Extending RCU for Realtime and Embedded Workloads

Paul E. McKenney, IBM LTC
Dipankar Sarma, IBM ISL
Ingo Molnar, Red Hat
Suparna Bhattacharya, IBM ISL

2006 Ottawa Linux Symposium

July 21, 2006
(revised July 31, 2006)

Overview

- Introduction to RCU
- Realtime response and Classic RCU
- Lower-overhead realtime read-side primitives
- More scalable grace-period detection
- Better balance of throughput and latency for RCU callback invocation
- Lower per-structure memory overhead
- Priority boosting of RCU read-side critical sections
- Sleepable RCU(?)

Introduction to RCU

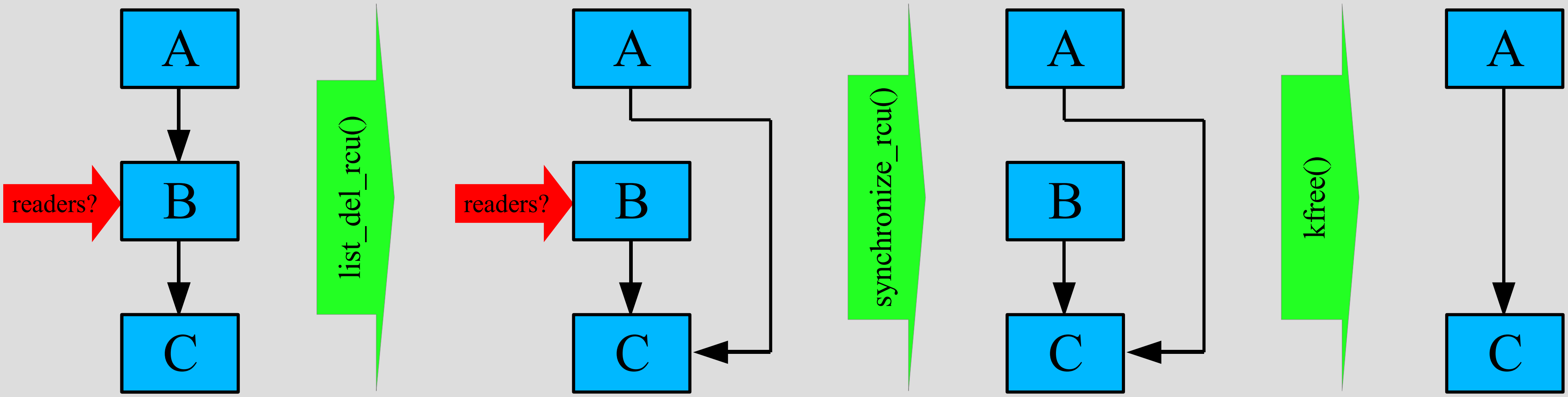
Why Not Just Use Locks??? Or Atomic Instructions???

	XServe		IBM POWER	
CPUs	2x 2.0 GHz PowerPC® G5		8x 1.45 GHz POWER4+™	
	Nanoseconds	Cycles	Nanoseconds	Cycles
Fence	78	156	76	110
cmpxchg	52	104	59	86
Lock Round Trip	231	462	243	352

- Atomic instructions and memory barriers are ***expensive...***
- And are required for locks, which also impose deadlock, latency, ...
- RCU allows readers to avoid these expensive instructions.
- (Yes, one can just make ***all*** instructions expensive, but... Realtime???)

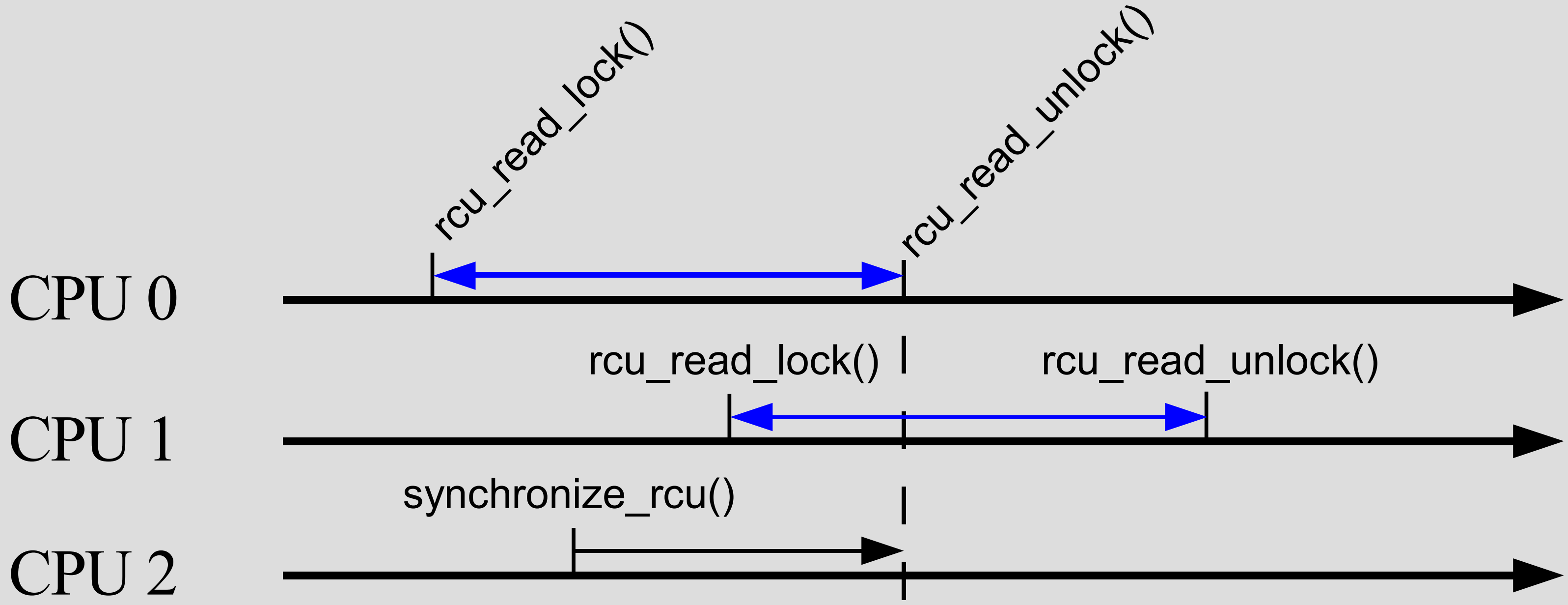
Introduction to RCU

- RCU is most often used as reader-writer lock
  - With ***very*** low-overhead (deterministic) readers
    - For non-CONFIG_PREEMPT:
    - #define rcu_read_lock()
    - #define rcu_read_unlock()
  - But readers run concurrently with writers
  - Writers must retain old versions: avoid trashing readers



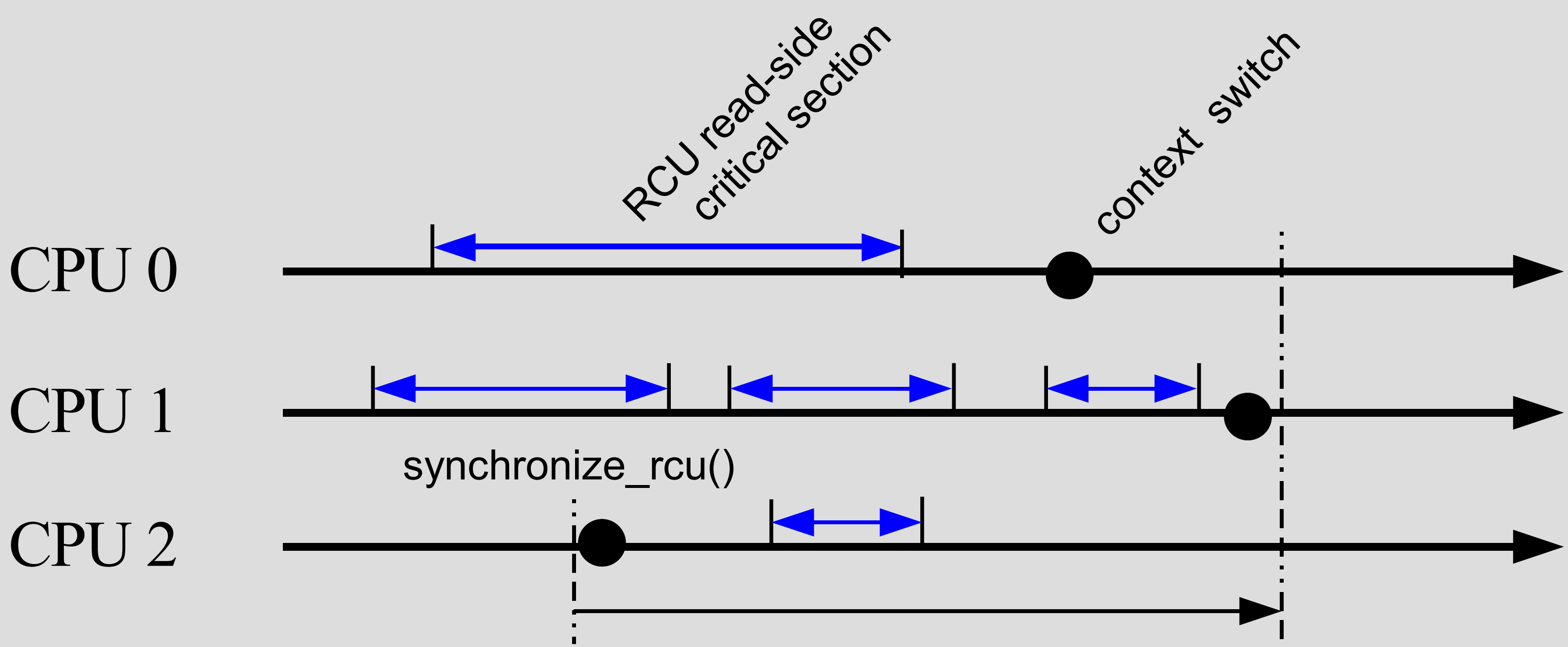
Introduction to RCU

- RCU is an API, with multiple implementations
  - rcu_read_lock() and rcu_read_unlock()
  - synchronize_rcu() and call_rcu()
  - rcu_assign_pointer() and rcu_dereference()
  - (There are ~20 additional non-core members of the RCU API)



Introduction to RCU

- Multiple RCU implementations
  - "Classic RCU" leverages context switches
    - RCU read-side critical sections not permitted to block
    - Therefore, context switch means all RCU readers on that CPU done
    - Once all CPUs context-switch, ***all*** prior RCU readers are done
  - Realtime RCU implementations presented on later slides



RCU and Reader-Writer Locking

```
int search(long key, int *result)
{
  struct list_head *lp;
  struct el *p;

  read_lock();
  list_for_each_entry(p, head, lp) {
    if (p->key == key) {
      *result = p->data;
      read_unlock();
      return 1;
    }
  }
  read_unlock();
  return 0;
}
```

```
int search(long key, int *result)
{
  struct list_head *lp;
  struct el *p;

  rcu_read_lock();
  list_for_each_entry_rcu(p, head, lp) {
    if (p->key == key) {
      *result = p->data;
      rcu_read_unlock();
      return 1;
    }
  }
  rcu_read_unlock();
  return 0;
}
```

RCU and Reader-Writer Locking

```
int delete(long key)
{
  struct el *p;

  write_lock(&listmutex);
  list_for_each_entry(p, head, lp) {
    if (p->key == key) {
      list_del(&p->list);
      write_unlock(&listmutex);
      kfree(p);
      return 1;
    }
  }
  write_unlock(&listmutex);
  return 0;
}
```

```
int delete(long key)
{
  struct el *p;

  spin_lock(&listmutex);
  list_for_each_entry(p, head, lp) {
    if (p->key == key) {
      list_del_rcu(&p->list);
      spin_unlock(&listmutex);
      synchronize_rcu();
      kfree(p);
      return 1;
    }
  }
  spin_unlock(&listmutex);
  return 0;
}
```

But note that RCU allows search and delete to run concurrently!
Not all algorithms permit this: in theory can transform, but hurts performance.

Other Uses for RCU

- Determine when all pre-existing SMIs/NMIs have completed
- Determine when all pre-existing irq handlers have completed
  - But -rt version of this needs work
  - Because -rt's irq handlers can be preempted
    - Thomas Gleixner has a fix for this
- Determine when all current readers have detected a change in mode
  - SRCU uses synchronize_sched() in this way
- Force each CPU to execute an smp_mb()
  - SRCU uses synchronize_sched() in this way

Realtime Response and RCU

Realtime Response and RCU

- What are realtime's requirements on RCU?
  - 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
- ***Italics*** == trouble for Classic RCU
  - Because it suppresses preemption.
    - Which is really bad for realtime scheduling latency!!!
  - But otherwise you get limitless grace periods: OOM!!!

Realtime Response and RCU

Key:
- "n": undesireable
- "N": disqualifies from some situations
- "X": immediate and total disqualification

	Reliable	Callable From IRQ	Preemptible Read Side	Small Memory Footprint	Sync-Free Reads	Indpt of Memory Blocks	Nestable Read Side	Uncond R-W Upgrade	Compatible API
Classic RCU			N	N					
rcu-preempt patch (ca. 2002)				X	N				
Jim Houston Patch			N		N				
Reader-Writer Locking					N		X	N	n
Unconditional Hazard Pointers				X	n	N			
Hazard Pointers: Failure				n	n	N			X
Hazard Pointers: Panic	X			n	n	N			
Hazard Pointers: Blocking		X		n	n	N			
Per-Object Reference Counters				N	n	N			
rcu_donereference()					n	N			X
Lock-Based Deferred Free					N				
Read-Side Counter GP Suppression				N	n				
2.6.17-rt7 RCU					n				

Realtime Response and RCU

- The rest of this presentation looks at ways of improving 2.6.17-rt5 RCU
  - Reduce read-side overhead
  - Improve grace-period detection scalability
  - Improve callback throughput/latency
  - Lower per-structure memory overhead
  - Boost priority of RCU read-side critical sections
    - For example, when preempted or waiting on a lock

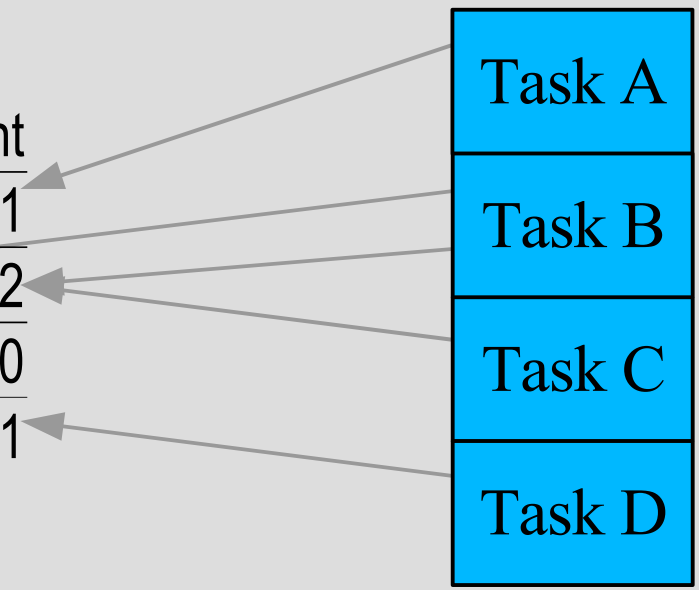
Realtime Read-Side Overhead

Realtime Read-Side Overhead

- -rt: atomic instructions and memory barriers
- optatomic: no atomics if no preemption
- optmb: no memory barriers if no preemption
- nonatomic: never atomic or memory barriers
  - Still working on stability and on performance
- For comparison: CONFIG_PREEMPT and non-CONFIG_PREEMPT

Realtime RCU Counters

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

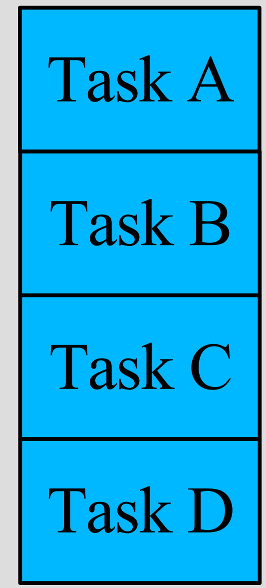


Each task references the counter that it incremented in rcu_read_lock(), allowing rcu_read_unlock() to decrement it (or them).

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

Realtime RCU Animated

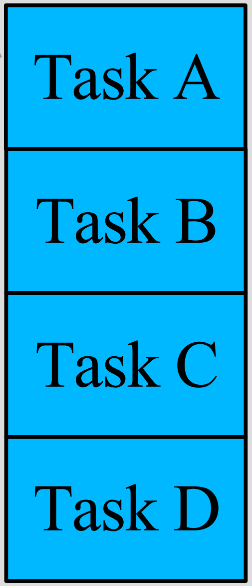
	Previous Count	Current Count
CPU 0	0	0
CPU 1	0	0
CPU 2	0	0
CPU 3	0	0



Initial state.

Realtime RCU Animated

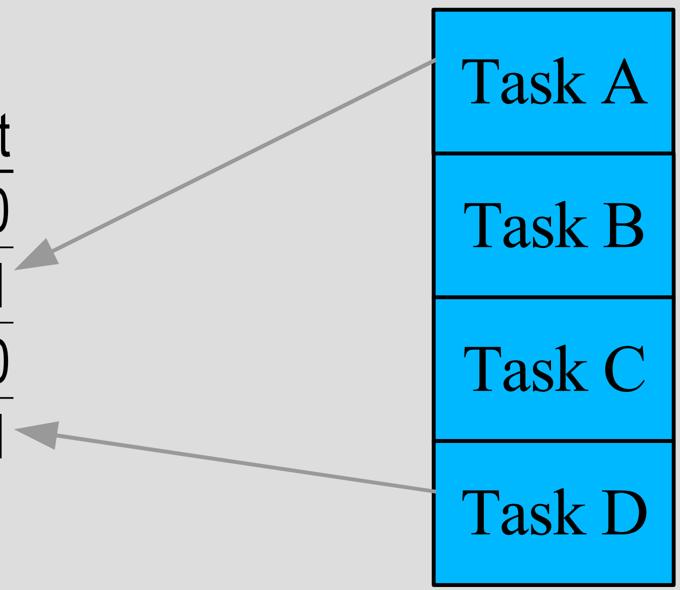
	Previous Count	Current Count
CPU 0	0	0
CPU 1	0	1
CPU 2	0	0
CPU 3	0	0



Task A rcu_read_lock().

Realtime RCU Animated

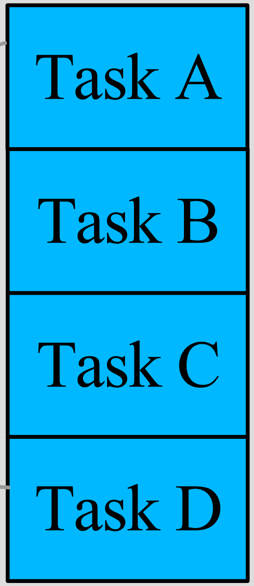
	Previous Count	Current Count
CPU 0	0	0
CPU 1	0	1
CPU 2	0	0
CPU 3	0	1



Task D rcu_read_lock().

Realtime RCU Animated

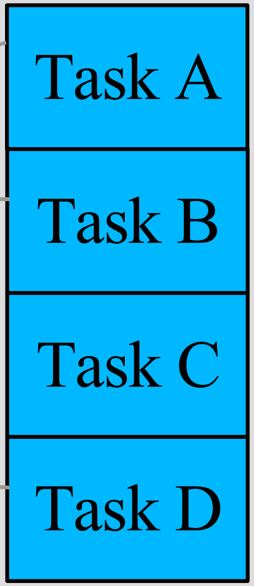
	Previous Count	Current Count
CPU 0	0	0
CPU 1	1	0
CPU 2	0	0
CPU 3	1	0



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

Realtime RCU Animated

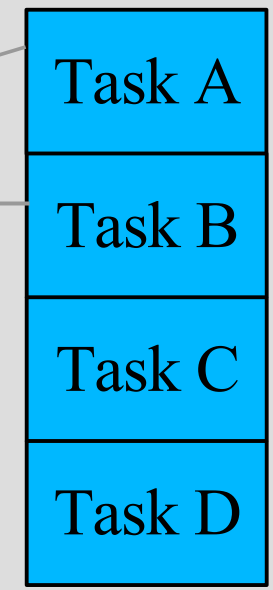
	Previous Count	Current Count
CPU 0	0	1
CPU 1	1	0
CPU 2	0	0
CPU 3	1	0



Task B rcu_read_lock().

Realtime RCU Animated

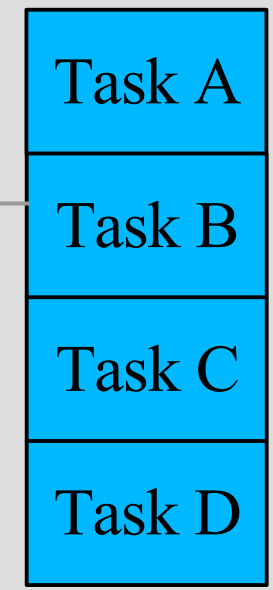
	Previous Count	Current Count
CPU 0	0	1
CPU 1	1	0
CPU 2	0	0
CPU 3	0	0



Task D rcu_read_unlock().

Realtime RCU Animated

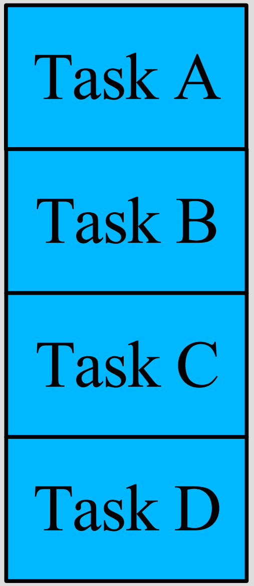
	Previous Count	Current Count
CPU 0	0	1
CPU 1	0	0
CPU 2	0	0
CPU 3	0	0



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

Realtime RCU Animated

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



Task B rcu_read_unlock().

But what issues are we failing to consider?

Other Realtime RCU Issues

- Memory barriers!
- Concurrent rcu_read_lock() and synchronize_rcu()
  - What if counter-roles flip races with increment?
- Concurrent rcu_read_lock() and earlier rcu_read_unlock() that is now on other CPU?
- IRQ handler doing rcu_read_lock() after interrupting RCU read-side critical section?
- And so on...

2.6.17-rt5 rcu_read_lock()

```
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);
}
```

2.6.17-rt5 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);
}
```

2.6.17-rt5 RCU Read Side

- 172 ns on 700 MHz i386: could do better.
  - Atomic operations and memory barriers!!!
- But both rcu_read_lock() and rcu_read_unlock() disable preemption.
  - If rcu_read_lock() sees zero in its CPU's current counter, no one else can possibly change it.
  - If rcu_read_unlock() sees a value of one in a counter that it is to decrement, no one else can possibly change it.
- Optimization: Don't use atomic operations in this case.

“optatomic” rcu_read_lock()

```
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]);
    current->rcu_read_lock_cpu = smp_processor_id();
    if (atomic_read(current->rcu_flipctr1) == 0) {
      atomic_set(current->rcu_flipctr1,
           atomic_read(current->rcu_flipctr1) + 1);
      smp_mb();
    } else {
      atomic_inc(current->rcu_flipctr1);
      smp_mb__after_atomic_inc();  /* will 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);
}
```

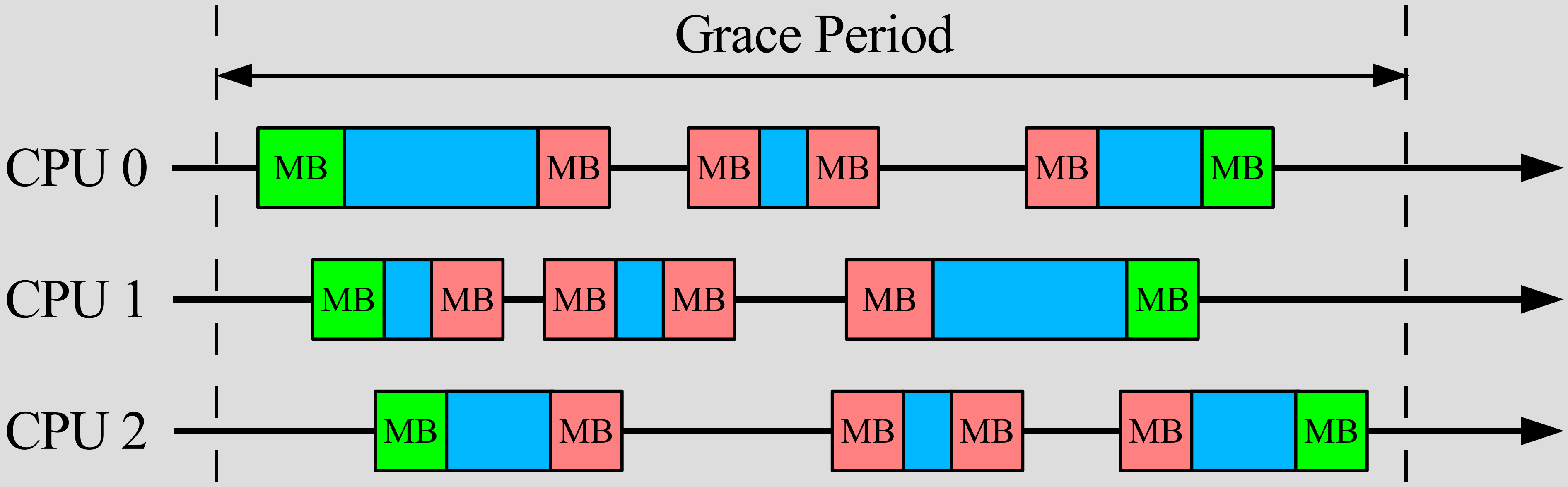
“optatomic” rcu_read_unlock()

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

  local_irq_save(oldirq);
  if (--current->rcu_read_lock_nesting == 0) {
    if ((atomic_read(current->rcu_flipctr1) == 1) &&
        (current->rcu_read_lock_cpu == smp_processor_id())) {
      smp_mb();
      atomic_set(current->rcu_flipctr1,
           atomic_read(current->rcu_flipctr1) - 1);
    } else {
      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);
}
```

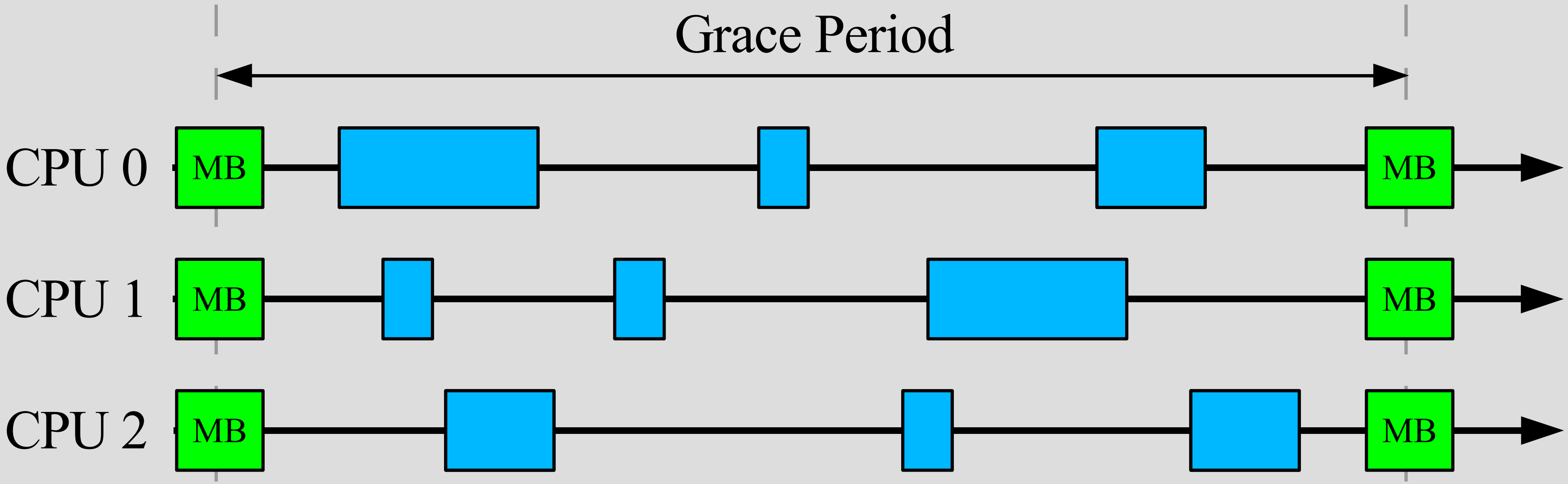
“optatomic” Read Side

- 232 ns on 700 MHz i386: got worse!!!
  - Because i386 memory barriers are atomics...
- ***Really*** need to get rid of the memory barriers
  - Because most are unneeded anyway!
  - Incorporate into grace-period processing...



“optatomic” Update Side

- Associate the required memory barriers with grace-period processing
  - Less common than read-side critical sections
  - Gross simplifications in diagram below



“optmb” rcu_read_lock()

```
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]);
    current->rcu_read_lock_cpu = smp_processor_id();
    if (atomic_read(current->rcu_flipctr1) == 0) {
      atomic_set(current->rcu_flipctr1,
           atomic_read(current->rcu_flipctr1) + 1);
    } else {
      atomic_inc(current->rcu_flipctr1);
    }
    if (unlikely(flipctr != (rcu_ctrlblk.completed & 0x1))) {
      current->rcu_flipctr2 =
        &(__get_cpu_var(rcu_flipctr)[!flipctr]);
      /* Can again optimize to non-atomic on fastpath. */
      atomic_inc(current->rcu_flipctr2);
    }
  }
  local_irq_restore(oldirq);
}
```

“optmb” rcu_read_unlock()

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

  local_irq_save(oldirq);
  if (--current->rcu_read_lock_nesting == 0) {
    if ((atomic_read(current->rcu_flipctr1) == 1) &&
        (current->rcu_read_lock_cpu == smp_processor_id())) {
      atomic_set(current->rcu_flipctr1,
           atomic_read(current->rcu_flipctr1) - 1);
    } else {
      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);
}
```

“optmb” Read Side

- 115 ns on 700 MHz i386: improvement!
- But code path is still long and slow
  - Want to get rid of ***all*** mb()s and atomics from read-side primitives
  - Use nested grace periods to simplify read-side!
    - After flipping the roles of the counters, wait until all CPUs acknowledge the flip: eliminate races
      - A nested grace period
    - Retain memory barriers in grace-period handling
      - But grace period now becomes “fuzzy”
      - Must wait for two grace periods rather than one

“nonatomic” rcu_read_lock()

```
void rcu_read_lock(void)
{
  int idx;
  int nesting;
  unsigned long oldirq;

  local_irq_save(oldirq);
  nesting = current->rcu_read_lock_nesting;
  if (nesting != 0) {
    current->rcu_read_lock_nesting = nesting + 1;
  } else {
    idx = rcu_ctrlblk.completed & 0x1;
    smp_read_barrier_depends();
    barrier();
    __get_cpu_var(rcu_flipctr)[idx]++;
    barrier();
    current->rcu_read_lock_nesting = nesting + 1;
    barrier();
    current->rcu_flipctr_idx = idx;
  }
  local_irq_restore(oldirq);
}
```

Note: handles rcu_read_lock() from within NMI/SMI handlers

“nonatomic” rcu_read_unlock()

```
void rcu_read_unlock(void)
{
  int idx;
  int nesting;
  unsigned long oldirq;

  local_irq_save(oldirq);
  nesting = current->rcu_read_lock_nesting;
  if (nesting > 1) {
    current->rcu_read_lock_nesting = nesting - 1;
  } else {
    idx = current->rcu_flipctr_idx;
    smp_read_barrier_depends();
    barrier();
    current->rcu_read_lock_nesting = nesting - 1;
    barrier();
    __get_cpu_var(rcu_flipctr)[idx]--;
  }
  local_irq_restore(oldirq);
}
```

“nonatomic” Read Side

- 94 ns on 700 MHz i386: ***much*** better!
  - But still a factor of nine slower than CONFIG_PREEMPT implementation of RCU...
- Next steps:
  - Integrate CPU hotplug, need that now...
  - Get rid of the interrupt disabling: major source of overhead at the moment
  - And maybe get rid of preemption disabling as well, though this might not be possible
    - Would like to dump the task-local increment, but it is needed in order to priority-boost RCU read-side tasks
    - Might be able to fold into priority disabling...

Realtime Read-Side Overhead

Kernel	n	ns	Std
2.6.15-rt16	92	172.02	0.22
2.6.15-rt16 optatomic	131	232.06	0.35
2.6.15-rt16 optmb	84	115.09	0.08
2.6.15-rt16 nonatomic	20	93.89	0.16
2.6.15 CONFIG_PREEMPT	393	10.87	0.06
2.6.15 non-CONFIG_PREEMPT	61	0.63	0.06

- Good news: well over halfway to CONFIG_PREEMPT.
- Bad news: almost an order of magnitude still to go.
- May be able to reduce further by removing local_irq_disable().

RCU Callback Throughput and Latency

RCU Callback Throughput and Latency

- Callback scheduling priority and batching
- SLAB_DESTROY_BY_RCU
  - Example: Christoph Lameter's struct-file patch
    - Greatly reduces the number of call_rcu() invocations
    - But requires read-side checks
- Self-limiting updates:
  - limiting number of call_rcu()s in flight
  - limiting update rate
  - update by trusted person
  - call_rcu_bh()
  - synchronize_rcu()

Per-Struct Callback Overhead

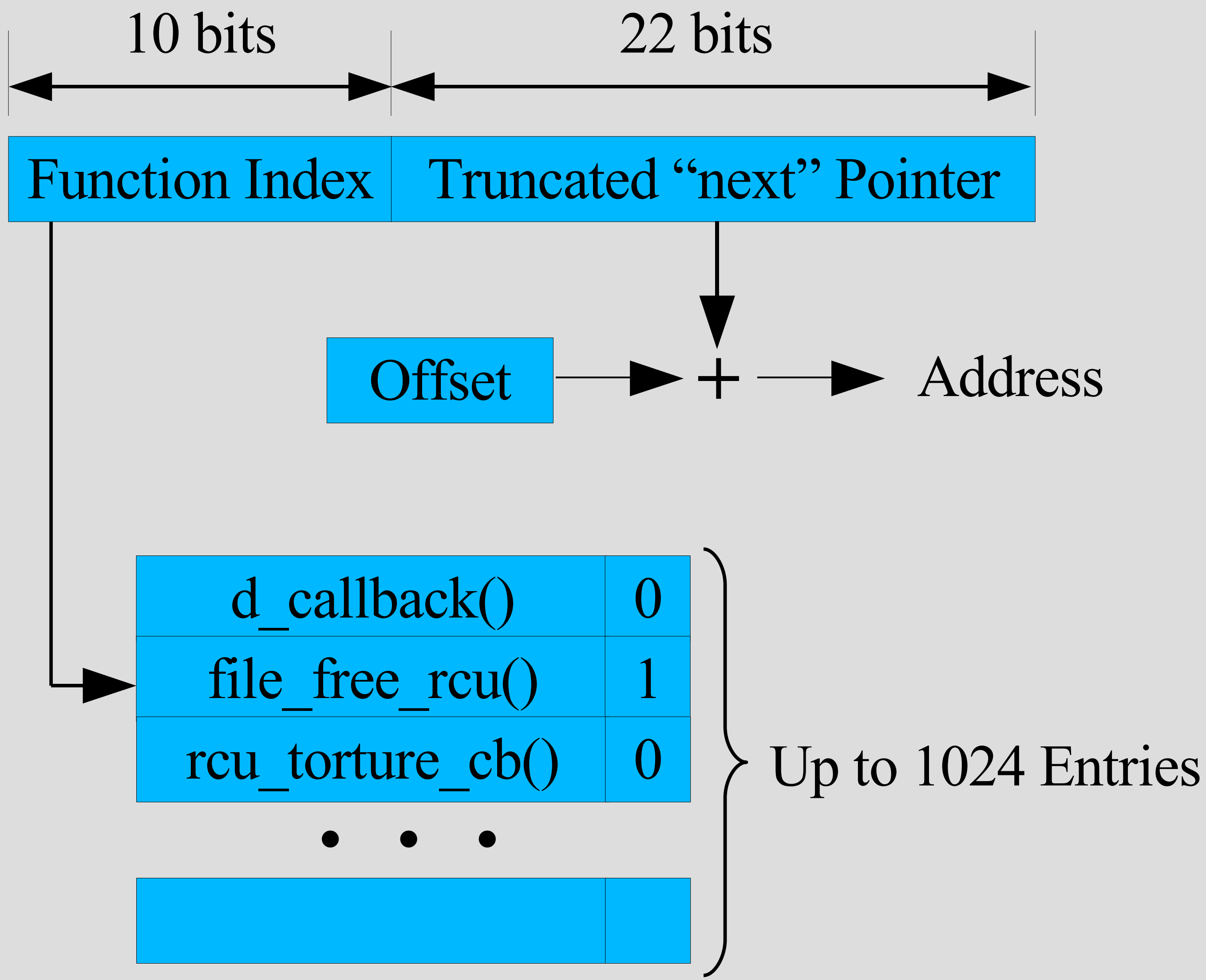
Per-Struct Callback Overhead

- Some people want to use Linux 2.6 kernels on extremely small systems.
  - 2 MB (yes, ***megabytes***) of physical memory.
  - The 8-byte overhead of struct rcu_head is a concern for these small systems.
  - Can we make things better for Linux on tiny embedded systems?

Per-Struct Callback Overhead

- Possible approaches:
  1. Use synchronize_rcu() rather than call_rcu()
     - gives self-limiting property to updates
     - but can result in update bottleneck
  2. Use “union” to hide rcu_head overhead
     - must union with fields that are not used after removal
     - great when it works, but not always possible
  3. Shrink rcu_head structure by mapping functions
     - works on small machines (<16 MB RAM)
       - limits the number of RCU callback functions
       - only saves half of the rcu_head
       - requires a table to map function index to pointer (see next slide)

Per-Struct Callback Overhead



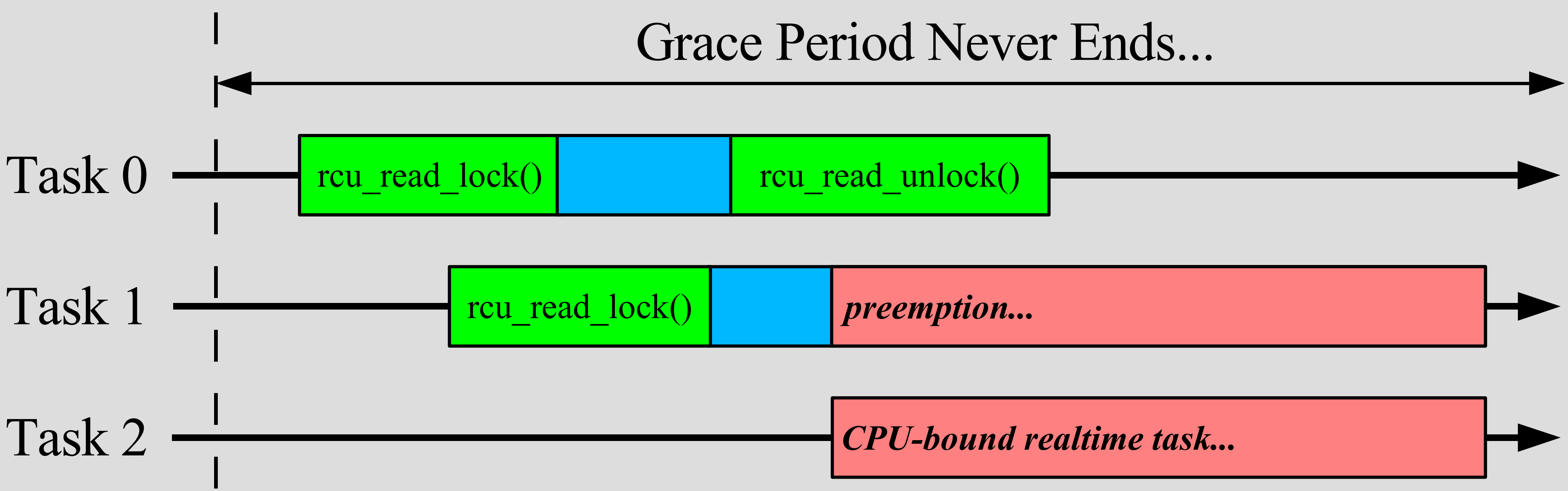
Per-Struct Callback Overhead

- The first two seem preferable:
  - Use of synchronize_rcu() and the union save eight bytes rather than just four
  - They don't limit the addressing or the number of callback function
- But people interested in extremely small systems might wish to experiment with the squeezed-down struct rcu_head

RCU Read-Side Priority Boost

RCU Read-Side Priority Boost

- Problem: RCU read-side critical sections can be preempted by CPU-bound realtime tasks
  - Halts grace periods, results in OOM



RCU Read-Side Priority Boost

- “Correct” solution: Don't code realtime tasks to be CPU-bound
  - CPU-bound high-priority realtime tasks will prevent any lower-priority realtime tasks from meeting their deadlines
    - Possible exception: tight loop on one CPU, everything else on other CPUs
    - But -rt currently not structured to support this
  - But OOMing in response to a user-level bug is socially irresponsible: tough to debug
- Real solution: allow RCU read-side critical sections to be priority boosted
  - When and how to boost priority?

When to Boost? How High?

- Nonsensical to boost in rcu_read_lock()
  - High overhead
  - Unnecessary in most cases: priority only matters when you are ***not*** running!
- Usually doesn't make sense to boost non-realtime tasks to realtime priorities
  - Unless low on memory: error condition
  - Could maintain a list of candidates for a second boost
- Challenge: race between boosting priority and rcu_read_unlock()

Sleepable RCU

Sleepable RCU

- Problem: RCU read-side sleep forbidden
  - Restricted exceptions in -rt
    - preemption and blocking for mutex, which can in principle be awakened via priority boosting
  - Reason: read-side sleeping can OOM
- Solution: per-subsystem grace periods
  - Each subsystem keeps a “struct srcu_struct”:
    - init_srcu_struct(&s), cleanup_srcu_struct(&s)
  - Read side must keep track of index:
    - idx = srcu_read_lock(&s); ... srcu_read_unlock(&s, idx);
  - Update side uses synchronize_srcu(&s)
  - No call_srcu() -- self-throttling update enforced
  - Sleeping read side holds up only its own updates

SRCU API

- void init_srcu_struct(struct srcu_struct *sp);
- void cleanup_srcu_struct(struct srcu_struct *sp);
- int srcu_read_lock(struct srcu_struct *sp);
- void srcu_read_unlock(struct srcu_struct *sp, int idx);
- void synchronize_srcu(struct srcu_struct *sp);
- long srcu_batches_completed(struct srcu_struct *sp);

SRCU Operation: Trick #1

- Variables “x” and “y” are initially both zero
- Task A:

```
for (;;) {
        b = y; barrier(); a = x;
        BUG_ON(b == 0 || a == 1);
}
```

- Task B:

```
x = 1;
synchronize_sched();
y = 1;
```

- Task A's assertion guaranteed ***not*** to fire

SRCU Operation: Trick #2

- Variables “x”, “y”, and “z” are initially both zero
- Task A:

```
for (;;) {
        c = z; barrier(); a = x;
        BUG_ON(c == 0 || a == 1);
```

- Task B:

```
x = 1;
synchronize_sched(); /* many smb_mb()s, etc. */
y = 1;
```

- Task C:

```
for (;;)
        if (y == 1) z == 1;
```

- Task A's assertion guaranteed ***not*** to fire

srcu_read_lock()

```
int srcu_read_lock(struct srcu_struct *sp)
{
  int idx;

  preempt_disable();
  idx = sp->completed & 0x1;
  barrier();  /* ensure compiler looks -once- at sp->completed. */
  per_cpu_ptr(sp->per_cpu_ref, smp_processor_id())->c[idx]++;
  srcu_barrier();  /* ensure compiler won't misorder critical section. */
  preempt_enable();
  return idx;
}
```

```
#ifndef CONFIG_PREEMPT
#define srcu_barrier() barrier()
#else /* #ifndef CONFIG_PREEMPT */
#define srcu_barrier()
#endif /* #else #ifndef CONFIG_PREEMPT */
```

srcu_read_unlock()

```
void srcu_read_unlock(struct srcu_struct *sp, int idx)
{
  preempt_disable();
  srcu_barrier();  /* ensure compiler won't misorder critical section. */
  per_cpu_ptr(sp->per_cpu_ref, smp_processor_id())->c[idx]--;
  preempt_enable();
}
```

synchronize_srcu()

```
void synchronize_srcu(struct srcu_struct *sp)
{
  int idx;

  idx = sp->completed;
  mutex_lock(&sp->mutex);

  if ((sp->completed - idx) >= 2) {
    mutex_unlock(&sp->mutex);
    return;
  }
  synchronize_sched();  /* Force memory barrier on all CPUs. */
  idx = sp->completed & 0x1;
  sp->completed++;
  synchronize_sched();  /* Force memory barrier on all CPUs. */
  while (srcu_readers_active_idx(sp, idx))
    schedule_timeout_interruptible(1);
  synchronize_sched();  /* Force memory barrier on all CPUs. */
  mutex_unlock(&sp->mutex);
}
```

Potential Uses of SRCU

- Notifier chains (see Alan Stern's patch)
- Possible latency fixes for reader-writer semaphores in -rt
- Possible way of waiting for preemptible irq handlers
  - However, there are other ways of fixing this
- But mostly just because people have been asking me for something like this for many more years than I care to admit to!!!

Conclusions

- Goal is to converge realtime RCU if at all possible (at least with CONFIG_PREEMPT)
  - Reduce testing/maintenance burden
- Significant progress possible on reducing struct rcu_head memory consumption
- SRCU available should there be latency issues with reader-writer semaphores
  - where readers ***must*** block
- Summary: RCU is still growing and evolving
  - More than a decade after Paul first thought it to be fully mature...

Legal Statement

- This work represents the view of the authors and does not necessarily represent the view of IBM or Red Hat.
- Linux is a registered trademark of Linus Torvalds.
- IBM, PowerPC, and POWER4+ are trademarks or registered trademarks of International Business Machines Corporation in the United States and/or other countries.
- Other company, product, and service names may be trademarks or service marks of others.