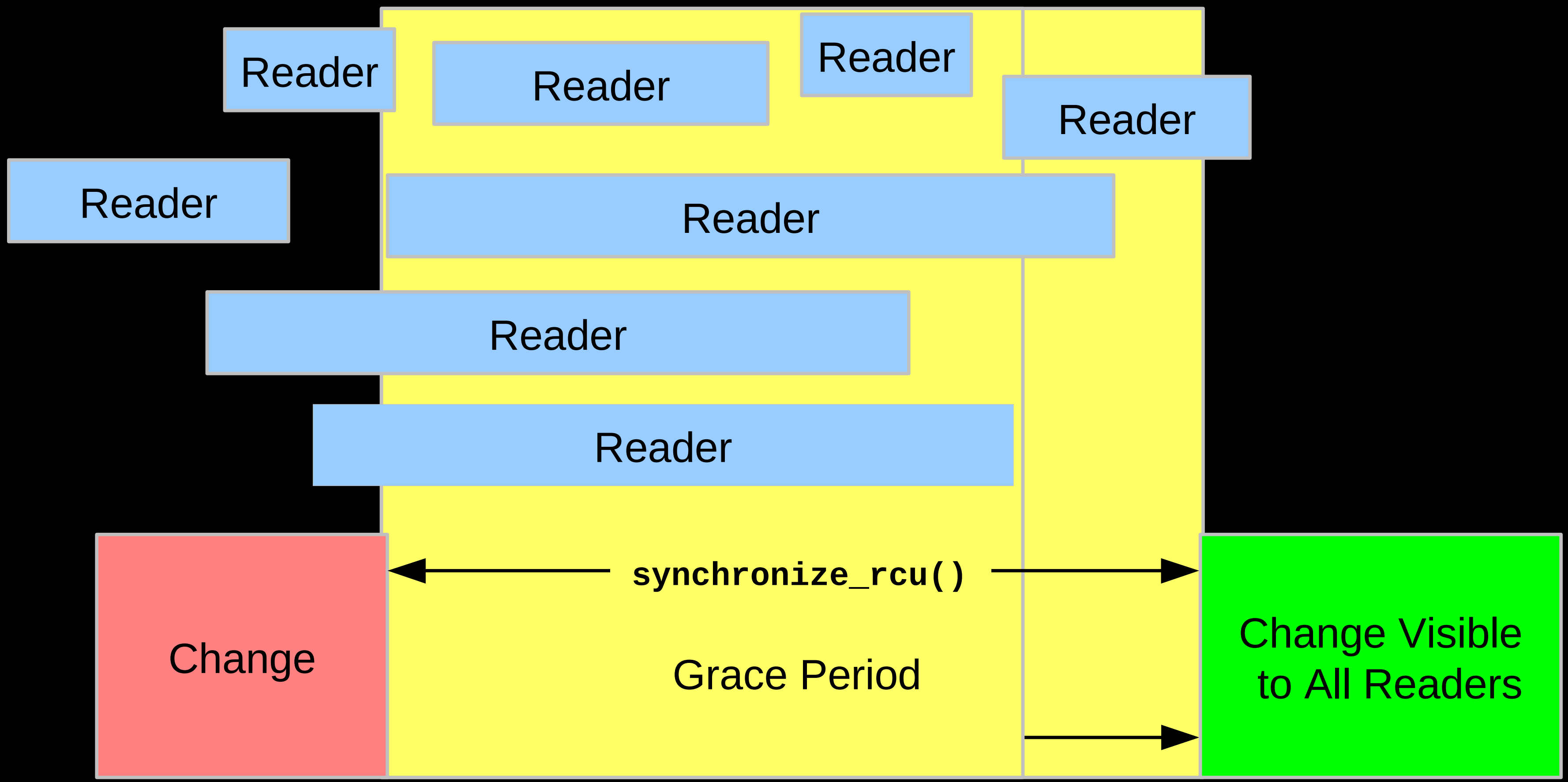
So what happens if you try to extend an RCU read-side critical section across a grace period?

RCU Grace Period: A Self-Repairing Graphical View



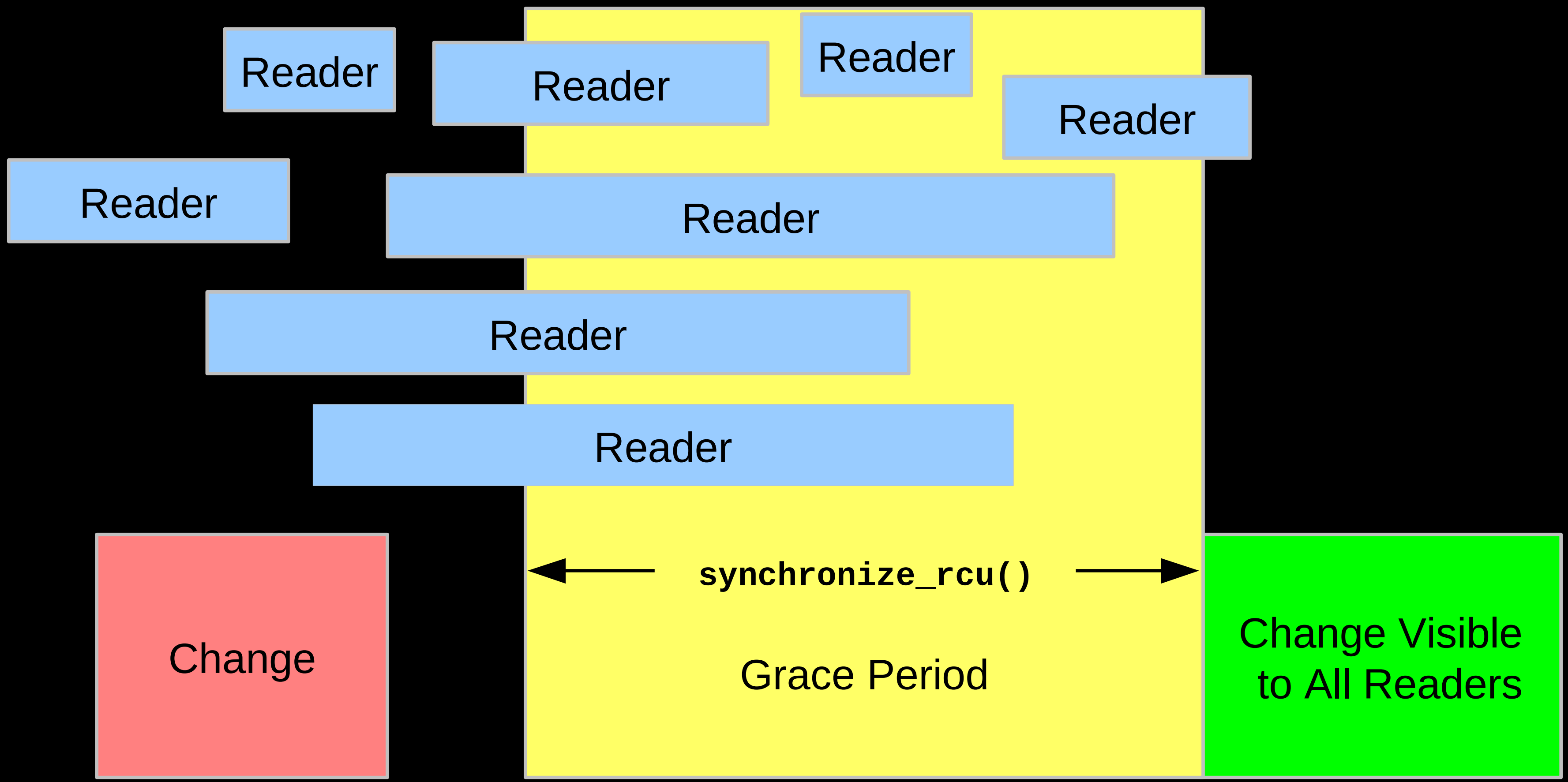
A grace period is not permitted to end until all pre-existing readers have completed.

RCU Grace Period: A Lazy Graphical View



But it is OK for RCU to be lazy and allow a grace period to extend longer than necessary

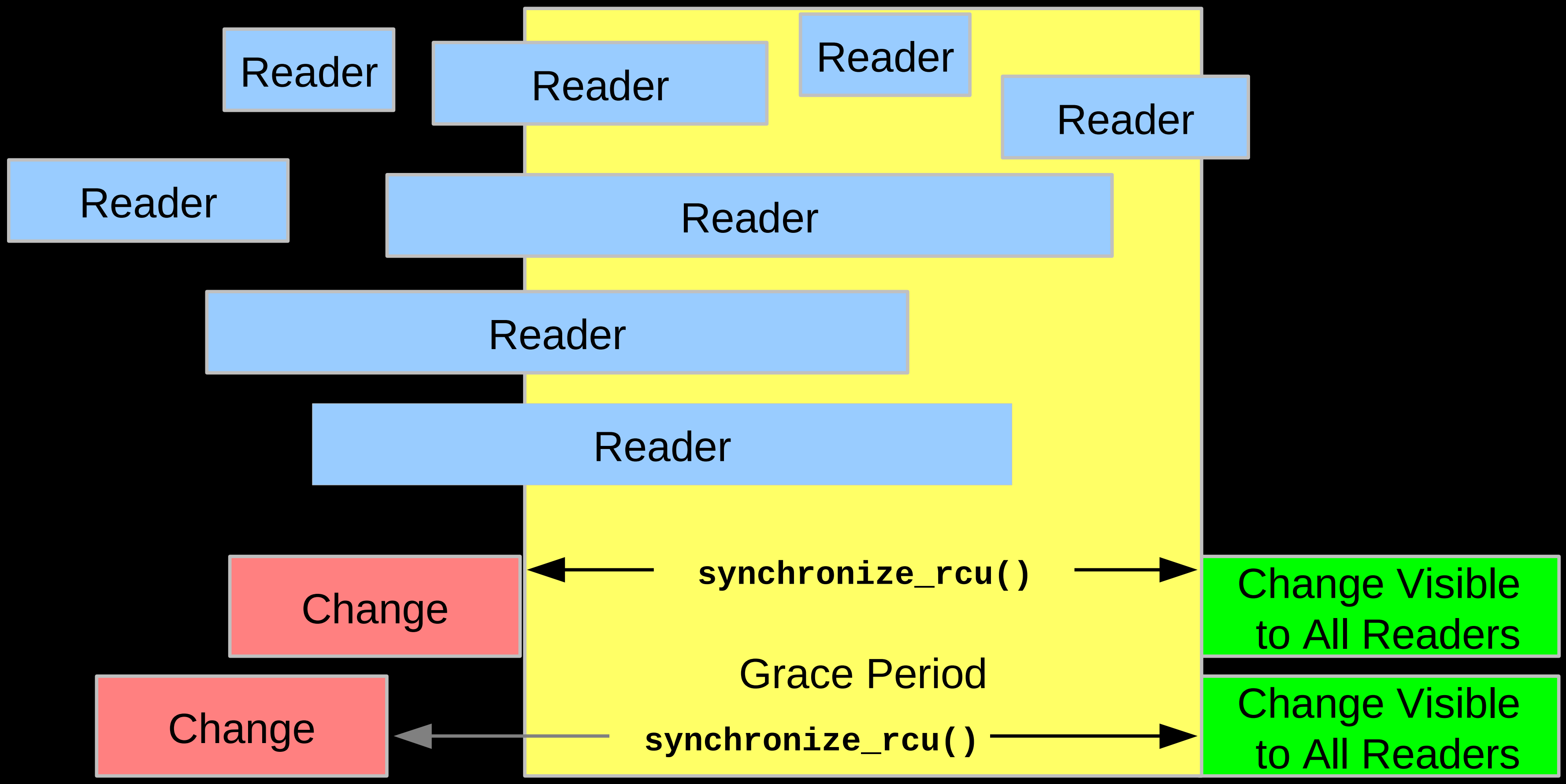
RCU Grace Period: A *Really* Lazy Graphical View



And it is also OK for RCU to be even more lazy and start a grace period later than necessary
But why is this useful?

IBM

RCU Grace Period: A Usefully Lazy Graphical View

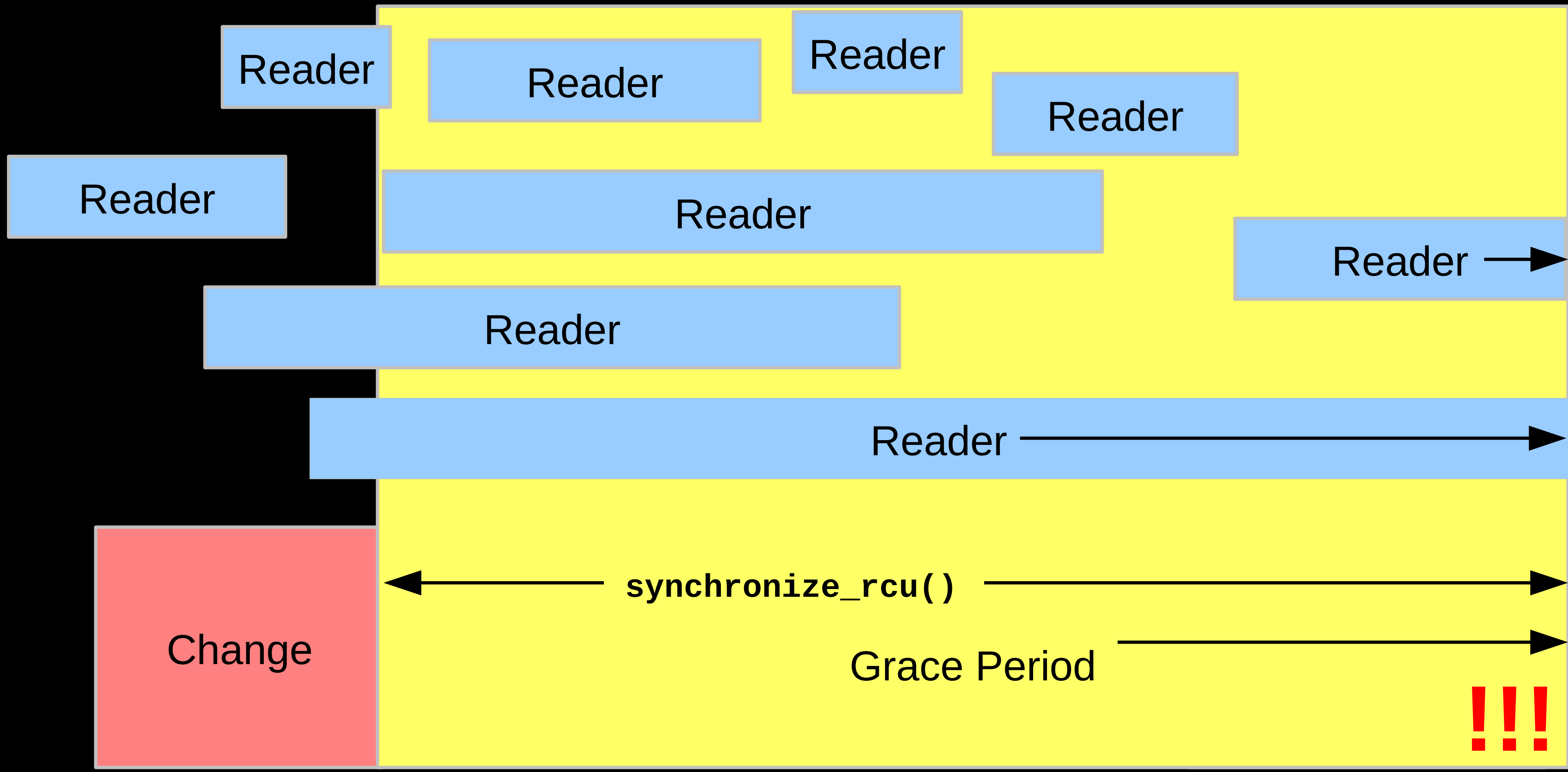


Starting a grace period late can allow it to serve multiple updates, decreasing the per-update RCU overhead. But...

The Costs and Benefits of Laziness

- Starting the grace period later increases the number of updates per grace period, reducing the per-update overhead
- Delaying the end of the grace period increases grace-period latency
- Increasing the number of updates per grace period increases the memory usage
  - Therefore, starting grace periods late is a good tradeoff if memory is cheap and communication is expense, as is the case in modern multicore systems
    - And if real-time threads avoid waiting for grace periods to complete
  - However...

RCU Grace Period: A Too-Lazy Graphical View



And it is OK for the system to complain (or even abort) if a grace period extends too long. Too-long of grace periods are likely to result in death by memory exhaustion anyway.

RCU Asynchronous Grace-Period Detection

RCU Asynchronous Grace-Period Detection

- The call_rcu() function registers an RCU callback, which is invoked after a subsequent grace period elapses
- API:

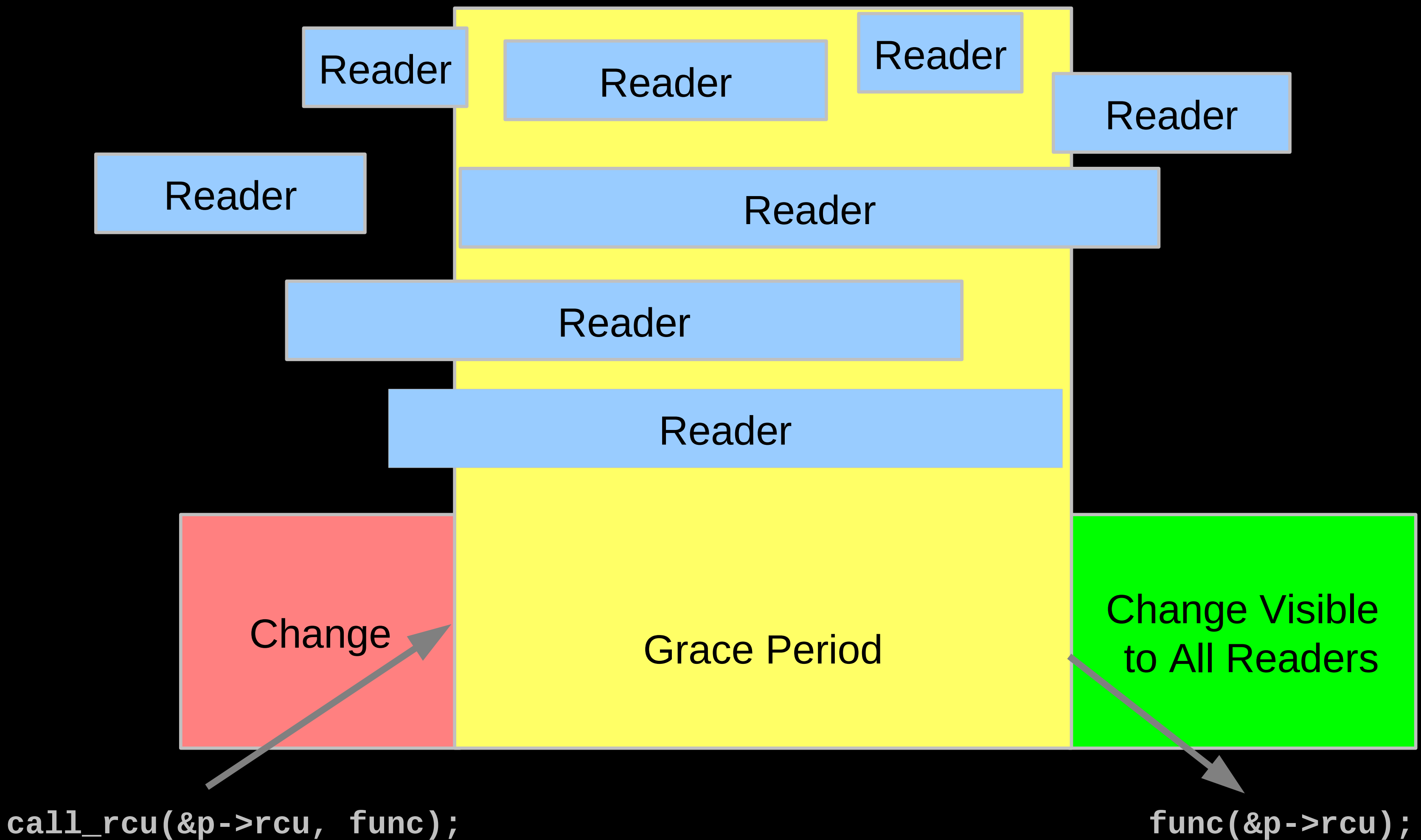
```
call_rcu(struct rcu_head head,
         void (*func)(struct rcu_head *rcu));
```

- The `rcu_head` structure:

```
struct rcu_head {
        struct rcu_head *next;
        void (*func)(struct rcu_head *rcu);
};
```

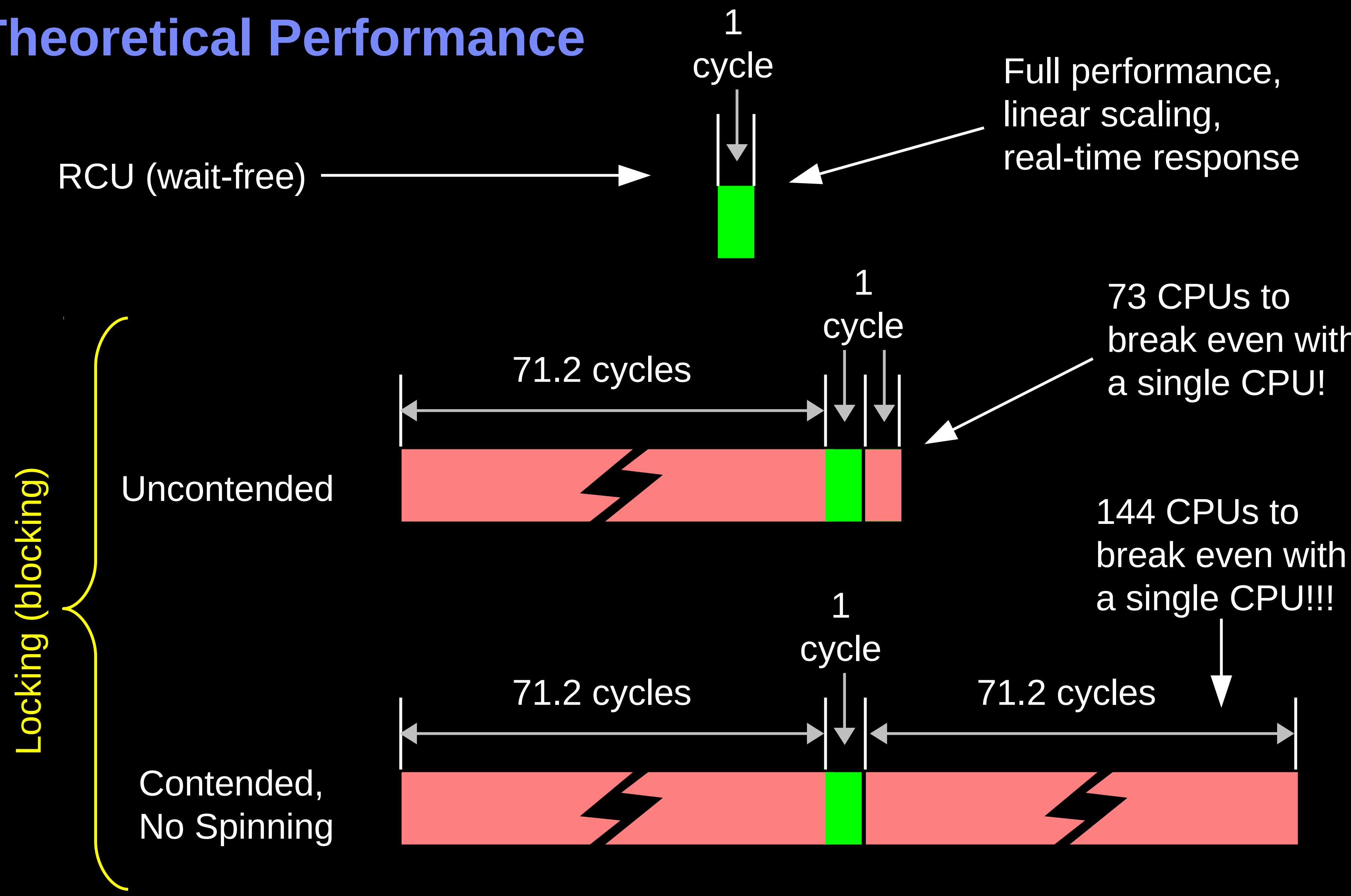
- The `rcu_head` structure is normally embedded within the RCU-protected data structure

RCU Grace Period: An Asynchronous Graphical View



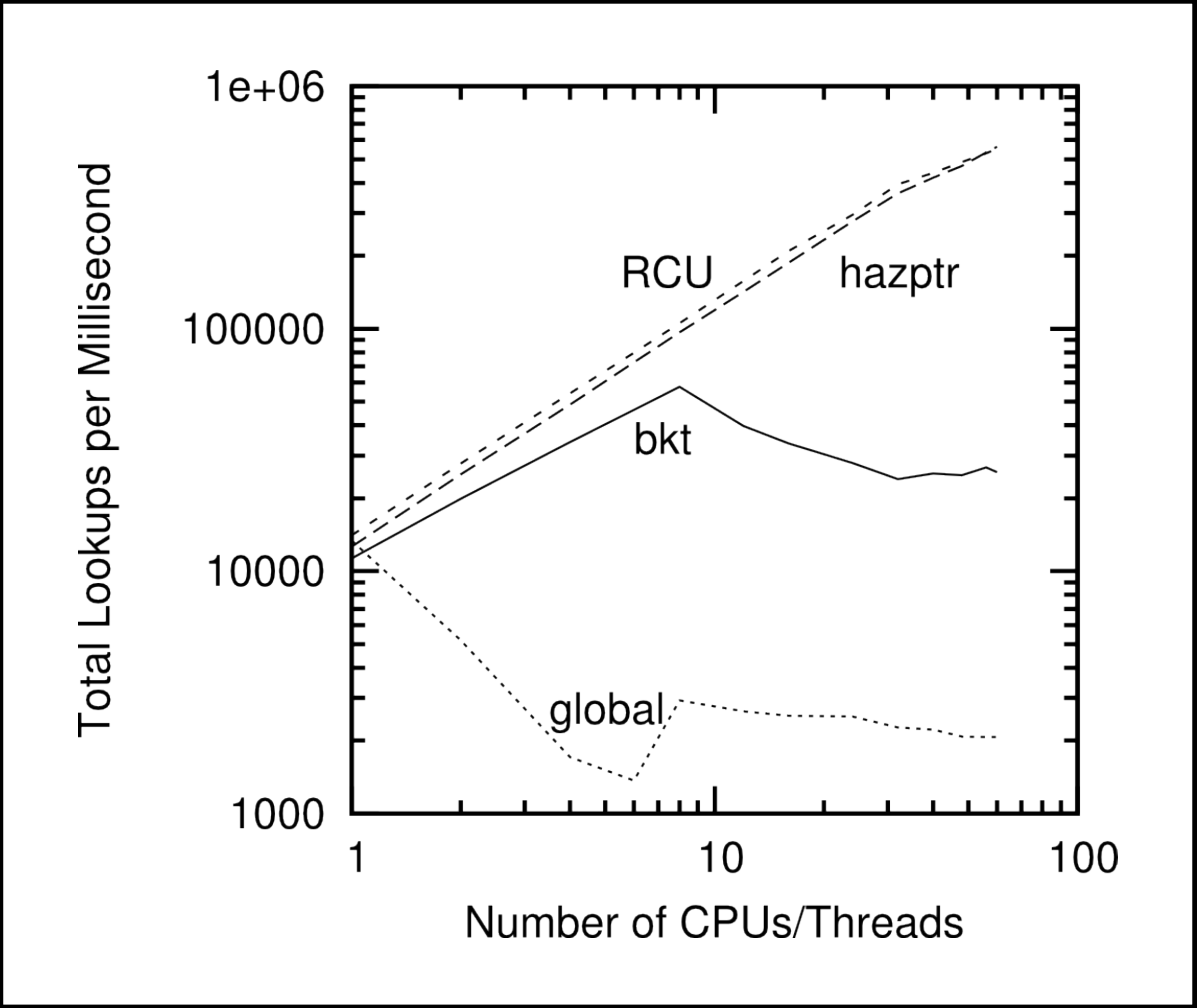
Performance

Theoretical Performance



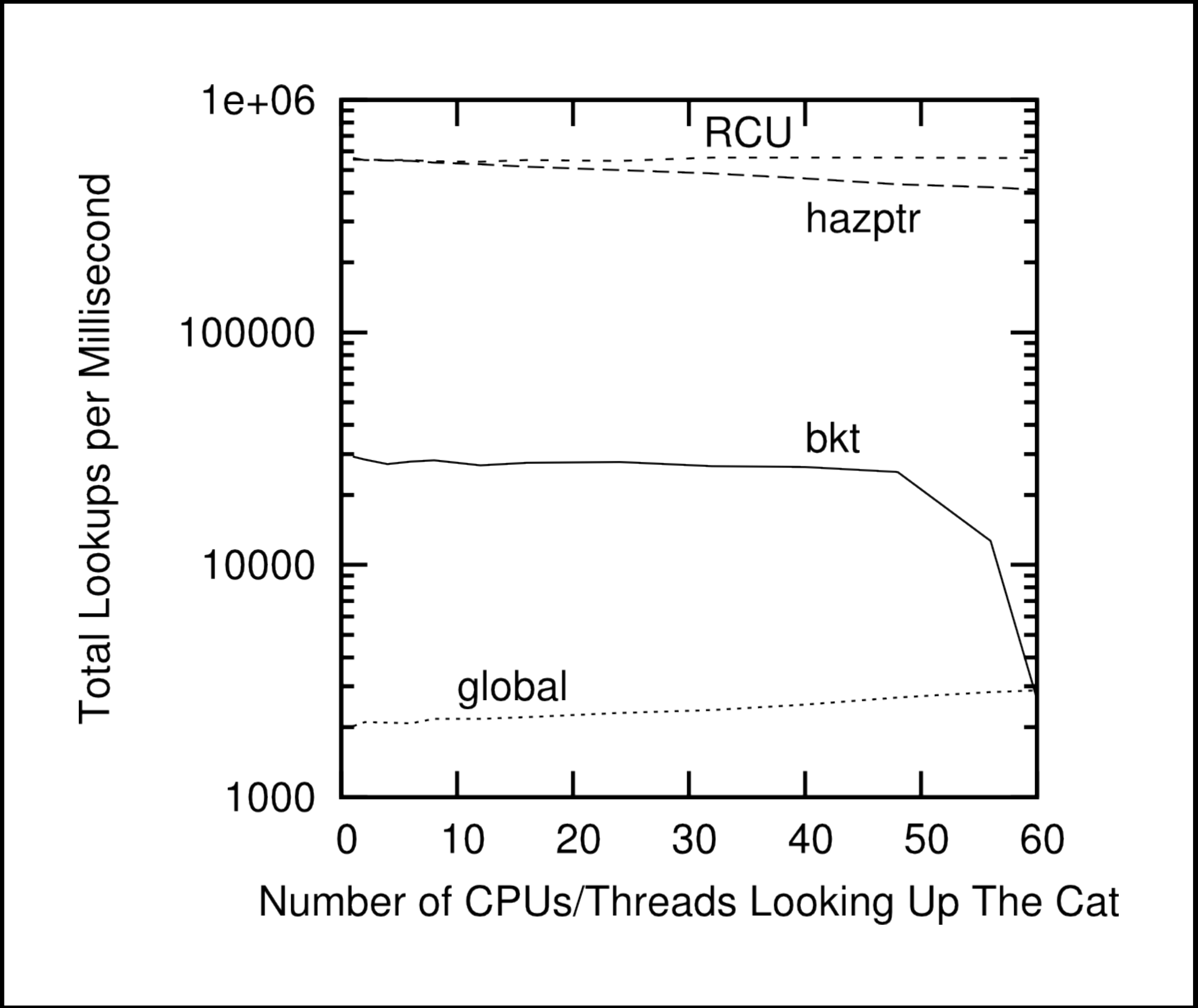
Measured Performance

Schrödinger's Zoo: Read-Only



RCU and hazard pointers scale quite well!!!

Schrödinger's Zoo: Read-Only Cat-Heavy Workload



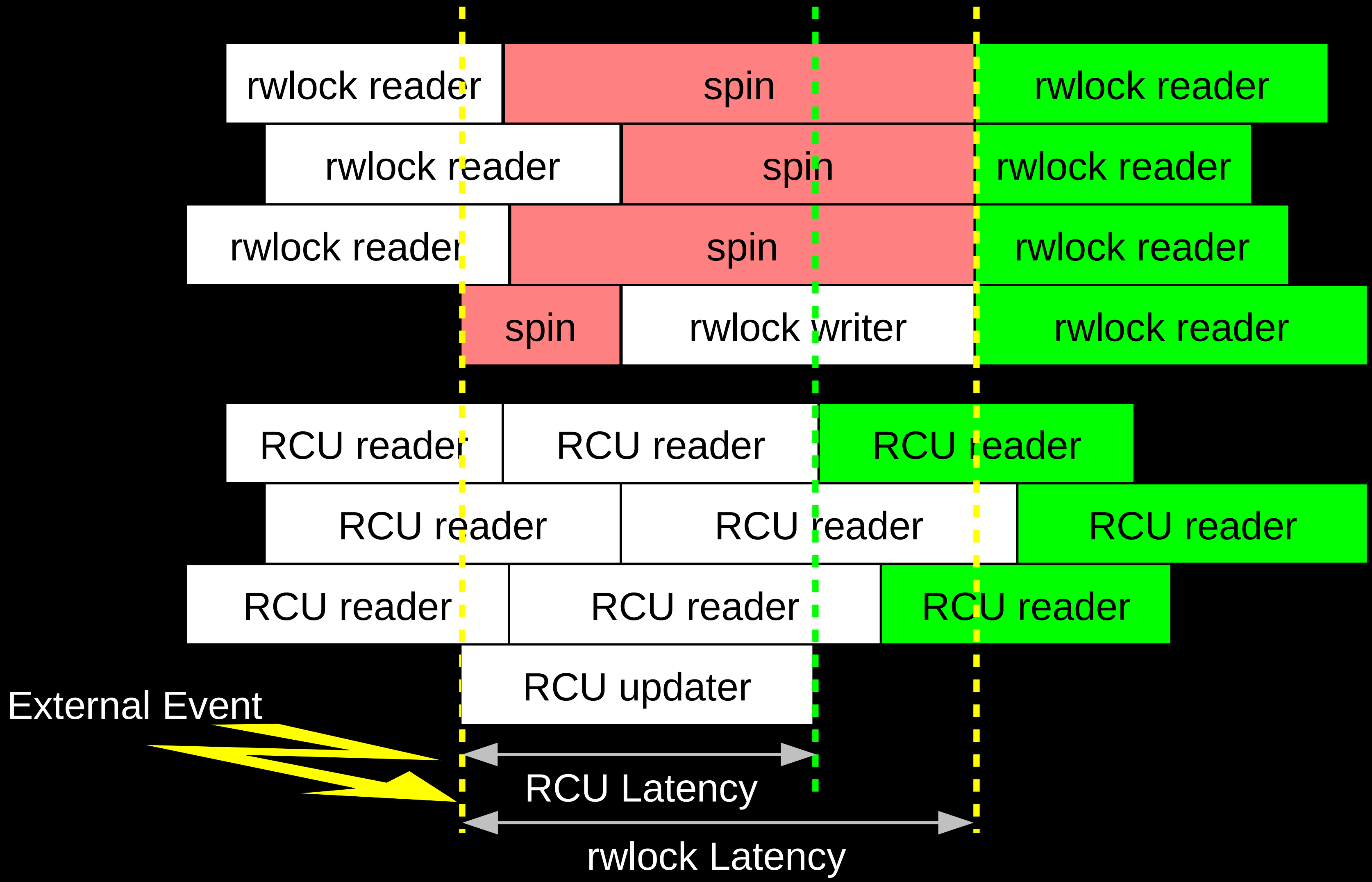
RCU handles locality quite well, hazard pointers not bad, bucket locking horribly

Schrödinger's Zoo: Reads and Updates

Mechanism	Reads	Failed Reads	Cat Reads	Adds	Deletes
Global Locking	799	80	639	77	77
Per-Bucket Locking	13,555	6,177	1,197	5,370	5,370
Hazard Pointers	41,011	6,982	27,059	4,860	4,860
RCU	85,906	13,022	59,873	2,440	2,440

Real-Time Response to Changes

RCU vs. Reader-Writer-Lock Real-Time Latency



RCU Performance: “Free is a *Very* Good Price!!!”

RCU Performance: “Free is a *Very* Good Price!!!” And Nothing Is Faster Than Doing Nothing!!!

RCU Area of Applicability

Read-Mostly, Stale &
Inconsistent Data OK
(RCU Works Great!!!)

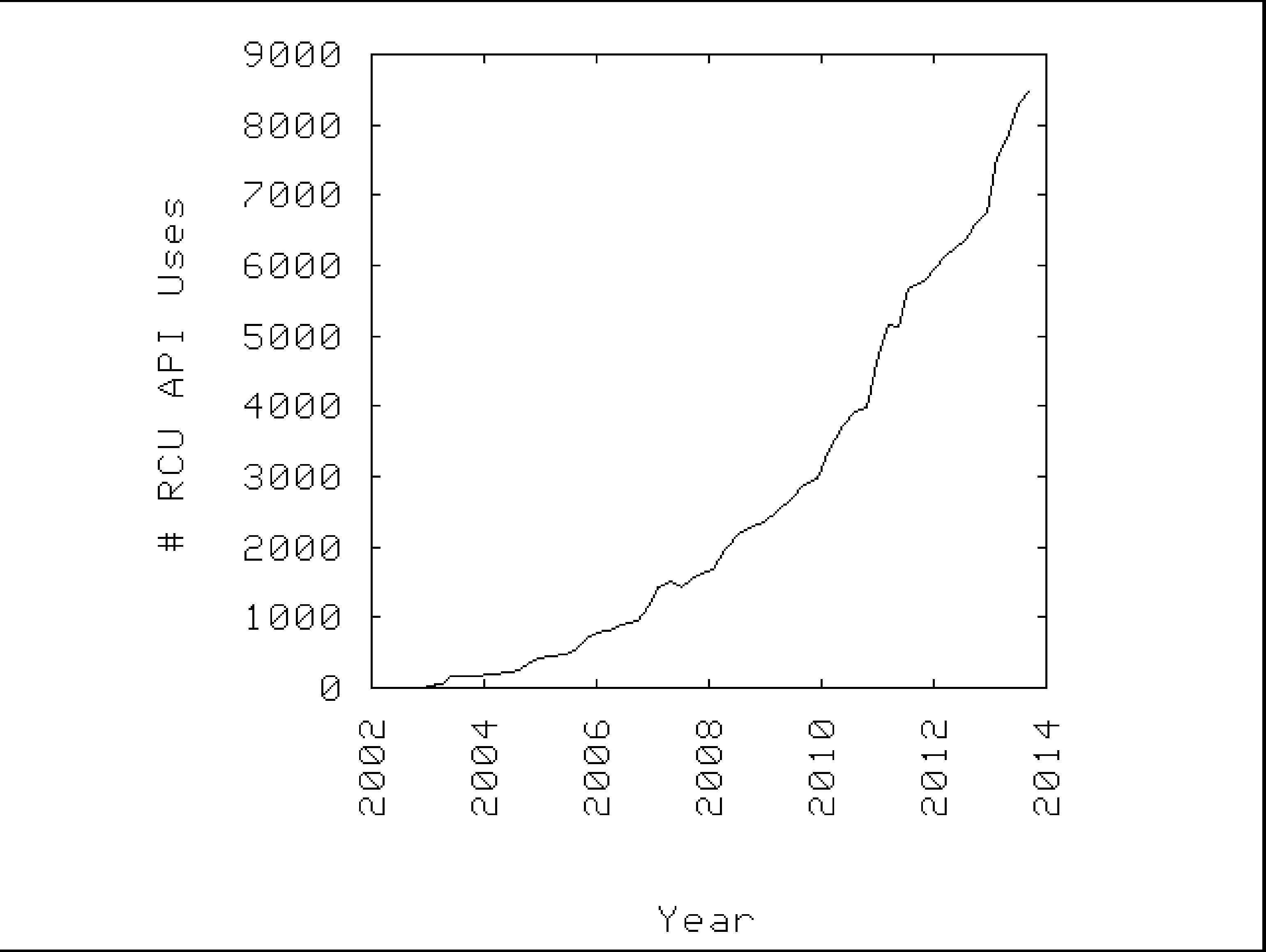
Read-Mostly, Need Consistent Data
(RCU Works OK)

Read-Write, Need Consistent Data
(RCU *Might* Be OK...)

Update-Mostly, Need Consistent Data
(RCU is ***Really*** Unlikely to be the Right Tool For The Job, But It Can:
(1) Provide Existence Guarantees For Update-Friendly Mechanisms
(2) Provide Wait-Free Read-Side Primitives for Real-Time Use)

Schrodinger's zoo is in blue: Can't tell exactly when an animal is born
or dies anyway! Plus, no lock you can hold will prevent an animal's death...

RCU Applicability to the Linux Kernel



Summary

Summary

- Synchronization overhead is a big issue for parallel programs
- Straightforward design techniques can avoid this overhead
  - Partition the problem: “Many instances of something good!”
  - Avoid expensive operations
  - Avoid mutual exclusion
- RCU is part of the solution
  - Excellent for read-mostly data where staleness and inconsistency OK
  - Good for read-mostly data where consistency is required
  - Can be OK for read-write data where consistency is required
  - Might not be best for update-mostly consistency-required data
  - Used heavily in the Linux kernel
- Much more information on RCU is available...

Graphical Summary

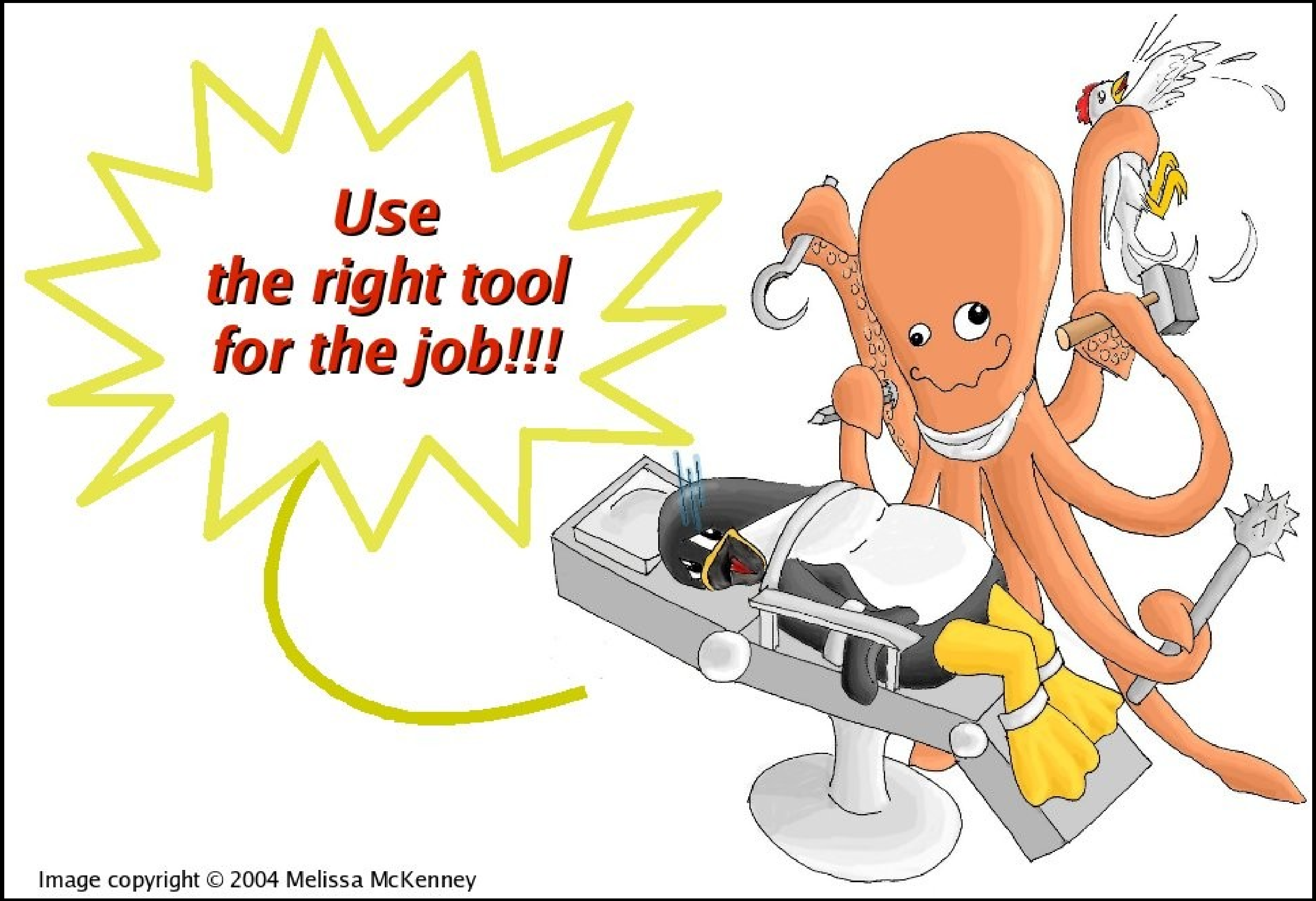
To Probe Further:

- https://queue.acm.org/detail.cfm?id=2488549
  - "Structured Deferral: Synchronization via Procrastination" (also in July 2013 CACM)
- http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.159 and http://www.computer.org/cms/Computer.org/dl/trans/td/2012/02/extras/ttd2012020375s.pdf
  - "User-Level Implementations of Read-Copy Update"
- git://lttng.org/userspace-rcu.git (User-space RCU git tree)
- http://people.csail.mit.edu/nickolai/papers/clements-bonsai.pdf
  - Applying RCU and weighted-balance tree to Linux mmap_sem.
- http://www.usenix.org/event/atc11/tech/final_files/Triplett.pdf
  - RCU-protected resizable hash tables, both in kernel and user space
- http://www.usenix.org/event/hotpar11/tech/final_files/Howard.pdf
  - Combining RCU and software transactional memory
- http://wiki.cs.pdx.edu/rp/: Relativistic programming, a generalization of RCU
- http://lwn.net/Articles/262464/, http://lwn.net/Articles/263130/, http://lwn.net/Articles/264090/
  - "What is RCU?" Series
- http://www.rdrop.com/users/paulmck/RCU/RCUdissertation.2004.07.14e1.pdf
  - RCU motivation, implementations, usage patterns, performance (micro+sys)
- http://www.livejournal.com/users/james_morris/2153.html
  - System-level performance for SELinux workload: >500x improvement
- http://www.rdrop.com/users/paulmck/RCU/hart_ipdps06.pdf
  - Comparison of RCU and NBS (later appeared in JPDC)
- http://doi.acm.org/10.1145/1400097.1400099
  - History of RCU in Linux (Linux changed RCU more than vice versa)
- http://read.seas.harvard.edu/cs261/2011/rcu.html
  - Harvard University class notes on RCU (Courtesy of Eddie Koher)
- http://www.rdrop.com/users/paulmck/RCU/ (More RCU information)

Legal Statement

- This work represents the view of the author and does not necessarily represent the view of IBM.
- IBM and IBM (logo) are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.
- Linux is a registered trademark of Linus Torvalds.
- Other company, product, and service names may be trademarks or service marks of others.
- Credits:
  - This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851.
  - Joint work with Mathieu Desnoyers, Alan Stern, Michel Dagenais, Manish Gupta, Maged Michael, Phil Howard, Joshua Triplett, Jonathan Walpole, and the Linux kernel community.
  - Additional reviewers: Carsten Weinhold and Mingming Cao.

Questions?


Use
the right tool
for the job!!!
Image copyright © 2004 Melissa McKenney