Performance and Scalability Technologies in the Linux Kernel

Paul E. McKenney
IBM Distinguished Engineer & CTO Linux
Linux Technology Center

Course Objectives and Goals

- Introduction to Performance, Scalability, and Real-Time Issues on Modern Multicore Hardware: Is Parallel Programming Hard, And If So, Why?
- **Performance and Scalability Technologies in the Linux Kernel**
- Creating Performant and Scalable Linux Applications
- Real-Time Technologies in the Linux Kernel
- Creating Real-Time Linux Applications

Overview

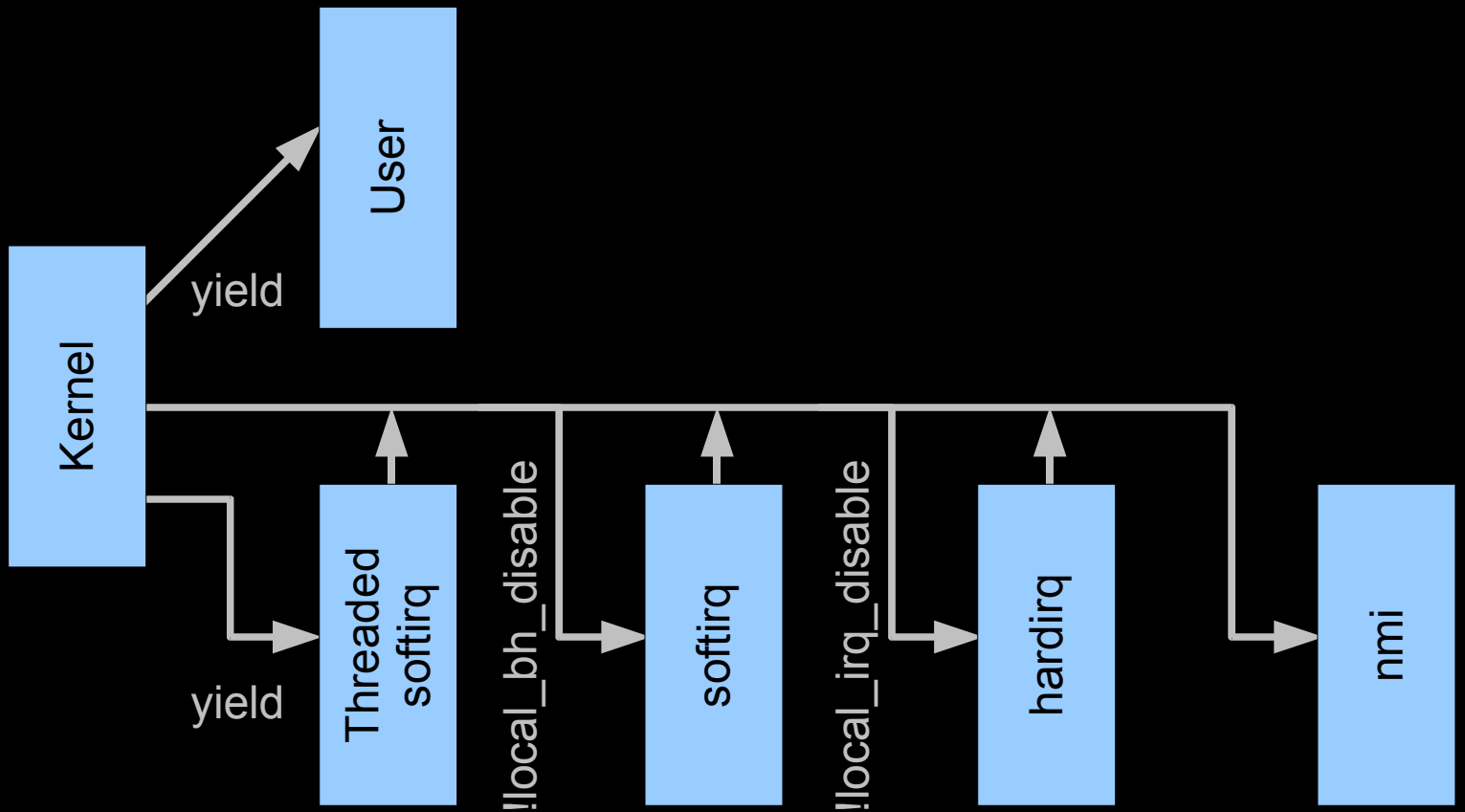
- Programming Environments in Linux Kernel
- Synchronization Primitives
- Per-CPU Variables
- The Existence Problem
- Solutions to the Existence Problem

Overview

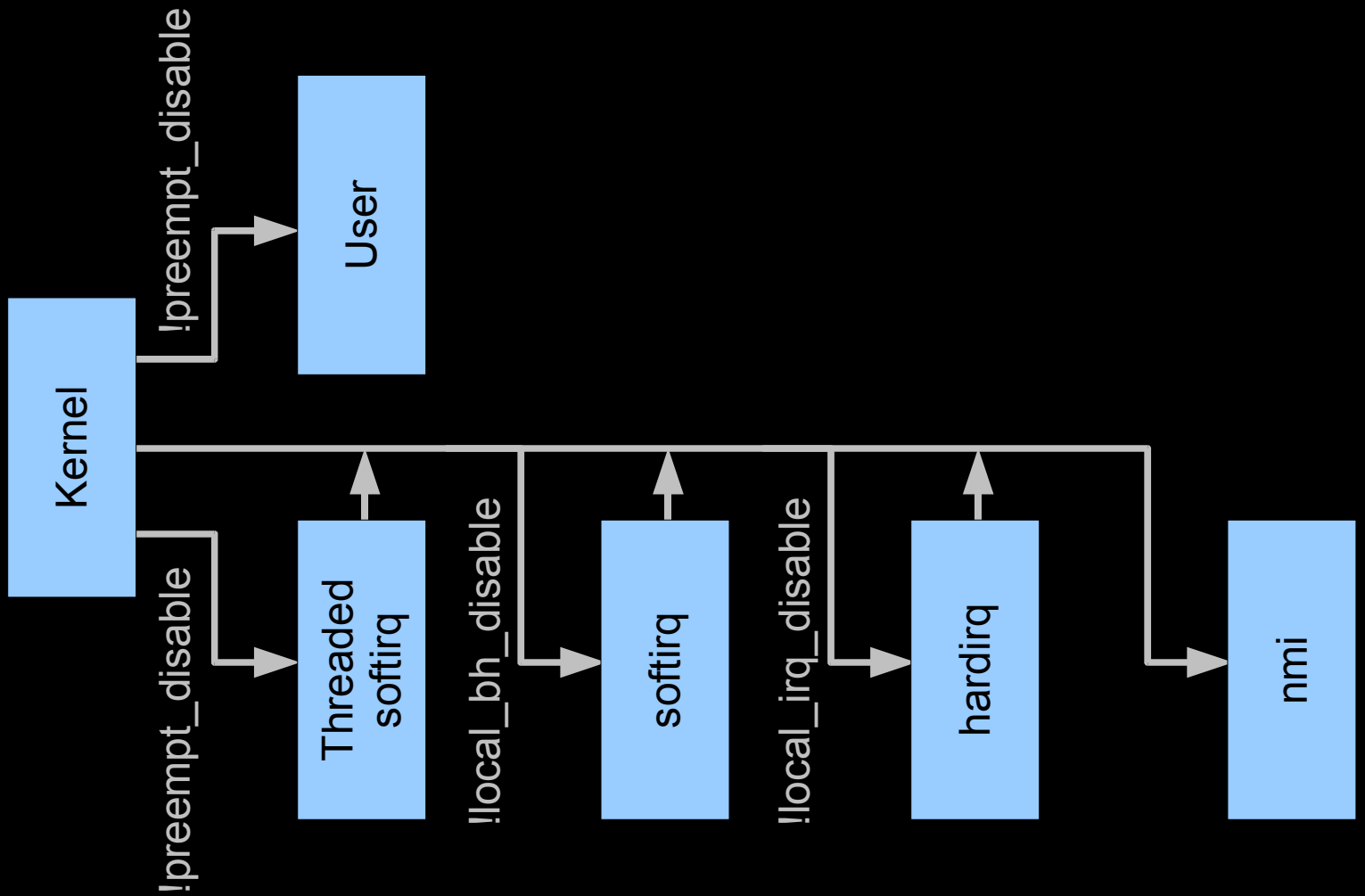
- Programming Environments in Linux Kernel
- Synchronization Primitives
- Per-CPU Variables
- The Existence Problem
- Solutions to the Existence Problem
- RCU API
- Why Free and Open-Source Software?

Programming Environments in Linux Kernel

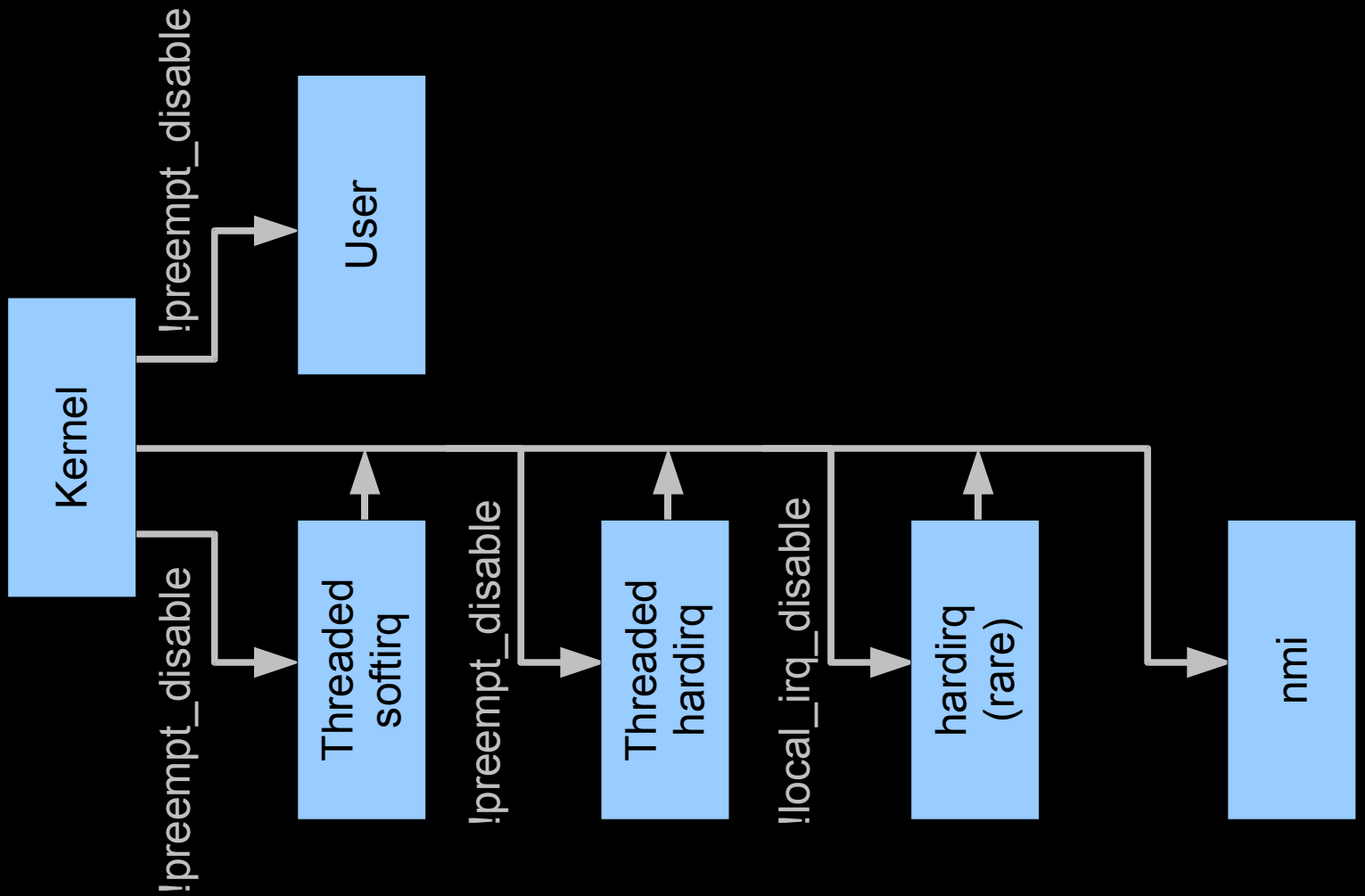
!PREEMPT Environments in Linux Kernel



PREEMPT Environments in Linux Kernel

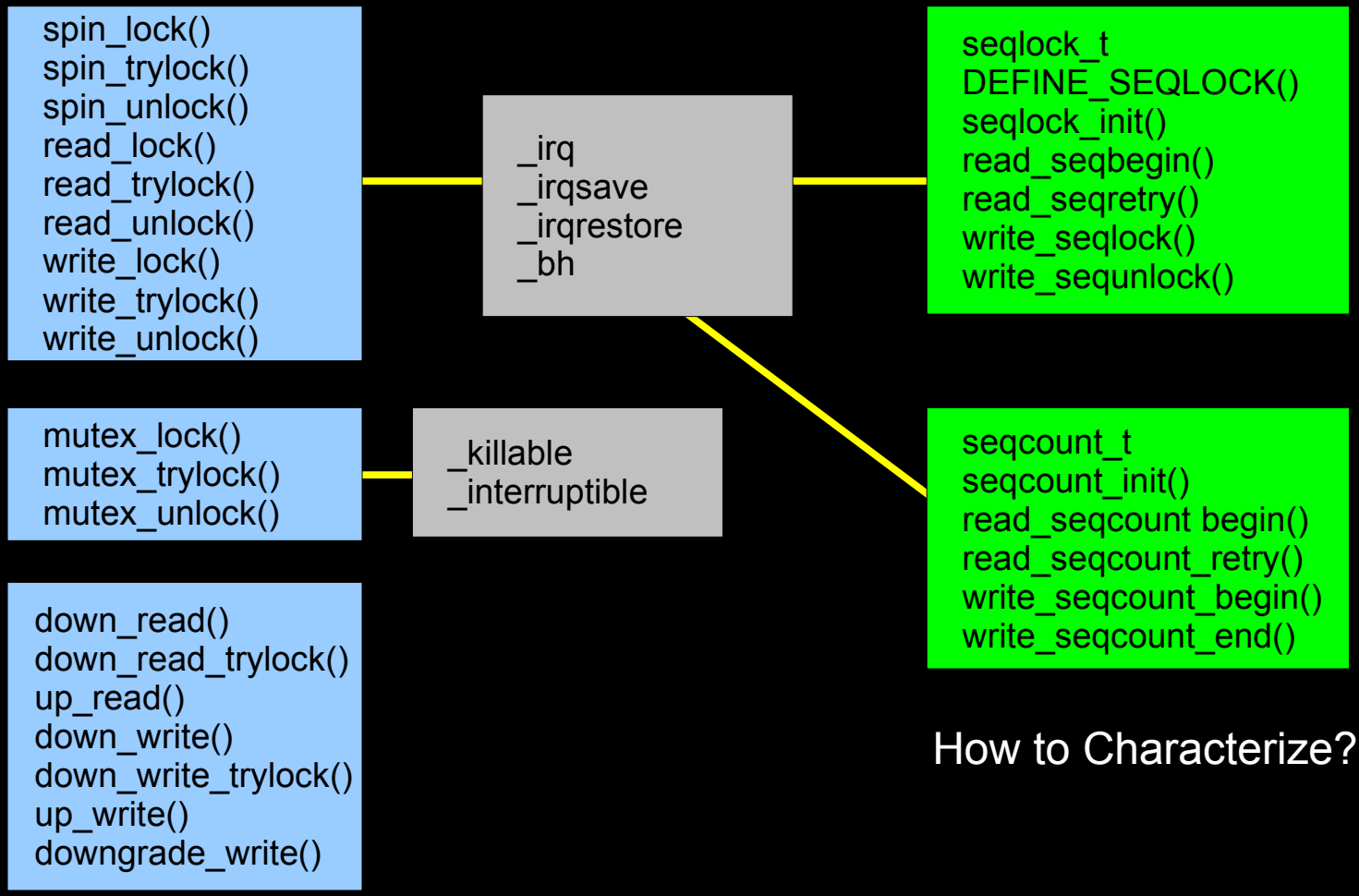


PREEMPT_RT Environments in Linux Kernel



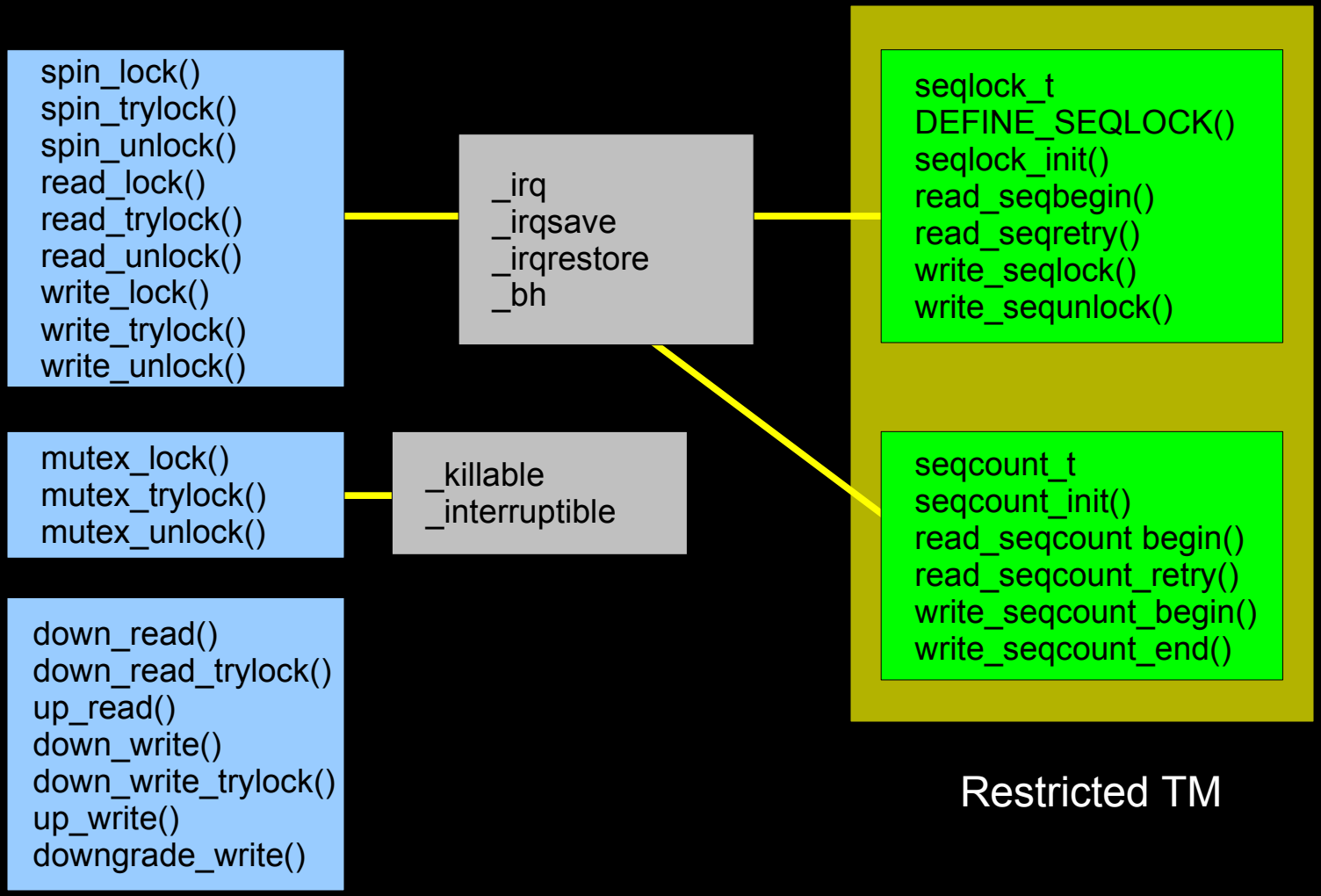
Synchronization Primitives

Synchronization Primitives (Partial)



How to Characterize?

Synchronization Primitives (Partial)



Sequence Lock Example (Reader)

```
void ktime_get_ts(struct timespec *ts)
{
        struct timespec tomono;
        unsigned long seq;

        do {
                seq = read_seqbegin(&xtime_lock);
                getnstimeofday(ts);
                tomono = wall_to_monotonic;

        } while (read_seqretry(&xtime_lock, seq));

        set_normalized_timespec(ts, ts->tv_sec + tomono.tv_sec,
                                ts->tv_nsec + tomono.tv_nsec);
}
```

Sequence Lock Example (Writer)

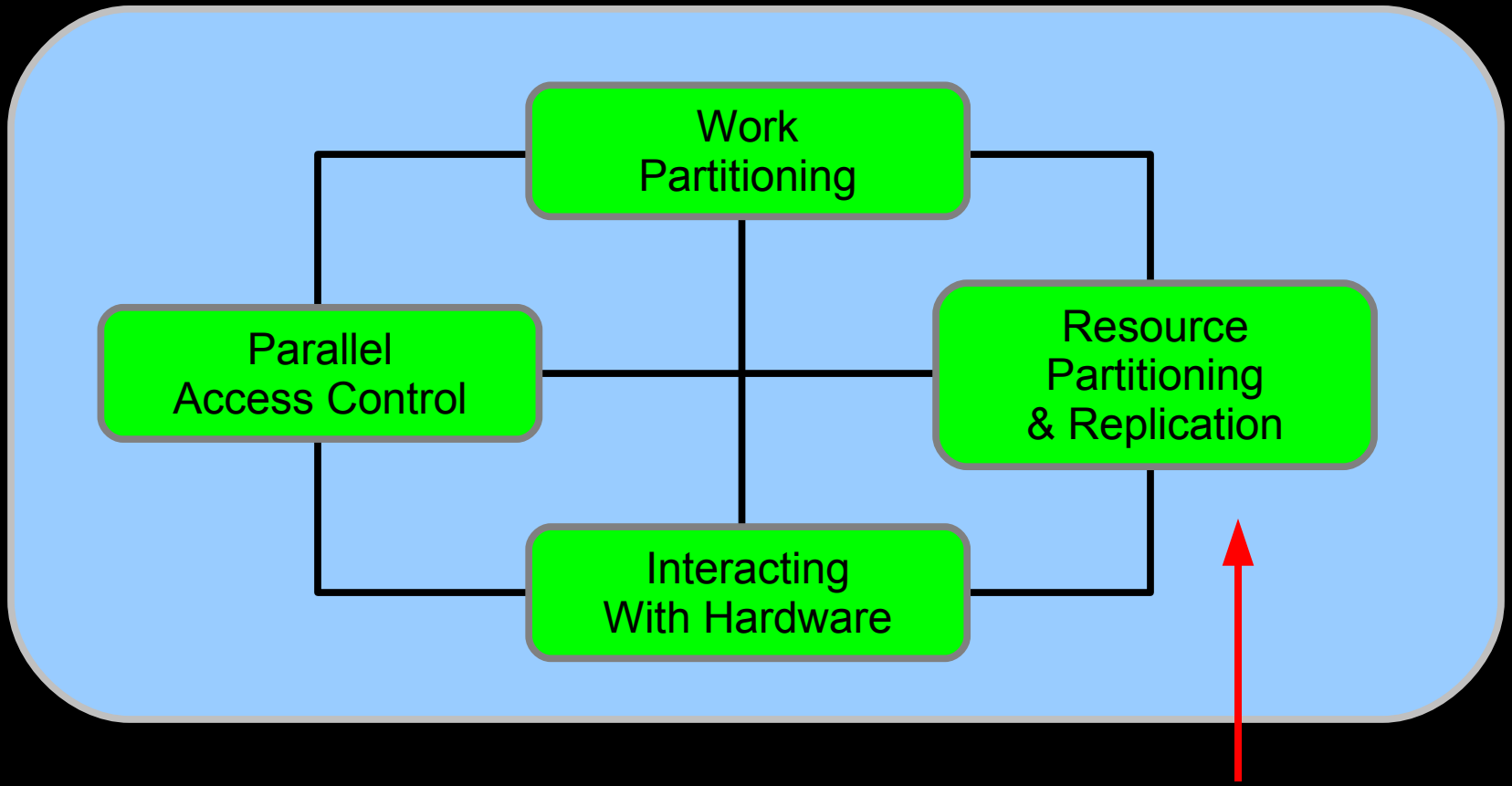
```
static inline void warp_clock(void)
{
        write_seqlock_irq(&xtime_lock);
        wall_to_monotonic.tv_sec -= sys_tz.tz_minuteswest * 60;
        xtime.tv_sec += sys_tz.tz_minuteswest * 60;
        update_xtime_cache(0);
        write_sequnlock_irq(&xtime_lock);
        clock_was_set();
}
```

Sleep/Wakeup Primitives (Partial)

- wait_event(), wait_event_interruptible(), wait_event_timeout(), wait_event_killable(), wait_event_interruptible_timeout()
  - wake_up(), wake_up_process(), ...
- wait_on_bit()
  - wake_up_bit()

- These are favored over traditional sleep_on() APIs for software-engineering reasons
  - “Unsafe at any speed” primitives get fixed
  - Sometimes repeatedly...

Synchronization Primitives



But do this first!!!!
Job #1 is *not* selecting primitives!

Per-CPU Variables

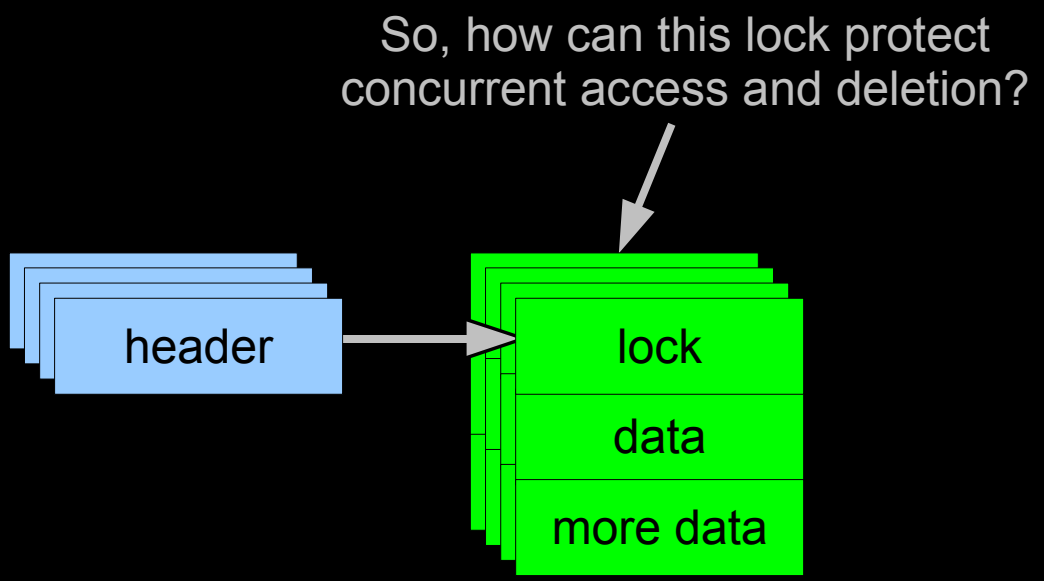
Per-CPU Variables

- DEFINE_PER_CPU(type, name)
- DECLARE_PER_CPU(type, name)
- per_cpu(name, cpu)
- __get_cpu_var(name)
- __raw_get_cpu_var(name)
- for_each_online_cpu(var)
  - But careful!!! CPUs can come and go...
  - get_online_cpus() and put_online_cpus()

The Existence Problem

The Existence Problem

The need for full partitioning suggests partitioning synchronization primitives, too!



http://www.usenix.org/events/osdi99/full_papers/gamsa/gamsa.pdf

Solution to the Existence Problem?

- Access

```
p = header;
spin_lock(&p->lock);
do_something_with(p);
spin_unlock(&p->lock);
```

- Deletion

```
p = header;
spin_lock(&p->lock);
if (header == p)
    header = NULL;
else
    p = NULL;
spin_unlock(&p->lock);
if (p != NULL)
    kfree(p);
```

If so, why? If not, why not, and what would be a solution?

Solution to the Existence Problem?

- Access

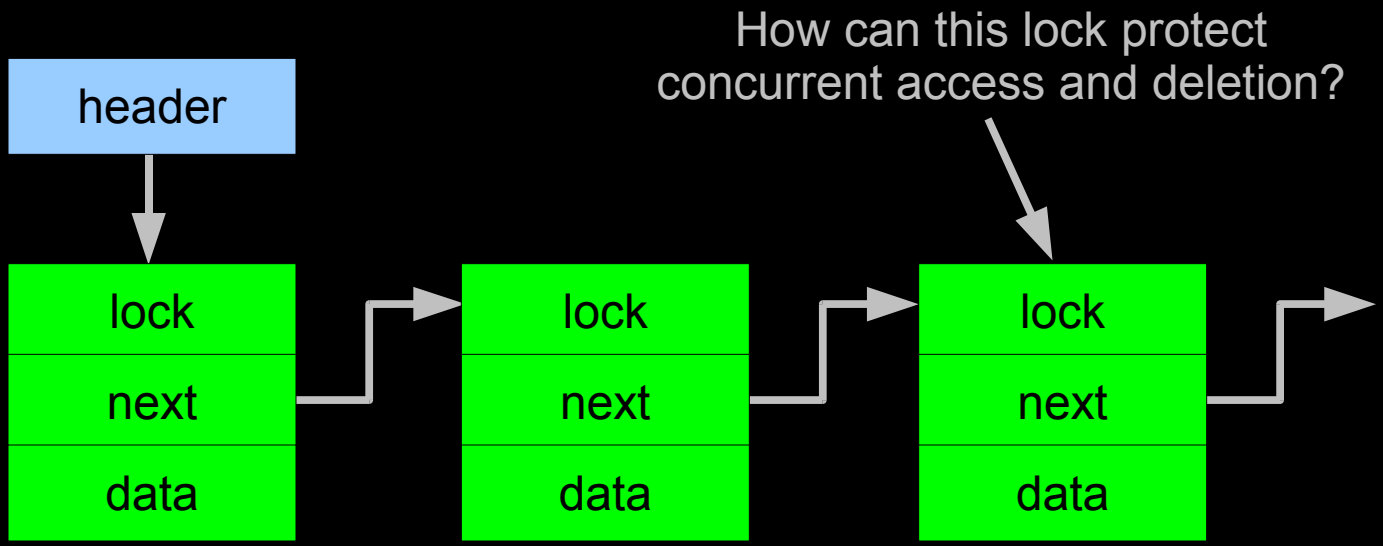
```
p = header;
if (p != NULL){
    spin_lock(&p->lock);
    do_something_with(p);
    spin_unlock(&p->lock);
}
```

- Deletion

```
q = p = header;
if (p != NULL) {
    spin_lock(&p->lock);
    if (header == p)
        header = NULL;
    else
        q = NULL;
    spin_unlock(&p->lock);
    if (q != NULL)
        kfree(q);
}
```

Never forget the NULL-pointer checks...

The Existence Problem Extended



Solutions to the Existence Problem

Solutions to the Existence Problem

- Single-threaded programs
- Locking chain extending back to “header”
  - Similarly, transaction covering back to “header”
- Global lock
- Hashed global array of locks
- Per-CPU global locks
- Garbage collector
- Type-safe memory
- Global reference count
- Others?

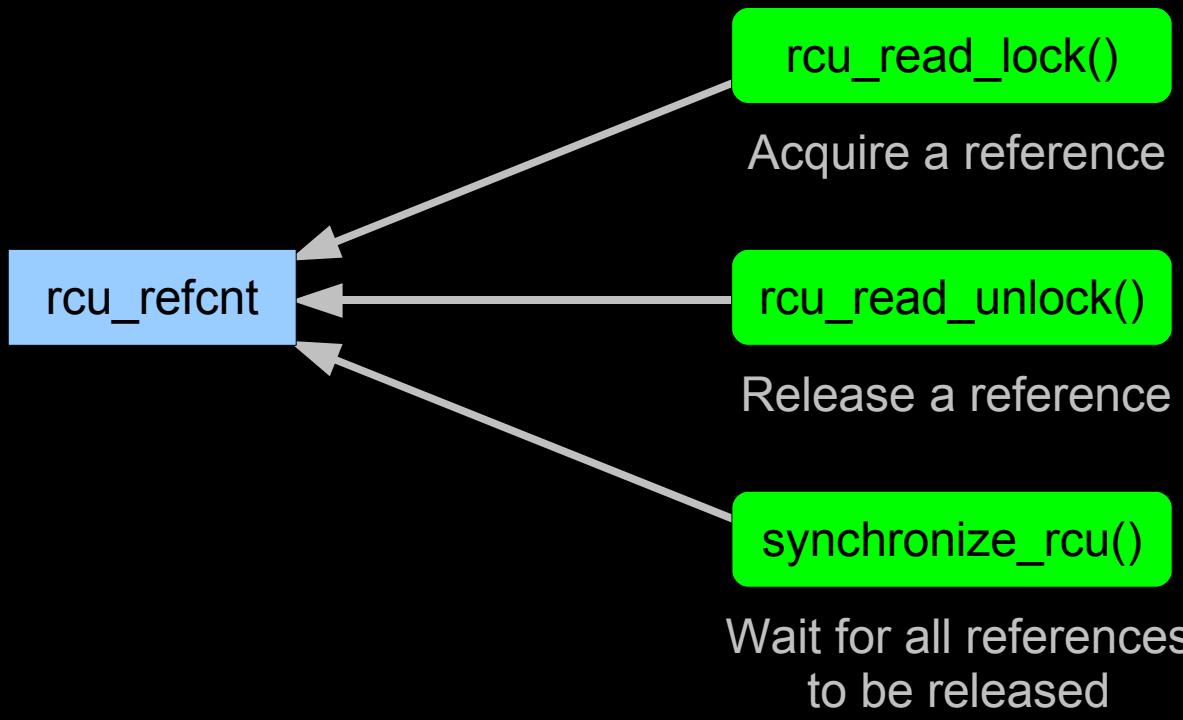
Solutions to the Existence Problem

- Single-threaded programs
  - Can be the right thing to do, but lose scalability
  - And existence can be a problem even in single-threaded code via signals, events, and callbacks
- Locking/TM chain extending back to “header”
  - High locking/STM overhead for nesting, increased probability of hitting HTM transaction-size limitations
  - Deadlock issues with locking
  - Increased probability of encompassing non-idempotent operation in both HTM and STM
- Global lock
  - Poor scalability and performance

Solutions to the Existence Problem

- Hashed global array of locks
  - Poor performance due to lack of memory locality
  - Deadlock issues, especially for large programs
    - Engineering solutions well-known but complex
- Per-CPU global locks
  - Requires possibly-awkward partitioning over CPUs
  - Deadlock issues, especially for large programs
    - Engineering solutions well-known but complex
- Type-safe memory
  - Complex code to detect and handle reallocation
- Garbage collector
  - Great if your environment provides one, but overhead and reclamation time can rule out GC

Global Reference Count



Solution to the Existence Problem?

- Access

```
rcu_read_lock();
p = rcu_dereference(header);
if (p != NULL)
    do_something_with(p);
rcu_read_unlock();
```

- Deletion

```
rcu_read_lock();
p = rcu_dereference(header);
q = p;
if (p != NULL) {
    spin_lock(&p->lock);
    if (header == p)
        header = NULL;
    else
        q = NULL;
    spin_unlock(&p->lock);
}
rcu_read_unlock();
if (q != NULL) {
    synchronize_rcu();
    kfree(q);
}
```

Reference acquired under rcu_read_lock() guaranteed to exist until rcu_read_unlock()

Global Reference Count

```
 1 atomic_t rcu_refcnt;
 2
 3 static void rcu_read_lock(void)
 4 {
 5   atomic_inc(&rcu_refcnt);
 6   smp_mb__after_atomic_inc();
 7 }
 8
 9 static void rcu_read_unlock(void)
10 {
11   smp_mb__before_atomic_dec();
12   atomic_dec(&rcu_refcnt);
13 }
14
15 void synchronize_rcu(void)
16 {
17   smp_mb();
18   while (atomic_read(&rcu_refcnt) != 0) {
19     poll(NULL, 0, 10);
20   }
21   smp_mb();
22 }
```

Reference counter

Acquire reference

Release reference

Wait for references to be released

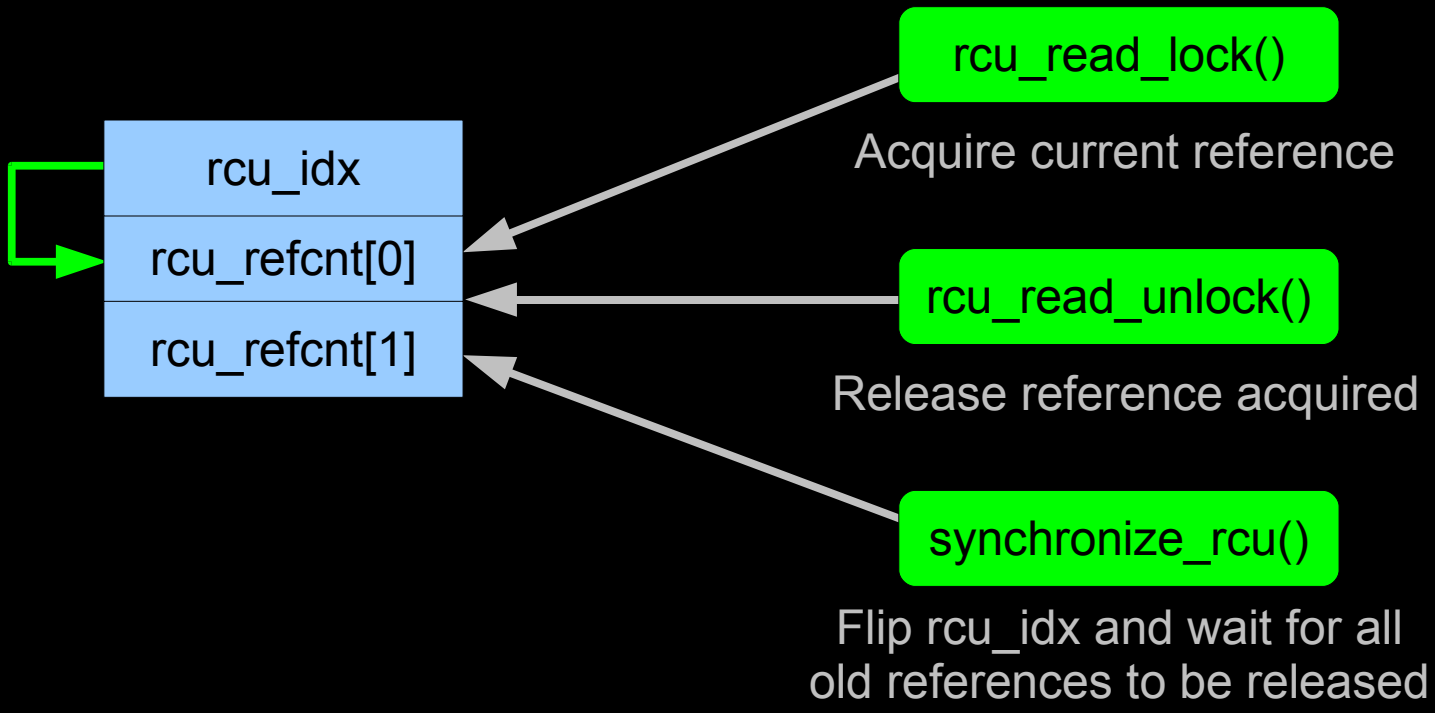
Does this work? Why or why not?

Global Reference Count Issues

- Updater starvation!!! (Why?)
- Horrible read-side performance under heavy contention (40us on 64-CPU Power-5 system)
- Mediocre read-side performance under light contention (100ns on Power-5 system)
- Extremely fast updates: 40 nanoseconds
  - But only in absence of readers

Two thumbs down!!!

Global Reference Count Pair Data



Global Reference Count Pair Data

```
1 atomic_t rcu_refcnt[2];
2 atomic_t rcu_idx;
3 DEFINE_SPINLOCK(rcu_gp_lock);
4 DEFINE_PER_CPU(int, rcu_nesting);
5 DEFINE_PER_CPU(int, rcu_read_idx);
```

Global Reference Count Pair Reader Primitives

```
 1 static void rcu_read_lock(void)
 2 {
 3   int i;
 4   int n;
 5
 6   n = __get_cpu_var(rcu_nesting);
 7   if (n == 0) {
 8     i = atomic_read(&rcu_idx);
 9     __get_cpu_var(rcu_read_idx) = i;
10     atomic_inc(&rcu_refcnt[i]);
11   }
12   __get_cpu_var(rcu_nesting) = n + 1;
13   smp_mb();
14 }
15
16 static void rcu_read_unlock(void)
17 {
18   int i;
19   int n;
20
21   smp_mb();
22   n = __get_cpu_var(rcu_nesting);
23   if (n == 1) {
24     i = __get_cpu_var(rcu_read_idx);
25     atomic_dec(&rcu_refcnt[i]);
26   }
27   __get_cpu_var(rcu_nesting) = n - 1;
28 }
```

Acquire reference

Release reference

Global Reference Count Pair Updater Primitives

```
 1 void synchronize_rcu(void)
 2 {
 3   int i;
 4
 5   smp_mb();
 6   spin_lock(&rcu_gp_lock);
 7   i = atomic_read(&rcu_idx);
 8   atomic_set(&rcu_idx, !i);
 9   smp_mb();
10   while (atomic_read(&rcu_refcnt[i]) != 0) {
11     poll(NULL, 0, 10);
12   }
13   spin_unlock(&rcu_gp_lock);
14   smp_mb();
15 }
```

Wait for references to be released

Does this work? Why or why not?

Issues With Global Reference Count Pair

- Buggy!!!
  - CPU 0 rcu_read_lock() line 8: i = rcu_idx == 0
  - CPU 1 invokes synchronize_rcu()
    - Now rcu_idx == 1
  - CPU 0 rcu_read_lock() line 9: atomically increments rcu_refcnt[0], enters read-side critical section, acquires a reference to some data element
  - CPU 1 removes that same data element
  - CPU 1 invokes synchronize_rcu() again:
    - Line 7 fetches i = rcu_idx == 1
    - Line 8 sets rcu_idx = 0
    - Lines 10-11 wait for rcu_refcnt[1] to go to zero
  - CPU 1 kfree()s the data element
    - While CPU 0 is still using it!!!

Issues With Global Reference Count Pair

- **Buggy!!!**
  - CPU 0 rcu_read_lock() line 8: i = rcu_idx == 0
  - CPU 1 invokes synchronize_rcu()
    - Now rcu_idx == 1
  - CPU 0 rcu_read_lock() line 9: atomically increments rcu_refcnt[0], enters read-side critical section, acquires a reference to some data element
  - CPU 1 removes that same data element
  - CPU 1 invokes synchronize_rcu() again:
    - Line 7 fetches i = rcu_idx == 1
    - Line 8 sets rcu_idx = 0
    - Lines 10-11 wait for rcu_refcnt[1] to go to zero
  - CPU 1 kfree()s the data element
    - **While CPU 0 is still using it!!!**

Everyone who has attempted an implementation has committed this bug!

Global Reference Count Pair Updater Primitives

```
 1 void synchronize_rcu(void)
 2 {
 3   int i;
 4
 5   smp_mb();
 6   spin_lock(&rcu_gp_lock);
 7   i = atomic_read(&rcu_idx);
 8   atomic_set(&rcu_idx, !i);
 9   smp_mb();
10   while (atomic_read(&rcu_refcnt[i]) != 0) {
11     poll(NULL, 0, 10);
12   }
13   smp_mb();
14   atomic_set(&rcu_idx, i);
15   smp_mb();
16   while (atomic_read(&rcu_refcnt[!i]) != 0) {
17     poll(NULL, 0, 10);
18   }
19   spin_unlock(&rcu_gp_lock);
20   smp_mb();
21 }
```

Wait for references to be released

Flip counter twice

Does this work? Why or why not?

Global Reference Count Pair Issues

- Horrible read-side performance under heavy contention (40us on 64-CPU Power-5 system)
- Double counter flip and update-side lock slow: 200ns in isolation, 40us on 64-CPU Power 5)
  - And no more concurrent updates!!!
- Mediocre-to-poor read-side performance under light contention (150ns on Power-5 system)
- No updater starvation

One thumb down!!!

Per-CPU Reference Count?

- Partitioning for performance and scalability
- But threads (tasks) take references, not CPUs!
  - So, disable preemption while holding reference
    - We *are* executing in the kernel, after all!!!
  - As is currently the case when holding spinlocks
    - And when holding "raw" spinlocks in PREEMPT_RT
- But if we are going to disable preemption...
  - The fact that the task is running on a given CPU *is* the reference!!!
  - Each context switch then implicitly releases the outgoing task's reference ...
  - ... and acquires the incoming task's reference

"Running on CPU" as Reference

```
 1 static void rcu_read_lock()
 2 {
 3   preempt_disable();  /* no-op for !PREEMPT */
 4 }
 5
 6 static void rcu_read_unlock()
 7 {
 8   preempt_enable();  /* no-op for !PREEMPT */
 9 }
10
11 void synchronize_rcu()
12 {
13   int cpu;
14
15   for_each_online_cpu(cpu)
16     run_on(cpu);
17 }
```

Acquire reference

Release reference

Wait for references to be released

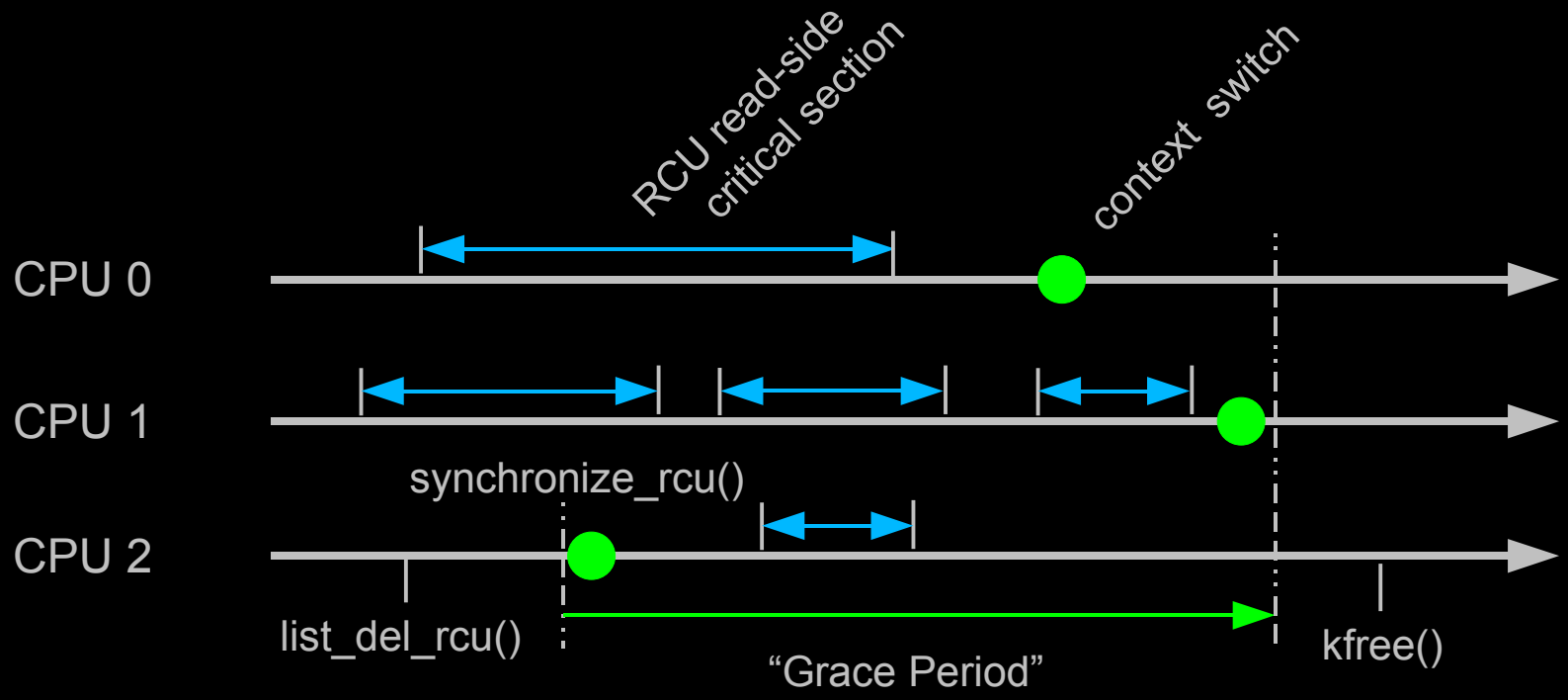
Does this work? Why or why not?
Can rcu_read_lock() participate in a deadlock cycle?

"Running on CPU" as Reference Evaluation

- Excellent read-side scalability and performance: "free" is a *very* good price!!!
- Scheduling on each CPU in turn works well for small numbers of CPUs, but does not scale well
  - Actual Linux-kernel implementation uses batching to achieve excellent update-side scalability
  - But is considerably more complex
- No updater starvation

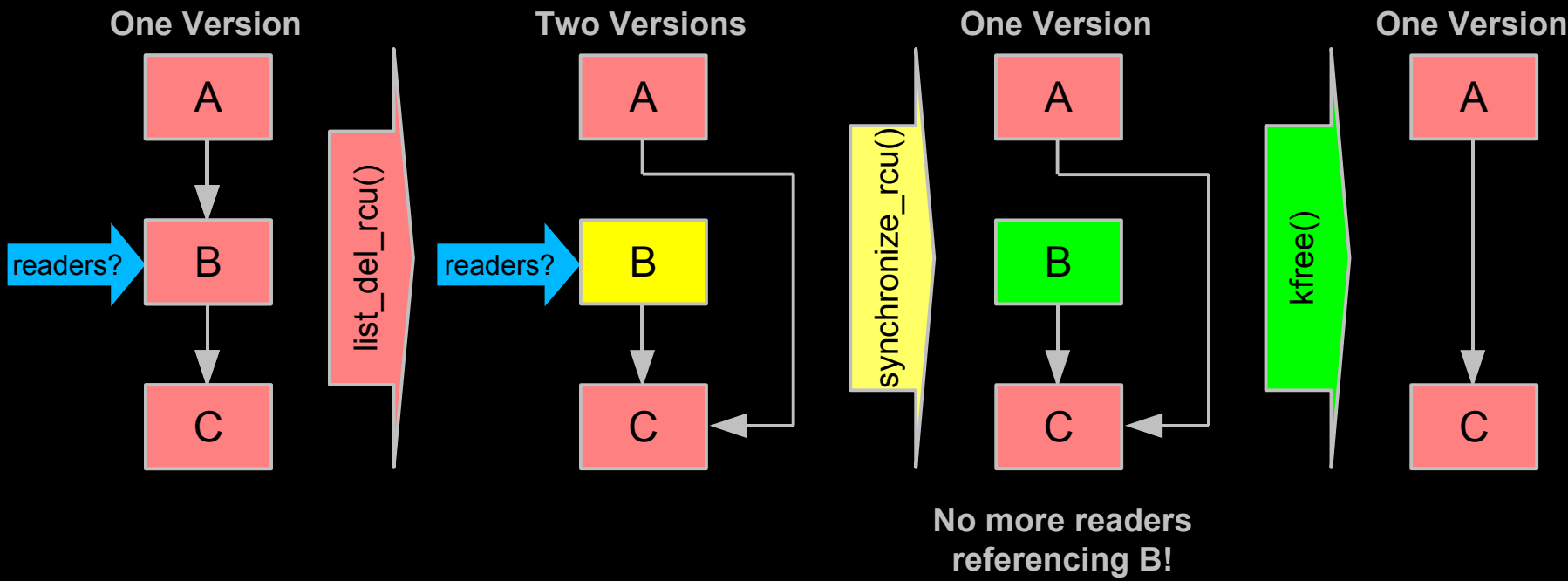
Two thumbs up!!!

“Running on CPU” as Reference: Schematic



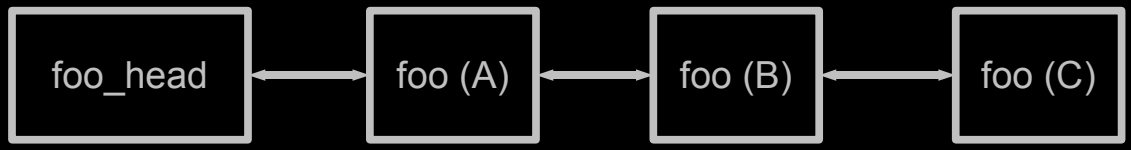
"Running on CPU" as Reference: Use Case

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



"Running on CPU" as Reference: Use Case

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

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

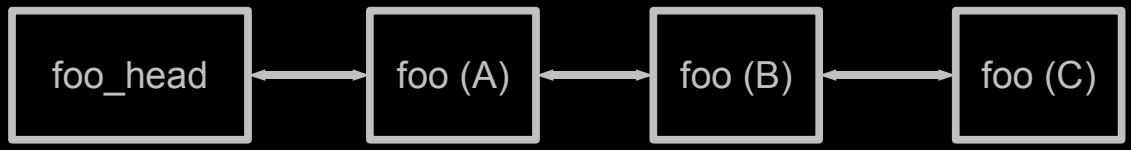



```
int search(struct foo_head *fhp, int k)
{
   struct foo *p;
   struct list_head *head = &fhp->list;

   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;
}
```

```
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;
}
```

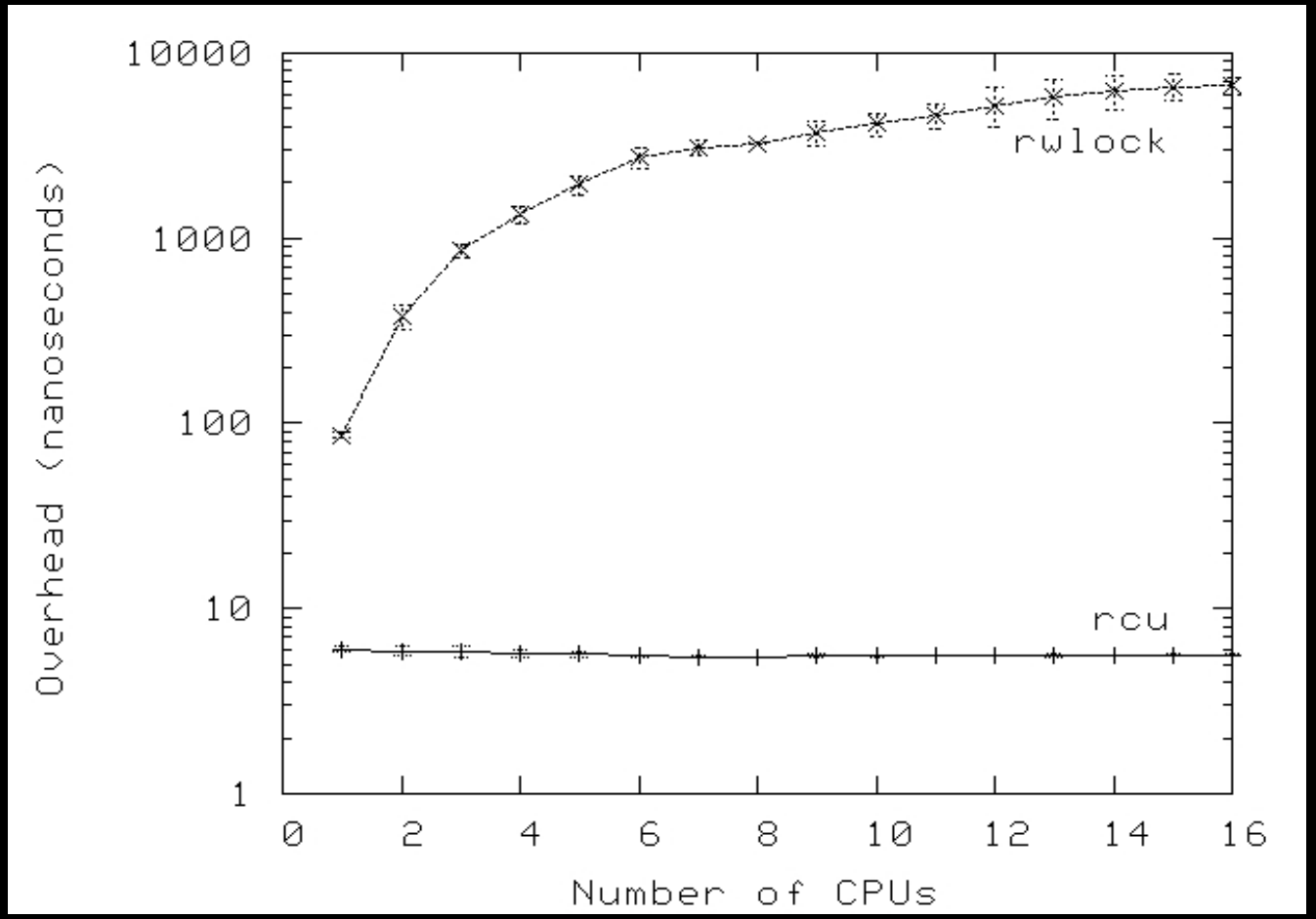
"Running on CPU" as Reference: rwlock

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




```
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(p);
         write_unlock(&fhp->mutex);
         /* synchronize_rcu(); */
         kfree(p);
         return 1;
      }
   }
   write_unlock(&fhp->mutex);
   return 0;
}
```

RCU vs. Reader-Writer Locking Performance



CONFIG_PREEMPT kernel build

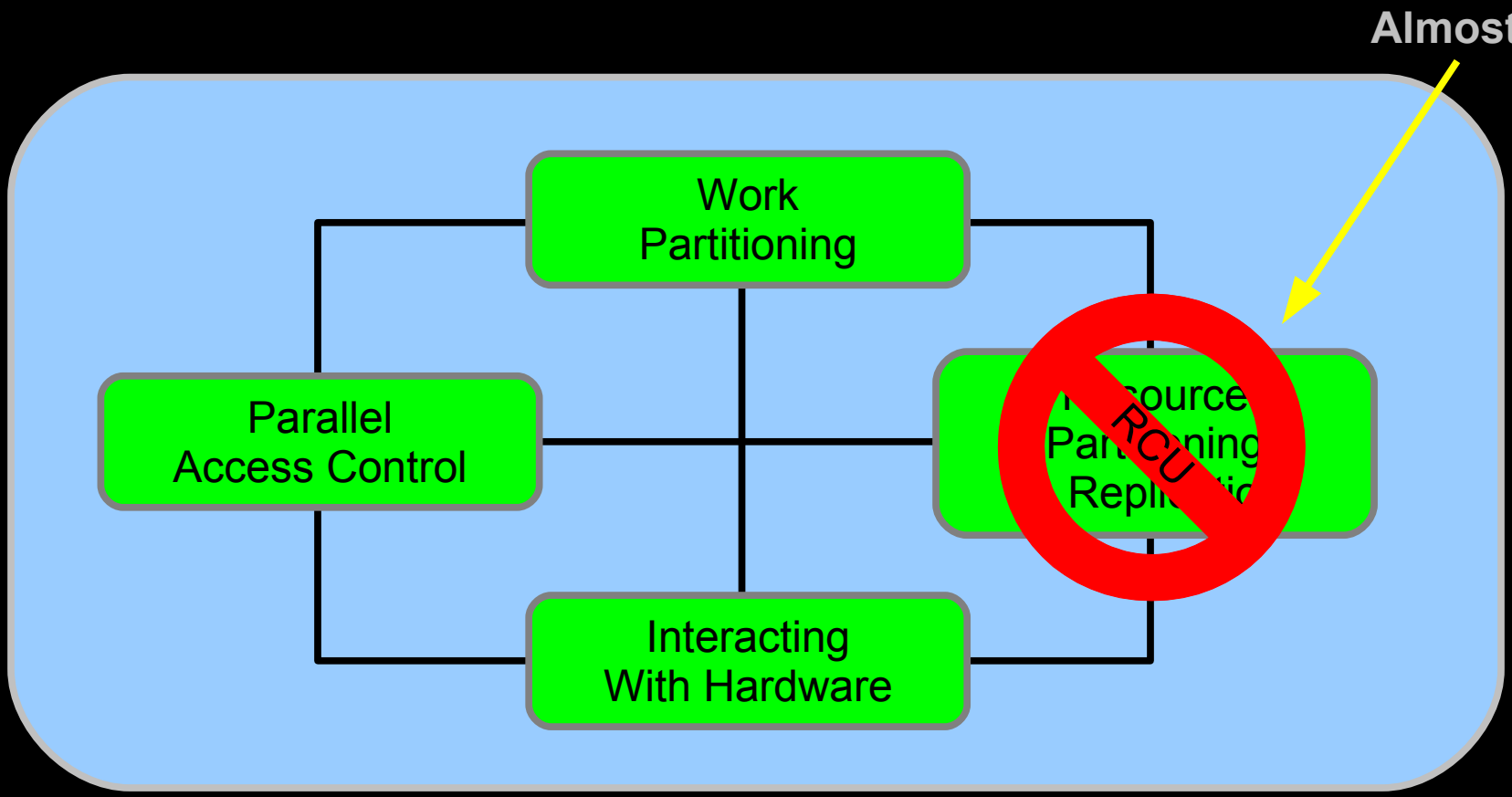
What is RCU???

- RCU is a:
  - reader-writer lock replacement
  - restricted reference-counting mechanism
  - bulk reference-counting mechanism
  - poor-man's garbage collector
  - way of providing existence guarantees
  - way of waiting for things to finish
- Use RCU in:
  - read-mostly situations or
  - for deterministic response from read-side primitives and from asynchronous update-side primitives

RCU as a Solution to the Existence Problem

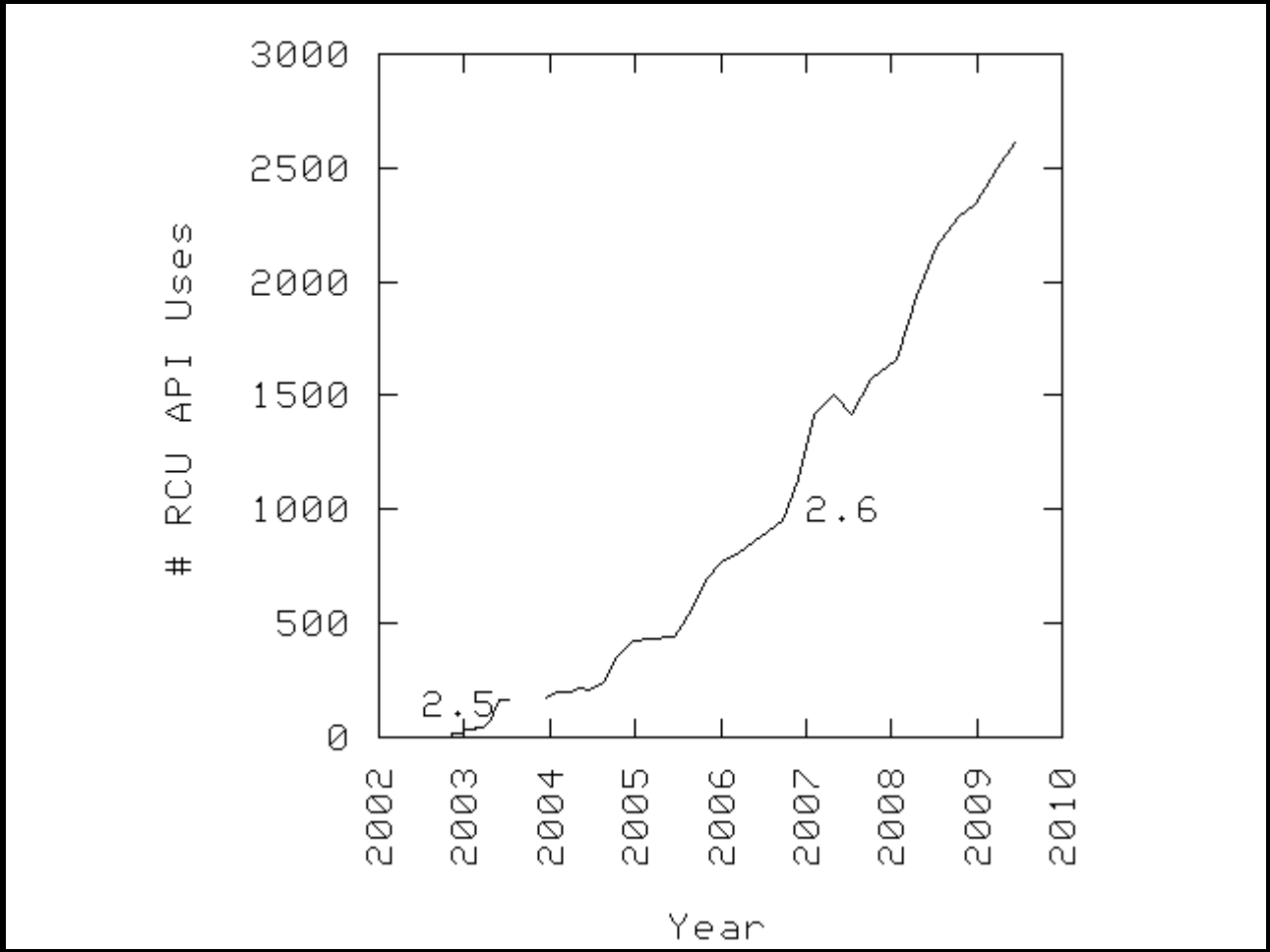
- For read-mostly data structures, RCU provides the benefits of the data-parallel model
  - But without the need to actually partition or replicate the RCU-protected data structures
  - Readers access data without needing to exclude each others or updates
    - Extremely lightweight read-side primitives
- And RCU provides additional read-side performance and scalability benefits
  - With a few limitations and restrictions....

RCU for Read-Mostly Data Structures

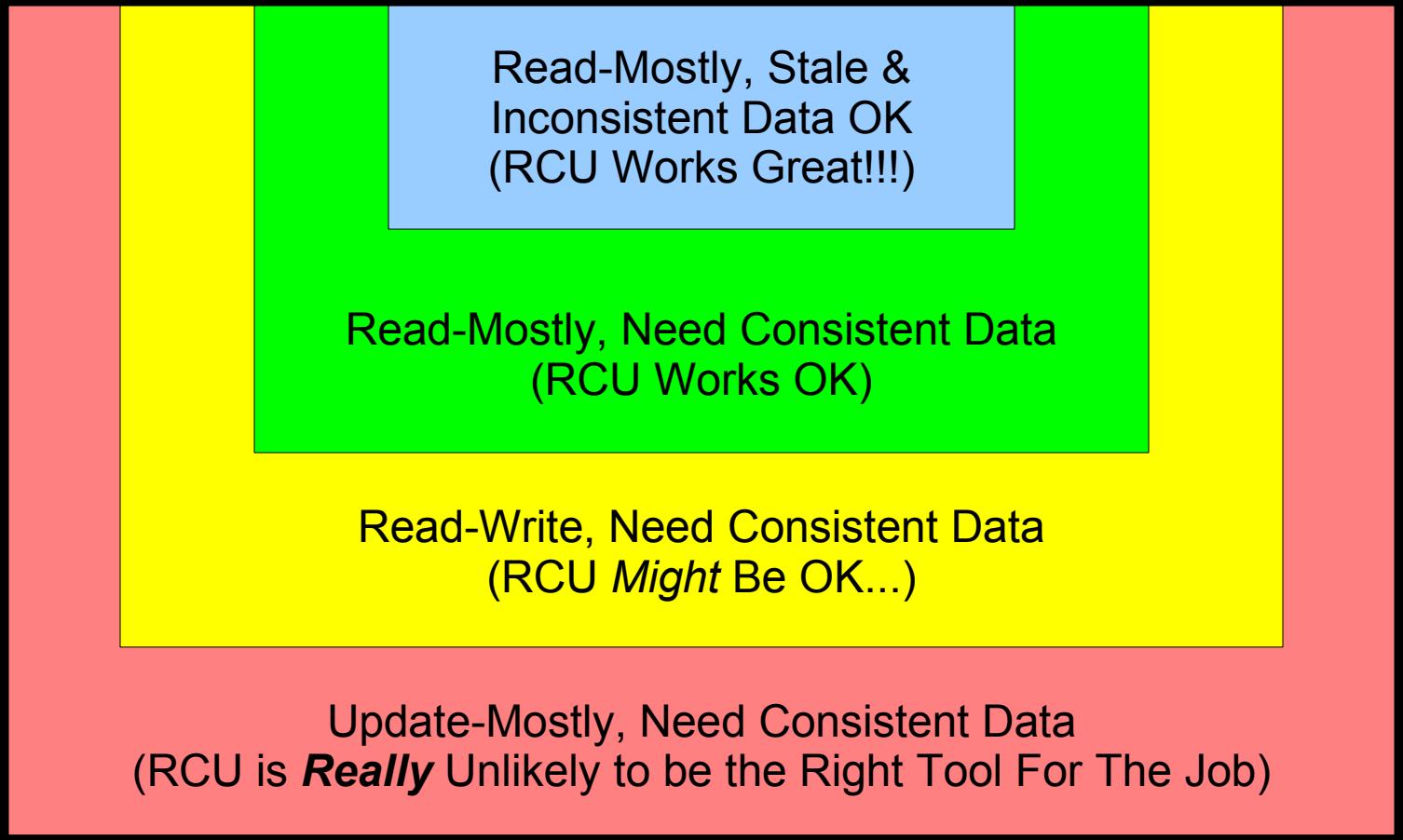


RCU data-parallel approach: ~~first partition resources~~, then partition work, and only then worry about parallel access control, and only for updates.

RCU Usage in the Linux Kernel



RCU Area of Applicability



Summary

- Programming Environments in Linux Kernel
- Synchronization Primitives
- Per-CPU Variables
- The Existence Problem
- Solutions to the Existence Problem

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.
- This material is based upon work supported by the National Science Foundation under Grant No. CNS-0719851.
  - Joint work with Manish Gupta, Maged Michael, Phil Howard, Joshua Triplett, and Jonathan Walpole

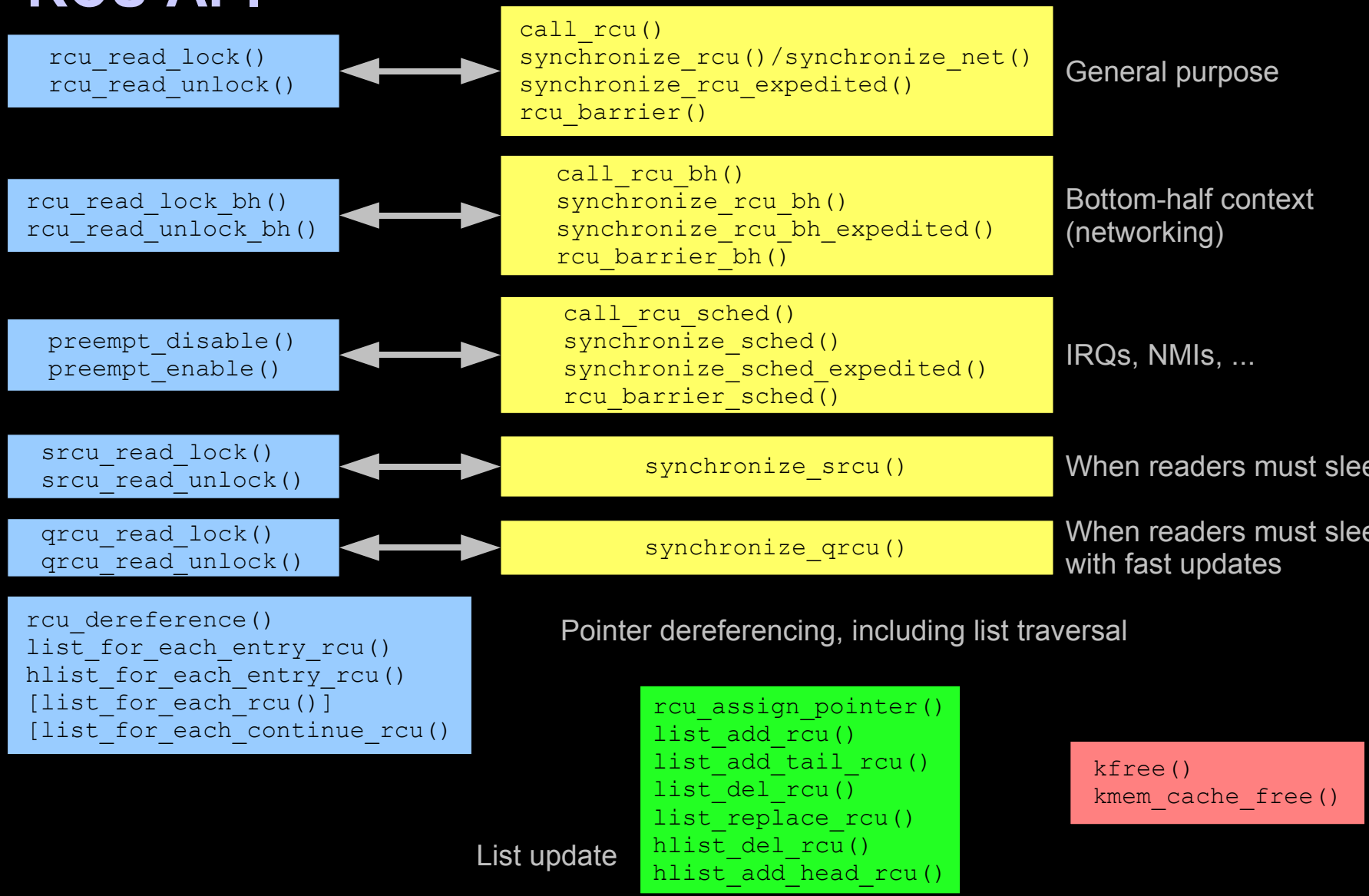
Questions?

To probe further:

- Linux Device Drivers, 3rd edition, J. Corbet, A. Rubini, G. Kroah-Hartman
- Linux Kernel Development, 2nd edition, Robert Love
- Linux Weekly News: lwn.net (Google for "whatever site:lwn.net")
- Linux Kernel source (http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.30.tar.bz2)
  - "Documentation" directory
- http://lwn.net/Articles/262464/ (What is RCU, Fundamentally?)
- http://lwn.net/Articles/263130/ (What is RCU's Usage?)
- http://lwn.net/Articles/264090/ (What is RCU's API?)
- http://www.rdrop.com/users/paulmck/RCU/lockperf.2004.01.17a.pdf
  - linux.conf.au paper comparing RCU vs. locking performance
- 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://www.rdrop.com/users/paulmck/RCU/ (More RCU information)

RCU API

RCU API

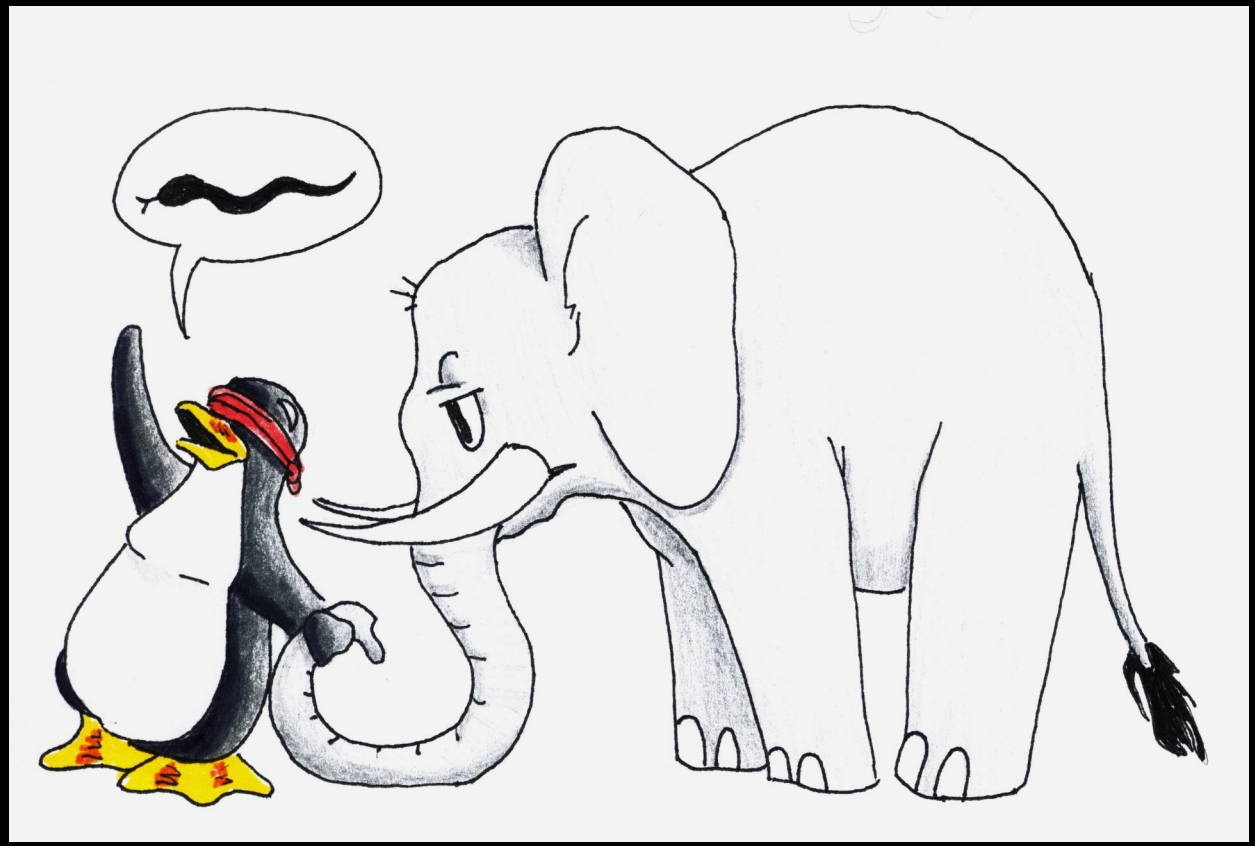


Why Free and Open-Source Software?

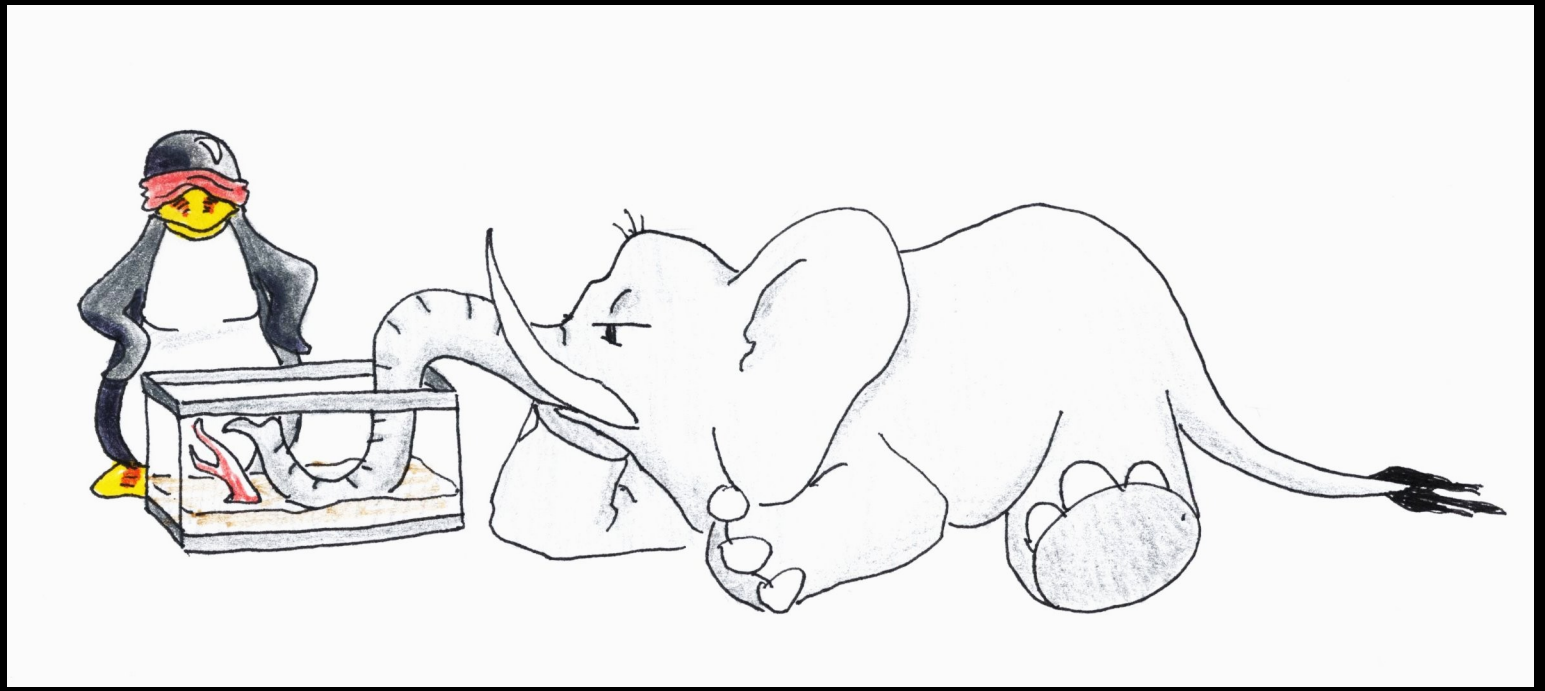
Why Free and Open-Source Software?

The Parable of
The Six Blind Penguins and the Elephant

Proprietary Programming: Requirements



Proprietary Programming: “Solution”



But sooner or later...

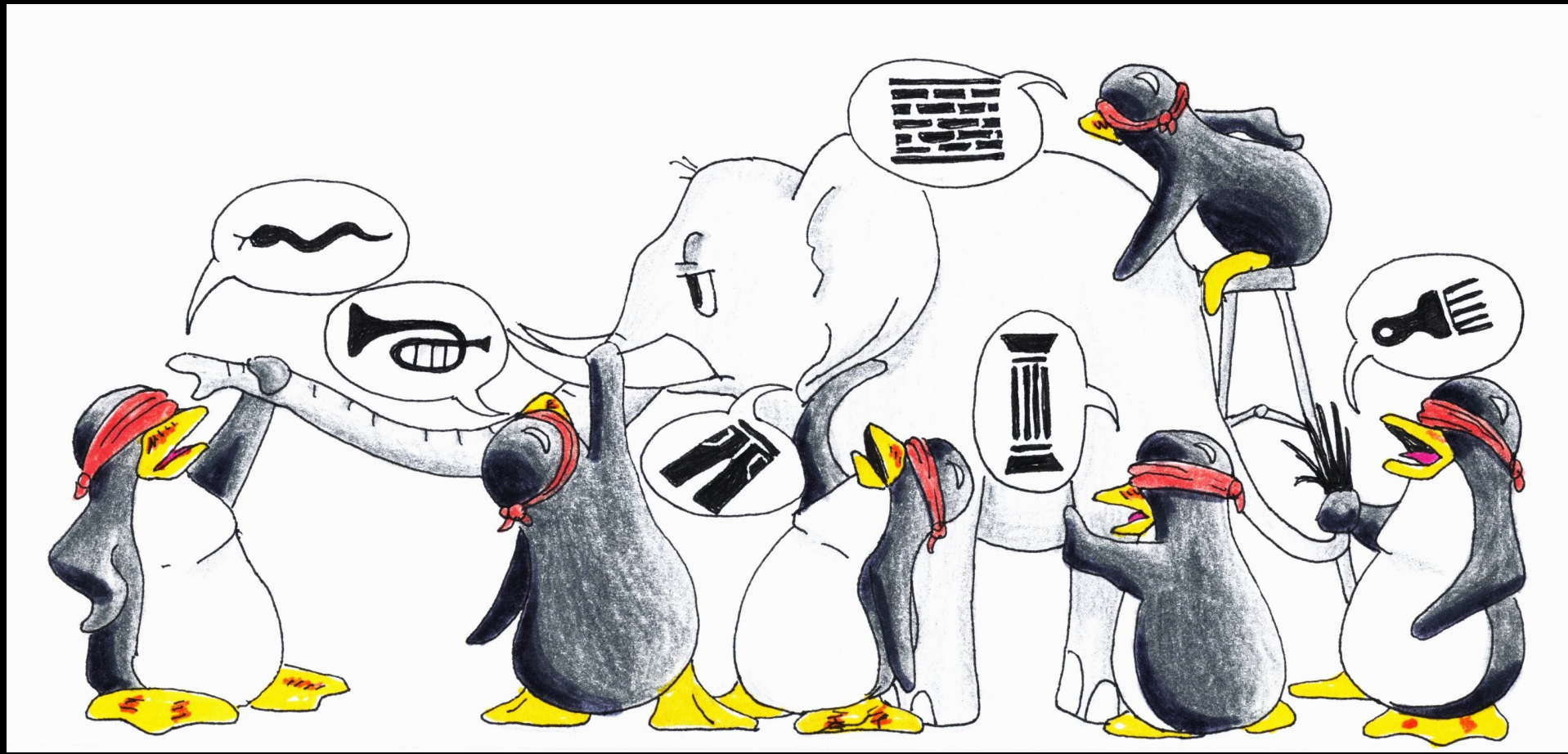
Example: DYNIX/ptx RCU Implementation

- In late 1990s, knew everything there was to know about RCU:
  - rcu_read_lock()
  - rcu_read_unlock()
  - call_rcu()
  - kfree()
  - kmem_cache_free()
  - kmem_deferred_free()
- But DYNIX/ptx was only a database server...

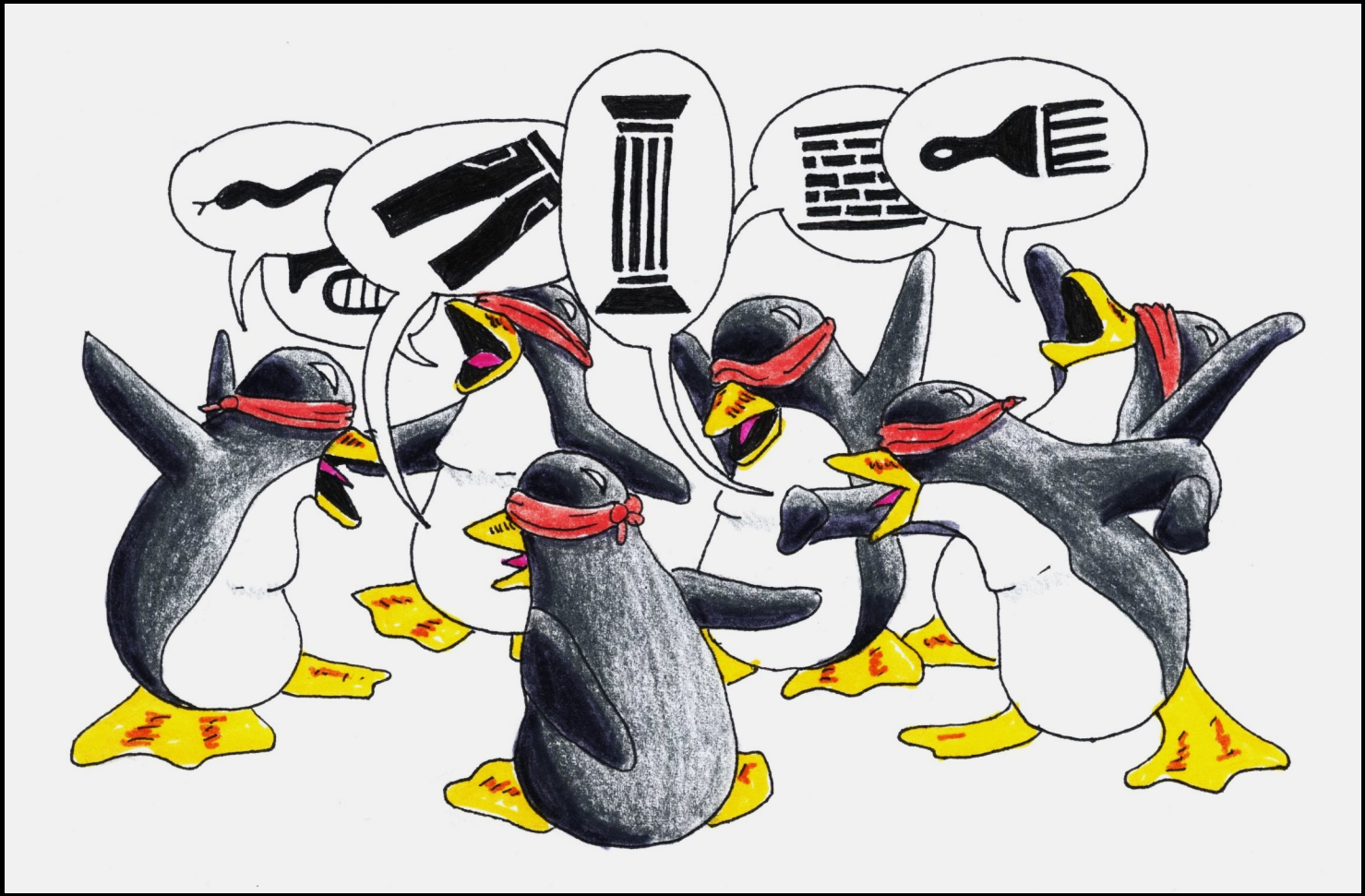
The Entire Elephant Will Make Itself Known...

Which it did for RCU, but in Linux

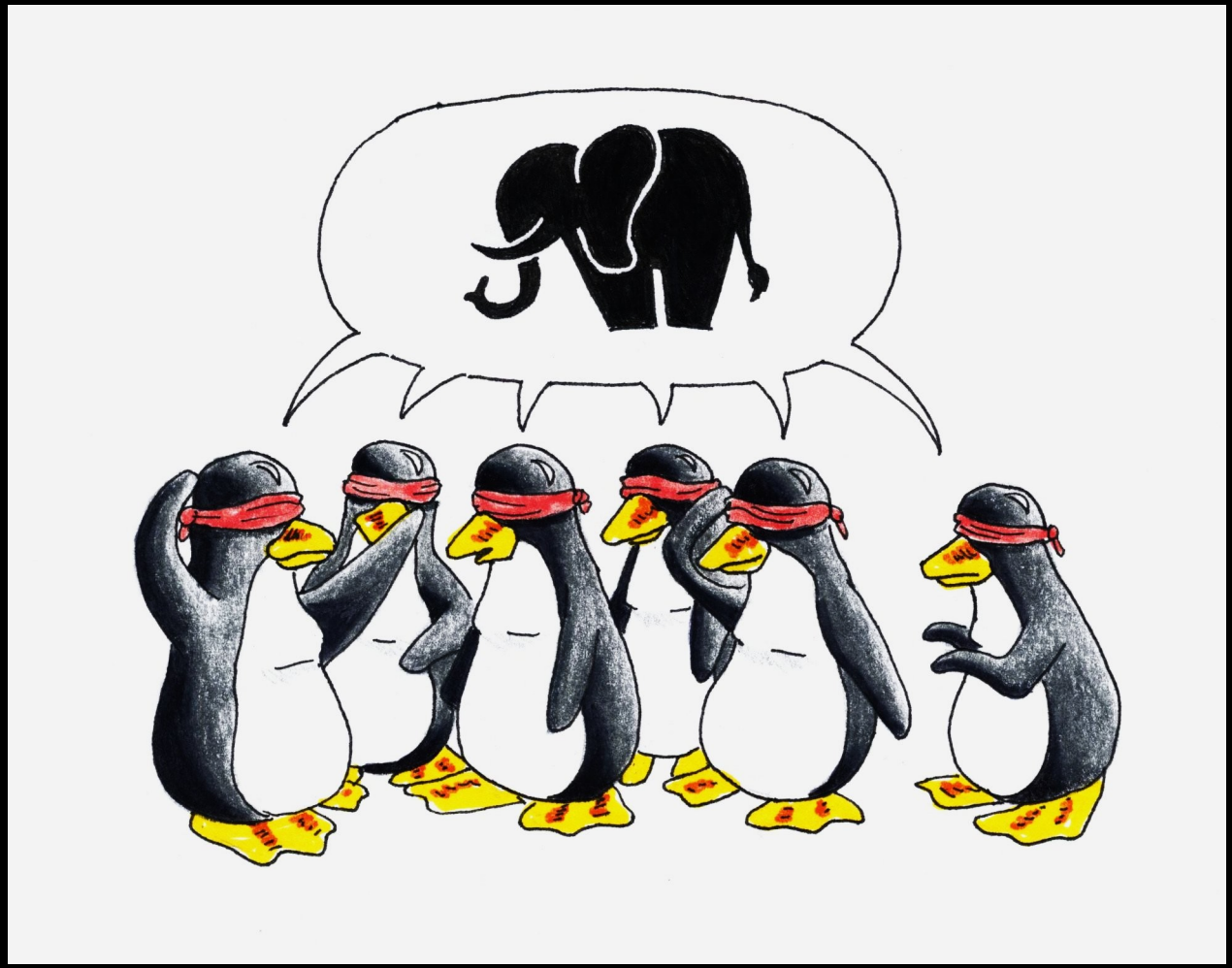
FOSS Programming: Requirements



Just Another Day on LKML...



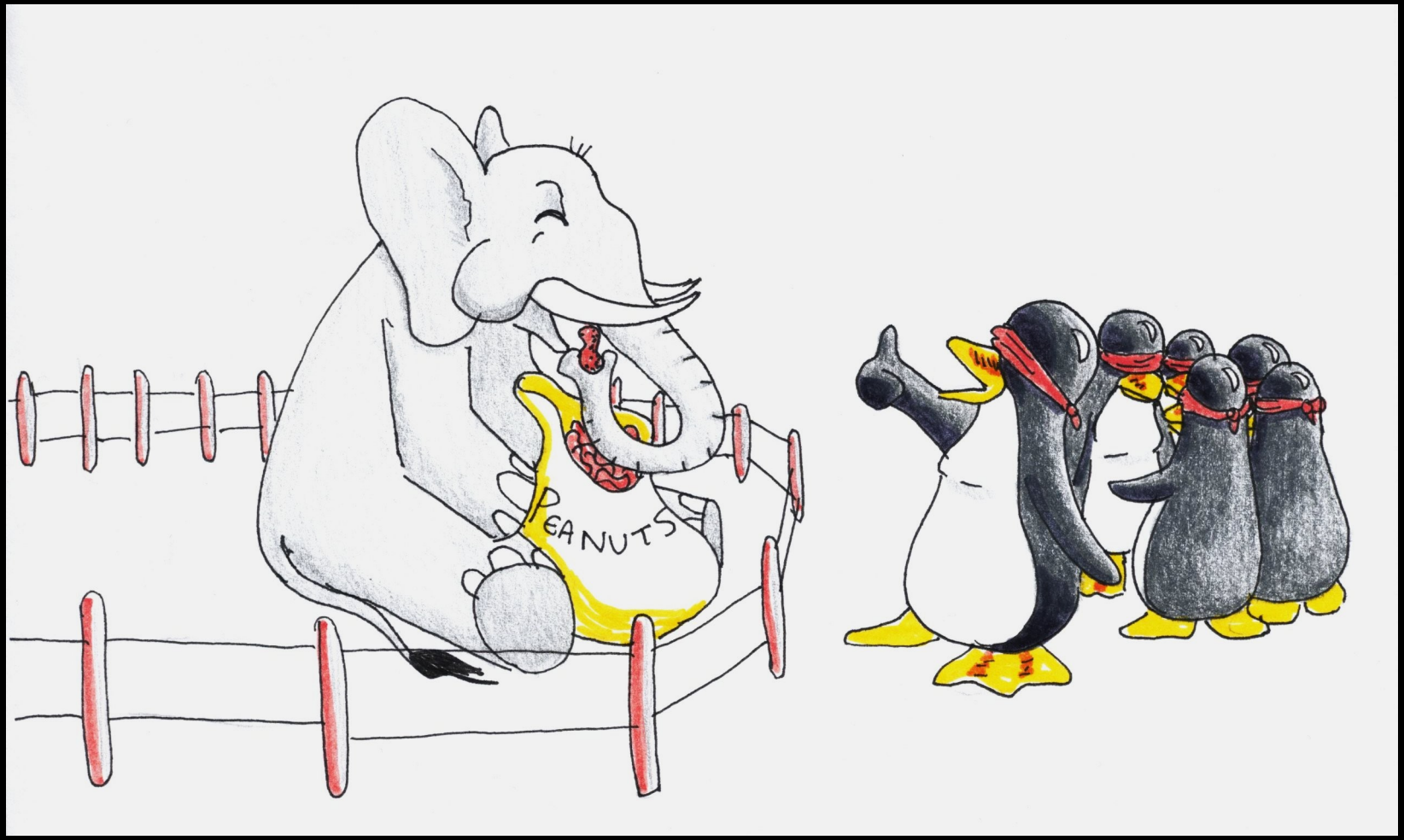
But Sometimes Consensus is Achieved



Linux Community Taught Me Much About RCU

- Simplicity as a first-class requirement
- Support for 20+ CPU families
  - Must hide memory barriers from RCU users
- Real-time response: tens of microseconds
- Run on small-memory systems (2MB!!!)
- Heavy networking workloads
- Denial-of-service attacks
- Wait for interrupt- and NMI-handler completion
- Unloadable kernel modules
- Blocking in RCU read-side critical sections
- Sophisticated RCU-based list/tree manipulation
- Deep sub-millisecond RCU grace periods

And an Appropriate Solution Produced Thereby



This is RCU

- rcu_read_lock()
- rcu_read_unlock()
- call_rcu()
- kfree()
- kmem_cache_free()

This is RCU in Linux

- rcu_read_lock()
- rcu_read_unlock()
- rcu_read_lock_bh()
- rcu_read_unlock_bh()
- preempt_disable()
- preempt_enable()
- srcu_read_lock()
- srcu_read_unlock()
- rcu_dereference()
- list_for_each_entry_rcu()
- hlist_for_each_entry_rcu()
- synchronize_rcu()
- synchronize_net()
- call_rcu()
- rcu_barrier()
- call_rcu_bh()
- synchronize_sched()
- synchronize_srcu()
- rcu_assign_pointer()
- list_add_rcu()
- list_add_tail_rcu()
- list_del_rcu()
- list_replace_rcu()
- hlist_del_rcu()
- hlist_add_after_rcu()
- hlist_add_before_rcu()
- hlist_add_head_rcu()
- hlist_replace_rcu()
- list_splice_init_rcu()
- kfree()
- kmem_cache_free()

This is RCU in Linux

- rcu_read_lock()
- rcu_read_unlock()
- rcu_read_lock_bh()
- rcu_read_unlock_bh()
- preempt_disable()
- preempt_enable()
- srcu_read_lock()
- srcu_read_unlock()
- rcu_dereference()
- list_for_each_entry_rcu()
- hlist_for_each_entry_rcu()
- synchronize_rcu()
- synchronize_net()
- call_rcu()
- rcu_barrier()
- call_rcu_bh()
- synchronize_sched()
- synchronize_srcu()
- rcu_assign_pointer()
- list_add_rcu()
- list_add_tail_rcu()
- list_del_rcu()
- list_replace_rcu()
- hlist_del_rcu()
- hlist_add_after_rcu()
- hlist_add_before_rcu()
- hlist_add_head_rcu()
- hlist_replace_rcu()
- list_splice_init_rcu()
- kfree()
- kmem_cache_free()

Any Questions?