Using a Malicious User-Level RCU to Torture RCU-Based Algorithms

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

January 22, 2009

Overview

- Why Concurrency?
- Hardware Issues with Concurrency
- RCU Fundamentals
- RCU Requirements
- Challenges for User-Level RCU
- A Pair of User-Level RCU Implementations
- Future Work and Summary

Why Concurrency?

- **Higher performance (otherwise do sequential!)**
- **Acceptable productivity (machines now cheap)**
- **Reasonable generality (amortize development cost)**

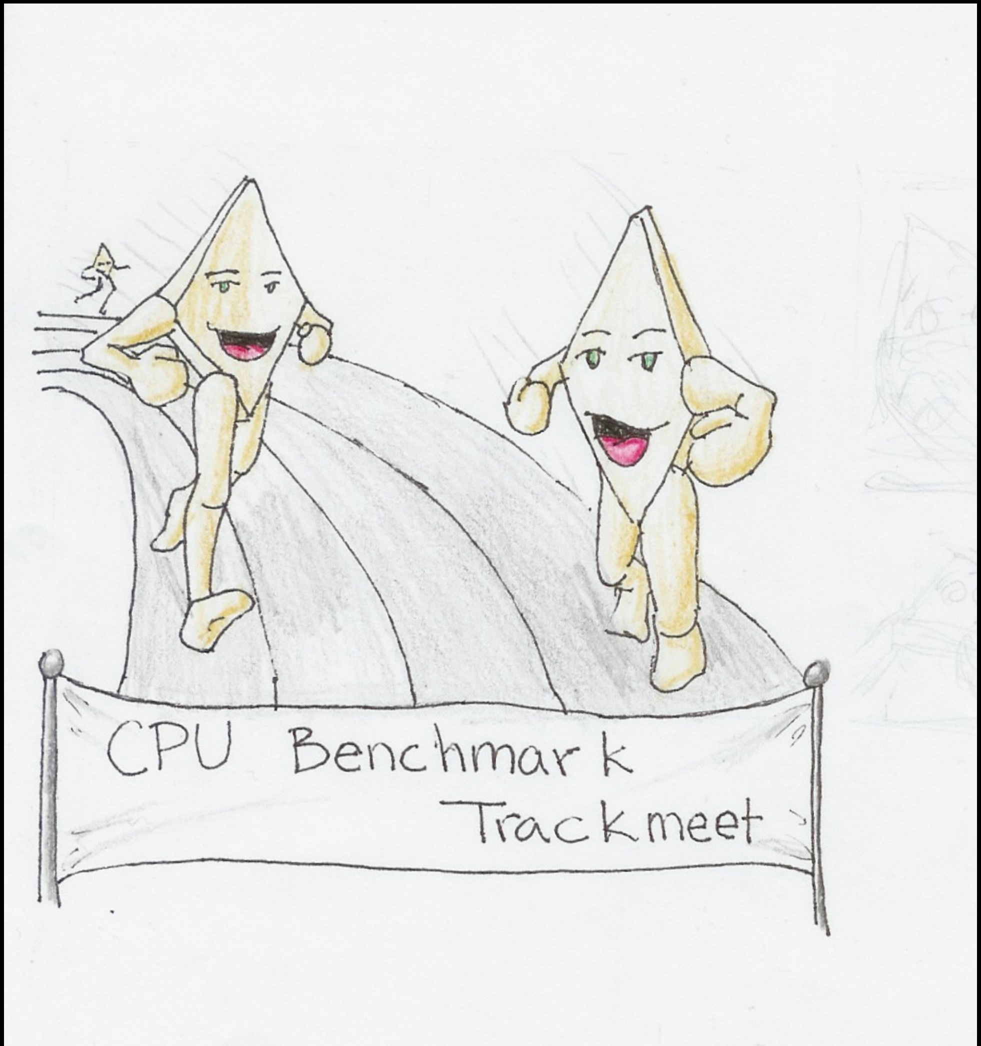
- **Or because it is fun!!!**
  - (Though your manager/professor/SO/whatever might have a different opinion on this point...)

- **Software reliability goes without saying, aside from this self-referential bullet point**
  - If it doesn't have to be reliable: “return 0;” is simple and fast

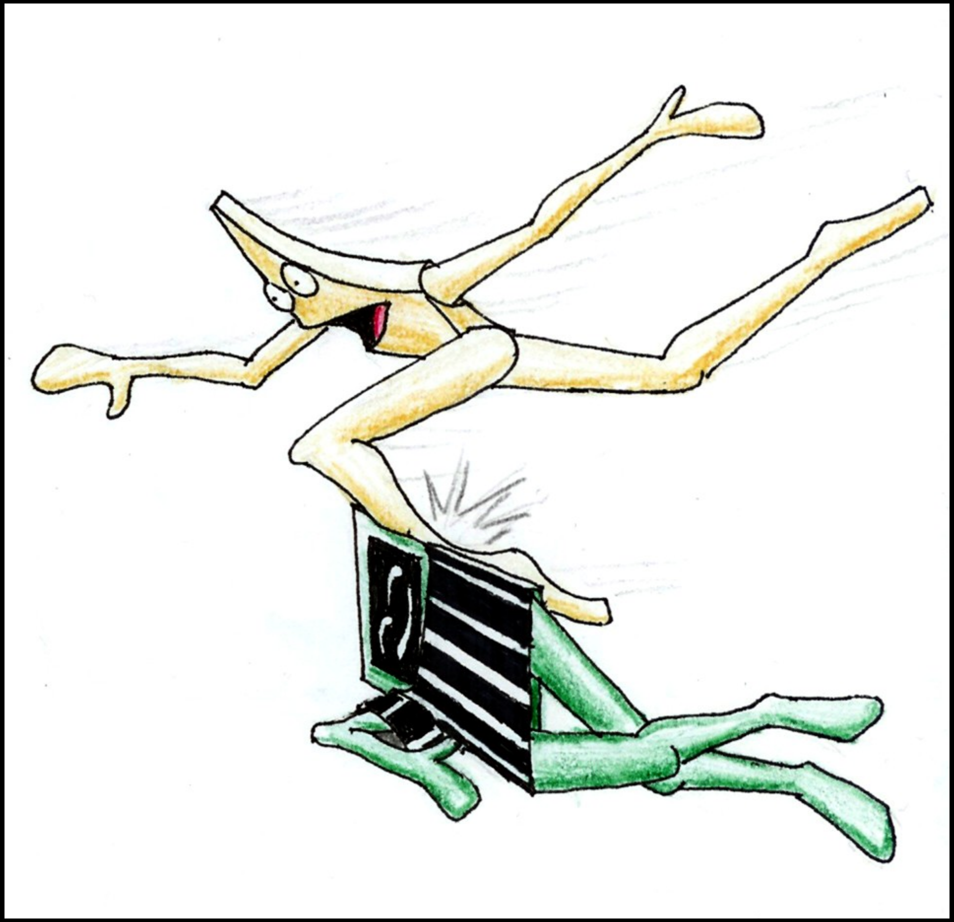
Concurrency Problem #1: Poor Performance

- **This is a severe problem in cases where performance was the only reason to exploit concurrency...**
- **Lots of effort, little (or no) result**
- **Why???**

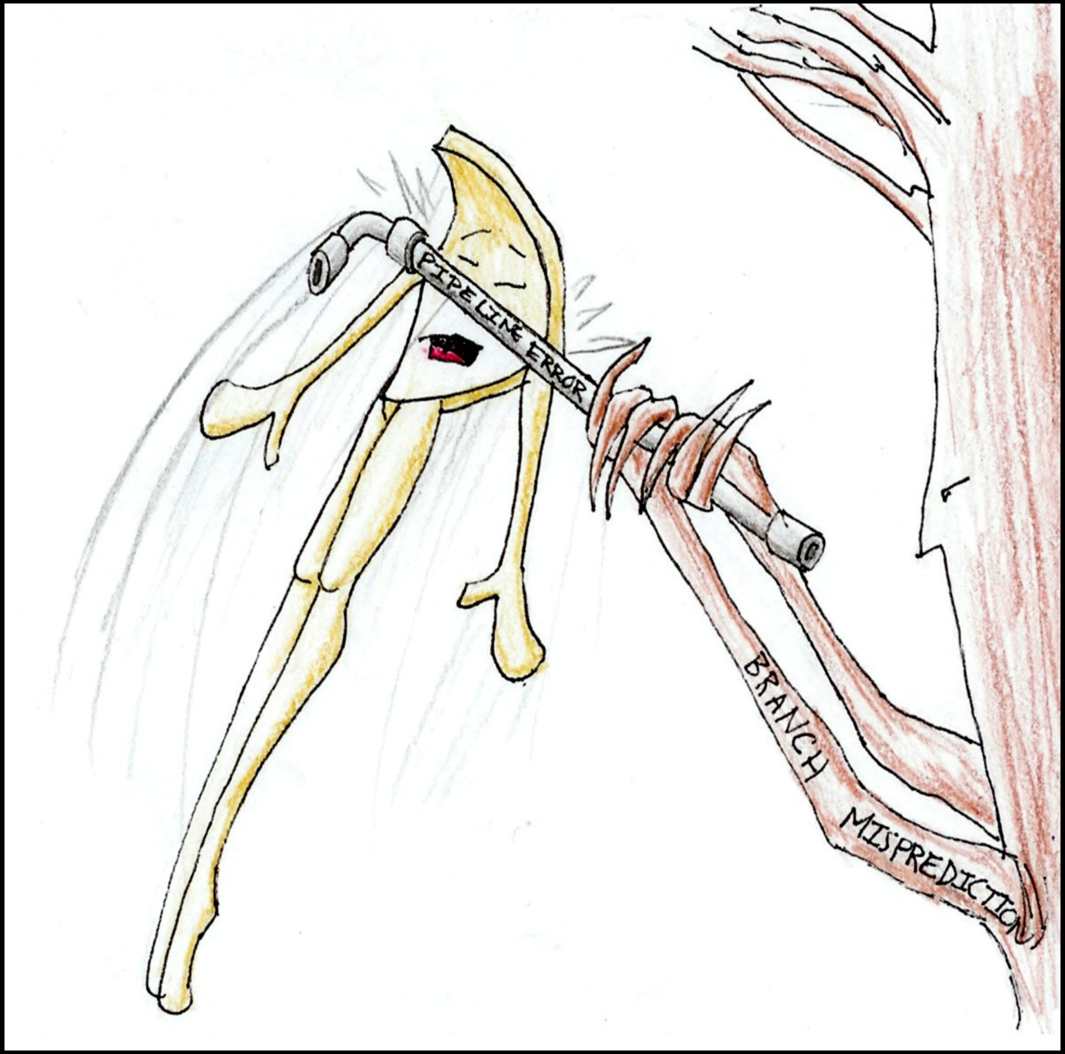
CPU Performance: The Marketing Pitch



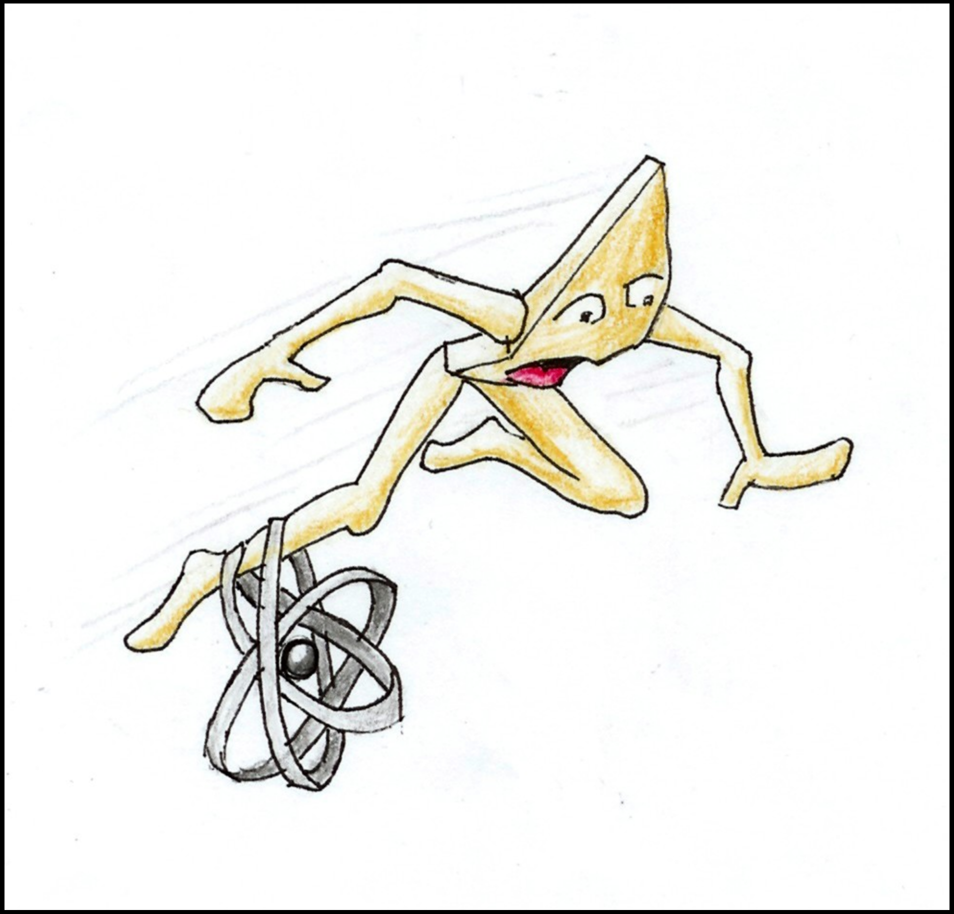
CPU Performance: Memory References



CPU Performance: Pipeline Flushes

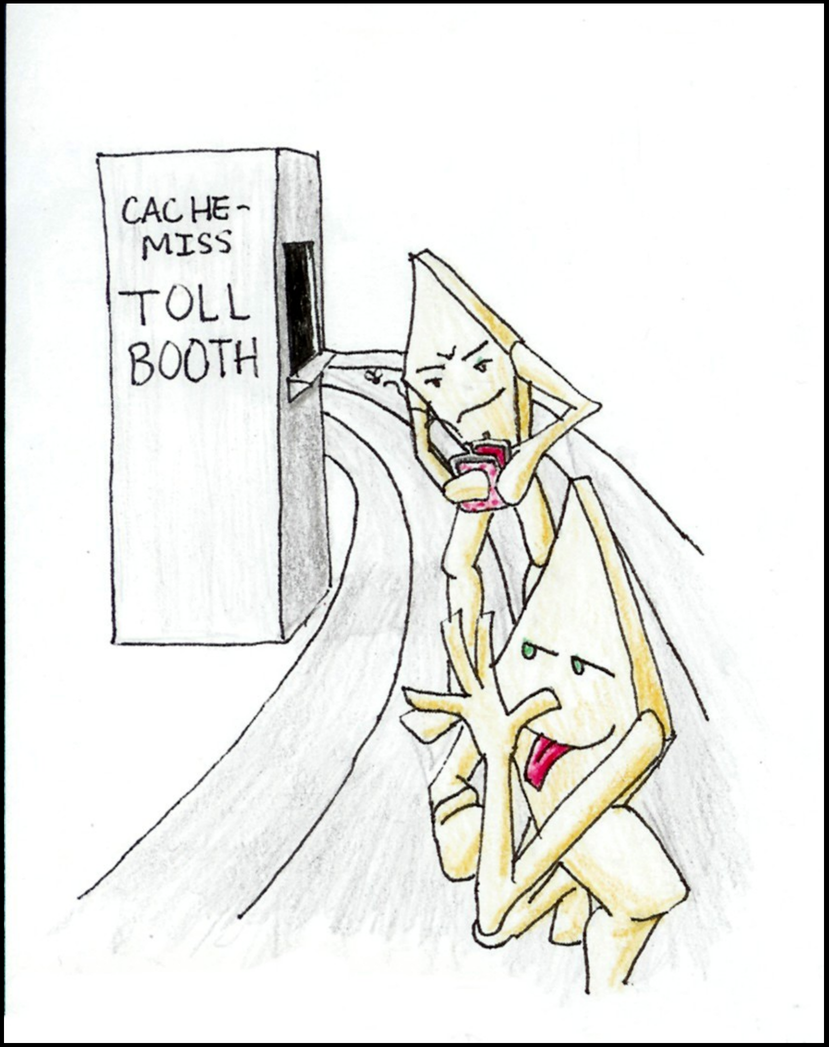


CPU Performance: Atomic Instructions



CPU Performance: Memory Barriers

CPU Performance: Cache Misses



And Don't Even Get Me Started on I/O...

CPU Performance: 4-CPU 1.8GHz Opteron 844

Need to be here!

Operation	Cost (ns)	Ratio
Clock period	0.6	1
Best-case CAS	37.9	63.2
Best-case lock	65.6	109.3
Single cache miss	139.5	232.5
CAS cache miss	306.0	510.0

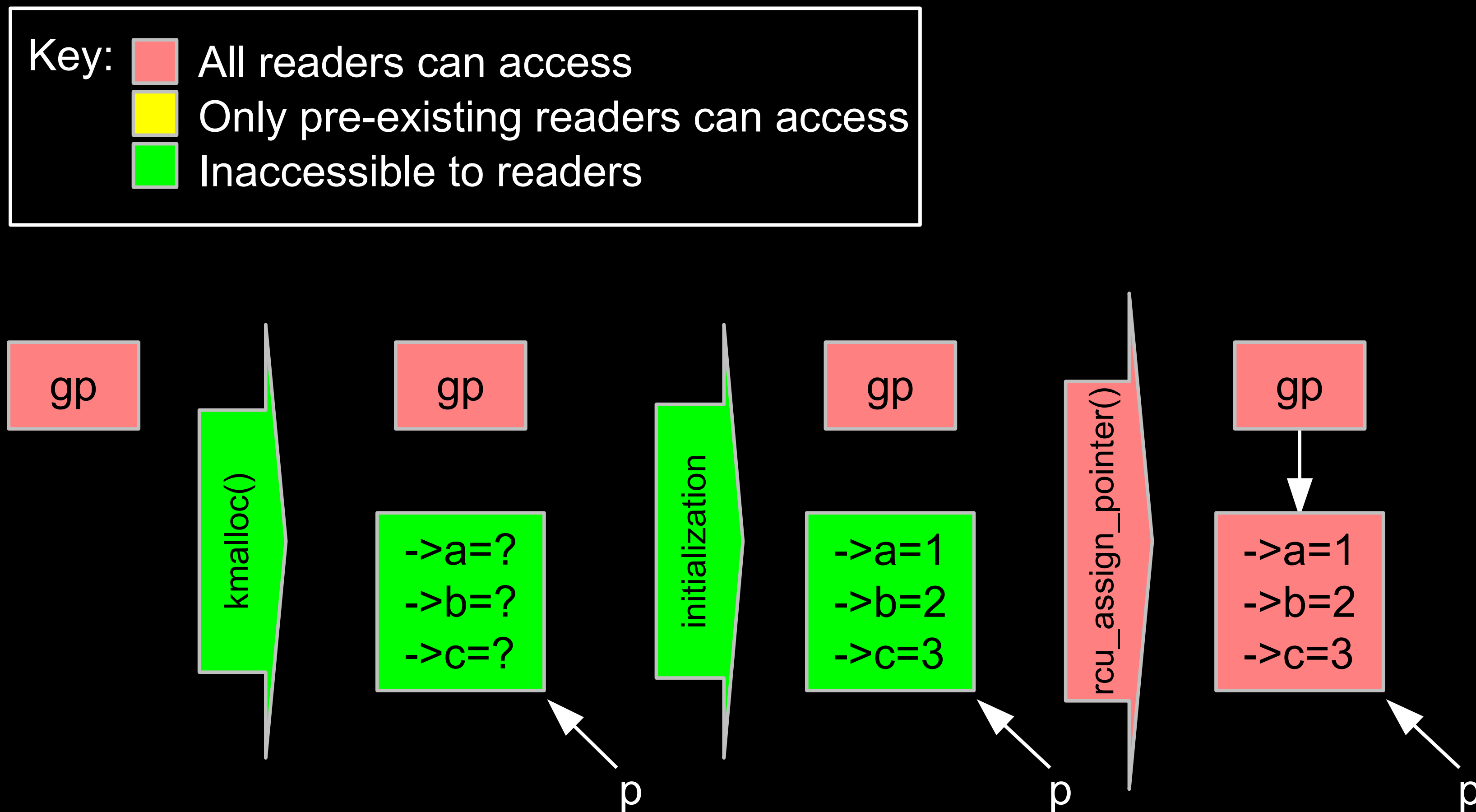
Typical synchronization mechanisms do this a lot

Larger machines usually incur larger penalties...
(1) Use coarse-grained parallelism: embarrassingly parallel is good!
(2) Make use of low-cost operations: For example, user-level RCU

What is RCU Fundamentally?

- **Synchronization mechanism in Linux kernel**
  - Favors readers: extremely fast and deterministic RCU read-side primitives (on the order of 1-10ns)
    - Use RCU primarily useful in read-mostly situations
  - Readers run concurrently with readers and updaters
  - Updaters must synchronize with each other somehow
    - Locks, atomic operations (but careful!!!), single update task...
- **Three components of RCU:**
  - Publish-subscribe mechanism (for insertion)
  - Wait for pre-existing RCU readers (for deletion)
    - This is slow – multiple milliseconds
  - Maintain multiple versions (for concurrent readers)

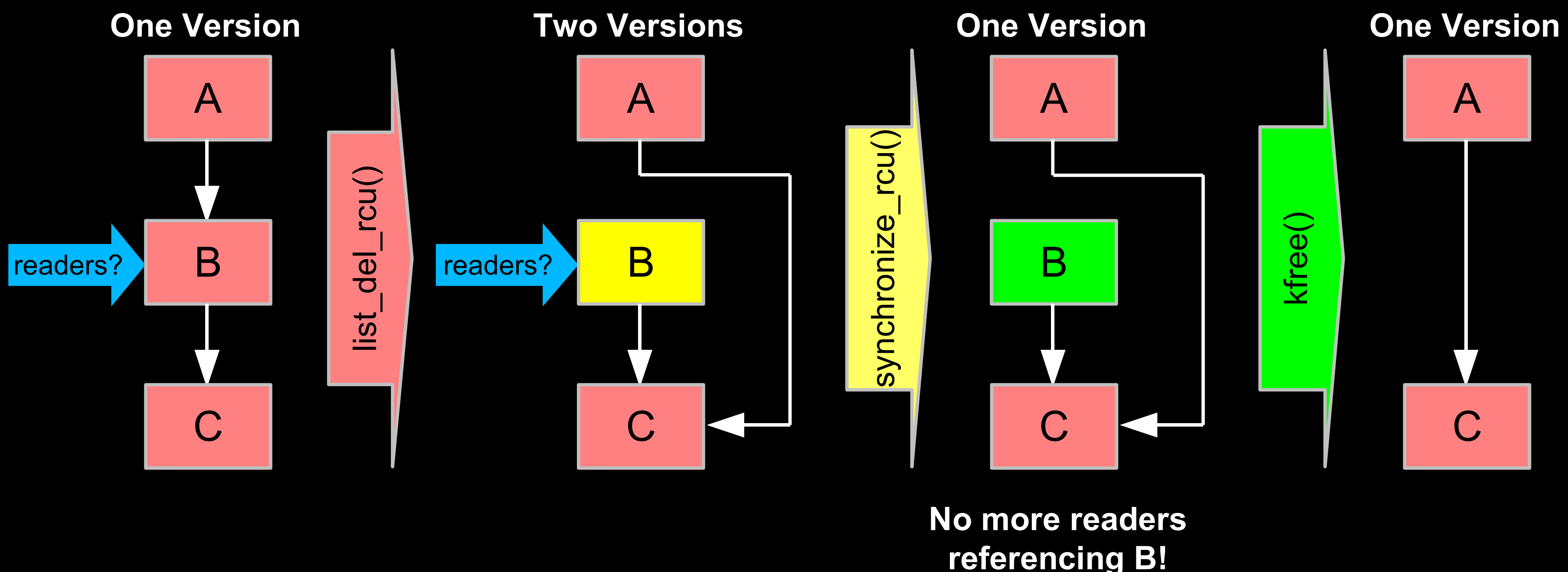
RCU List Insertion: Publication & Subscription



Readers subscribe using rcu_dereference() within an rcu_read_lock()/rcu_read_unlock() pair

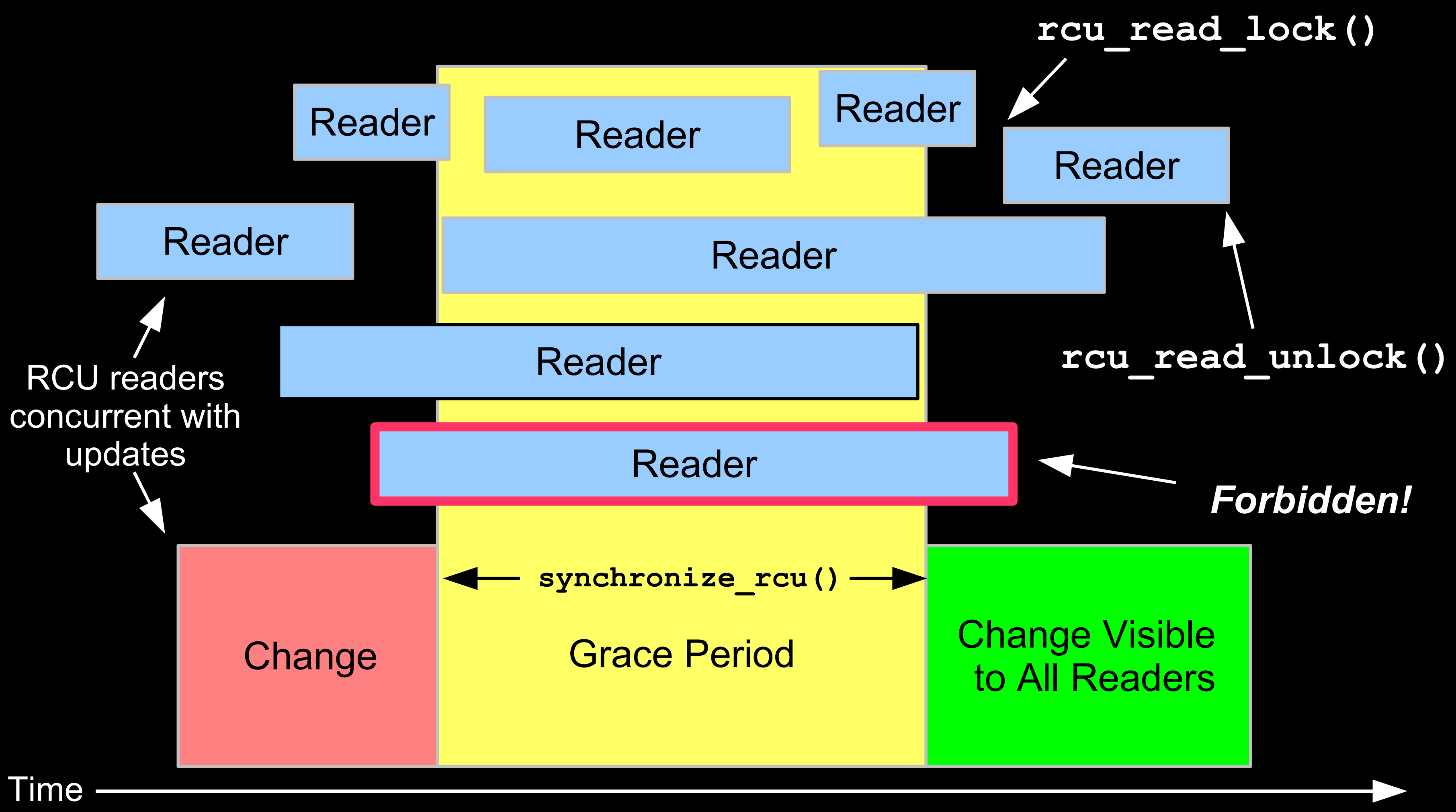
RCU List Deletion: Wait For Pre-Existing Readers

- **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())



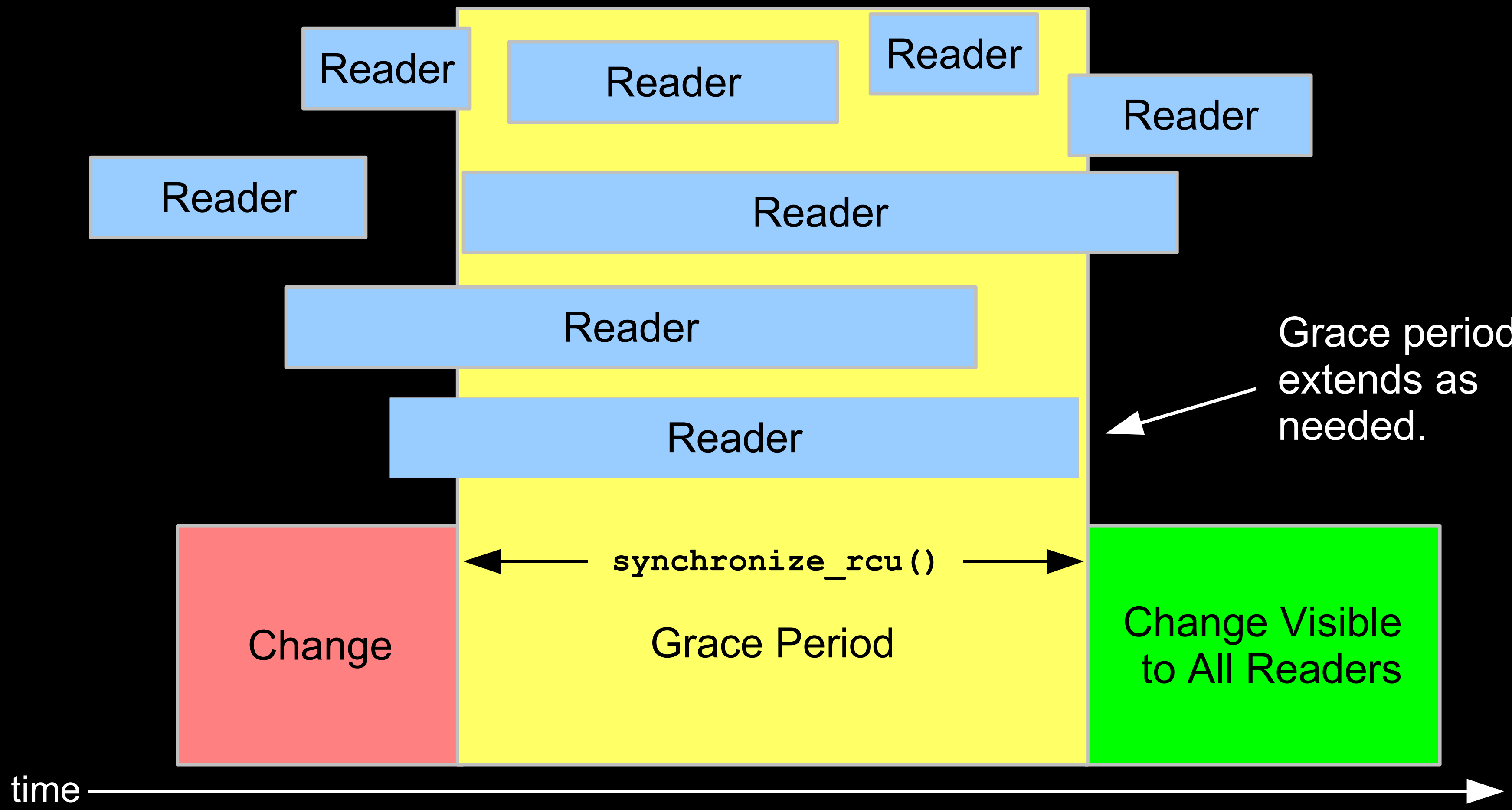
Readers subscribe using rcu_dereference() within an rcu_read_lock()/rcu_read_unlock() pair

RCU List Deletion: Wait For Pre-Existing Readers



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

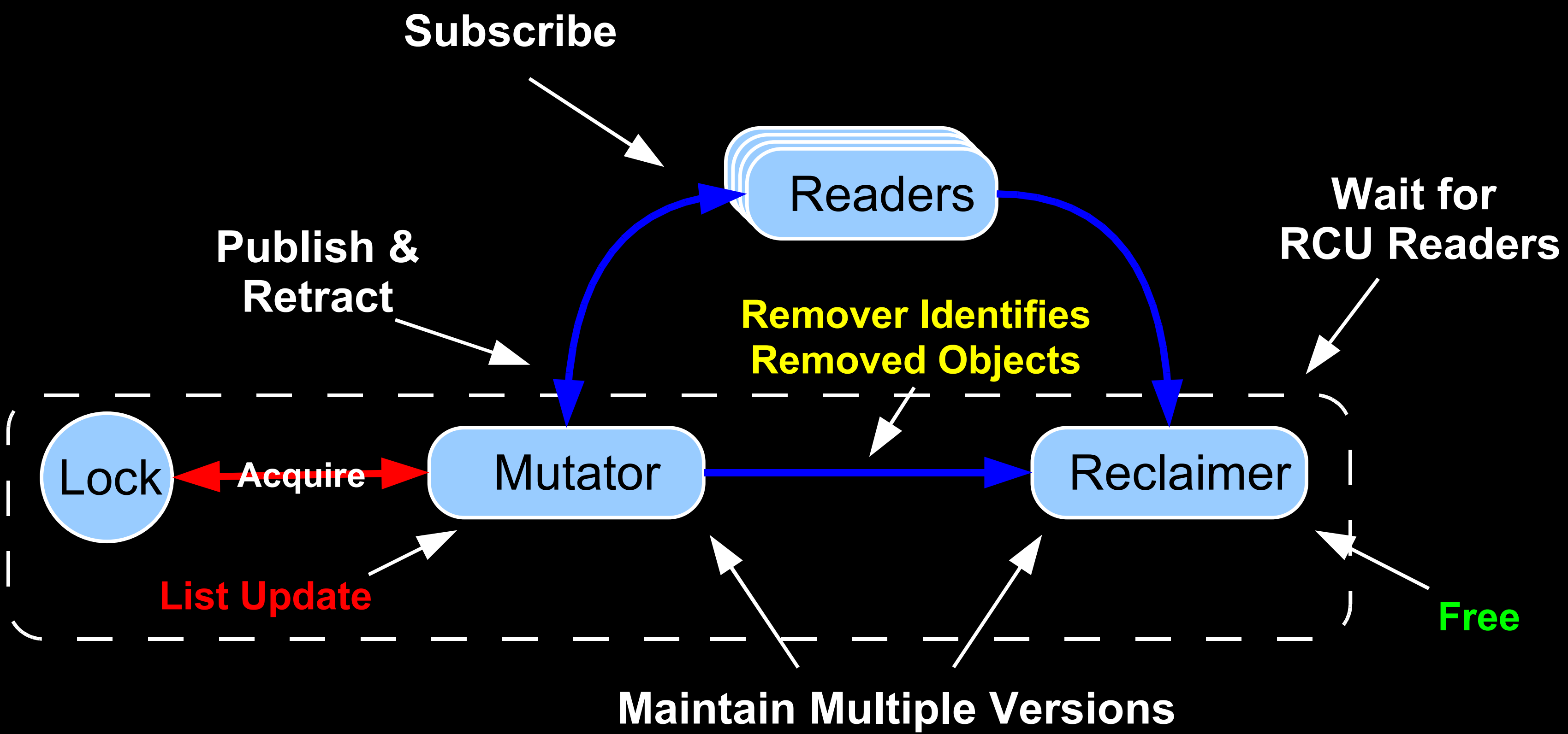
RCU List Deletion: Wait For Pre-Existing Readers



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

What Is RCU Fundamentally? (Summary)

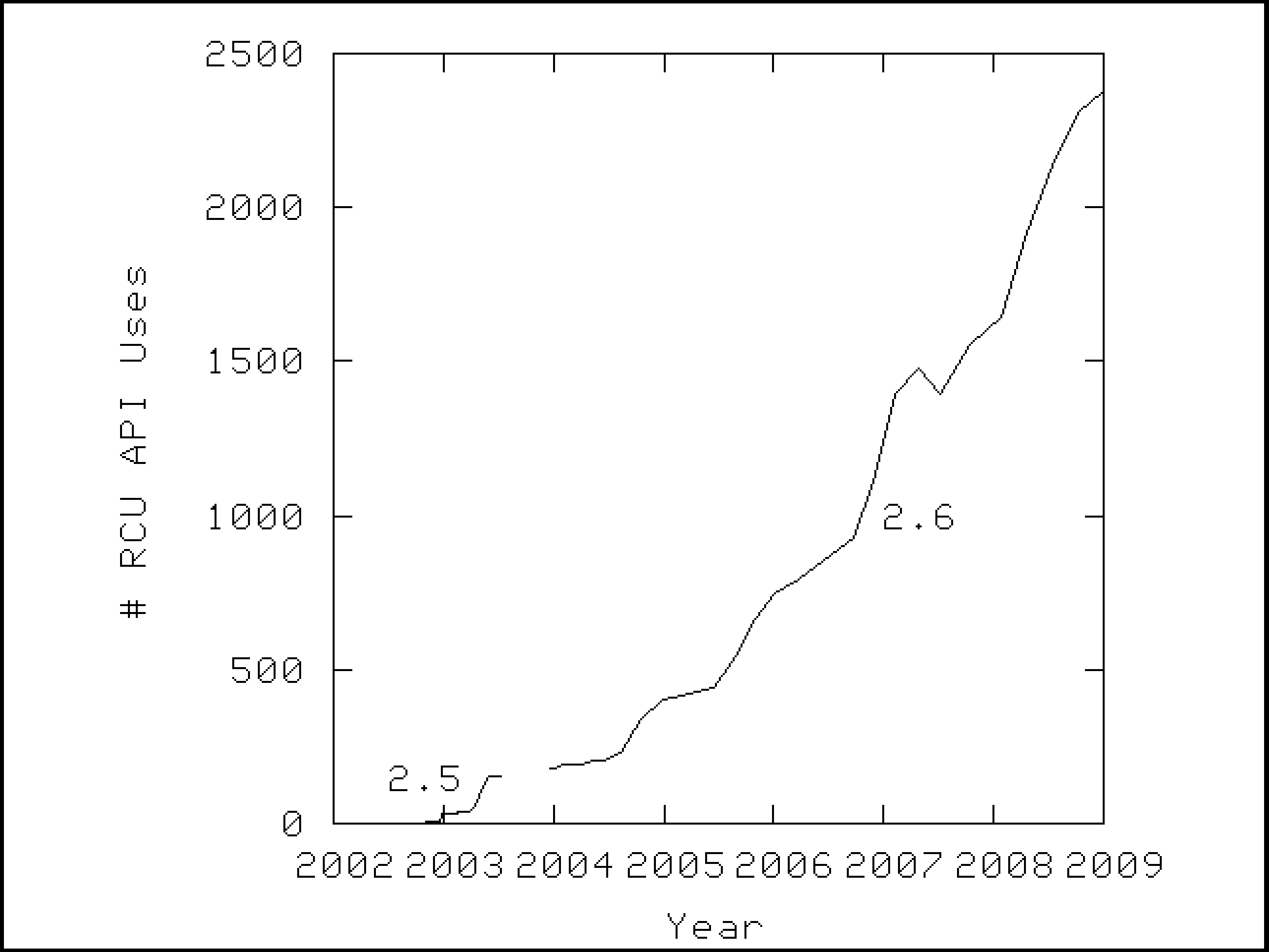
- **Relationship among RCU Components**



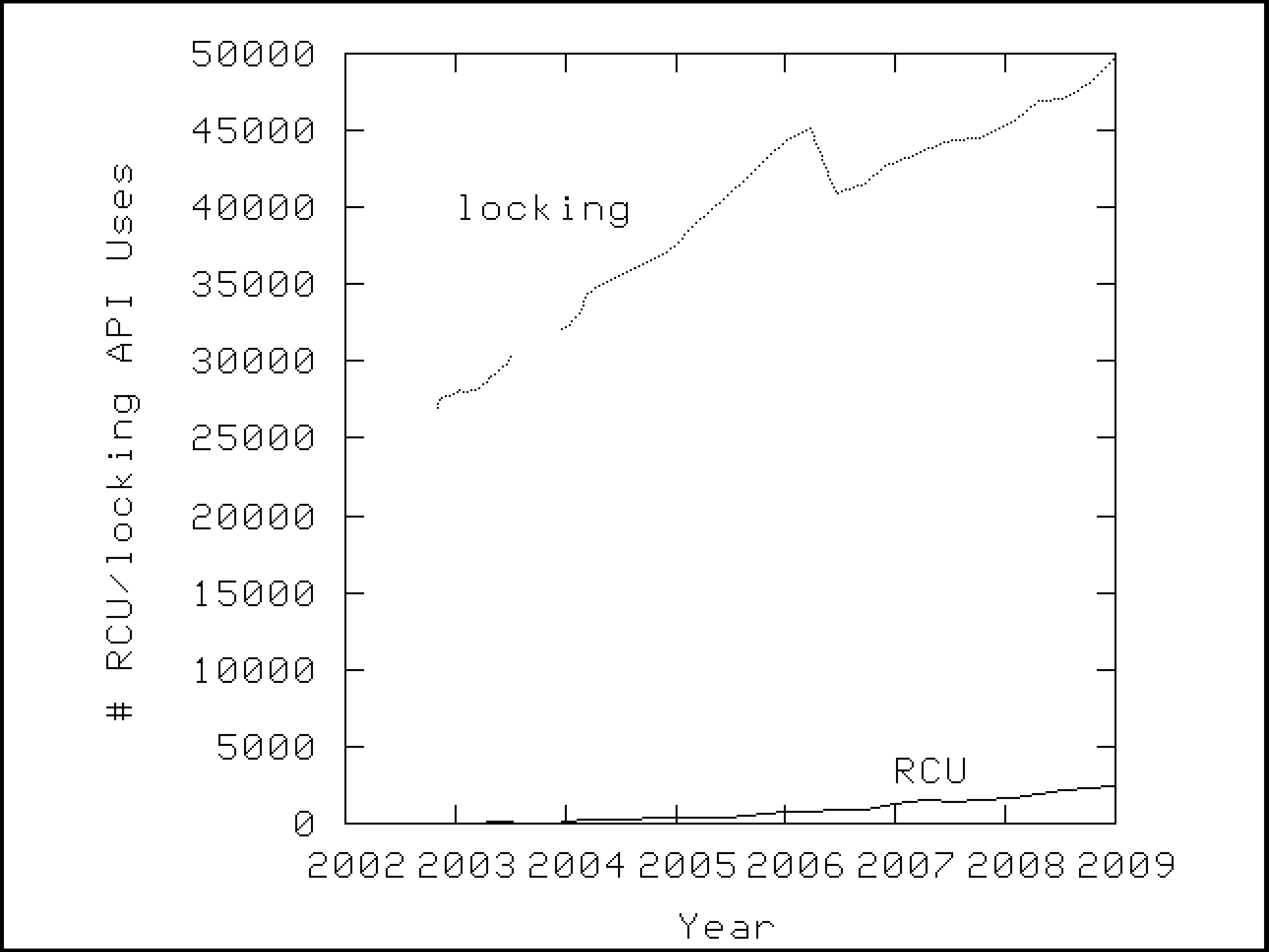
What is RCU's Usage?

- **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 providing type-safe memory
  - 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

What is RCU's Usage in the Linux Kernel?



What is RCU's Usage in the Linux Kernel?



Too Probe More Deeply into RCU...

- **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/**
  - Paul McKenney's RCU page.

RCU Advantages and Disadvantages

- **+ Low-overhead linearly scaling read-side primitives**
- **+ Deterministic read-side primitives (real time)**
- **+ Deadlock-immune read-side primitives**
  - (But don't do synchronize_rcu() in read-side critical section!!!)
- **+ Less need to partition read-mostly data structures**
- **+ Easier handling of new-reference/deletion races**
- **- High latency/overhead update-side primitives**
  - (But super-linear scaling due to batching implementations)
- **- Freed memory goes cache-cold**
  - (Hence application to read-mostly data structures)
- **- Updates run concurrently with readers**
  - (Common design patterns handle this issue)
- **- Only runs in kernels**
  - And the Linux-kernel implementation is ***very*** forgiving!!!

Linux-Kernel RCU Implementations Too Forgiving!!!

- **Preemptable-RCU experience is a case in point...**
- **5 May 2008: Alexey Dobriyan: oops from RCU code**
  - Running 170 parallel kernel builds on a 2-CPU x86 box
  - Takes about two full ***days*** to fail
  - I cannot reproduce, and cannot get .config from Alexey
- **7 June 2008: Alexey tries rcutorture, which fails**
  - I still cannot reproduce, and still cannot get .config from Alexey
- **24 June 2008: Nick Piggin: lockless-pagecache oops**
  - I cannot reproduce, and no .config from Nick, either

Linux-Kernel RCU Implementations Too Forgiving!!!

- **July 10 2008: Nick Piggin finds bug**
  - Preemptable RCU broken unless CPU_HOTPLUG enabled
    - My setup cheerfully and silently ignored disabling CPU_HOTPLUG!!!
    - Unless I also disabled several other config parameters
  - Result: synchronize_rcu() was completely ignoring rcu_read_lock()!!!
    - Thus becoming a pure delay of a few tens of milliseconds
  - It nevertheless ran 170 parallel kernel builds for about two ***days!!!***
  - **Suppose someone forgets rcu_read_lock()? How to test???**
- **From Nick's email:**
  - “Annoyed this wasn't a crazy obscure error in the algorithm I could fix :) I spent all day debugging it and had to make a special test case (rcutorture didn't seem to trigger it), and a big RCU state logging infrastructure to log millions of RCU state transitions and events. Oh well.”
- **Alexey's response did much to explain lack of .config**

RCU Requirements Summary

- **Update-side primitive waits for pre-existing readers**
  - Contained update latency
- **Low (deterministic) read-side overhead**
  - For debugging, need ability to force very short grace period
- **Freely nestable read side primitives**
  - (Some uses can do not need this)
- **Unconditional read-to-update upgrade**
- **Linear read-side scalability**
- **Independent of memory allocation**
- **Update-side scalability**
- **Some way of stress-testing algorithms using RCU!!!**

- **Note that an automatic garbage collector qualifies as an RCU implementation**

User-Level RCU Challenges

- **Cannot portably identify CPU**
- **Cannot portably disable preemption**
- **No equivalent of in-kernel scheduling-clock interrupt**
- **Less control of application**
  - If you are writing a user-level library, the application you will link with might not even been thought of yet!
  - So cannot necessarily rely on timely interaction with all threads
  - Which every current RCU implementation requires...

Addressing User-Level RCU Challenges

- **Cannot portably identify CPU**
  - Focus instead on processes and/or threads
- **Cannot portably disable preemption**
  - Avoid need for this by process/thread focus
- **No equivalent of in-kernel scheduling-clock interrupt**
  - Drive grace periods from update-side primitives
  - Or provide separate thread(s) for this purpose
- **Less control of application**
  - "Learn to let go..."
  - And provide optimized RCU implementations for applications that ***can*** periodically execute RCU code

User-Level RCU: Trivial Approach

```
static void rcu_read_lock(void)
{
  atomic_inc(&rcu_ref_cnt);
  smp_mb();
}

static void rcu_read_unlock(void)
{
  smp_mb();
  atomic_dec(&rcu_ref_cnt);
}
```

Read-side cost?

User-Level RCU: Trivial Approach

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

  smp_mb();
  while (atomic_read(&rcu_ref_cnt) != 0) {
      /*@@@ poll(NULL, 0, 10); */
    }
  }
  smp_mb();
}
```

Extremely fast grace-period latency in absence of readers, but...

User-Level RCU: Super-Trivial Approach

```
static void rcu_read_lock(void)
{
  spin_lock(&rcu_gp_lock);
}

static void rcu_read_unlock(void)
{
  spin_unlock(&rcu_gp_lock);
}

void synchronize_rcu(void)
{
  spin_lock(&rcu_gp_lock);
  spin_unlock(&rcu_gp_lock);
}
```

Hey! Who really needs read-side parallelism, anyway?
And deadlock immunity is overrated!!!

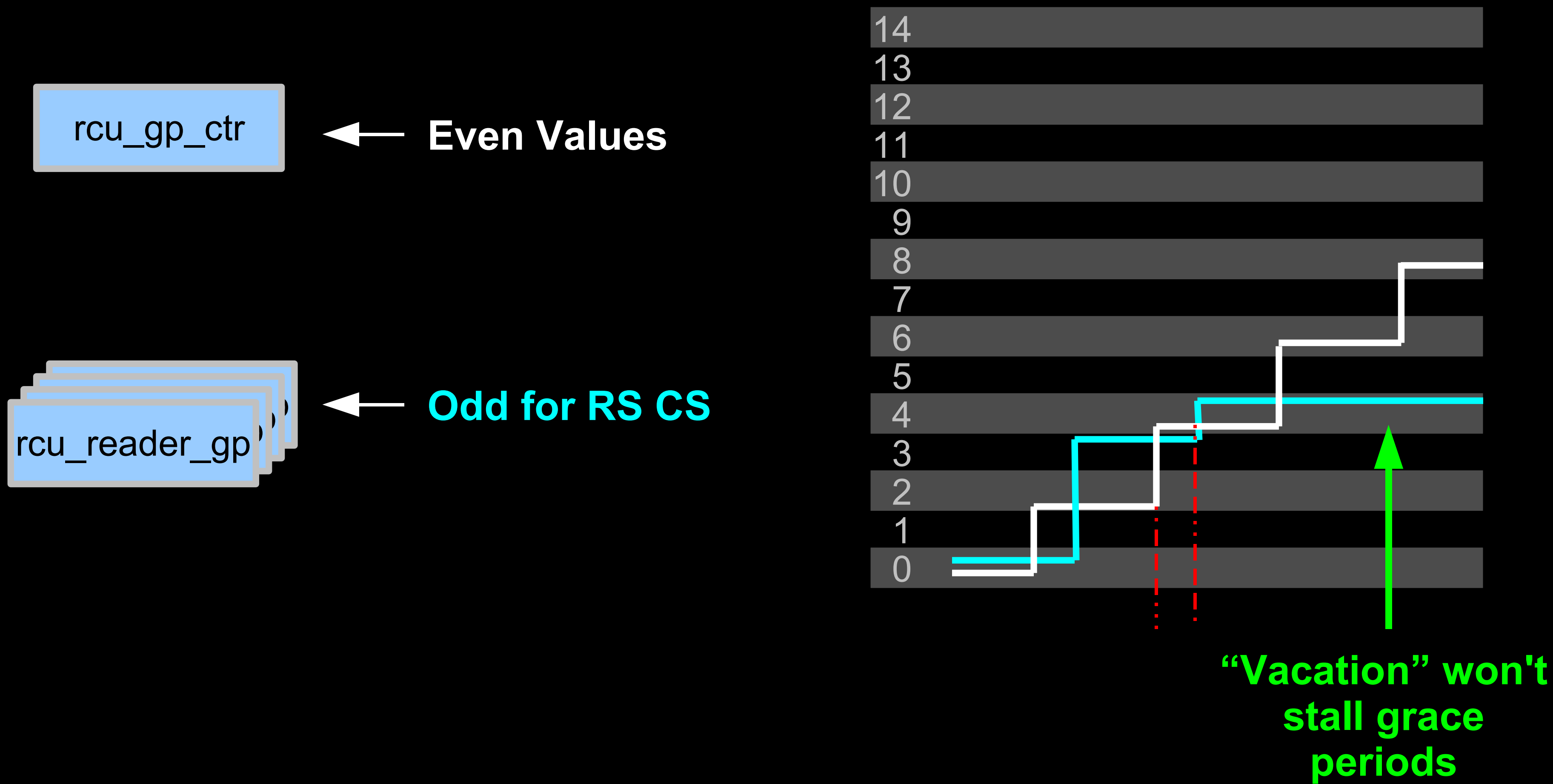
Other Approaches

- **Split counter (http://lwn.net/Articles/253651/)**
  - A pair of reference counters plus an index selecting "current"
  - rcu_read_lock() increments rcu_ref_cnt[current]
  - rcu_read_unlock() decrements whatever the corresponding rcu_read_lock() incremented
  - synchronize_rcu() complements current, then waits until rcu_ref_cnt[!current] decrements down to zero
  - But requires coordinated access to current and rcu_ref_cnt[] element
    - Provided in Linux kernel by interrupt disabling and scheduling-clock rrupt
    - Neither of which are available to user-level code
    - Would require expensive explicit locks at user level!!!
  - Memory contention on rcu_ref_cnt[current]
- **Use per-thread lock**
  - rcu_read_lock() acquires its thread's lock
  - rcu_read_unlock() releases it
  - synchronize_rcu() acquires & immediately releases each lock
  - Reduces the deadlock vulnerabilities, also read-side overhead
    - Too bad about signal handlers using RCU, though...

Other Approaches

- **Per-thread split counter (http://lwn.net/Articles/253651/)**
  - A pair of reference counters plus an index selecting "current"
  - rcu_read_lock() increments rcu_ref_cnt[threadidx][current]
  - rcu_read_unlock() decrements whatever the corresponding rcu_read_lock() incremented
  - synchronize_rcu() complements current, then waits until all of the rcu_ref_cnt[][!current] counters decrement down to zero
- **What is wrong with this approach?**

User-Level RCU: Simple “Hands-Free” Approach



User-Level RCU: Simple “Hands-Free” Code

```
static void rcu_read_lock(void)
{
  __get_thread_var(rcu_reader_gp) = rcu_gp_ctr + 1;
  smp_mb();
}

static void rcu_read_unlock(void)
{
  smp_mb();
  __get_thread_var(rcu_reader_gp) = rcu_gp_ctr;
}
```

User-Level RCU: Simple “Hands-Free” Code

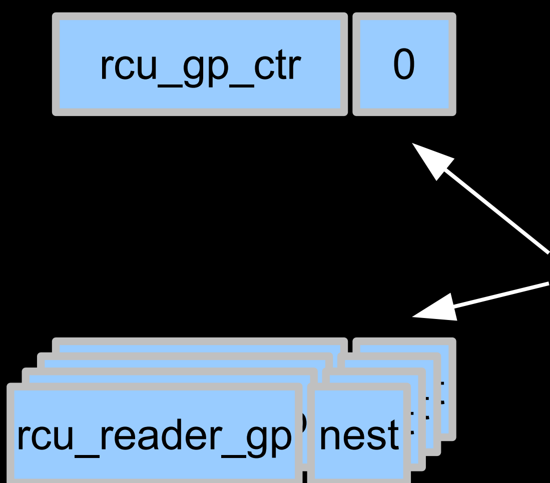
```
void synchronize_rcu(void)
{
  int t;

  smp_mb();
  spin_lock(&rcu_gp_lock);
  rcu_gp_ctr += 2;
  smp_mb();
  for_each_thread(t) {
    while ((per_thread(rcu_reader_gp, t) & 0x1) &&
           ((per_thread(rcu_reader_gp, t) - rcu_gp_ctr) < 0)) {
      /*@@@ poll(NULL, 0, 10); */
    }
  }
  spin_unlock(&rcu_gp_lock);
  smp_mb();
}
```

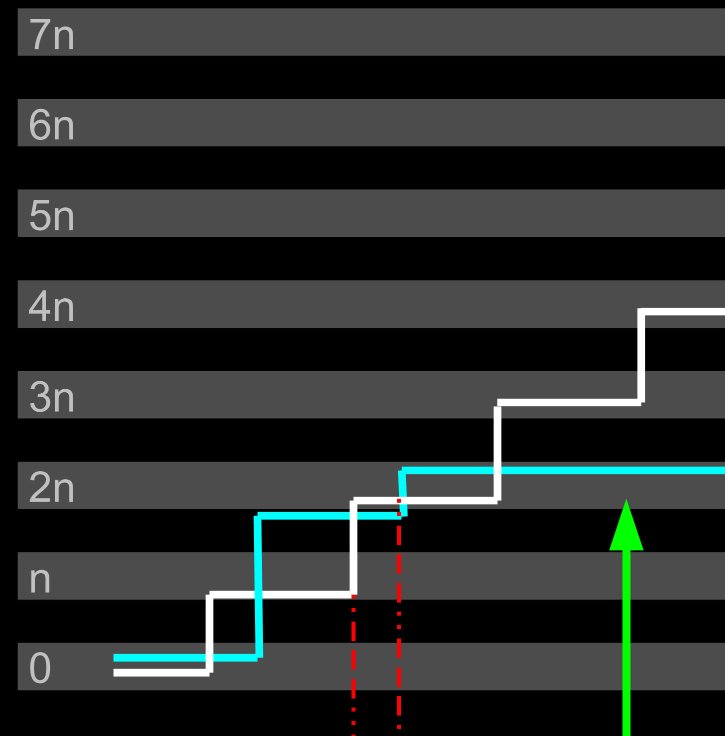
How Does This Solution Measure Up?

- **Update-side primitive wait for pre-existing RCU readers**
- **Low (deterministic) read-side overhead**
- **Freely nestable read side primitives**
- **Unconditional read-to-update upgrade**
- **Linear read-side scalability**
- **Independent of memory allocation**
- **Update-side scalability**

User-Level RCU: Nestable Approach



“Fat bottom bit”



“Vacation” won't stall grace periods

Must be in one quantity for atomicity.

User-Level RCU: Nestable Code

```
static void rcu_read_lock(void)
{
  long tmp;

  tmp = __get_thread_var(rcu_reader_gp);
  if ((tmp & RCU_GP_CTR_NEST_MASK) == 0)
    tmp = rcu_gp_ctr;
  tmp++;
  __get_thread_var(rcu_reader_gp) = tmp;
  smp_mb();
}


static void rcu_read_unlock(void)
{
  long tmp;

  smp_mb();
  __get_thread_var(rcu_reader_gp)--;
}
```

User-Level RCU: Nestable Code

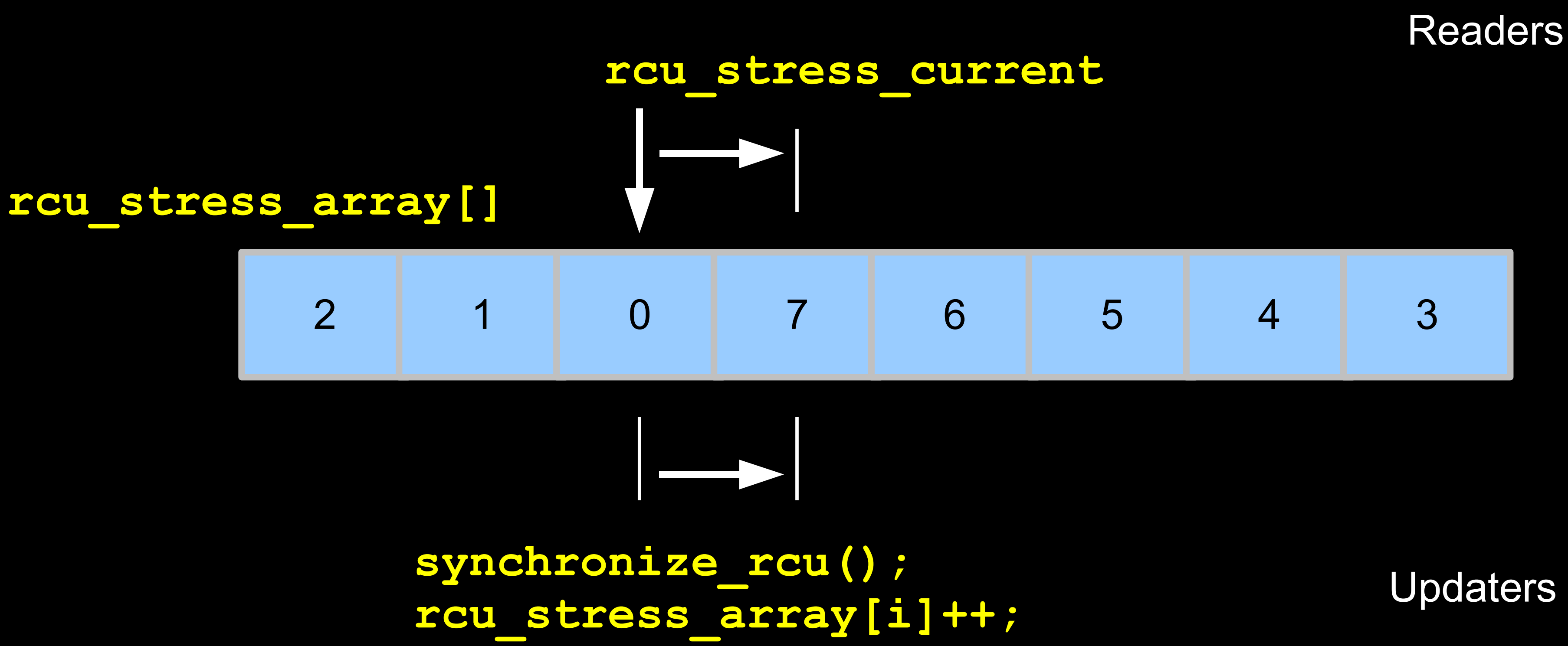
```
void synchronize_rcu(void)
{
  int t;

  smp_mb();
  spin_lock(&rcu_gp_lock);
  rcu_gp_ctr += RCU_GP_CTR_BOTTOM_BIT;
  smp_mb();
  for_each_thread(t) {
    while (rcu_gp_ongoing(t) &&
           ((per_thread(rcu_reader_gp, t) - rcu_gp_ctr) < 0)) {
      /*@@@ poll(NULL, 0, 10); */
    }
  }
  spin_unlock(&rcu_gp_lock);
  smp_mb();
}
```

How Does Nestable Solution Measure Up?

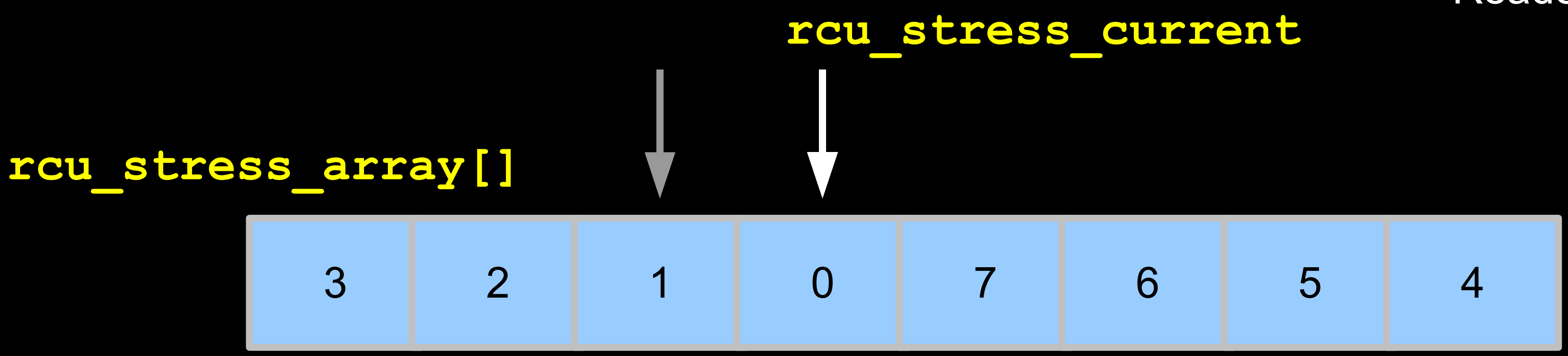
- **Update-side primitive wait for pre-existing RCU readers**
- **Low (deterministic) read-side overhead**
- **Freely nestable read side primitives**
- **Unconditional read-to-update upgrade**
- **Linear read-side scalability**
- **Independent of memory allocation**
- **Update-side scalability**

RCU Torture Testing Data Structures



RCU Torture Testing Data Structures

Readers



Updaters

Readers should see value of 0 and 1 only: otherwise, RCU is broken

RCU Torture Testing: Updater Thread

```
while (goflag == GOFLAG_RUN) {
  i = rcu_stress_idx + 1;
  if (i >= RCU_STRESS_PIPE_LEN)
    i = 0;
  p = &rcu_stress_array[i];
  p->pipe_count = 0;
  rcu_assign_pointer(rcu_stress_current, p);
  rcu_stress_idx = i;
  for (i = 0; i < RCU_STRESS_PIPE_LEN; i++)
    if (i != rcu_stress_idx)
      rcu_stress_array[i].pipe_count++;
  synchronize_rcu();
  n_updates++;
}
```

Malice Testing: Reader Threads

```
rcu_read_lock();
p = rcu_dereference(rcu_stress_current);
for (i = 0; i < 100; i++)
  garbage++;
pc = p->pipe_count;
rcu_read_unlock();
```

```
rcu_read_lock();
p = rcu_dereference(rcu_stress_current);
for (i = 0; i < 100; i++)
  garbage++;
rcu_read_unlock();   /* Malice. */
pc = p->pipe_count;  /* BUG!!!  */
```

Performance and Level of Malice

RCU Variant	Performance (ns, 64 CPUs)	Degree of Malice (Probability of Detection) 0	100	1,000	10,000
rcu	63		0.20458%	0.28930%	16.62725%
rcu_lock	17,123	22.63980%	21.87630%	27.23635%	77.45645%
rcu_lock_percpu	141	0.41581%	0.95454%	0.44058%	98.25215%
rcu_nest	64		0.10677%	0.33591%	21.91355%
rcu_nest_qs	26				
rcu_qs	0				
rcu_rcg	39,177	0.01351%	0.26418%	68.92650%	92.53230%
rcu_rcpg	37,056	0.00023%	0.20246%	23.64110%	91.99550%
rcu_rcpl	114	0.00020%	0.26680%	0.38274%	96.22135%
rcu_rcpls	114	0.00005%	0.25493%	0.38453%	97.22010%
rcu_ts	101		0.17684%	0.31986%	43.60365%

Mean of three trials of 10-second duration.
1-2 significant decimal digits in results.

Future Work

- **Implement full Linux-kernel RCU API**
  - Currently, just have the bare bones
    - rcu_read_lock()
    - rcu_read_unlock()
    - synchronize_rcu()
    - Prototype containing call_rcu()
- **Choose a particular implementation for user-level debugging of RCU algorithms**
  - But more experience will be needed
- **Try it out on a real user-land application**

Conclusions

- **User-level RCU implementation possible, even for library functions**
- **Extremely low grace-period latency**
  - Suggests use as a torture-test environment for RCU algorithms
  - Subject of an upcoming presentation at linux.conf.au
  - Though latency will increase with number of CPUs
- **OK read-side overhead**
  - Less than 30% of the overhead of a single cache miss!
- **Full RCU semantics**

To Probe Deeper

- **Other Parallel Algorithms and Tools**
  - http://www.rdrop.com/users/paulmck/scalability/
- **What is RCU?**
  - Fundamentally: http://lwn.net/Articles/262464/
  - Usage: http://lwn.net/Articles/263130/
  - API: http://lwn.net/Articles/264090/
  - Linux-kernel usage: http://www.rdrop.com/users/paulmck/RCU/linuxusage.html
  - Other RCU stuff: http://www.rdrop.com/users/paulmck/RCU/
- **Parallel Performance Programming (very raw draft)**
  - Contains source code for user-level RCU implementations
  - git://git.kernel.org/pub/scm/linux/kernel/git/paulmck/perfbook.git

Legal Statement

- **This work represents the views of the authors and does not necessarily represent the view of IBM.**
- **Linux is a copyright of Linus Torvalds.**
- **Other company, product, and service names may be trademarks or service marks of others.**

Backup