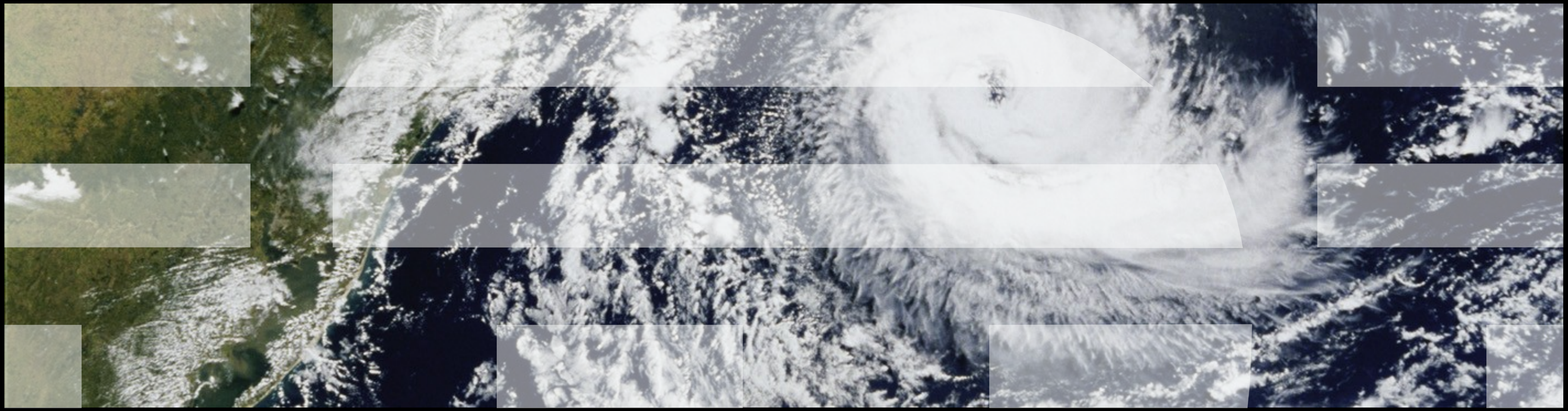
Paul E. McKenney, IBM Distinguished Engineer, Linux Technology Center

linux.conf.au January 31, 2013

Real-Time Response on Multicore Systems:
It is Bigger Than I Thought



History of Real Time (AKA Preemptible) RCU

- December 2004: realized that I needed to fix RCU...
- March 2005: first hint that solution was possible
  - I proposed flawed approach, Esben Neilsen proposed flawed but serviceable approach
- May 2005: first design fixing flaws in Esben's approach
- June 2005: first patch submitted to LKML
- August 2005: patch accepted in -rt
- November 2006: priority boosting patch
- Early 2007: priority boosting accepted into -rt
- September 2007: preemptible RCU w/o atomics
- January 2008: preemptible RCU in mainline
- December 2009: scalable preemptible RCU in mainline
- July 2011: RCU priority boosting in mainline

The -rt Patchset Was Used in Production Early On

- 2006: aggressive real-time on 64-bit systems
  - Real-time Linux kernel (x86_64, 4-8 processors, deadlines down to 70 microseconds, measured latencies less than 40 microseconds)
    - I only did RCU. Ingo Molnar, Sven Dietrich, K. R. Foley, Thomas Gleixner, Gene Heskett, Bill Huey, Esben Nielsen, Nick Piggin, Lee Revell, Steven Rostedt, Michal Schmidt, Daniel Walker, and Karsten Wiese did the real work, as did many others joining the project later on.
    - Plus a huge number of people writing applications, supporting customers, packaging distros, and actually using -rt ...

The -rt Patchset Was Used in Production Early On

- 2006: aggressive real-time on 64-bit systems
  - Real-time Linux kernel (x86_64, 4-8 processors, deadlines down to 70 microseconds, measured latencies less than 40 microseconds)
    - I only did RCU. Ingo Molnar, Sven Dietrich, K. R. Foley, Thomas Gleixner, Gene Heskett, Bill Huey, Esben Nielsen, Nick Piggin, Lee Revell, Steven Rostedt, Michal Schmidt, Daniel Walker, and Karsten Wiese did the real work, as did many others joining the project later on.
    - Plus a huge number of people writing applications, supporting customers, packaging distros, and actually using -rt …
    - And kudos to Linus for actually putting up with us...

The -rt Patchset Was Used in Production Early On

- 2006: aggressive real-time on 64-bit systems
  - Real-time Linux kernel (x86_64, 4-8 processors, deadlines down to 70 microseconds, measured latencies less than 40 microseconds)
    - I only did RCU. Ingo Molnar, Sven Dietrich, K. R. Foley, Thomas Gleixner, Gene Heskett, Bill Huey, Esben Nielsen, Nick Piggin, Lee Revell, Steven Rostedt, Michal Schmidt, Daniel Walker, and Karsten Wiese did the real work, as did many others joining the project later on.
    - Plus a huge number of people writing applications, supporting customers, packaging distros, and actually using -rt …
    - And kudos to Linus for actually putting up with us... Most of the time, anyway

The -rt Patchset Was Used in Production Early On

- 2006: aggressive real-time on 64-bit systems
  - Real-time Linux kernel (x86_64, 4-8 processors, deadlines down to 70 microseconds, measured latencies less than 40 microseconds)
    - I only did RCU. Ingo Molnar, Sven Dietrich, K. R. Foley, Thomas Gleixner, Gene Heskett, Bill Huey, Esben Nielsen, Nick Piggin, Lee Revell, Steven Rostedt, Michal Schmidt, Daniel Walker, and Karsten Wiese did the real work, as did many others joining the project later on.
    - Plus a huge number of people writing applications, supporting customers, packaging distros, and actually using -rt …
    - And kudos to Linus for actually putting up with us... Most of the time, anyway
- But some were not inclined to believe SMP -rt worked, so...

The Writeup

Source: Paul E. McKenney "SMP and Embedded Real Time", Linux Journal, Feb 2007, http://www.linuxjournal.com/article/9361

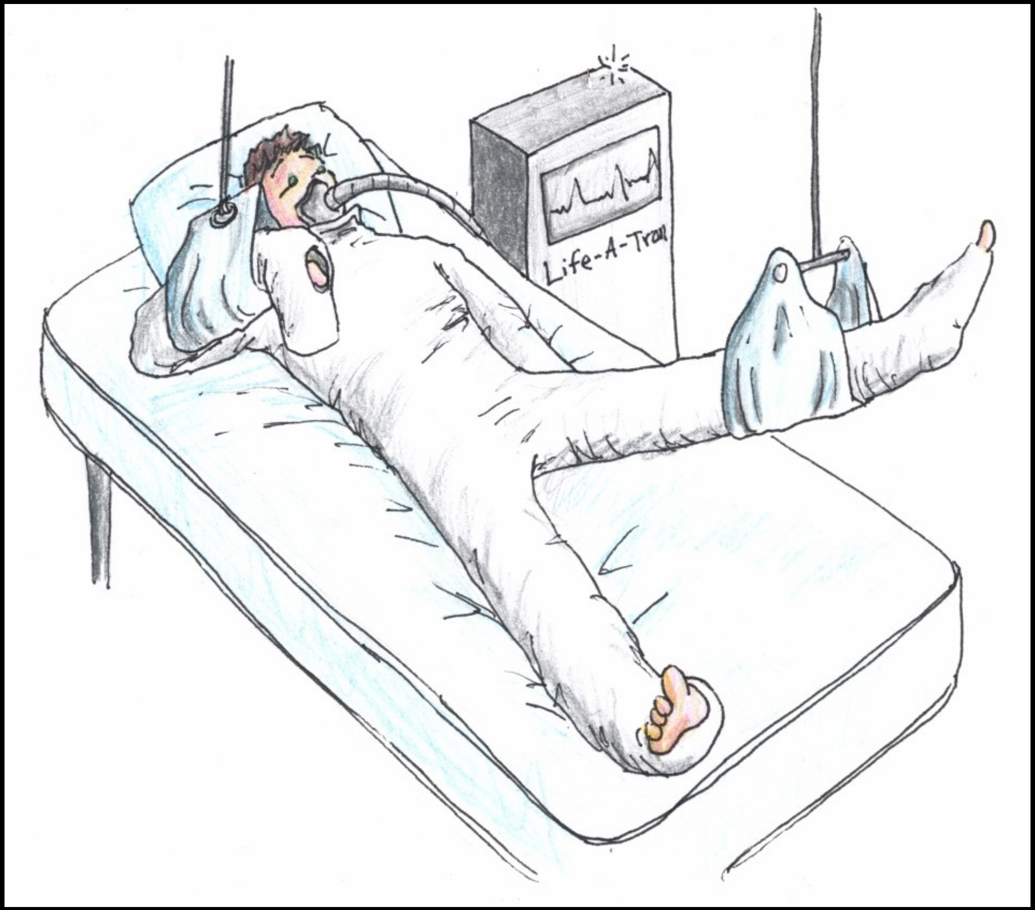
The Limits of Hard Real Time in the Hard Real World

You show me a hard real-time system,
and I will show you a hammer that will cause it to miss its deadlines.

The Limits of Hard Real Time in the Hard Real World

You can make your system more robust,
but I can get a bigger hammer.

But Do Hardware Failures Count?



Rest assured, sir, that should there be a failure,
it will not be due to software!

"SMP and Embedded Real Time"

- Five Real-Time Myths:
  - Embedded systems are always uniprocessor systems
  - Parallel programming is mind crushingly difficult
  - Real time must be either hard or soft
  - Parallel real-time programming is impossibly difficult
  - There is no connection between real-time and enterprise systems
- Despite the cute cartoons, this message was not well-received in all quarters...

Nevertheless, I Believe That “SMP and Embedded Real Time” Has Stood the Test of Time

Nevertheless, I Believe That “SMP and Embedded Real Time” Has Stood the Test of Time

Except For One Really Big Error...

Big Error in "SMP and Embedded Real Time"

- February 8, 2012
  - Dimitri Sivanic reports 200+ microsecond latency spikes from RCU
  - My initial response, based on lots of experience otherwise:
    - "You must be joking!!!"

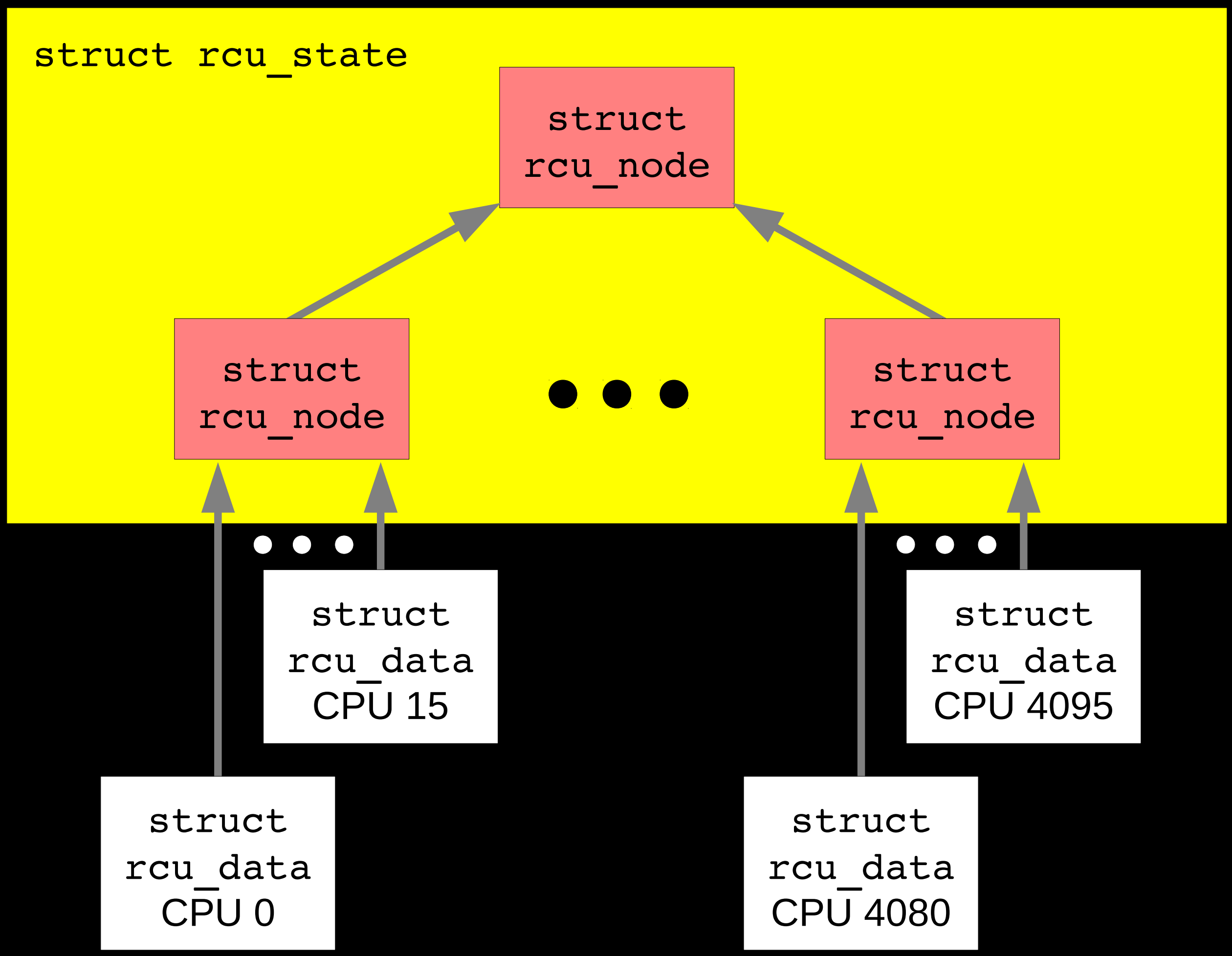
Big Error in “SMP and Embedded Real Time”

- February 8, 2012
  - Dimitri Sivanic reports 200+ microsecond latency spikes from RCU
  - My initial response, based on lots of experience otherwise:
    - “You must be joking!!!”
  - Further down in Dimitri's email: NR_CPUS=4096
    - “You mean it took *only* 200 microseconds?”

Big Error in "SMP and Embedded Real Time"

- February 8, 2012
  - Dimitri Sivanic reports 200+ microsecond latency spikes from RCU
  - My initial response, based on lots of experience otherwise:
    - "You must be joking!!!"
  - Further down in Dimitri's email: NR_CPUS=4096
    - "You mean it took *only* 200 microseconds?"
- My big error: I was thinking in terms of 4-8 CPUs, maybe eventually as many as 16-32 CPUs
  - More than two orders of magnitude too small!!!

RCU Initialization



Level 0: 1 rcu_node

Level 1: 4 rcu_nodes

Level 2: 256 rcu_nodes

Total: 261 rcu_nodes

But Who Cares About Such Huge Systems?

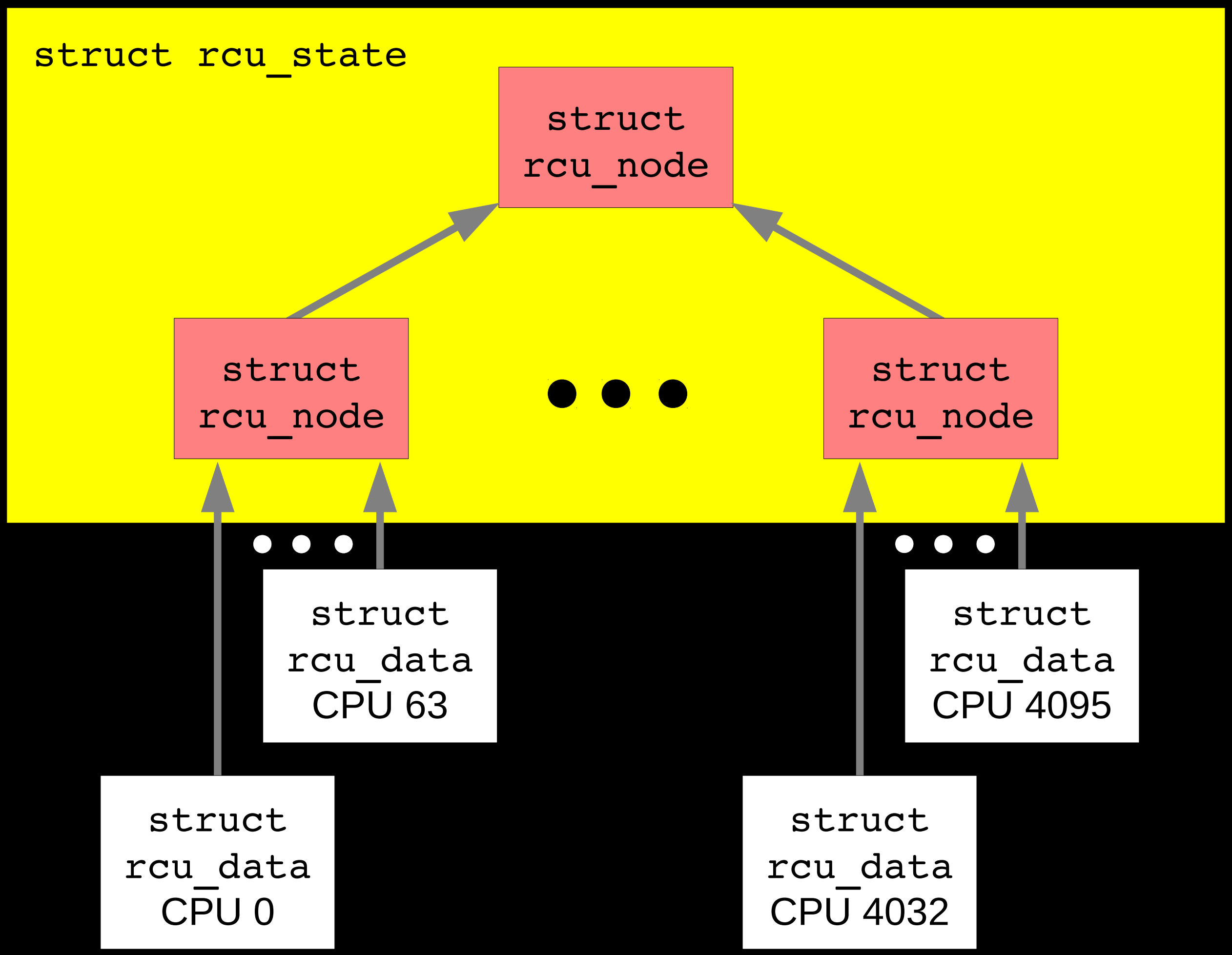
But Who Cares About Such Huge Systems?

- Their users do! :-)
- And you need to care about them as well

But Who Cares About Such Huge Systems?

- Their users do! :-)
- And you need to care about them as well
- Systems are still getting larger
  - I do remember 8-CPU systems being called “huge” only ten years ago
  - Today, *laptops* with 8 CPUs are readily available
  - And CONFIG_SMP=n is now inadequate for many *smartphones*
  - And the guys with huge systems provide valuable testing services
- Some Linux distributions build with NR_CPUS=4096
  - Something about only wanting to provide a single binary...
  - RCU must adjust, for example, increasing CONFIG_RCU_FANOUT

RCU Initialization, CONFIG_RCU_FANOUT=64



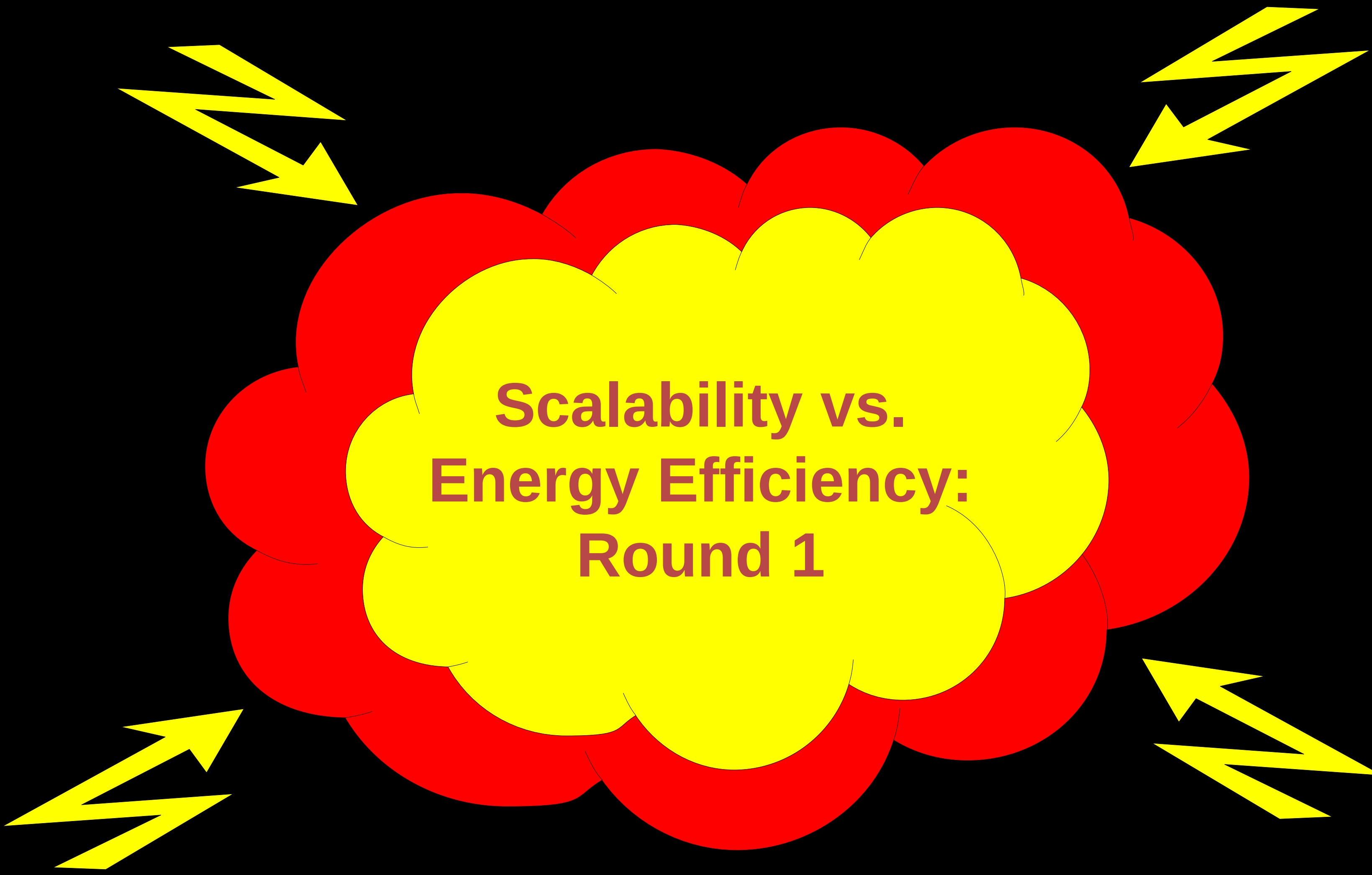
Level 0: 1 rcu_node

Level 2: 64 rcu_nodes

Total: 65 rcu_nodes

Decreases latency from 200+ to 60-70 microseconds. "Barely acceptable" to users. But...

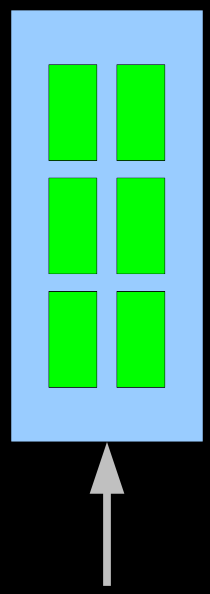
CONFIG_RCU_FANOUT=64 Consequences



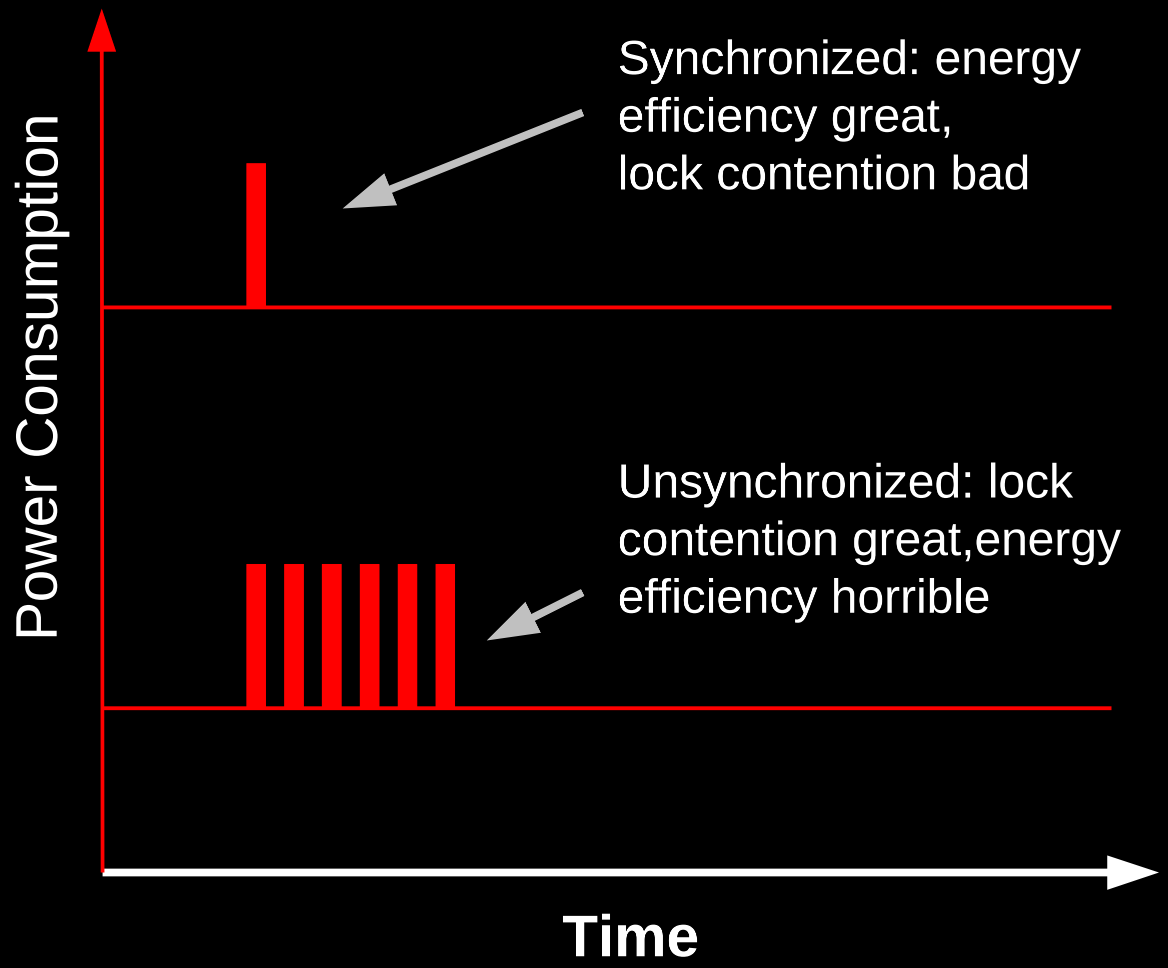
CONFIG_RCU_FANOUT=64 Consequences

- Huge systems want 64 CPUs per leaf rcu_node structure
- Smaller energy-efficient systems want scheduling-clock interrupts delivered to each socket simultaneously
  - Reduces the number of per-socket power transitions under light load
- If all 64 CPUs attempt to acquire their leaf rcu_node structure's lock concurrently: Massive lock contention

Issues With Scheduler-Clock Synchronization



Six-CPU package with single power domain



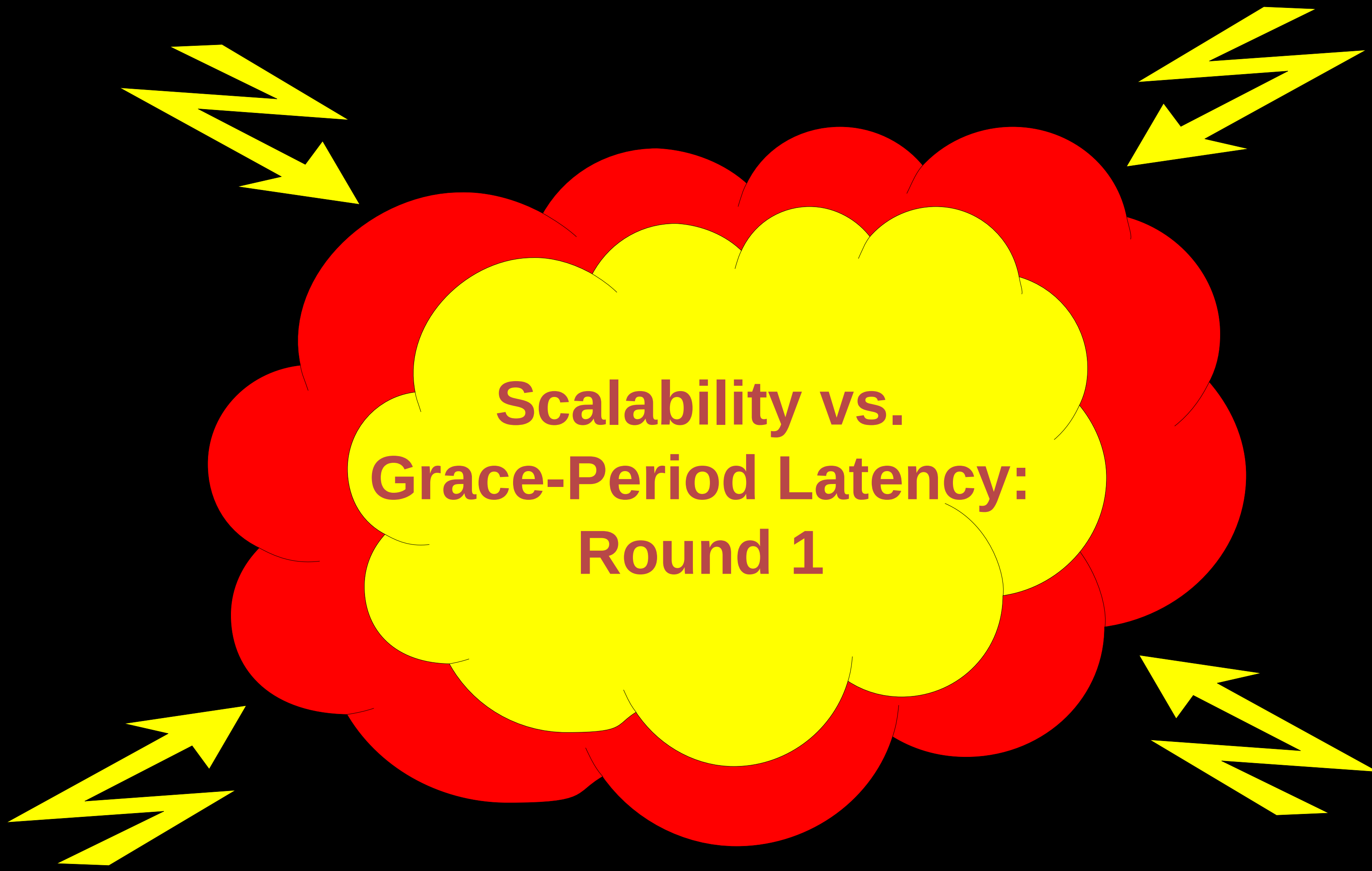
CONFIG_RCU_FANOUT=64 Consequences

- Huge systems want 64 CPUs per leaf rcu_node structure
- Smaller energy-efficient systems want scheduling-clock interrupts delivered to each socket simultaneously
  - Reduces the number of per-socket power transitions under light load
- If all 64 CPUs attempt to acquire their leaf rcu_node structure's lock concurrently: Massive lock contention
- Solution: Mike Galbraith added a boot parameter controlling scheduling-clock-interrupt skew
  - Later, Frederic Weisbecker's patch should help, but still have the possibility of all CPUs taking scheduling-clock interrupts
- Longer term: schedule events for energy and scalability

Unintended Consequences

- RCU polls CPUs to learn which are in dyntick-idle mode
  - force_quiescent_state() samples per-CPU counter
- Only one force_quiescent_state() at a time per RCU flavor
  - Mediated by trylock
- When 4096 CPUs trylock the same lock simultaneously, the results are not pretty: massive memory contention
- Immediate solution (Dimitri Sivanic):
  - Better mapping of rcu_state fields onto cachelines
  - Longer delay between force_quiescent_state() invocations, but...

Longer Polling Delay Consequences



Increased Polling Interval Consequences

- Increasing the polling interval increases the expected grace-period latency
- And people are already complaining about the grace periods taking too long!

Increased Polling Interval Consequences

- Increasing the polling interval increases the expected grace-period latency
- And people are already complaining about the grace periods taking too long!
- Short-term solution: Control polling interval via boot parameter/sysfs; people can choose what works for them

Increased Polling Interval Consequences

- Increasing the polling interval increases the expected grace-period latency
- And people are already complaining about the grace periods taking too long!
- Short-term solution: Control polling interval via boot parameter/sysfs; people can choose what works for them
- Longer-term solution: Move grace period startup, polling, and cleanup to kthread, eliminating force_quiescent_state()'s lock
  - But this does not come for free...
  - And there are force_quiescent_state() calls from RCU_FAST_NO_HZ

Grace-Period kthread Issues

- Increases binding between RCU and the scheduler
- Single lock mediates kthread wait_event()/wake_up()
  - But preemption points reduce PREEMPT=n latency
  - So there is at least some potential benefit from taking this path

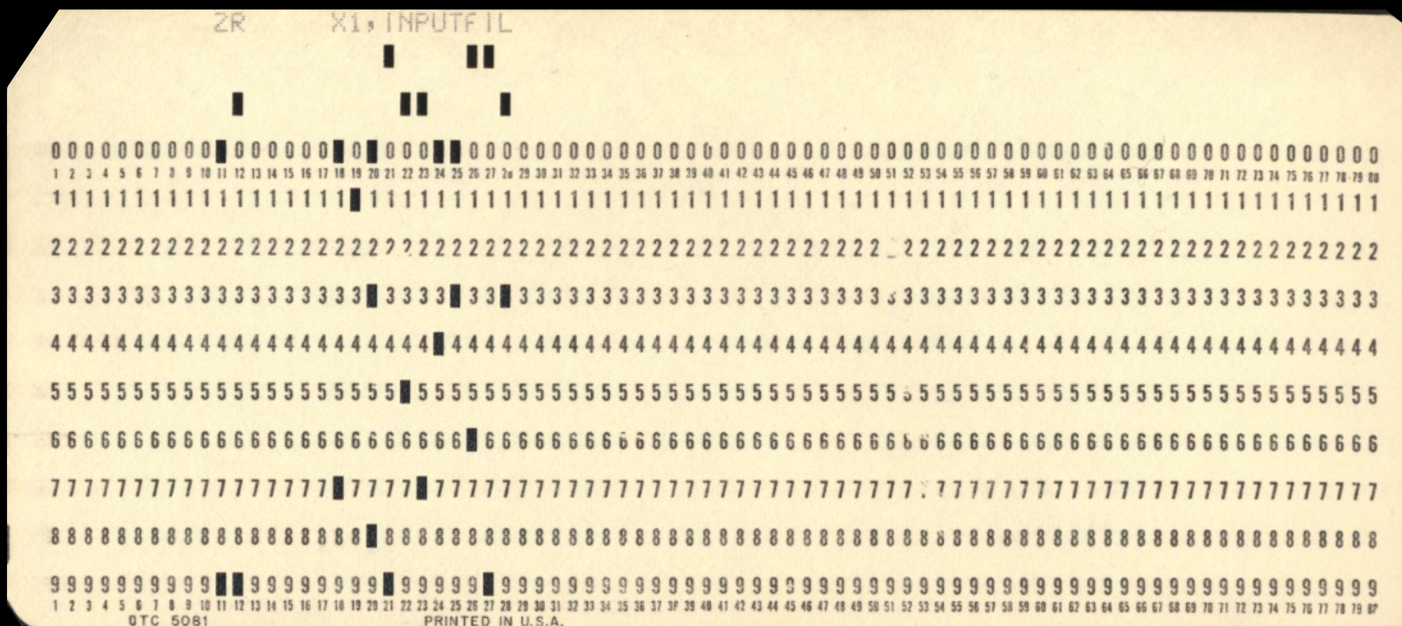
Grace-Period kthread Issues and Potential Benefits

- Increases binding between RCU and the scheduler
- Single lock mediates kthread wait_event()/wake_up()
  - But preemption points reduce PREEMPT=n latency
  - So there is at least some potential benefit from taking this path
- Estimate of latency reduction:
  - Reducing rcu_node structures from 261 to 65 resulted in latency reduction from roughly 200 to 70 microseconds
  - Reducing rcu_node structures to *one* per preemption opportunity might reduce latency to about 30 microseconds (linear extrapolation)
  - But why not just run the test?

Grace-Period kthread Issues and Potential Benefits

- Increases binding between RCU and the scheduler
- Single lock mediates kthread wait_event()/wake_up()
  - But preemption points reduce PREEMPT=n latency
  - So there is at least some potential benefit from taking this path
- Estimate of latency reduction:
  - Reducing rcu_node structures from 261 to 65 resulted in latency reduction from roughly 200 to 70 microseconds
  - Reducing rcu_node structures to *one* per preemption opportunity might reduce latency to about 30 microseconds (linear extrapolation)
  - But why not just run the test?
    - Because time on a 4096-CPU system is hard to come by
    - Fortunately, I have a very long history of relevant experience...

Coping With 4096-CPU System Scarcity



About That Single Global Lock...

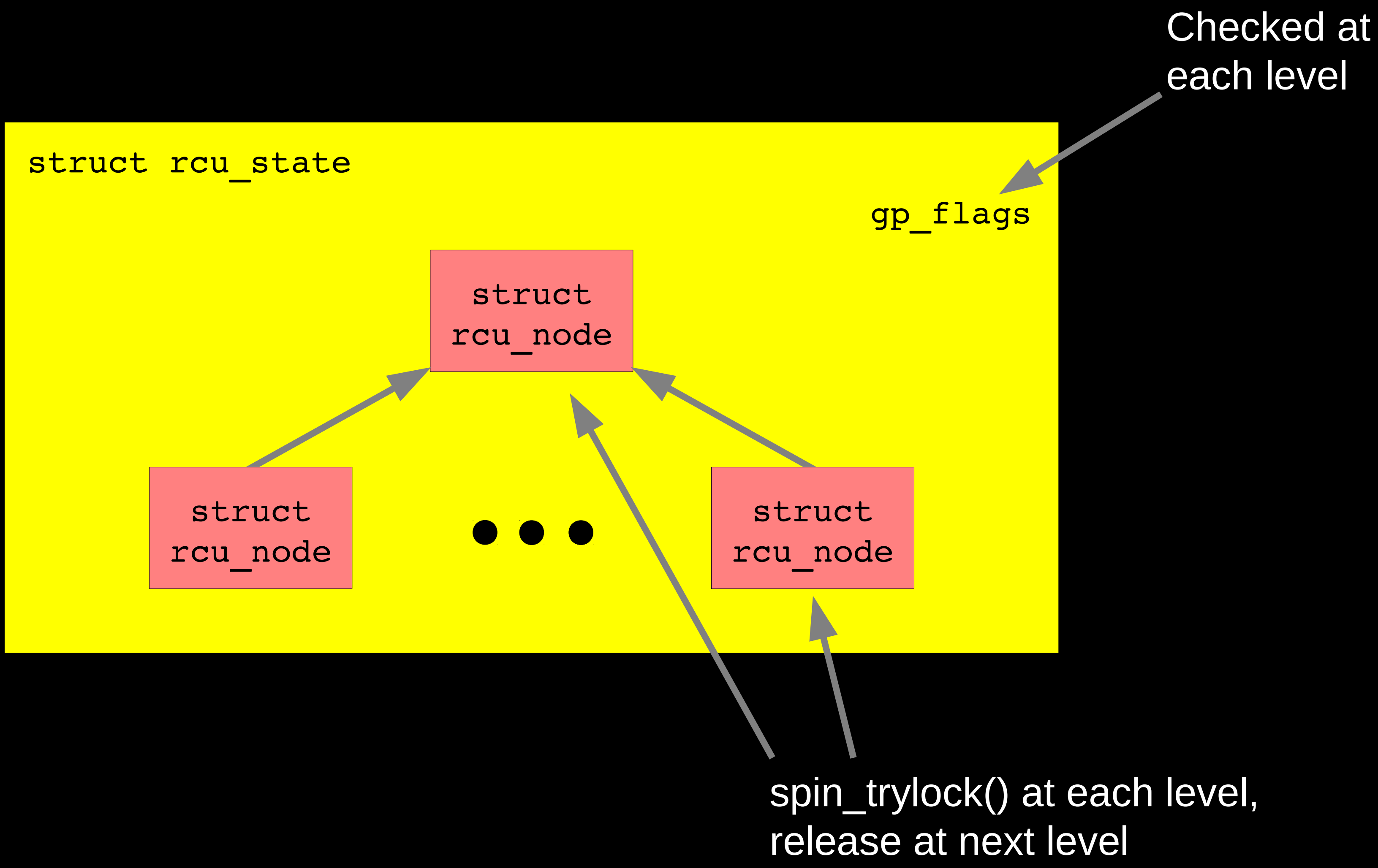
About That Single Global Lock...

- Grace-period operations are global events
  - So if already running or being awakened, no action required
- This situation can be handled by a variation on a tournament lock (Graunke & Thakkar 1990)

About That Single Global Lock...

- Grace-period operations are global events
  - So if already running or being awakened, no action required
- This situation can be handled by a variation on a tournament lock (Graunke & Thakkar 1990)
  - A variation that does not share the poor performance noted by Graunke and Thakkar

Conditional Tournament Lock



Conditional Tournament Lock Code

```
rnp = per_cpu_ptr(rsp->rda, raw_smp_processor_id())->mynode;
for (; rnp != NULL; rnp = rnp->parent) {
  ret = (ACCESS_ONCE(rsp->gp_flags) & RCU_GP_FLAG_FQS) ||
        !raw_spin_trylock(&rnp->fqslock);
  if (rnp_old != NULL)
    raw_spin_unlock(&rnp_old->fqslock);
  if (ret) {
    rsp->n_force_qs_lh++;
    return;
  }
  rnp_old = rnp;
}
```

Conditional Tournament Lock Code

```
rnp = per_cpu_ptr(rsp->rda, raw_smp_processor_id())->mynode;
for (; rnp != NULL; rnp = rnp->parent) {
  ret = (ACCESS_ONCE(rsp->gp_flags) & RCU_GP_FLAG_FQS) ||
        !raw_spin_trylock(&rnp->fqslock);
  if (rnp_old != NULL)
    raw_spin_unlock(&rnp_old->fqslock);
  if (ret) {
    rsp->n_force_qs_lh++;
    return;
  }
  rnp_old = rnp;
}
```

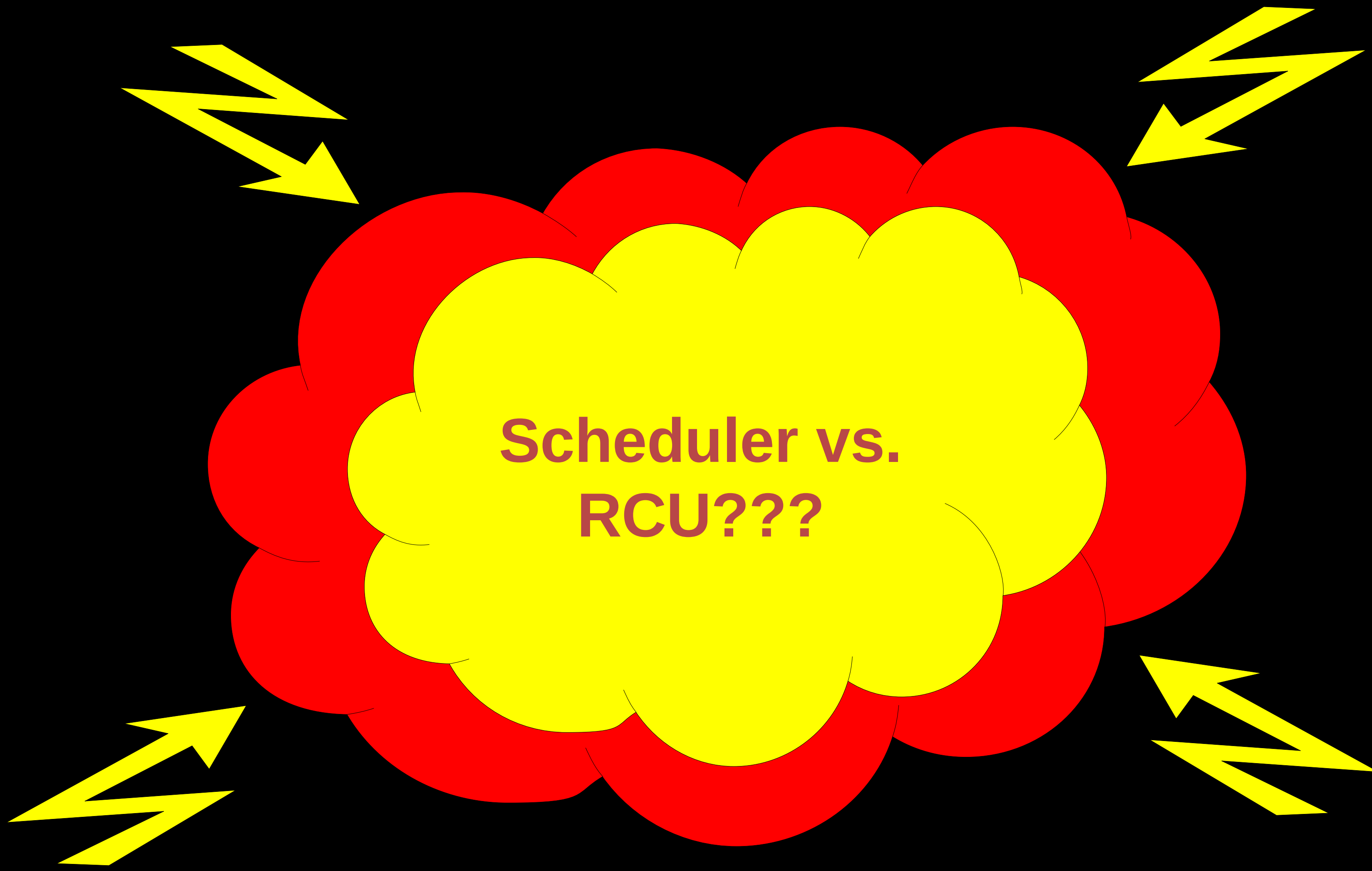
Effectiveness TBD

Other Possible Issues

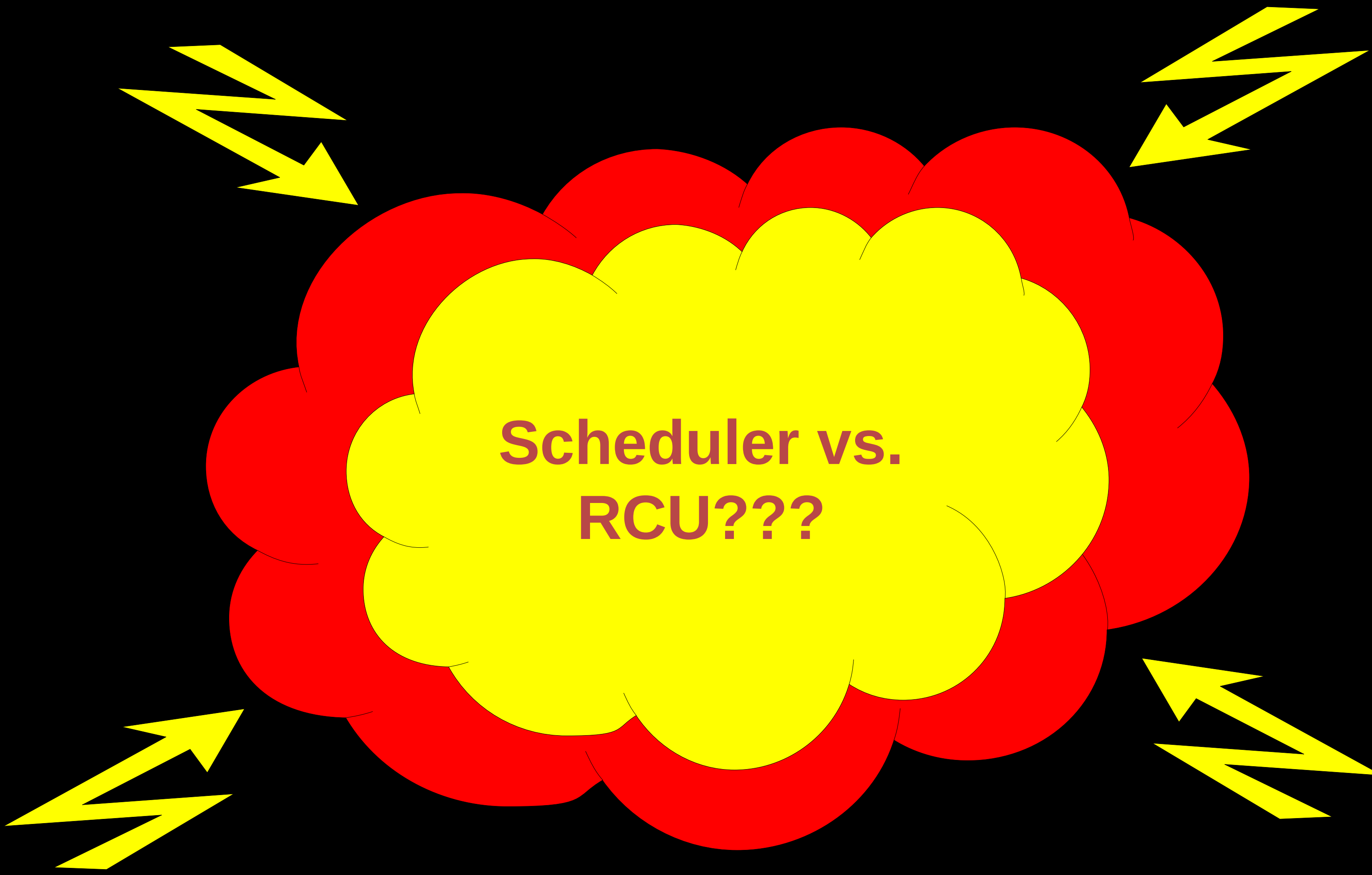
Other Possible Issues

- The synchronize_*_expedited() primitives loop over all CPUs
  - Parallelize? Optimize for dyntick-idle state?
- The rcu_barrier() primitives loop over all CPUs
  - Parallelize? Avoid running on other CPUs?
- Should force_quiescent_state() use state in non-leaf rcu_nodes?
  - This actually degrades worst-case behavior
- Lots of force_quiescent_state() use from RCU_FAST_NO_HZ
  - Use callback numbering to (hopefully) get rid of this
- Grace-period initialization/cleanup hits all rcu_node structures
  - Parallelize?
- NR_CPUS=4096 on small systems (RCU handles at boot)
- And, perhaps most important...

Possible Issue With RCU in a kthread

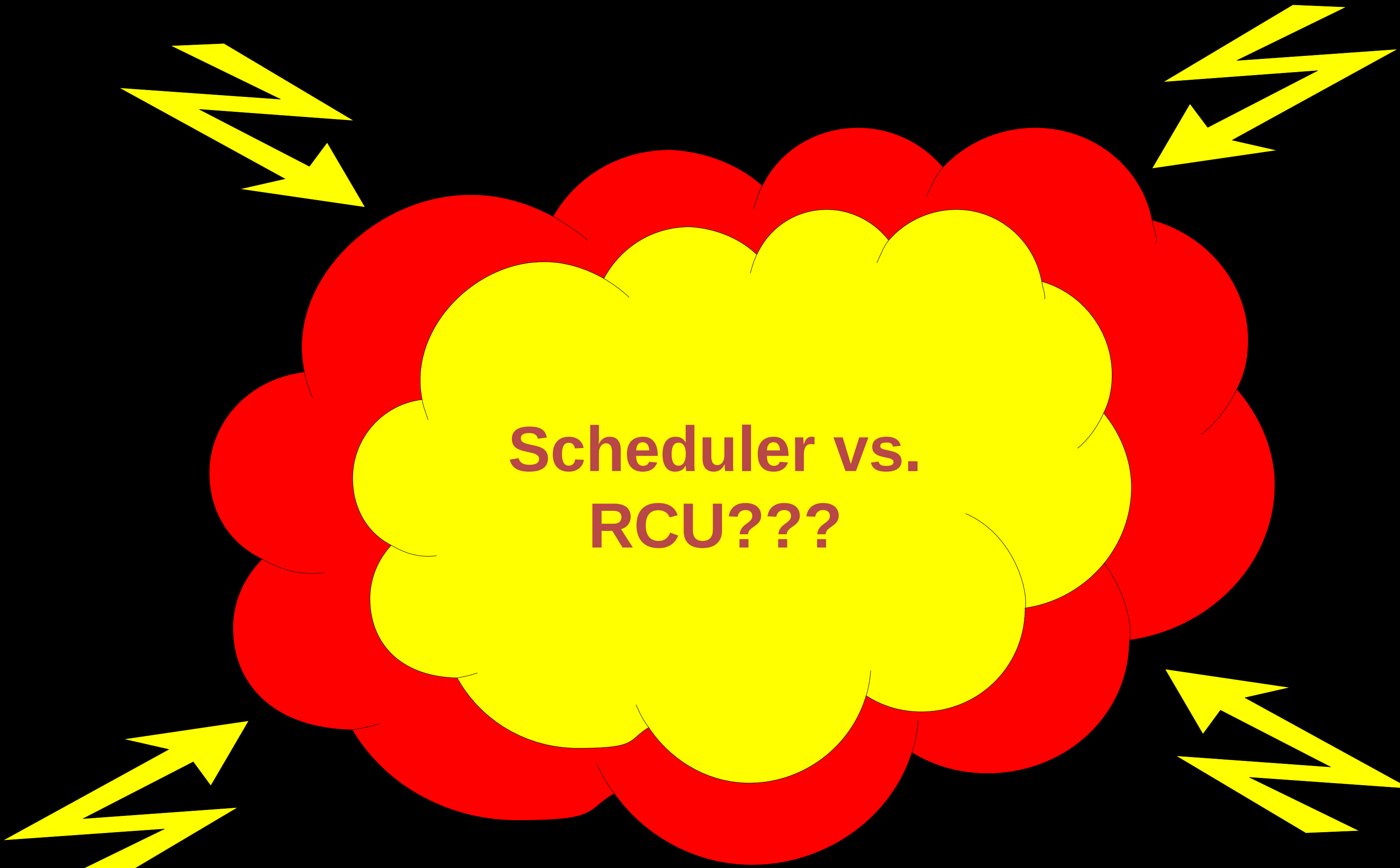


Possible Issue With RCU in a kthread



When these two fight, they both lose!

Possible Issue With RCU in a kthread



When these two fight, they both lose!
Much better if they both win!!!

The Linux Scheduler and RCU

- RCU uses the scheduler and the scheduler uses RCU
  - Plenty of opportunity for both RCU and the scheduler to lose big time!
  - See for example: http://lwn.net/Articles/453002/
  - Or this more-recent deadlock: https://lkml.org/lkml/2012/7/2/163

The Linux Scheduler and RCU

- RCU uses the scheduler and the scheduler uses RCU
  - Plenty of opportunity for both RCU and the scheduler to lose big time!
  - See for example: http://lwn.net/Articles/453002/
  - Or this more-recent deadlock: https://lkml.org/lkml/2012/7/2/163
- But driving RCU's grace periods from a kthread should be OK

The Linux Scheduler and RCU

- RCU uses the scheduler and the scheduler uses RCU
  - Plenty of opportunity for both RCU and the scheduler to lose big time!
  - See for example: http://lwn.net/Articles/453002/
  - Or this more-recent deadlock: https://lkml.org/lkml/2012/7/2/163
- But driving RCU's grace periods from a kthread should be OK
  - As long as the scheduler doesn't wait for a grace period on any of its wake-up or context-switch fast paths

The Linux Scheduler and RCU

- RCU uses the scheduler and the scheduler uses RCU
  - Plenty of opportunity for both RCU and the scheduler to lose big time!
  - See for example: http://lwn.net/Articles/453002/
  - Or this more-recent deadlock: https://lkml.org/lkml/2012/7/2/163
- But driving RCU's grace periods from a kthread should be OK
  - As long as the scheduler doesn't wait for a grace period on any of its wake-up or context-switch fast paths: Either directly or indirectly

The Linux Scheduler and RCU

- RCU uses the scheduler and the scheduler uses RCU
  - Plenty of opportunity for both RCU and the scheduler to lose big time!
  - See for example: http://lwn.net/Articles/453002/
  - Or this more-recent deadlock: https://lkml.org/lkml/2012/7/2/163
- But driving RCU's grace periods from a kthread should be OK
  - As long as the scheduler doesn't wait for a grace period on any of its wake-up or context-switch fast paths: Either directly or indirectly
  - And as long as the scheduler doesn't exit an RCU read-side critical section while holding a runqueue or pi lock if that RCU read-side critical section had any chance of being preempted
- Driving RCU's graces periods kthread simplifies RCU:
  - dyntick-idle: No more stalls due to sleeping CPUs
  - force_quiescent_state(): no more races with grace-period completion

Conclusions

Conclusions

- They say that the best way to predict the future is to invent it

Conclusions

- They say that the best way to predict the future is to invent it
  - I am here to tell you that even this method is not foolproof

Conclusions

- They say that the best way to predict the future is to invent it
  - I am here to tell you that even this method is not foolproof
- SMP, real time, and energy efficiency are each well known
  - The real opportunities for new work involve combinations of them
- Some need for 10s-of-microseconds latency on 4096 CPUs
  - Translates to mainstream need on tens or hundreds of CPUs
    - Supporting this is not impossible

Conclusions

- They say that the best way to predict the future is to invent it
  - I am here to tell you that even this method is not foolproof
- SMP, real time, and energy efficiency are each well known
  - The real opportunities for new work involve combinations of them
- Some need for 10s-of-microseconds latency on 4096 CPUs
  - Translates to mainstream need on tens or hundreds of CPUs
    - Supporting this is not impossible: It will only require a little mind crushing ;-)

Conclusions

- They say that the best way to predict the future is to invent it
  - I am here to tell you that even this method is not foolproof
- SMP, real time, and energy efficiency are each well known
  - The real opportunities for new work involve combinations of them
- Some need for 10s-of-microseconds latency on 4096 CPUs
  - Translates to mainstream need on tens or hundreds of CPUs
    - Supporting this is not impossible: It will only require a little mind crushing ;-)
- There is still much work to be done on the Linux kernel
  - But even more work required for open-source applications
- The major large-system challenges are at the design level

Conclusions

- They say that the best way to predict the future is to invent it
  - I am here to tell you that even this method is not foolproof
- SMP, real time, and energy efficiency are each well known
  - The real opportunities for new work involve combinations of them
- Some need for 10s-of-microseconds latency on 4096 CPUs
  - Translates to mainstream need on tens or hundreds of CPUs
    - Supporting this is not impossible: It will only require a little mind crushing ;-)
- There is still much work to be done on the Linux kernel
  - But even more work required for open-source applications
- The major large-system challenges are at the design level
- Sometimes taking on crazy requirements simplifies things!!!

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?