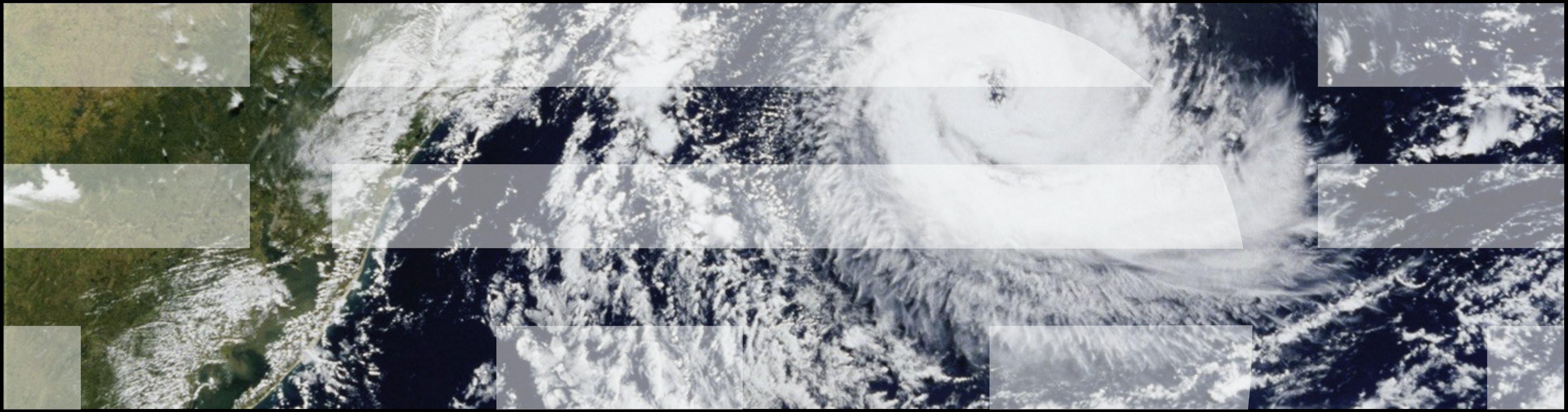
Paul E. McKenney, IBM Distinguished Engineer, Linux Technology Center & Linaro

February 15, 2012

Making RCU Safe For Battery-Powered Devices



Overview

- What is RCU?
- “The Good Old Days”
- Overview of RCU's many variants of energy efficiency
- Current state of RCU energy efficiency
- Future directions

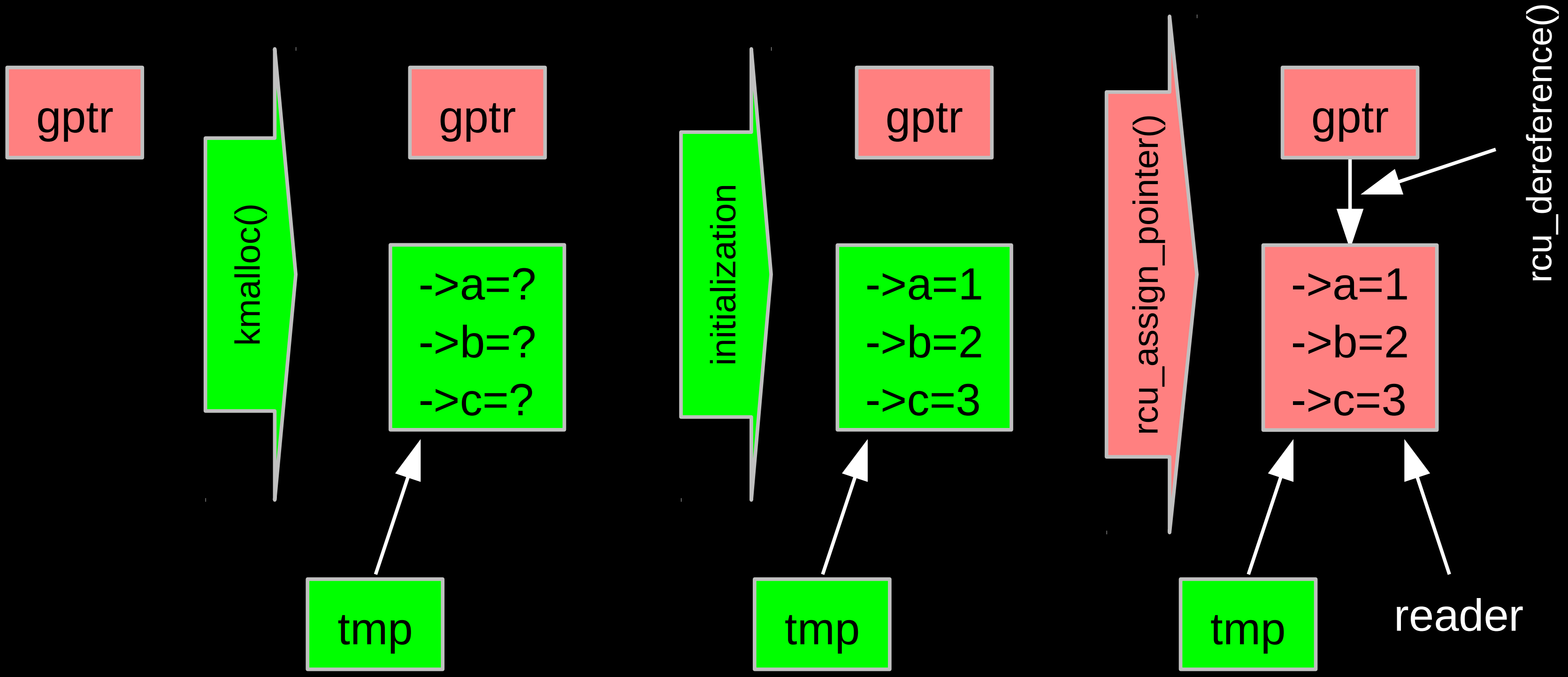
What is RCU?

A Very Brief Introduction to RCU

- Synchronization technique sometimes used in place of reader-writer locking
  - Extremely low read-side overhead: can be zero in actual use
    - Extreme performance, scalability, and real-time response
    - “Free is a very good price!”
  - RCU readers progress even in presence of writers and vice versa
- Most useful for read-mostly data: increasingly important
  - Routing tables, security policies, storage configuration, …
  - All of which could change at any time, but rarely do change in practice
- RCU operation:
  - Publication of and subscription to new data
  - RCU removal from linked list
  - Waiting for pre-existing readers (for zero-cost readers)

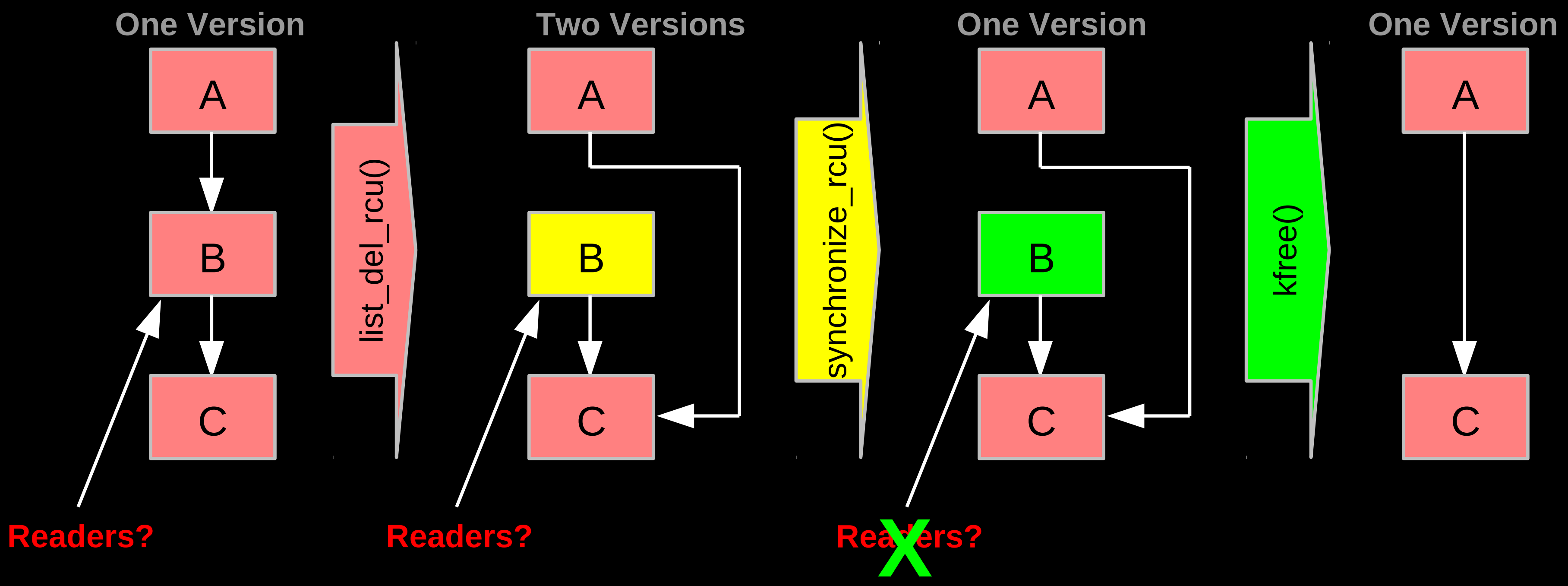
Publication of And Subscription To New Data

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



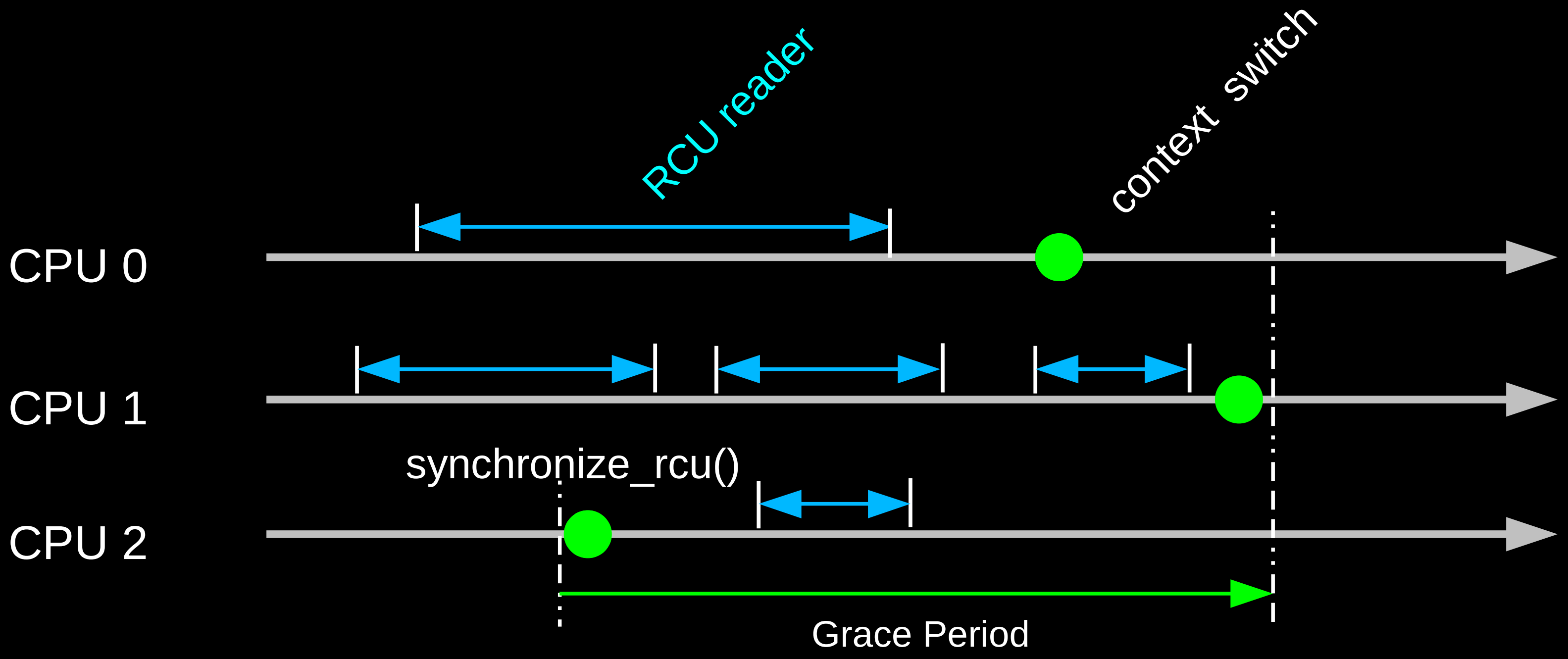
RCU Removal From Linked List

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

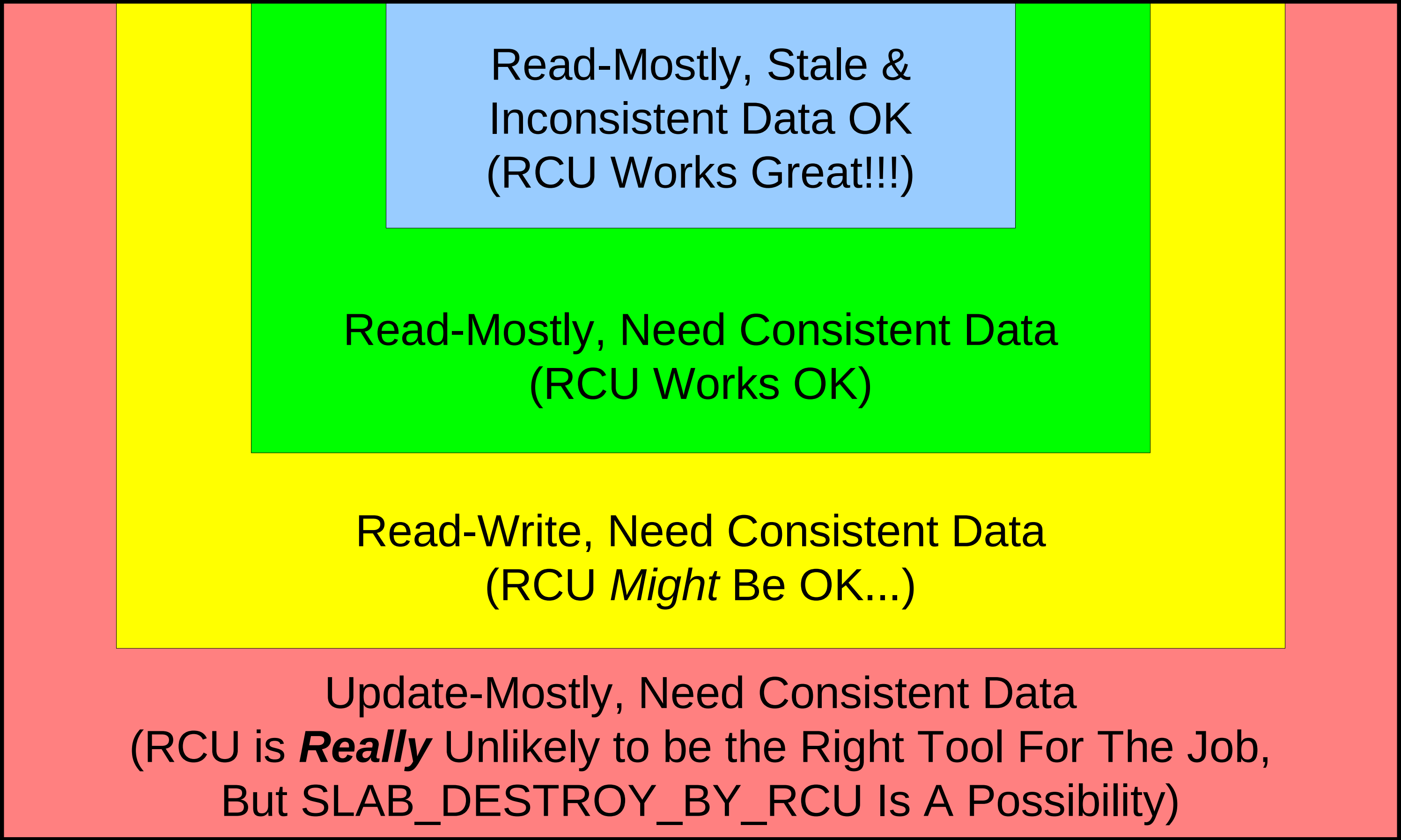


Waiting for Pre-Existing Readers

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



RCU Area of Applicability



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

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

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

Update-Mostly, Need Consistent Data
(RCU is ***Really*** Unlikely to be the Right Tool For The Job,
But SLAB_DESTROY_BY_RCU Is A Possibility)

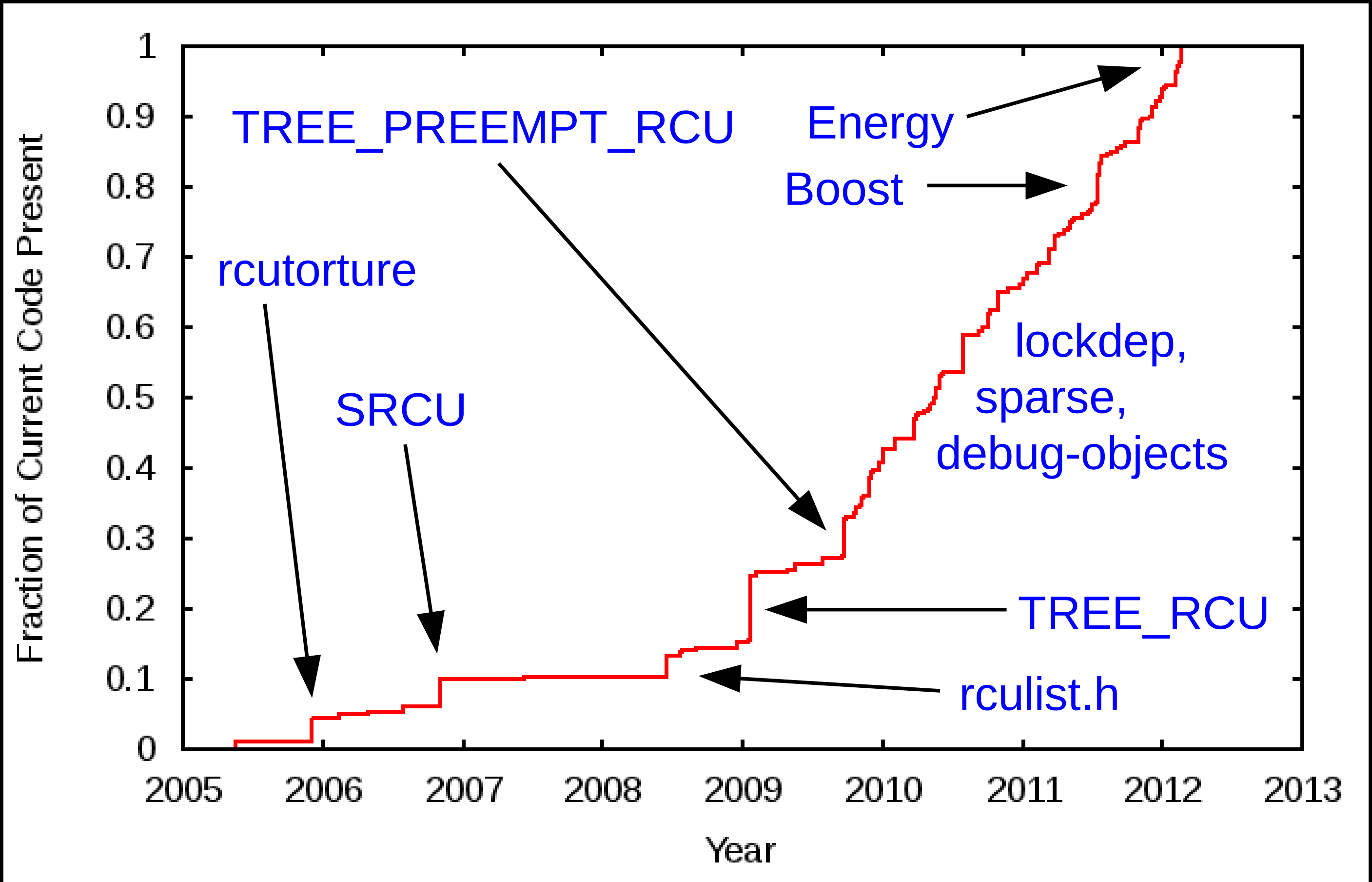
Use the right tool for the job!!!

For More Information on RCU...

- Documentation/RCU in the Linux® kernel source code
- “User-Level Implementations of Read-Copy Update” (Mathieu Desnoyers et al.)
  - http://doi.ieeecomputersociety.org/10.1109/TPDS.2011.159
- “The RCU API, 2010 Edition”
  - http://lwn.net/Articles/418853/
- “What is RCU” LWN series
  - 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?)
- “Introducing technology into the Linux kernel: a case study”
  - http://doi.acm.org/10.1145/1400097.1400099
- “Meet the Lockers” (Neil Brown)
  - http://lwn.net/Articles/453685/
- “Read-Copy Update” (2001 OLS paper, still used in a number of college courses)
  - http://www.linuxsymposium.org/2001/abstracts/readcopy.php
- Plus more at: http://www.rdrop.com/users/paulmck/RCU

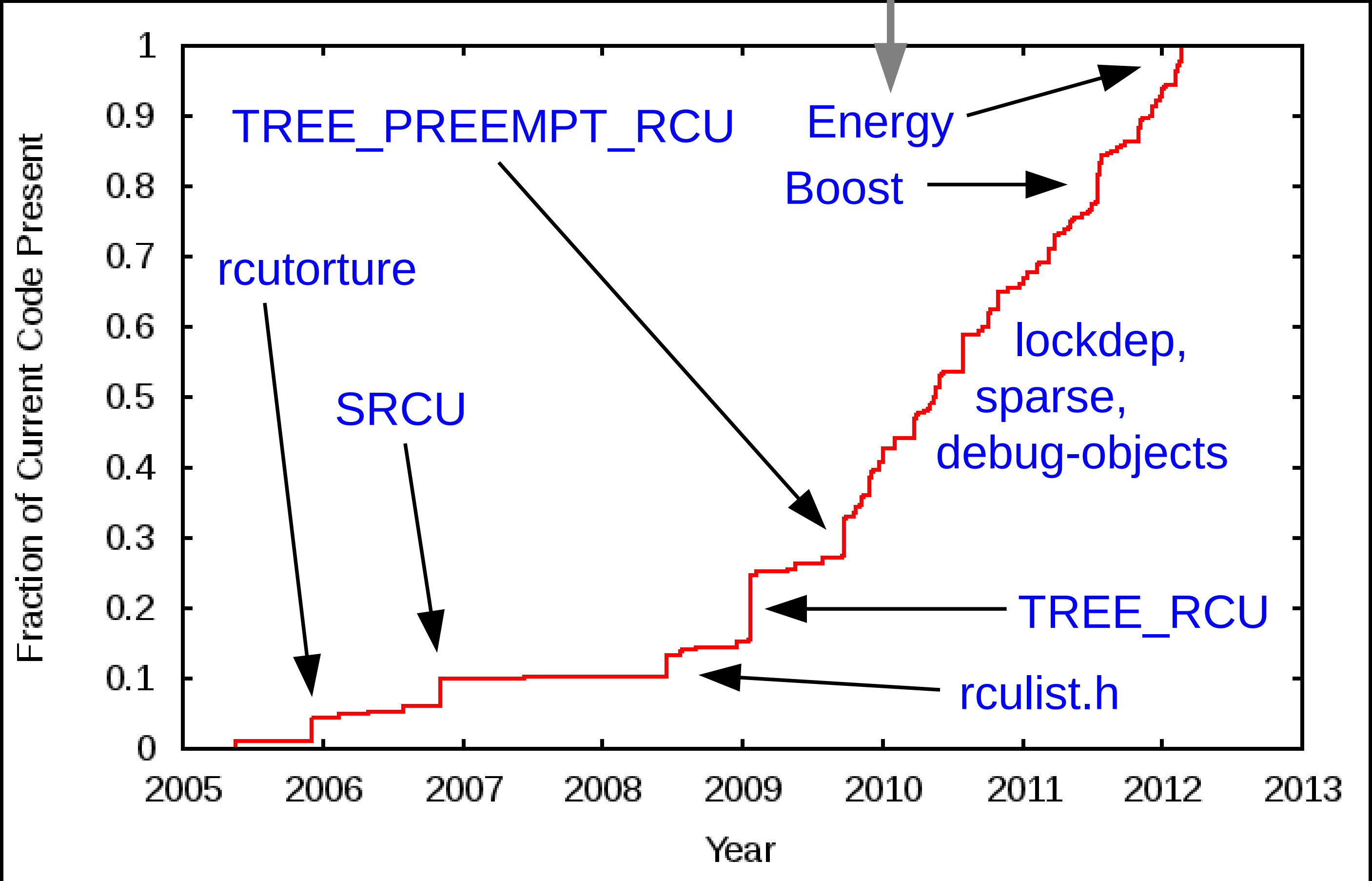
“The Good Old Days”

Not Much "Good Old Days" Code Left in RCU



Not Much "Good Old Days" Code Left in RCU

Why did I wait until 2011 to conserve energy???



Why Did I Wait Until 2011 to Conserve Energy?

- The fact is that I didn't wait until 2011!!!
- But RCU's energy-efficiency code is unusual in that it has been rewritten a great many times
  - –RCU has been concerned about energy efficiency for about ten years
  - –Not much energy-efficiency code in RCU in the 1990s: Why?
- Other minor changes:
  - –Expedited grace periods
  - –Additions to rcutorture
  - –Additional list-traversal primitives
  - –Reworking of CPU hotplug code
  - –Plus the usual list of fixes, improvements, and adaptations

“The Good *Really* Old Days”

- RCU used by DYNIX/ptx: Heavy database servers
- Used for a number of applications:
  - Fraud detection in large financial systems
  - Inventory monitoring/control for large retail firms
  - Rental car tracking/billing
  - Manufacturing coordination/control
    - Including manufacturing of airliners

Airliner Manufacturing Plants Have Lots of These:

Author: William M. Plate Jr. (Public Domain)

Airliner Manufacturing Plants Have Lots of These

At About 40KW Each

Author: William M. Plate Jr. (Public Domain)

And if You Think That *Welders* Are Power-Hungry...

GE90-115B turbofan - front {{Le Bourget 2005}} Copyright © 2005 David Monniaux {{GFDL}} {{cc-by-sa-2.0}} {{cc-by-sa-2.0-fr}}

If You Are Running a Bunch of Welders or Turbines...

- Not only are you not going to care much about RCU's contribution to power consumption...

If You Are Running a Bunch of Welders or Turbines...

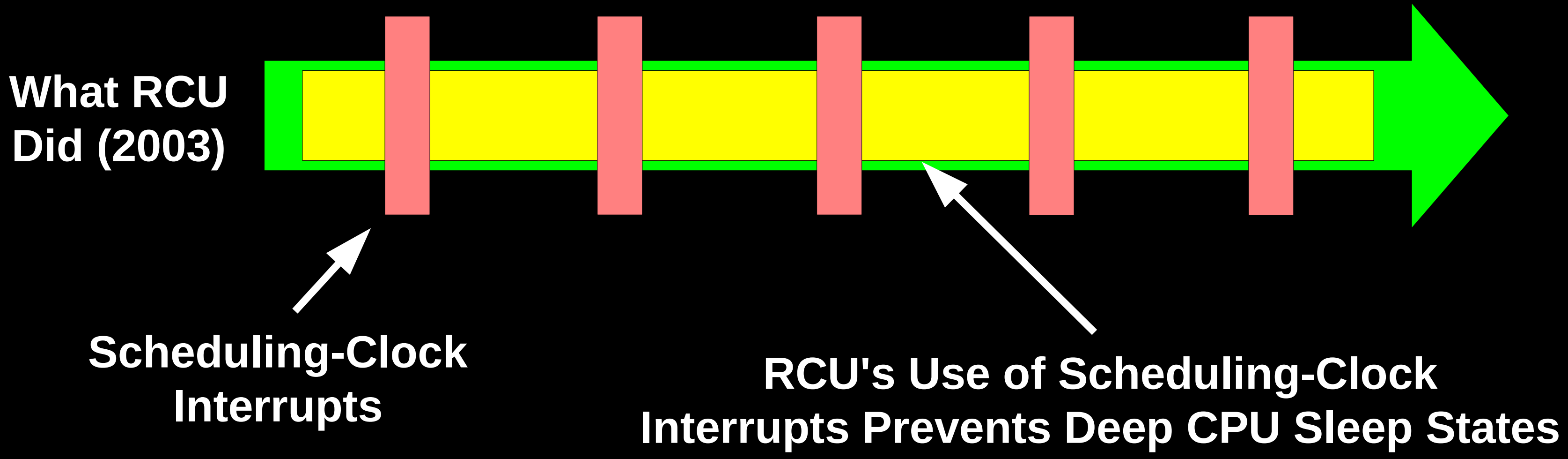
- Not only are you not going to care much about RCU's contribution to power consumption...
- You are not going to care much about the whole server's contribution to power consumption!
- But of course things look very different for small battery-powered devices...

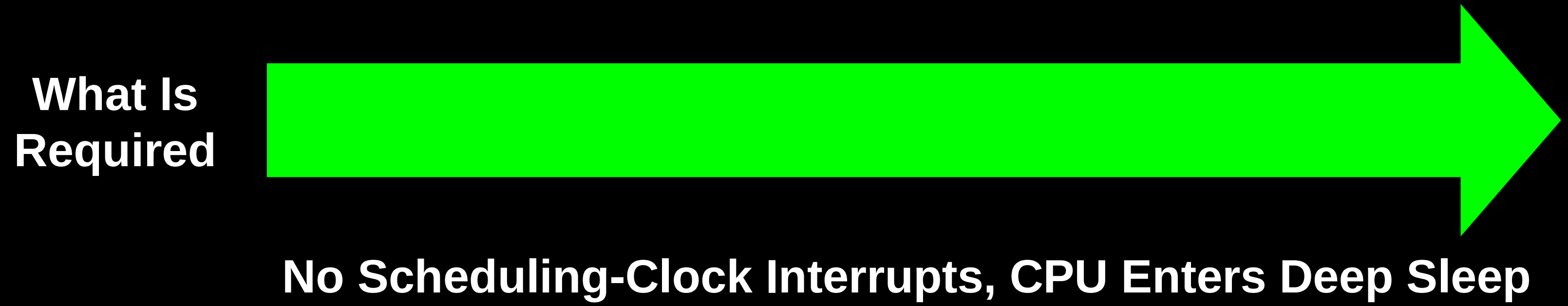
RCU's Many Energy-Efficiency Implementations

Initial RCU Did Have One Energy-Efficiency Feature

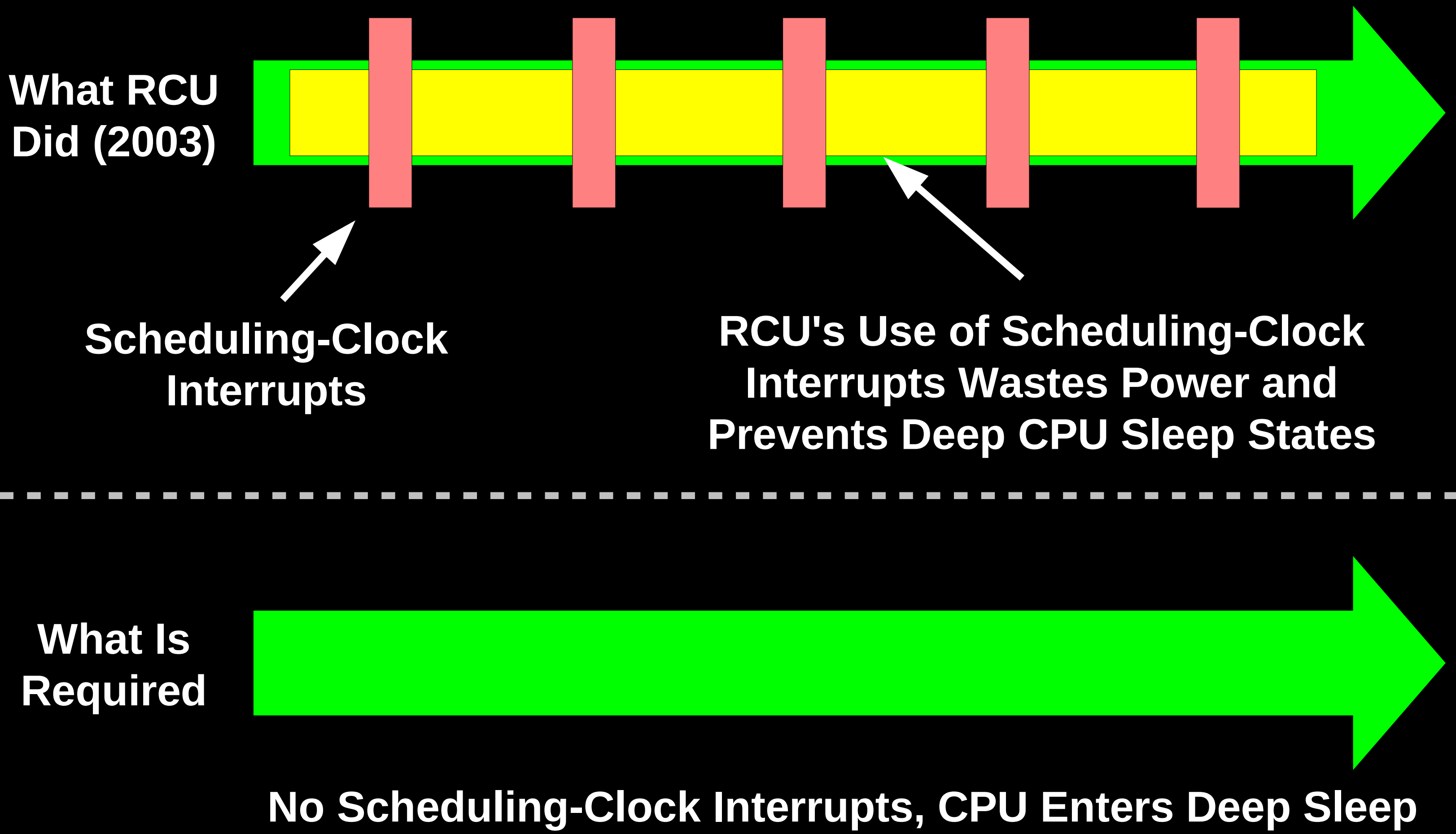
- Initial DYNIX/ptx RCU had light-weight read-side primitives
  - “Free” is a *very* good price!!!
- This meant that the system returned to idle more quickly than it would with heavier-weight synchronization primitives
  - But 1990s systems consumed more power idle than when running!
  - This was because the idle loop fit into cache, thus allowing the CPU to execute useless instructions at a very high rate
- But today's CPUs have many energy-efficiency features
  - And have very low idle power, especially for long-duration idle periods
- So why does RCU need to worry about energy efficiency???
  - After all, it is just a synchronization primitive!!!

RCU Driven From Scheduling Clock Interrupt


What RCU
Did (2003)
Scheduling-Clock
Interrupts
RCU's Use of Scheduling-Clock
Interrupts Prevents Deep CPU Sleep States



RCU Driven From Scheduling Clock Interrupt


What RCU Did (2003)
Scheduling-Clock Interrupts
RCU's Use of Scheduling-Clock Interrupts Wastes Power and Prevents Deep CPU Sleep States
What Is Required
No Scheduling-Clock Interrupts, CPU Enters Deep Sleep

Which is why RCU has a dyntick-idle subsystem!

RCU and Dyntick Idle (AKA CONFIG_NO_HZ=y)

- List of implementations:
  - 2004: Dyntick-idle bit vector
    - Manfred Spraul locates theoretical bug

RCU and Dyntick Idle (AKA CONFIG_NO_HZ=y)

- List of implementations:
  - 2004: Dyntick-idle bit vector
    - Manfred Spraul locates theoretical bug
    - A few months before the mainframe guys encounter it

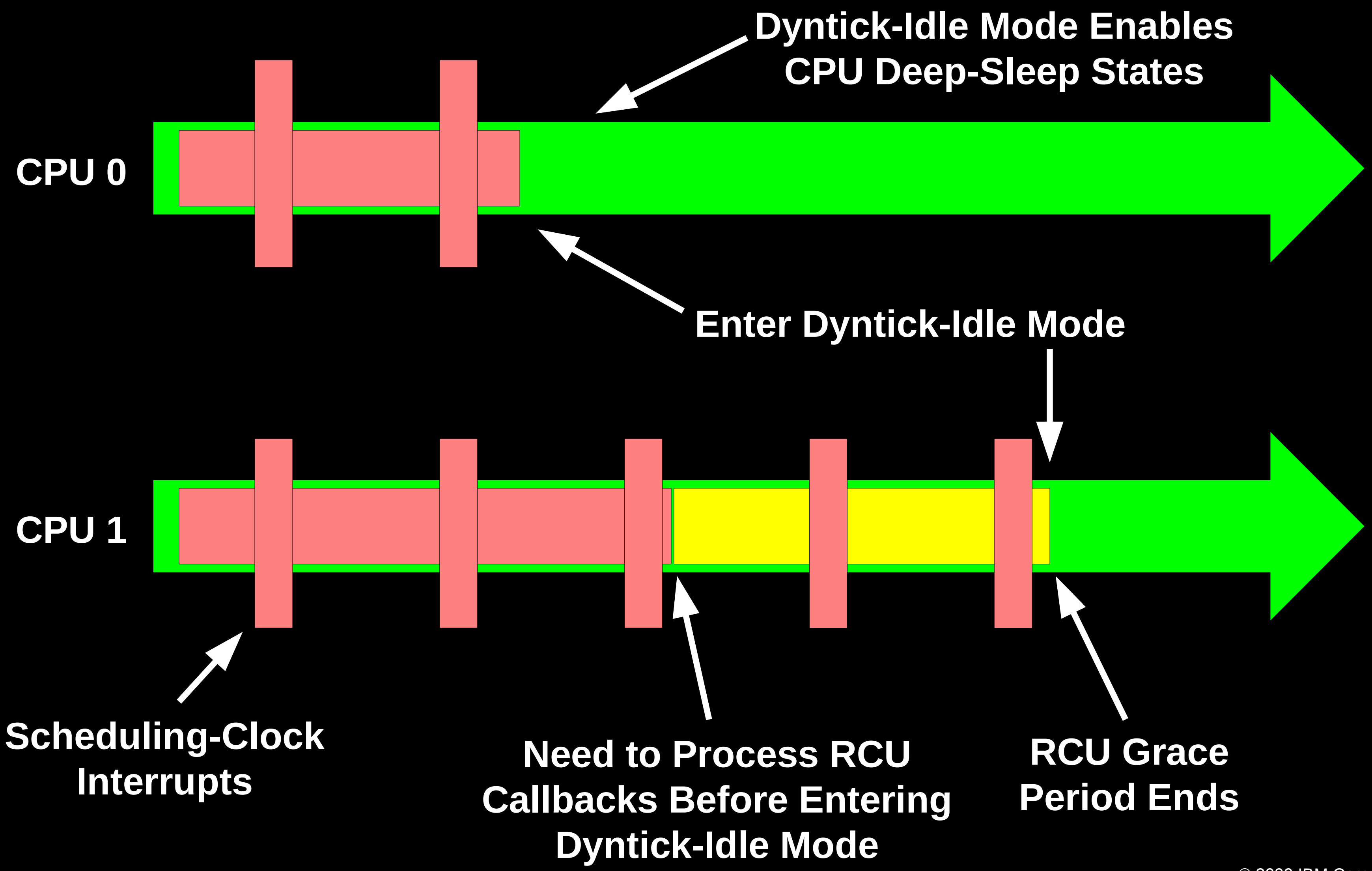
RCU and Dyntick Idle (AKA CONFIG_NO_HZ=y)

- List of implementations:
  - 2004: Dyntick-idle bit vector
    - Manfred Spraul locates theoretical bug
    - A few months before the mainframe guys encounter it
    - But after it had been in-tree for *four years*

RCU and Dyntick Idle (AKA CONFIG_NO_HZ=y)

- List of implementations:
  - 2004: Dyntick-idle bit vector
    - Manfred Spraul locates theoretical bug
    - A few months before the mainframe guys encounter it
    - But after it has been in-tree for *four years*
  - 2008: -rt version (with Steven Rostedt)
    - Very complex: http://lwn.net/Articles/279077/
  - 2009: Separate out NMI accounting
    - Greatly simplified: No proof of correctness required ;-)

RCU and Dyntick Idle as of Early 2010


Dyntick-Idle Mode Enables CPU Deep-Sleep States
CPU 0
Enter Dyntick-Idle Mode
CPU 1
Scheduling-Clock Interrupts
Need to Process RCU Callbacks Before Entering Dyntick-Idle Mode
RCU Grace Period Ends

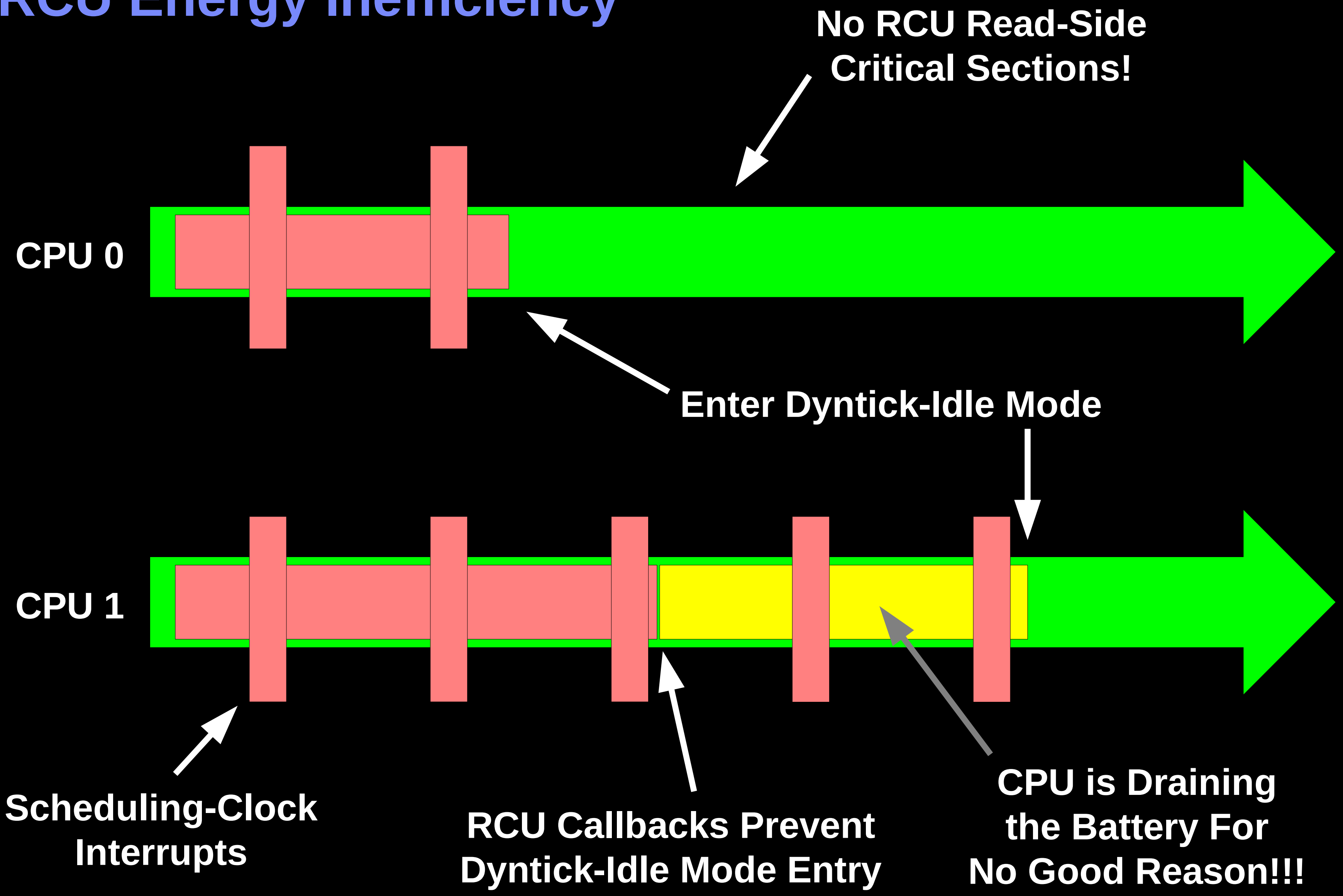
So RCU is Perfectly Energy Efficient, Right?

So RCU is Perfectly Energy Efficient, Right?

- Well, I thought that RCU was *very* energy efficient
- Then in early 2010 I got a call from someone working on a battery-powered multicore system
- And he was ***very*** upset with RCU

- Why?

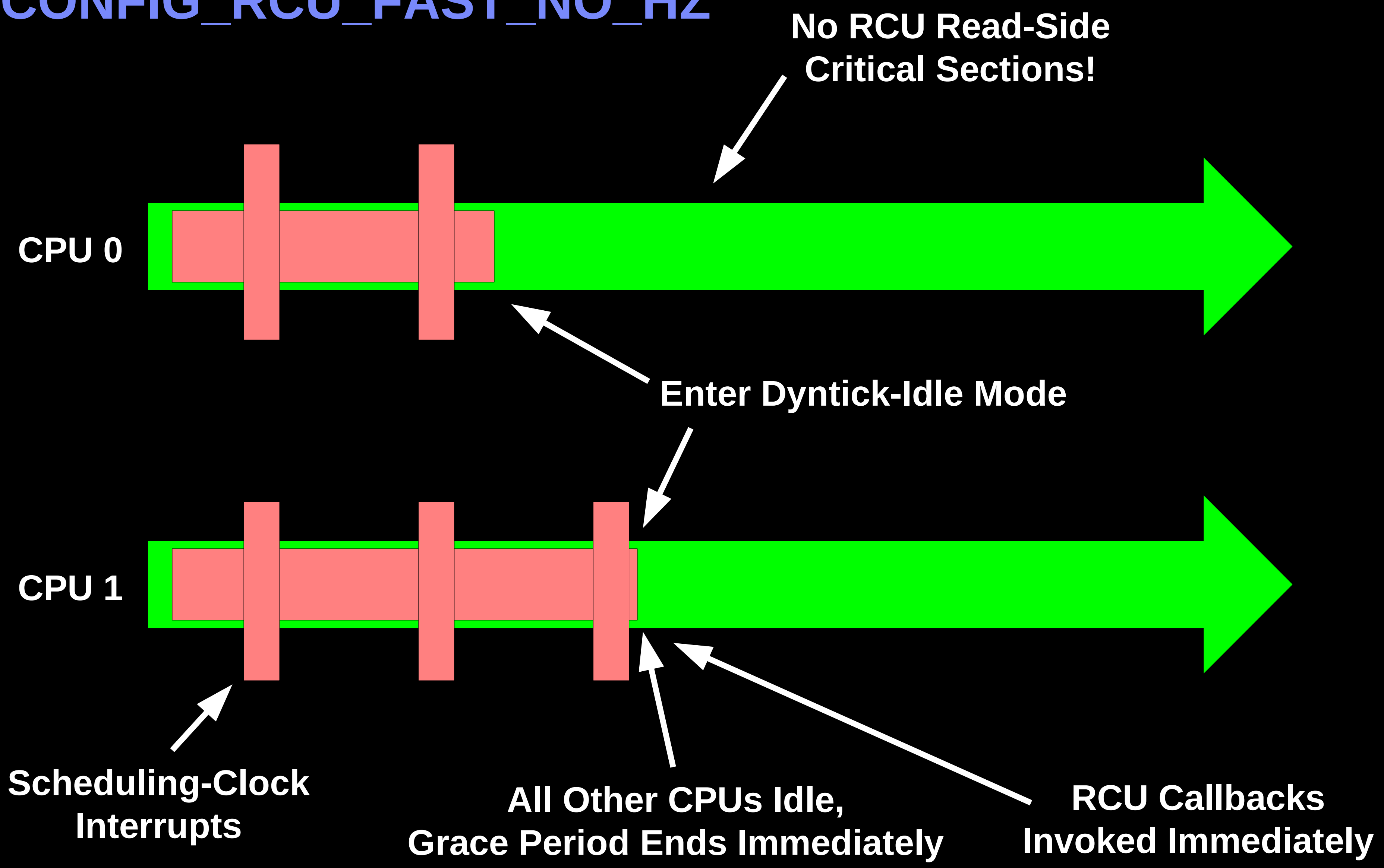
RCU Energy Inefficiency


No RCU Read-Side Critical Sections!
CPU 0
Enter Dyntick-Idle Mode
CPU 1
Scheduling-Clock Interrupts
RCU Callbacks Prevent Dyntick-Idle Mode Entry
CPU is Draining the Battery For No Good Reason!!!

RCU and Dyntick Idle (AKA CONFIG_NO_HZ=y)

- List of implementations:
  - 2004: Dyntick-idle bit vector
    - Manfred Spraul locates theoretical bug
    - A few months before the mainframe guys encounter it
    - But after it has been in-tree for *four* years
  - 2008: -rt version (with Steven Rostedt)
    - Very complex: http://lwn.net/Articles/279077/
  - 2009: Separate out NMI accounting
    - Greatly simplified: No proof of correctness required
  - 2010: CONFIG_RCU_FAST_NO_HZ for small systems
    - Force last CPU into dyntick-idle mode

CONFIG_RCU_FAST_NO_HZ



No RCU Read-Side Critical Sections!
CPU 0
Enter Dyntick-Idle Mode
CPU 1
Scheduling-Clock Interrupts
All Other CPUs Idle, Grace Period Ends Immediately
RCU Callbacks Invoked Immediately

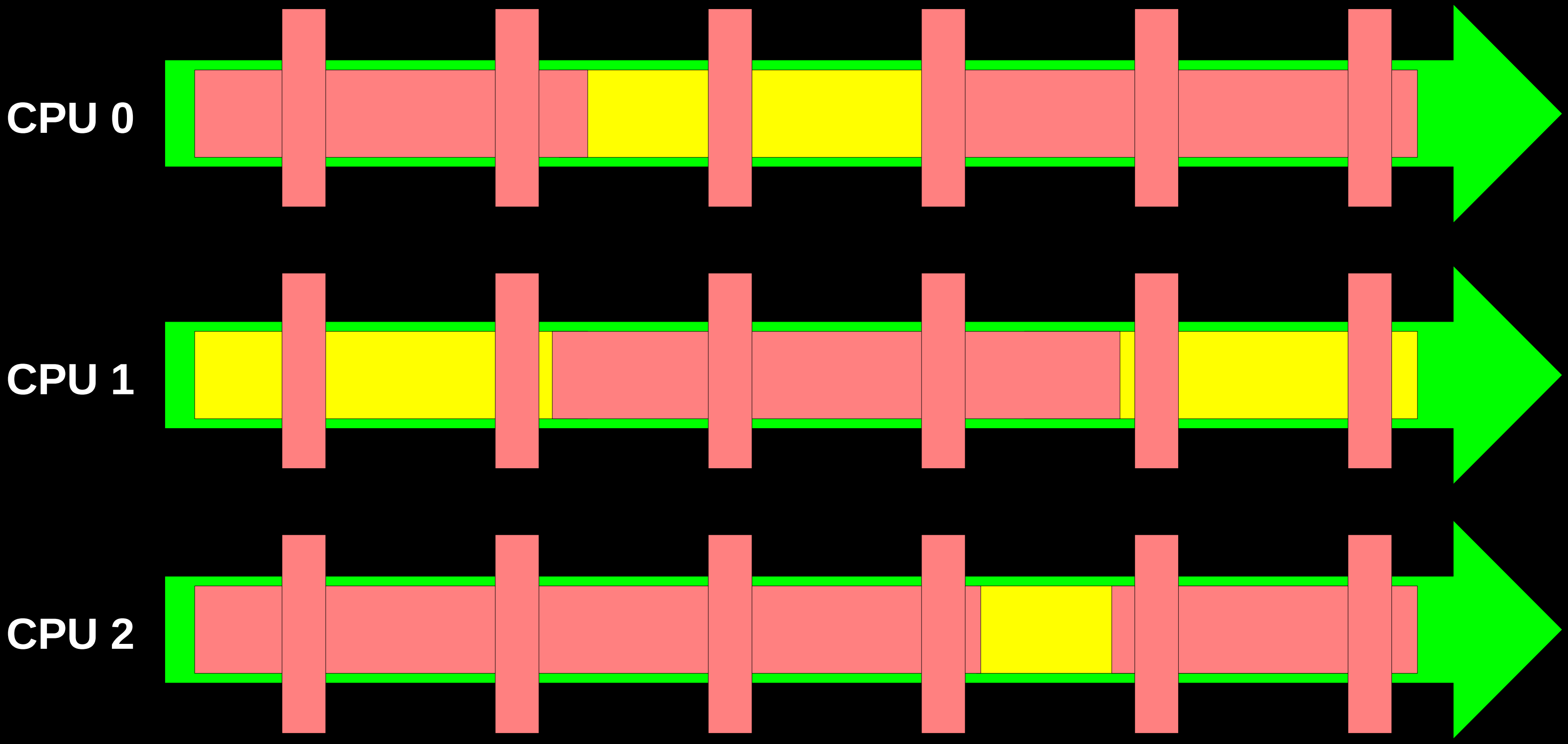
So RCU is Perfectly Energy Efficient, Right?

So RCU is Perfectly Energy Efficient, Right?

- This time, I was wiser:
  - I suspected CONFIG_FAST_NO_HZ needed on large systems
- And someone mentioned this to me in late 2011
- But some things never change: He was ***very*** upset with RCU

- Why?

Might *Never* Have All But One CPU Dyntick-Idled!!!

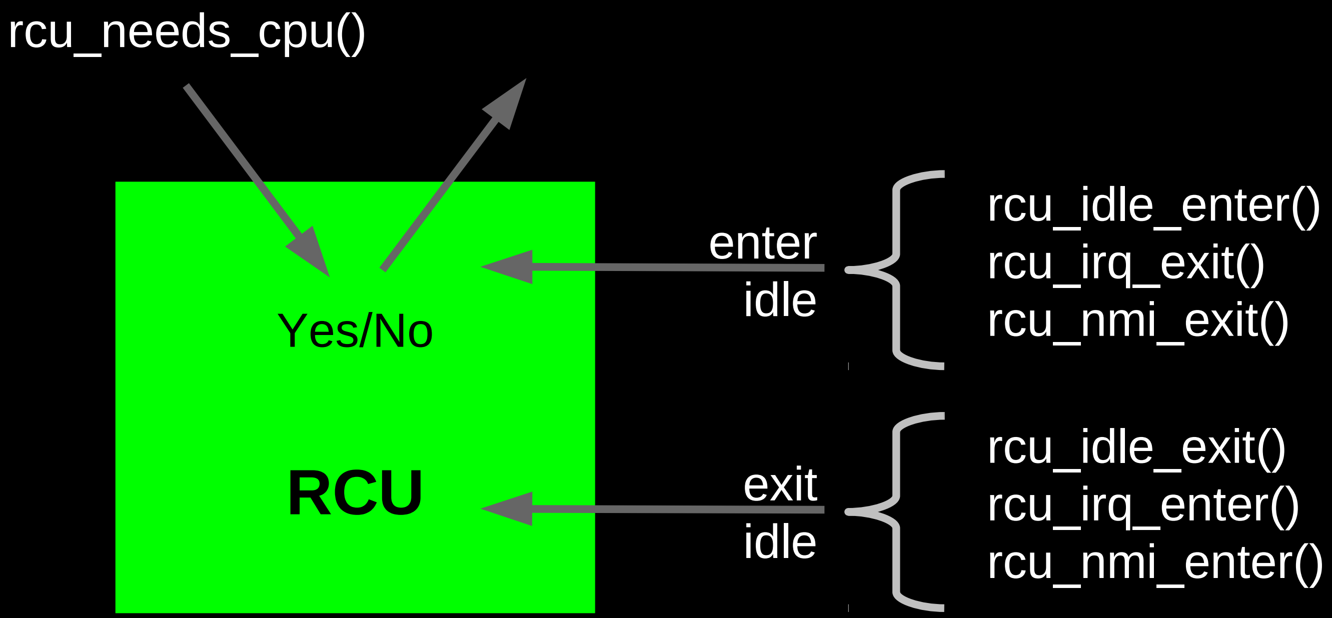


The more CPUs you have, the worse this effect gets

RCU and Dyntick Idle (AKA CONFIG_NO_HZ=y)

- List of implementations:
  - 2004: Dyntick-idle bit vector
    - Manfred Spraul locates theoretical bug
    - A few months before the mainframe guys encounter it
    - But after it has been in-tree for *four* years
  - 2008: -rt version (with Steven Rostedt)
    - Very complex: http://lwn.net/Articles/279077/
  - 2009: Separate out NMI accounting
    - Greatly simplified: No proof of correctness required
  - 2010: CONFIG_RCU_FAST_NO_HZ for small systems
    - Force last CPU into dyntick-idle mode
  - 2012: CONFIG_RCU_FAST_NO_HZ for large systems
    - Force CPUs with callbacks into dyntick-idle, but wake them up later

CONFIG_RCU_FAST_NO_HZ for Large Systems



CONFIG_RCU_FAST_NO_HZ for Large Systems

- Constraints:
  - The RCU core code is a state machine driven out of the scheduling-clock interrupt handler that runs primarily in softirq context
  - Cannot indefinitely delay callbacks: would otherwise result in hangs
  - Cannot spin indefinitely trying to enter dyntick-idle mode
    - At some point it is better to accept periodic scheduling-clock interrupts
  - Need to control idle-entry overhead if entering/exiting idle frequently
  - Cannot use conventional looping constructs due to deadlock issues
  - Cannot assume that rcu_needs_cpu() is called in a quiescent state
  - Some architectures enter interrupt handlers that they never exit
    - And vice versa

Initial Version of Code

```
void rcu_prepare_for_idle(int cpu)
{
        int i;

        while (rcu_cpu_has_callbacks(cpu)) {
                rcu_sched_qs();
                force_quiescent_state(&rcu_sched_state, 0);
                rcu_process_callbacks();
        }
}
```

Initial Version of Code

```
void rcu_prepare_f     te(int cpu)
{
        int i;

        while      _cpu_has_      backs(cpu))
                  u_sched_qs
                   rce_quiescen     te(&rcu_      ed_state, 0);
                    _process_callb      ();
        }
}
```

RCU callbacks might spawn more RCU callbacks indefinitely

Better a scheduling-clock interrupt than spinning while idle!

Limit Number of Attempts to RCU_IDLE_FLUSHES

```
void rcu_prepare_for_idle(int cpu)
{
        int i;

        for (i = 0; i < RCU_IDLE_FLUSHES; i++) {
                if (!rcu_cpu_has_callbacks(cpu))
                        return;
                rcu_sched_qs();
                force_quiescent_state(&rcu_sched_state, 0);
                rcu_process_callbacks();
        }
        invoke_rcu_core();
}
```

Limit Number of Attempts to RCU_IDLE_FLUSHES

```
void rcu_prepare_fo
{
        int i;

        for (i    ; i <      DLE_FLUSHES;      {
                   (!rcu_c      _callbacks(    )
                        retu
                  u_sched_qs();
                  rce_quiescent_s    (&rcu    ed_state, 0);
                    process_callbac
        }
        invoke_rcu_
}
```

High overhead for frequent switches to idle!

Hold Off Future Attempts if Unsuccessful

```
DEFINE_PER_CPU(unsigned long, rcu_dyntick_holdoff);

void rcu_prepare_for_idle(int cpu)
{
        int i;

        if (per_cpu(rcu_dyntick_holdoff, cpu) == jiffies)
                return;
        for (i = 0; i < RCU_IDLE_FLUSHES; i++) {
                if (!rcu_cpu_has_callbacks(cpu))
                        return;
                rcu_sched_qs();
                force_quiescent_state(&rcu_sched_state, 0);
                rcu_process_callbacks();
        }
        if (rcu_cpu_has_callbacks(cpu)) {
                per_cpu(rcu_dyntick_holdoff, cpu) = jiffies;
                invoke_rcu_core();
        }
}
```

Hold Off Future Attempts if Unsuccessful

```
DEFINE_PER_CPU(unsigned long, rcu_dyntick_holdoff);

void rcu_prepare_for_idl
{
        int i;

        if (per_c     cu_      k_holdoff, cpu)      ffies)
                     n;
        for (i =     i < RCU_       USHES; i++) {
                    !rcu_cpu_ha       backs(cpu))
                        return;
                    sched_qs();
                      quiescent_state(     sche     ate, 0);
                r     cess_callbacks()
        }
        if (rcu_cpu_             lbacks(cpu))
                per_cpu                        cpu) = jiffies;
                invoke_rcu_c
        }
}
```

Cannot clear all RCU callbacks often enough!

Allow Idle with Callbacks: Set Timer

```
DEFINE_PER_CPU(unsigned long, rcu_dyntick_holdoff);
DEFINE_PER_CPU(struct hrtimer, … );

void rcu_prepare_for_idle(int cpu)
{
        int i;

        if (per_cpu(rcu_dyntick_holdoff, cpu) == jiffies)
                return;
        for (i = 0; i < RCU_IDLE_FLUSHES; i++) {
                if (!rcu_cpu_has_callbacks(cpu))
                        return;
                rcu_sched_qs();
                force_quiescent_state(&rcu_sched_state, 0);
                rcu_process_callbacks();
        }
        if (rcu_cpu_has_callbacks(cpu))
                If (rcu_pending()) {
                        per_cpu(rcu_dyntick_holdoff, cpu) = jiffies;
                        invoke_rcu_core();
                } else
                        hrtimer_start( … );
}
```

Allow Idle with Callbacks: Set Timer

```
DEFINE_PER_CPU(unsigned long, rcu_dyntick_holdoff);
DEFINE_PER_CPU(struct hrtimer, … );

void rcu_prepare_for_idle(
{
        int i;

        if (per_cpu     dynt     doff, cpu) == jiffi
                re
        for (i = 0     RCU_IDLE_      ; i++) {
                i    u_cpu_has_ca      (cpu))
                        return;
                rc    ed_qs();
                fo    iescent_state(&rc      state
                rcu_    ss_callbacks();
        }
        if (rcu_cpu_ha     acks(cpu))
                If (rcu_     ()) {
                        pe              , cpu) = jiffies;
                        invoke_
                } else
                        hrtimer_start( … );
}
```

Results in useless hrtimer events!!!

Allow Idle with Callbacks: Set and Cancel Timer

```
DEFINE_PER_CPU(unsigned long, rcu_dyntick_holdoff);
DEFINE_PER_CPU(struct hrtimer, … );

void rcu_prepare_for_idle(int cpu)
{
        int i;

        if (per_cpu(rcu_dyntick_holdoff, cpu) == jiffies)
                return;
        for (i = 0; i < RCU_IDLE_FLUSHES; i++) {
                if (!rcu_cpu_has_callbacks(cpu))
                        return;
                rcu_sched_qs();
                force_quiescent_state(&rcu_sched_state, 0);
                rcu_process_callbacks();
        }
        if (rcu_cpu_has_callbacks(cpu))
                if (rcu_pending()) {
                        per_cpu(rcu_dyntick_holdoff, cpu) = jiffies;
                        invoke_rcu_core();
                } else
                        hrtimer_start( … );
}

void rcu_cleanup_after_idle(int cpu)
{
        hrtimer_cancel(( … );
}
```

Allow Idle with Callbacks: Set and Cancel Timer

```
DEFINE_PER_CPU(unsigned long, rcu_dyntick_holdoff);
DEFINE_PER_CPU(struct hrtimer, … );

void rcu_prepare_for_idle(int c
{
        int i;

        if (per_cpu(rcu            cpu) == jiffies)
                retur
        for (i = 0; i    _IDLE_FL        ++) {
                if (!    u_has_call        ))
                         turn;
                rcu_     qs();
                force    cent_state(&rcu_s         te, 0);
                rcu_p    _callbacks();
        }
        if (rcu_cpu_ha        acks(cpu))
                if (rcu_       ()) {
                        p       rcu_dyntick_holdoff, cp         s;
                        in        core();
                } else
                        hrtime
}

void rcu_cleanup_after_idle(int cpu)
{
        hrtimer_cancel(( … );
}
```

kfree_rcu() callbacks don't need timer!!!

Allow Idle with Callbacks: Lazy RCU Callbacks

```
DEFINE_PER_CPU(unsigned long, rcu_dyntick_holdoff);
DEFINE_PER_CPU(struct hrtimer, … );


void rcu_prepare_for_idle(int cpu)
{
        int i;


        if (per_cpu(rcu_dyntick_holdoff, cpu) == jiffies)
                return;
        for (i = 0; i < RCU_IDLE_FLUSHES; i++) {
                if (!rcu_cpu_has_callbacks(cpu))
                        return;
                rcu_sched_qs();
                force_quiescent_state(&rcu_sched_state, 0);
                rcu_process_callbacks();
        }
        if (rcu_cpu_has_callbacks(cpu))
                if (rcu_pending()) {
                        per_cpu(rcu_dyntick_holdoff, cpu) = jiffies;
                        invoke_rcu_core();
                } else if (rcu_cpu_has_nonlazy_callbacks())
                        hrtimer_start( … );
}

void rcu_cleanup_after_idle(int cpu)
{
        hrtimer_cancel(( … );
}
```

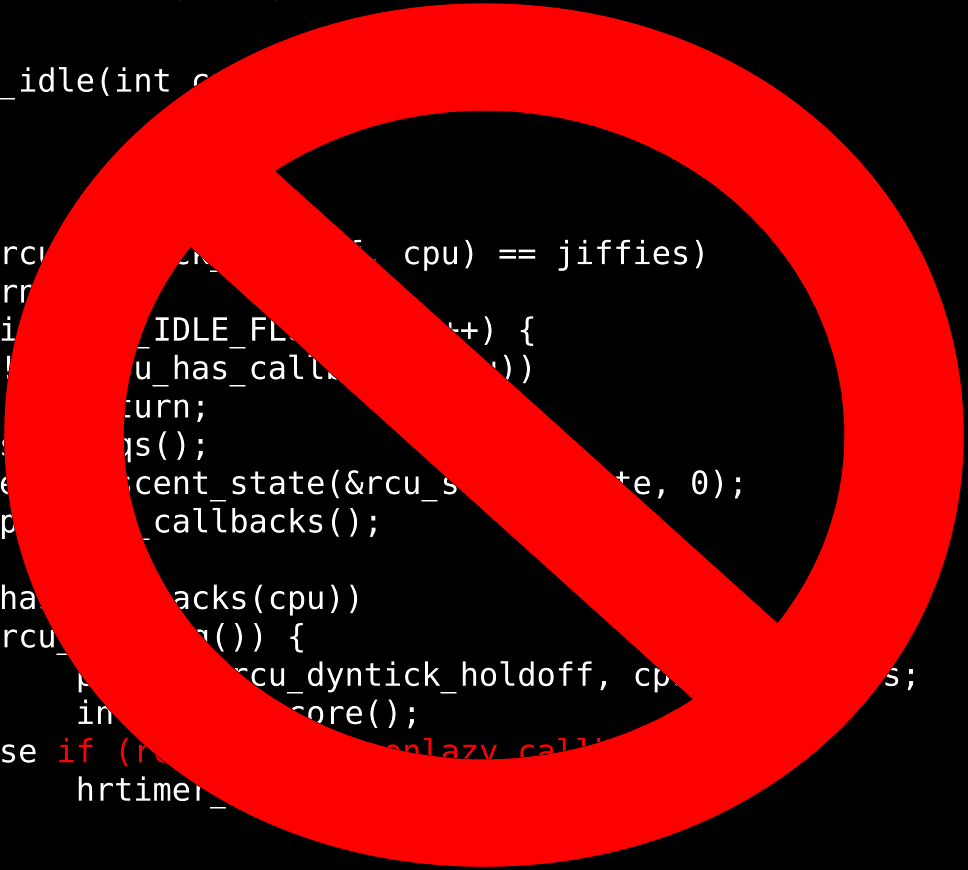
Allow Idle with Callbacks: Lazy RCU Callbacks

```
DEFINE_PER_CPU(unsigned long, rcu_dyntick_holdoff);
DEFINE_PER_CPU(struct hrtimer, … );

void rcu_prepare_for_idle(int c
{
        int i;

        if (per_cpu(rcu      ck        f, cpu) == jiffies)
                return
        for (i = 0; i      _IDLE_FL        ++) {
                if (!      u_has_call        ))
                          urn;
                rcu_       qs();
                force      cent_state(&rcu_s         te, 0);
                rcu_p      _callbacks();
        }
        if (rcu_cpu_ha      acks(cpu))
                if (rcu_      g()) {
                        p      rcu_dyntick_holdoff, cp            s;
                        in      core();
                } else if (r        nlazy_call
                        hrtimer_
}

void rcu_cleanup_after_idle(int cpu)
{
        hrtimer_cancel(( … );
}
```



What if some task wakes??? Scheduling latency!!!

Controlling Scheduling Latency

```
DEFINE_PER_CPU(unsigned long, rcu_dyntick_holdoff);
DEFINE_PER_CPU(struct hrtimer, … );

void rcu_prepare_for_idle(int cpu)
{
        int i;

        if (per_cpu(rcu_dyntick_holdoff, cpu) == jiffies)
                return;
        for (i = 0; i < RCU_IDLE_FLUSHES; i++) {
                if (!rcu_cpu_has_callbacks(cpu) || need_resched())
                        return;
                rcu_sched_qs();
                force_quiescent_state(&rcu_sched_state, 0);
                rcu_process_callbacks();
        }
        if (rcu_cpu_has_callbacks(cpu))
                if (rcu_pending()) {
                        per_cpu(rcu_dyntick_holdoff, cpu) = jiffies;
                        invoke_rcu_core();
                } else if (rcu_cpu_has_nonlazy_callbacks())
                        hrtimer_start( … );
}

void rcu_cleanup_after_idle(int cpu)
{
        hrtimer_cancel(( … );
}
```

Controlling Scheduling Latency

```
DEFINE_PER_CPU(unsigned long, rcu_dyntick_holdoff);
DEFINE_PER_CPU(struct hrtimer, … );

void rcu_prepare_for_idle(int c
{
        int i;

        if (per_cpu(rcu                  cpu) == jiffies)
                return
        for (i = 0; i      _IDLE_FL        ++) {
                if (!      u_has_call         ) || need_resched(
                           turn;
                rcu_       qs();
                force      cent_state(&rcu_s            te, 0);
                rcu_p      _callbacks();
        }
        if (rcu_cpu_ha       acks(cpu))
                if (rcu_      g()) {
                        p       rcu_dyntick_holdoff, cp             s;
                        in         core();
                } else if (r            nlazy_call
                        hrtimer_
}

void rcu_cleanup_after_idle(int cpu)
{
        hrtimer_cancel(( … );
}
```

Lockdep begs to differ!!!

Other Issues and Fixes

- Lockdep issues: Use state-machine implementation
  - Per-CPU loop variable
  - Half of loop executed during idle entry
  - The other half is executed within softirq
    - Exiting softirq initiates another idle entry
- Jiffies counter overflow
  - Do “per_cpu(rcu_dyntick_holdoff, cpu) = jiffies – 1” on non-holdoff exit
- The hrtimer handler never is actually executed!
  - Too bad!!! Life is like that sometimes!
- Special case for kfree_rcu() is OK, but call_rcu() mostly just frees memory
  - Expect a call_rcu_lazy() in a -rcu git tree near you...
- User code incurs scheduling-clock ticks even when only one per CPU
  - Frederic Weisbecker is working on this

Lessons Learned and Relearned

Lessons Learned, Old and New

- Workload matters!!!
  - Different workloads have different requirements
  - A given workload's requirements change over time
    - More important, one's understanding of requirements changes over time!
  - Supporting a single workload is easier than supporting many of them

Lessons Learned, Old and New

- Workload matters!!!
  - Different workloads have different requirements
  - A given workload's requirements change over time
    - More important, one's understanding of requirements changes over time!
  - Supporting a single workload is easier than supporting many of them
- Energy-efficiency and performance benchmarkers
  - You would never believe what either group will do for 5%...

Lessons Learned, Old and New

- Workload matters!!!
  - Different workloads have different requirements
  - A given workload's requirements change over time
    - More important, one's understanding of requirements changes over time!
  - Supporting a single workload is easier than supporting many of them
- Energy-efficiency and performance benchmarkers
  - You would never believe what either group will do for 5%...
- Median age of randomly chosen line of RCU code: < 2 years

Lessons Learned, Old and New

- Workload matters!!!
  - Different workloads have different requirements
  - A given workload's requirements change over time
    - More important, one's understanding of requirements changes over time!
  - Supporting a single workload is easier than supporting many of them
- Energy-efficiency and performance benchmarkers
  - You would never believe what either group will do for 5%...
- Median age of randomly chosen line of RCU code: < 2 years
- The guys who request an enhancement are rarely the guys who are willing to test your patches

Lessons Learned, Old and New

- Workload matters!!!
  - Different workloads have different requirements
  - A given workload's requirements change over time
    - More important, one's understanding of requirements changes over time!
  - Supporting a single workload is easier than supporting many of them
- Energy-efficiency and performance benchmarkers
  - You would never believe what either group will do for 5%...
- Median age of randomly chosen line of RCU code: < 2 years
- The guys who request an enhancement are rarely the guys who are willing to test your patches
- The importance of the community

A Brief History of RCU Issues

- ~1993: SMP scalability (30 CPUs) for RDBMS workloads
- 1996: NUMA (64 CPUs) for RDBMS workloads
- 2002: SMP scalability (~30 CPUs) for general workloads
- 2004: SMP scalability (~512 CPUs) for HPC workloads
  - And some concern about energy efficiency
- 2005: Real-time response (~4 CPUs)
- 2008: SMP scalability (>1024 CPUs) for HPC workloads
  - 100s of CPUs for more general workloads
- 2009: Real-time response (~30 CPUs) for general workloads
- 2010: Energy efficiency (~2 CPUs), real-time response when CPU-bound
- 2011: Energy efficiency (lots of CPUs)
- 2012: RCU causes 200-microsecond latency spikes...

A Brief History of RCU Issues

- ~1993: SMP scalability (30 CPUs) for RDBMS workloads
- 1996: NUMA (64 CPUs) for RDBMS workloads
- 2002: SMP scalability (~30 CPUs) for general workloads
- 2004: SMP scalability (~512 CPUs) for HPC workloads
  - And some concern about energy efficiency
- 2005: Real-time response (~4 CPUs)
- 2008: SMP scalability (>1024 CPUs) for HPC workloads
  - 100s of CPUs for more general workloads
- 2009: Real-time response (~30 CPUs) for general workloads
- 2010: Energy efficiency (~2 CPUs), real-time response when CPU-bound
- 2011: Energy efficiency (lots of CPUs)
- 2012: RCU causes 200-microsecond latency spikes... For NR_CPUS=4096

And So I Owe The Linux Community Many Thanks

- Because of the many RCU-related challenges from the Linux community, some of my most important work and collaborations have been in the past ten years

And So I Owe The Linux Community Many Thanks

- Because of the many RCU-related challenges from the Linux community, some of my most important work and collaborations have been in the past ten years
- Not many people my age can truthfully say that
- Here is hoping for ten more years!!! ;-)

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.

Questions