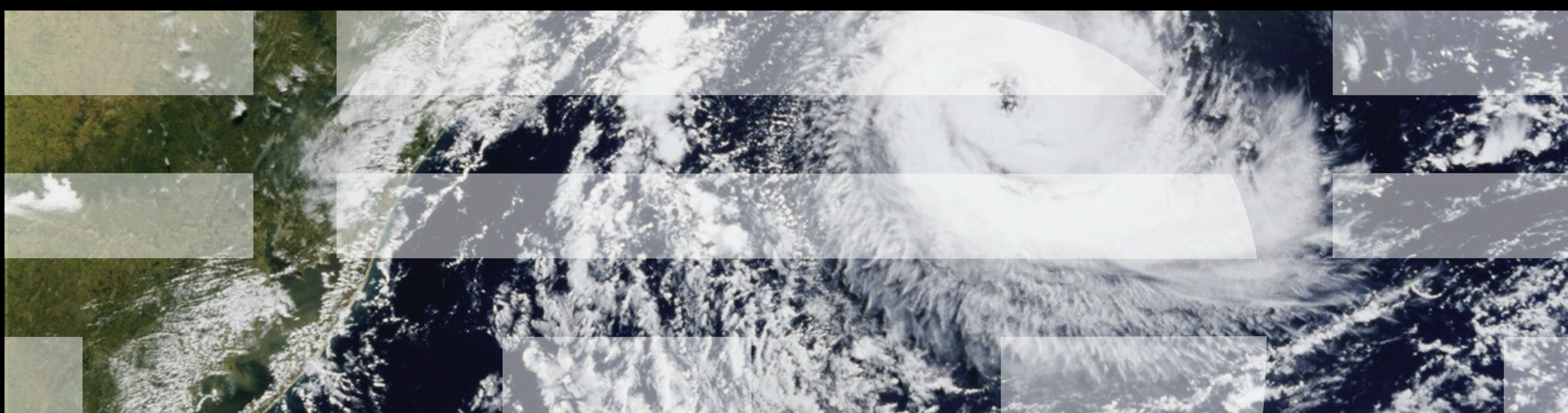
Paul E. McKenney, IBM Distinguished Engineer, Linux Technology Center
Member, IBM Academy of Technology
linux.conf.au, Geelong, Australia, February 3, 2016

What Happens When 4096 Cores All Do synchronize_rcu_expedited()?



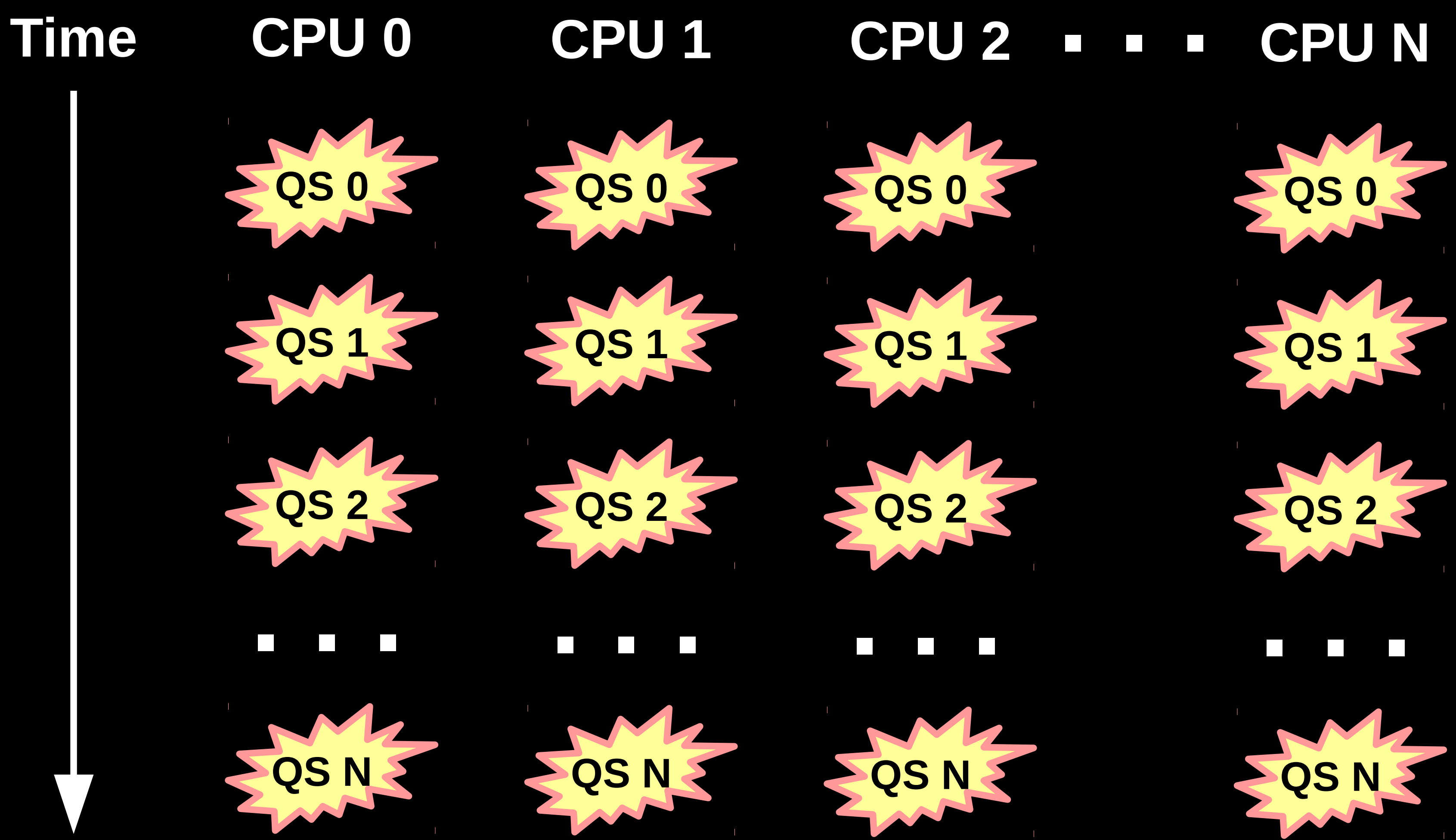
Overview

- What ***Should*** Happen When 4096 Cores All Do synchronize_rcu_expedited()?
- Overview of Algorithm for synchronize_rcu_expedited()
- Expedited Grace Period Example
- Benchmarking
- Benchmarking on 4096 CPUs
- Summary and Lessons (Re)learned

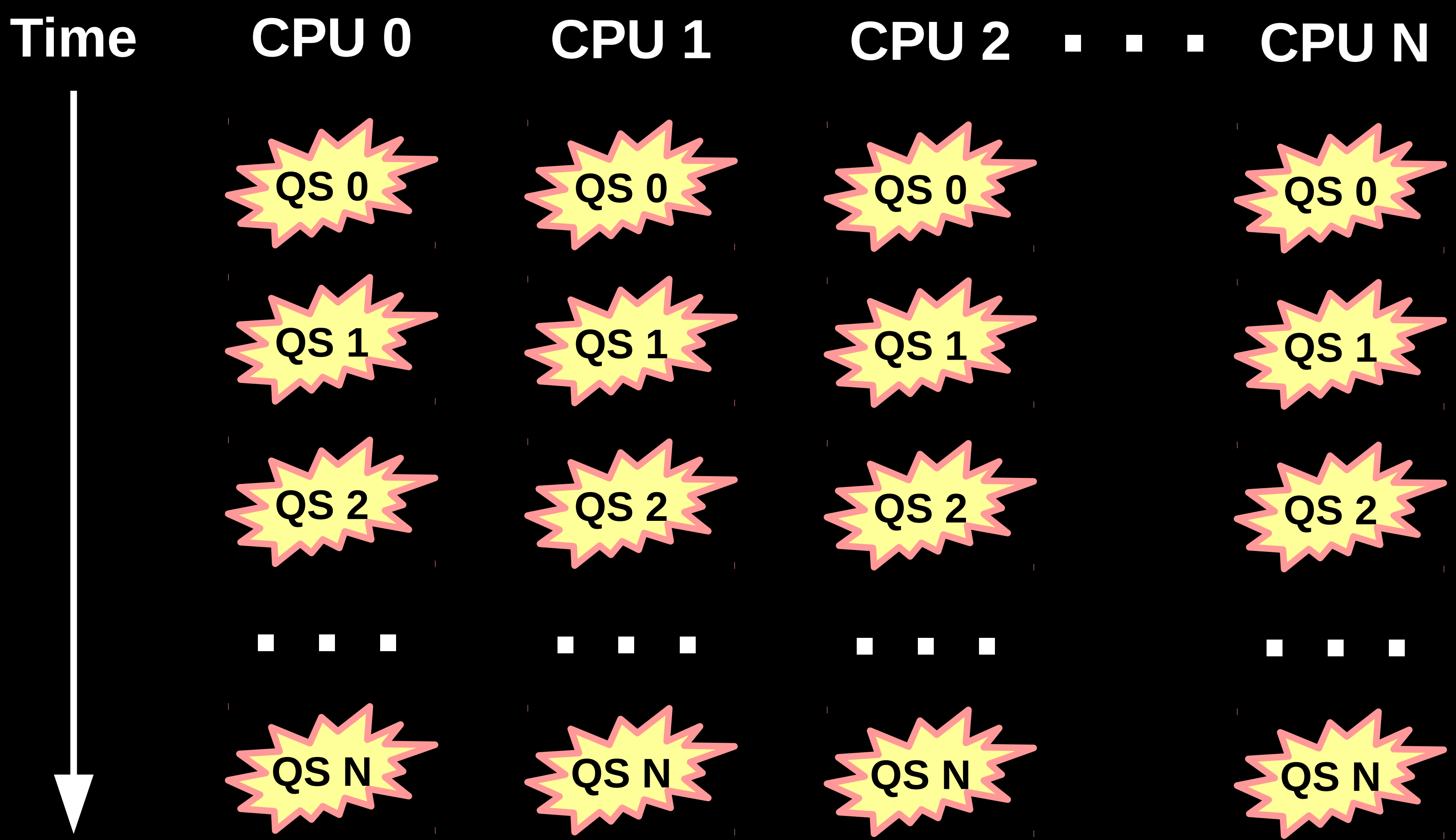
What *Should* Happen When 4096 Cores All Do synchronize_rcu_expedited()?

What Should *Not* Happen When 4096 Cores All Do synchronize_rcu_expedited()?

What Should *Not* Happen When 4096 Cores All Do synchronize_rcu_expedited()?



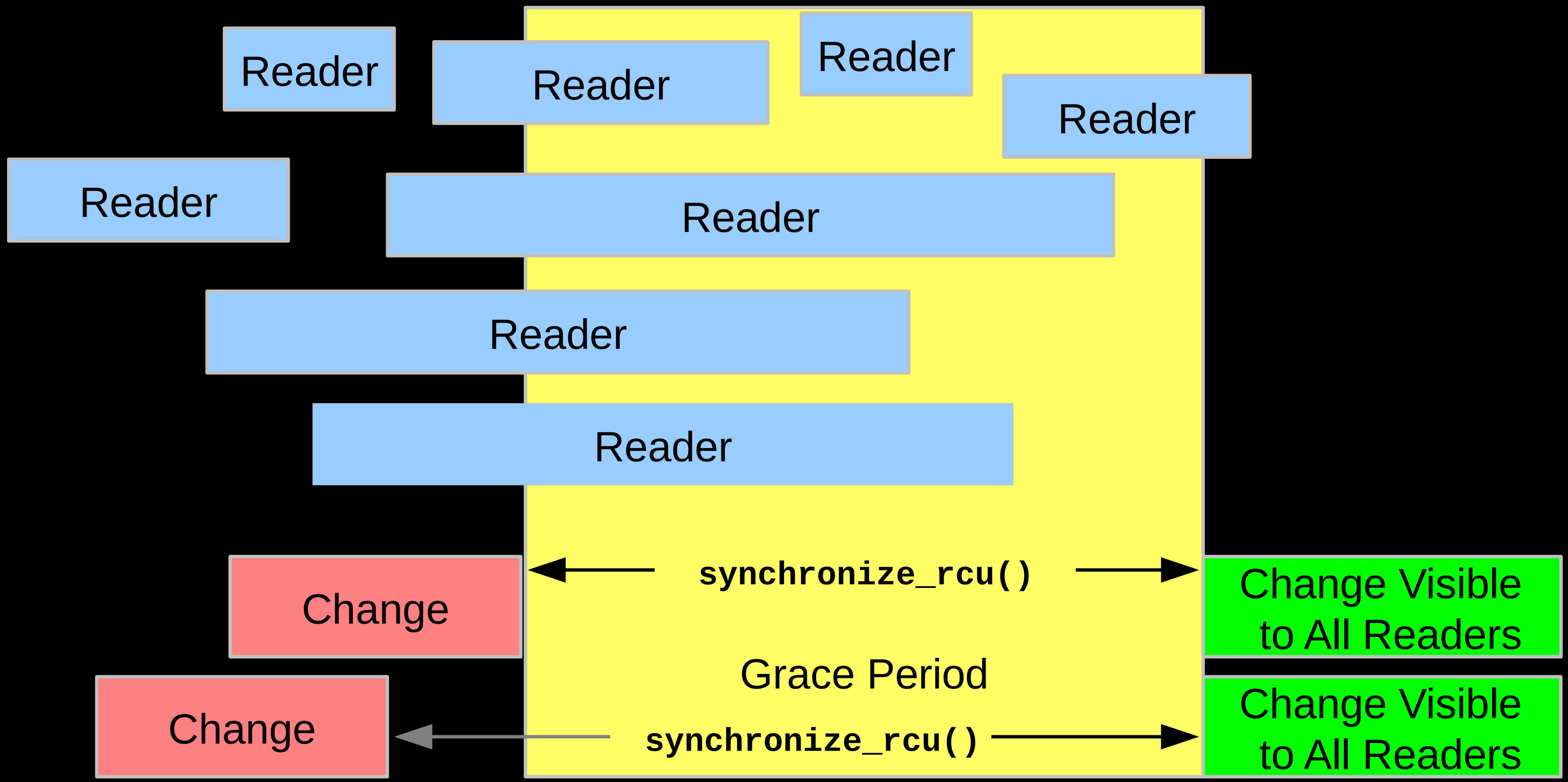
What Should *Not* Happen When 4096 Cores All Do synchronize_rcu_expedited()? Then What Instead?



RCU Grace Period Properties

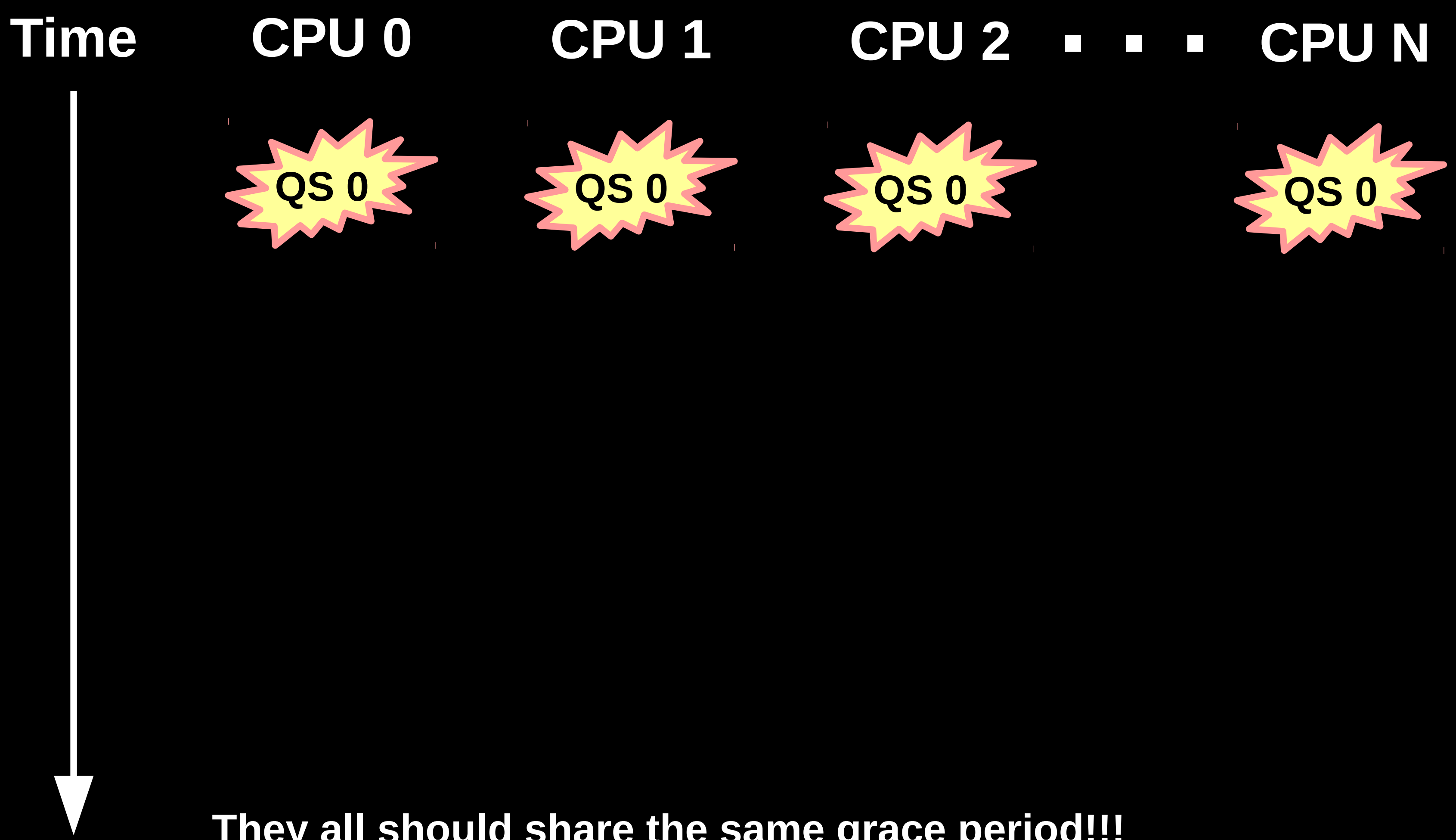
- *Grace Period:* Time during which every CPU/task spends some time outside of an RCU read-side critical section
  - Any critical section in progress at the beginning of a grace period must end before that grace period ends
    - RCU read-side critical section spans rcu_read_lock() to rcu_read_unlock()
    - RCU grace period wait: synchronize_rcu_expedited() and friends
- Grace periods are independent of CPU/task requesting them
- A single grace period can serve several requests
- In fact, single non-expedited grace periods often serve thousands of requests in Linux kernels

RCU Grace Period Properties Shown Graphically



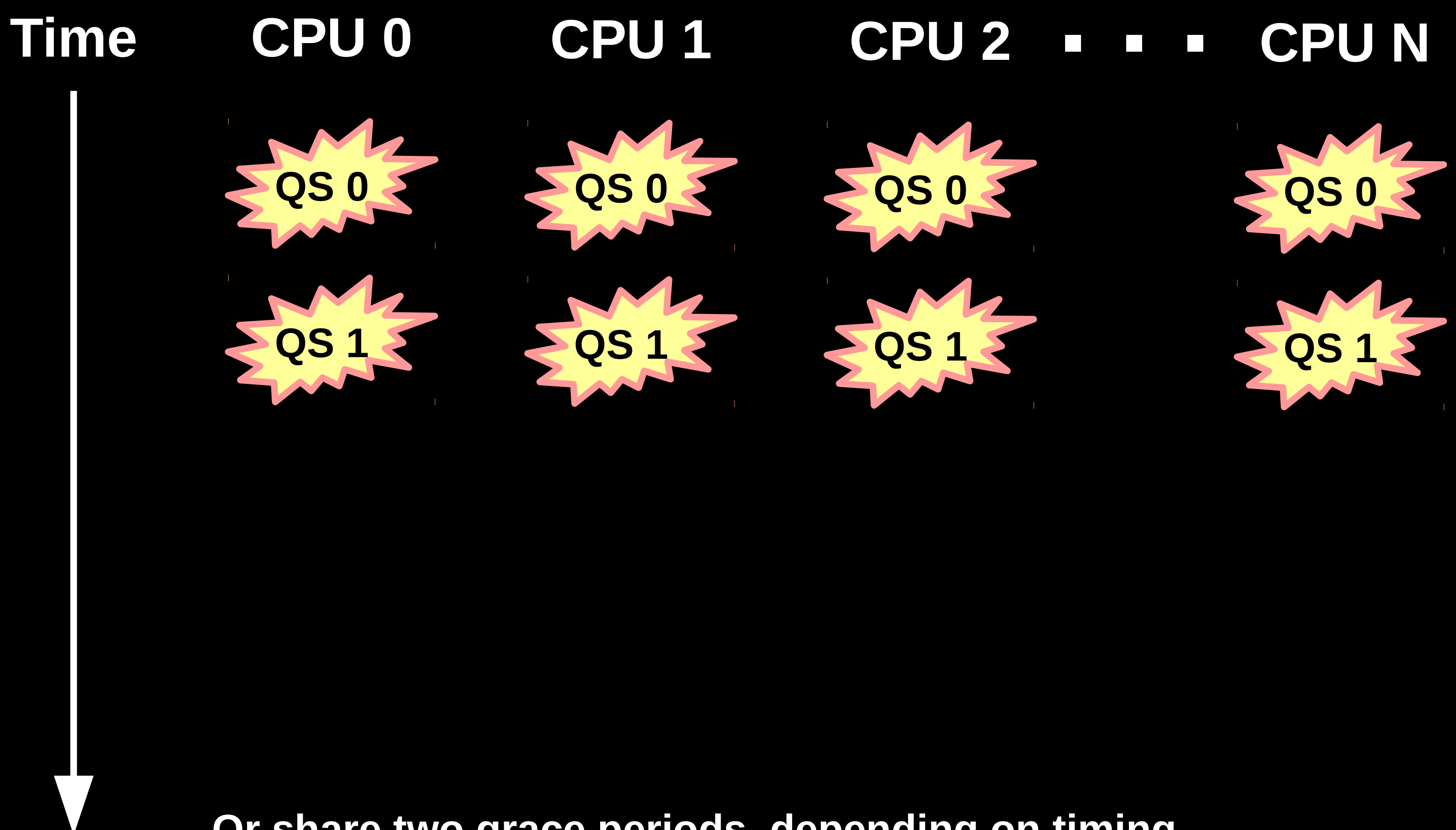
A grace period can serve multiple updates, decreasing the per-update RCU overhead.

What Should Happen Instead When 4096 Cores All Do synchronize_rcu_expedited()?



They all should share the same grace period!!!

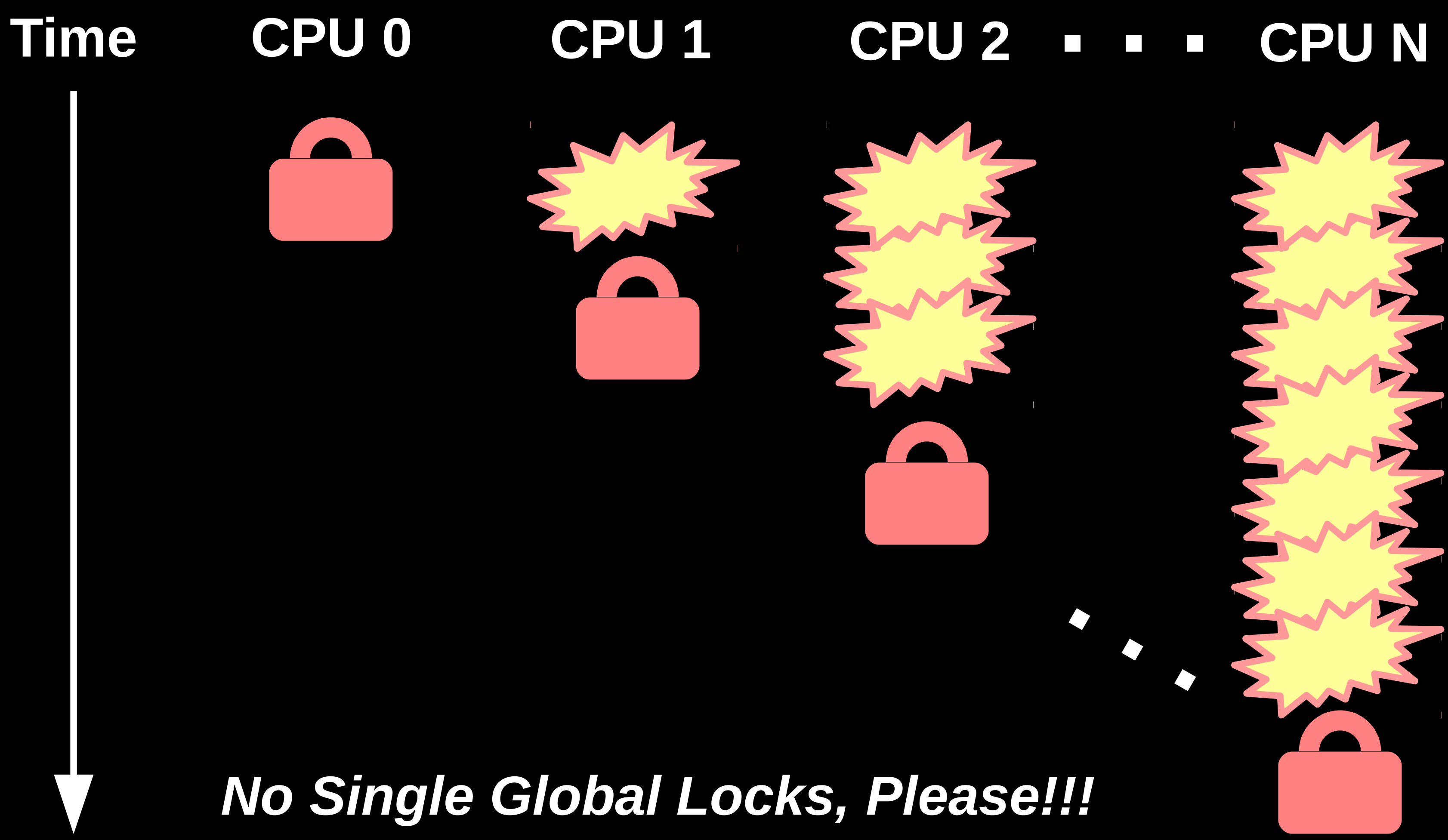
What Should Happen Instead When 4096 Cores All Do synchronize_rcu_expedited()? (Or Maybe This)



Or share two grace periods, depending on timing.

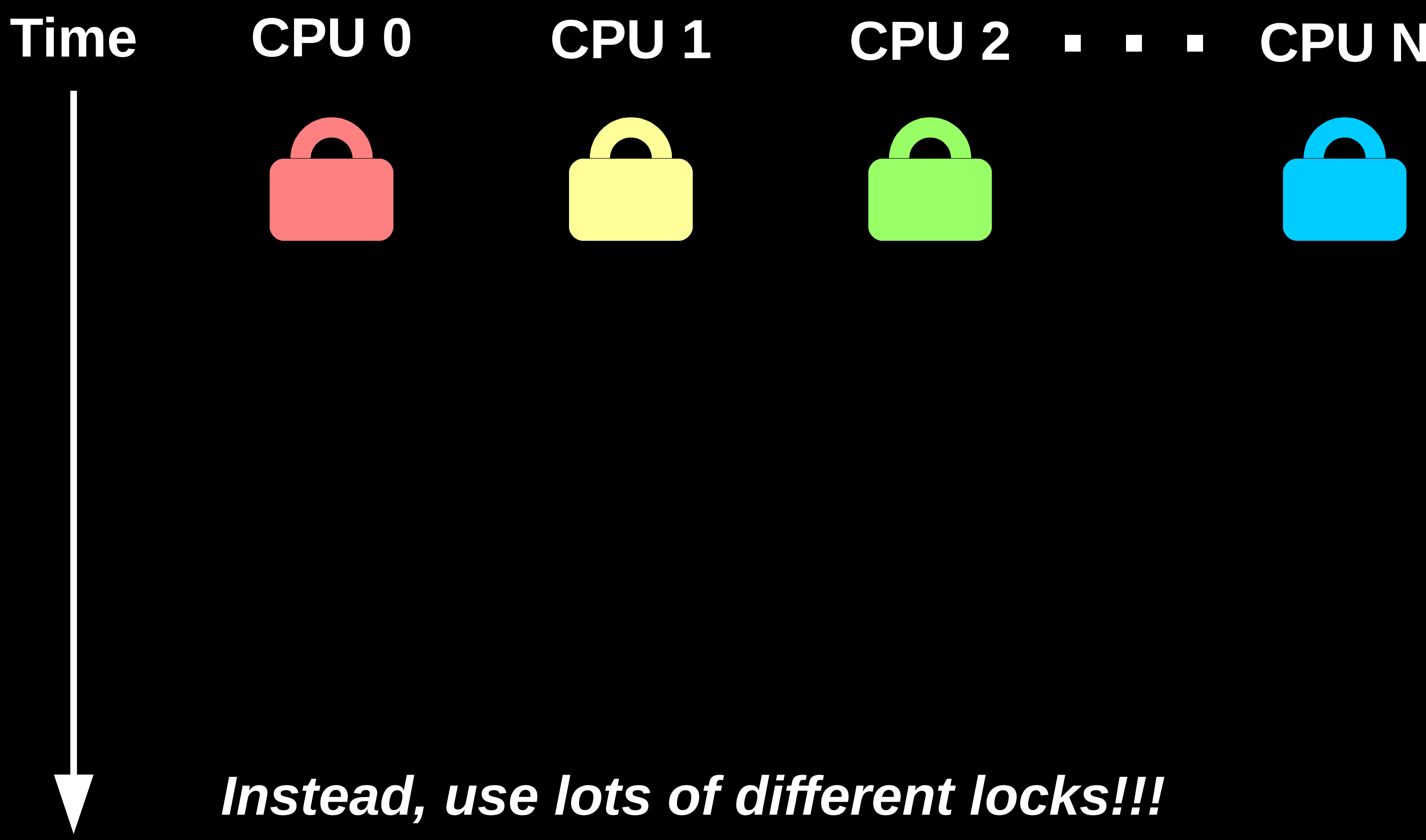
What Else Should *Not* Happen When 4096 Cores All Do synchronize_rcu_expedited()?

What Else Should *Not* Happen When 4096 Cores All Do synchronize_rcu_expedited()?



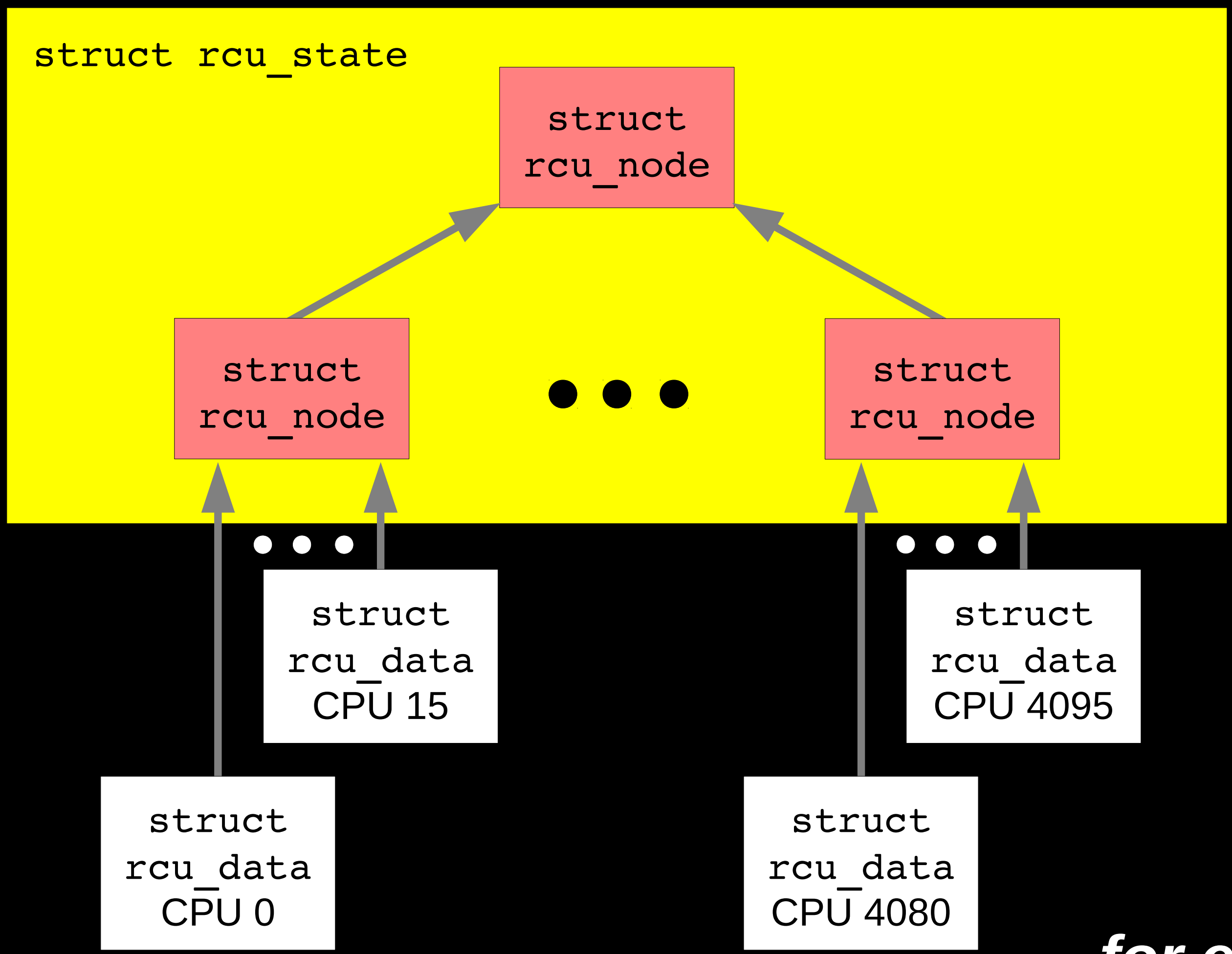
No Single Global Locks, Please!!!

What Should Happen Instead When 4096 Cores All Do synchronize_rcu_expedited()?



Instead, use lots of different locks!!!

Tree RCU's rcu_node Combining Tree to the Rescue!



Level 0: 1 rcu_node

Level 1: 4 rcu_nodes

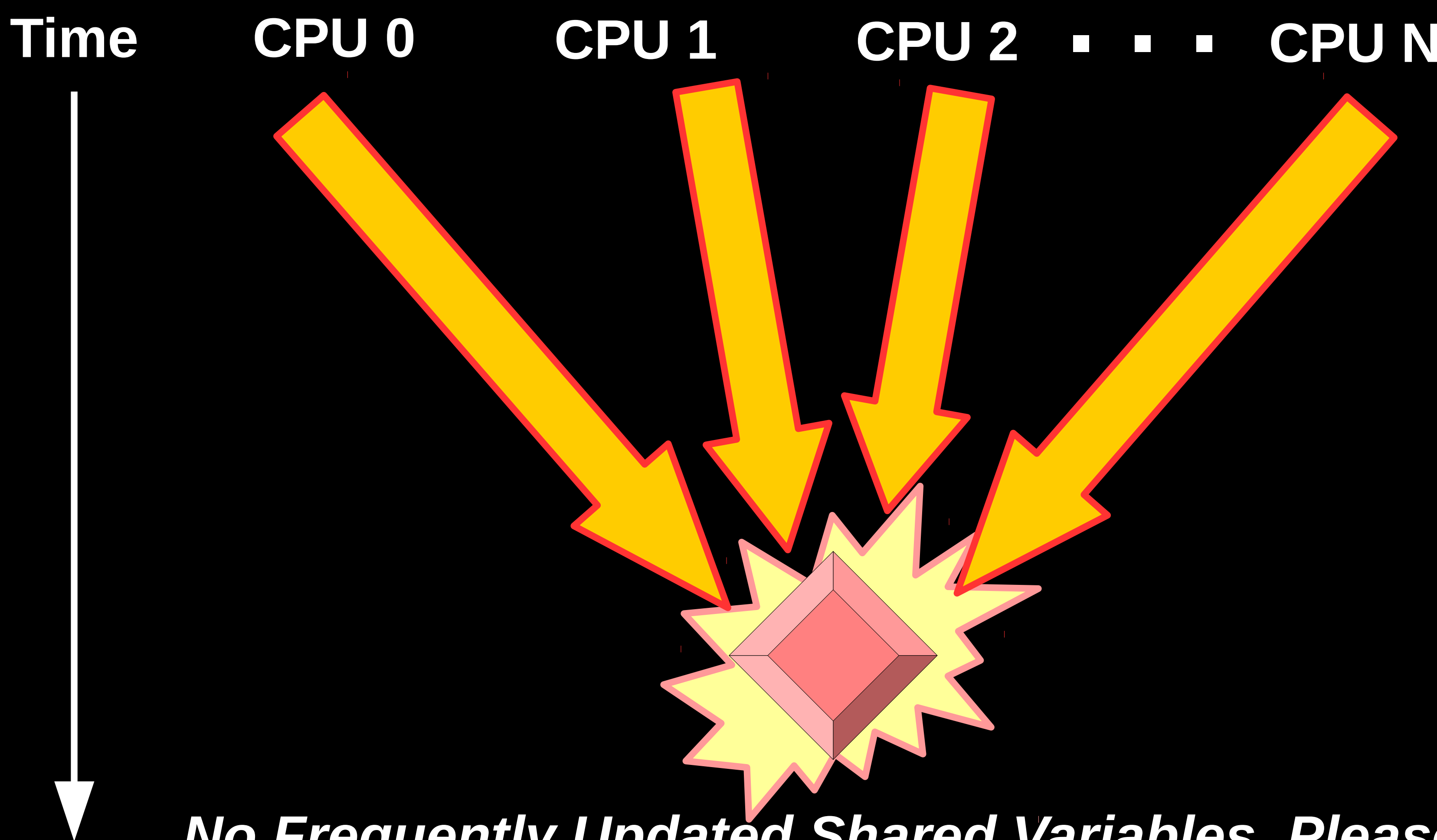
Level 2: 256 rcu_nodes

Total: 261 rcu_nodes

Separate locks for each instance!!!

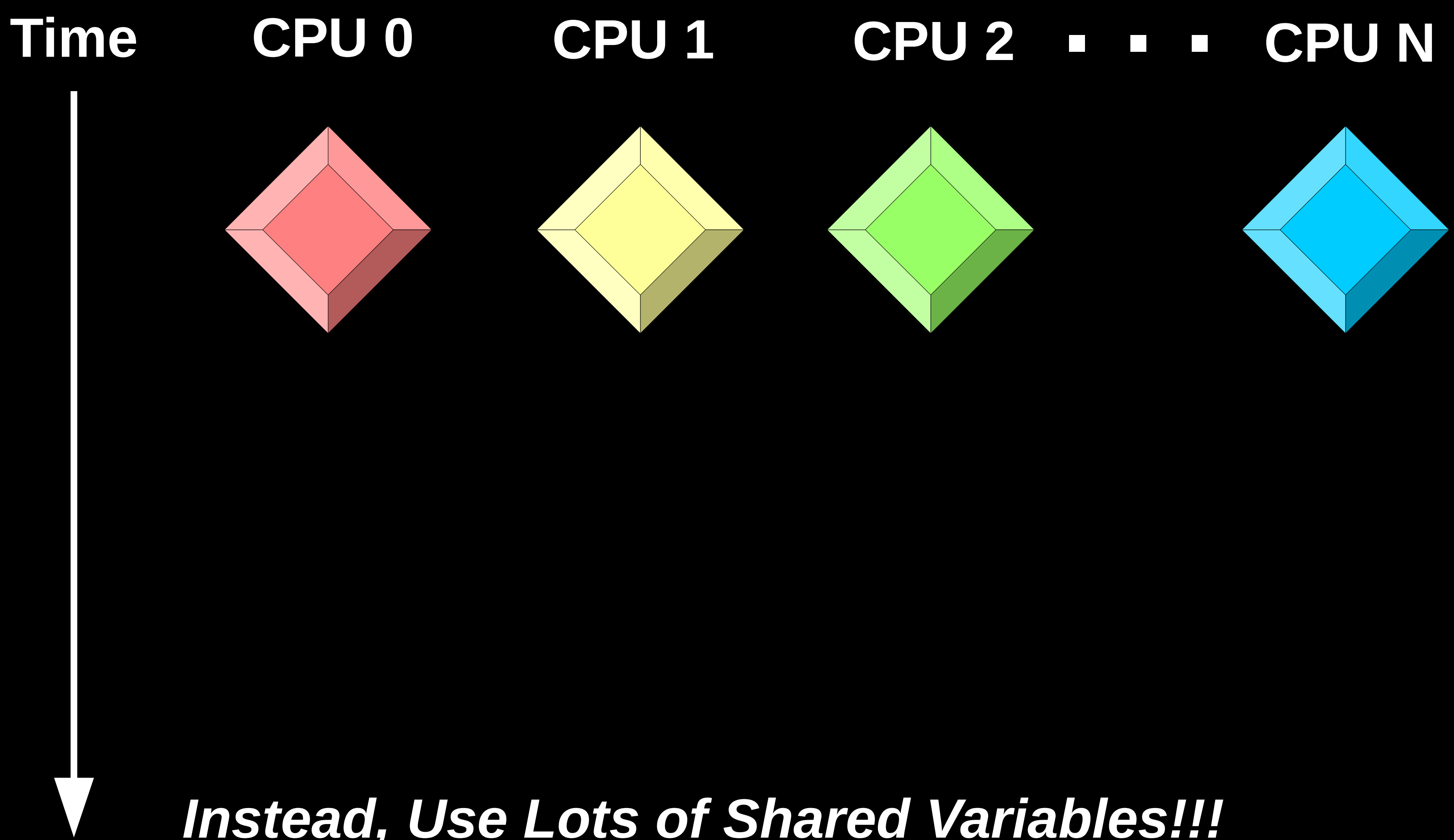
What Else Should *Not* Happen When 4096 Cores All Do synchronize_rcu_expedited()?

What Else Should *Not* Happen When 4096 Cores All Do synchronize_rcu_expedited()?



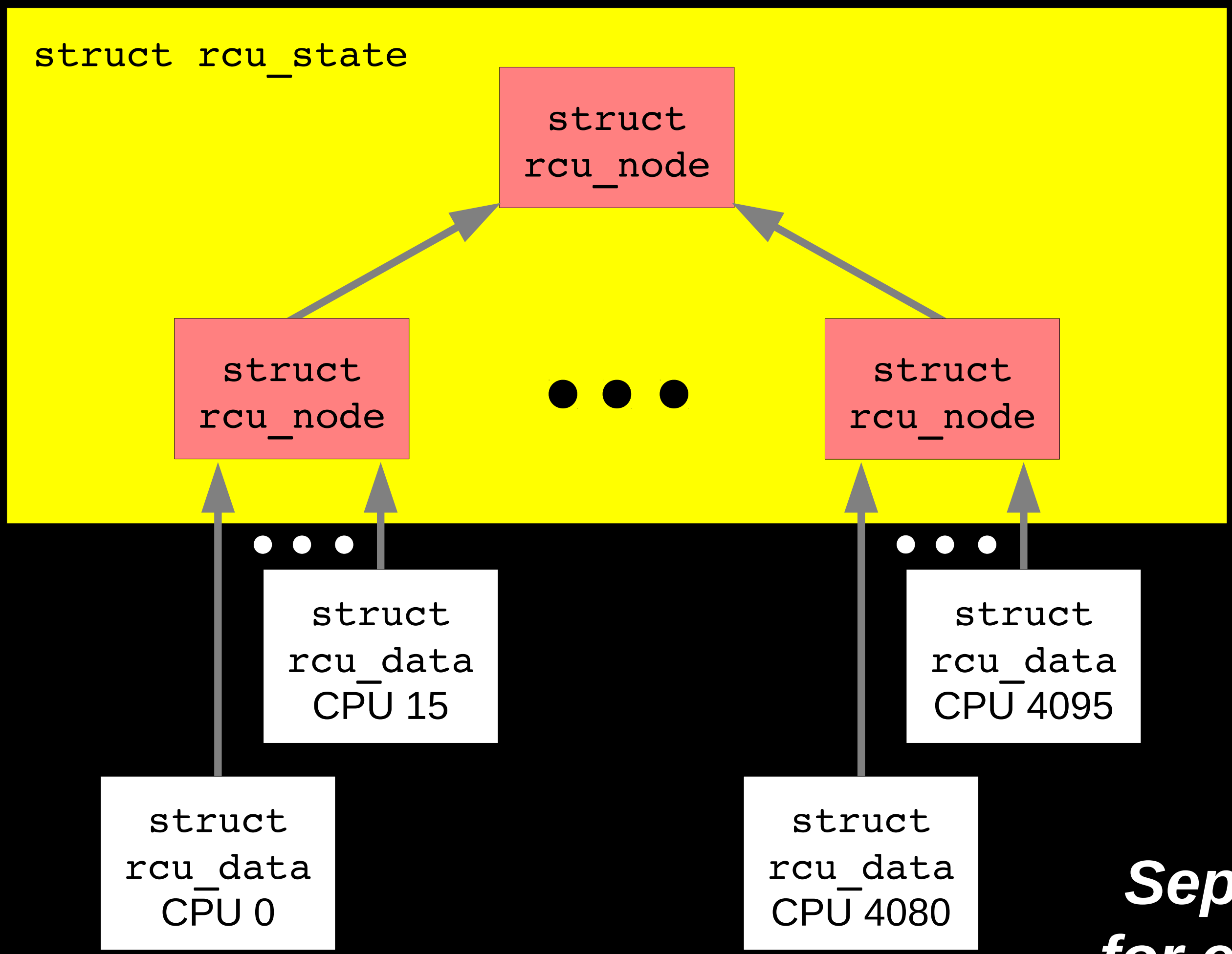
No Frequently Updated Shared Variables, Please!!!

What Should Happen Instead When 4096 Cores All Do synchronize_rcu_expedited()?



Instead, Use Lots of Shared Variables!!!

Again, Tree RCU's Combining Tree to the Rescue!



Level 0: 1 rcu_node

Level 1: 4 rcu_nodes

Level 2: 256 rcu_nodes

Total: 261 rcu_nodes

Separate variables for each instance!!!

Non-Requirements

- Real-time response for synchronize_rcu_expedited()
  - Must wait for readers in any case
  - RCU priority boosting carried out, but more diagnostic than realtime
  - So some variation in timings is to be expected
- Constant synchronize_rcu_expedited() latency
  - After all, synchronize_rcu() latency increases with number of CPUs
- Big-system performance of synchronize_rcu_expedited() to the exclusion of all else
  - Heavy update workloads better served by synchronize_rcu()

Overall synchronize_rcu_expedited() Algorithm

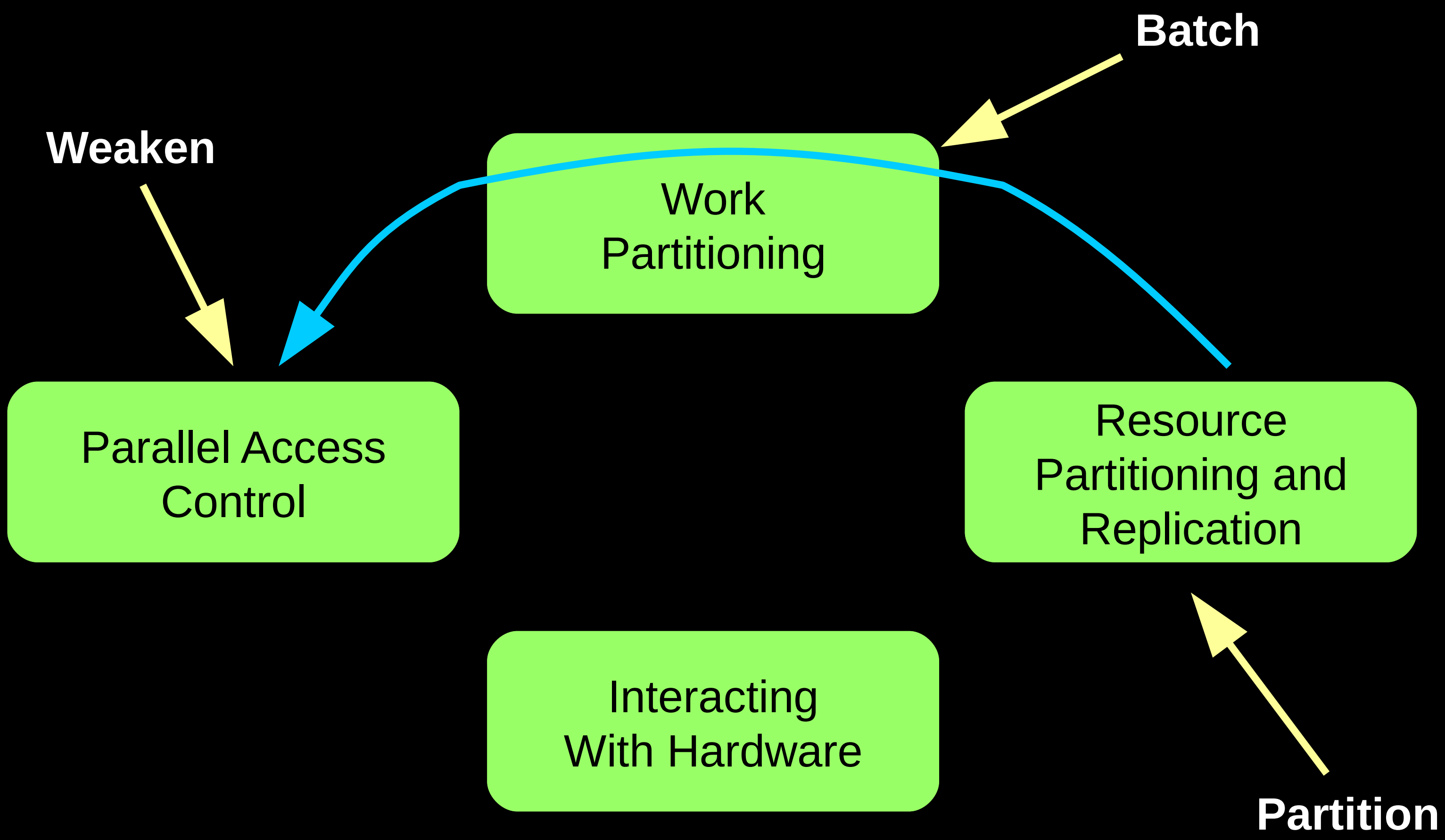
High-Level synchronize_rcu_expedited() Algorithm

- For each non-idle online CPU:
  - Send IPI
  - Handler determines if CPU is in quiescent state (context switch, user-mode execution, idle, cond_resched_rcu_qs()...)
  - If so, report quiescent state
  - If not, set CPU-local state so that next quiescent-state entry is reported
- When all non-idle online CPUs has reported a quiescent state, grace period is complete

High-Level synchronize_rcu_expedited() Algorithm

- For each non-idle online CPU:
  - Send IPI
  - Handler determines if CPU is in quiescent state (context switch, user-mode execution, idle, cond_resched_rcu_qs()...)
  - If so, report quiescent state
  - If not, set CPU-local state so that next quiescent-state entry is reported
- When all non-idle online CPUs has reported a quiescent state, grace period is complete
- The trick is doing this without bottlenecks...

Overall Approach to Concurrent-Code Optimization



Optimize Expedited Grace Periods

- Partition
  - Use the rcu_node combining tree!

Optimize Expedited Grace Periods

- Partition
  - Use the rcu_node combining tree!
- Batch
  - Need a mechanism to piggyback off others' expedited grace periods

Optimize Expedited Grace Periods

- Partition
  - Use the rcu_node combining tree!
- Batch
  - Need a mechanism to piggyback off others' expedited grace periods
- Weaken
  - My normal advice would be to use RCU, but this ***is*** RCU...

Optimize Expedited Grace Periods

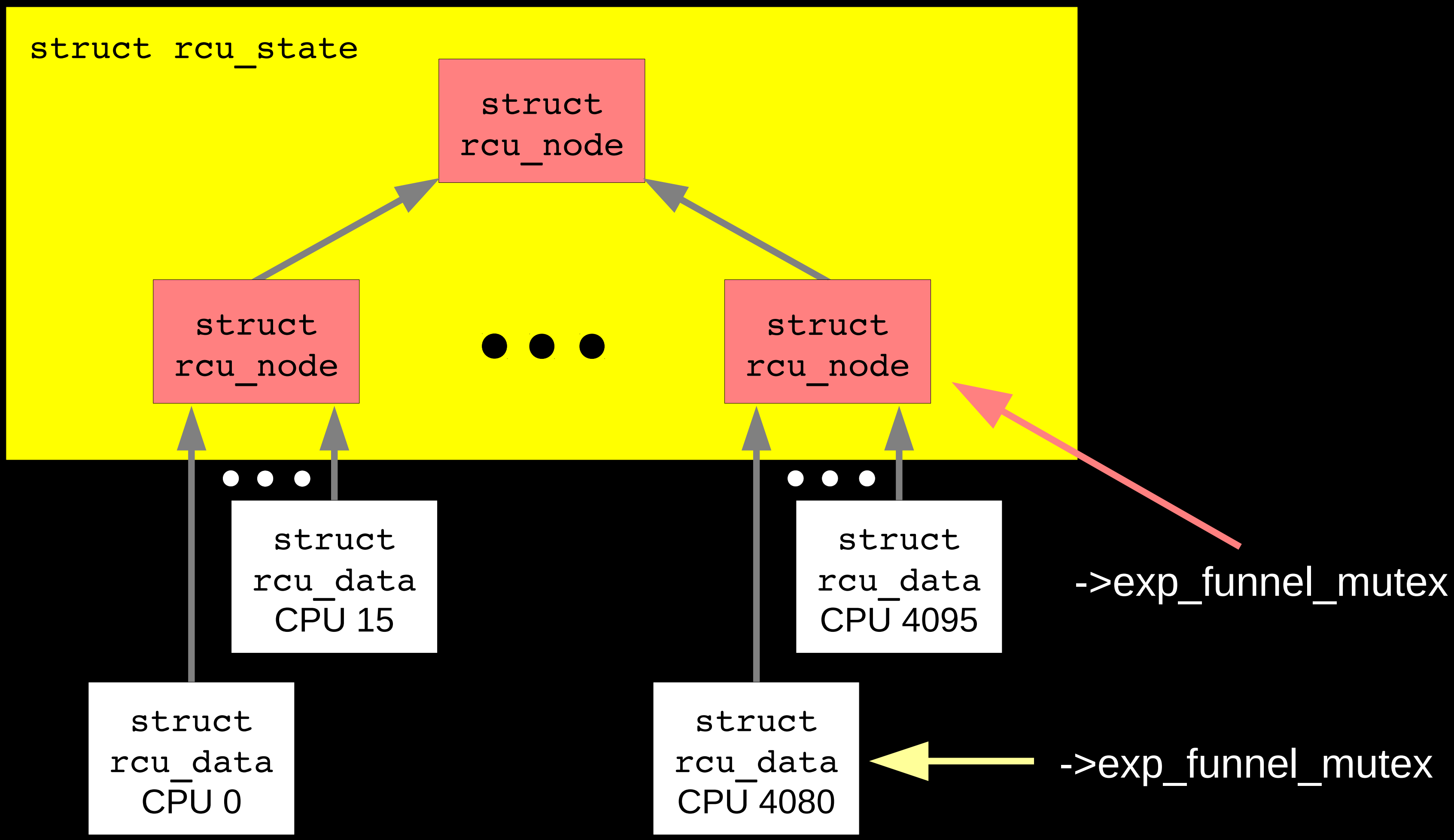
- Partition
  - Use the rcu_node combining tree!
- Batch
  - Need a mechanism to piggyback off others' expedited grace periods
- Weaken
  - My normal advice would be to use RCU, but this ***is*** RCU...
- Hardware
  - Need to be portable, so no FPGAs or GPGPUs for the time being...

Optimize Expedited Grace Periods

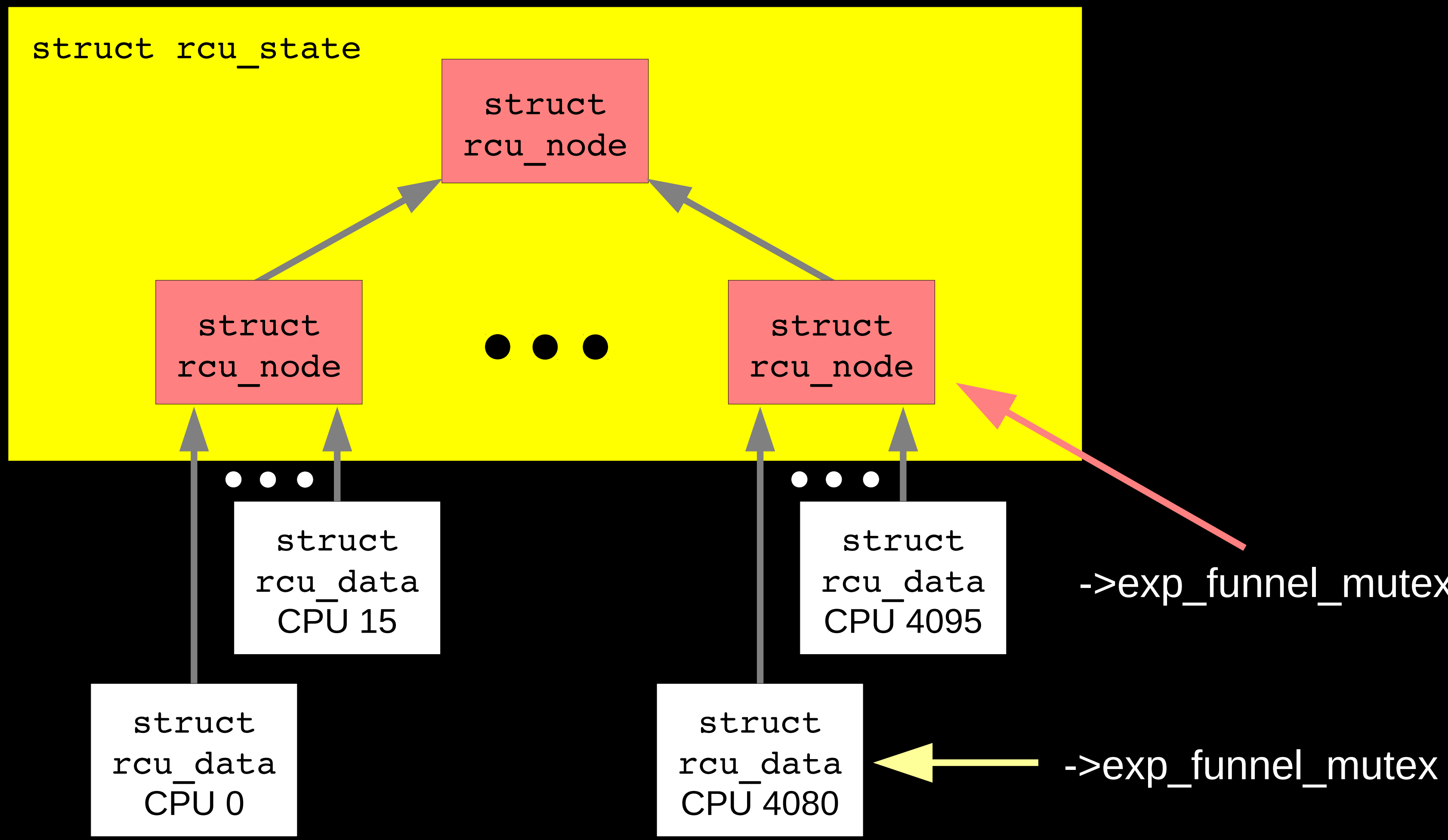
- Partition
  - Use the rcu_node combining tree!
- Batch
  - Need a mechanism to piggyback off others' expedited grace periods
- Weaken
  - My normal advice would be to use RCU, but this ***is*** RCU...
- Hardware
  - Need to be portable, so no FPGAs or GPGPUs for the time being...
- We therefore stick with partitioning and batching

Partitioning Expedited Grace Periods

Partitioning Expedited Grace Periods



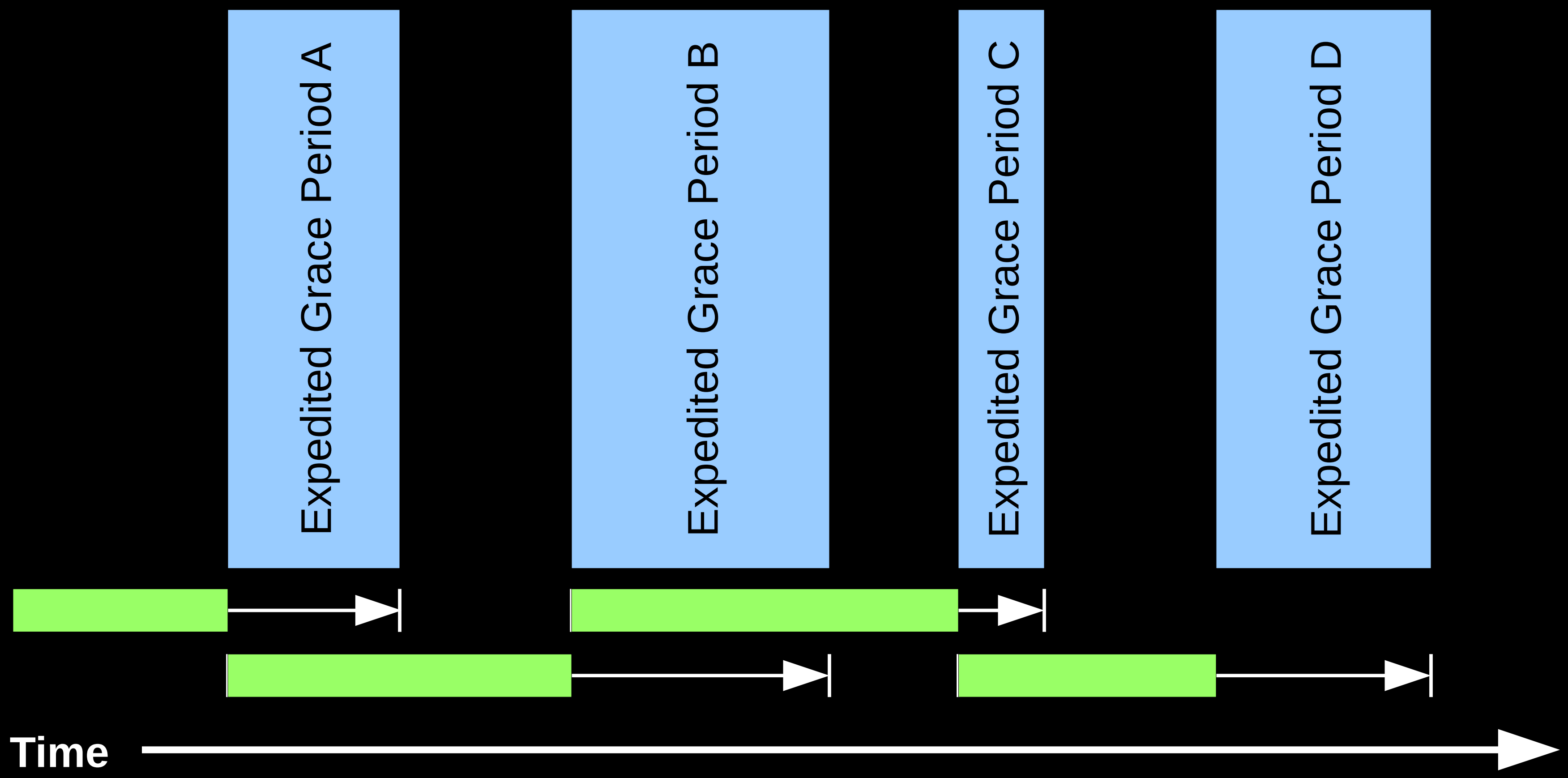
Partitioning Expedited Grace Periods



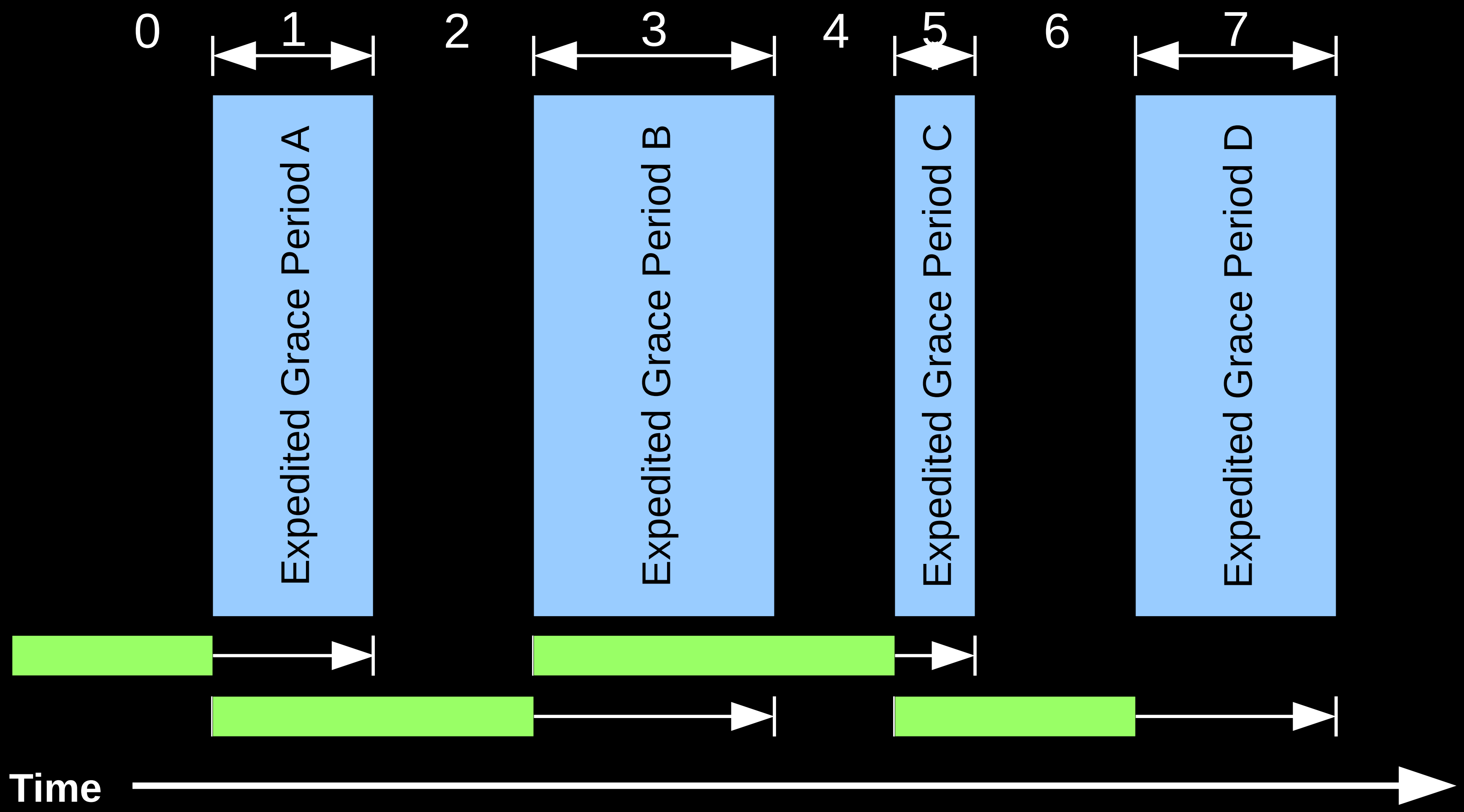
But we still have lock-contention bottleneck at root rcu_node structure!!!

Batching Expedited Grace Periods

Batching Expedited Grace Periods



Batching Expedited Grace Periods: Numbering



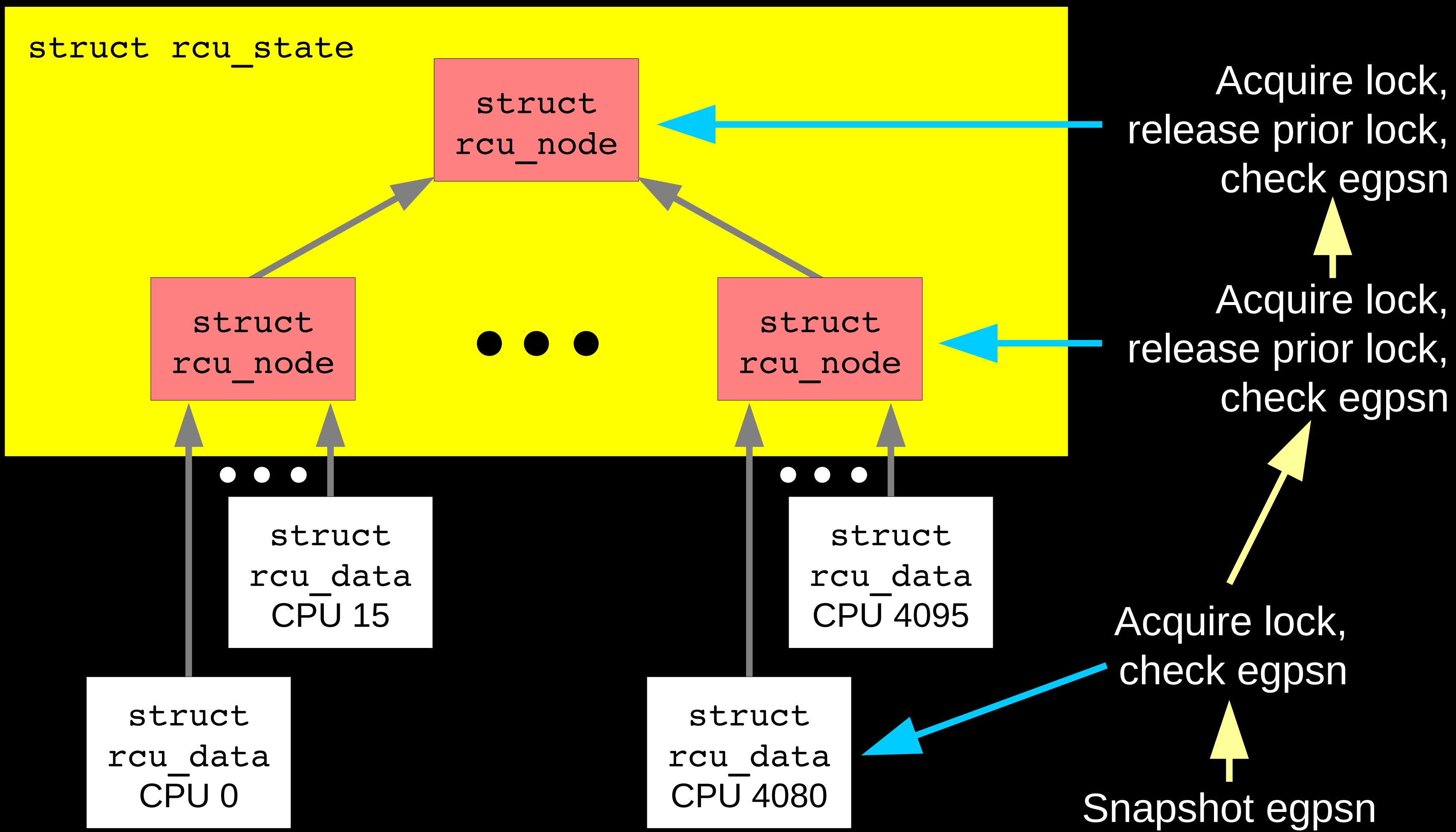
Batching Expedited Grace Periods: Using Numbering

- Start at zero, wait until two
- Start at one, wait until four
- Start at two, wait until four
- Start at three, wait until six
- Start at four, wait until six
- Start at five, wait until eight
- Start at six, wait until eight
- General rule: wait = (start + 3) & ~0x1

Batching Expedited Grace Periods: Using Numbering

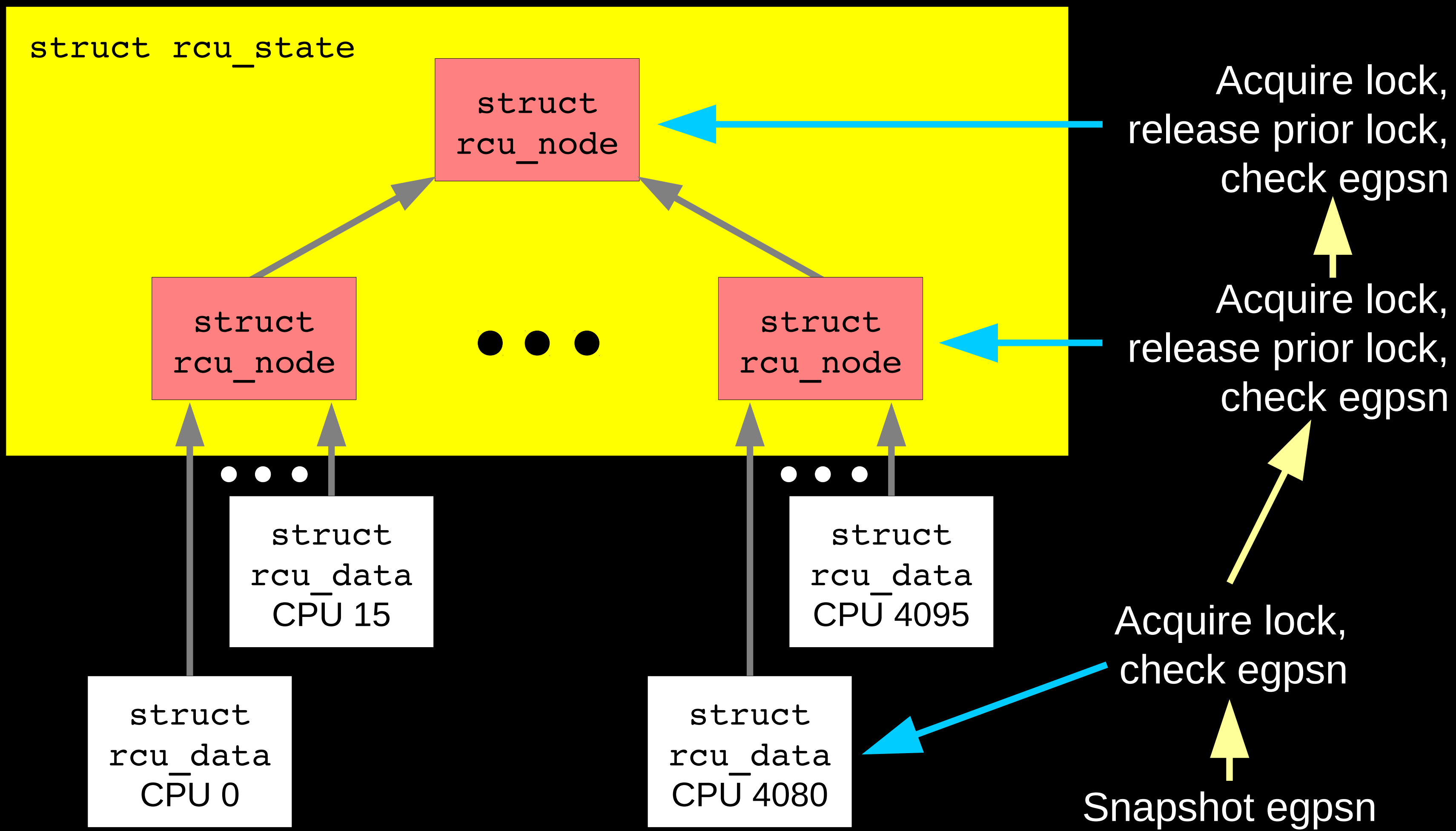
- Snapshot expedited grace-period sequence number (egpsn)
  - Add three and clear low-order bit
- Acquire locks to start grace period
  - If egpsn has reached snapshot, done!
  - Release locks and exit
- Increment egpsn
- Start expedited grace period
- Wait for expedited grace period to complete
- Increment egpsn

Batching Expedited Grace Periods: Using Numbering



IBM

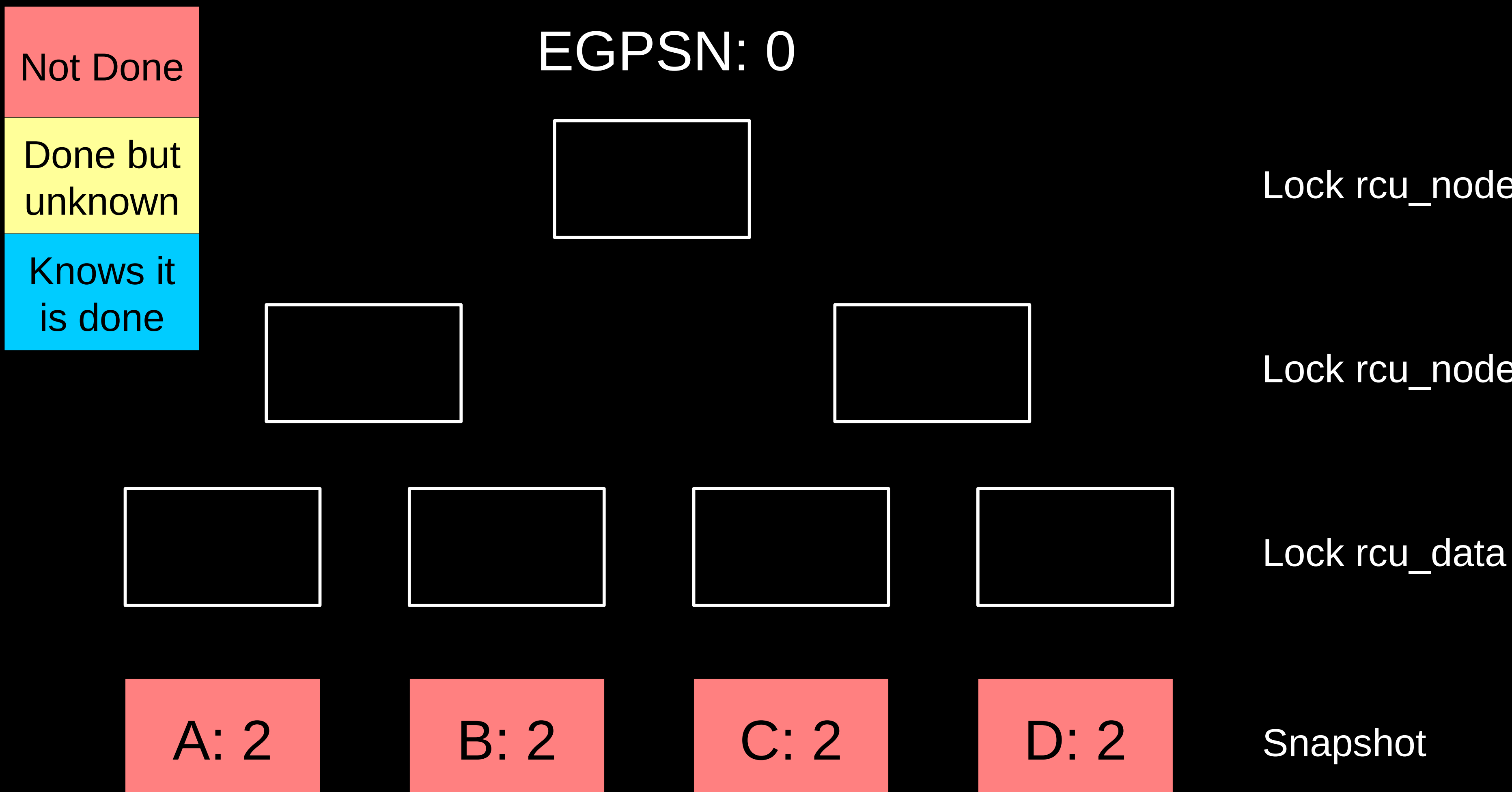
Batching Expedited Grace Periods: Using Numbering



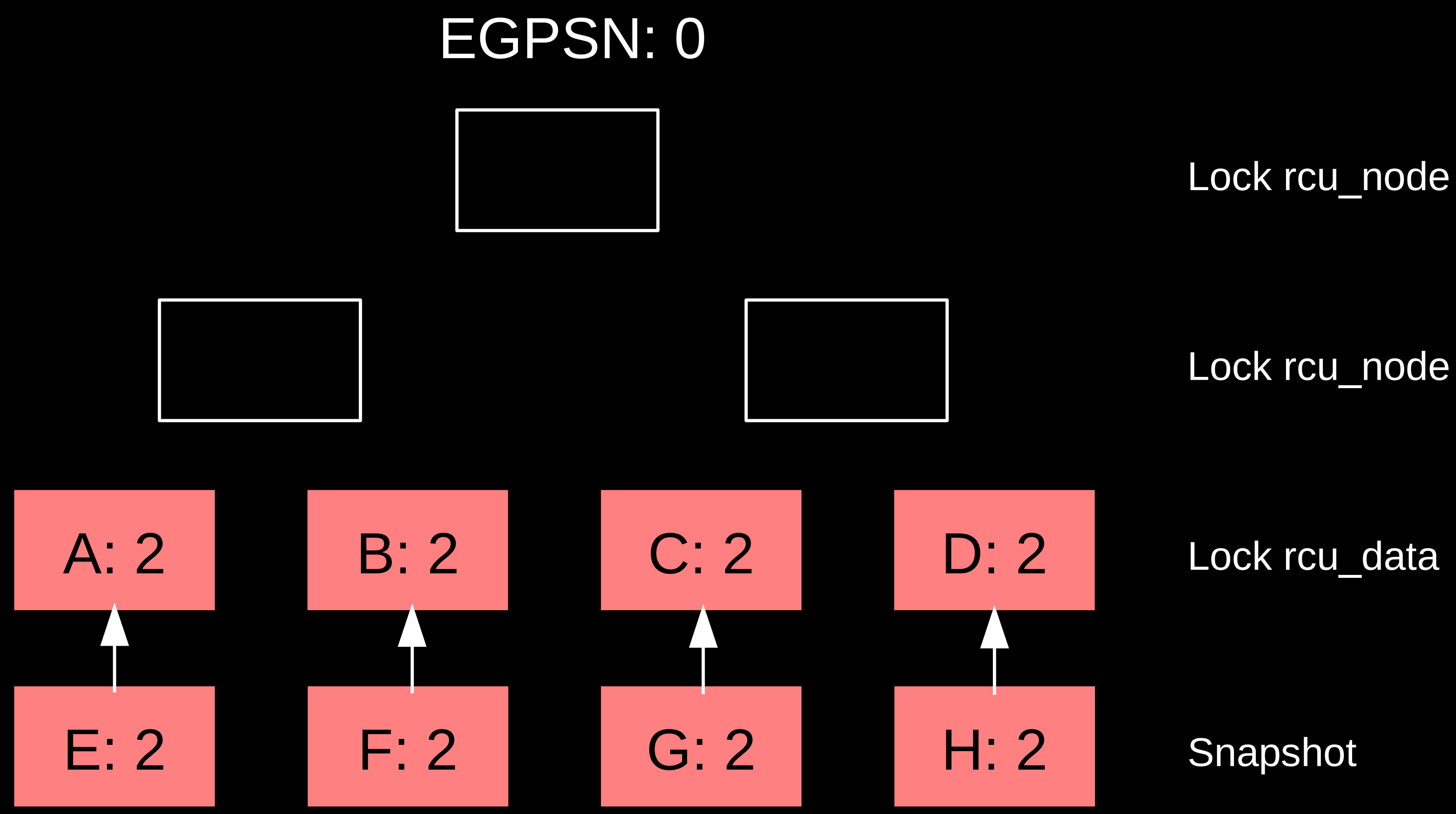
Optimization: Try acquiring root-level lock first, fall back if unavailable

Expedited Grace Period Example

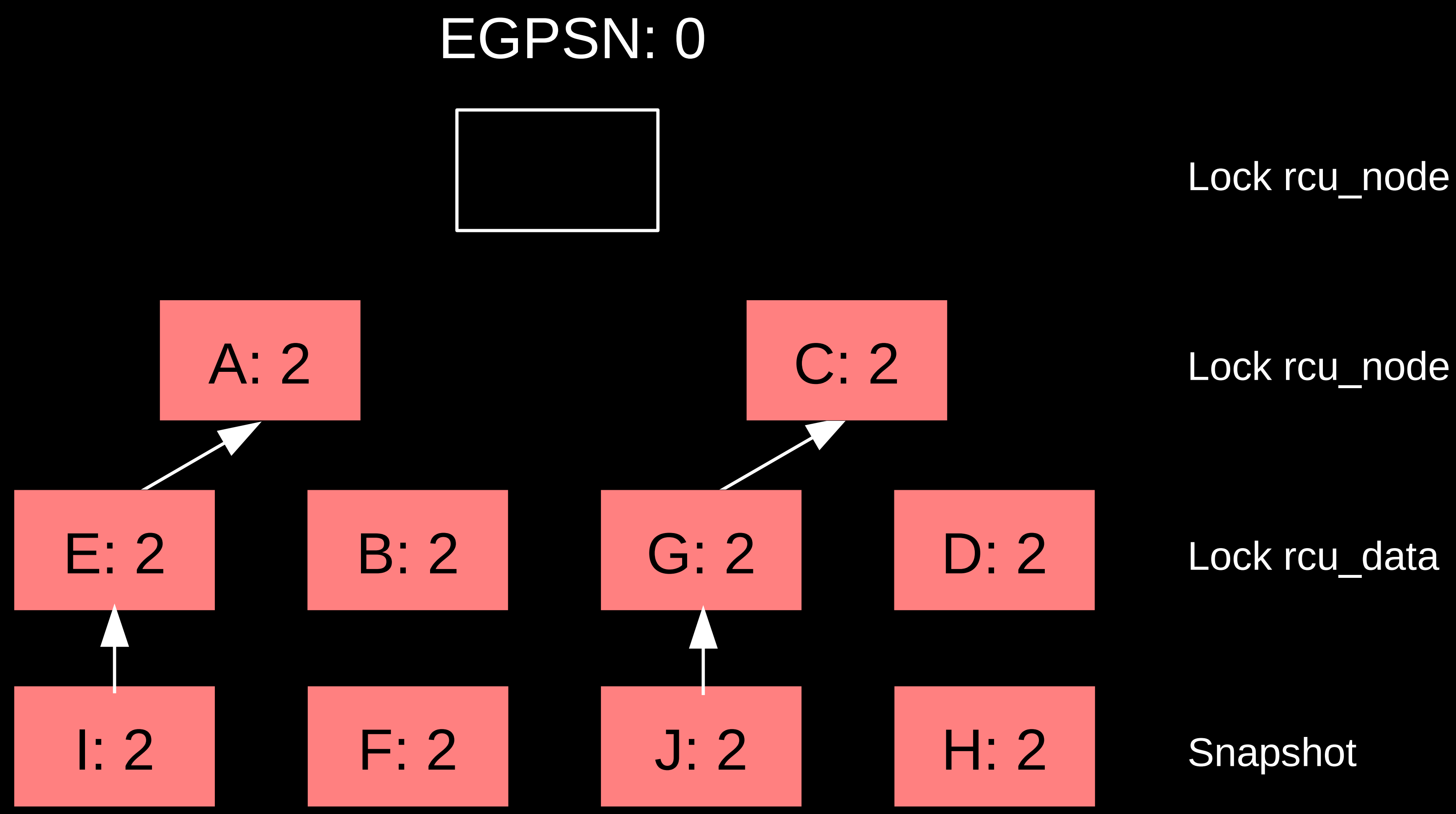
Expedited Grace Period Example



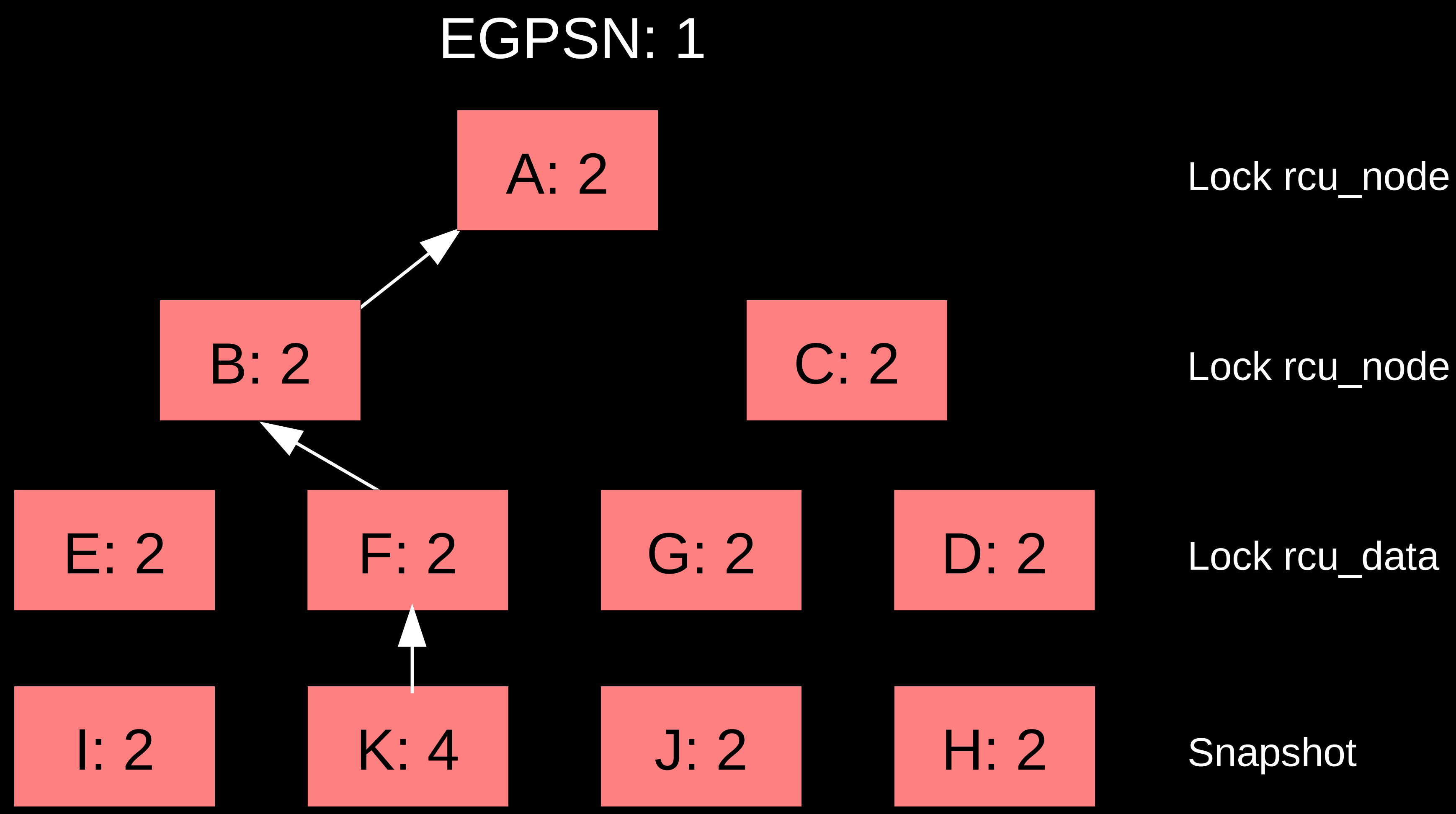
Expedited Grace Period Example



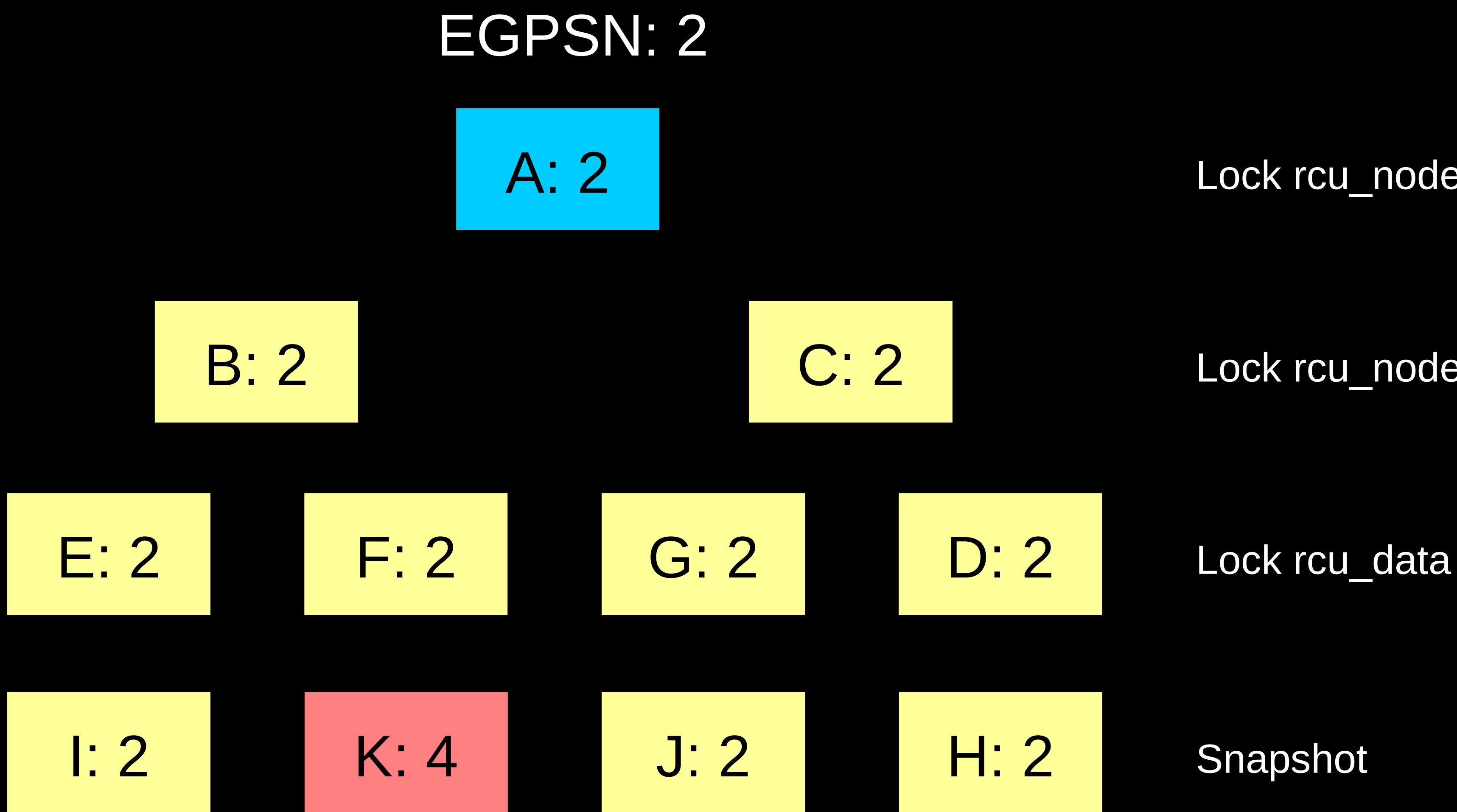
Expedited Grace Period Example



Expedited Grace Period Example

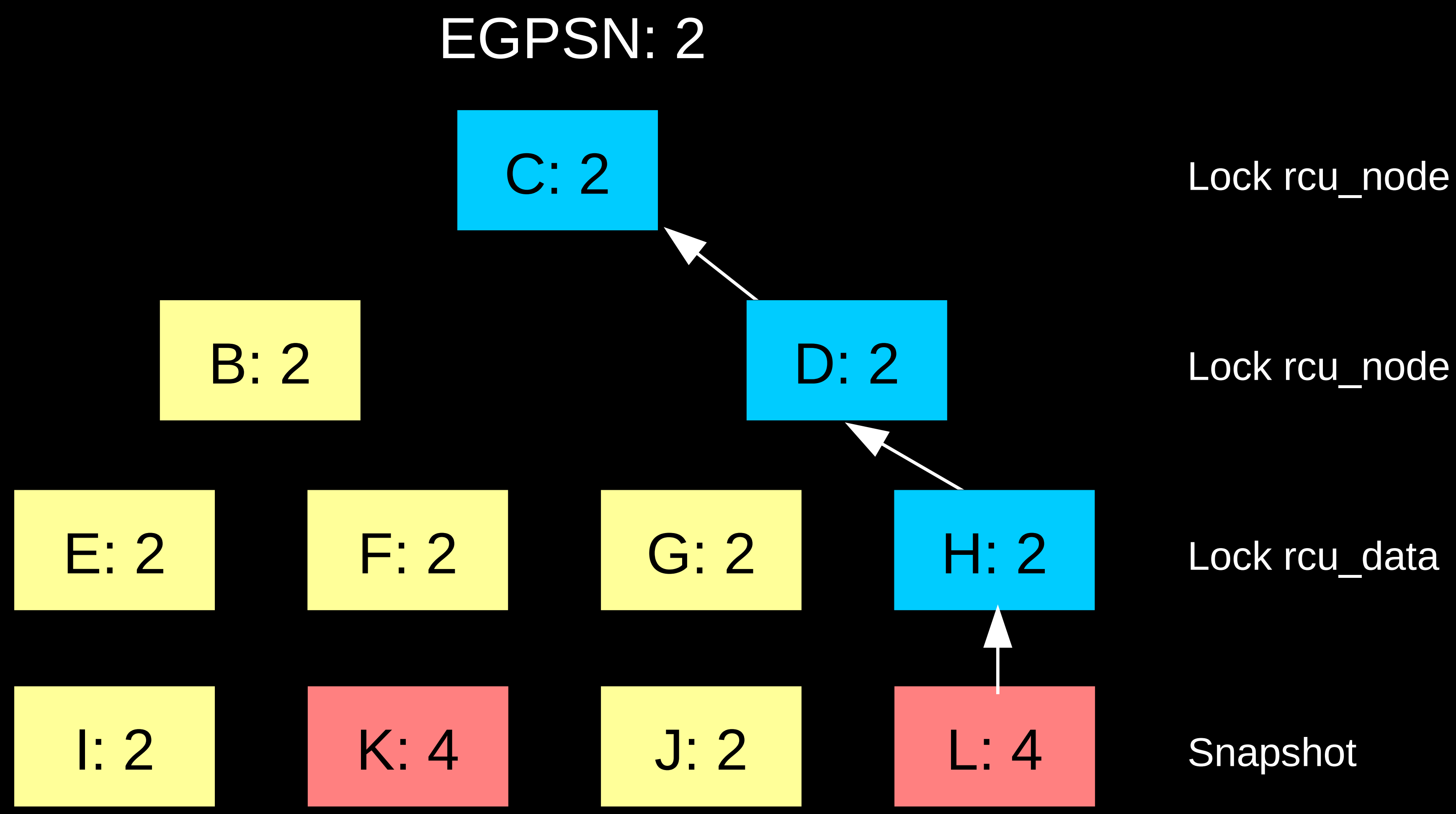


Expedited Grace Period Example

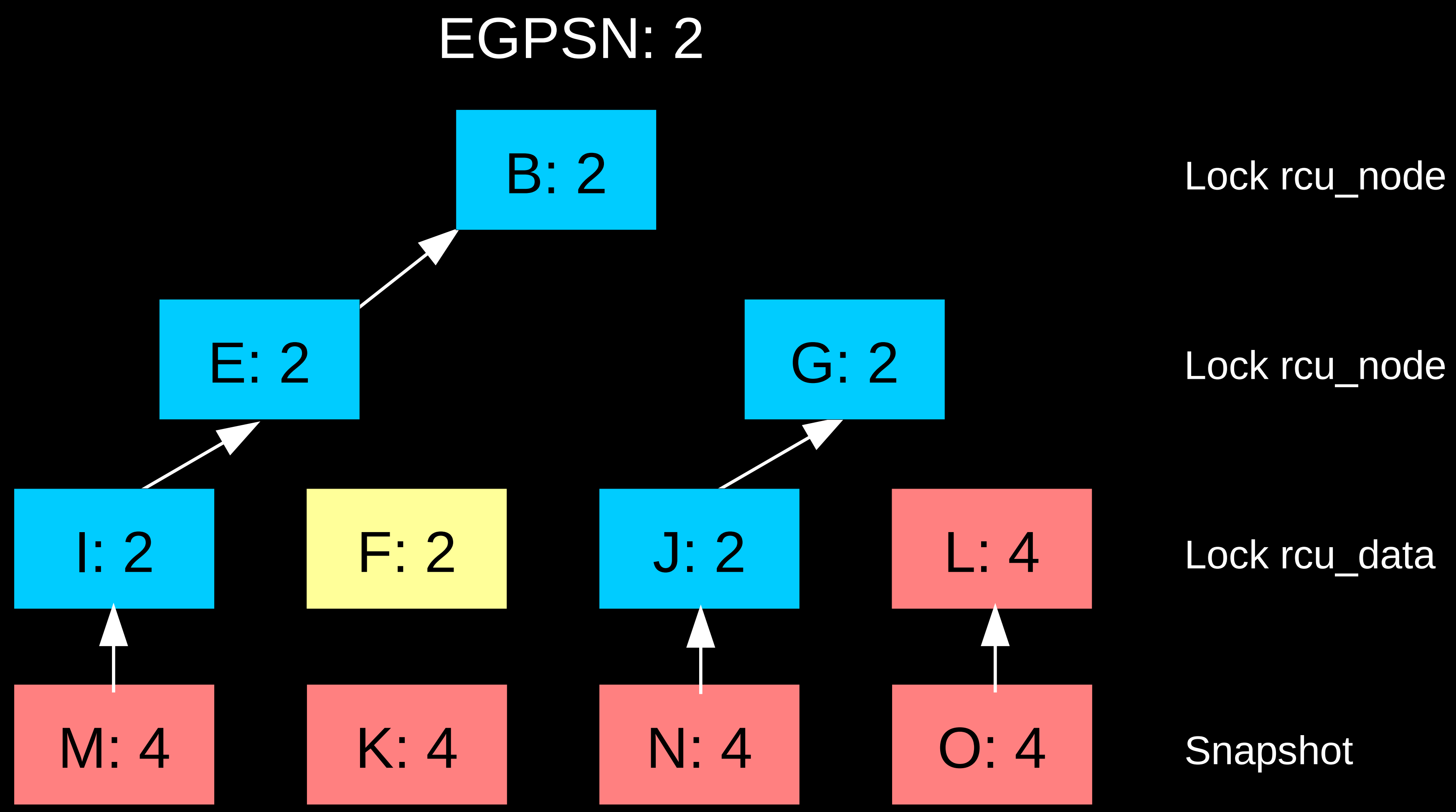


One expedited grace period serves ten requests!!!

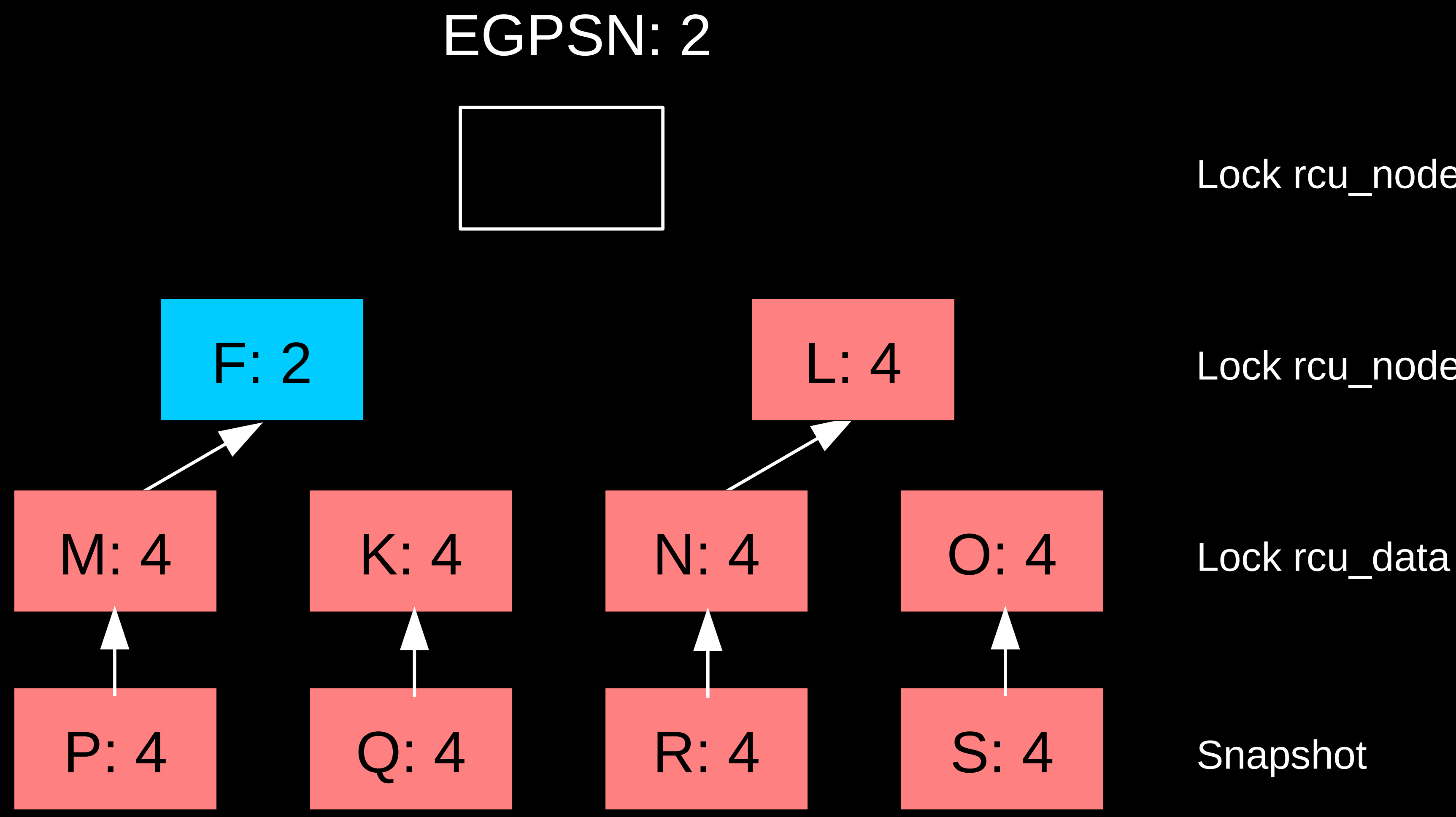
Expedited Grace Period Example



Expedited Grace Period Example

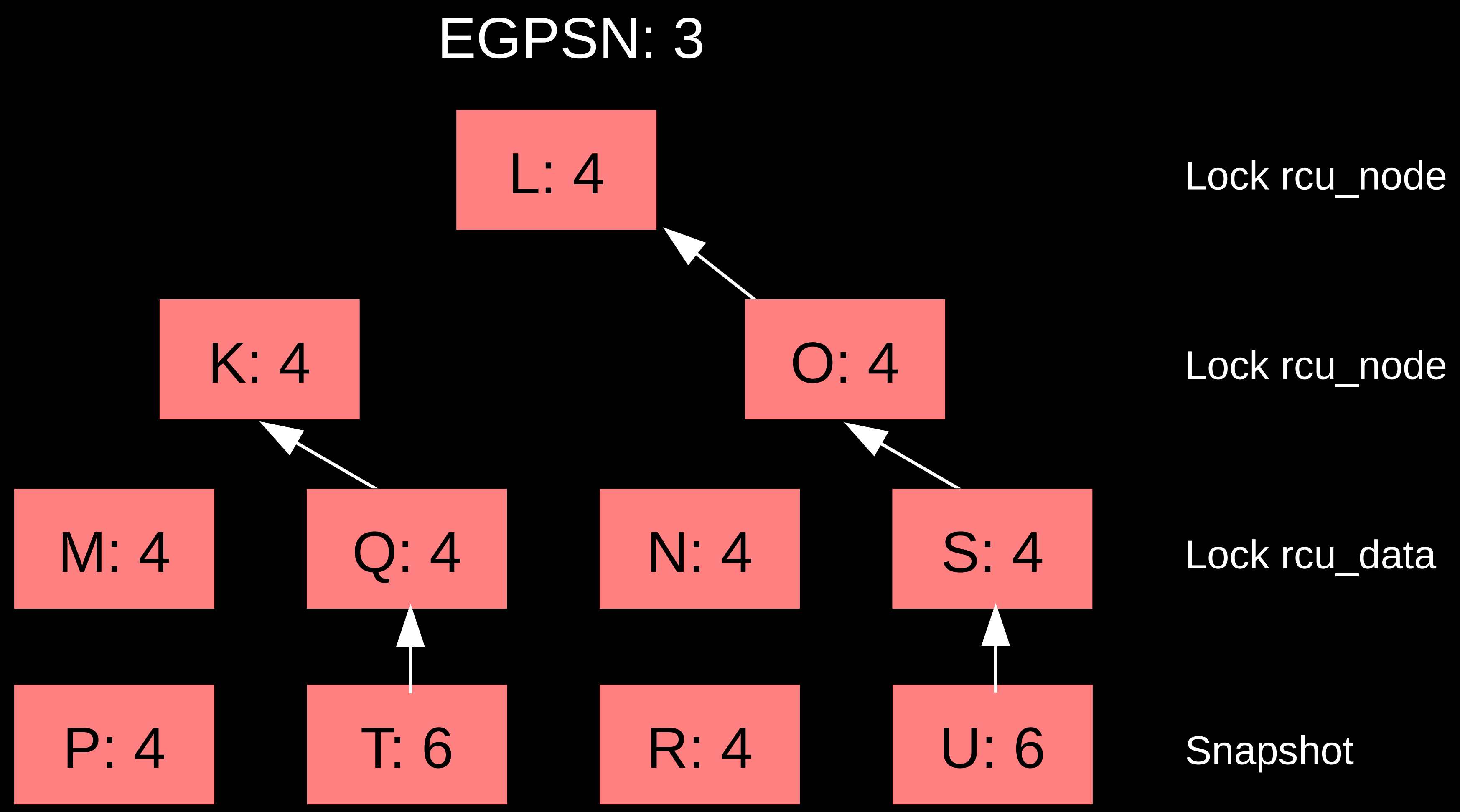


Expedited Grace Period Example

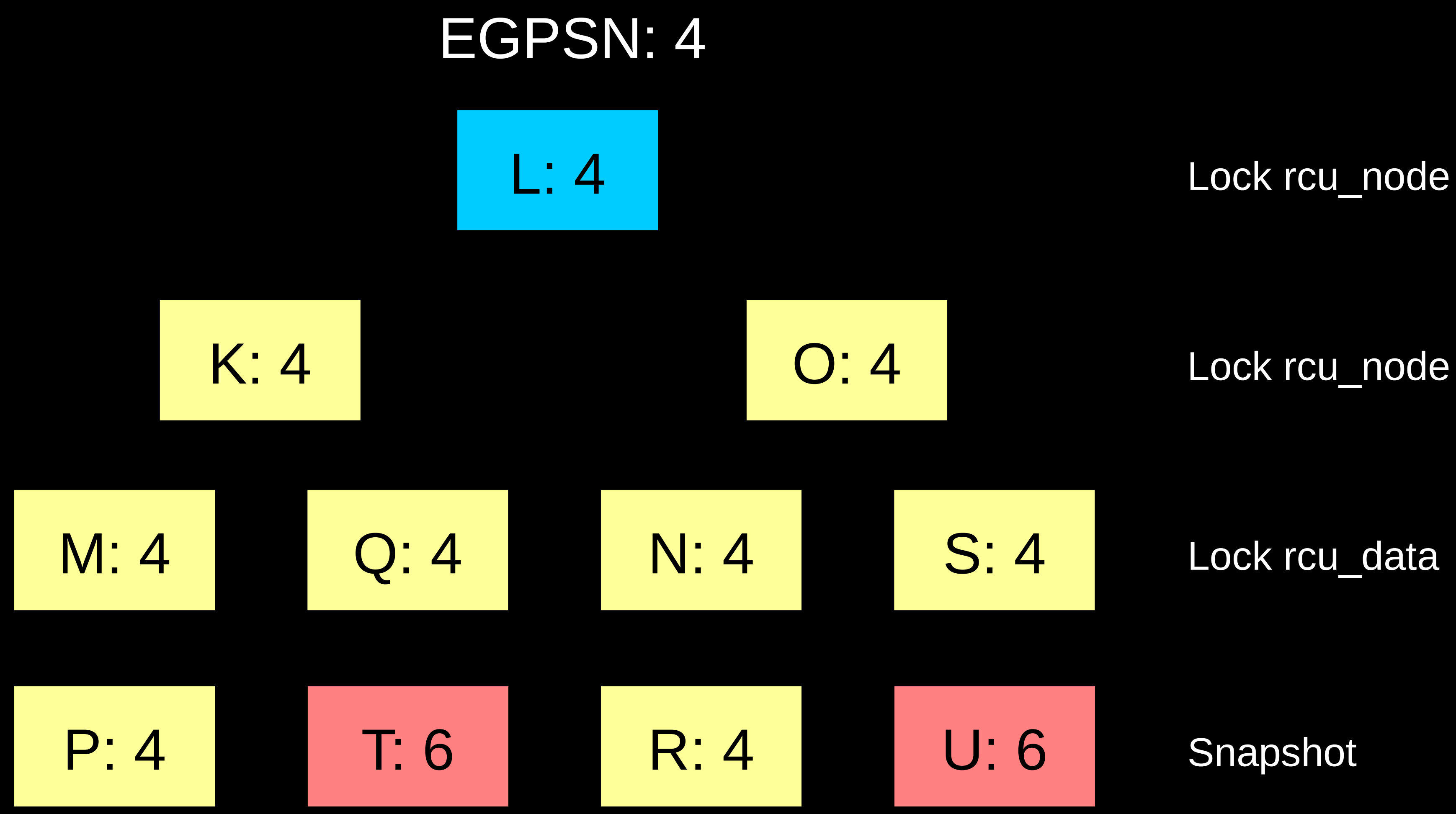


Fully parallel recognition of batching!

Expedited Grace Period Example



Expedited Grace Period Example



This time, one expedited grace period serves nine requests

Great Performance and Scalability!!!

Great Performance and Scalability!!!
In Theory, Anyway...

Let's Do Some Benchmarking!!!

Let's Do Some Benchmarking!!!
How Hard Can It Be???

Let's Do Some Benchmarking!!! How Hard Can It Be???

- Tight loops doing synchronize_sched_expedited() with other tight loops doing rcu_read_lock(): rcu_read_unlock()

Let's Do Some Benchmarking!!! How Hard It Can Be...

- Tight loops doing synchronize_sched_expedited() with other tight loops doing rcu_read_lock(): rcu_read_unlock()
  - Which resulted in horrid grace-period latencies: hundreds of ms!!!
- Small update:
  - rcu_read_lock(): cond_resched_rcu_qs(); rcu_read_unlock()

Let's Do Some Benchmarking!!! How Hard It Can Be...

- Tight loops doing synchronize_sched_expedited() with other tight loops doing rcu_read_lock(): rcu_read_unlock()
  - Which resulted in horrid grace-period latencies: hundreds of ms!!!
- Small update:
  - rcu_read_lock(): cond_resched_rcu_qs(); rcu_read_unlock()
  - Some improvement, but still not good
- Make expedited grace periods note interrupt from idle

Let's Do Some Benchmarking!!! How Hard It Can Be...

- Tight loops doing synchronize_sched_expedited() with other tight loops doing rcu_read_lock(): rcu_read_unlock()
  - Which resulted in horrid grace-period latencies: hundreds of ms!!!
- Small update:
  - rcu_read_lock(): cond_resched_rcu_qs(); rcu_read_unlock()
  - Some improvement, but still not good
- Make expedited grace periods note interrupt from idle
  - Still painful
- Pin the looping kthreads to their own CPUs

Let's Do Some Benchmarking!!! How Hard It Can Be...

- Tight loops doing synchronize_sched_expedited() with other tight loops doing rcu_read_lock(): rcu_read_unlock()
  - Which resulted in horrid grace-period latencies: hundreds of ms!!!
- Small update:
  - rcu_read_lock(): cond_resched_rcu_qs(); rcu_read_unlock()
  - Some improvement, but still not good
- Make expedited grace periods note interrupt from idle
  - Still painful
- Pin the looping kthreads to their own CPUs
  - Better, but still not great – and essentially no batching!!!

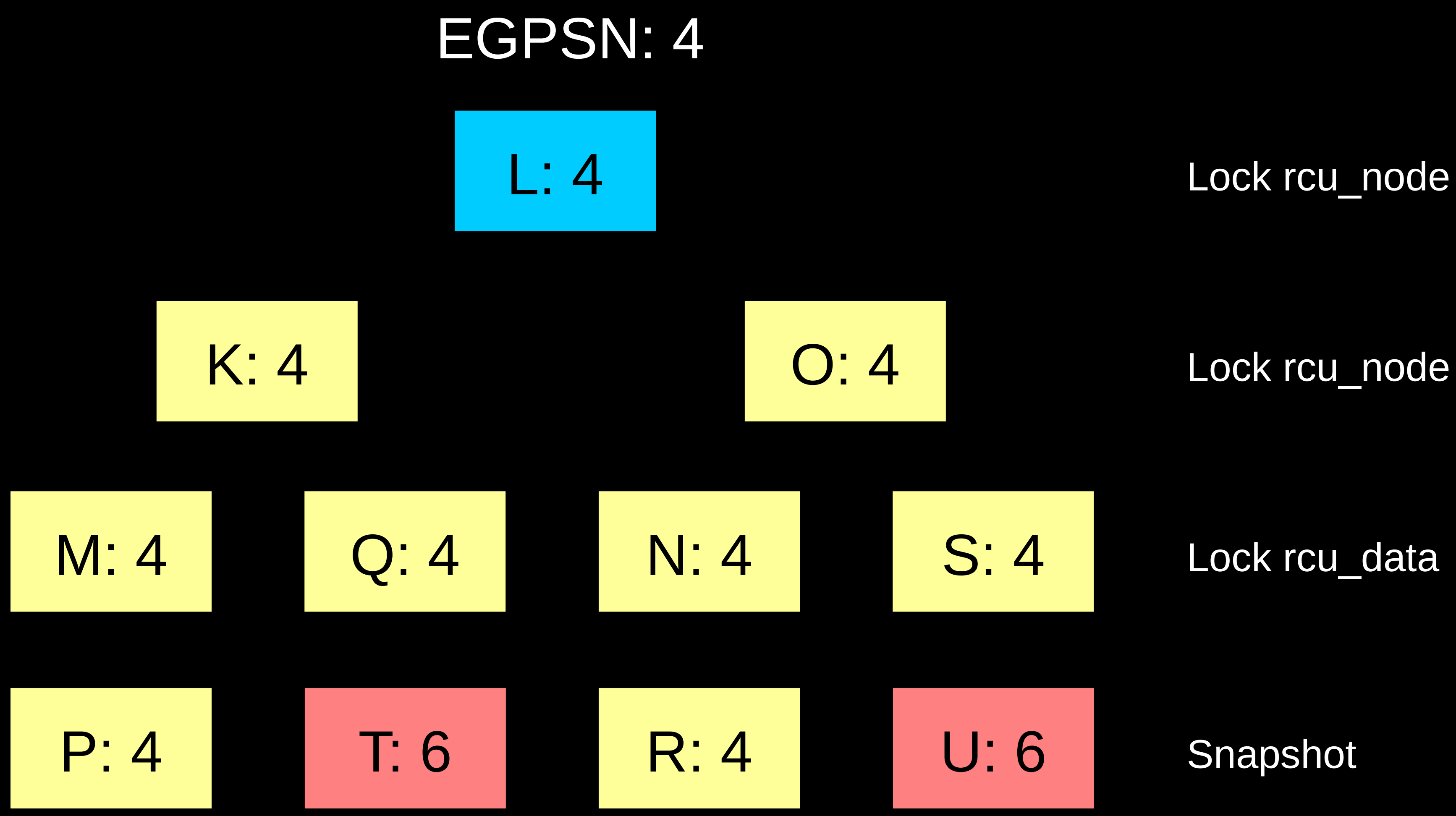
Let's Do Some Benchmarking!!!
How Hard It Can Be...

- Set kthreads doing grace periods to real-time priority
  - Tens of ms instead of hundreds of ms, better, but...
- Get the readers out of the way
  - Not much difference...
- Make cond_resched_rcu_qs() respond to expedited grace period requests
  - Not much difference
- Get IRC from Sasha Levin saying that KASAN complains about address-out-of-range errors
  - What exactly does C do with double subscripts? The wrong thing...
  - So ditch the double subscripts in favor of explicit pointer traversals

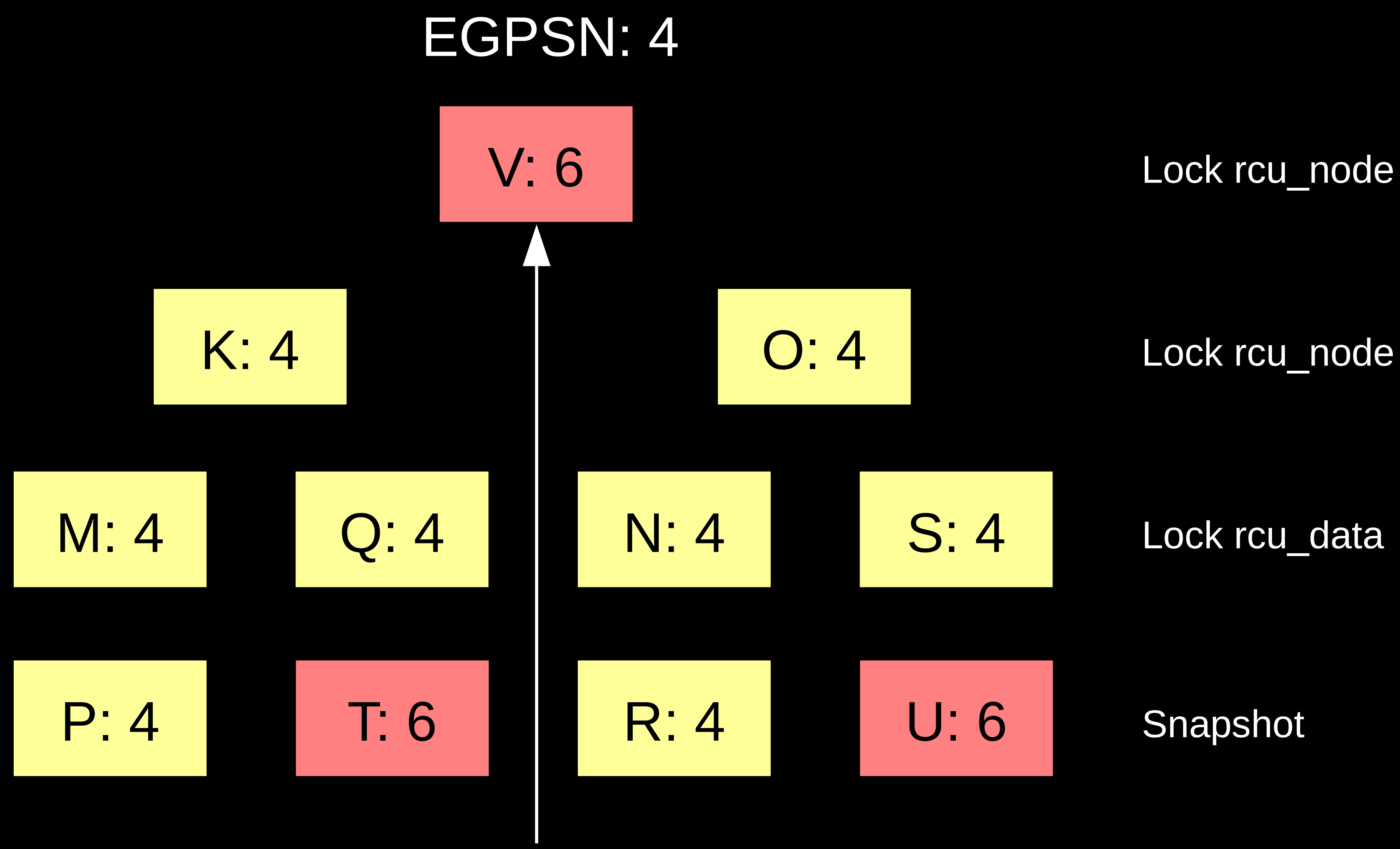
Let's Do Some Benchmarking!!! How Hard It Can Be...

- Collect data via ftrace rather than printk
  - Gets rid of some preemptions...
  - Still greater than 10 milliseconds worst case, so look at ftrace!
- New arrivals jumping the queue!!!

Queue-Jumping Problem



Queue-Jumping Problem



Let's Do Some Benchmarking!!!
How Hard It Can Be...

- Collect data via ftrace rather than printk
  - Gets rid of some preemptions...
  - Still greater than 10 milliseconds worst case, so look at ftrace!
- New arrivals jumping the queue!!!
  - So eliminate the queue-jumping optimization
  - But only minor improvements in worst case and in batching
- New arrivals still jumping the queue due to wakeup latency

Queue-Jumping Problem Redux

EGPSN: 5

V: 6 — Lock rcu_node

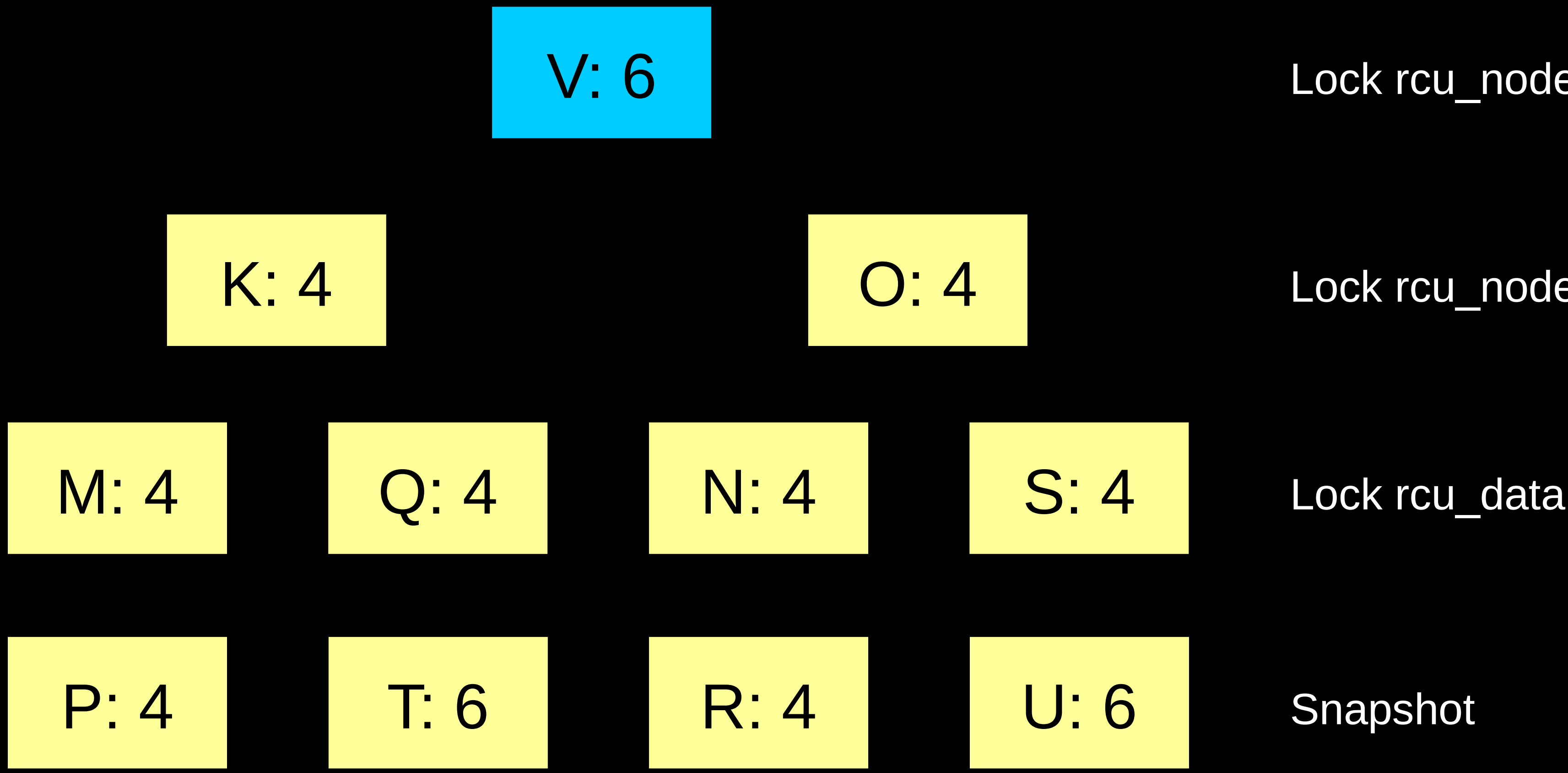
K: 4 | O: 4 — Lock rcu_node

M: 4 | Q: 4 | N: 4 | S: 4 — Lock rcu_data

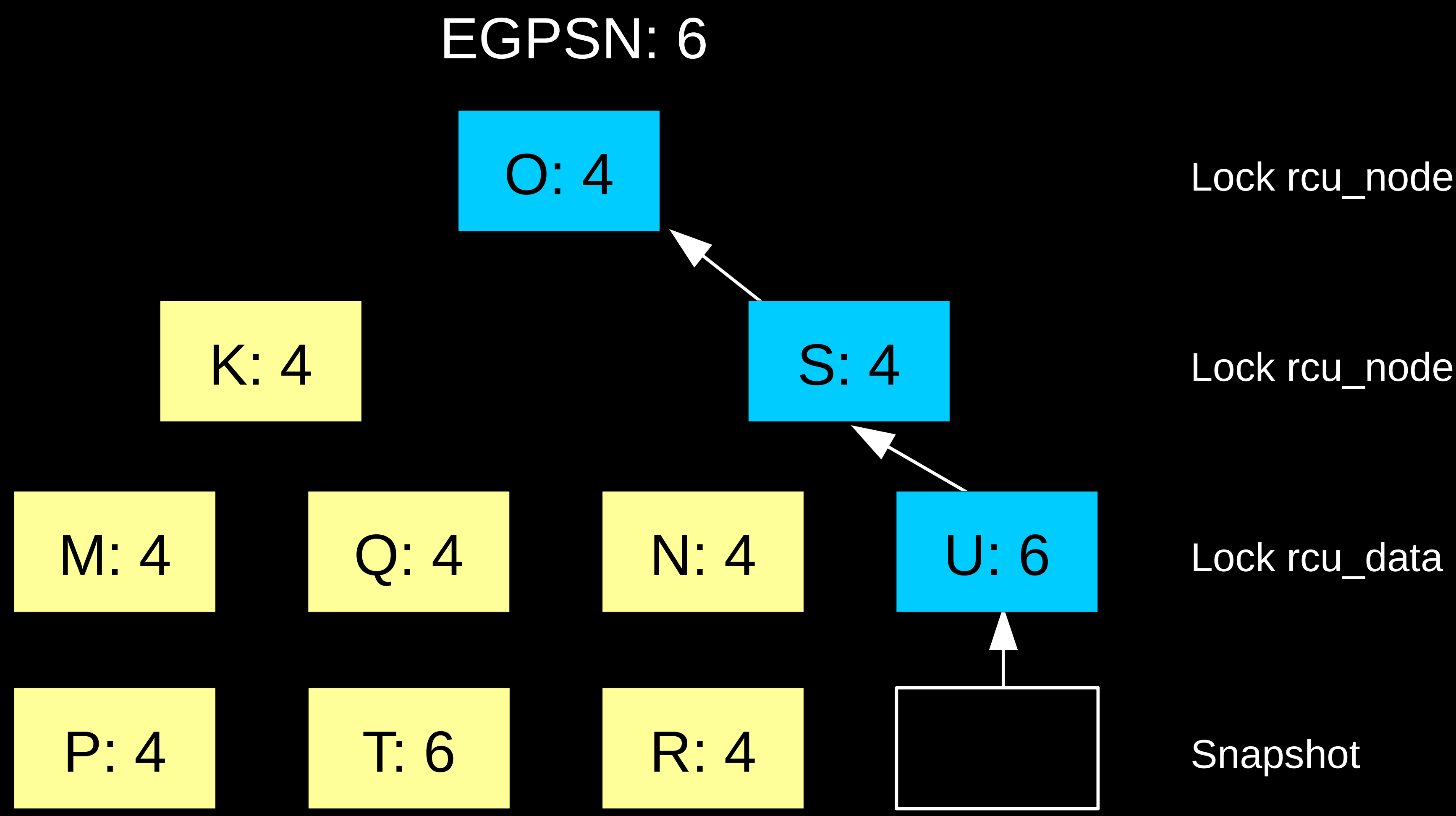
P: 4 | T: 6 | R: 4 | U: 6 — Snapshot

Queue-Jumping Problem Redux

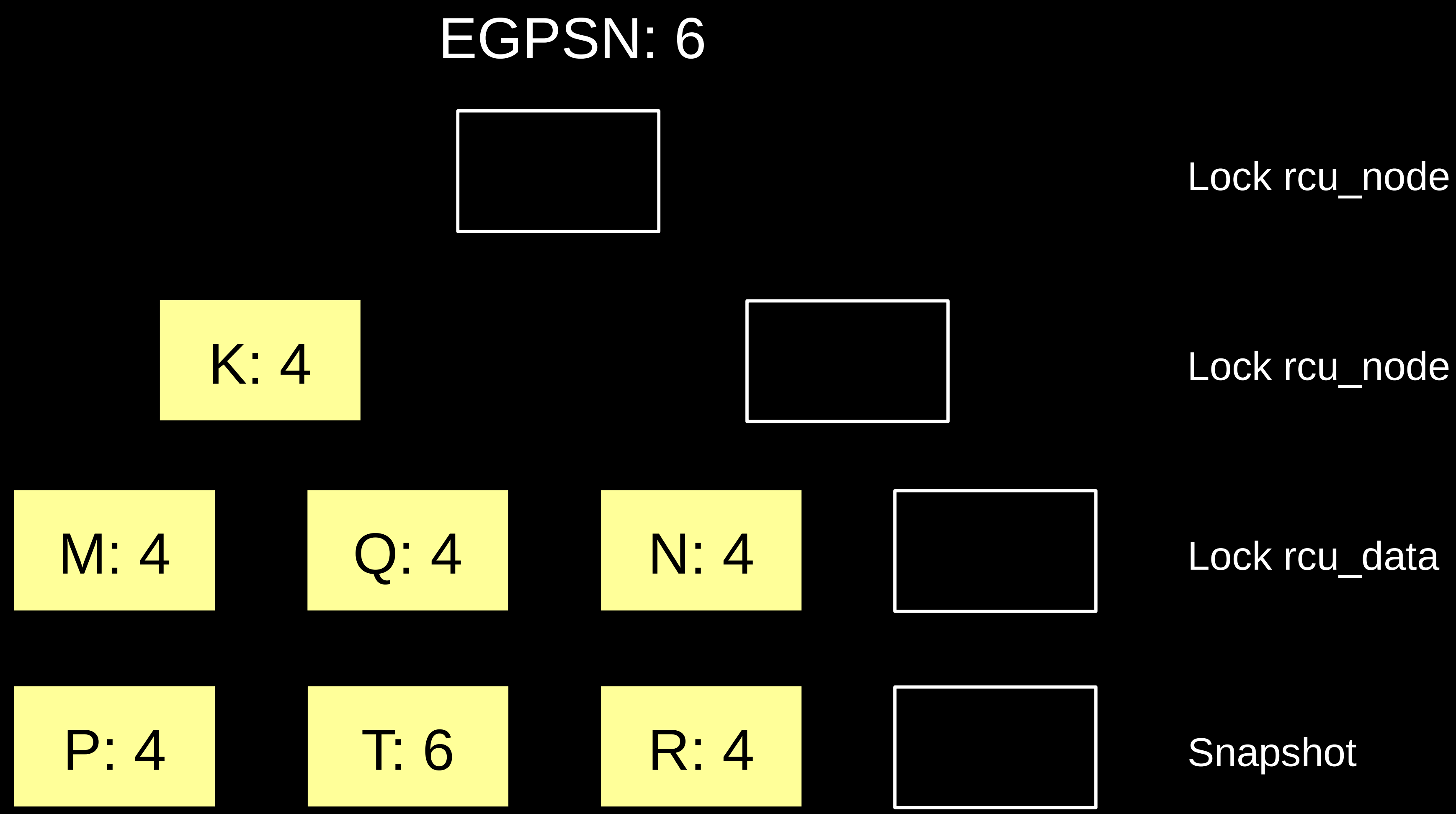
EGPSN: 6



Queue-Jumping Problem Redux

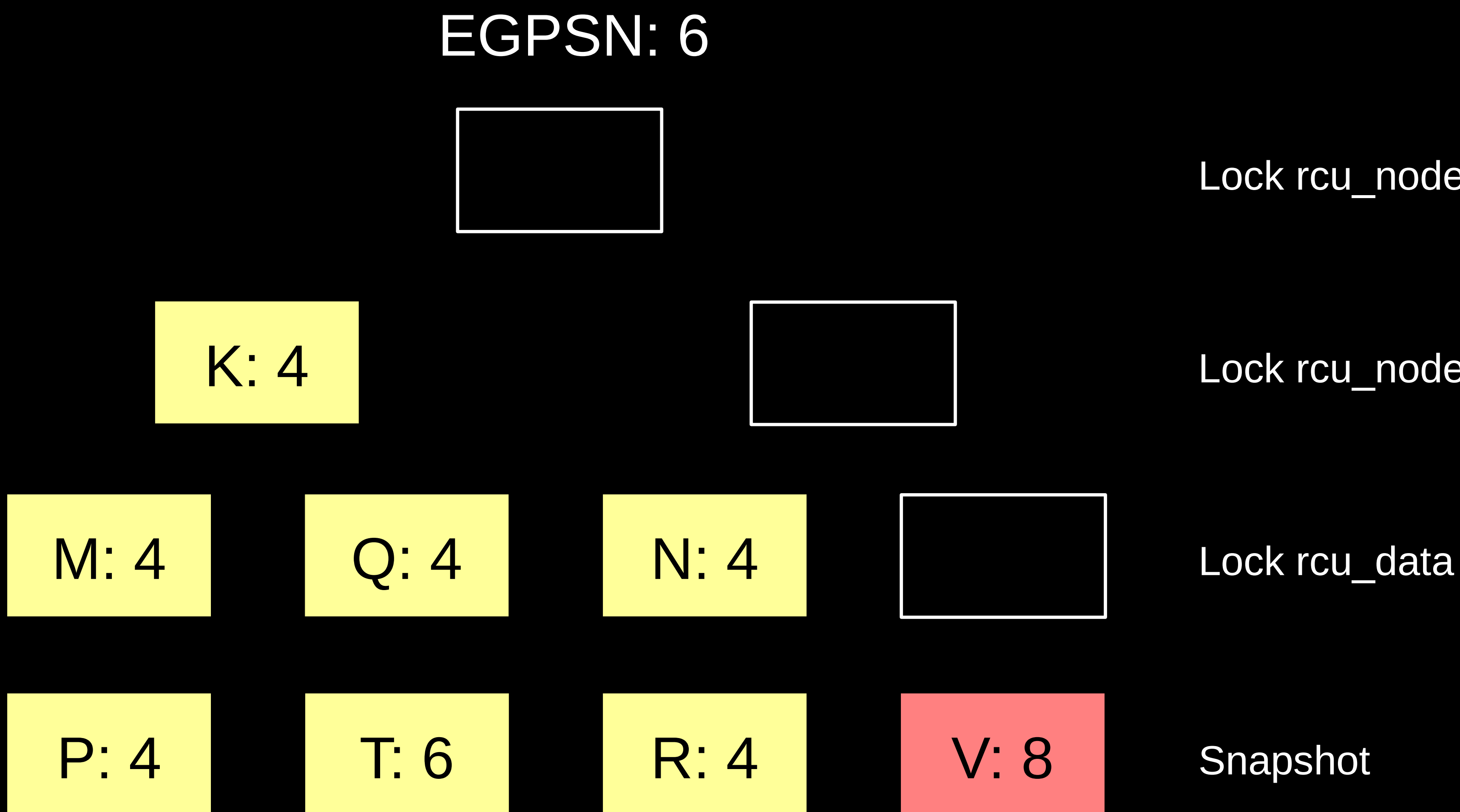


Queue-Jumping Problem Redux

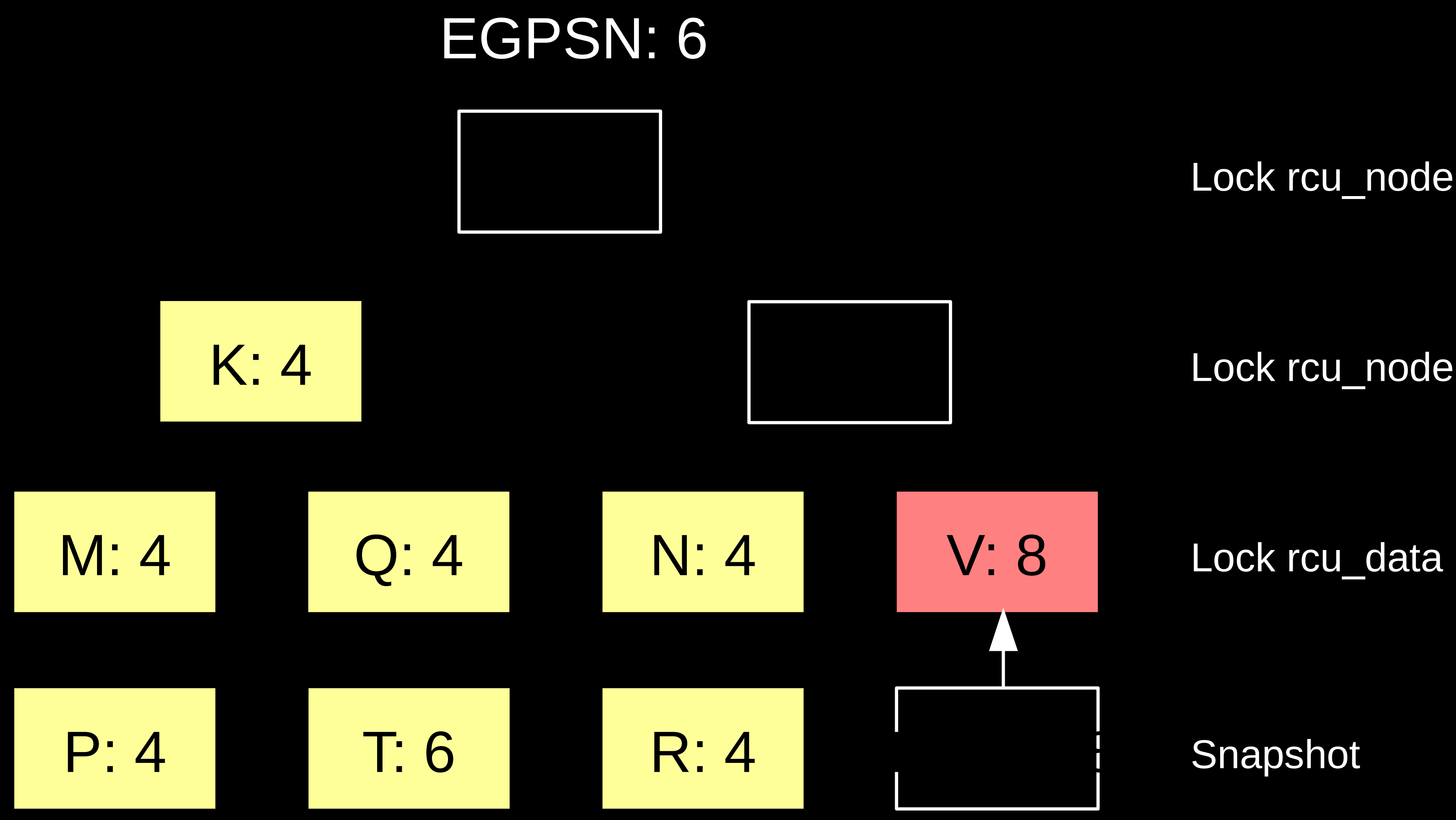


Wakeup delay can be significant, and in the meantime...

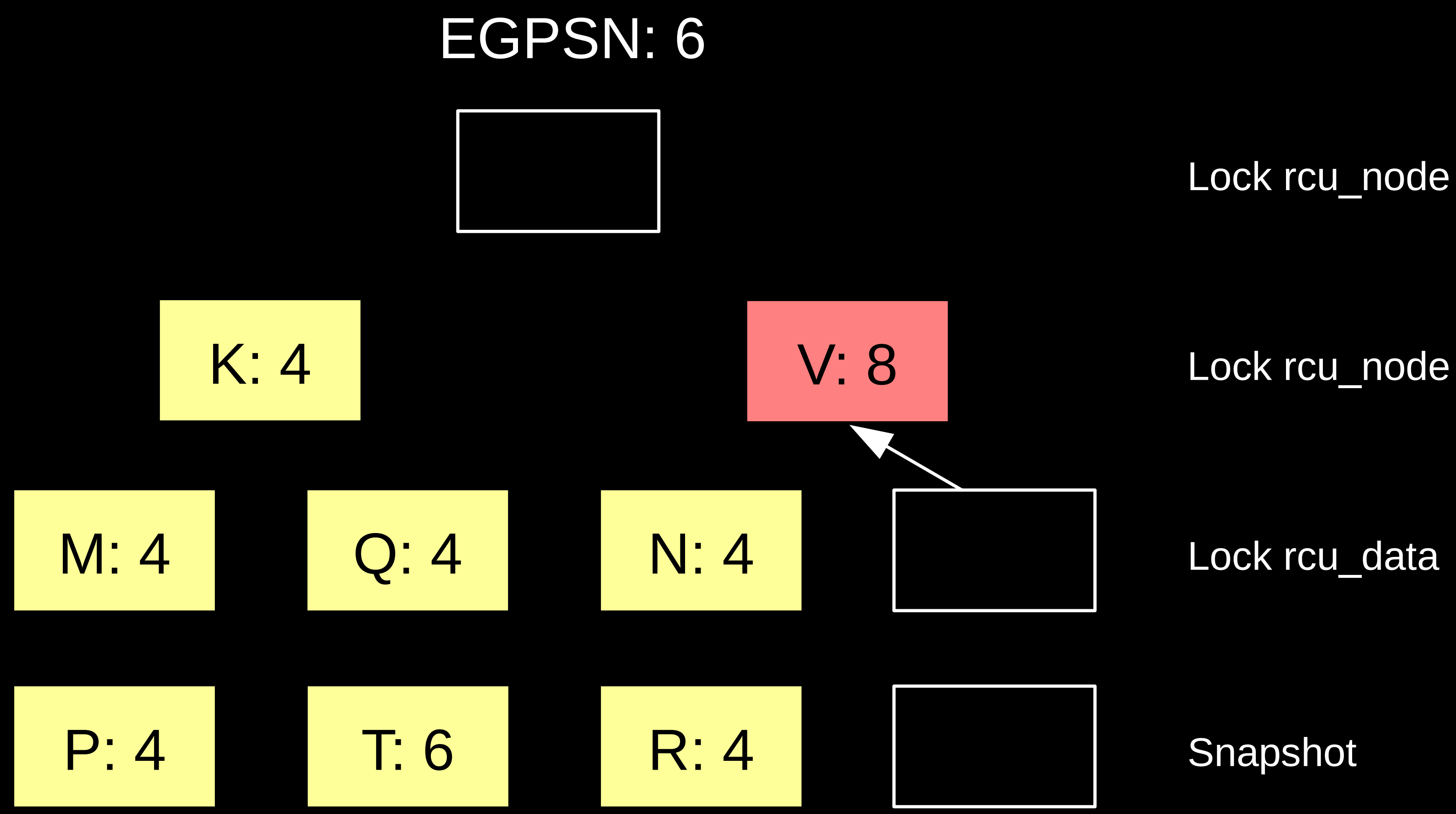
Queue-Jumping Problem Redux



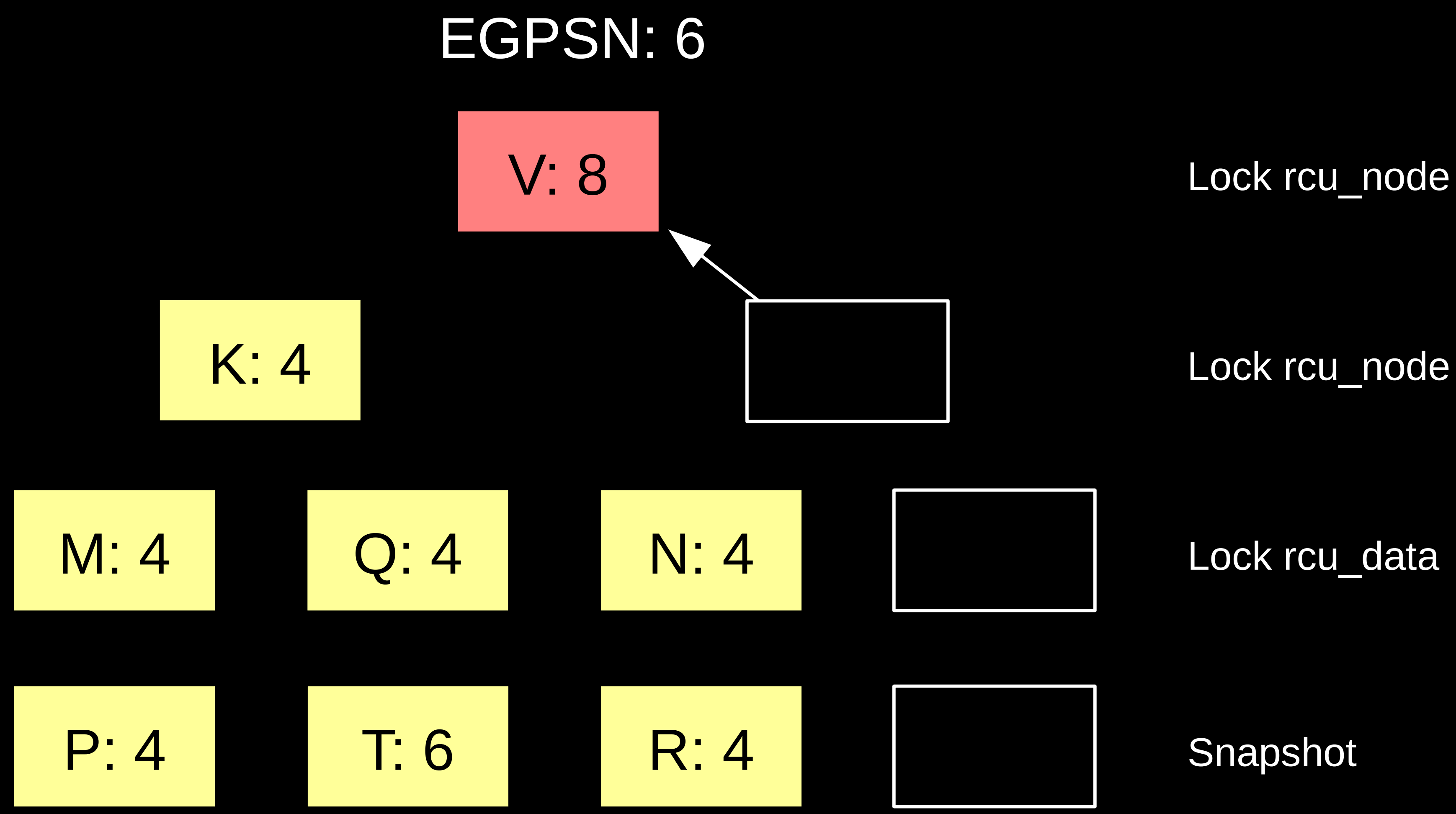
Queue-Jumping Problem Redux



Queue-Jumping Problem Redux



Queue-Jumping Problem Redux



Tasks K, M, P, Q, and T stuck waiting on Task V!!!

Let's Do Some Benchmarking!!! How Hard It Can Be...

- Collect data via ftrace rather than printk
  - Gets rid of some preemptions...
  - Still greater than 10 milliseconds worst case, so look at ftrace!
- New arrivals jumping the queue!!!
  - So eliminate the queue-jumping optimization
  - But only minor improvements in worst case and in batching
- New arrivals still jumping the queue due to wakeup latency
  - So switch from mutex to rt_mutex (worry about mainlining later...)
  - Much better!!! 6x batching on four CPUs, sub-10-ms latencies
  - But 4.7 milliseconds is not exactly expedited...

Let's Do Some Benchmarking!!! How Hard It Can Be...

- Automation causes entire benchmark to run at boot time
  - Not the best time for low OS jitter!
  - Delay the test until after boot completes (after a few false starts)
  - Maximum grace-period latency below 1ms, good batching
  - But getting RCU CPU stall warnings and RT throttling
- So put thread to SCHED_OTHER before ftrace_dump(), get rid of readers, and delay before ftrace_dump()
  - 99th percentile at 10 microseconds, max at about 500 microseconds
  - More like it!

Let's Do Some Benchmarking!!! How Hard It Can Be...

- Automation causes entire benchmark to run at boot time
  - Not the best time for low OS jitter!
  - Delay the test until after boot completes (after a few false starts)
  - Maximum grace-period latency below 1ms, good batching
  - But getting RCU CPU stall warnings and RT throttling
- So put thread to SCHED_OTHER before ftrace_dump(), get rid of readers, and delay before ftrace_dump()
  - 99^{th} percentile at 10 microseconds, max at about 500 microseconds
  - More like it!
- But six CPUs is a small fraction of 4096 CPUs!!!

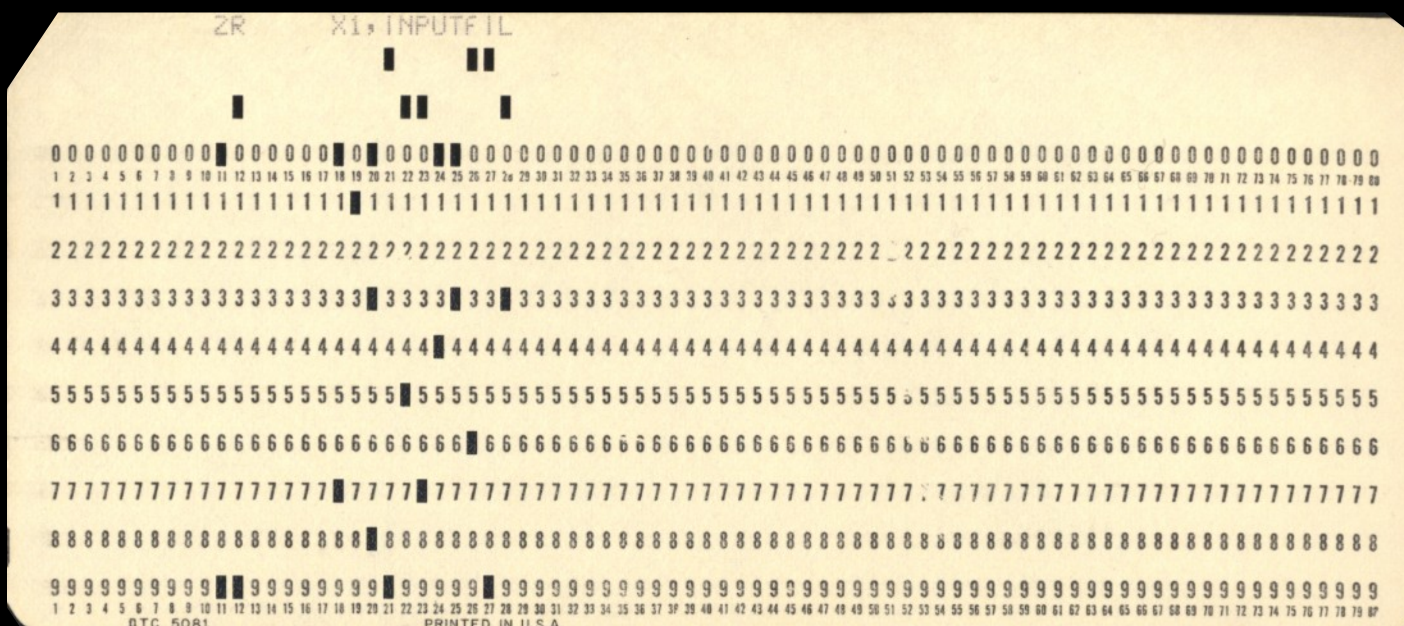
Benchmarking on 4096 CPUs

Benchmarking on 4096 CPUs

- I don't actually have access to a 4096-CPU system
  - Just to the bug reports filed by people who do have such systems
- But, as noted in the past, I have relevant experience:

Benchmarking on 4096 CPUs

- I don't actually have access to a 4096-CPU system
  - Just to the bug reports filed by people who do have such systems
- But, as noted in the past, I have relevant experience:



Benchmarking on 4096 CPUs

- Dirty trick #1: Note that synchronize_rcu_expedited() blocks
  - Can therefore run large numbers of tasks on smaller number of CPUs

Benchmarking on 4096 CPUs

- Dirty trick #1: Note that synchronize_rcu_expedited() blocks
  - Can therefore run large numbers of tasks on smaller number of CPUs
- But extremely long runtimes for 256 tasks on 32 CPUs...

Benchmarking on 4096 CPUs

- Dirty trick #1: Note that synchronize_rcu_expedited() blocks
  - Can therefore run large numbers of tasks on smaller number of CPUs
- But extremely long runtimes for 256 tasks on 32 CPUs...
  - Problem: Tasks with enough measurements compete for CPU time with those that are not yet done
    - But we need them to be running in order to provide needed load
    - Just not at realtime priority
  - Solution: Once a given tasks has enough measurements, drop it to non-realtime priority
    - Allows scheduler to determine which tasks are important
    - Decreases runtime by more than a factor of three
    - So that I might be able to collect enough data in time for this talk!!!

Dirty Trick #1 Results (32 CPUs, 256 Tasks)

Min	Mean	99th Percentile	Maximum	Batching
1 us	35.6 us	276 us	806 us	68.9
3 us	40.7 us	284 us	512 us	81.6
1 us	59.3 us	257 us	1146 us	149.6

Benchmarking on 4096 CPUs

- Dirty trick #1: Note that synchronize_rcu_expedited() blocks
  - Can therefore run large numbers of tasks on smaller number of CPUs
- Dirty trick #2: Decrease fanouts to obtain a full-height RCU combining tree with smaller numbers of CPUs
  - 54 CPUs, RCU_FANOUT=3, RCU_FANOUT_LEAF=2: Four levels

Dirty Trick #2 Results (54 CPUs, 256 Tasks, 4 Levels)

Min	Mean	99th Percentile	Maximum	Batching
12 us	591.5 us	3492 us	5562 us	96.8
9 us	597.8 us	3777 us	5859 us	97.8
108 us	6739.5 us	34021 us	38133 us	126.2
59 us	12610.5 us	86876 us	140910 us	130.7
11 us	797.8 us	5127 us	11827 us	105.0
6 us	568.0 us	2254 us	5042 us	80.1

Horrible results, probably due to new interactions in the taller tree.
And greater interference from other users on this shared machine.

Dirty Trick #2 Results (54 CPUs, 256 Tasks, 2 Levels)

Min	Mean	99th Percentile	Maximum	Batching
11 us	220.2 us	553 us	690 us	182.2
6 us	169.4 us	1034 us	1558 us	178.1
5 us	166.9 us	1177 us	3025 us	111.7

Increased confidence of likely new interactions in the taller tree.
And greater interference from other users on this shared machine.

Benchmarking on 4096 CPUs

- Dirty trick #1: Note that synchronize_rcu_expedited() blocks
  - Can therefore run large numbers of tasks on smaller number of CPUs
- Dirty trick #2: Decrease fanouts to obtain a full-height RCU combining tree with smaller numbers of CPUs
  - 54 CPUs, RCU_FANOUT=3, RCU_FANOUT_LEAF=2: Four levels
  - But lab machine uses rotating rust, and it therefore takes a good long time to dump out the ftrace data
  - Longer-term fix: Do the data reduction in the kernel

Benchmarking on 4096 CPUs

- Dirty trick #1: Note that synchronize_rcu_expedited() blocks
  - Can therefore run large numbers of tasks on smaller number of CPUs
- Dirty trick #2: Decrease fanouts to obtain a full-height RCU combining tree with smaller numbers of CPUs
  - 54 CPUs, RCU_FANOUT=3, RCU_FANOUT_LEAF=2: Four levels
  - But lab machine uses rotating rust, and it therefore takes a good long time to dump out the ftrace data
  - Longer-term fix: Do the data reduction in the kernel
  - Even longer-term fix: Use a system with 4096 real CPUs

Benchmarking on 4096 CPUs

- Dirty trick #1: Note that synchronize_rcu_expedited() blocks
  - Can therefore run large numbers of tasks on smaller number of CPUs
- Dirty trick #2: Decrease fanouts to obtain a full-height RCU combining tree with smaller numbers of CPUs
  - 54 CPUs, RCU_FANOUT=3, RCU_FANOUT_LEAF=2: Four levels
  - But lab machine uses rotating rust, and it therefore takes a good long time to dump out the ftrace data
  - Longer-term fix: Do the data reduction in the kernel
  - Even longer-term fix: Use a system with 4096 real CPUs
- More dirty tricks will likely be required!

Summary and Lessons (Re)learned

Summary and Lessons (Re)learned

- Benchmarking is not as easy as it looks ;-)
- Obvious optimizations often aren't
  - Uncontended-case fastpath to root node problematic
- Maintaining request order is important in this case
  - Which is unfortunate, as this can be complex and expensive
- Fixed a couple of performance bugs:
  - Make expedited grace period IPI handlers check for idle
  - Make cond_resched_rcu_qs() satisfy expedited grace periods
  - And I have at least one more to fix!
- At the end of the day, real full-scale testing is needed
  - There are likely to be other performance bugs
    - IPIs sent serially, wakeups likely to be a bottleneck, ...
  - But it is good to get a couple of them out of the way!!!

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?