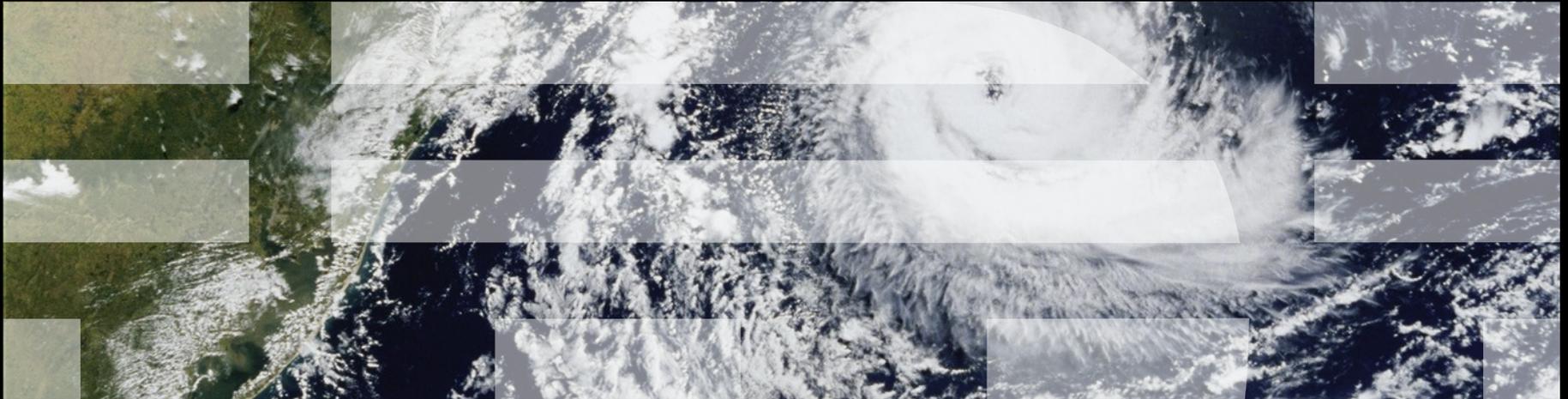Paul E. McKenney, IBM Distinguished Engineer (Linux Technology Center)

Member, IBM Academy of Technology

Accommodating the Laws of Physics: RCU

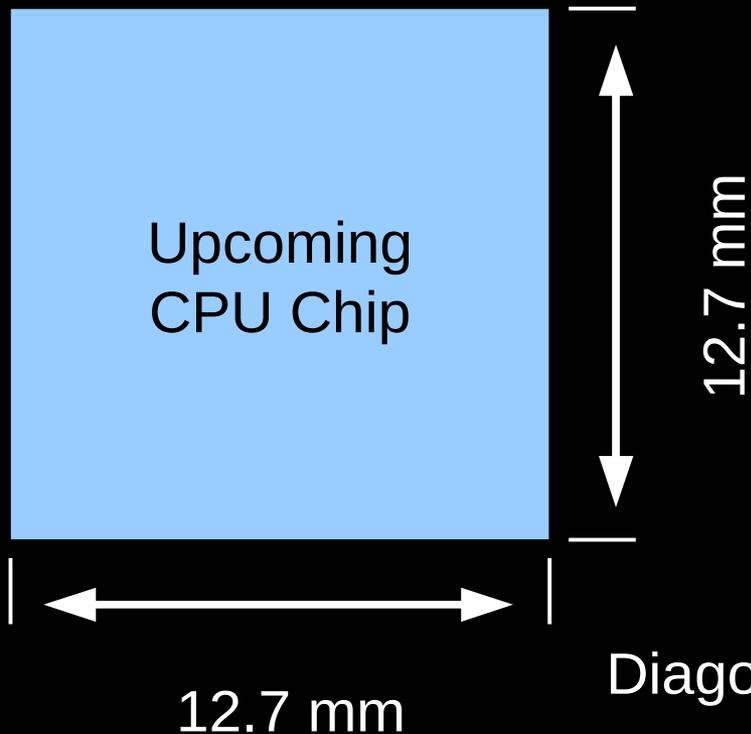*The SIGPLAN Programming Languages Mentoring Workshop*

January 22, 2013

The Laws of Physics

And other trivial issues...

Speed of Light (to Say Nothing of Electrons) is Finite; Size of Computers is Non-Zero

Upcoming CPU Chip

12.7 mm

12.7 mm

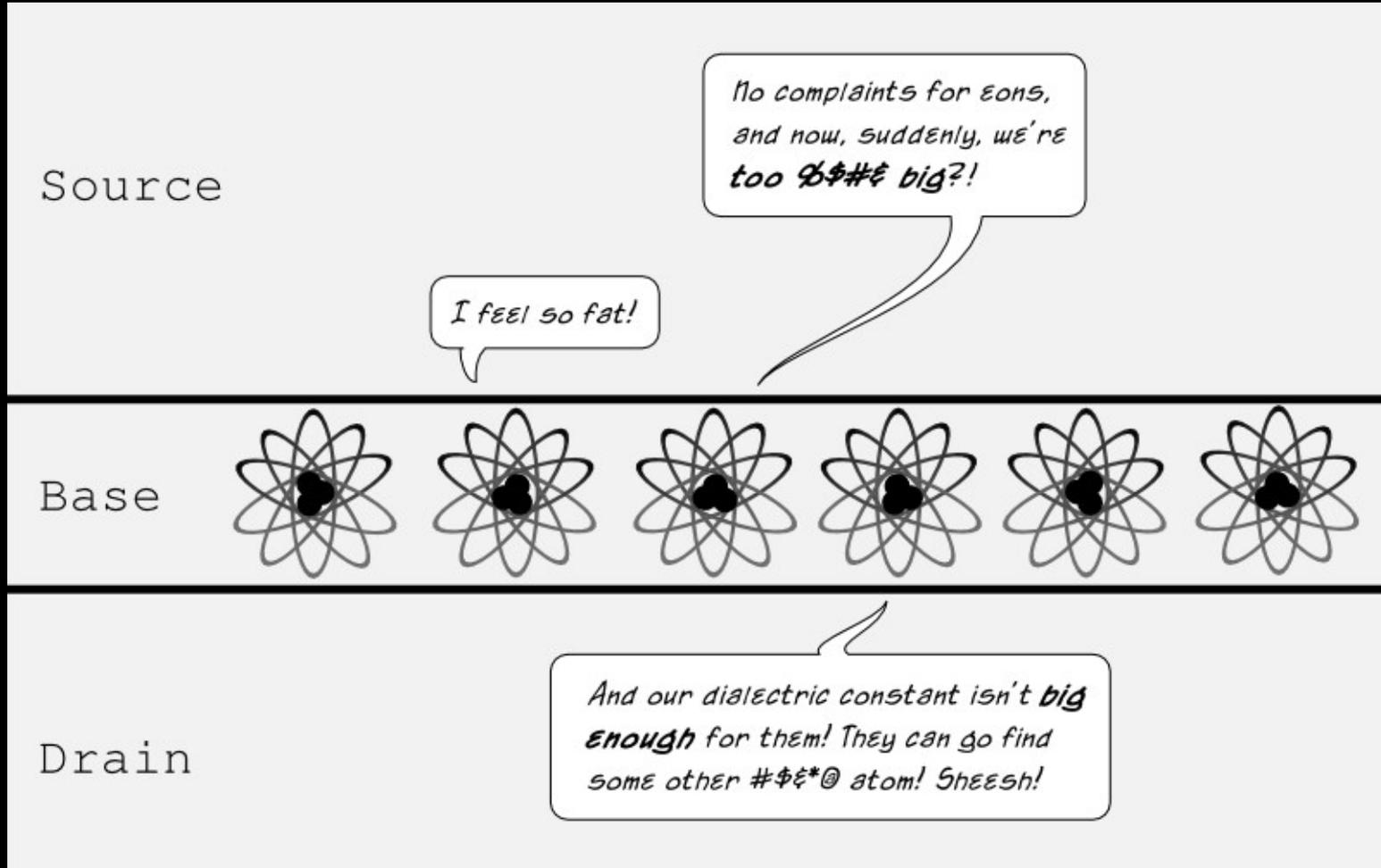Diagonally across chip and back (35.8mm):
3.6 clocks at 1GHz
17.9 clocks at 5GHz
Out for the request, back to return the data

Source: http://en.wikipedia.org/wiki/List_of_upcoming_intel_processors

Problem With Physics #1: Finite Speed of Light
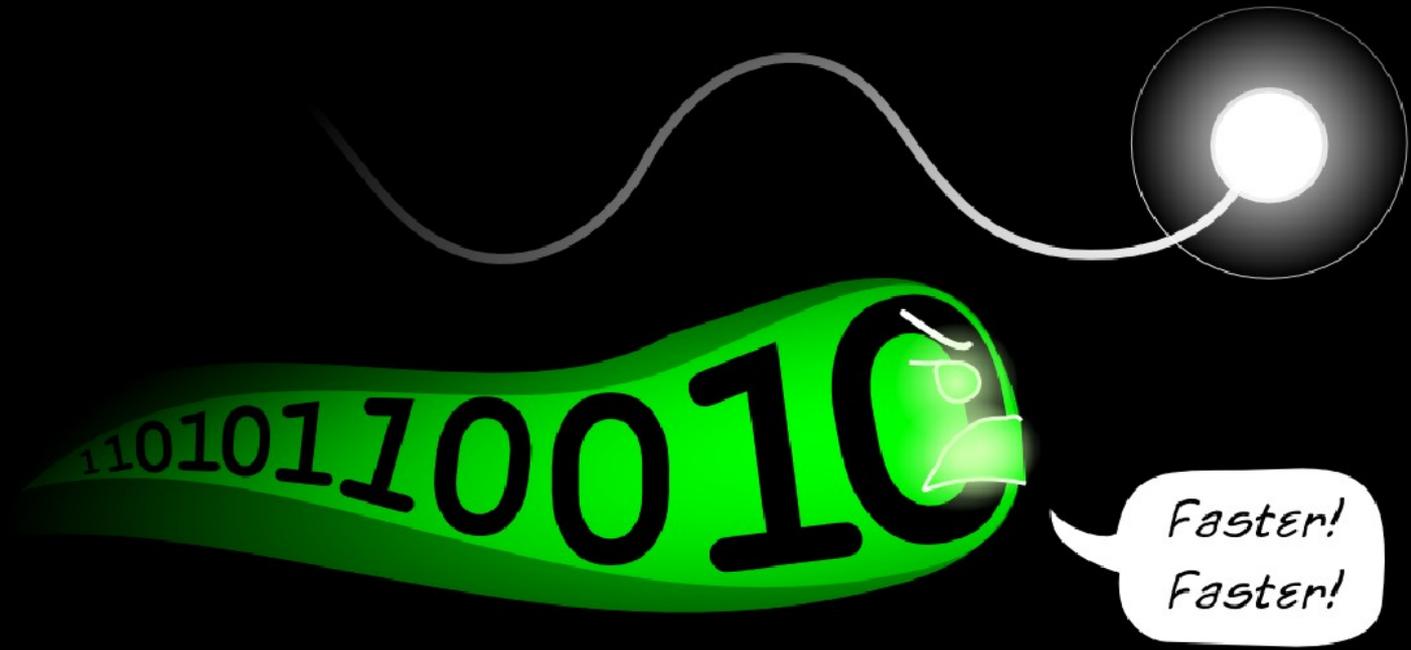
Problem With Physics #2: Atomic Nature of Matter

(c) 2012 Melissa Broussard, Creative Commons Share-Alike

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

That 3.6 and 17.9 clocks now looks pretty good... Buffering, queueing and caching result in substantial additional performance degradation!

7

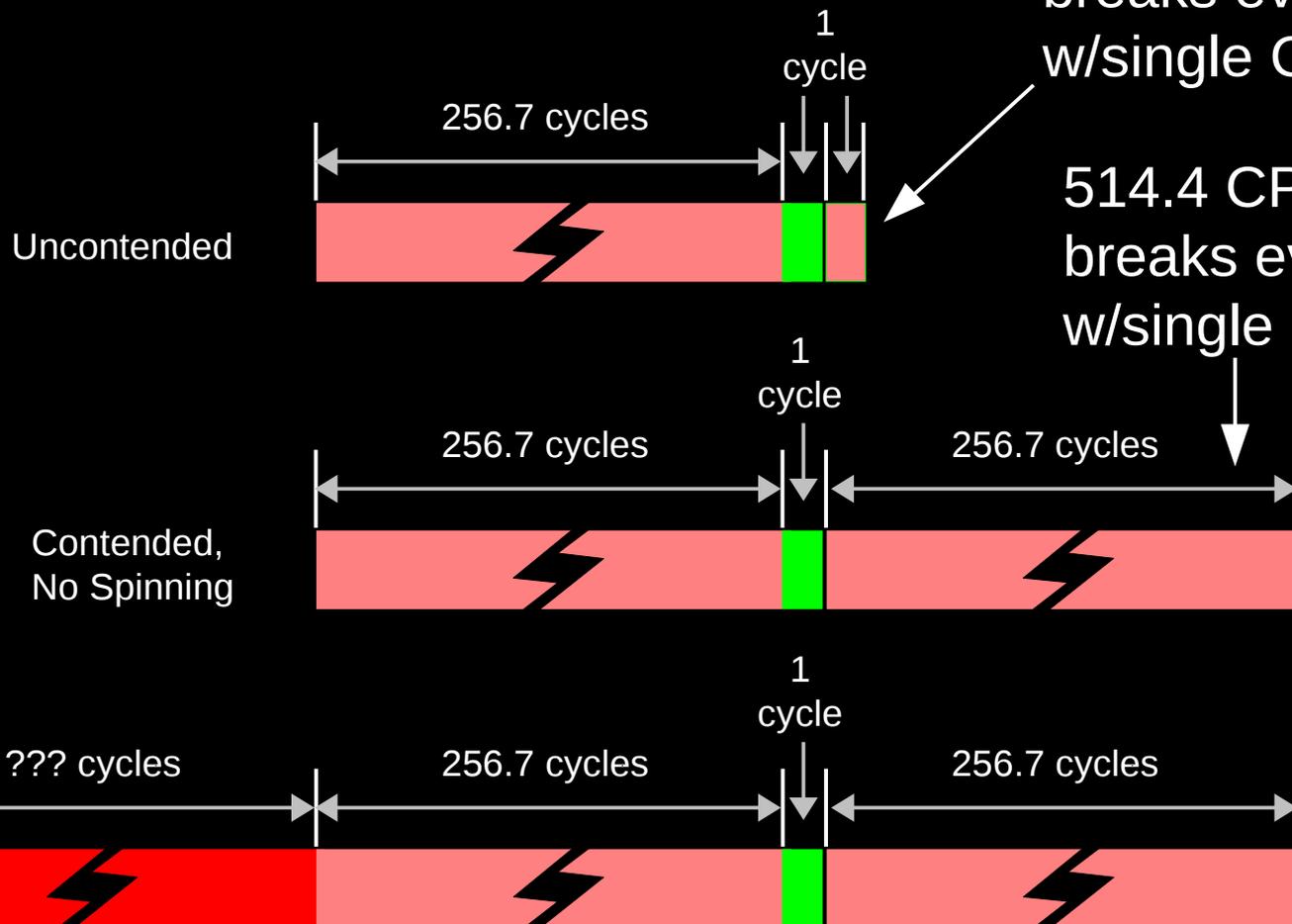# But What Do The Operation Timings Really Mean???

IBM

But What Do The Operation Timings Really Mean???

- Single instruction protected by *contended* lock

258.7 CPUs breaks even w/single CPU!

514.4 CPUs breaks even w/single CPU!!!

Arbitrarily large number of CPUs to break even with single CPU!!! Not so good for real-time!!!

1 cycle

256.7 cycles

Uncontended

1 cycle

256.7 cycles   256.7 cycles

Contended, No Spinning

1 cycle

??? cycles   256.7 cycles   256.7 cycles

Contended, Spinning

Also Applies to Reader-Writer Locking, Non-Blocking Synchronization and Transactional Memory

Though read-only transactions can be heavily optimized, but not as heavily as RCU can.

Can't Hardware Do Better Than This???

- There might be some ways to improve hardware:
  - 3D lithography: Too bad about power and heat dissipation!
  - Extreme ultraviolet lithography: Making progress, but limited
  - Liquid immersion lithography: Making progress, but limited
  - Asynchronous logic: big in the '60s, starting to be used again
  - Exotic materials (e.g., graphene): Promising, but still a research toy
  - Light rather than electrons: Promising, but still a research toy
  - Quantum tunneling: Tantalizing, but no way to send data faster
  - ***Wormholes: Works great on Star Trek!!!***
  - ***Hyperspace: Works great on Star Wars!!!***

- Although hardware will continue to improve, software needs to do its part: "Free lunch" exponential performance improvement of 80s and 90s is over

11

How Can Software Live With This Hardware???

Two Basic Ways To Proceed...

Acquire

Critical Section

1 cycle

256.7 cycles

Release

Uncontended
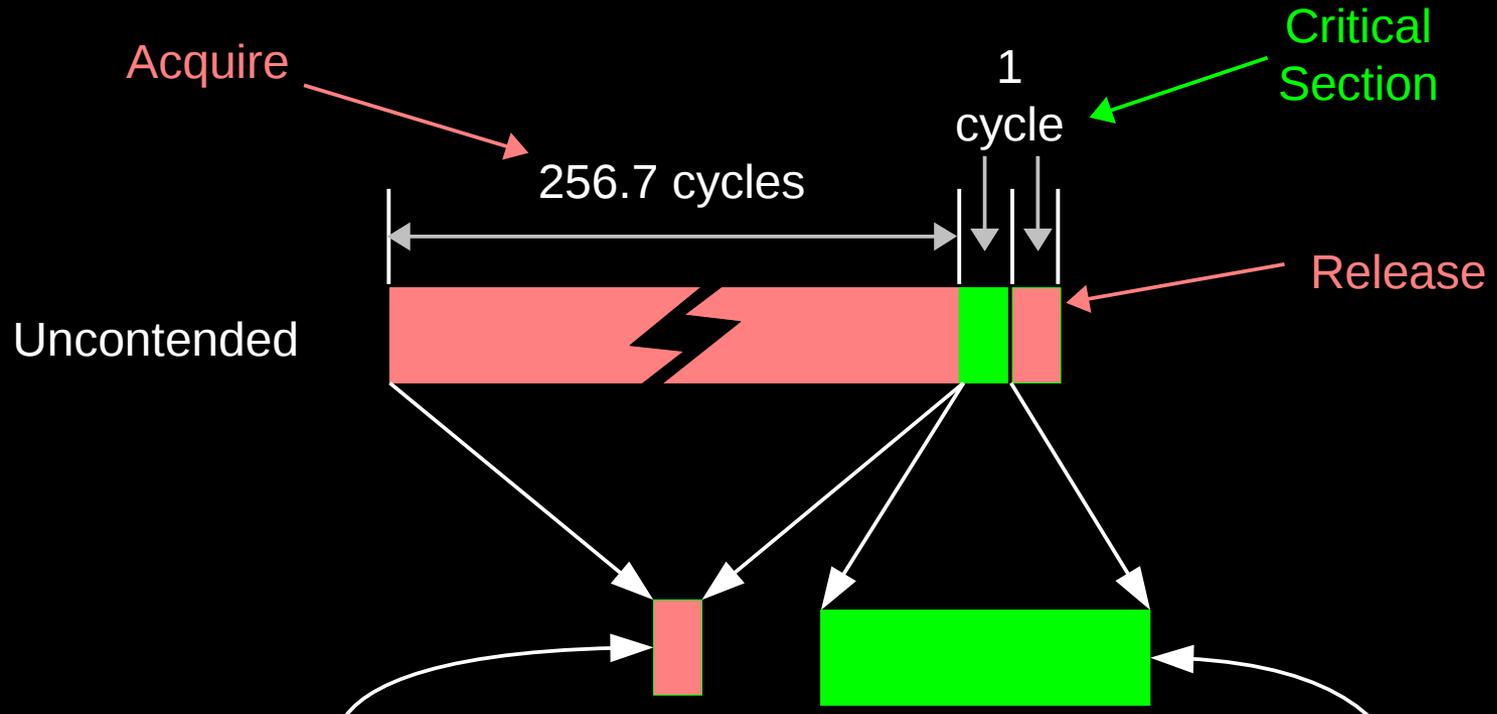
1: Reduce synchronization overhead

2: Increase critical section duration

We will focus on option #1, for readers. (In real life, you need to do both.)

Design Principle: Avoid Expensive Operations

16-CPU 2.8GHz Intel X5550 (Nehalem) System

Use cheap-and-cheerful operations

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

14

Taking It To The Limit...

**"Only those who have gone too far
can possibly tell you how far you can go!!!"**

Taking It To The Limit...

- 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

Taking It To The Limit...

- 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
- But how can these possibly be useful???

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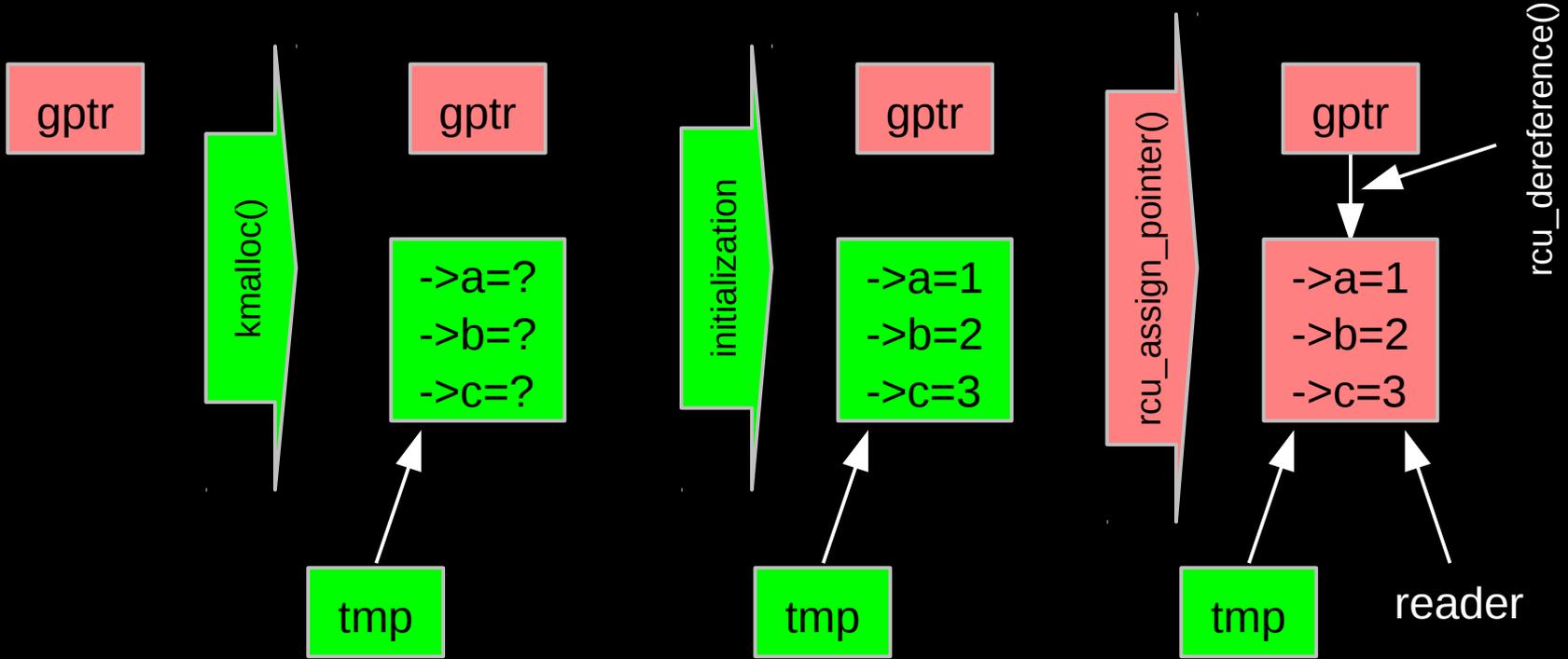
But if all we do is add, we have a big memory leak!!!

RCU Removal From Linked List

- Combines waiting for readers and multiple versions:
  - Writer removes element B from the list (list_del_rcu())
  - Writer waits for all pre-existing readers to finish (synchronize_rcu())
  - Writer can then free B (kfree())
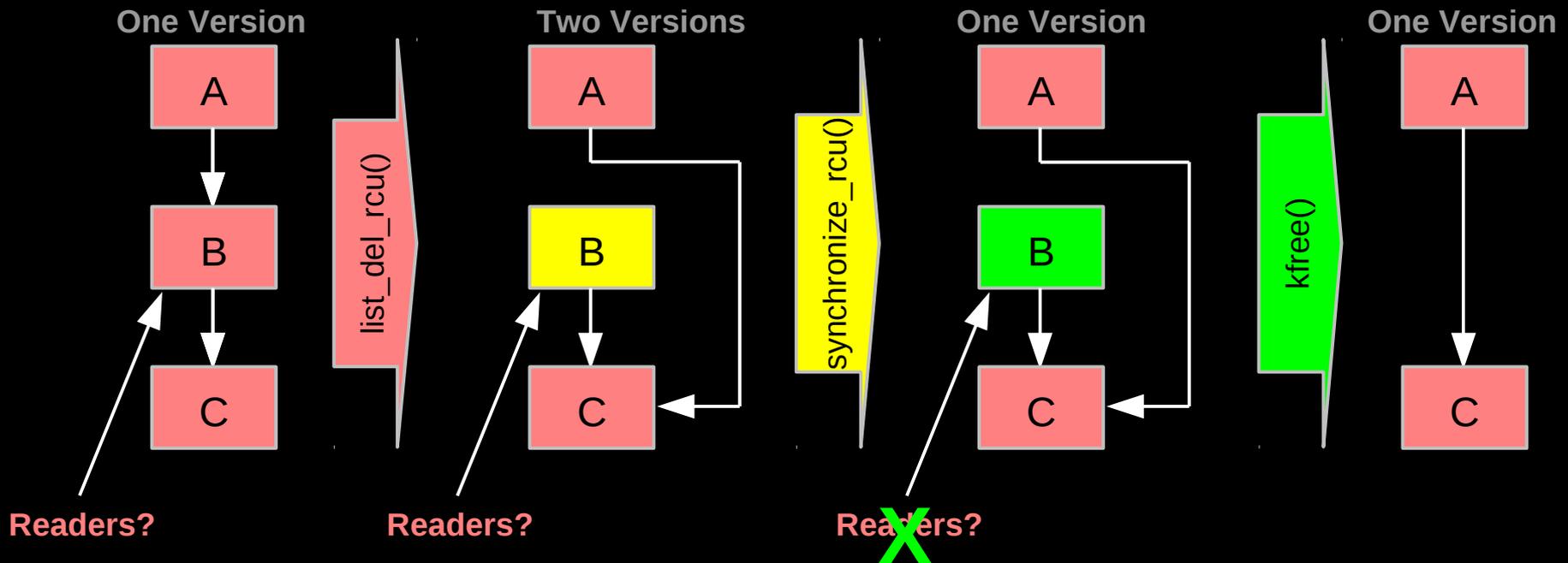
But if readers leave no trace in memory, how can we possibly tell when they are done???

19

How Can RCU Tell When Readers Are Done???

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

20

Waiting for Pre-Existing Readers: QSBR

- Non-preemptive environment (CONFIG_PREEMPT=n)
  - Tasks holding pure spinlocks are not allowed to block due to deadlock issues
  - Same rule for RCU readers, which are also not permitted to block

21

Waiting for Pre-Existing Readers: QSBR

- Non-preemptive environment (CONFIG_PREEMPT=n)
  - Tasks holding pure spinlocks are not allowed to block due to deadlock issues
  - Same rule for RCU readers, which are also not permitted to block

- CPU context switch means all that CPU's prior readers are done

- *Grace period* ends after all CPUs execute a context switch

Waiting for Pre-Existing Readers: QSBR

- Non-preemptive environment (CONFIG_PREEMPT=n)
  - Tasks holding pure spinlocks are not allowed to block due to deadlock issues
  - Same rule for RCU readers, which are also not permitted to block

- CPU context switch means all that CPU's prior readers are done

- *Grace period* ends after all CPUs execute a context switch

RCU reader

context switch

CPU 0

CPU 1

synchronize_rcu()

CPU 2

list_del_rcu()

Grace Period
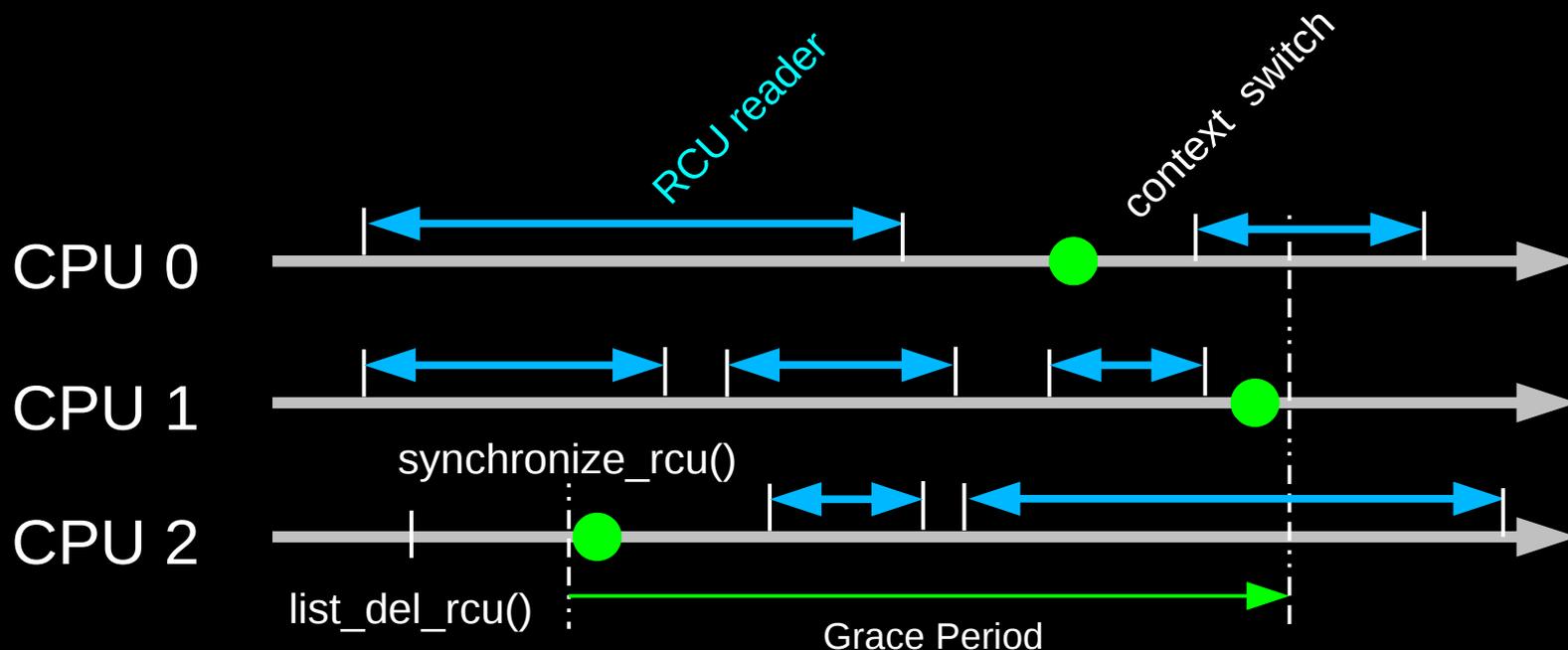
Toy Implementation of RCU: 20 Lines of Code

- 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(); \
        ACCESS_ONCE(p) = (v); \
})
void synchronize_rcu(void)
{
        int cpu;

        for_each_online_cpu(cpu)
                run_on(cpu);
}
```
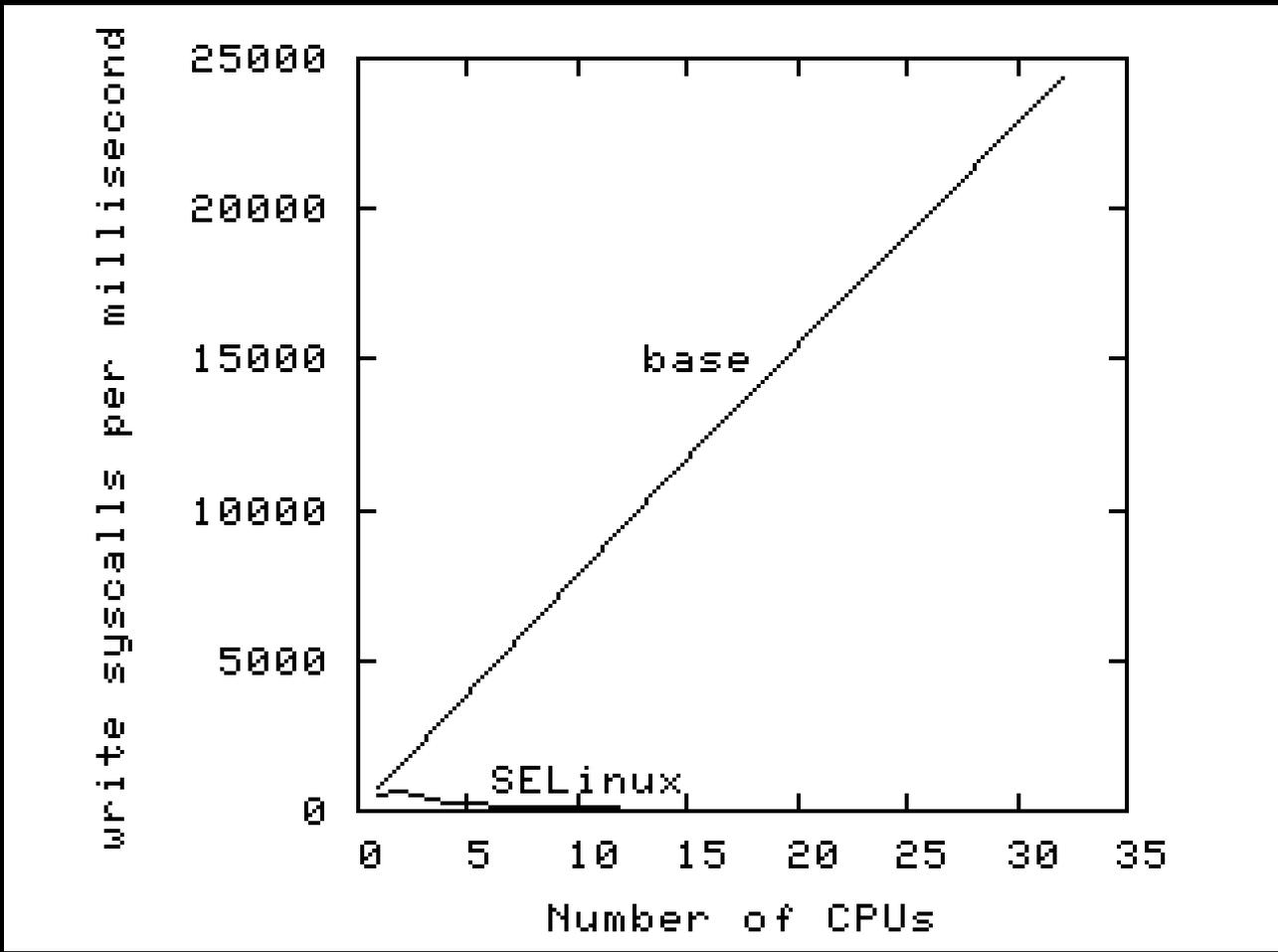
Toy Implementation of RCU: 20 Lines of Code

```
void synchronize_rcu(void)
{
        int cpu;

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

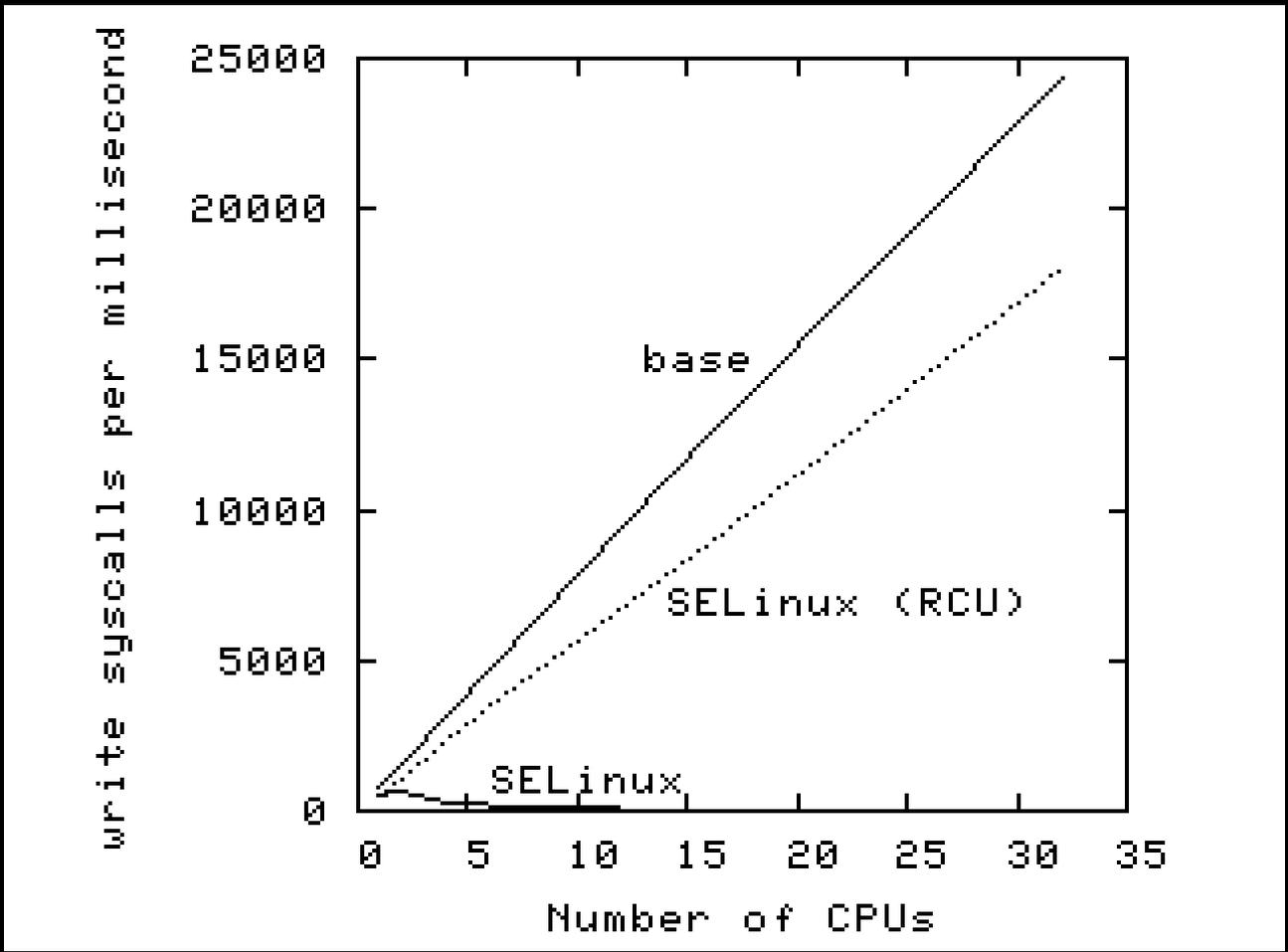And some people still insist that RCU is complicated... ;-)

25

Performance

Linux Kernel write() System Call: SELinux (Logscale)

Adding CPUs makes SELinux *slower!!!*

27
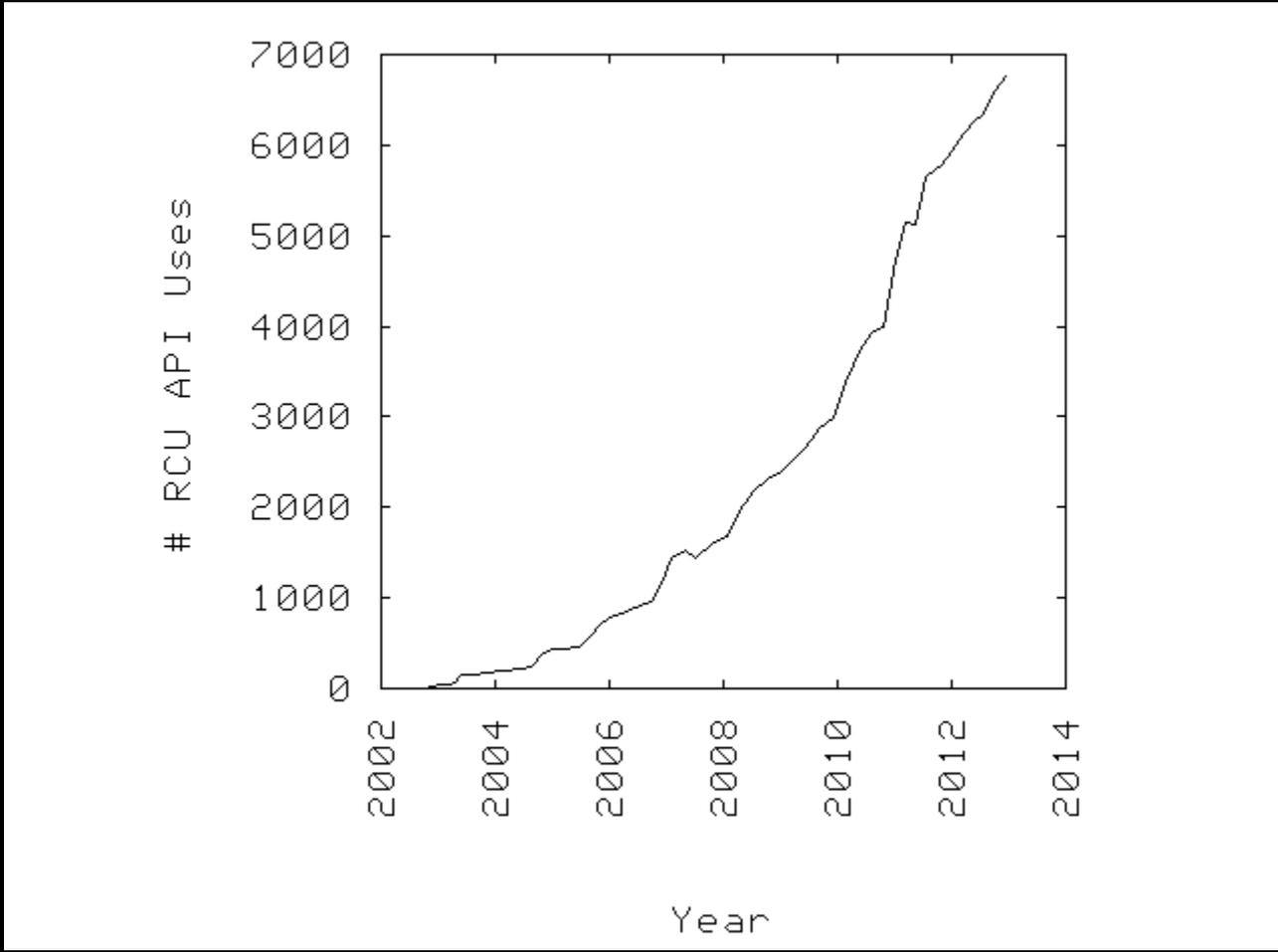
Linux Kernel write() System Call: SELinux (RCU)

RCU provides linear scalabilty *and* order-of-magnitude improvements

28

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 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)

RCU Applicability to the Linux Kernel

Open Questions

Open Questions: RCU Validation and Verification

- **Formal semantics for dependency ordering**
  - Peter Sewell et al., Cambridge University

- **Formal semantics for RCU; formal validation of uses of RCU**
  - WIP by Alexey Gotsman et al., IMDEA Software Institute

- **Key challenge: Formal validation of software in the large**
  - When I have needed to validate small software artifacts, this has almost always indicated that a better approach existed
  - But I am certainly not going to validate 13MLOC of the Linux kernel!
    - This is a job for large-scale formal validation
    - Needs to accommodate multiple synchronization mechanisms
  - WIP by many: Richard Bornat, Alexey Gotsman, Philippa Gardner, Peter Sewell, Peter O'Hearn, …
    - But a huge amount of work required – this is decidedly non-trivial

32

Open Questions: RCU for User-Space Applications

- RCU for user-level crowd simulation
  - Vigueras et al., Universidad de Valencia

- RCU for user-level relational databases
  - WIP by Giani et al., University of Toronto

- User-level networking
  - Stephen Hemminger, Vyatta

- Memcached (network-connected hash table)
  - Some work by Josh Triplett, Portland State University (2011 USENIX)

- Accelerate other parallel frameworks
  - OpenMP, etc.

- Key challenge: Huge number of applications and libraries

Open Questions: RCU Algorithms & Data Structures

- Lists, queues, stacks, hash tables, radix trees
  - Numerous uses in the Linux kernel

- Red-black trees
  - Linux community members have prototyped these
  - Phil Howard, Portland State University (HOTPAR'2011)

- Bonsai trees
  - Austin Clements, MIT (ASPLOS 2012)

- WIP for Judy arrays
  - Mathieu Desnoyers, Efficios (liburcu)

Open Questions: Other Specialized Mechanisms?

- RCU achieves spectacular performance via specialization

- A few other specializations have extreme performance:
  - Partitioning:
    - Split counters, see Chapter 4 of http://kernel.org/pub/linux/kernel/people/paulmck/perfbook/perfbook.html
    - Per-bucket-locked hash tables
    - Data ownership or sharding
  - Read-side sharing:
    - Hazard pointers
    - Sequence locking
  - Probabilistic computing (see RACES'2012)

- Additional classes of useful specialized mechanisms?

Summary

Summary

- Two SW design techniques can help meet the severe challenge posed by the laws of physics:
  - Reduce synchronization overhead ("Cheap and cheerful!!!")
  - Increase critical-section size ("Get your money's worth!!!")

- RCU is part of the "cheap and cheerful" solution
  - RCU synchronization operates via social engineering and demonstrates the awesome power of procrastination

- "The best way to predict the future is to invent it"
  - But even this method is not foolproof
  - In fact, I cannot even tell you how far you can go...

Summary

- Two SW design techniques can help meet the severe challenge posed by the laws of physics:
    - Reduce synchronization overhead ("Cheap and cheerful!!!")
    - Increase critical-section size ("Get your money's worth!!!")

- RCU is part of the "cheap and cheerful" solution
    - RCU synchronization operates via social engineering and demonstrates the awesome power of procrastination

- "The best way to predict the future is to invent it"
    - But even this method is not foolproof
    - In fact, I cannot even tell you how far you can go...
    - ... because I went as far as I could imagine, and it still worked!!!

To Probe Further:

- http://www.seas.gwu.edu/~gparmer/ospert12/bigrt.2012.07.10a.pdf
  - Real-Time Response on Multicore Systems: It is Bigger Than You Think
- 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 Eddie Koher)
- http://www.rdrop.com/users/paulmck/RCU/ (More RCU information)

39

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, Mingming Cao, and Jonathan Walpole.
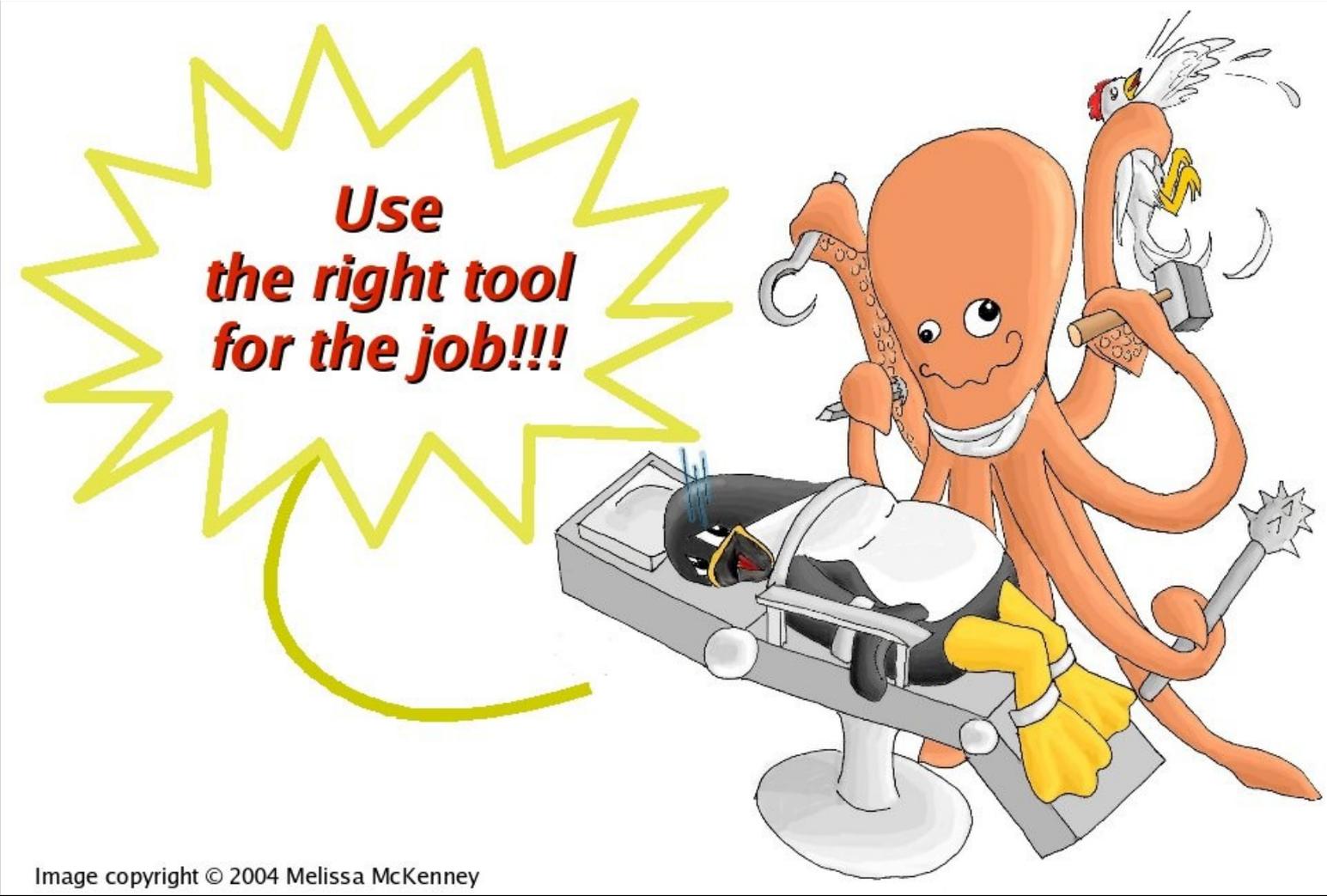
Questions?

Image copyright © 2004 Melissa McKenney

Backup

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?

- If not, why would you trust algorithms from someone ignorant of the properties of the underlying hardware???

43

RCU Usage Within the Linux Kernel

- Design Use Cases:
  - Publish-Subscribe
  - Wait for Completion
  - Restricted Reference Count
  - Bulk Reference Count
  - Existence Guarantees
  - Type-Safe Memory
  - Poorhouse Garbage Collector
  - Reader-Writer Lock Replacement

- Algorithmic Transformations
  - Impose Level of Indirection
  - Mark Obsolete Object
  - Retry Readers

Linux Kernel write() System Call: SELinux (RCU)

Keep SELinux off the x-axis...

45
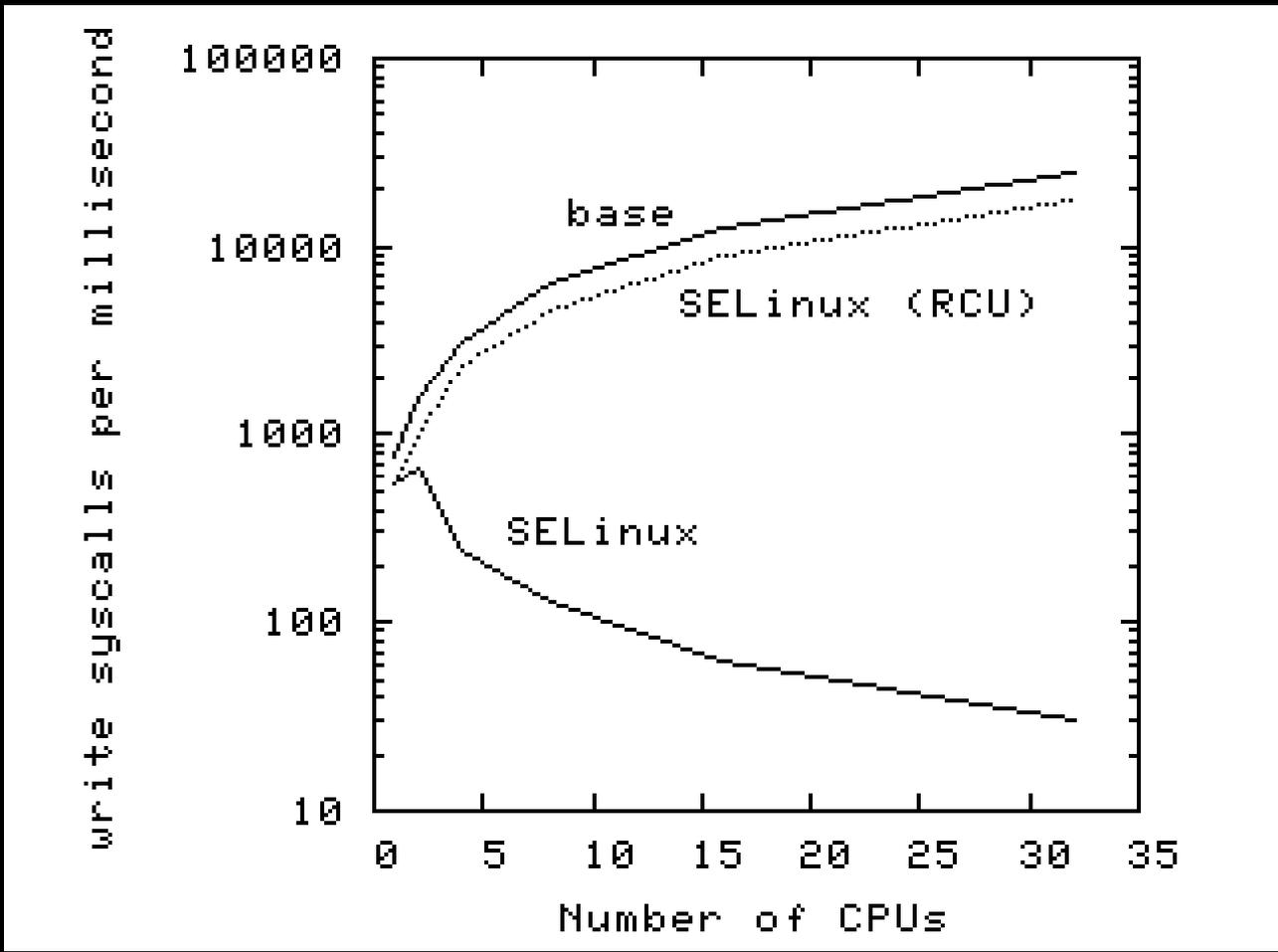
RCU Without Disabling Preemption

- User-level RCU allows preemption with free readers
  - "User-Level Implementation of Read-Copy Update"
  - http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.159
  - Or liburcu package on recent Linux distributions
  - Instead of context switch, momentary quiescent state:
    - rcu_quiescent_state(): End of polling loop, transaction, whatever
  - Or extended quiescent state for long-term blocking:
    - rcu_thread_online(): Beginning of event handling
    - rcu_thread_offline(): End of event handling
  - Either way, application tells RCU when to ignore a given thread
    - And rcu_read_lock() and rcu_read_unlock() can still be free

- In-kernel preemptible RCU has non-free readers
  - But extremely lightweight in the common case
  - Local counter increment, decrement, and check

But Isn't RCU Only For OS Kernels?

But Isn't RCU Only For OS Kernels? Not at All!!!

- User-space RCU has been available for some years
  - http://lttng.org/urcu: Project page
  - git://lttng.org/userspace-rcu.git: Git tree
  - "User-Level Implementations of Read-Copy Update", Mathieu Desnoyers, Paul E. McKenney, Alan Stern, Michel Dagenais, and Jonathan Walpole, IEEE Transactions on Parallel and Distributed Systems, v23, February 2012, pp 375-382

- A few users:
  - LTTng (http://lttng.org/)
  - Knot DNS (http://www.knot-dns.cz/)
  - Crowd simulation (http://dx.doi.org/10.1007/s11227-012-0766-x)

- Available in a number of recent distros as liburcu

RCU Usage Within the Linux Kernel

- Design Use Cases:
  - Wait for Completion
  - Restricted Reference Count
  - Bulk Reference Count
  - Existence Guarantees
  - Type-Safe Memory
  - Poorhouse Garbage Collector
  - Reader-Writer Lock Replacement

- Algorithmic Transformations
  - Impose Level of Indirection
  - Mark Obsolete Object
  - Retry Readers

49

Reader-Writer Locking Replacement

- Problem: Need low-overhead readers
  - Avoid cache-miss overhead associated with locking
  - In cases where temporal mutual exclusion is not critically important

- Solution: Use RCU readers with linked data structure
  - Pack the variables into a structure, so that a pointer can be updated atomically to readers

Reader-Writer Locking

```
struct foo_head {
    struct list_head list;
    rwlock_t mutex;
};

struct foo {
    struct list_head list;
    int key;
};

int search(struct foo_head *fhp, int k)
{
    struct foo *p;
    struct list_head *head = &fhp->list;

    read_lock(&fhp->mutex);
    list_for_each_entry_rcu(p, head, list) {
        if (p->key == k) {
            read_unlock(&fhp->mutex);
            return 1;
        }
    }
    read_unlock(&fhp->mutex);
    return 0;
}
```

```
foo_head  <-->  foo  <-->  foo
```

```
int delete(struct foo_head *fhp, int k)
{
    struct foo *p;
    struct list_head *head = &fhp->list;

    write_lock(&fhp->mutex);
    list_for_each_entry(p, head, list) {
        if (p->key == k) {
            list_del_rcu(p);
            write_unlock(&fhp->mutex);
            /* */
            kfree(p);
            return 1;
        }
    }
    write_unlock(&fhp->mutex);
    return 0;
}
```

RCU as Reader-Writer Locking Replacement

```c
struct foo_head {
    struct list_head list;
    spinlock_t mutex;
};

struct foo {
    struct list_head list;
    int key;
};

int search(struct foo_head *fhp, int k)
{
    struct foo *p;
    struct list_head *head = &fhp->list;

    rcu_read_lock();
    list_for_each_entry_rcu(p, head, list) {
        if (p->key == k) {
            rcu_read_unlock();
            return 1;
        }
    }
    rcu_read_unlock();
    return 0;
}
```

```c
int delete(struct foo_head *fhp, int k)
{
    struct foo *p;
    struct list_head *head = &fhp->list;

    spin_lock(&fhp->mutex);
    list_for_each_entry(p, head, list) {
        if (p->key == k) {
            list_del_rcu(p);
            spin_unlock(&fhp->mutex);
            synchronize_rcu();
            kfree(p);
            return 1;
        }
    }
    spin_unlock(&fhp->mutex);
    return 0;
}
```
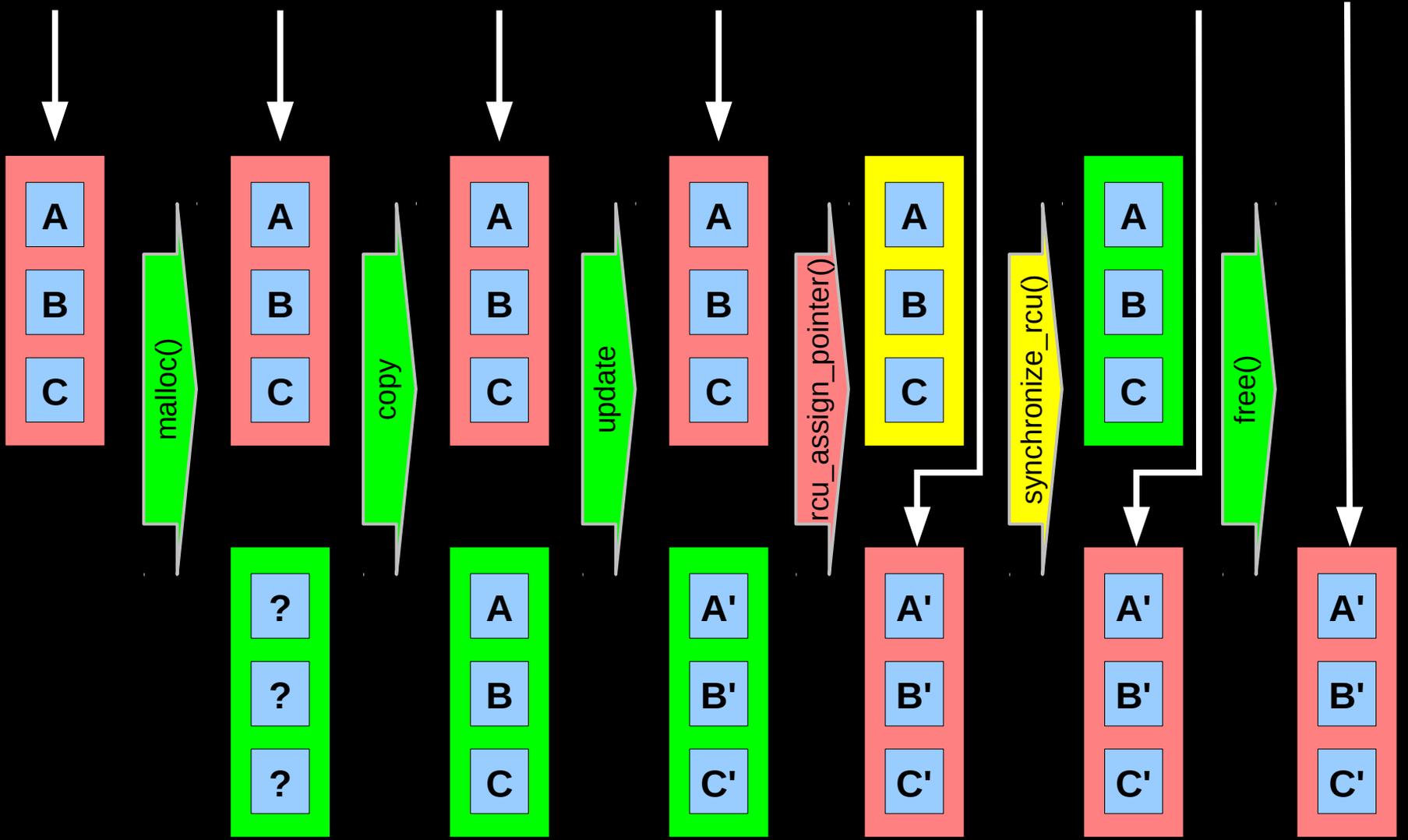
| foo_head | | foo | | foo |

52

Imposing Level of Indirection

- Problem: Need consistent view of several unrelated variables
  - RCU cannot provide a consistent view of these variables

- Solution: Impose level of indirection (ILOI)
  - Pack the variables into a structure, so that a pointer can be updated atomically to readers
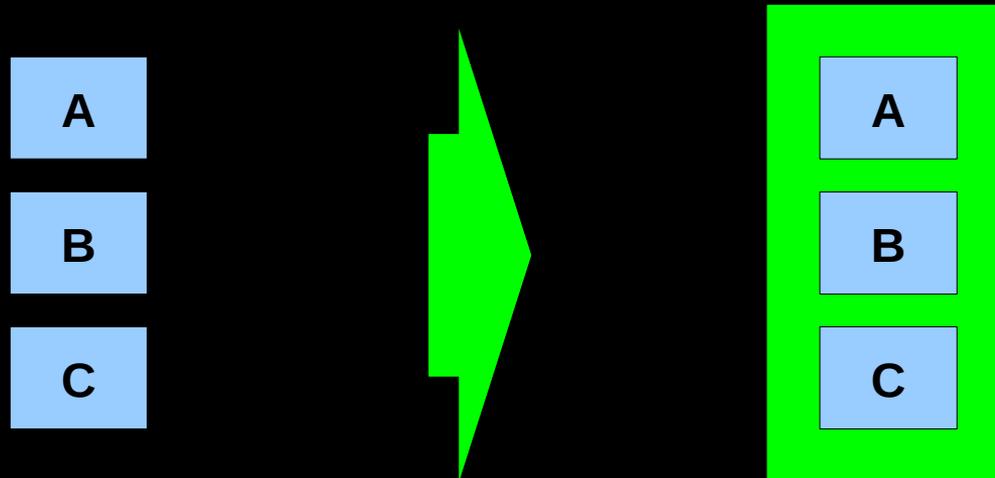
Imposing Level of Indirection: Update Process

Marking Obsolete Objects

▪ Problem: Need low-overhead readers
  – Avoid cache-miss overhead associated with locking
  – In cases where temporal mutual exclusion is not critically important
  – But where readers cannot tolerate stale data

▪ Solution: Use RCU as a reader-writer locking replacement
  – And keep per-data-element lock and "deleted" flag
  – When readers encounter a deleted data element, they pretend not to have found it – otherwise, they continue holding the element's lock

Marking Obsolete Objects

```
struct foo_head {
    struct list_head list;
    spinlock_t mutex;
};


Struct foo
*search(struct foo_head *fhp, int k)
{
    struct foo *p, *q;
    struct list_head *head = &fhp->list;

    rcu_read_lock();
    list_for_each_entry_rcu(p, head, list) {
        if (p->key == k) {
            q = p;
            spin_lock(&p->mutex);
            if (p->deleted) {
                q = NULL;
                spin_unlock(&p->mutex);
            }
            rcu_read_unlock();
            return q;
        }
    }
    rcu_read_unlock();
    return NULL;
}
```

```
struct foo {
    struct list_head list;
    int key;
    bool deleted;
    spinlock_t mutex;
};


void delete(struct foo_head *fhp,
            struct foo *p)
{
    struct list_head *head = &fhp->list;

    spin_lock(&fhp->mutex);
    list_del_rcu(p);
    p->deleted = true;
    spin_unlock(&p->mutex);
    spin_unlock(&fhp->mutex);
    synchronize_rcu();
    kfree(p);
}
```

Retrying Readers

▪ Problem: Need low-overhead readers
  – Avoid cache-miss overhead associated with locking
  – In cases where updates must be excluded

▪ Solution: Use RCU as a reader-writer locking replacement in conjunction with sequence locking
  – RCU provides existence guarantees, protecting pointer traversals
  – Sequence locking excludes updaters
  – Read-side critical sections must be idempotent