2010 linux.conf.au Wellington, NZ

Simplicity Through Optimization

It Doesn't Always Work This Way, But It Is Sure Nice When It Does!!!

Paul E. McKenney, Distinguished Engineer
IBM Linux Technology Center

January 21, 2010

Overview

- **A Puzzle From 1984**
- **How Optimization Goes Bad And Why**
- **RCU 1993-2008: Example Optimization Gone Bad**
- **RCU 2009-2010: Simplicity Through Optimization**
- **Next Steps**
- **Lessons Learned**

A Puzzle From 1984

That would be the year, not the book!!!

A Puzzle From 1984: BSD 2.8 on PDP-11/23

- **64Kbyte address space, 256K-1M physical memory**
- **Three seconds to fork()/exec() minimal program**
  - Which helps explain all the “case ... esac” usage in “sh”
- **We got a larger disk, migrated FSes from old disk**
- **Worked great for awhile, then started getting corrupted source files**
- **Application deadlines loomed, so just created a .BAD directory, and moved all corrupted files there**
- **After awhile, the problem went away**
- ***What was happening???***

How Optimization Goes Bad And Why

Optimization Going Bad

- **Powerful optimization strategy**
  - *Take advantage of special cases!!!*
- **Single special case simple, but specialized**

Optimization Going Bad

- **Powerful optimization strategy**
  - *Take advantage of special cases!!!*
- **Single special case simple, but specialized**
  - OK sometimes, but when you must handle general problem:

Special-Case Selection and Dispatch

Special Case #1

Special Case #2

Special Case #3

Special Case #4

Special Case #5

A Puzzle From 1984: BSD 2.8 on PDP-11/23

- **64Kbyte address space, 256K-1M physical memory**
- **Three seconds to fork()/exec() minimal program**
  - Which helps explain all the “case ... esac” usage
- **We got a larger disk, migrated FSes from old disk**
- **Worked great then started getting corrupted file data**
- **Application deadlines loomed, so just created a .BAD directory, and moved all corrupted files there**
- **Problem went away**
- ***What was happening???***

A Puzzle From 1984: BSD 2.8 on PDP-11/23

- **64Kbyte address space, 256K-1M physical memory**
- **Three seconds to fork()/exec() minimal program**
  - Which helps explain all the “case ... esac” usage
- **We got a larger disk, migrated FSes from old disk**
- **Worked great, then started getting corrupted file data**
- **Application deadlines loomed, so just created a .BAD directory, and moved all corrupted files there**
- **Problem went away**
- ***What was happening???***
  - Relocating the swap partition required kernel source change

A Puzzle From 1984: BSD 2.8 on PDP-11/23

- **64Kbyte address space, 256K-1M physical memory**
- **Three seconds to fork()/exec() minimal program**
  - Which helps explain all the “case ... esac” usage
- **We got a larger disk, migrated FSes from old disk**
- **Worked great, then started getting corrupted file data**
- **Application deadlines loomed, so just created a .BAD directory, and moved all corrupted files there**
- **Problem went away**
- ***What was happening???***
  - Relocating the swap partition required kernel source change
  - In three different places in the kernel source

A Puzzle From 1984: BSD 2.8 on PDP-11/23

- **64Kbyte address space, 256K-1M physical memory**
- **Three seconds to fork()/exec() minimal program**
  - Which helps explain all the “case ... esac” usage
- **We got a larger disk, migrated FSes from old disk**
- **Worked great, then started getting corrupted file data**
- **Application deadlines loomed, so just created a .BAD directory, and moved all corrupted files there**
- **Problem went away**
- ***What was happening???***
  - Relocating the swap partition required kernel source change
  - In three different places in the kernel source
  - We had found only one of them

A Puzzle From 1984: BSD 2.8 on PDP-11/23

- **64Kbyte address space, 256K-1M physical memory**
- **Three seconds to fork()/exec() minimal program**
  - Which helps explain all the “case ... esac” usage
- **We got a larger disk, migrated FSes from old disk**
- **Worked great, then started getting corrupted file data**
- **Application deadlines loomed, so just created a .BAD directory, and moved all corrupted files there**
- **Problem went away**
- ***What was happening???***
  - Relocating the swap partition required kernel source change
  - In three different places in the kernel source
  - We had found only one of them
  - ***Why does this matter???***

A Puzzle From 1984: Lessons For 2010

- Coordinated kernel-source dependency unacceptable
- Kernels are now expected to adapt to their surroundings as they change
- Kernels are now expected to adapt to configuration and administration changes dynamically, without rebuild
  - And without reboot
- In contrast with 1984, there is now a huge amount of read-mostly data in the kernel tracking such info
  - Almost never changes, but might change at any time
- In 2010, optimizations for read-mostly data are much more important than in 1984

A Puzzle From 1984: Lessons For 2010

- **Coordinated kernel-source dependency unacceptable**
- **Kernels are now expected to adapt to their surroundings as they change**
- **Kernels are now expected to adapt to configuration and administration changes dynamically, without rebuild**
  - **And without reboot**
- **In contrast with 1984, there is now a huge amount of read-mostly data in the kernel tracking such info**
  - **Almost never changes, but might change at any time**
- **In 2010, optimizations for read-mostly data are much more important than in 1984**
  - **And what is my favorite read-mostly optimization???**

RCU 1993-2008: Example Optimization Gone Bad

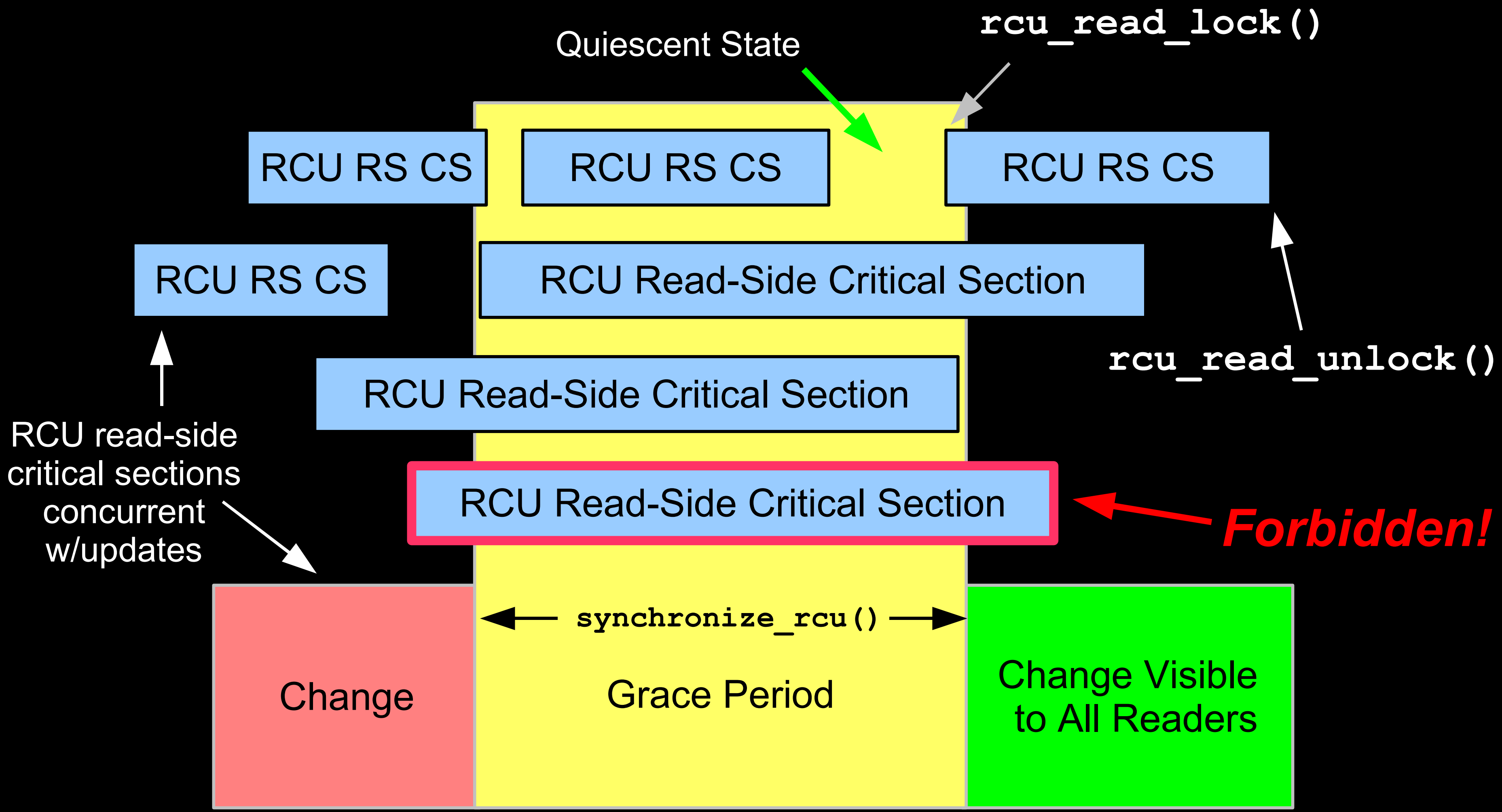
But First, What Is RCU?

- **"RCU read-side critical section" & "quiescent state":**

```
/* Quiescent state */
rcu_read_lock();
    /* RCU read-side critical section */
    rcu_read_lock();
      /* RCU read-side critical section */
    rcu_read_unlock();
    /* RCU read-side critical section */
rcu_read_unlock();
/* Quiescent state */
```

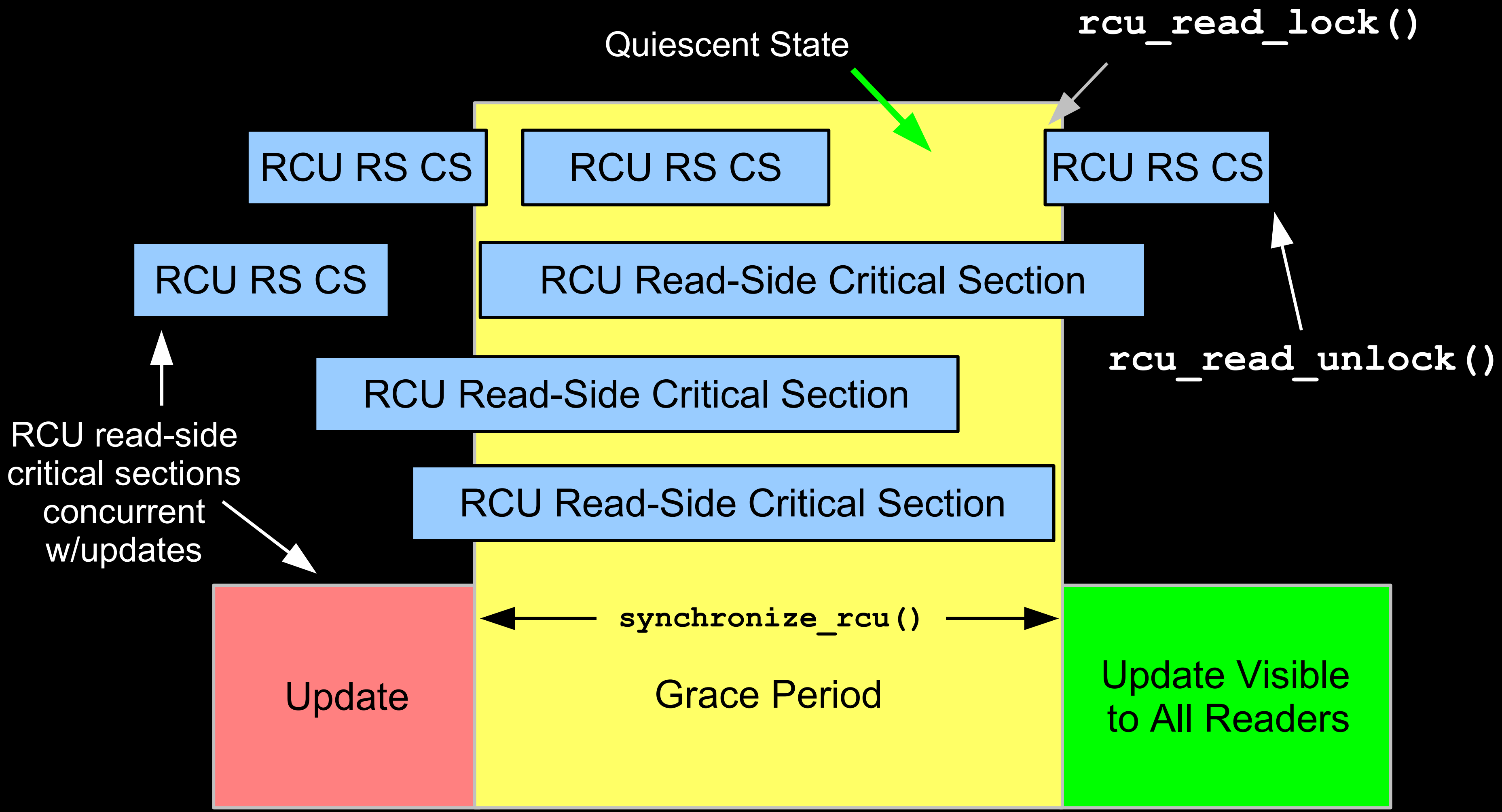
- **"Grace period": period of time during which all CPUs (or threads) pass through at least one quiescent state**
  - Any time period including a grace period is a grace period
  - Distinct grace periods may overlap partially or completely
  - All RCU read-side critical sections present at the start of a grace period must complete before the end of that grace period

Pictorial Version



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

Pictorial Version



Quiescent State

`rcu_read_lock()`

RCU RS CS

RCU RS CS

RCU RS CS

RCU RS CS

RCU Read-Side Critical Section

`rcu_read_unlock()`

RCU Read-Side Critical Section

RCU read-side critical sections concurrent w/updates

RCU Read-Side Critical Section

`synchronize_rcu()`

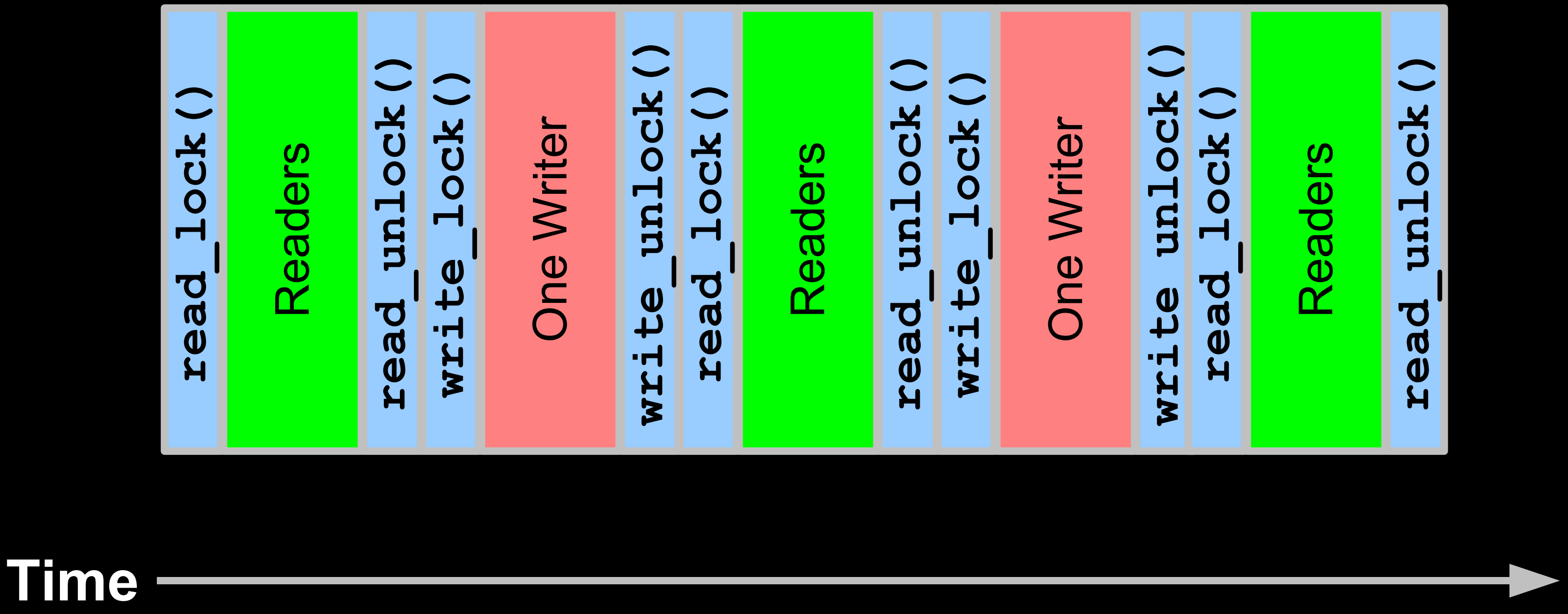
Update

Grace Period

Update Visible to All Readers

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

Conventional RW Locking: Exclusion in Time



`read_lock()`

Readers

`read_unlock()`

`write_lock()`

One Writer

`write_unlock()`

`read_lock()`

Readers

`read_unlock()`

`write_lock()`

One Writer

`write_unlock()`

`read_lock()`

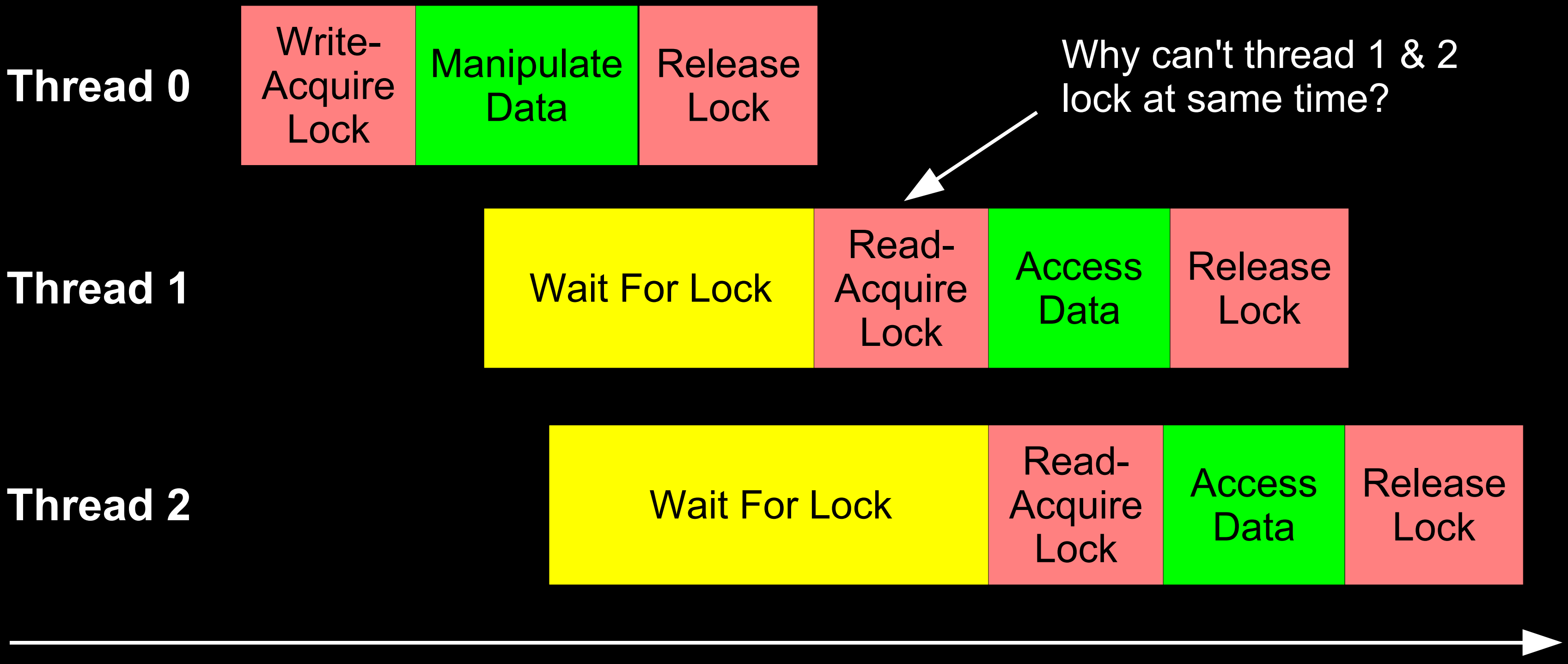
Readers

`read_unlock()`

Time

Problems: poor performance and bad scalability, non-real-time latencies

Illustration of Poor RW Locking Performance



Thread 0

Write-Acquire Lock

Manipulate Data

Release Lock

Why can't thread 1 & 2 lock at same time?

Thread 1

Wait For Lock

Read-Acquire Lock

Access Data

Release Lock

Thread 2

Wait For Lock

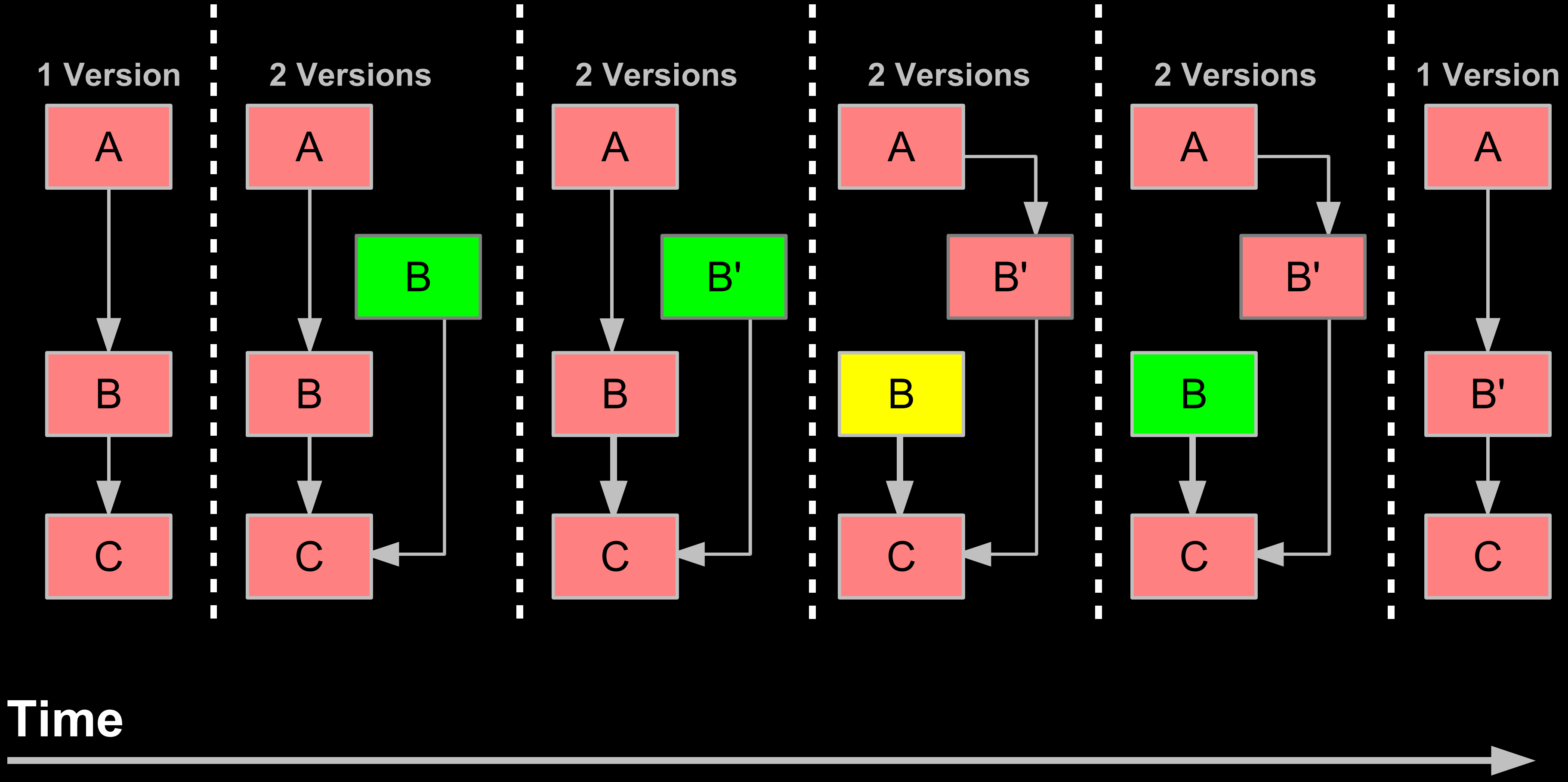
Read-Acquire Lock

Access Data

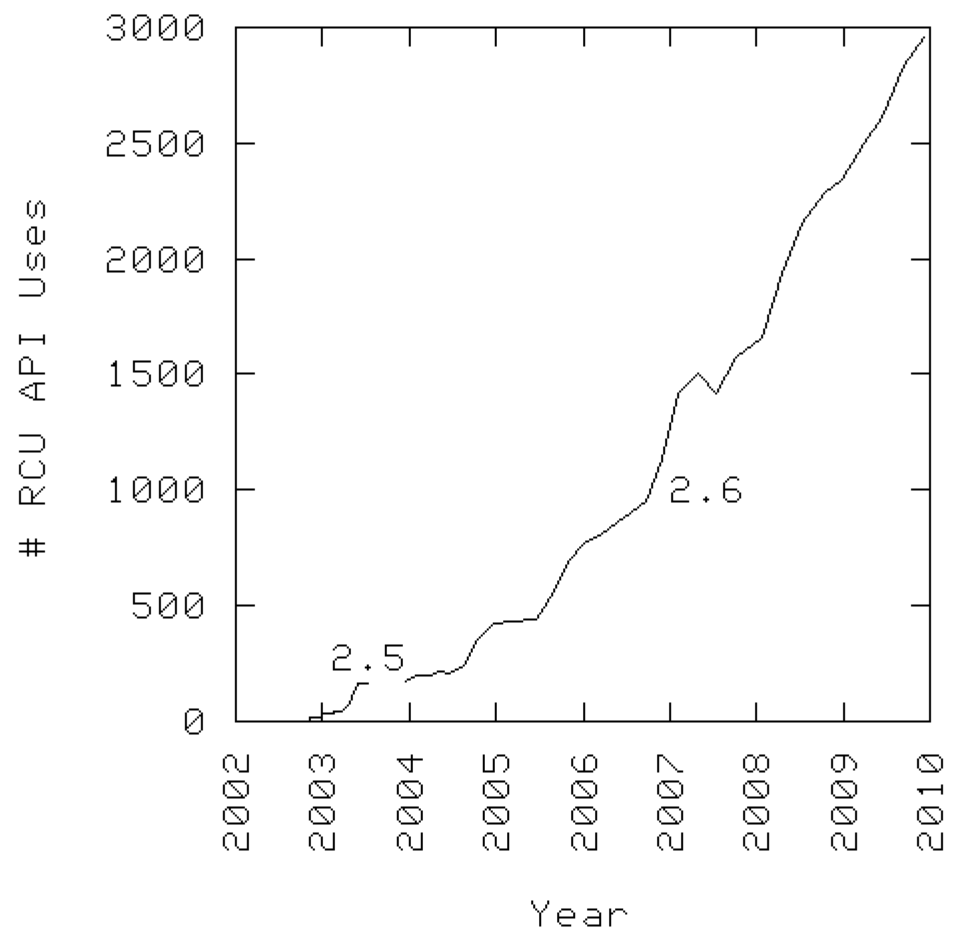
Release Lock

Courtesy of the atomic nature of matter and the finite speed of light.

RCU: Exclusion in Time and Space

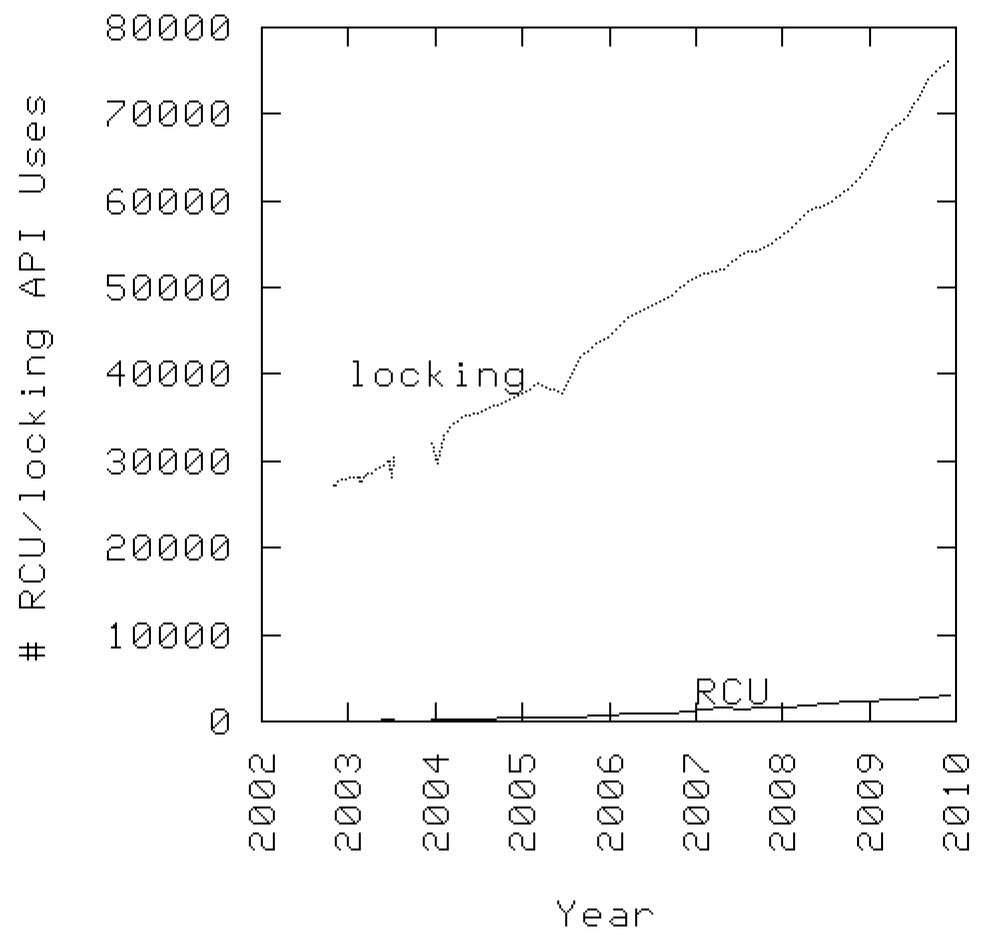


RCU Usage Within the Linux Kernel (2.6.32)



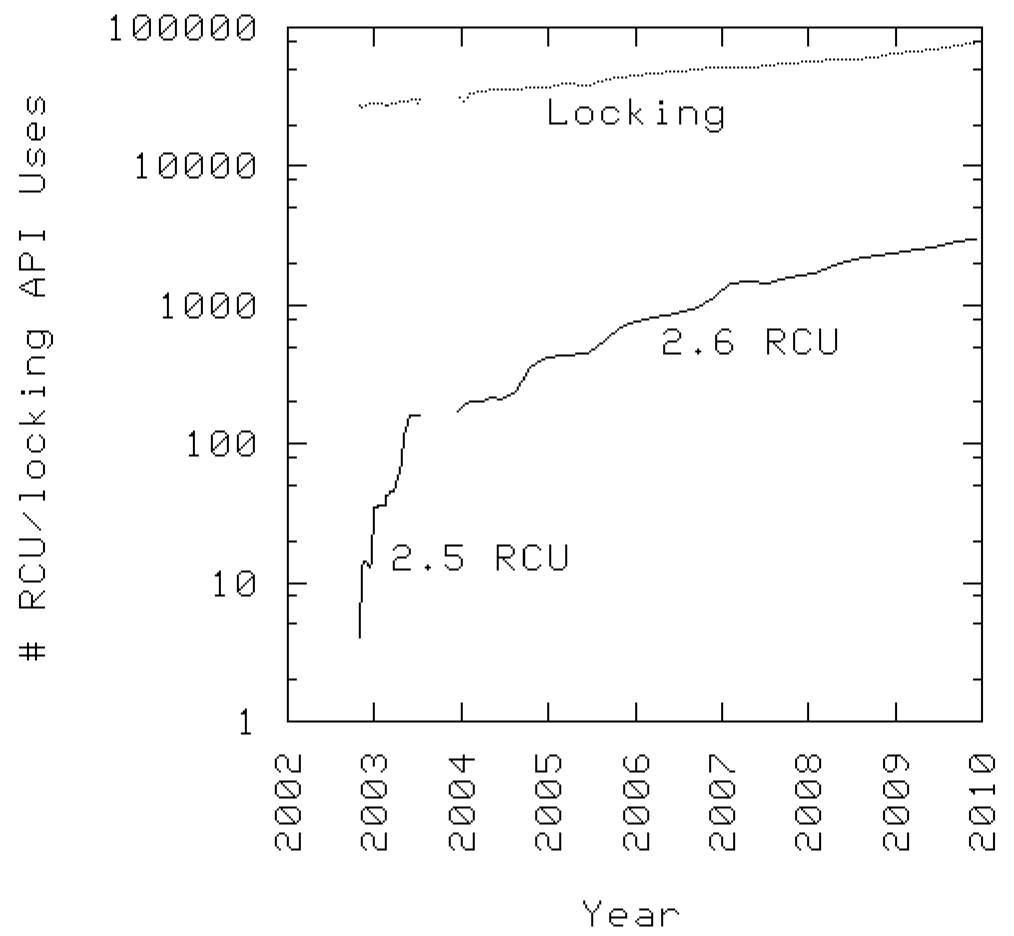
In case there was any doubt, the Linux community ***can*** handle RCU.

RCU Usage Within the Linux Kernel vs. Locking



I do not expect RCU to ever overtake locking because RCU is specialized.

RCU Usage Within the Linux Kernel vs. Locking



Same data on semi-log plot.

RCU 1993-2008: Example Optimization Gone Bad

- **Focus on the complexity of RCU's read side**
- **Before Linux, the ultimate in simplicity:**

```
#define rcu_read_lock()
#define rcu_read_unlock()
```

- **This implementation has a number of advantages...**

RCU 1993-2008: Example Optimization Gone Bad

- **Focus on the complexity of RCU's read side**
- **Before Linux, the ultimate in simplicity:**

```
#define rcu_read_lock()
#define rcu_read_unlock()
```

- **This implementation has a number of advantages:**
  - Good performance, scalability, real-time latency
  - Immunity to deadlock and livelock

RCU 1993-2008: Example Optimization Gone Bad

- **Focus on the complexity of RCU's read side**
- **Before Linux, the ultimate in simplicity:**

```
#define rcu_read_lock()
#define rcu_read_unlock()
```

- **This implementation has a number of advantages:**
  - Good performance, scalability, real-time latency
  - Immunity to deadlock and livelock
  - In short, “free is a very good price”

RCU 1993-2008: Example Optimization Gone Bad

- **Focus on the complexity of RCU's read side**
- **Before Linux, the ultimate in simplicity:**

```
#define rcu_read_lock()
#define rcu_read_unlock()
```

- **This implementation has a number of advantages:**
  - Good performance, scalability, real-time latency
  - Immunity to deadlock and livelock
  - In short, “free is a very good price”
- **But if the readers are doing absolutely nothing, how the heck does the writer know when they are done?**

How Can RCU Readers Do Nothing???

- **If the readers are doing absolutely nothing, how the heck does the writer know when they are done?**
  - The do-nothing readers work only if !CONFIG_PREEMPT
    - Or at user level if your application is appropriately structured

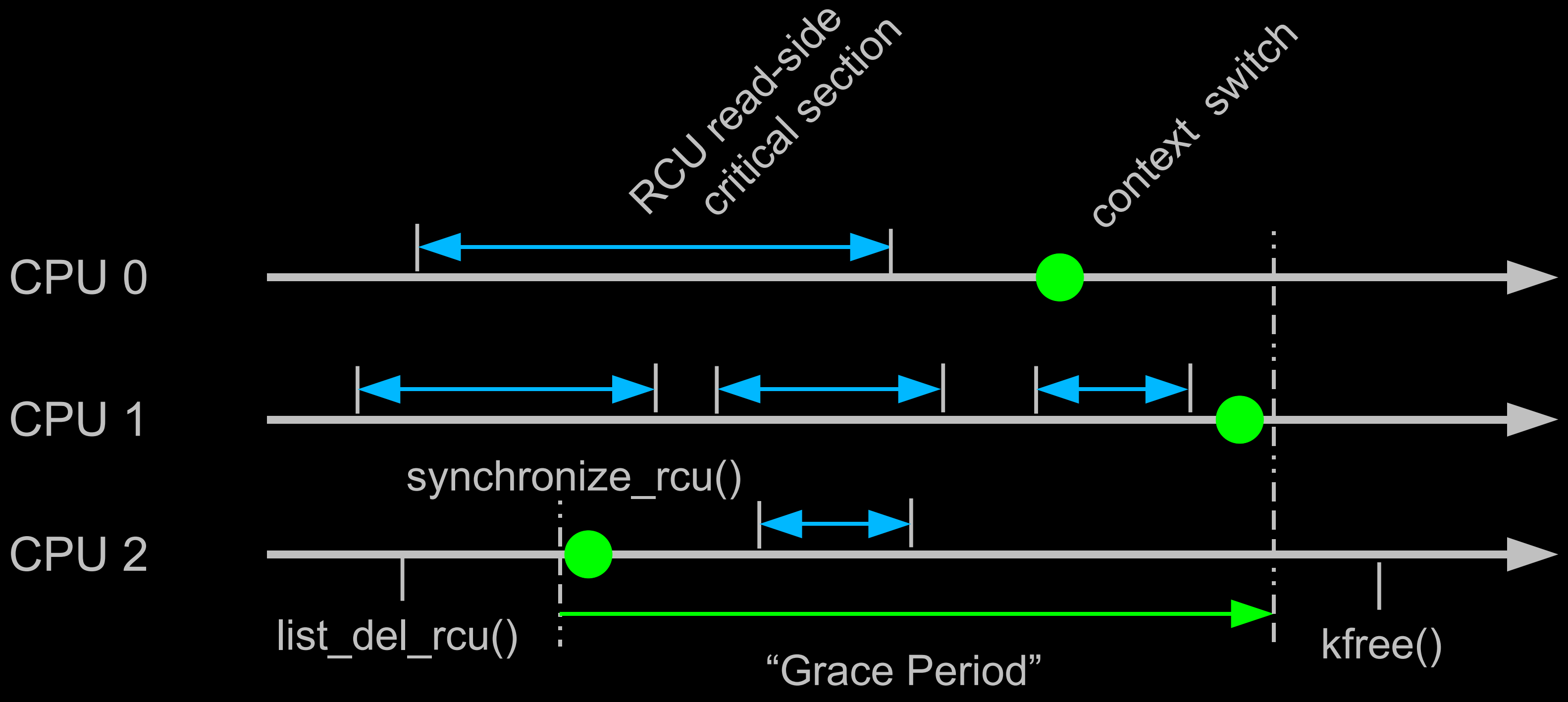
How Can RCU Readers Do Nothing???

- **If the readers are doing absolutely nothing, how the heck does the writer know when they are done?**
  - The do-nothing readers work only if !CONFIG_PREEMPT
    - Or at user level if your application is appropriately structured
  - In kernel, cannot block or preempt while holding a spinlock
    - Try it. If the guy holding the spinlock stops running, then all CPUs can be tied up spinning on the lock. The spinlock cannot be released until the guy holding it runs, and the guy holding it cannot run until at least one CPU becomes available, which won't happen until he releases the lock.
    - Self deadlock and “scheduling while atomic” console messages

How Can RCU Readers Do Nothing???

- **If the readers are doing absolutely nothing, how the heck does the writer know when they are done?**
  - The do-nothing readers work only if !CONFIG_PREEMPT
    - Or at user level if your application is appropriately structured
  - In kernel, cannot block or preempt while holding a spinlock
    - Try it. If the guy holding the spinlock stops running, then all CPUs can be tied up spinning on the lock. The spinlock cannot be released until the guy holding it runs, and the guy holding it cannot run until at least one CPU becomes available, which won't happen until he releases the lock.
    - Self deadlock and “scheduling while atomic” console messages
  - So, declare it illegal to block or be preempted while in an RCU read-side critical section!

Accommodating Lazy RCU Readers



Where to Find out More About RCU

- **Linux Device Drivers, J. Corbet, A. Rubini, G. Kroah-Hartman**
- **Linux Weekly News: lwn.net (Google for “rcu whatever site:lwn.net”)**
- **Linux Kernel source (http://kernel.org/pub/linux/kernel/v2.6/linux-2.6.30.tar.bz2)**
  - “Documentation/RCU” 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://lttng.org/?q=node/18**
  - Mathieu Desnoyers's user-level RCU repository
- **http://www.rdrop.com/users/paulmck/RCU/ (More RCU information)**

RCU 1993-2008: Example Optimization Gone Bad

- **Focus on the complexity of RCU's read side**
- **Before Linux, the ultimate in simplicity:**

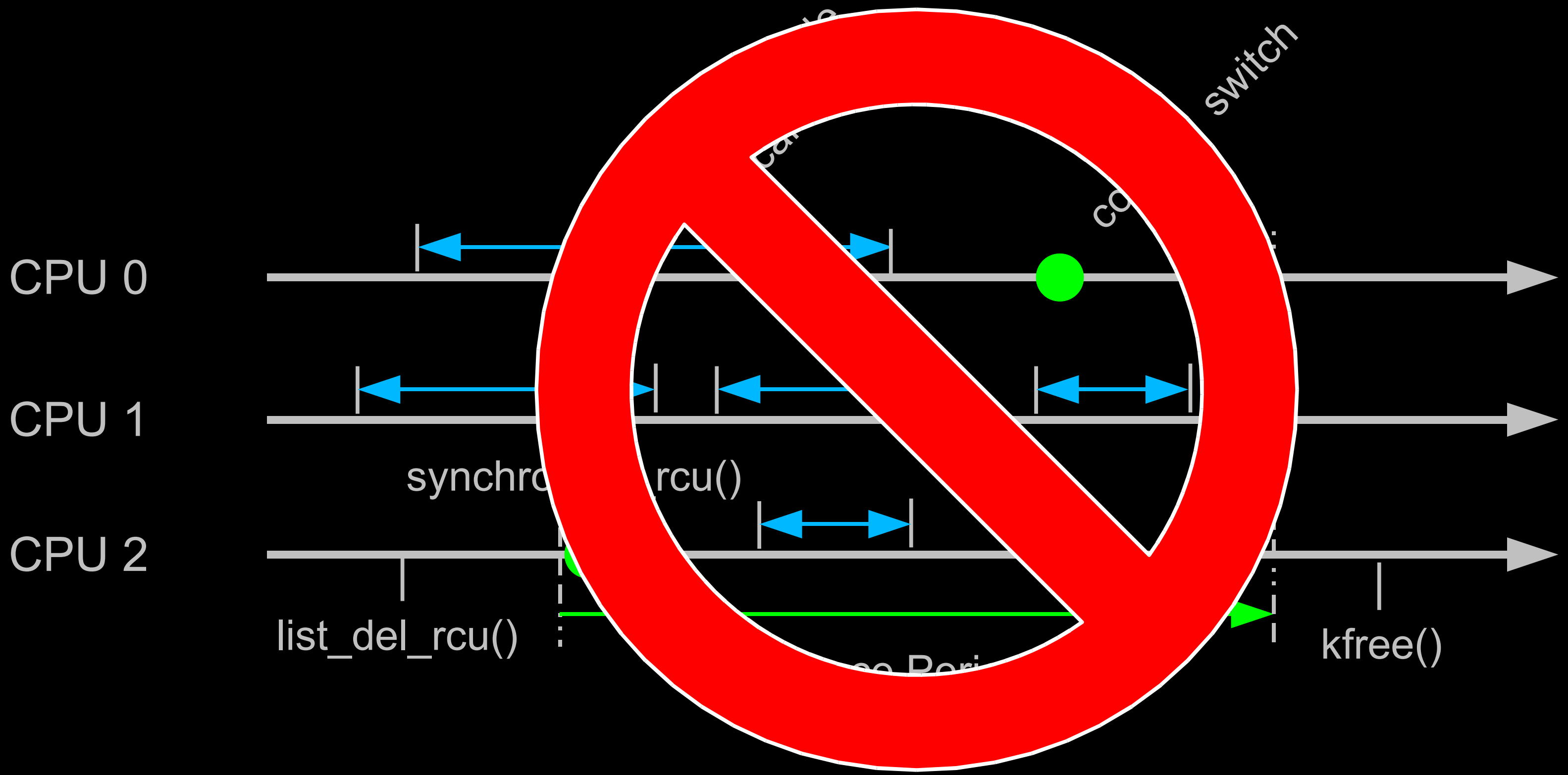
```
#define rcu_read_lock()
#define rcu_read_unlock()
```

- **Linux 2.6.0 had CONFIG_PREEMPT, still simple:**

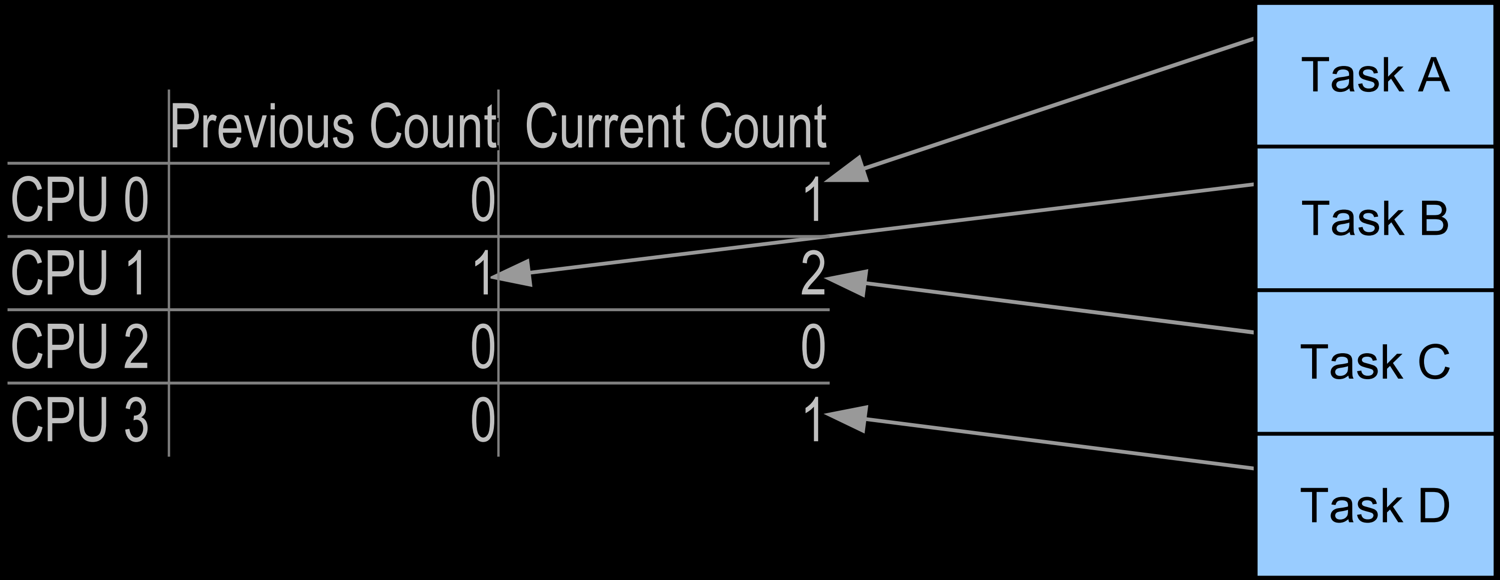
```
#define rcu_read_lock()     preempt_disable()
#define rcu_read_unlock()   preempt_enable()
```

- **And then there was the -rt patchset...**
  - Which needs to preempt RCU readers
  - Which...

Context Switches Might Not Be Quiescent State



Counter-Based Real-Time RCU



	Previous Count	Current Count
CPU 0	0	1
CPU 1	1	2
CPU 2	0	0
CPU 3	0	1

Each task references the counter that it atomically increments in rcu_read_lock(), allowing rcu_read_unlock() to atomically decrement it.

Each task keeps a counter of rcu_read_lock() nesting, so that only outermost rcu_read_lock() and rcu_read_unlock() access per-CPU counters.

Counter-Based Real-Time RCU

	Previous Count	Current Count
CPU 0	0	0
CPU 1	0	0
CPU 2	0	0
CPU 3	0	0

Task A

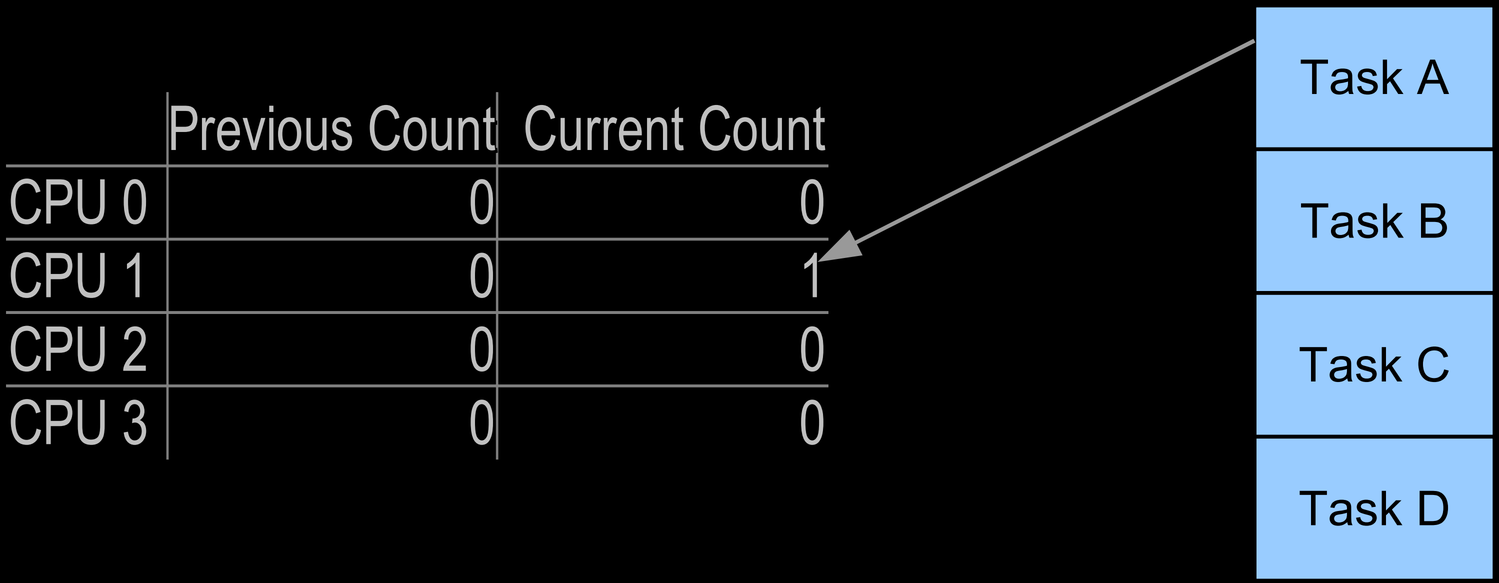
Task B

Task C

Task D

Initial state.

Counter-Based Real-Time RCU



	Previous Count	Current Count
CPU 0	0	0
CPU 1	0	1
CPU 2	0	0
CPU 3	0	0

Task A

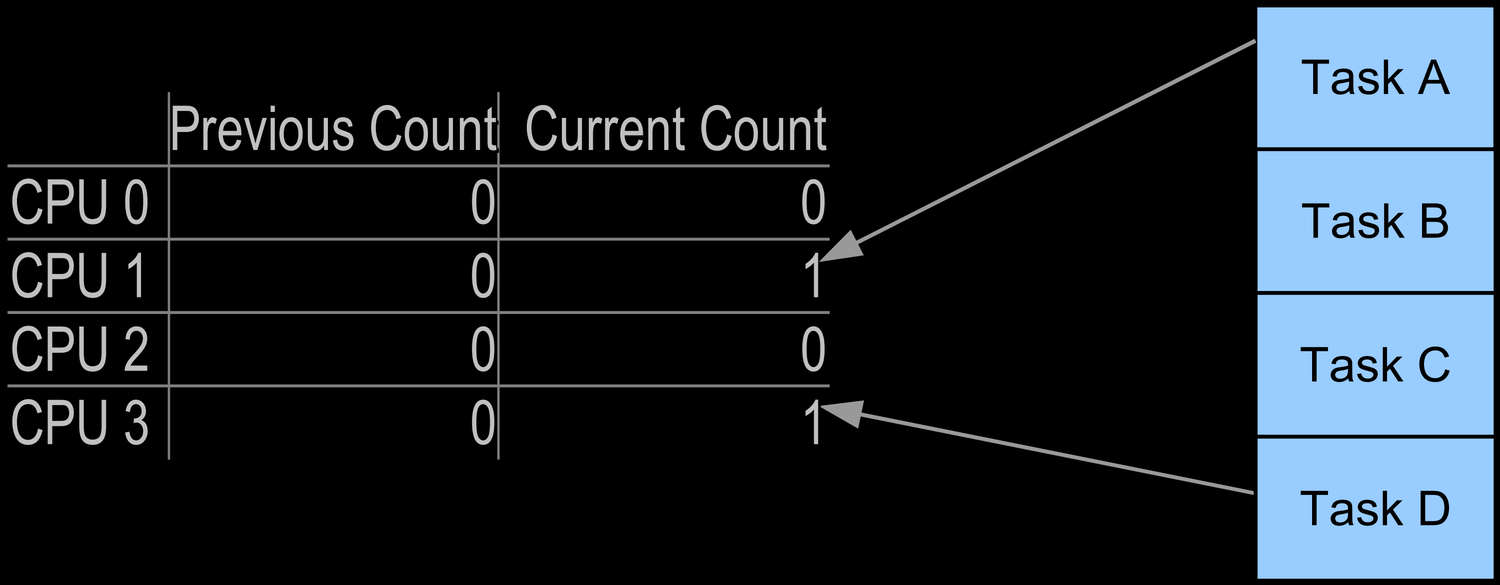
Task B

Task C

Task D

Task A rcu_read_lock().

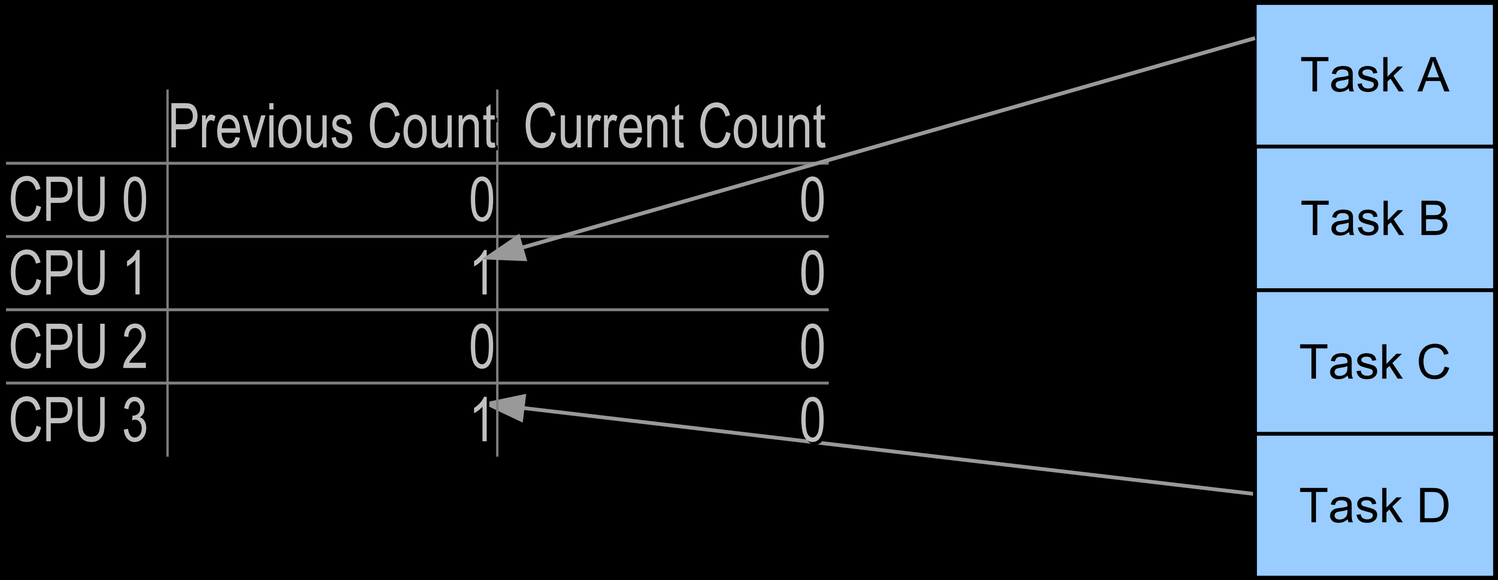
Counter-Based Real-Time RCU



	Previous Count	Current Count
CPU 0	0	0
CPU 1	0	1
CPU 2	0	0
CPU 3	0	1

Task D rcu_read_lock().

Counter-Based Real-Time RCU

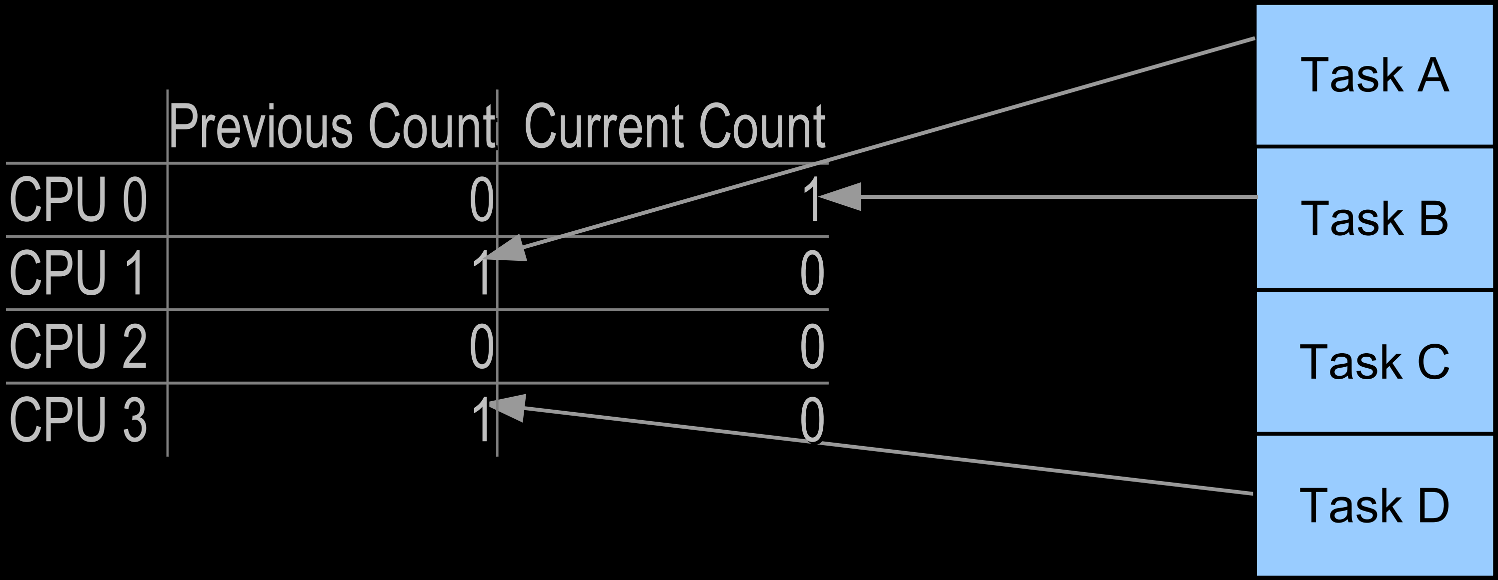


	Previous Count	Current Count
CPU 0	0	0
CPU 1	1	0
CPU 2	0	0
CPU 3	1	0

- Task A
- Task B
- Task C
- Task D

Task C synchronize_rcu() entry: Counters “flip”, or reverse roles.

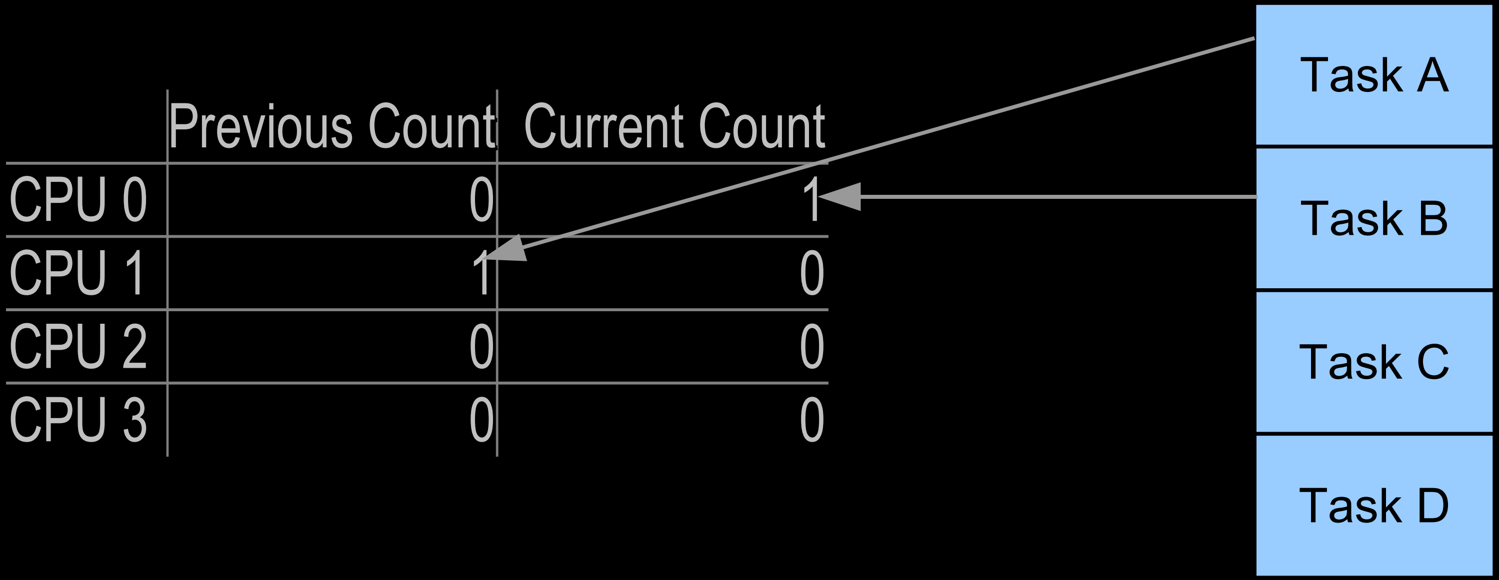
Counter-Based Real-Time RCU



	Previous Count	Current Count
CPU 0	0	1
CPU 1	1	0
CPU 2	0	0
CPU 3	1	0

Task B rcu_read_lock().

Counter-Based Real-Time RCU



	Previous Count	Current Count
CPU 0	0	1
CPU 1	1	0
CPU 2	0	0
CPU 3	0	0

Task A

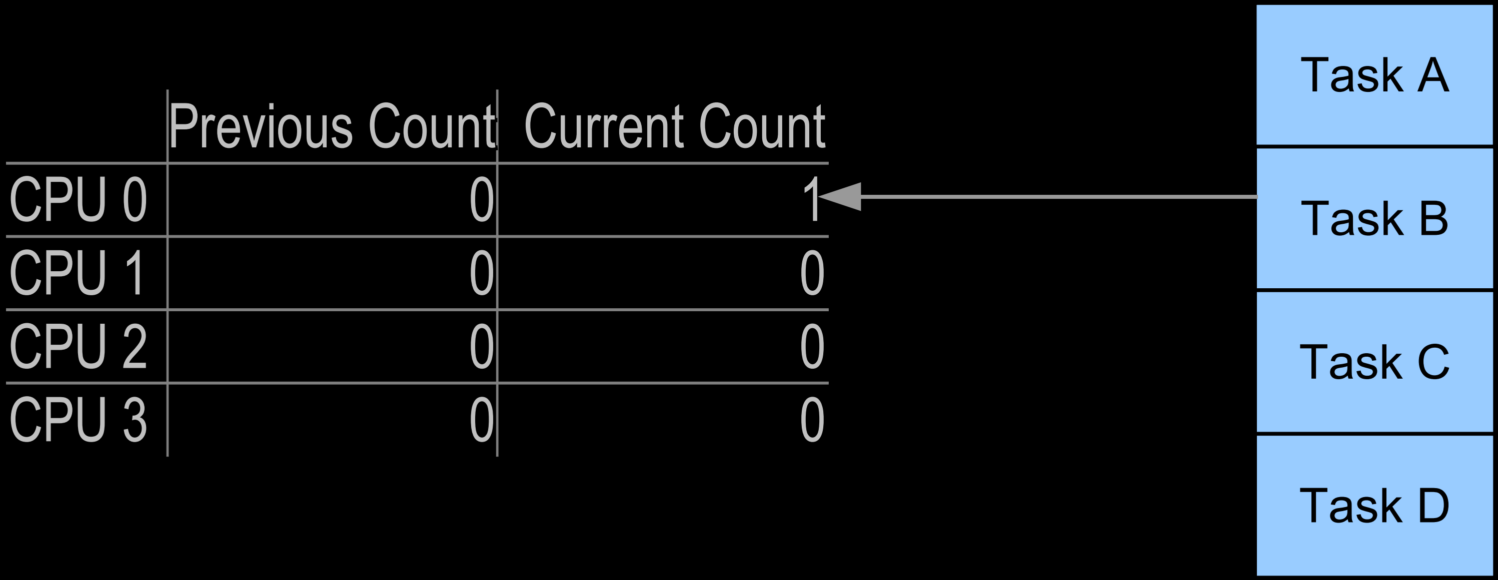
Task B

Task C

Task D

Task D rcu_read_unlock().

Counter-Based Real-Time RCU



	Previous Count	Current Count
CPU 0	0	1
CPU 1	0	0
CPU 2	0	0
CPU 3	0	0

Task A rcu_read_unlock(), Task C synchronize_rcu() returns.

Counter-Based Real-Time RCU

	Previous Count	Current Count
CPU 0	0	0
CPU 1	0	0
CPU 2	0	0
CPU 3	0	0

Task A

Task B

Task C

Task D

Task B rcu_read_unlock().

But what must rcu_read_lock() and rcu_read_unlock() do to make this work?

(For more info: http://www.rdrop.com/users/paulmck/RCU/OLSrtRCU.2006.08.11a.pdf)

RCU 1993-2008: Example Optimization Gone Bad

- **-rt patchset needed preemptible RCU readers:**

```
void rcu_read_lock(void)
{
  int flipctr;
  unsigned long oldirq;

  local_irq_save(oldirq);
  if (current->rcu_read_lock_nesting++ == 0) {
    flipctr = rcu_ctrlblk.completed & 0x1;
    smp_read_barrier_depends();
    current->rcu_flipctr1 = &(__get_cpu_var(rcu_flipctr)[flipctr]);
    atomic_inc(current->rcu_flipctr1);
    smp_mb__after_atomic_inc();  /* might optimize out... */
    if (unlikely(flipctr != (rcu_ctrlblk.completed & 0x1))) {
      current->rcu_flipctr2 =
        &(__get_cpu_var(rcu_flipctr)[!flipctr]);
      atomic_inc(current->rcu_flipctr2);
      smp_mb__after_atomic_inc();  /* might optimize out... */
    }
  }
  local_irq_restore(oldirq);
}
```

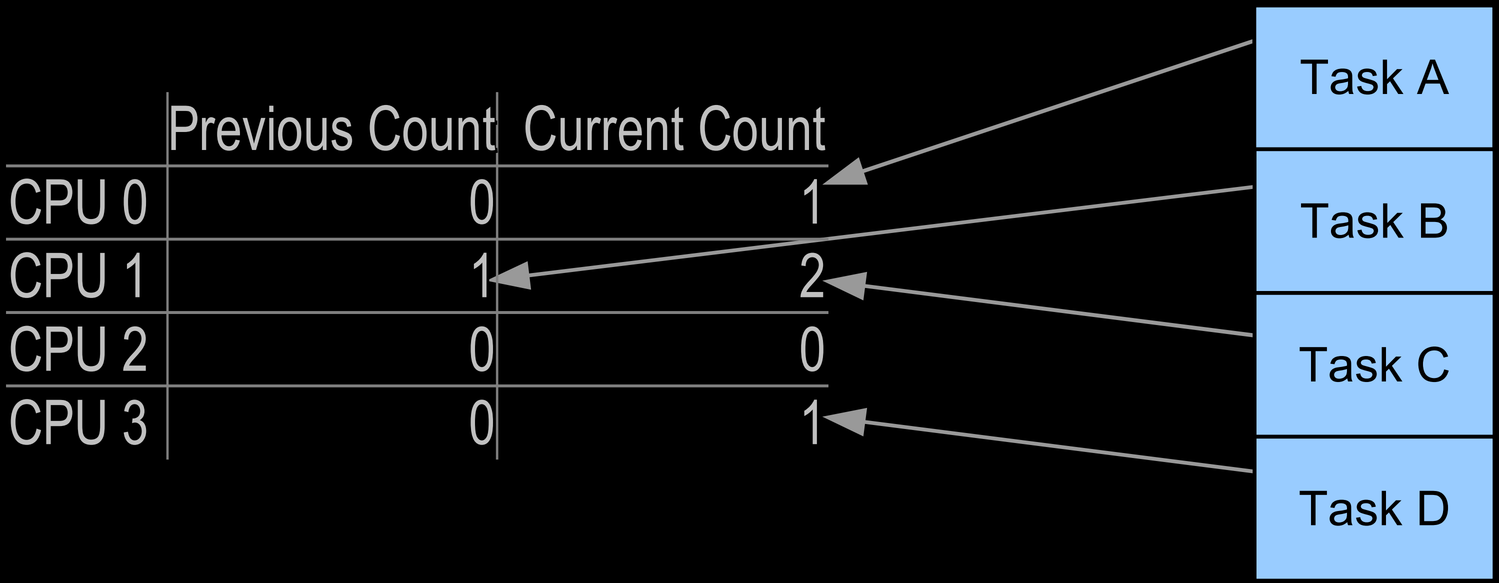
RCU 1993-2008: Example Optimization Gone Bad

- And -rt patchset also needs rcu_read_unlock():

```
void
rcu_read_unlock(void)
{
  unsigned long oldirq;

  local_irq_save(oldirq);
  if (--current->rcu_read_lock_nesting == 0) {
    smp_mb__before_atomic_dec();
    atomic_dec(current->rcu_flipctr1);
    current->rcu_flipctr1 = NULL;
    if (unlikely(current->rcu_flipctr2 != NULL)) {
      atomic_dec(current->rcu_flipctr2);
      current->rcu_flipctr2 = NULL;
    }
  }
  local_irq_restore(oldirq);
}
```

Atomic operations, memory barriers, common-case branches: yecch!!!

Real-Time RCU Without Memory Barriers



	Previous Count	Current Count
CPU 0	0	1
CPU 1	1	2
CPU 2	0	0
CPU 3	0	1

Each task references the column of the counter that it incremented in rcu_read_lock(), allowing rcu_read_unlock() to decrement the corresponding counter corresponding to whatever CPU it ends up on.

Again, each task keeps a nesting counter.

Real-Time RCU Without Memory Barriers

	Previous Count	Current Count
CPU 0	0	0
CPU 1	0	0
CPU 2	0	0
CPU 3	0	0

Task A

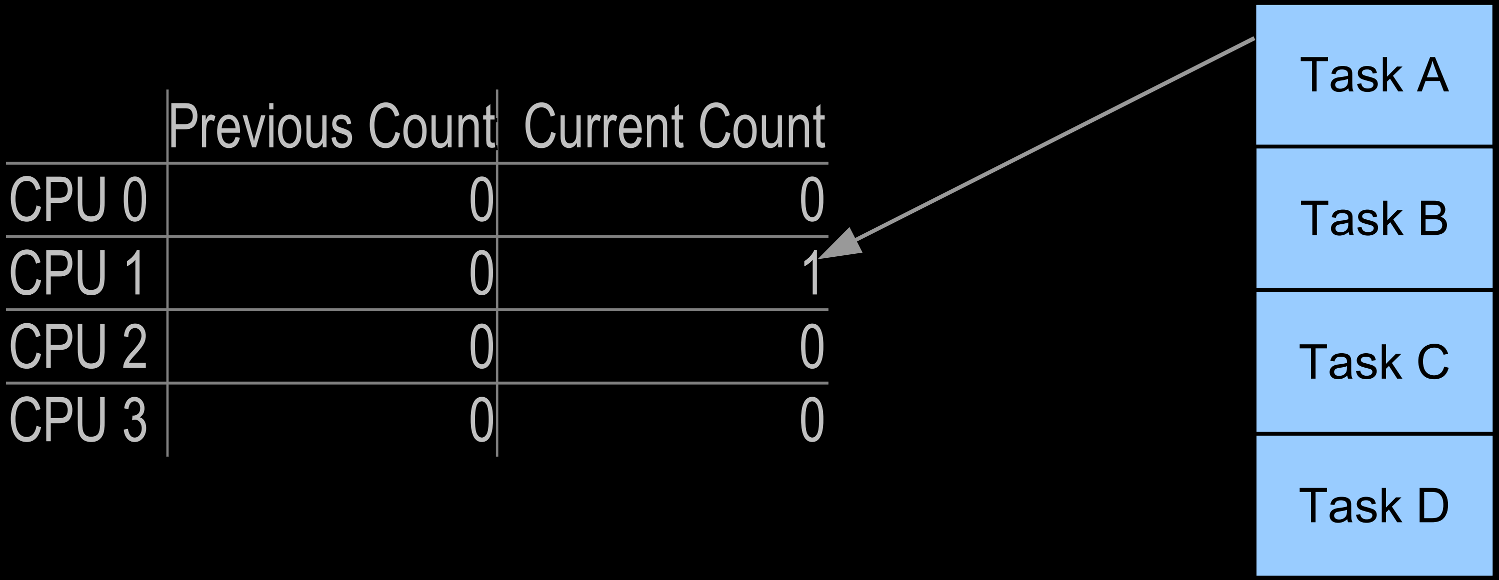
Task B

Task C

Task D

Initial state.

Real-Time RCU Without Memory Barriers



	Previous Count	Current Count
CPU 0	0	0
CPU 1	0	1
CPU 2	0	0
CPU 3	0	0

Task A

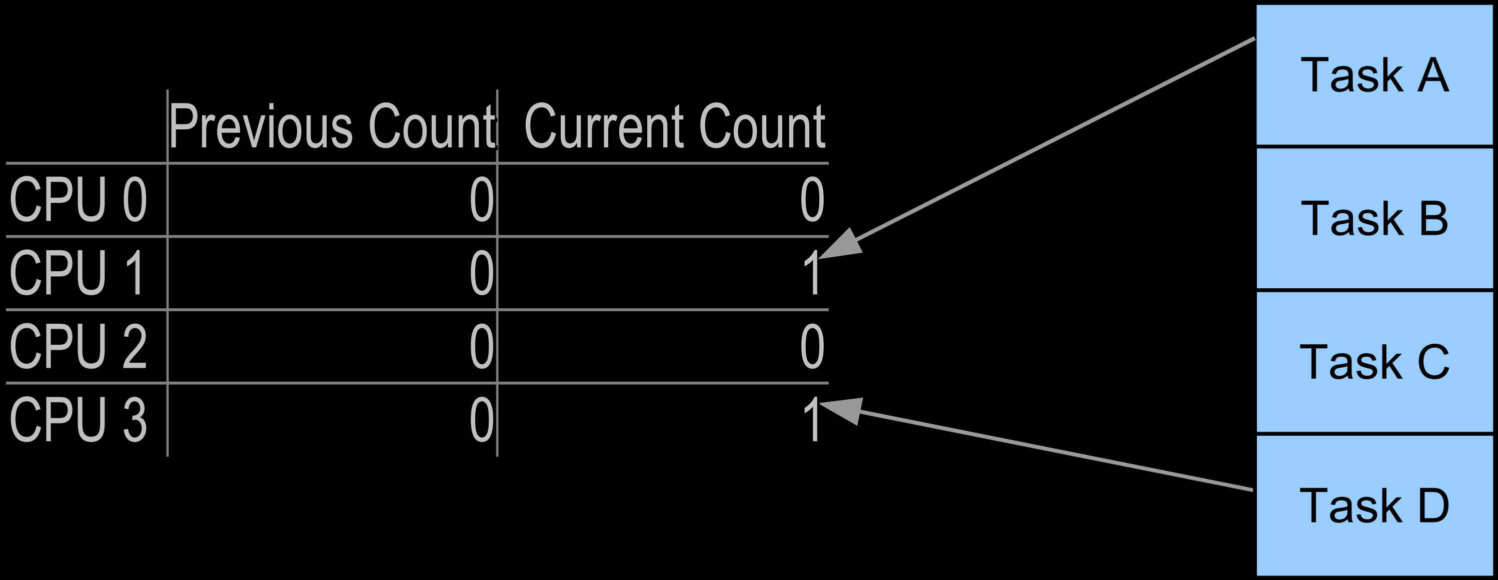
Task B

Task C

Task D

Task A rcu_read_lock().

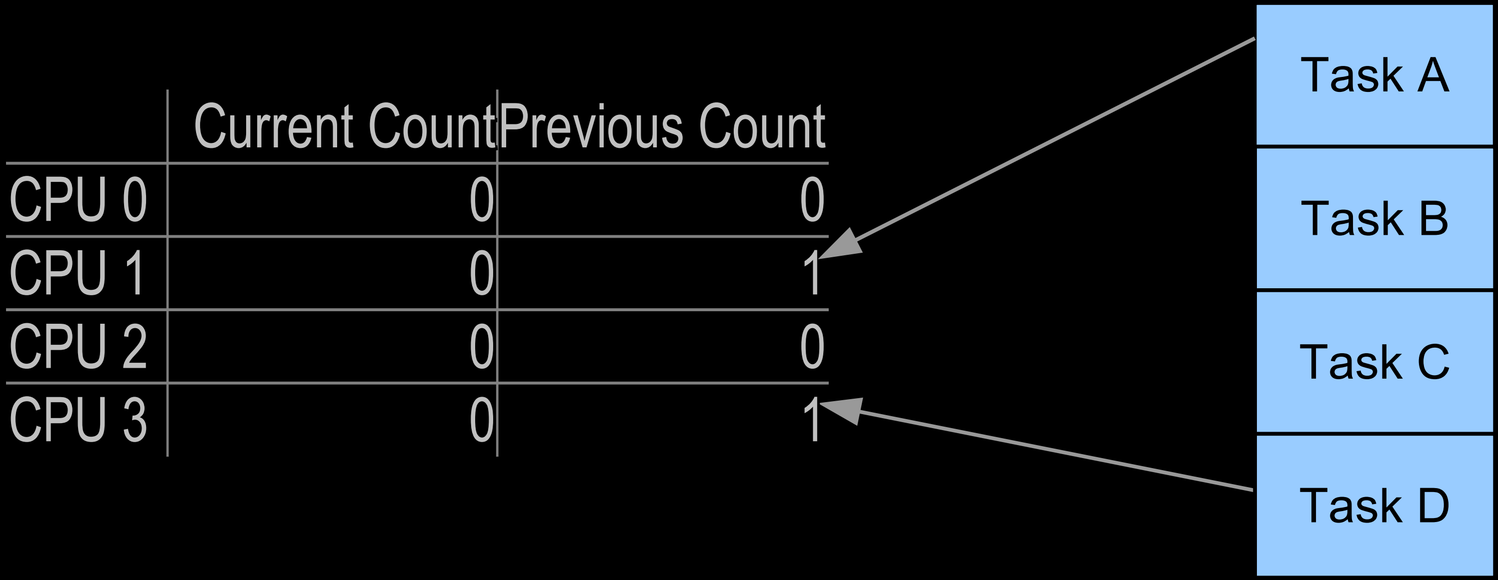
Real-Time RCU Without Memory Barriers



	Previous Count	Current Count
CPU 0	0	0
CPU 1	0	1
CPU 2	0	0
CPU 3	0	1

Task D rcu_read_lock().

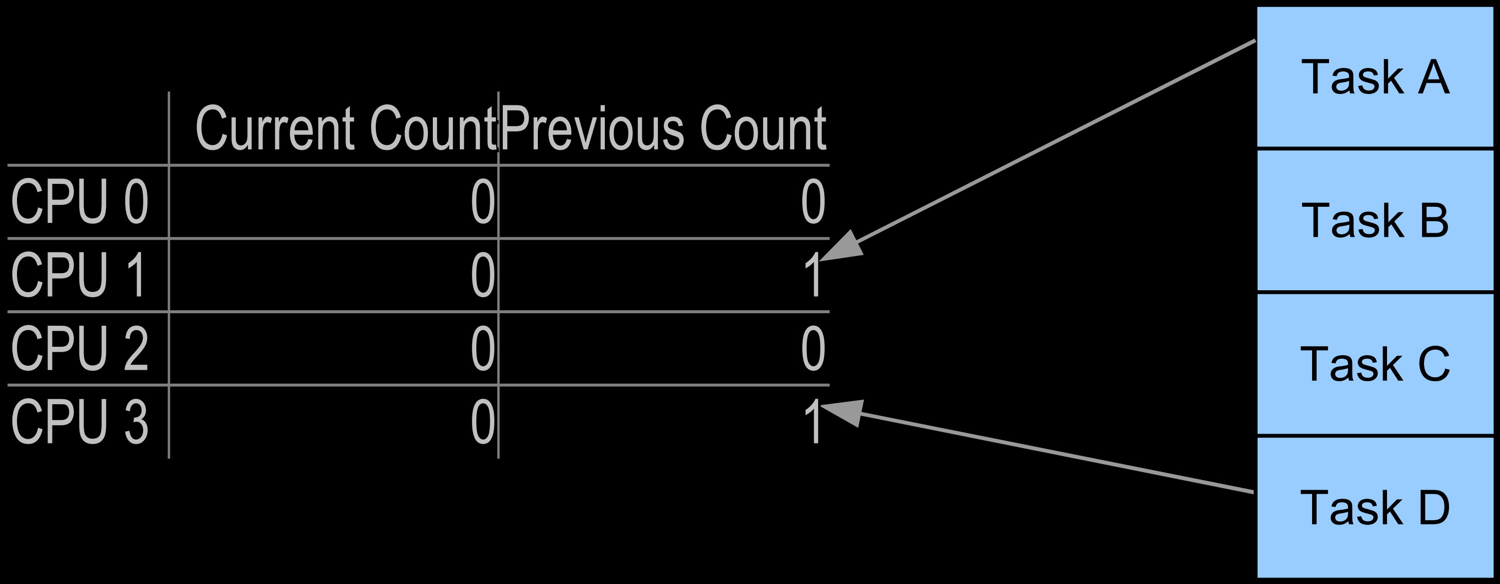
Real-Time RCU Without Memory Barriers



	Current Count	Previous Count
CPU 0	0	0
CPU 1	0	1
CPU 2	0	0
CPU 3	0	1

Task C synchronize_rcu() entry: Counters “flip”, or reverse roles.

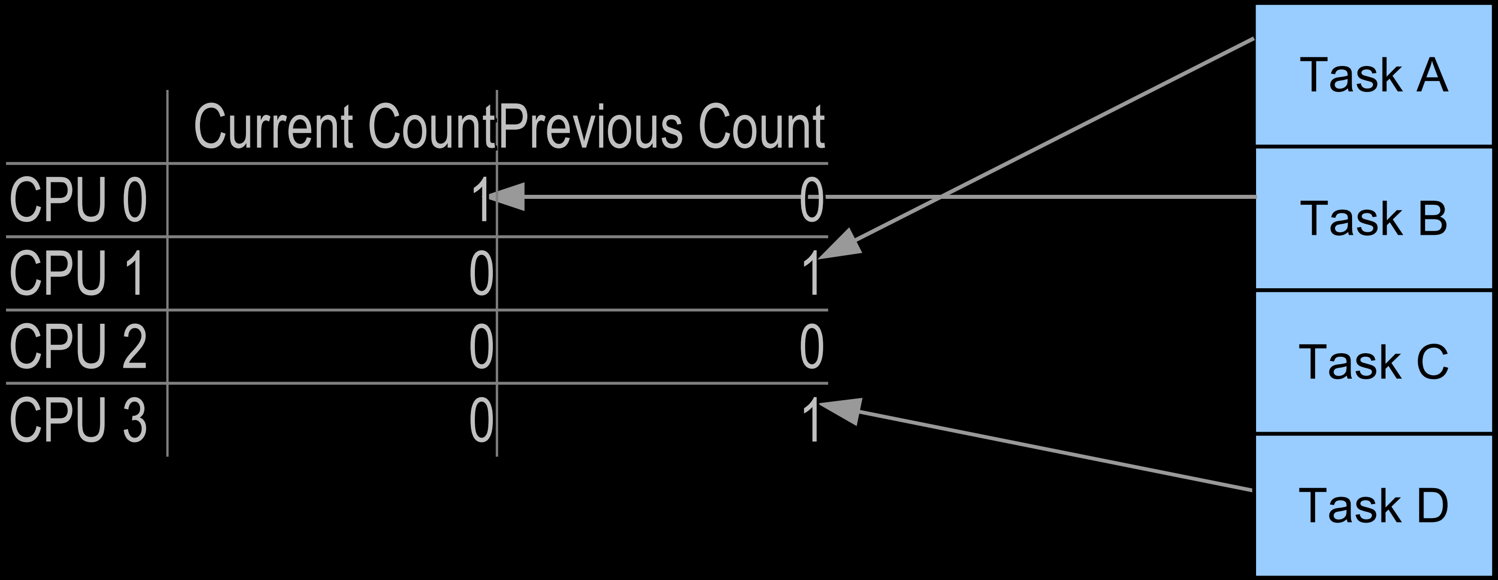
Real-Time RCU Without Memory Barriers



	Current Count	Previous Count
CPU 0	0	0
CPU 1	0	1
CPU 2	0	0
CPU 3	0	1

Task A is preempted, then resumes on CPU 2.

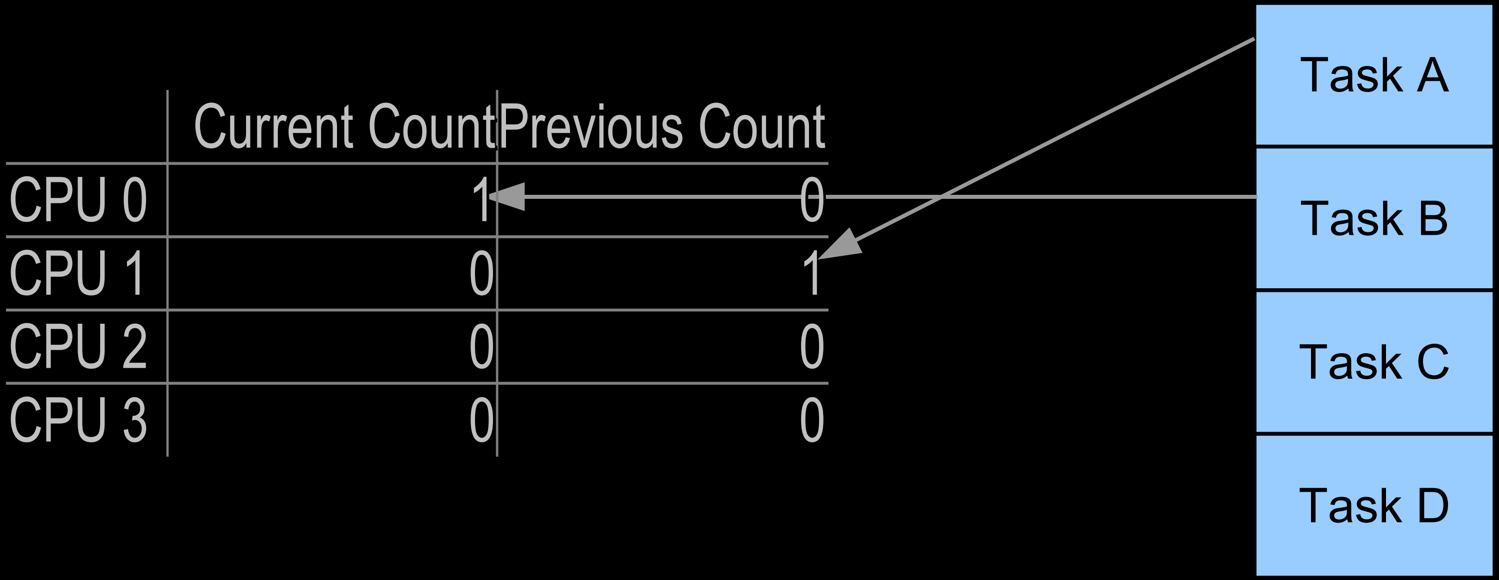
Real-Time RCU Without Memory Barriers



	Current Count	Previous Count
CPU 0	1	0
CPU 1	0	1
CPU 2	0	0
CPU 3	0	1

Task B rcu_read_lock().

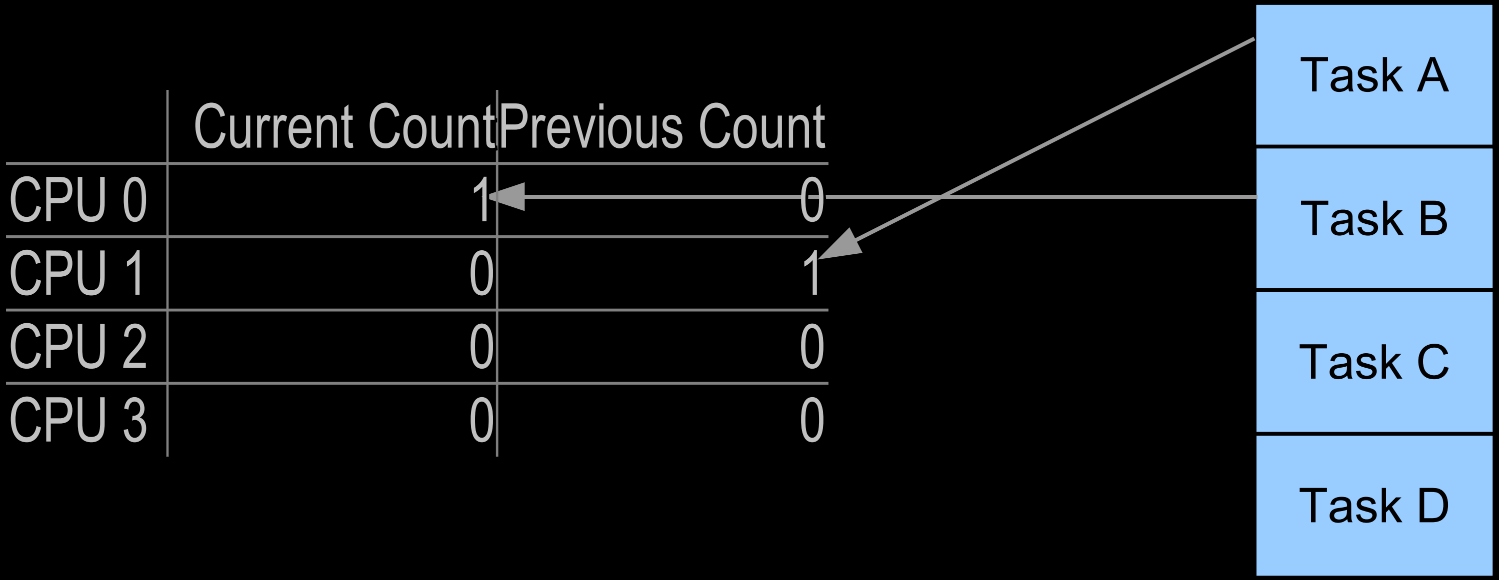
Real-Time RCU Without Memory Barriers



	Current Count	Previous Count
CPU 0	1	0
CPU 1	0	1
CPU 2	0	0
CPU 3	0	0

Task D rcu_read_unlock().

Real-Time RCU Without Memory Barriers

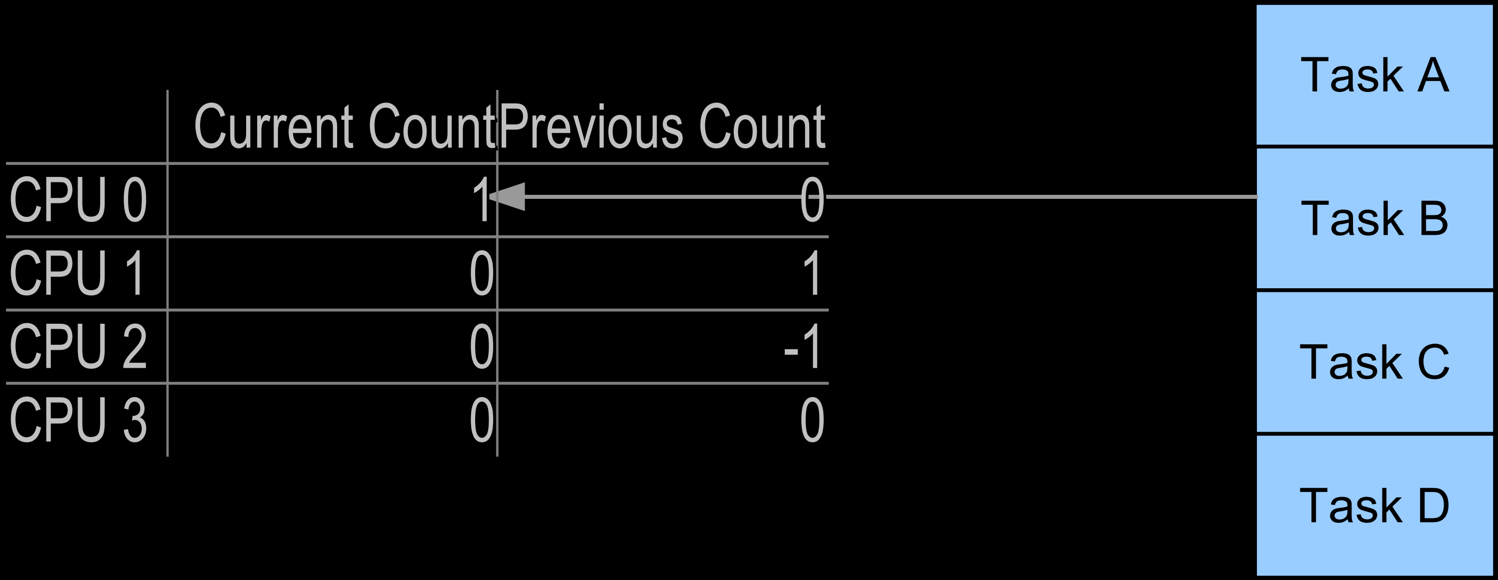


	Current Count	Previous Count
CPU 0	1	0
CPU 1	0	1
CPU 2	0	0
CPU 3	0	0

Recall that Task A is now running on CPU 2.

So, what happens when Task A does rcu_read_unlock()?

Real-Time RCU Without Memory Barriers



	Current Count	Previous Count
CPU 0	1	0
CPU 1	0	1
CPU 2	0	-1
CPU 3	0	0

Task A rcu_read_unlock(), Task C synchronize_rcu() returns.

Real-Time RCU Without Memory Barriers

	Current Count	Previous Count
CPU 0	0	0
CPU 1	0	1
CPU 2	0	-1
CPU 3	0	0

Task A
Task B
Task C
Task D

Task B rcu_read_unlock().

(For more info: http://lwn.net/Articles/253651/)

RCU 1993-2008: Example Optimization Gone Bad

- **2.6.25 preemptible rcu_read_lock():**

```
void __rcu_read_lock(void)
{
  int idx;
  struct task_struct *t = current;
  int nesting;

  nesting = ACCESS_ONCE(t->rcu_read_lock_nesting);
  if (nesting != 0) {
    t->rcu_read_lock_nesting = nesting + 1;
  } else {
    unsigned long flags;
    local_irq_save(flags);
    idx = ACCESS_ONCE(rcu_ctrlblk.completed) & 0x1;
    ACCESS_ONCE(RCU_DATA_ME()->rcu_flipctr[idx])++;
    ACCESS_ONCE(t->rcu_read_lock_nesting) = nesting + 1;
    ACCESS_ONCE(t->rcu_flipctr_idx) = idx;
    local_irq_restore(flags);
  }
}
```

RCU 1993-2008: Example Optimization Gone Bad

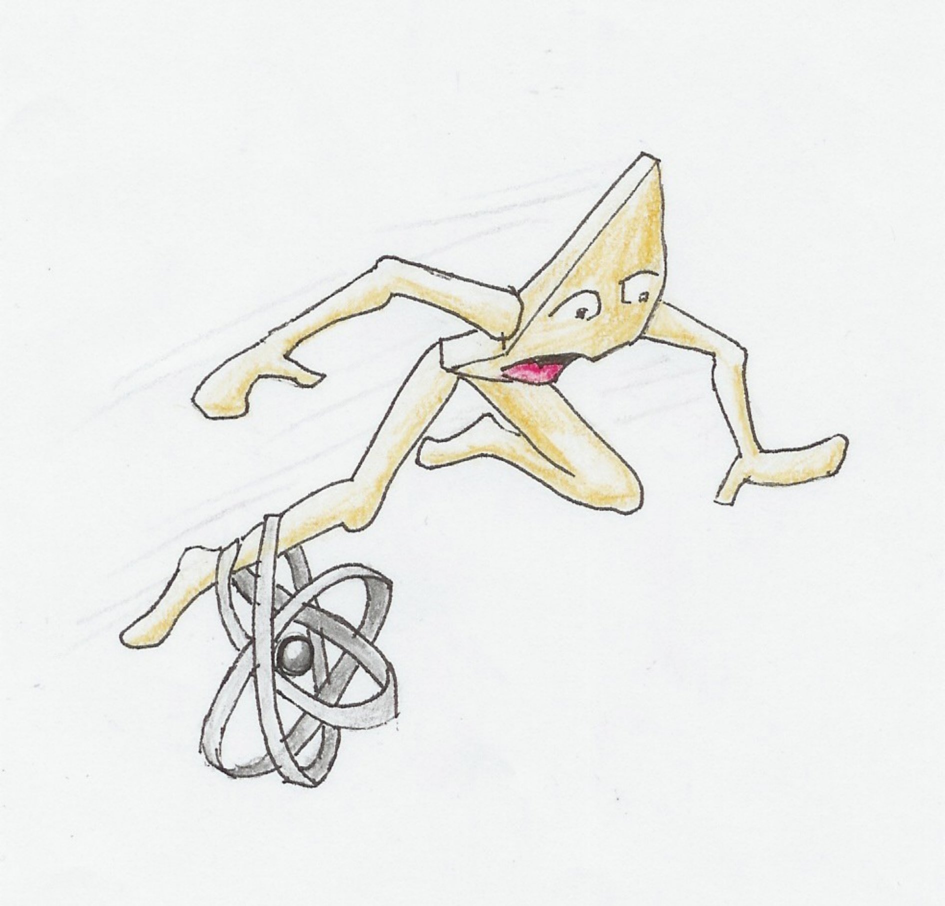
- **2.6.25 preemptible rcu_read_unlock():**

```
void __rcu_read_unlock(void)
{
  int idx;
  struct task_struct *t = current;
  int nesting;

  nesting = ACCESS_ONCE(t->rcu_read_lock_nesting);
  if (nesting > 1) {
    t->rcu_read_lock_nesting = nesting - 1;
  } else {
    unsigned long flags;

    local_irq_save(flags);
    idx = ACCESS_ONCE(t->rcu_flipctr_idx);
    ACCESS_ONCE(t->rcu_read_lock_nesting) = nesting - 1;
    ACCESS_ONCE(RCU_DATA_ME()->rcu_flipctr[idx])--;
    local_irq_restore(flags);
  }
}
```

Faster, but still lots of compiler constraints, array accesses, and bulk code. Also difficult to tell which tasks are holding things up.

So, What Do We Want, Anyway???

We Don't Want Atomic Instructions

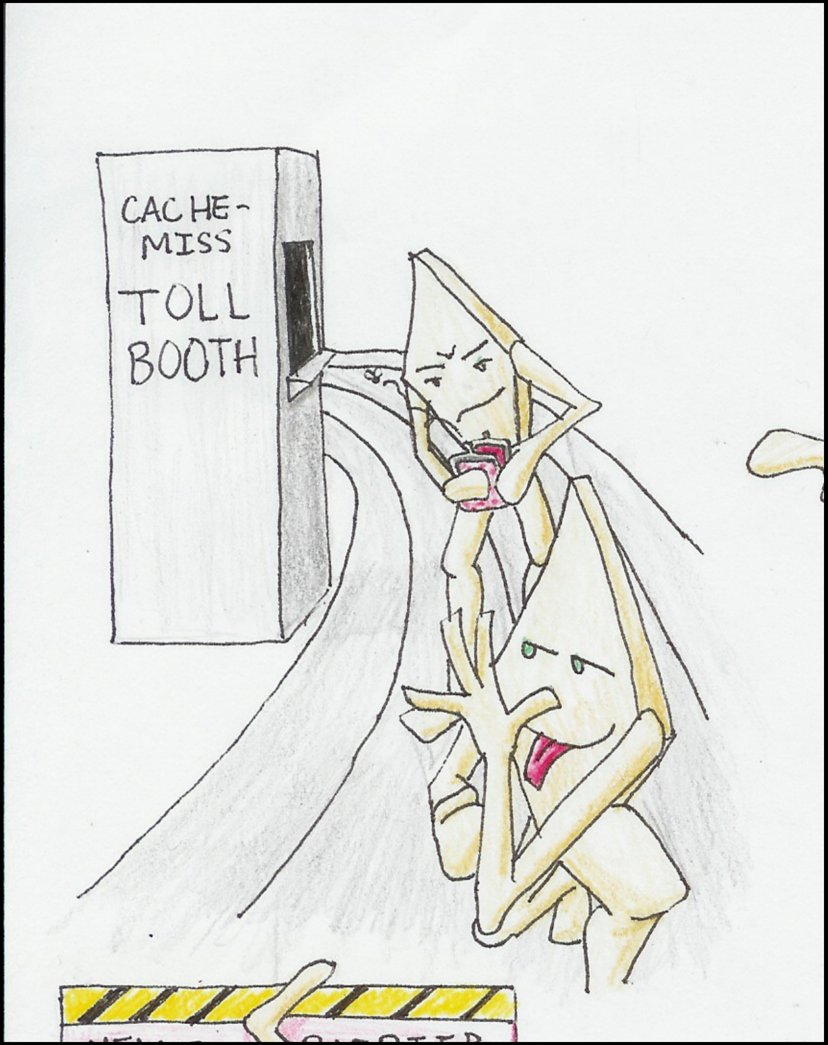


Even though they are cheaper than they used to be.

We Don't Want Memory Barriers

Even though they are also cheaper than they used to be.

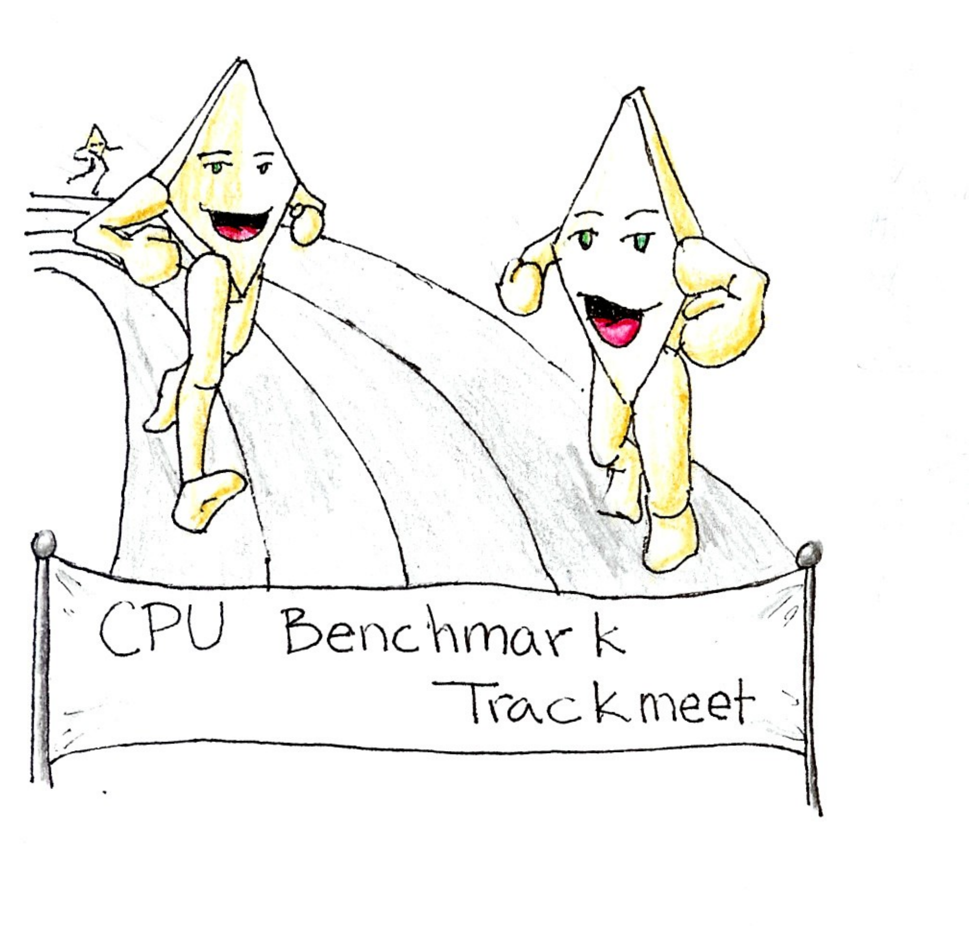
We Don't Want Cache Misses



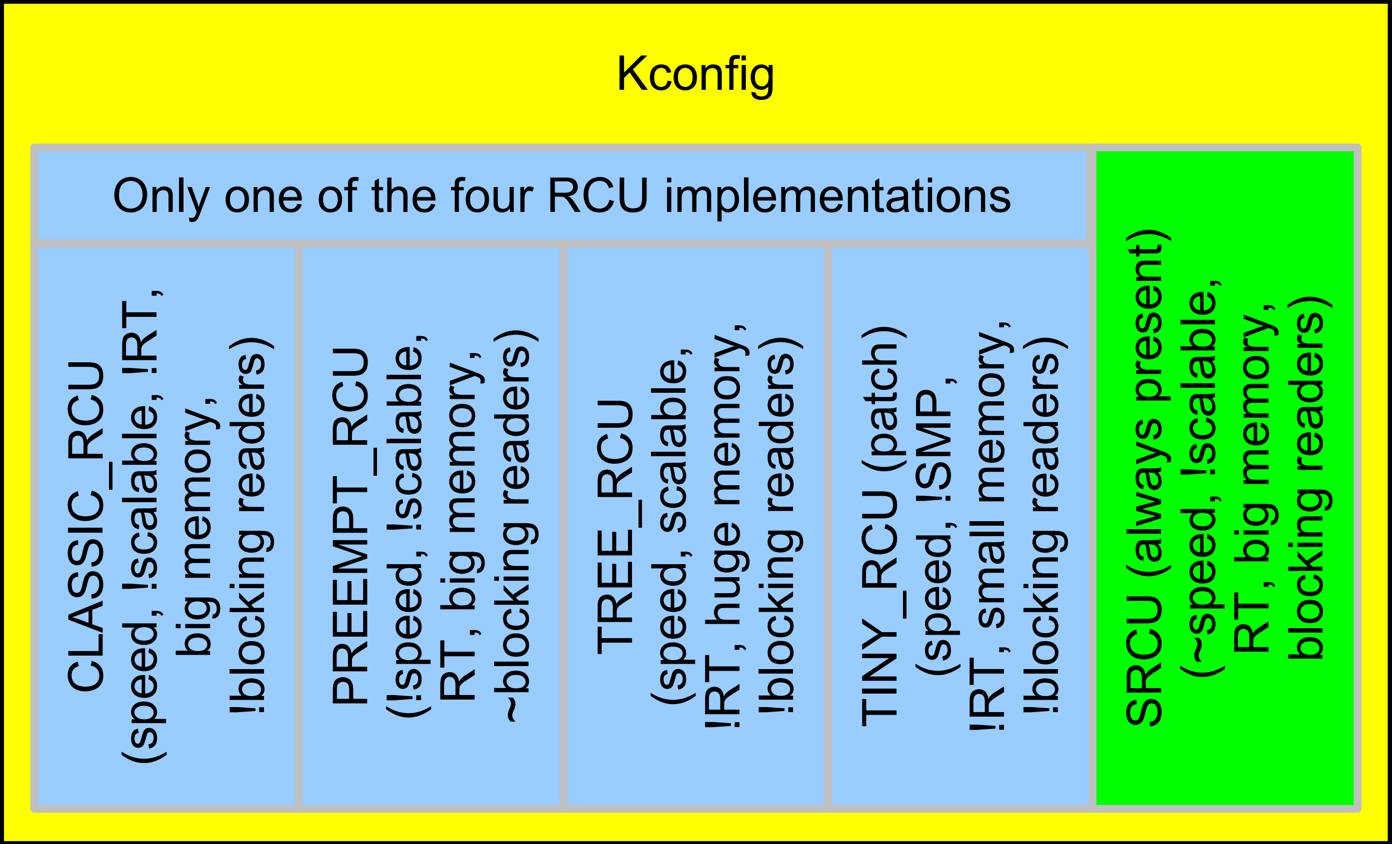
Which haven't really gotten that much cheaper...

We Don't Want Branch Misprediction

We Want Full Core CPU Performance



Don't Want Five Independent RCU Implementations



Kconfig

Only one of the four RCU implementations

CLASSIC_RCU
(speed, !scalable, !RT,
big memory,
!blocking readers)

PREEMPT_RCU
(!speed, !scalable,
RT, big memory,
~blocking readers)

TREE_RCU
(speed, scalable,
!RT, huge memory,
!blocking readers)

TINY_RCU (patch)
(speed, !SMP,
!RT, small memory,
!blocking readers)

SRCU (always present)
(~speed, !scalable,
RT, big memory,
blocking readers)

And Especially...

We Don't Want University Students Learning RCU From “The Design of Preemptible RCU”

RCU 2009-2010: Simplicity Through Optimization

RCU 2009-2010: Simplicity Through Optimization

- **Inspiration: user-level RCU implementations**
  - Inherently preemptible
  - But still simple, because RCU operates on individual threads
  - However, this is not practical for kernel preemptible RCU, due to the potentially huge number of tasks
  - Kernel preemptible RCU therefore does complex per-CPU accounting: modular arithmetic to find grace period end

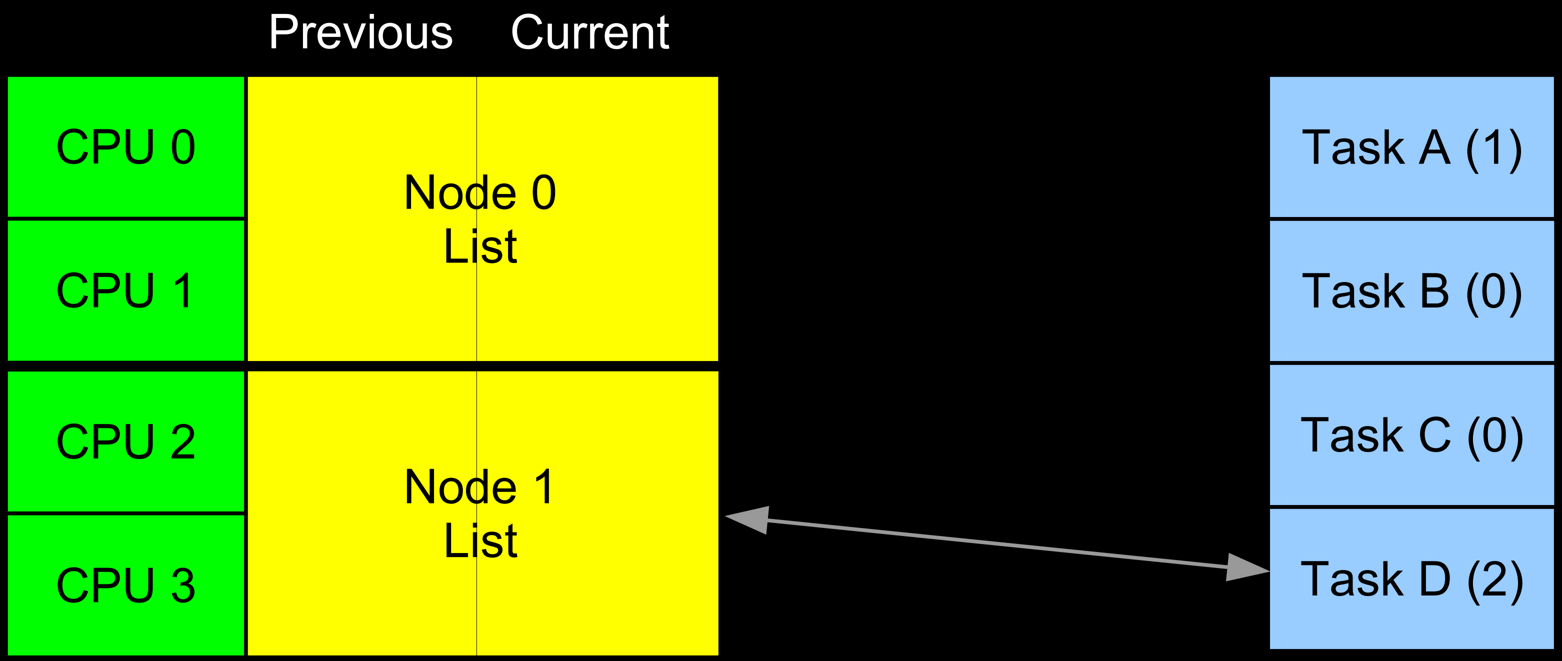
RCU 2009-2010: Simplicity Through Optimization

- **Inspiration: user-level RCU implementations**
  - Inherently preemptible
  - But still simple, because RCU operates on individual threads
  - However, this is not practical for kernel preemptible RCU, due to the potentially huge number of tasks
  - Kernel preemptible RCU therefore does complex per-CPU accounting: modular arithmetic to find grace period end
- **Key idea:**
  - Track *running* tasks
  - When task blocks, context switch is costly anyway
    - So do CPU-level accounting during context-switch events!!!
  - Allows tracking hold-out CPUs, thus integration with hierarchical RCU!!!

RCU 2009-2010: Simplicity Through Optimization

- **Inspiration: user-level RCU implementations**
  - Inherently preemptible
  - But still simple, because RCU operates on individual threads
  - However, this is not practical for kernel preemptible RCU, due to the potentially huge number of tasks
  - Kernel preemptible RCU therefore does complex per-CPU accounting: modular arithmetic to find grace period end
- **Key idea:**
  - Track *running* tasks
  - When task blocks, context switch is costly anyway
    - So do CPU-level accounting during context-switch events!!!
  - Allows tracking hold-out CPUs, thus integration with hierarchical RCU!!!
  - And all due to a horrible performance-measurement mistake

Real-Time RCU The Easy Way

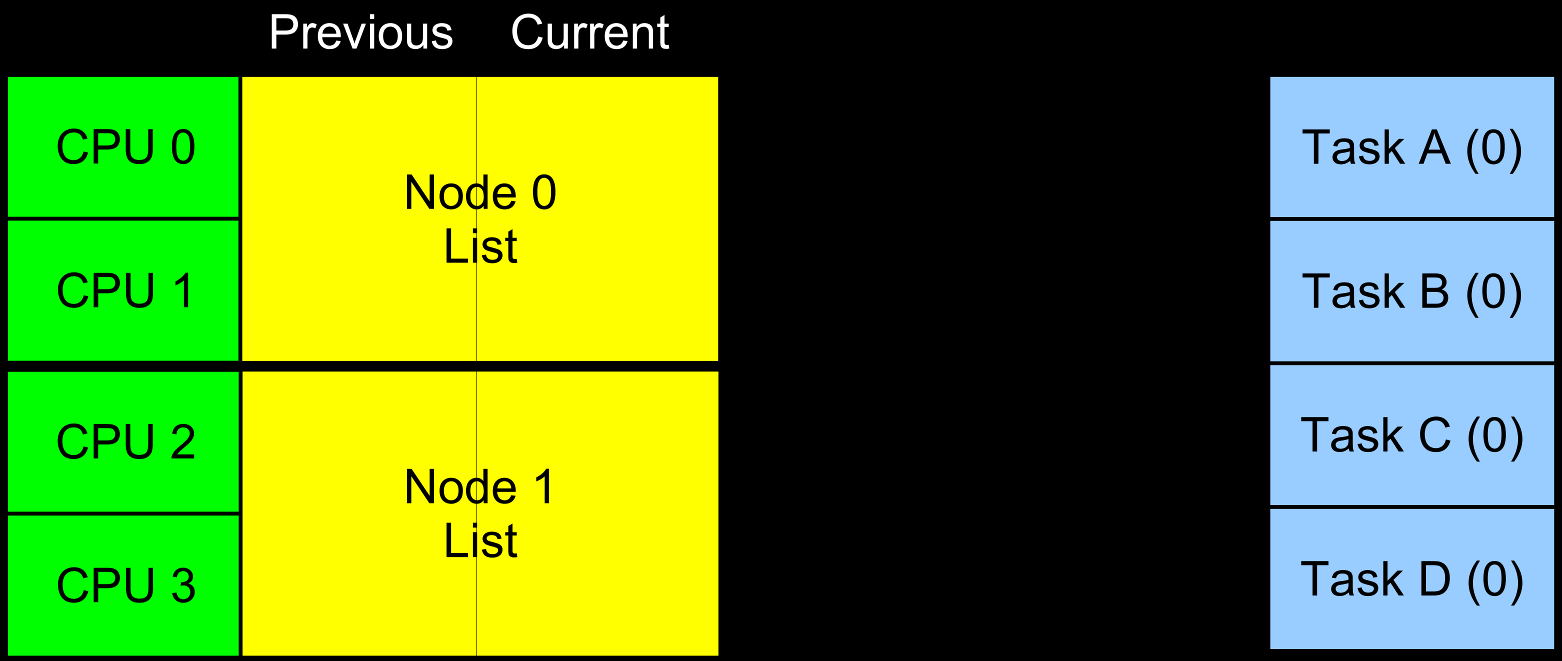


Each task maintains a nesting counter.

Only counters of currently running tasks are sampled, and on that CPU.

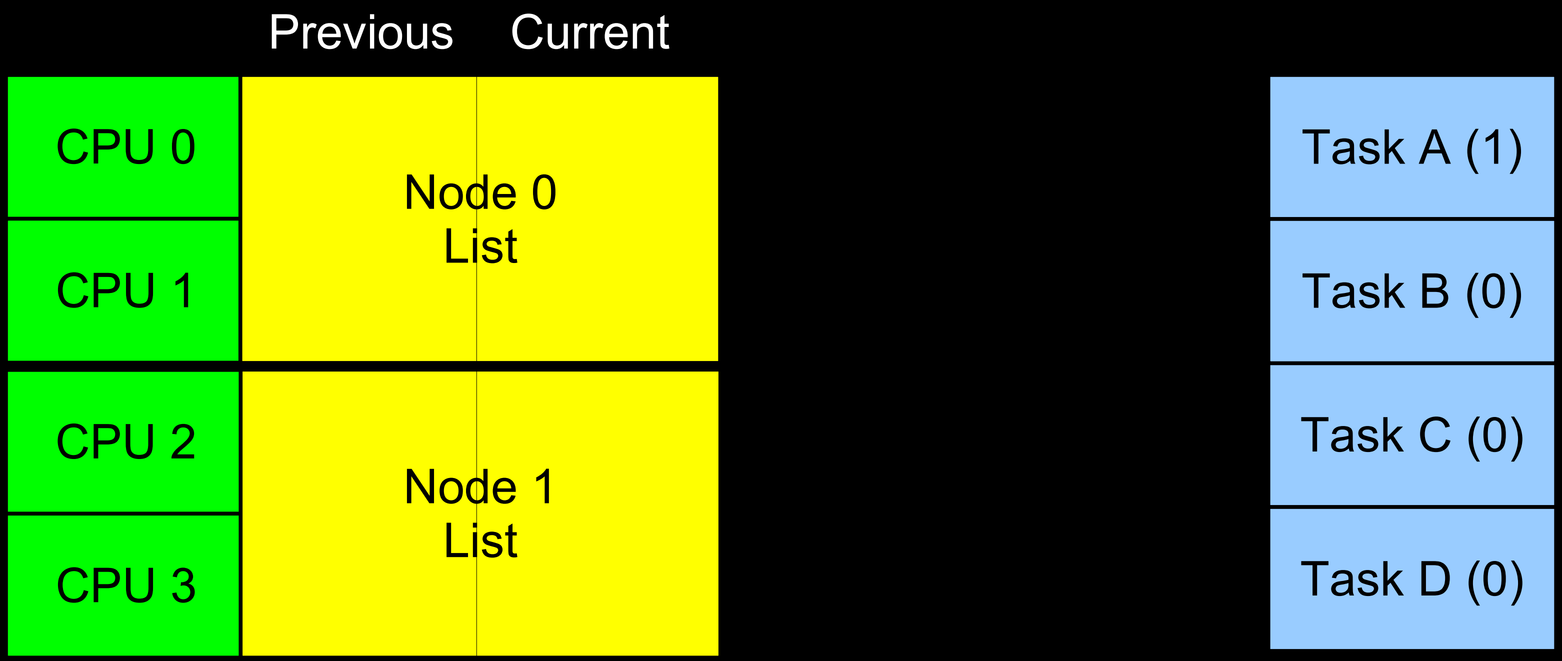
When a task blocks within an RCU read-side critical section, it is enqueued.

Real-Time RCU The Easy Way


Previous
Current
CPU 0
CPU 1
CPU 2
CPU 3
Node 0 List
Node 1 List
Task A (0)
Task B (0)
Task C (0)
Task D (0)

Initial state.

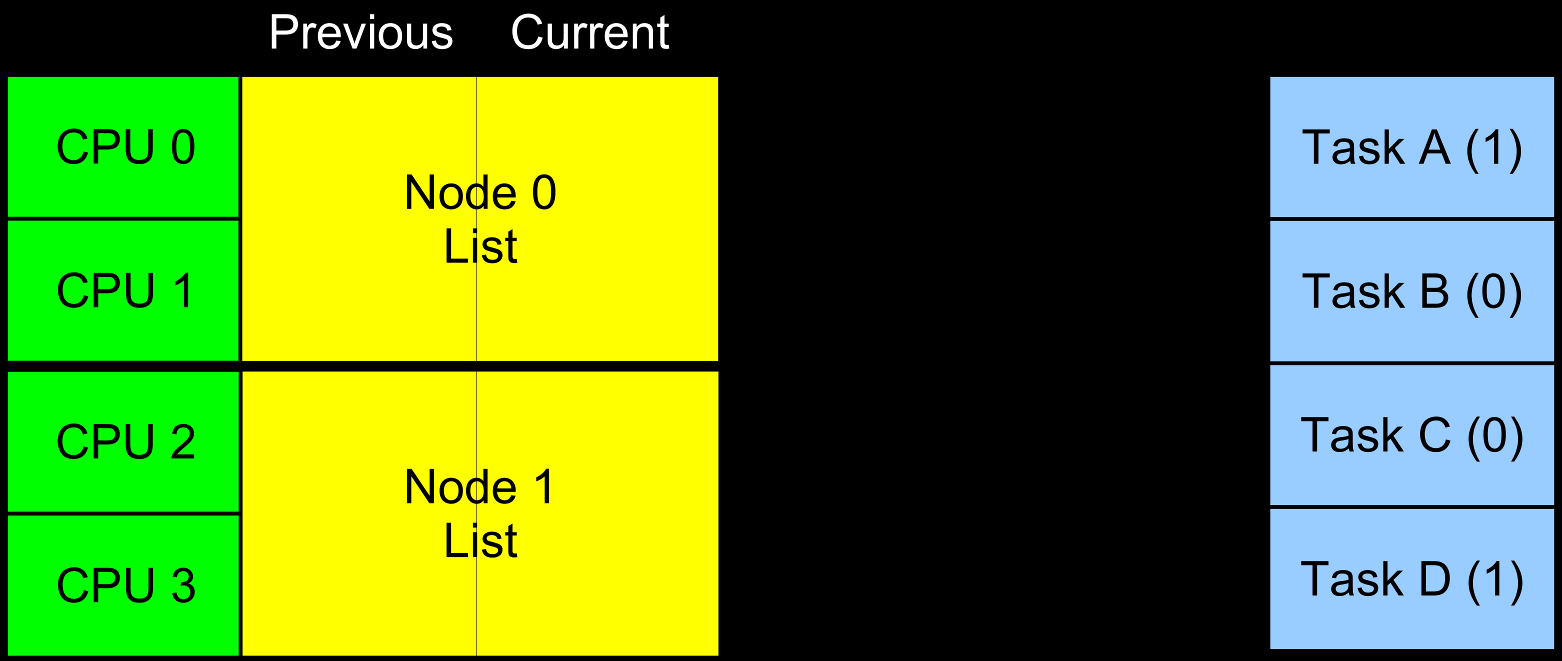
Real-Time RCU The Easy Way



Task A rcu_read_lock() on CPU 1.

So now waiting on CPU 1 to get done.

Real-Time RCU The Easy Way



Task D rcu_read_lock() on CPU 3.

So now waiting on CPU 1 and CPU 3 to get done.

Real-Time RCU The Easy Way

Current Previous

CPU 0

CPU 1

Node 0
List

CPU 2

CPU 3

Node 1
List

Task A (1)

Task B (0)

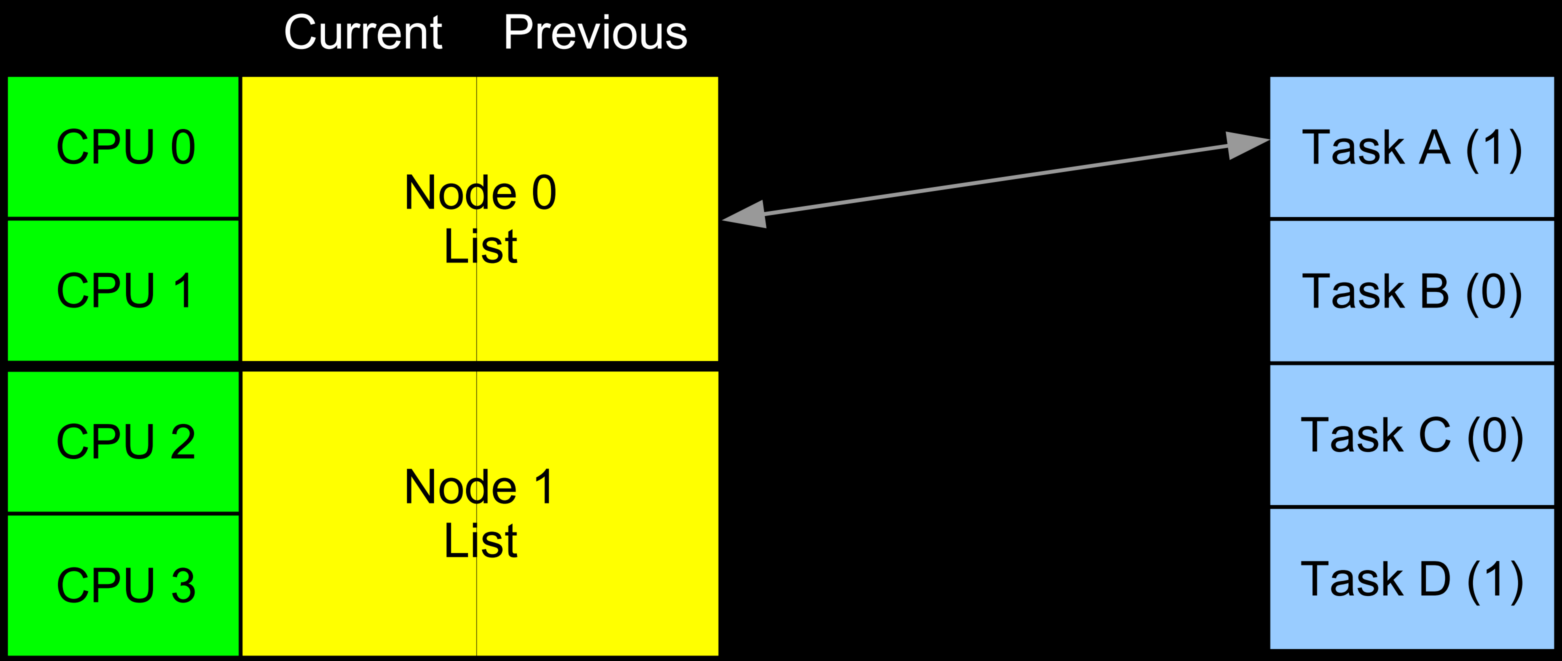
Task C (0)

Task D (1)

Task C synchronize_rcu() entry: Node lists “flip”, or reverse roles.

Need only wait on CPUs 1 and 3 (detected via scheduling clock interrupt).

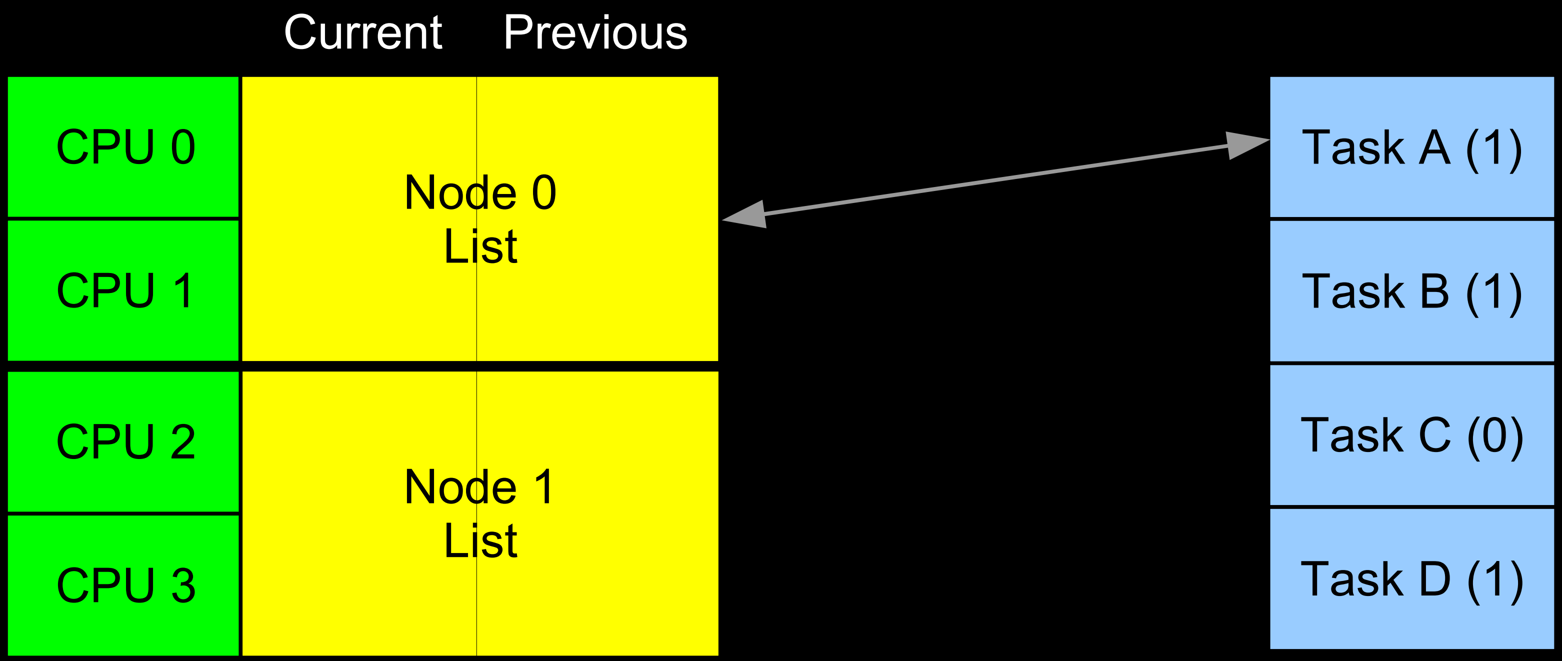
Real-Time RCU The Easy Way



Task A is preempted, and resumes on CPU 2.

So now we are waiting on CPU 3 and on Task A.

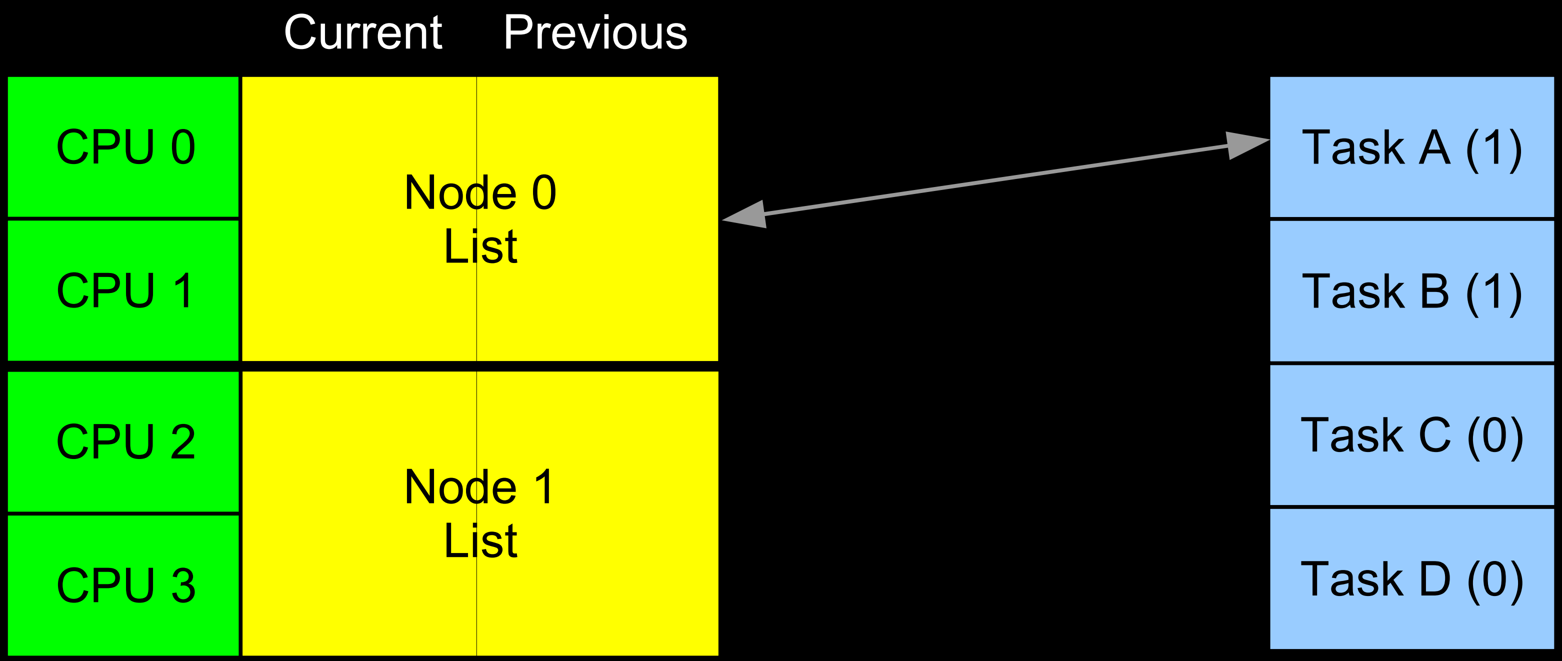
Real-Time RCU The Easy Way



Task B rcu_read_lock() on CPU 0.

But it started after the grace period began, so no need to wait on it. Yet.

Real-Time RCU The Easy Way



Current Previous

CPU 0
CPU 1
Node 0 List

CPU 2
CPU 3
Node 1 List

Task A (1)
Task B (1)
Task C (0)
Task D (0)

Task D rcu_read_unlock(), still on CPU 3.

Now waiting on Task A.

Real-Time RCU The Easy Way

Task A rcu_read_unlock(), still on CPU 2.

Everything we were waiting on is done, so Task C synchronize_rcu() returns.

Real-Time RCU The Easy Way

Task B rcu_read_unlock().

(For more info on hierarchical RCU: http://lwn.net/Articles/305782/ and TBD)

RCU 2009-2010: Simplicity Through Optimization

- **2.6.32 simplifies things in the common case:**

```
void __rcu_read_lock(void)
{
  ACCESS_ONCE(current->rcu_read_lock_nesting)++;
  barrier();
}

void __rcu_read_unlock(void)
{
  struct task_struct *t = current;

  barrier();
  if (--ACCESS_ONCE(t->rcu_read_lock_nesting) == 0 &&
      unlikely(ACCESS_ONCE(t->rcu_read_unlock_special)))
    rcu_read_unlock_special(t);
}
```

And provides 2-3x speedup for both read side and grace periods in common case. Please note that both rcu_read_lock() and rcu_read_unlock() fit on one slide.

RCU 2009-2010: Simplicity Through Optimization

- **2.6.32 simplifies things in the common case:**

```
void __rcu_read_lock(void)
{
  ACCESS_ONCE(current->rcu_read_lock_nesting)++;
  barrier();
}

void __rcu_read_unlock(void)
{
  struct task_struct *t = current;

  barrier();
  if (--ACCESS_ONCE(t->rcu_read_lock_nesting) == 0 &&
      unlikely(ACCESS_ONCE(t->rcu_read_unlock_special)))
    rcu_read_unlock_special(t);
}
```

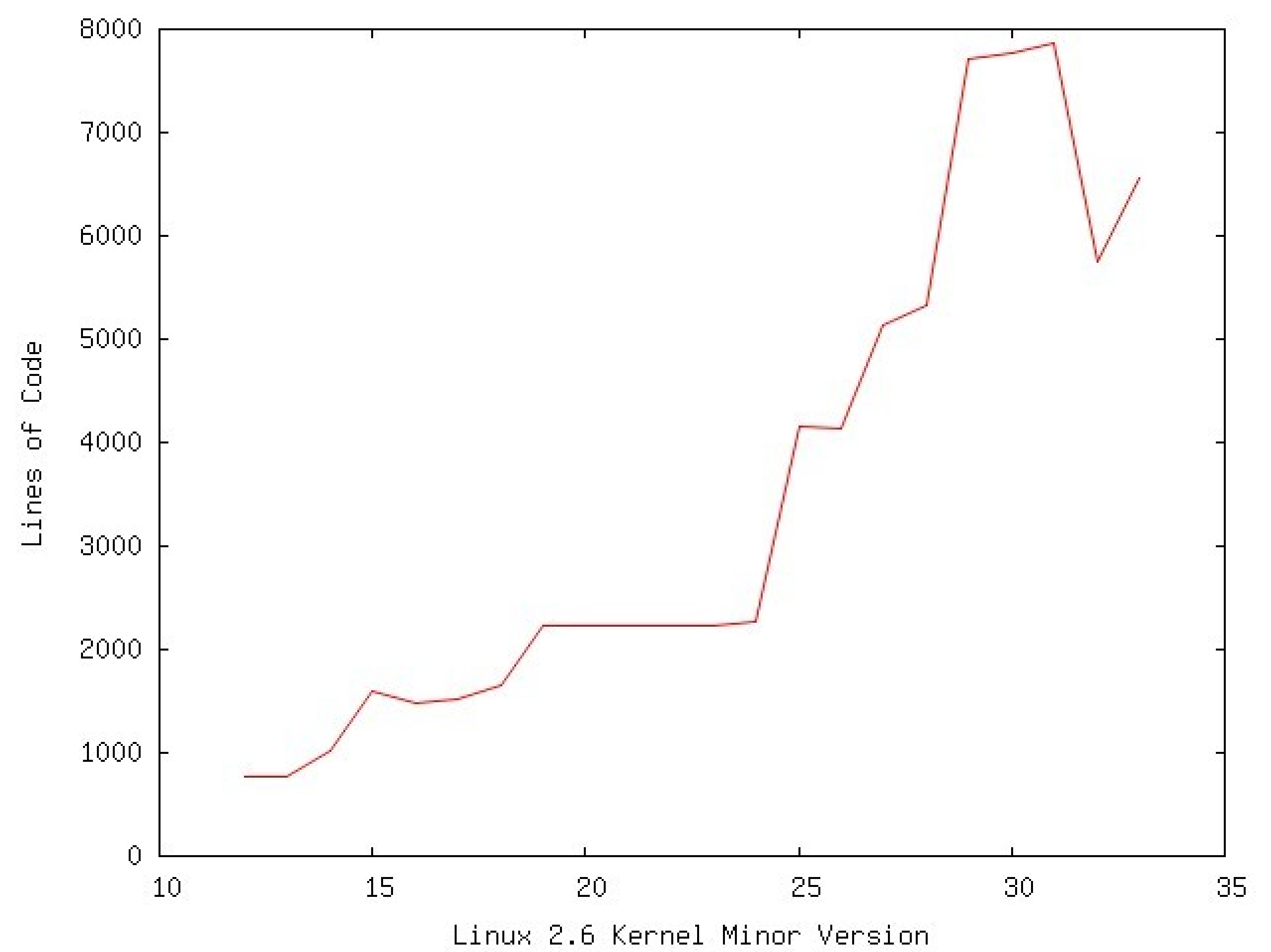
Overall Effect

- **Read-side performance improves by 2-3x over old preemptible RCU**
- **Grace-period latency improves by 2-3x over old preemptible RCU**
- **This implementation allowed the CLASSIC_RCU and PREEMPT_RCU implementations to be dropped, reducing kernel source by more than 2,000 LOC**

Overall Effect

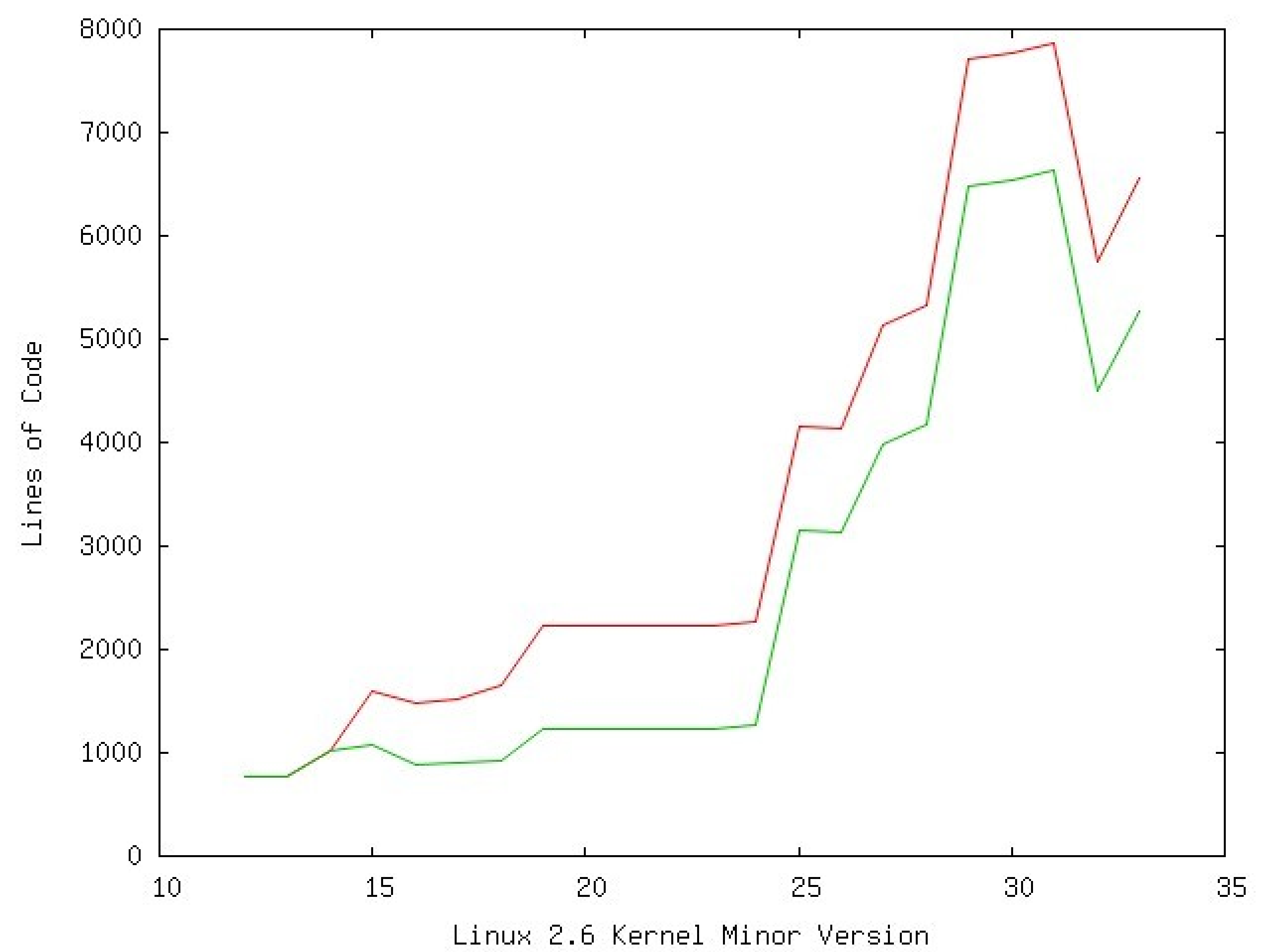
- **Read-side performance improves by 2-3x over old preemptible RCU**
- **Grace-period latency improves by 2-3x over old preemptible RCU**
- **This implementation allowed the CLASSIC_RCU and PREEMPT_RCU implementations to be dropped, reducing kernel source by more than 2,000 LOC**
  - But what is historical LOC trend?

RCU Code-Size Trend: 2002-2010



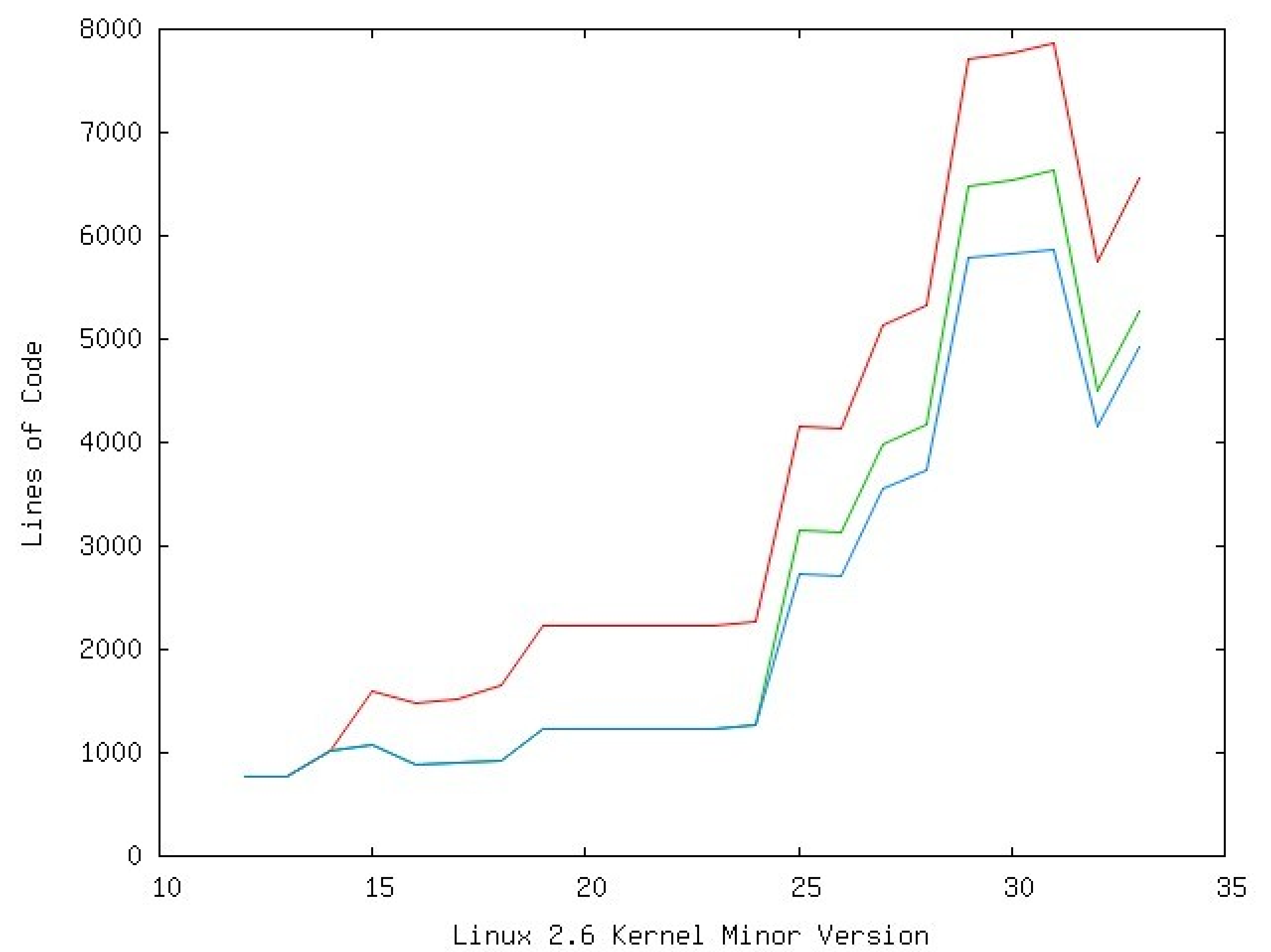
But we should not count rcutorture tests...

RCU Code-Size Trend: 2002-2010



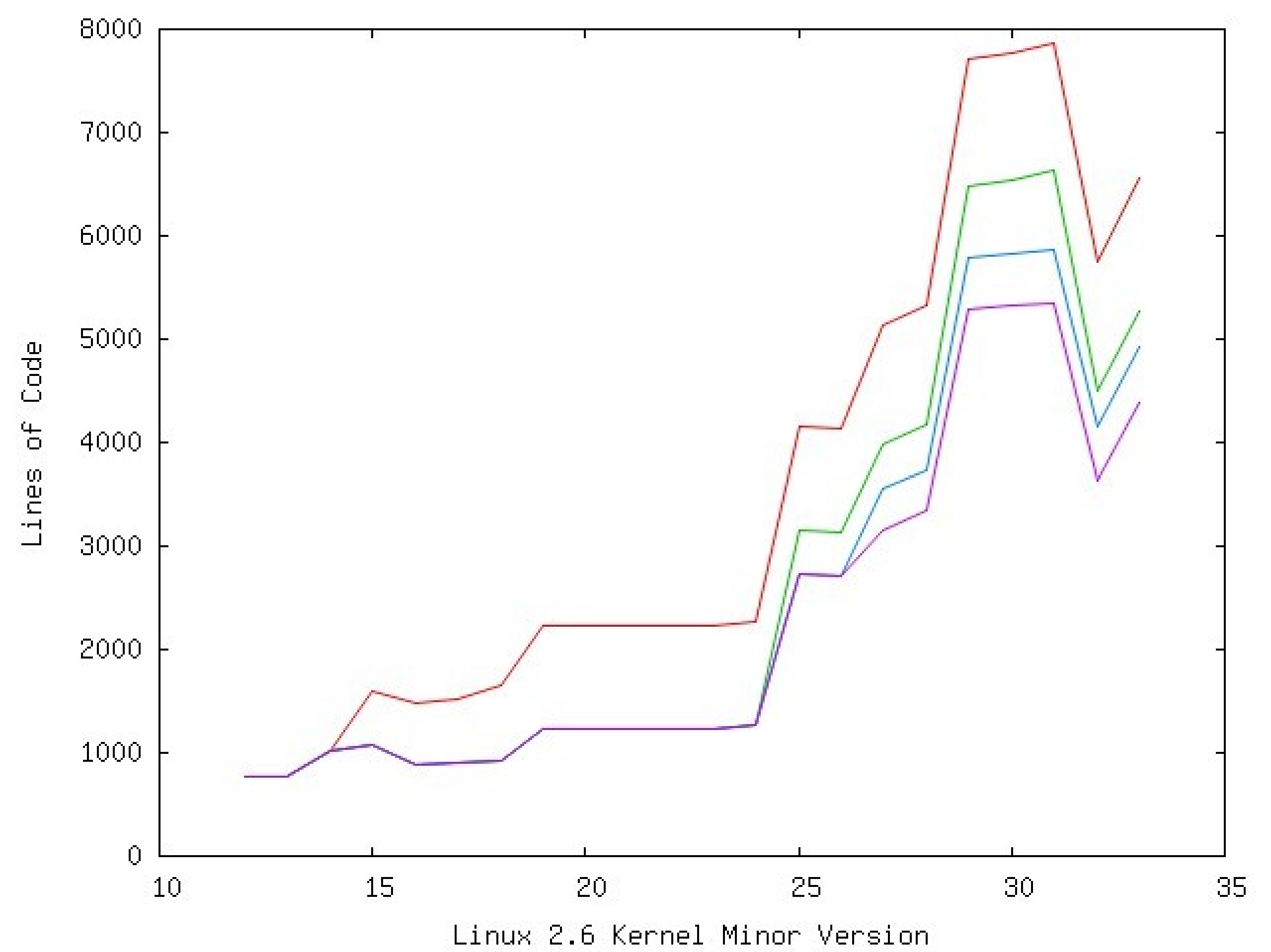
And we should not count RCU tracing...

RCU Code-Size Trend: 2002-2010



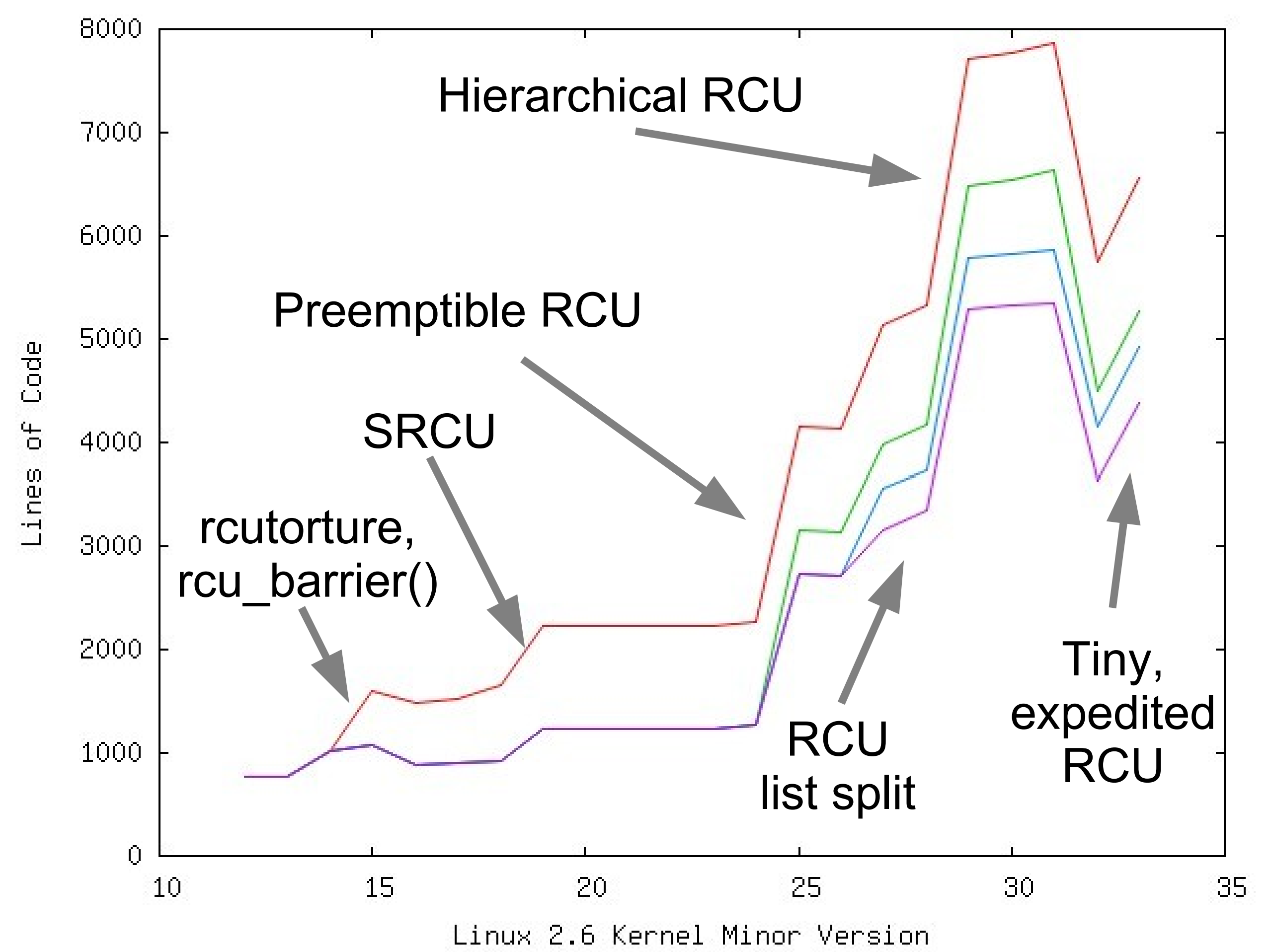
And we should not count higher-level list primitives...

RCU Code-Size Trend: 2002-2010



But there is still a large increase in base RCU code size!!!

RCU Code-Size Trend: 2002-2010



But RCU's Code Size Increased By Factor of Six!!!

- **But we added:**
  - rcu_barrier() to allow call_rcu() in kernel modules
  - SRCU to allow sleeping RCU readers
  - CPU hotplug interactions with RCU
  - Preemptible RCU for real-time use
  - “sparse” annotations for RCU read-side primitives
  - lockdep tracking of RCU read-side primitives
  - Dyntick interface so RCU lets sleeping CPUs snooze
  - Hierarchical RCU for systems with 1,000 CPUs (maybe more)
  - Expedited RCU grace periods
  - Hierarchical preemptible RCU for large real-time systems
  - Tiny RCU for UP systems with memory constraints

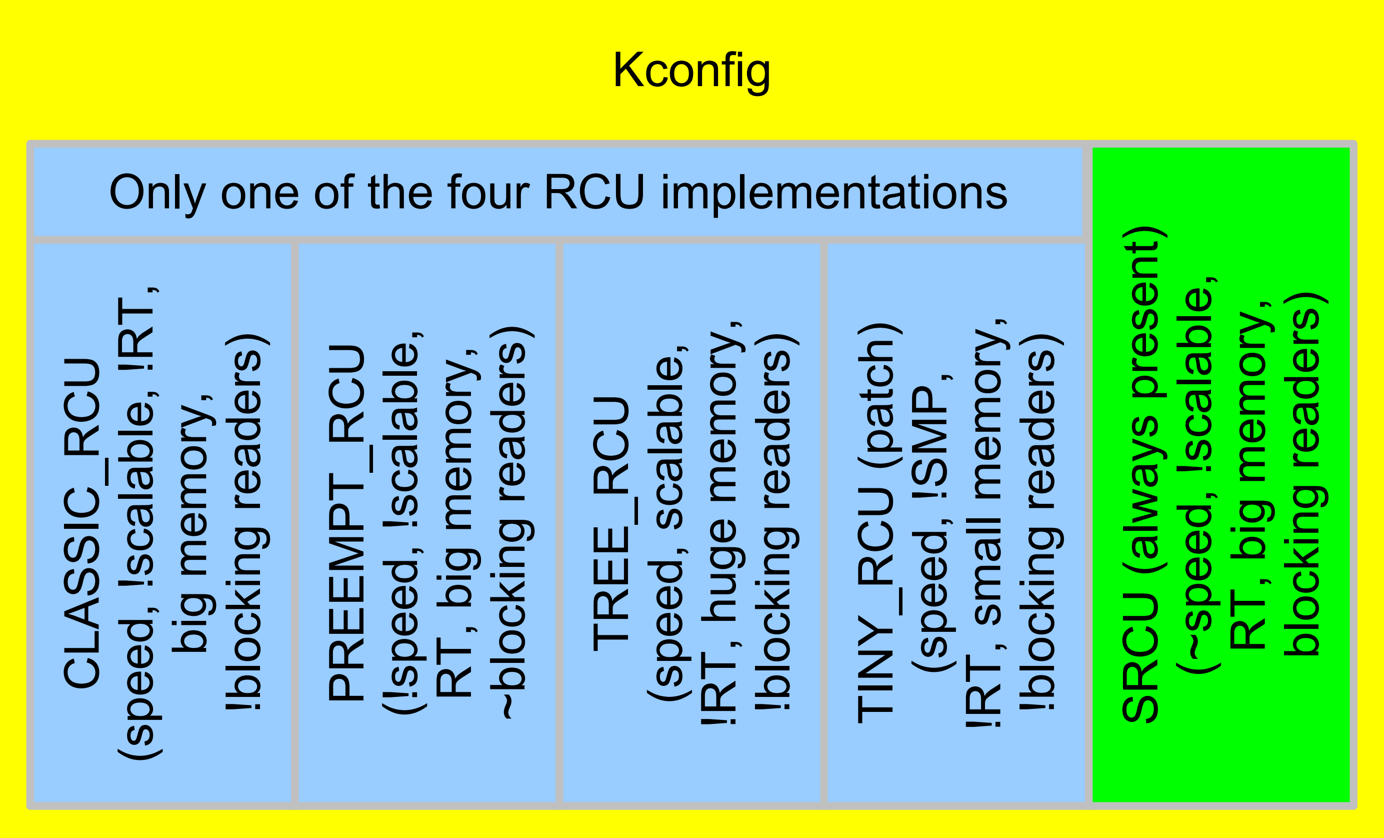
But RCU's Code Size Increased By Factor of Six!!!

- **But we added:**
  - rcu_barrier() to allow call_rcu() in kernel modules
  - SRCU to allow sleeping RCU readers
  - CPU hotplug interactions with RCU
  - Preemptible RCU for real-time use
  - “sparse” annotations for RCU read-side primitives
  - lockdep tracking of RCU read-side primitives
  - Dyntick interface so RCU lets sleeping CPUs snooze
  - Hierarchical RCU for systems with 1,000 CPUs (maybe more)
  - Expedited RCU grace periods
  - Hierarchical preemptible RCU for large real-time systems
  - Tiny RCU for UP systems with memory constraints
- **So this is a case of “legitimate bloat”**

But RCU's Code Size Increased By Factor of Six!!!

- **But we added:**
  - rcu_barrier() to allow call_rcu() in kernel modules
  - SRCU to allow sleeping RCU readers
  - CPU hotplug interactions with RCU
  - Preemptible RCU for real-time use
  - “sparse” annotations for RCU read-side primitives
  - lockdep tracking of RCU read-side primitives
  - Dyntick interface so RCU lets sleeping CPUs snooze
  - Hierarchical RCU for systems with 1,000 CPUs (maybe more)
  - Expedited RCU grace periods
  - Hierarchical preemptible RCU for large real-time systems
  - Tiny RCU for UP systems with memory constraints
- **So this is a case of “legitimate bloat”**
  - *That is my story and I am sticking to it!!! :-)*

RCU Before Simplification



Kconfig

Only one of the four RCU implementations

CLASSIC_RCU
(speed, !scalable, !RT,
big memory,
!blocking readers)

PREEMPT_RCU
(!speed, !scalable,
RT, big memory,
~blocking readers)

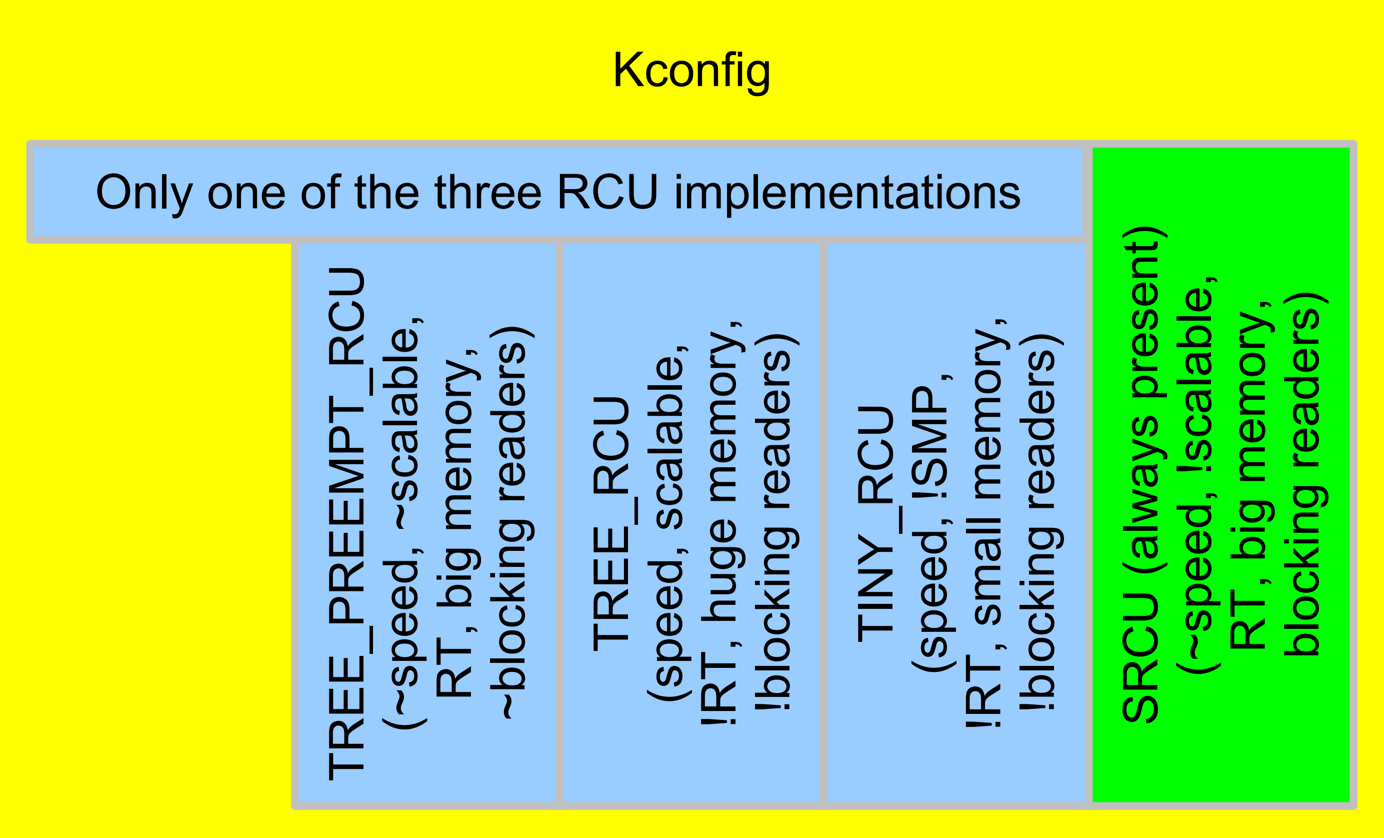
TREE_RCU
(speed, scalable,
!RT, huge memory,
!blocking readers)

TINY_RCU (patch)
(speed, !SMP,
!RT, small memory,
!blocking readers)

SRCU (always present)
(~speed, !scalable,
RT, big memory,
blocking readers)

(Not to scale)

RCU After Simplification



Kconfig

Only one of the three RCU implementations

TREE_PREEMPT_RCU
(~speed, ~scalable,
RT, big memory,
~blocking readers)

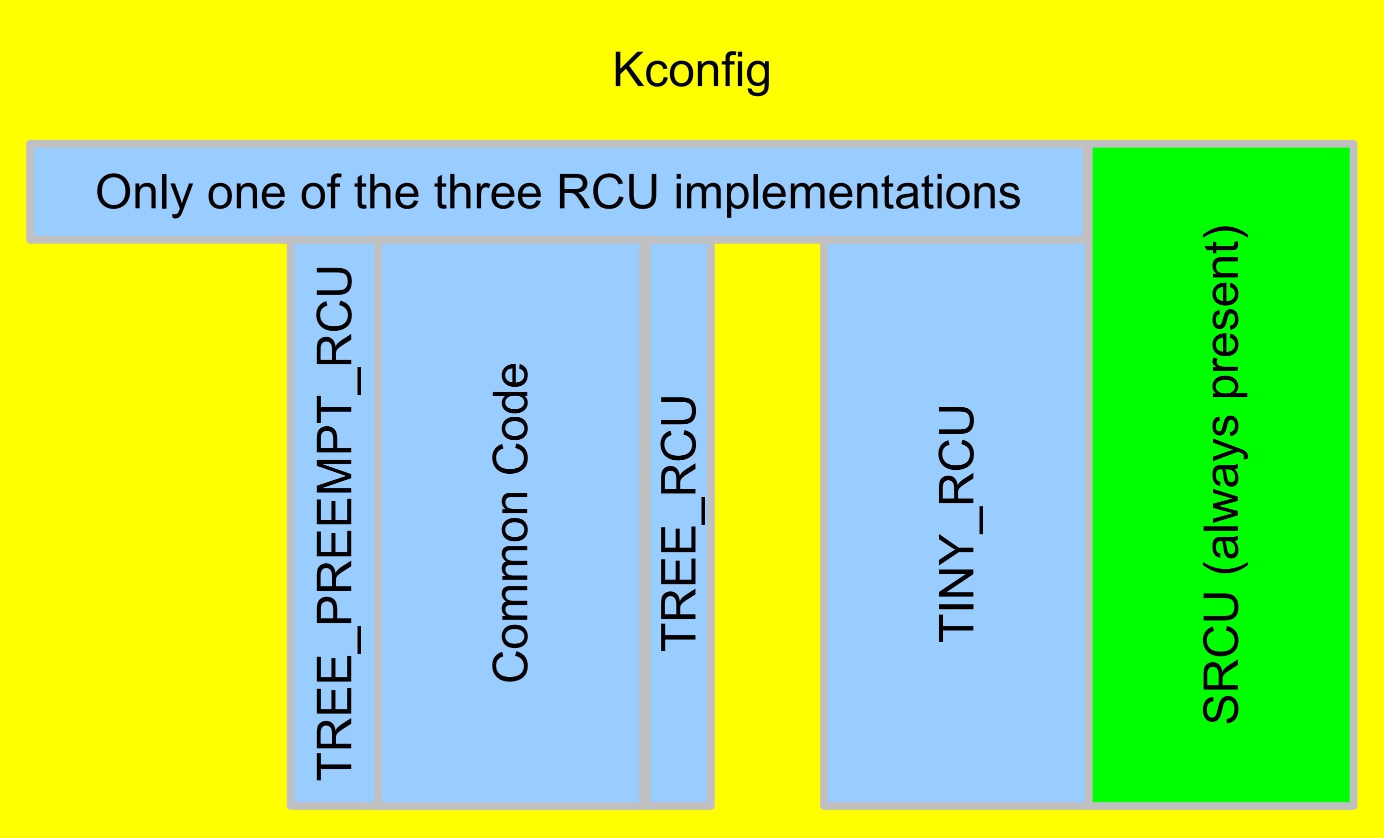
TREE_RCU
(speed, scalable,
!RT, huge memory,
!blocking readers)

TINY_RCU
(speed, !SMP,
!RT, small memory,
!blocking readers)

SRCU (always present)
(~speed, !scalable,
RT, big memory,
blocking readers)

(Not to scale)

RCU After Simplification: The Rest of the Story


Kconfig

Only one of the three RCU implementations

TREE_PREEMPT_RCU

Common Code

TREE_RCU

TINY_RCU

SRCU (always present)

(Not to scale)

Next Steps

Large Next Steps

- **Finish lockdep-enabled rcu_dereference()**
- **RCU priority boosting**
  - And corresponding rcutorture updates
- **kfree_rcu()**
- **TINY_PREEMPT_RCU**
- **Merge SRCU into TREE_RCU**
- **Make RCU independent of the scheduling-clock tick**
- **Make expedited primitives scale better**
- **Full inspection and documentation of TREE_RCU**

Smaller Next Steps

- **Get rid of list_for_each_continue_rcu() in favor of list_for_each_entry_continue_rcu()**
- **Add a notifier to panic_notifier_list to prevent stall warnings after panics**
- **Stop overflowing signed integers**
- **Clean up #ifdefs in kernel/rcutree.c**
- **Make RCU CPU stall detection unconditional**
- **Reduce TREE_RCU's need to send IPIs**
- **More abstractions under which to bury memory barriers**
- **Make TINY_RCU tinier (in object code, probably more source...)**
- **Make TREE_PREEMPT_RCU read-side primitives faster**
- **Remove !SMP special-case code from TREE_RCU**
- **Make call_rcu() be deterministic for real-time threads for TREE_RCU**
- **Deal with CPUs who are in dyntick-idle mode for a full wrap of ->gpnum**
- **Statistics of number of RCU read-side critical sections vs. number of requests for a grace period**

Effect of Next Steps

- Yes, each of these next steps will probably make the Linux kernel's RCU larger and more complex

Effect of Next Steps

- **Yes, each of these next steps will probably make the Linux kernel's RCU larger and more complex**
- **In other words, back to the usual experience**

Effect of Next Steps

- **Yes, each of these next steps will probably make the Linux kernel's RCU larger and more complex**
- **In other words, back to the usual experience**
- **But simplicity through optimization was fun while it lasted!!!**

Lessons Learned

Lessons Learned (or Relearned)

- **If you are doing something for the first time ever, your first implementation will probably not be optimal**
- **Good ways to come up with better implementations:**
  - Explain your code
    - If your code confuses someone, perhaps you should change it
    - When doing something new, confusion can be the most productive possible frame of mind
  - Document your code
  - Review other people's code
  - Help people to use your code
  - Code similar functionality in a different environment

Lessons Learned (or Relearned)

- **If you are doing something for the first time ever, your first implementation will probably not be optimal**
- **Good ways to come up with better implementations:**
  - Explain your code
    - If your code confuses someone, perhaps you should change it
    - When doing something new, confusion can be the most productive possible frame of mind
  - Document your code
  - Review other people's code
  - Help people to use your code
  - Code similar functionality in a different environment
- **Took three times to get real-time RCU right**

Lessons Learned (or Relearned)

- **Parallelism need not be counter-intuitive**

Lessons Learned (or Relearned)

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 Mathieu Desnoyers, Maged Michael, Joshua Triplett, and Jonathan Walpole

Questions