Towards Hard Realtime Response from the Linux Kernel: Adapting RCU to Hard Realtime

Paul E. McKenney
IBM Beaverton

2005 linux.conf.au

Overview

- Approaches to Realtime
- The Role of RCU in Realtime
- What Do Realtime Kernels Need from RCU?
- RCU Options for Aggressive Realtime
- Current Status
- Summary

Approaches to Realtime

Why Realtime Linux?

- Way too many RTOSes!
  - Software balkanization
  - But there are workloads that can only be handled by hand-coded assembly on bare metal
- "Nintendo generation" & sub-reflex response
  - Some of us old guys are impatient, too!!!
- With machines talking to machines, delays accumulate
- In developed countries, people are spendy

So What is the Big Deal?

- Linux was not designed to be a realtime OS
  - Neither was any other UNIX
- Non-realtime assumptions are scattered throughout the kernel
- Any excessive latency anywhere in the kernel, no matter how infrequently executed, will mess up realtime latency
- But the same used to be true of SMP...
  - And still is, to some extent...

Realtime Strategies

- Preemption
  - CONFIG_PREEMPT
    - Kernel is preemptable *except* for critical sections
  - CONFIG_PREEMPT_RT
    - Kernel is preemptable almost everywhere
- Nested OS (e.g., RT Linux, Adeos)
- Dual OS (dual core, hypervisor)
- Migration

Classifying RT Approaches

- Quality:
  - Hard vs. soft, probability, redundancy
  - Timeframe: ps, ns, us, ms, s, ...
  - Services: interrupt, process, user-mode, I/O, ...
- API: POSIX, Windows, Ad Hoc
- Visibility: global vs. split
- Configurations: UP/SMP, #devices, ...

non-CONFIG_PREEMPT

- Soft realtime
- 10s of milliseconds
- All services (but some I/O can still slow)
- POSIX API, limited RT extensions
- Global visibility
- All configurations, including SMP

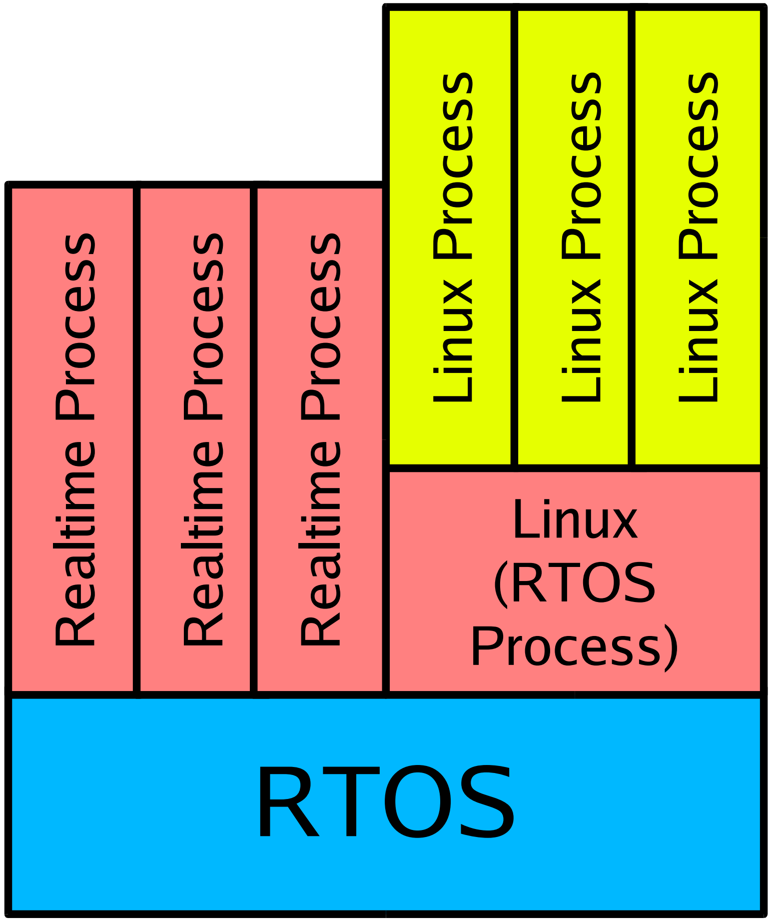
CONFIG_PREEMPT

- Soft realtime
- 100s of microseconds
- Process scheduling, some syscalls
- POSIX API, limited RT extensions
- Global visibility
- All configurations, including SMP
  - But UP much more common

CONFIG_PREEMPT_RT

- Soft realtime
- 10s of microseconds
- Process scheduling, a few syscalls
  - Can in theory give hard realtime to user code...
- POSIX API, limited RT extensions
- Global visibility
- All configurations, including SMP
  - But UP much more stable

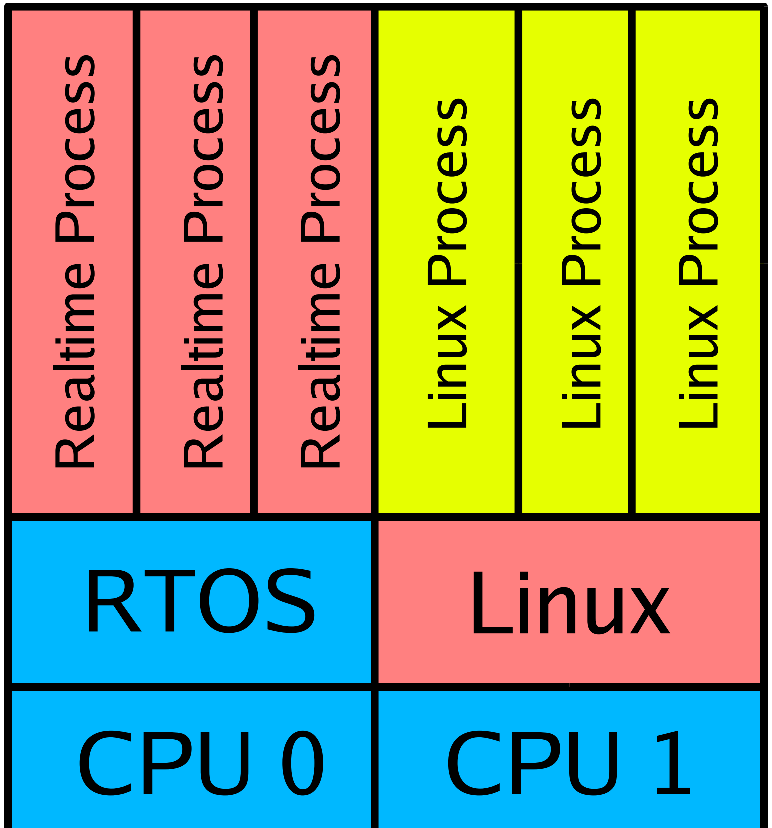
Nested OS


Realtime Process
Realtime Process
Realtime Process
Linux Process
Linux Process
Linux Process
Linux (RTOS Process)
RTOS

Nested OS

- Hard realtime
- ~10 microseconds
- “All realtime services”, often quite limited
- Subset of POSIX API, with RT extensions
- Linux runs as process in RTOS instance
- All configurations, including SMP
  - But UP much more common

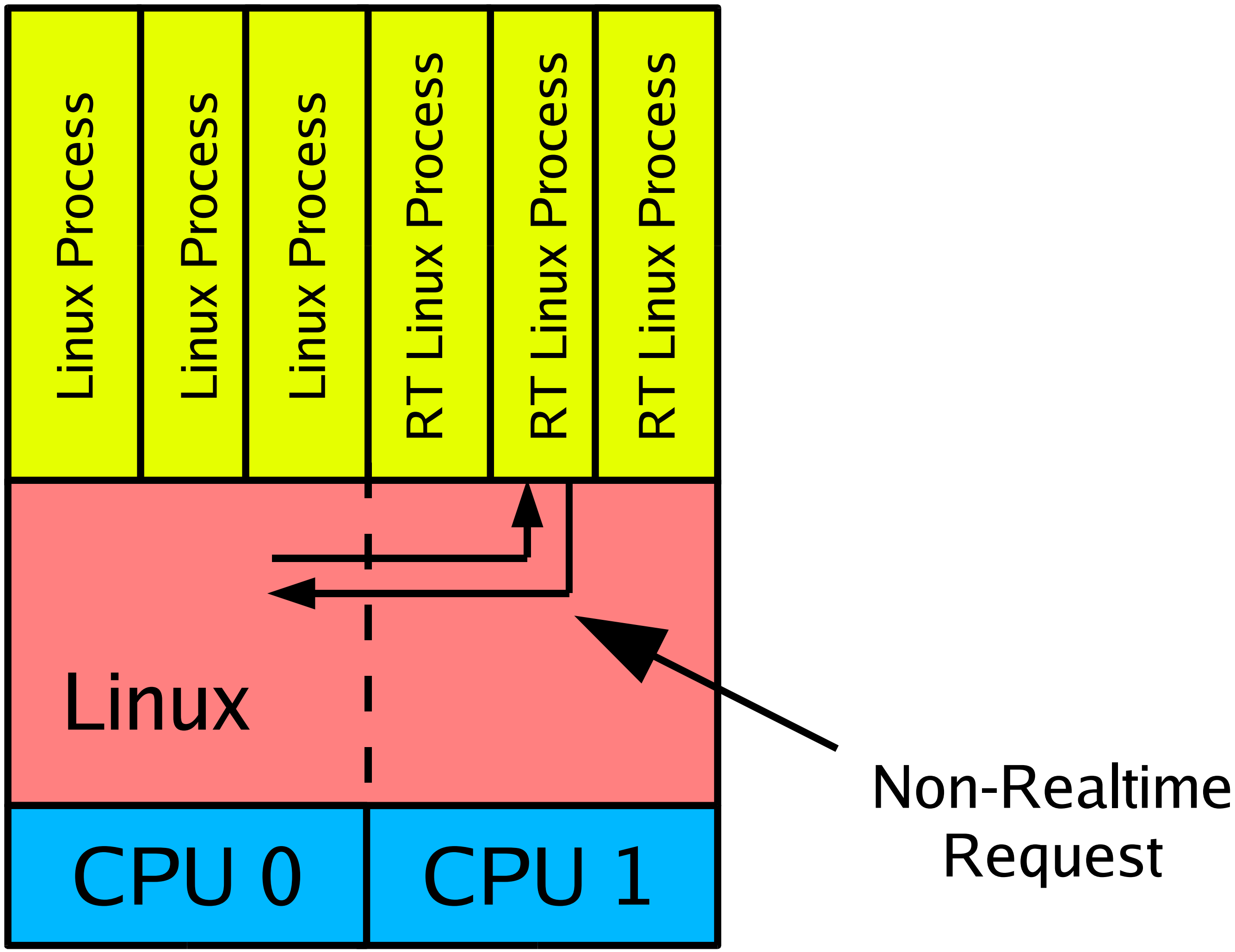
Dual OS / Dual Core



Dual OS / Dual Core

- Hard realtime
- Can be sub-microsecond
- "All realtime services", often quite limited
  - In extreme cases, bare metal
- Subset of POSIX API, with RT extensions
- Separate RTOS instance
- All configurations, including SMP

Migration


Linux Process
Linux Process
Linux Process
RT Linux Process
RT Linux Process
RT Linux Process
Linux
CPU 0
CPU 1
Non-Realtime
Request

Migration

- You have all heard about preemption...
  - CONFIG_PREEMPT_RT's RCU preempted my work on the migration version of realtime. :-/
- The abstract for this talk is therefore somewhat obsolete
- But if you want a speculative evaluation of migration...

Migration

- Hard realtime
- Can be sub-microsecond
- User-level execution
  - Can adapt system-call by system-call
  - Avoids “pinning” syscalls holding critical locks
- Subset of POSIX API, with RT extensions
- Global visibility
- All configurations, but need SMP (real or emulated)

Realtime Summary

- There are a number of realtime approaches for Linux
- They all have their own peculiar strengths and shortcomings
- Thus far, one size does *not* fit all
- It would be good to get to one size that fits all
  - Will require combination of approaches
  - Or will require additional innovation...

The Role of RCU in Realtime

What is Synchronization?

```
spin_lock(&my_lock);
p = head;
/* Can mess with struct to by p */
spin_unlock(&my_lock);

/* Can -not- mess with struct pointed to by p!!! */

spin_lock(&my_lock);
/* Can -not- mess with struct pointed to by p!!! */
p = head;
/* Can again mess with struct to by p */
spin_unlock(&my_lock);
```

What is Synchronization?

- Mechanism ***plus coding convention***
  - Locking: must hold lock to reference or update
  - NBS: must use carefully crafted sequences of atomic operations to do references and updates
  - RCU coding convention:
    - Must define "quiescent states" (QS)
      - e.g., context switch in non-CONFIG_PREEMPT kernels
    - QSes must not appear in read-side critical sections
    - CPU in QSes are guaranteed to have completed all preceding read-side critical sections
- RCU mechanism: "lazy barrier" that computes "grace period" given QSes.

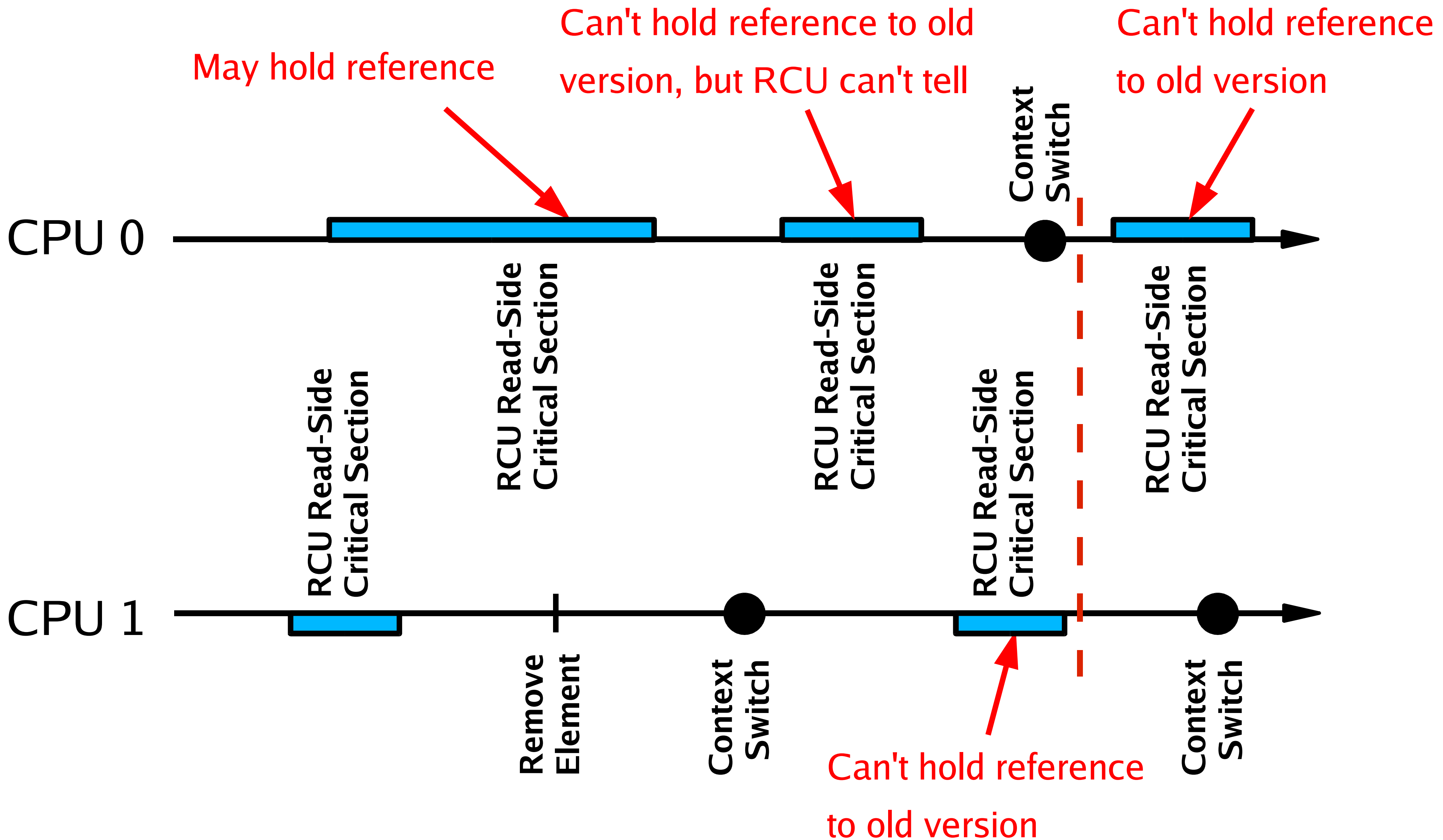
What is RCU? (1)

- Reader-writer synchronization mechanism
  - Best for read-mostly data structures
- Writers create new versions atomically
  - Normally create new and delete old elements
- Readers can access old versions independently of subsequent writers
  - Old versions garbage-collected by “poor man's” GC, deferring destruction
  - Readers must signal “GC” when done

What is RCU? (2)

- Readers incur little or no overhead
- Writers incur substantial overhead
  - Writers must synchronize with each other
  - Writers must defer destructive actions until readers are done
  - The “poor man's” GC also incurs some overhead

RCU's Deferred Destruction



What Do Realtime Kernels Need From RCU?

Realtime RCU Requirements

- Reliable
- Callable from IRQ
- ***Preemptible read side***
- ***Small memory footprint***
- Synchronization-free read side
- Independent of memory blocks
- Freely nestable read side
- Unconditional read-to-write upgrade
- Compatible API

Trouble in RCU-Land

	Reliable	Callable From IRQ	Preemptible Read Side	Small Memory Footprint	Sync-Free Read Side	Indpt of Memory Blocks	Nestable Read Side	Uncond R-W Upgrade	Compatible API
Classic RCU			N	N					
rcu-preempt				X	N				
Jim Houston Patch			N		N				
Reader-Writer Locking					N		N	N	n
Unconditional Hazard Pointers				X	n	N			
Hazard Pointers: Failure				n	n	N			N
Hazard Pointers: Panic	N			n	n	N			
Hazard Pointers: Blocking		N		n	n	N			
Reference Counters				N	n	N			
rcu_donereference()					n	N			N

RCU Options for Aggressive Realtime

I Wrote the Paper At This Point...

- RCU gets twisted pretty badly by realtime
- No good RCU implementation exists
  - There is not even a *poor* implementation, they all have *serious* shortcomings
- Some potential advantage
  - Marking read side with update lock nice
  - But what if no update lock? Or lots of them?
- Something better is needed!!!

Tom Hart Confused Me

	SMR	QSBR	EBR
Concurrently-readable	**CR-SMR**	**CR-QSBR**	**CR-EBR**
Lock-free	**LF-SMR**	**LF-QSBR**	**LF-EBR**

	SMR	QSBR	EBR
Lock-based update	**CR-SMR**	**CR-QSBR**	**CR-EBR**
Lock-free update	**LF-SMR**	**LF-QSBR**	**LF-EBR**

Solution From Confusion

	SMR	QSBR	EBR	“LBR”
Lock-based update	CR-SMR	CR-QSBR	CR-EBR	???
Lock-free update	LF-SMR	LF-QSBR	LF-EBR	???

We can use locking to force grace period...

Or counters to suppress grace periods

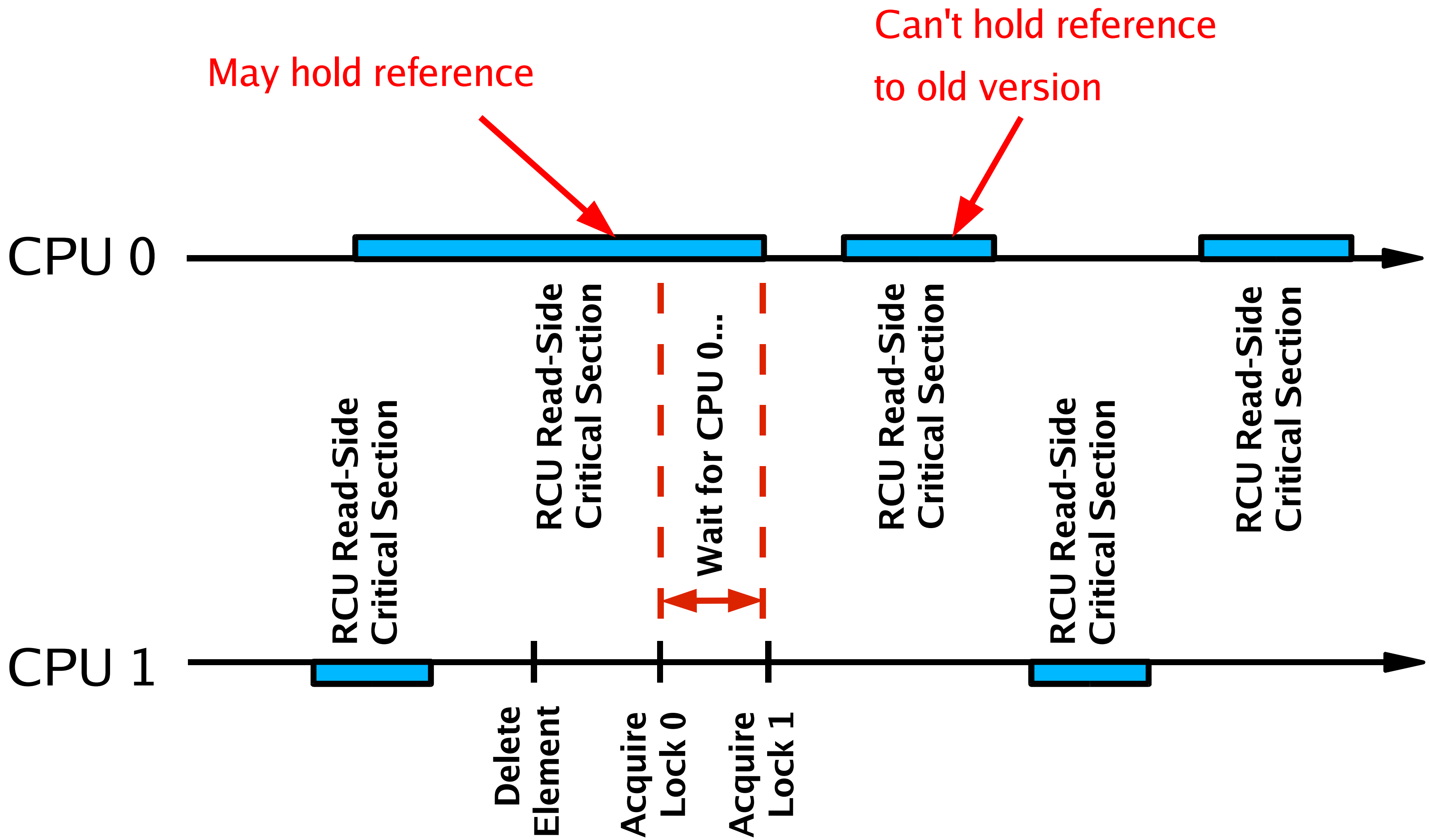
Simple Lock-Based Deferred Free

```
void rcu_read_lock(void)
{
   read_lock(&rcu_ctrlblk.lock);
}

void rcu_read_unlock(void)
{
   read_unlock(&rcu_ctrlblk.lock);
}

void synchronize_kernel(void)
{
   write_lock_bh(&rcu_ctrlblk.lock);
   write_unlock_bh(&rcu_ctrlblk.lock);
}
```

Grace Periods



Simple LBDF Issues

- Latencies can be communicated from one read side to another
  - Reader 1 in critical section
  - synchronize_kernel() waiting for reader 1's lock
  - Reader 2 blocked waiting for synchronize_kernel
- Locks on read side – heavy overhead!!!
- Also, PREEMPT_RT allows only one reader
- But this simple mechanism is Paul McKenney's PREEMPT_RT “lesson plan”
  - Crude version runs on 4-CPU machine
  - If using 0.7.41-14 version of PREEMPT_RT

Grace Period Suppression

- Increment a per-CPU counter in rcu_read_lock()
  - decrement same counter in rcu_read_unlock()
  - track counter in task struct (preemption!)
- While a given CPU's counter is non-zero, ignore any quiescent states that this CPU goes through
  - could result in extremely long grace periods
  - possibly fix via “count flipping”.

Current Status

Improvements in RCU-Land

	Reliable	Callable From IRQ	Preemptible Read Side	Small Memory Footprint	Sync-Free Read Side	Indpt of Memory Blocks	Nestable Read Side	Uncond R-W Upgrade	Compatible API
Classic RCU			N	N					
rcu-preempt				X	N				
Jim Houston Patch			N		N				
Reader-Writer Locking					N		N	N	n
Unconditional Hazard Pointers				X	n	N			
Hazard Pointers: Failure				n	n	N			N
Hazard Pointers: Panic	N			n	n	N			
Hazard Pointers: Blocking		N		n	n	N			
Reference Counters				N	n	N			
rcu_donereference()					n	N			N
Lock-Based Deferred Free					N				
Read-Side Counter GP Suppression				N	n				
Read-Side Counters w/ “Flipping”?					n				

Summary

Summary

- Numerous motivations for realtime Linux
- No mechanism currently suits all workloads
  - But Linux is becoming more capable
- RCU still an issue with aggressive realtime
  - However, several possible solutions identified
  - Some of which are positively mediocre

Legal Statement

- This work represents the view of the author, and does not necessarily represent the view of IBM.
- IBM is a registered trademarks of International Business Machines in the United States, other countries, or both.
- Linux is a registered trademark of The Open Group in the United States and other countries.
- Other company, product, and service names may be trademarks or service marks of others.