montavista VISION 2007

EMBEDDED LINUX DEVELOPERS CONFERENCE

Real Time Linux Technology

A Deeper Dive

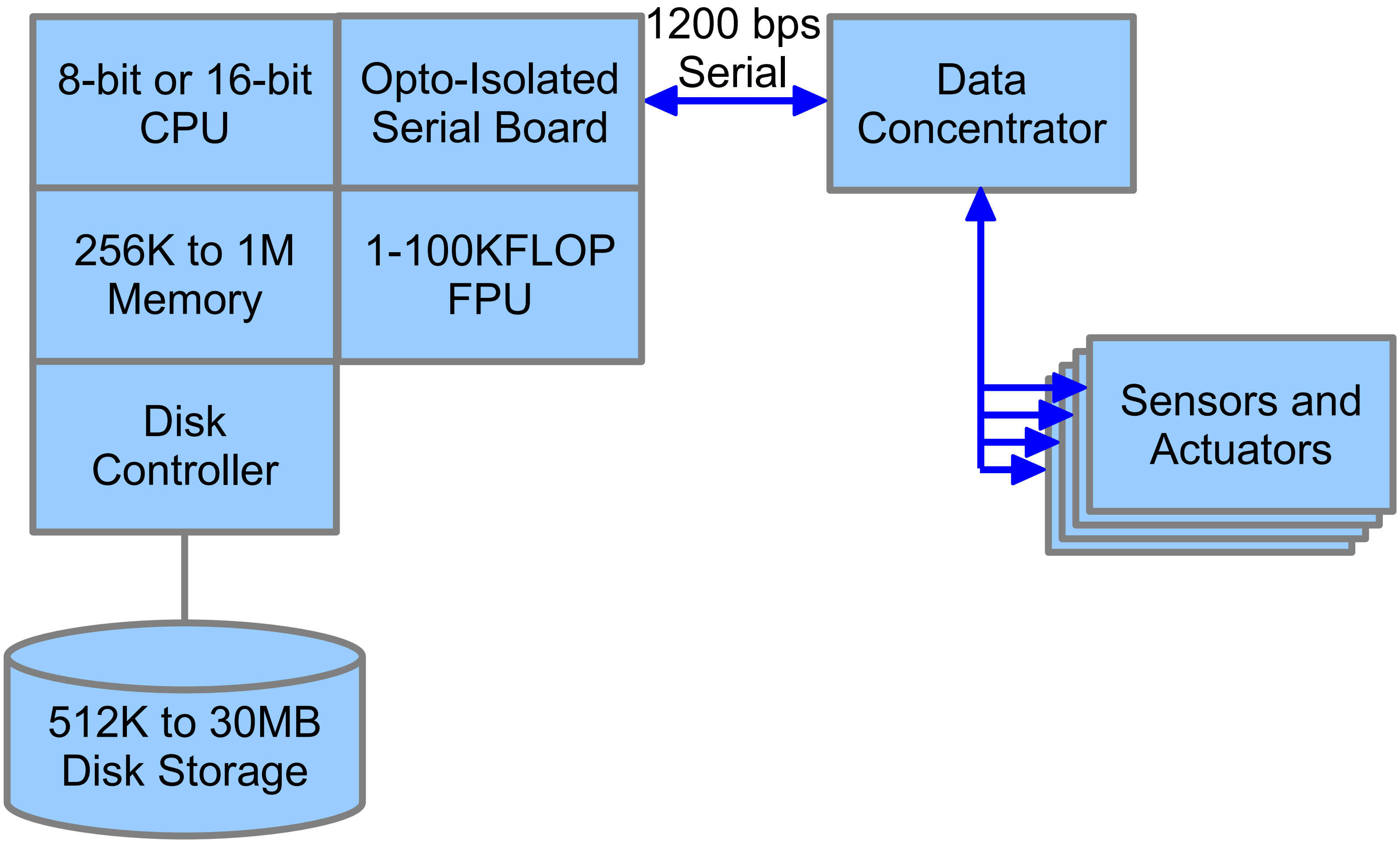
Paul E. McKenney

IBM Distinguished Engineer, Linux Technology Center

montavista

How I Got Here

Real Time Computing ca. 1980-1985

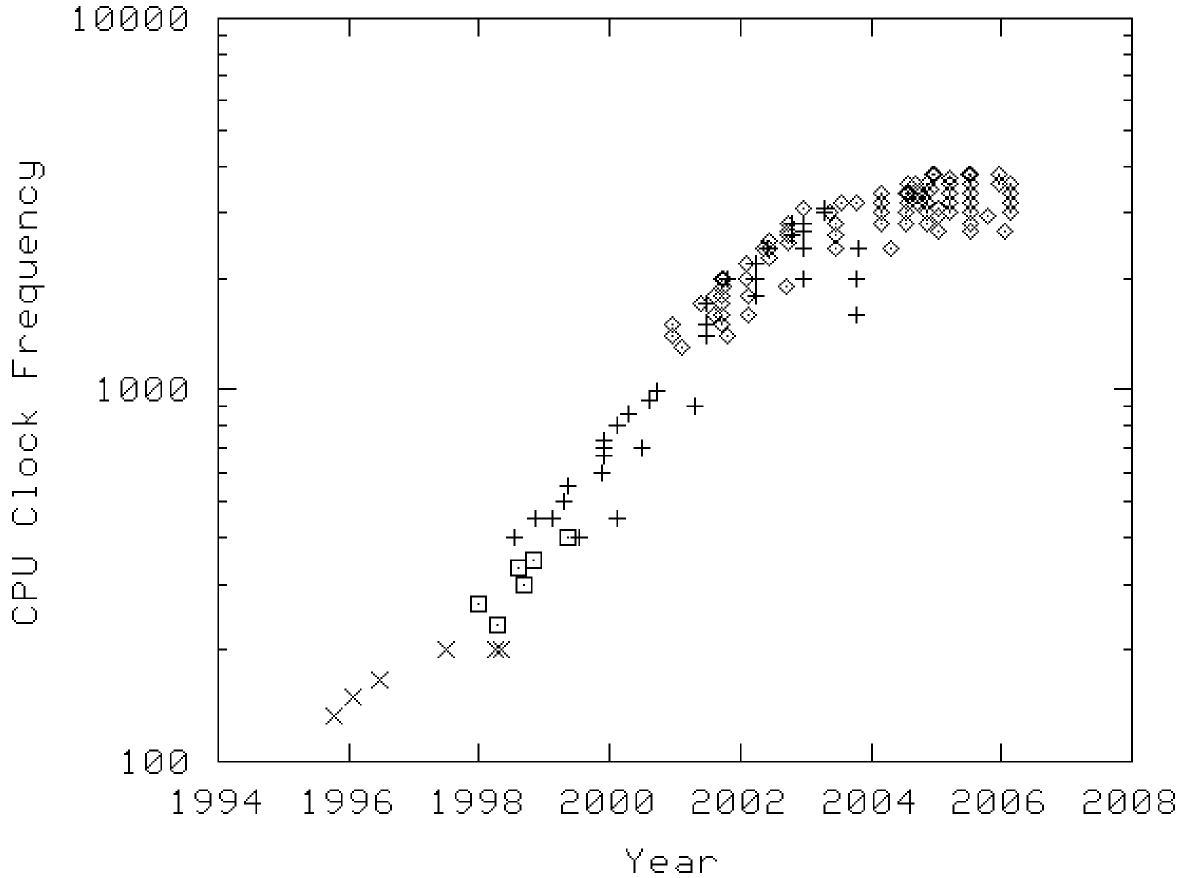


Non-Real-Time Interlude

- Systems administration (1986-8)
- Internet routing and congestion avoidance protocol (1988-1990)
- Parallel and NUMA algorithms, DYNIX/ptx, Digital Unix, AIX, Linux (1990-2004)
  - Some exposure to realtime via the MontaVista-lead PREEMPT effort interactions with RCU (2002-2004)
- Return to realtime:
  - Parallel realtime algorithms in Linux (2004-present)

Why Parallel Realtime?

Clock-Frequency Trend For Intel CPUs



Increased performance requires multiple hardware threads and multiple cores

Emergence of SMP Embedded Realtime Systems

Traditional Systems

Traditional Realtime:
Few CPUs
Latency Guarantees
Non-Standard

OR

Traditional SMP:
Many CPUs
No Guarantees
Standard (and OSS)

But Not Both!!!

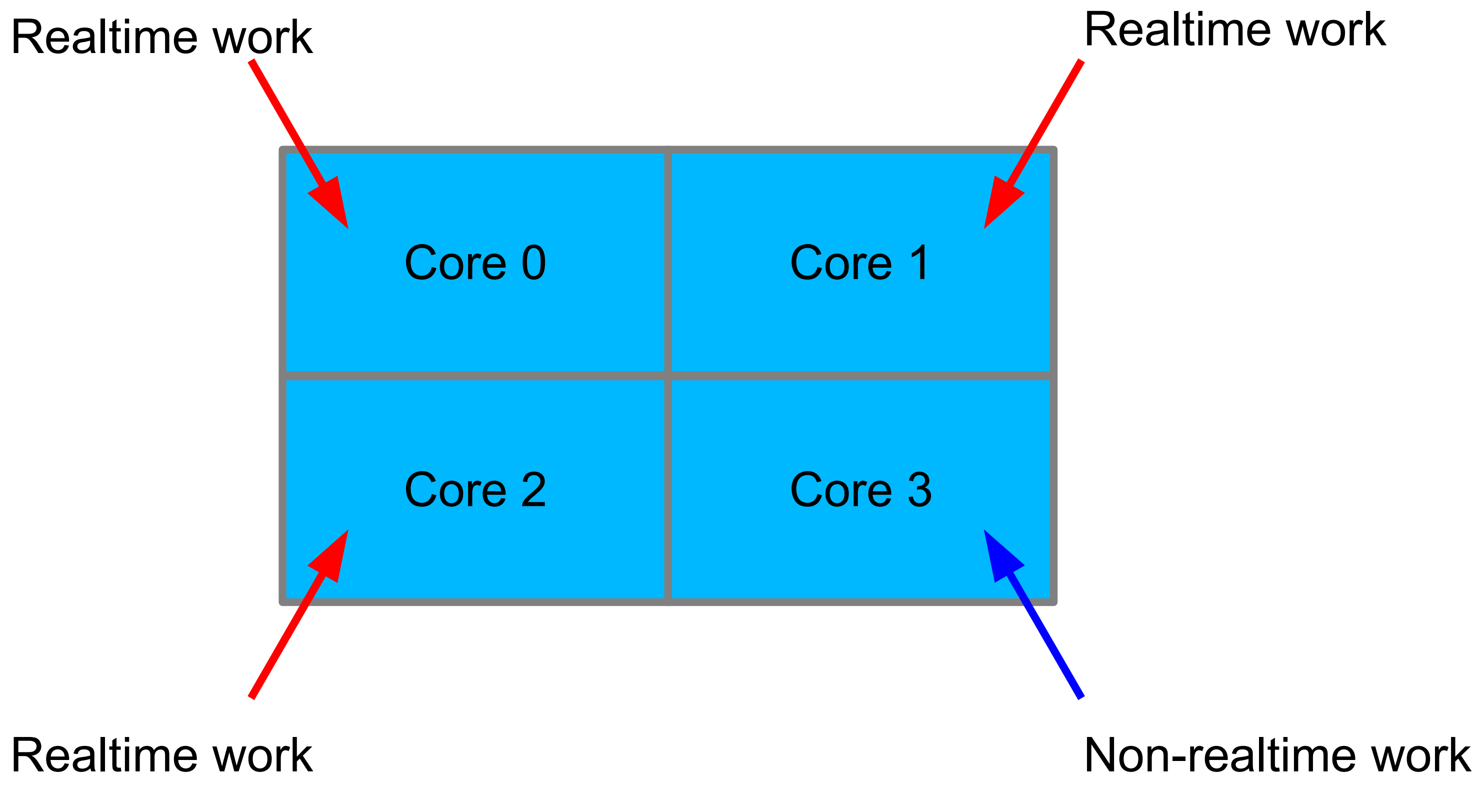
Convergence

Emerging Systems

SMP Realtime:
Many CPUs
Latency Guarantees
Standard (and OSS)

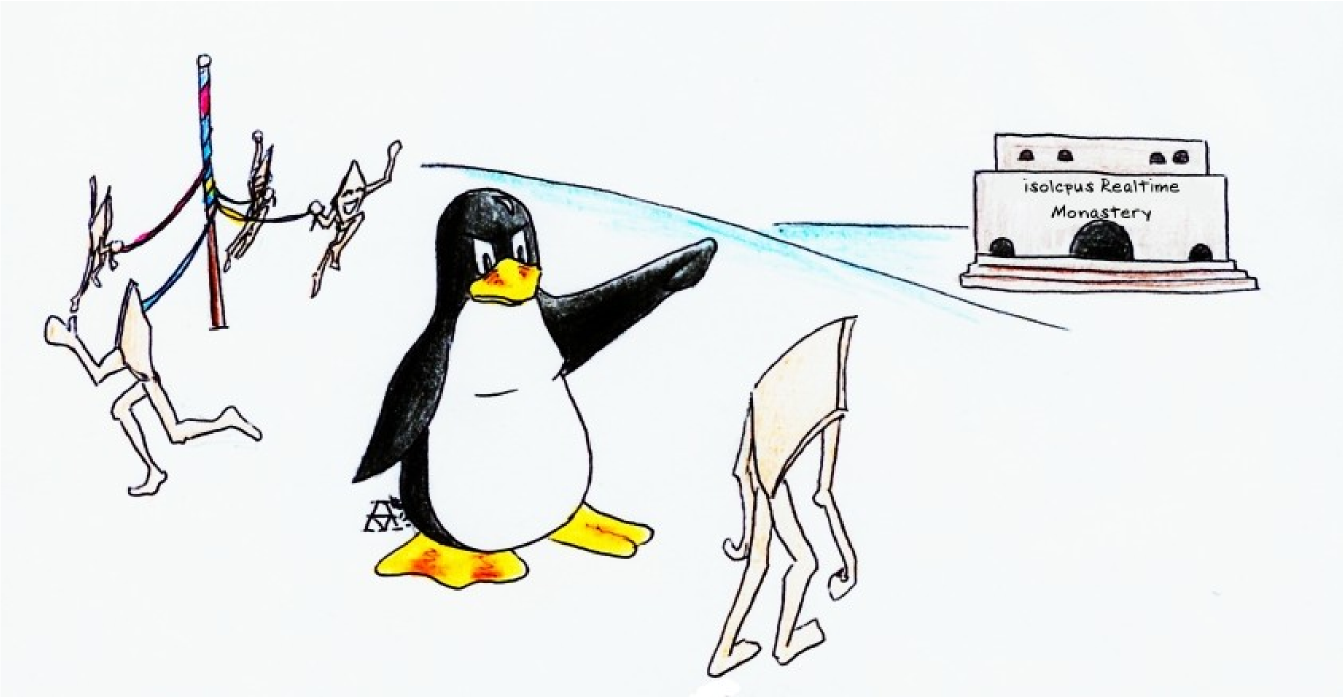
- User Demand (DoD, Financial, Gaming, ...)
- Techological Changes Leading to Commodity SMP
  - Commodity Hardware Multithreading
  - Commodity Multi-Core Dies
  - Tens to Hundreds of CPUs per Die – Or More

2004: Prototype Multi-Core *ARM* Chip!!!



Submitted simple patch to Linux-kernel mailing list in 2004...

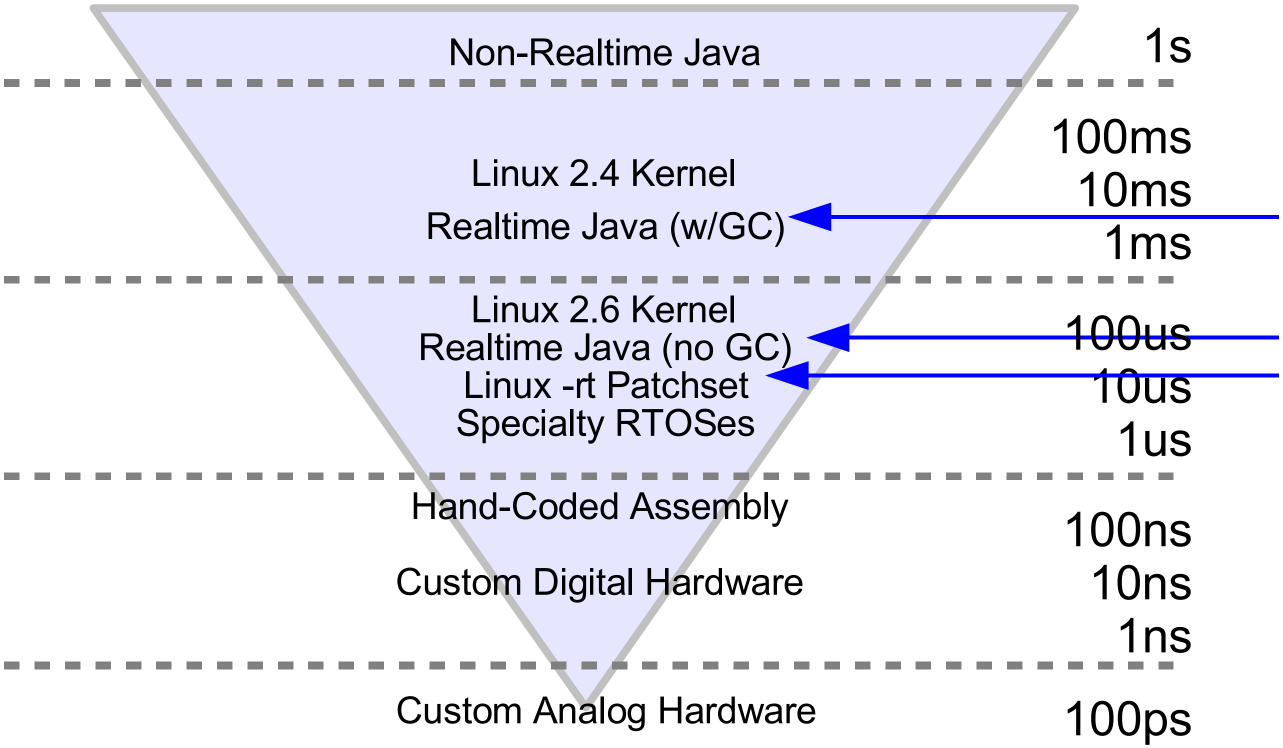
Leveraging Multiprocessor Systems for Realtime



Useful approach in many cases – but not so good if *all* CPUs must do realtime...

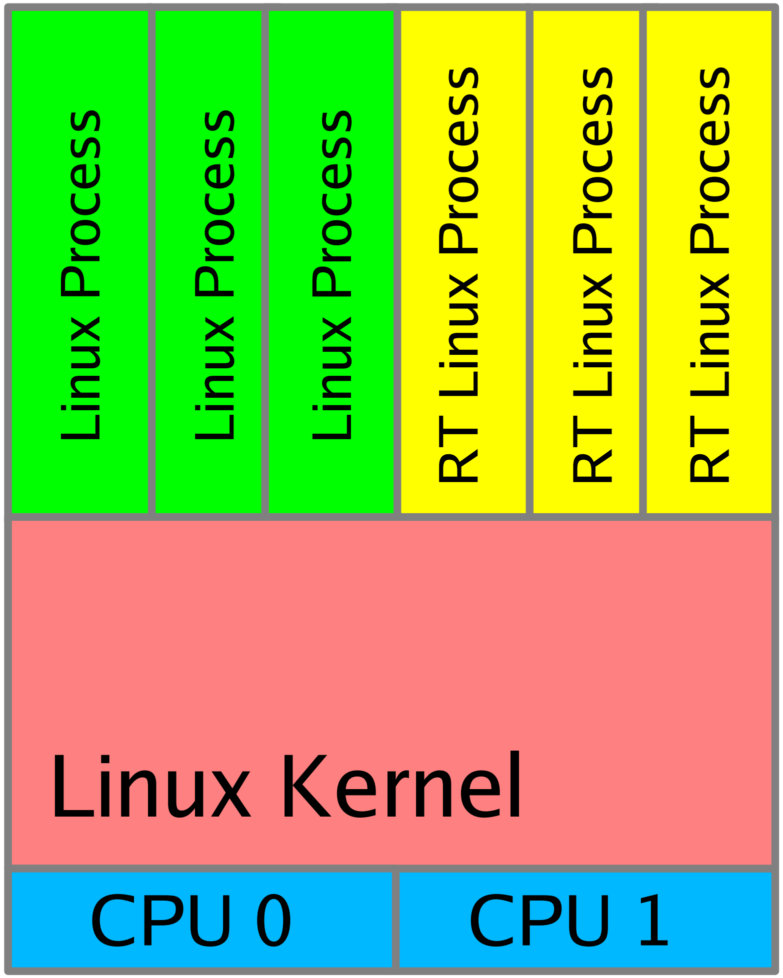
Real-Time Regimes

Real-Time Regimes

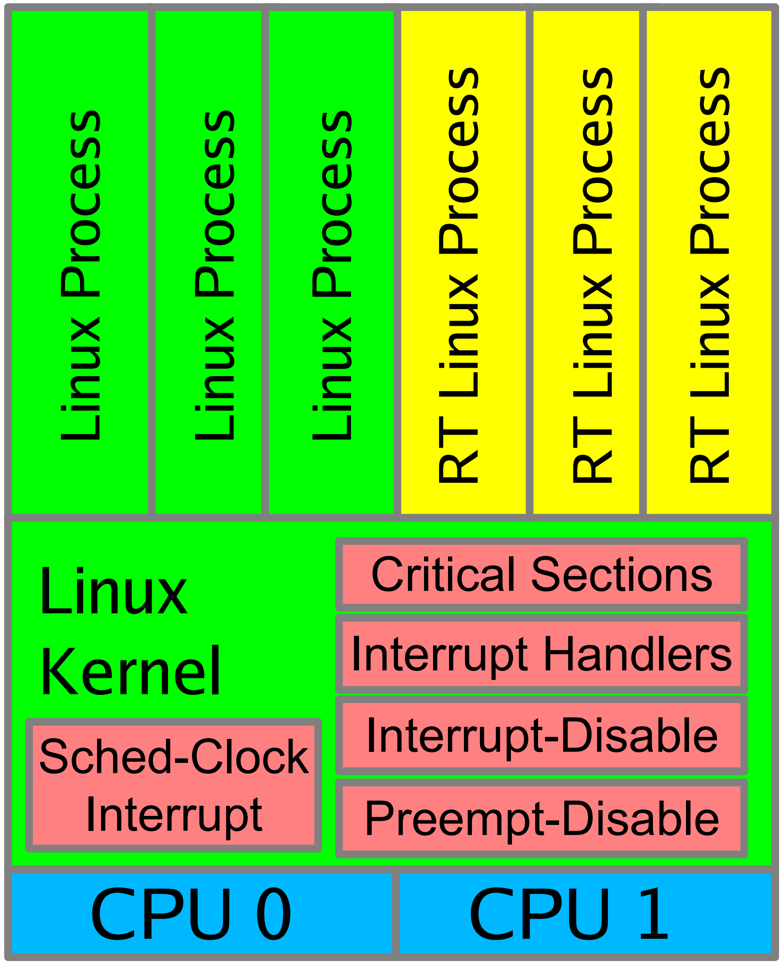


Preemption

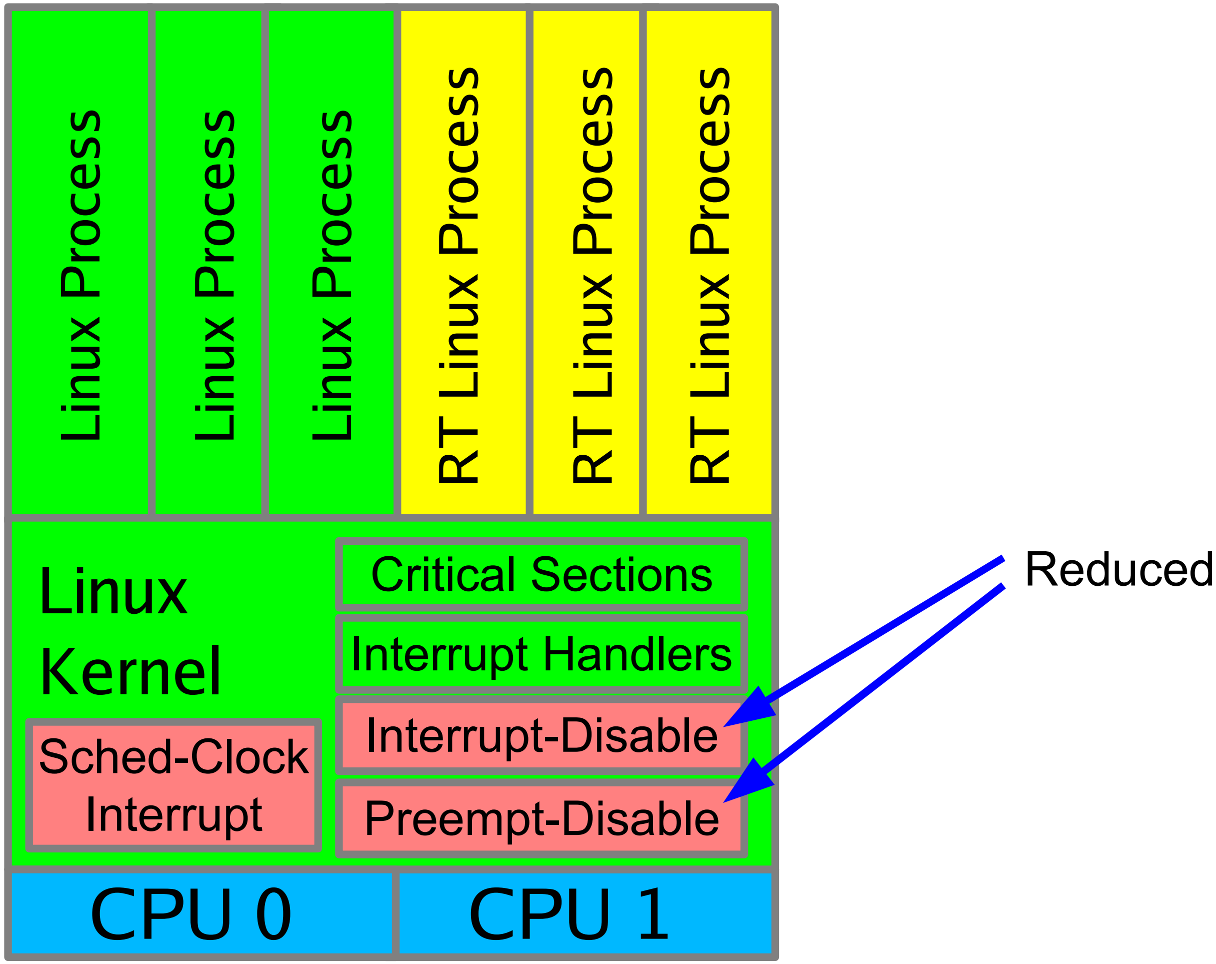
Vanilla Linux Kernel



Linux Kernel CONFIG_PREEMPT Build



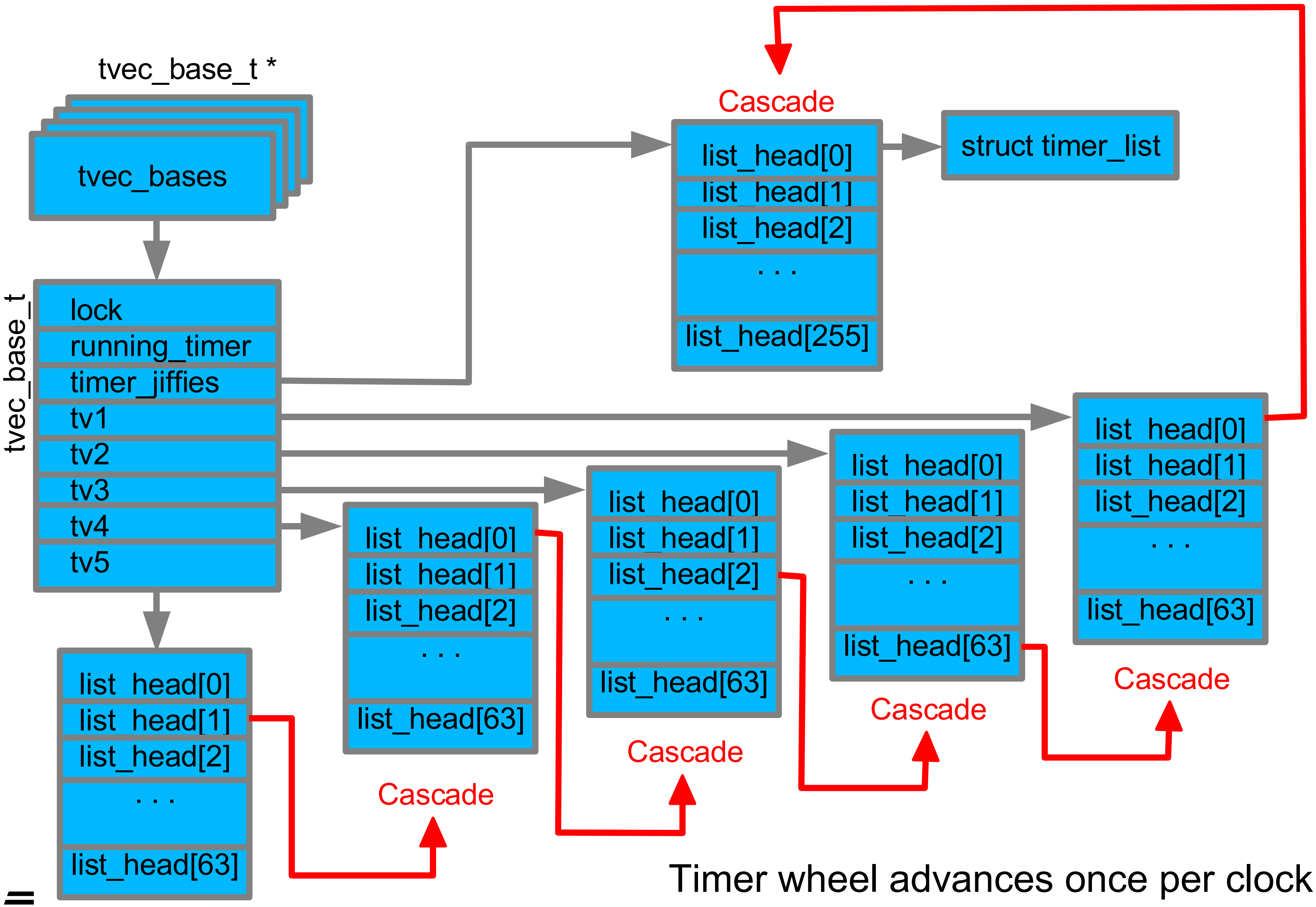
Linux Kernel CONFIG_PREEMPT Build



Preemptible Spinlocks

- Threads can be preempted while holding spinlocks
- Threads must therefore be permitted to block while acquiring spinlocks
  - Necessary to avoid self-deadlock scenario
- spinlock_t acquisition primitives can therefore block
- raw_spinlock_t provides "true spinlock" that disables preemption for special cases: scheduler, scheduling-clock interrupt
- Note that one uses the same primitives (e.g., spin_lock()) on both spinlock_t and raw_spinlock_t

Timers and -rt Patchset



Timer wheel advances once per clock tick

Timer Wheels: Advantages and Disadvantages

- Advantages:
  - O(1) insertion and removal operations
  - Batching of cascade operations improves throughput
  - Simple, well tested (both in Linux and elsewhere)
- Disadvantages:
  - Cascading operations ***major*** latency hit!!!
  - Unforgiving tradeoff between accuracy and overhead
- But when you need tens-of-microseconds latencies for some applications...

Linux Timer Wheel at 1KHz

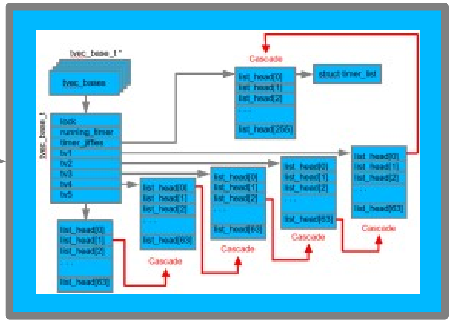
Linux Timer Wheel at 100KHz

Any Questions?

Solution: High-Resolution Timers

Timeouts: approximation OK, likely cancelled

add_timer(), mod_timer(), del_timer(), del_timer_sync(), ...



Timers: must be exact, rarely cancelled

hrtimer_init(), hrtimer_init_sleeper(), hrtimer_start(),
hrtimer_cancel(), hrtimer_forward(), ...

High-Resolution
Timers
Red-Black Tree

High-Resolution Timer API (1/2)

- hrtimer_init(timer, clock_id, mode)
  - timer: already-allocated struct hrtimer to use
  - clock_id: usually want CLOCK_MONOTONIC (***not*** CLOCK_REALTIME)
  - mode: HRTIMER_MODE_ABS or HRTIMER_MODE_REL
    - Note: if HRTIMER_MODE_REL, CLOCK_REALTIME treated as CLOCK_MONOTONIC
- hrtimer_init_sleeper(sl, task)
  - sl: already-allocated and hrtimer_init()ed hrtimer_sleeper to use
    - hrtimer_sleeper is a struct containing an hrtimer and a pointer to task_struct
  - task: task to be awakened upon timer expiry (sl->timer.function overridden)
- hrtimer_start(timer, tim, mode)
  - timer: hrtimer to start
  - tim: expiration time in ktime_t format
  - mode: absolute or relative (HRTIMER_MODE_ABS or HRTIMER_MODE_REL)
- hrtimer_cancel(timer)
  - timer: hrtimer to cancel – waits for the timer to finish if it has already fired
- hrtimer_try_to_cancel(timer) – as hrtimer_cancel(), fail if already fired

High-Resolution Timer API (2/2)

- hrtimer_forward(timer, now, interval)
  - timer: hrtimer to rearm in future
  - now: current time (from which the notion of “future” will be derived)
  - interval: time interval from time of last timer expiration
  - returns number of intervals required to get to future
- hrtimer_get_remaining(timer)
  - timer: timer for which to return remaining wait time
- hrtimer_get_next_event()
  - return nanoseconds to next timer expiry – useful for power-savings decisions
- ktime_get() -- get current time (ns), compatible with above APIs
- ktime_add_ns(kt, nsec) – arithmetic on nanosecond timestamps.
- hrtimer_get_res(which_clock, tp)
  - which_clock: CLOCK_MONOTONIC or CLOCK_REALTIME
  - tp: struct timespec into which to put resolution

High-Resolution Timer API Example Usage

- From futex_wait():

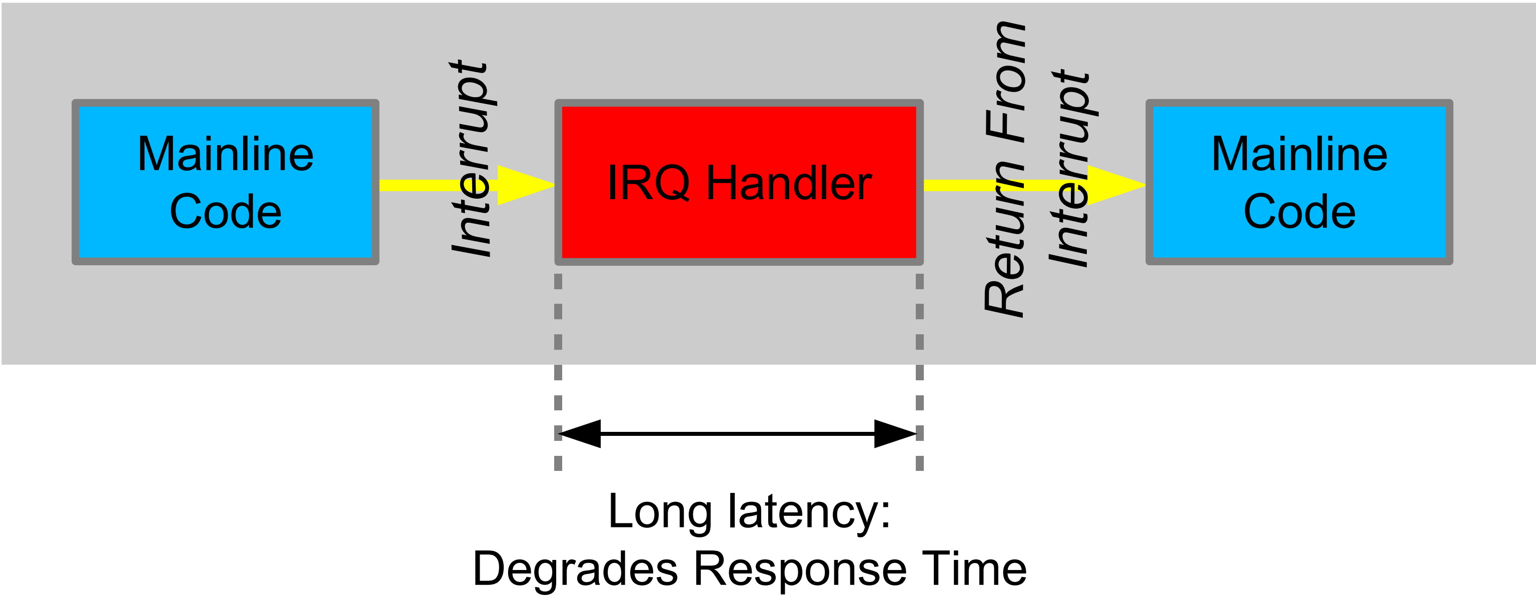
```
__set_current_state(TASK_INTERRUPTIBLE);
add_wait_queue(&q.waiters, &wait);
...
hrtimer_init(&t.timer, CLOCK_MONOTONIC, HRTIMER_MODE_ABS);
hrtimer_init_sleeper(&t, current);
t.timer.expires = *abs_time;

hrtimer_start(&t.timer, t.timer.expires, HRTIMER_MODE_ABS);

/*
 * the timer could have already expired, in which
 * case current would be flagged for rescheduling.
 * Don't bother calling schedule.
 */
if (likely(t.task))
        schedule();

hrtimer_cancel(&t.timer);

/* Flag if a timeout occured */
rem = (t.task == NULL)
```

Timers and -rt Patchset: To Probe Deeper

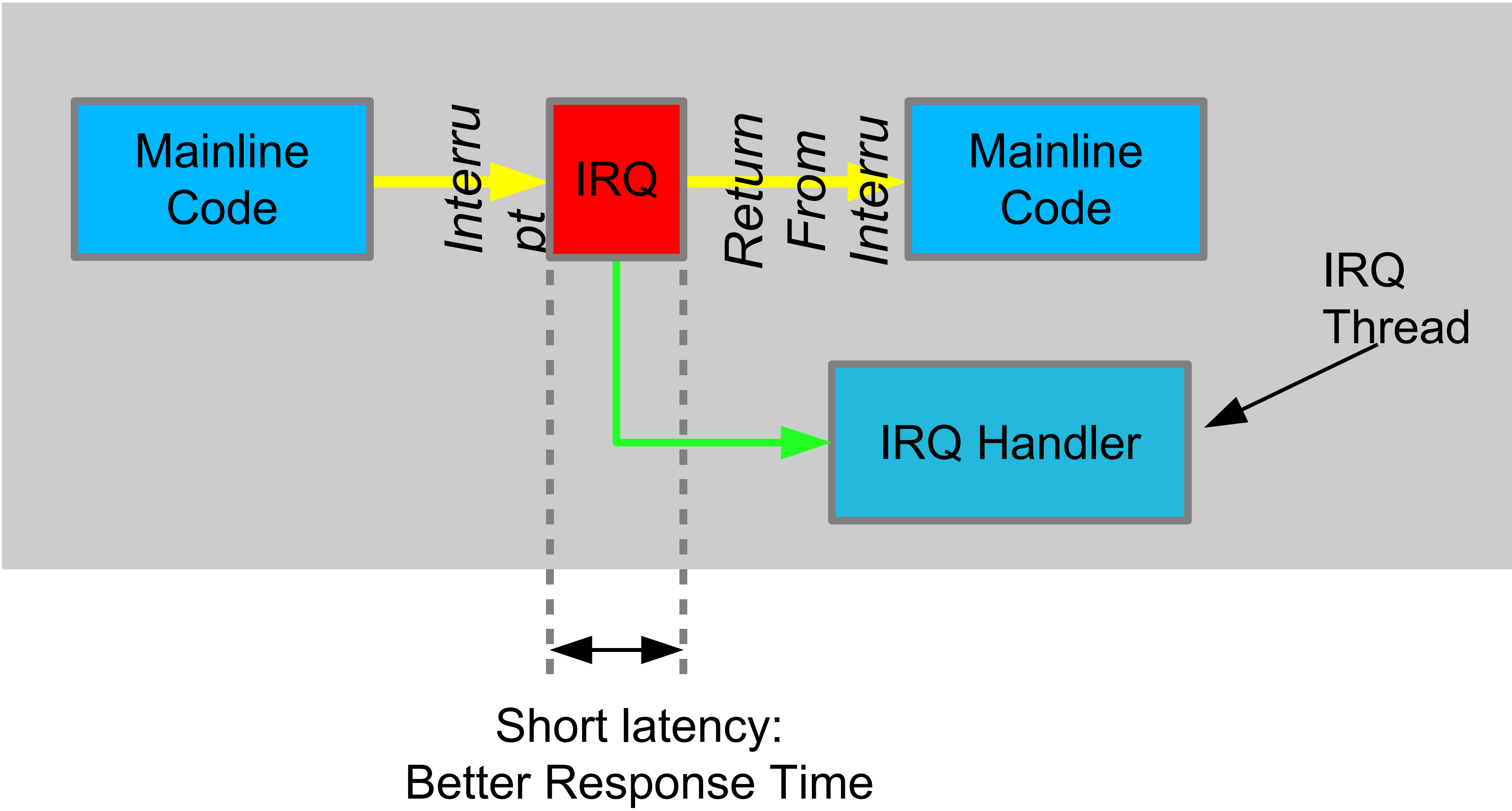
- http://lwn.net/Articles/152363/ (rationale for timer/hrtimer split)
- http://lwn.net/Articles/152436/ (timer implementation)
- http://lwn.net/Articles/167897/ (high-resolution timer API – dated)
- http://lwn.net/Articles/228143/ (deferrable timers)

Threaded Interrupt Handlers

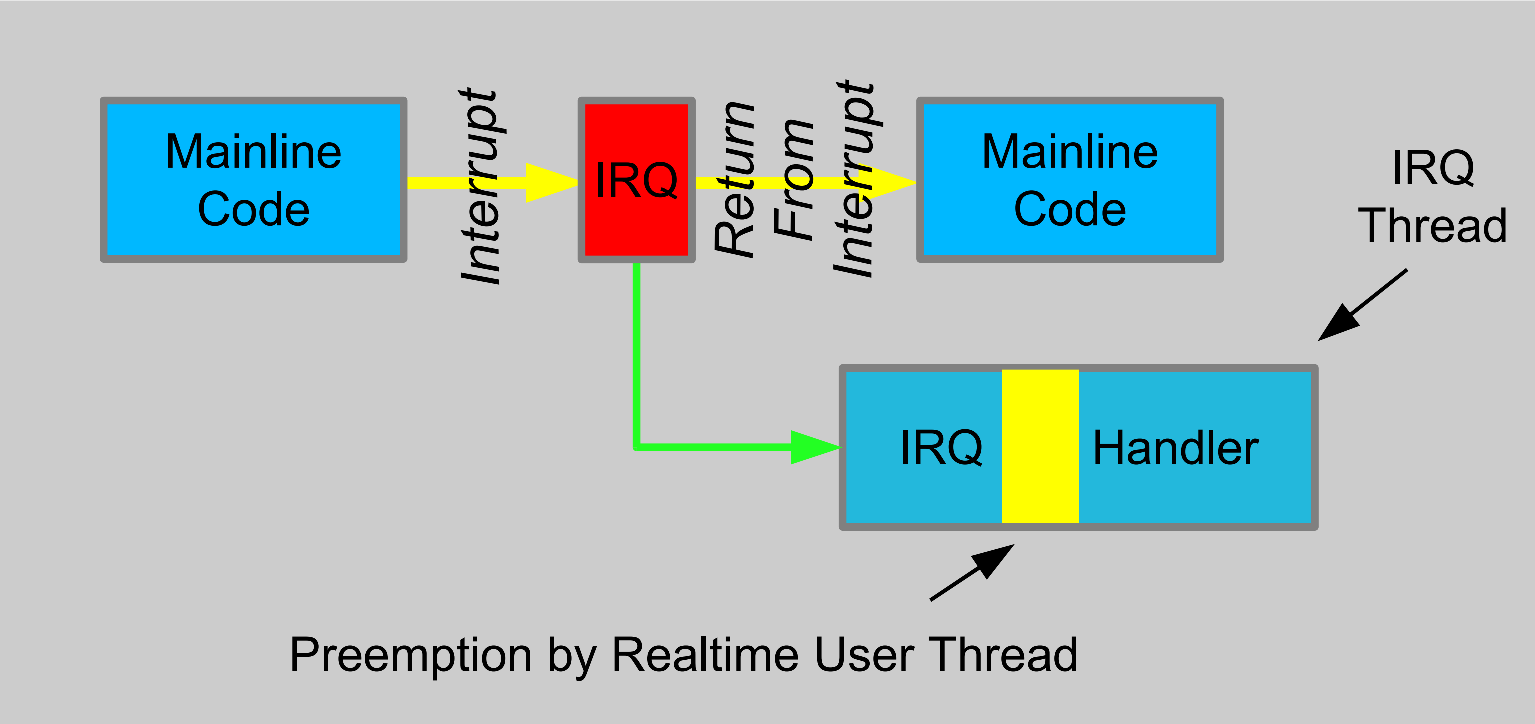
Linux's Non-Threaded Interrupt Handlers



-rt Patchset Threaded Interrupt Handlers



-rt Patchset Threaded Interrupt Handlers



Can get old hardirq behavior by specifying IRQ_NODELAY for given IRQ, but need very special handler: raw spinlocks, etc.

Writing Raw Interrupt Handlers

- When setting up irq:
  - Use IRQ_NODELAY in status field of irqdesc element
  - Use IRQF_NODELAY in action.flags field of irqdesc element
  - request_irq() propagates appropriately
- Must use raw_spinlock_t within handler
  - spinlock_t OK within non-IRQF_NODELAY handlers
- Example raw interrupt handlers:
  - Scheduling-clock interrupt, i8259 math_error_irq(), lpptest, xscale_pmu_interrupt(), and various irq-cascading handlers

Threaded Interrupts: To Probe Deeper

- http://lwn.net/Articles/106010/ (Approaches, October 2004)
- http://lwn.net/Articles/138174/ (Debate, June 2005)
- http://lwn.net/Articles/139062/ (softirq splitting, June 2005)

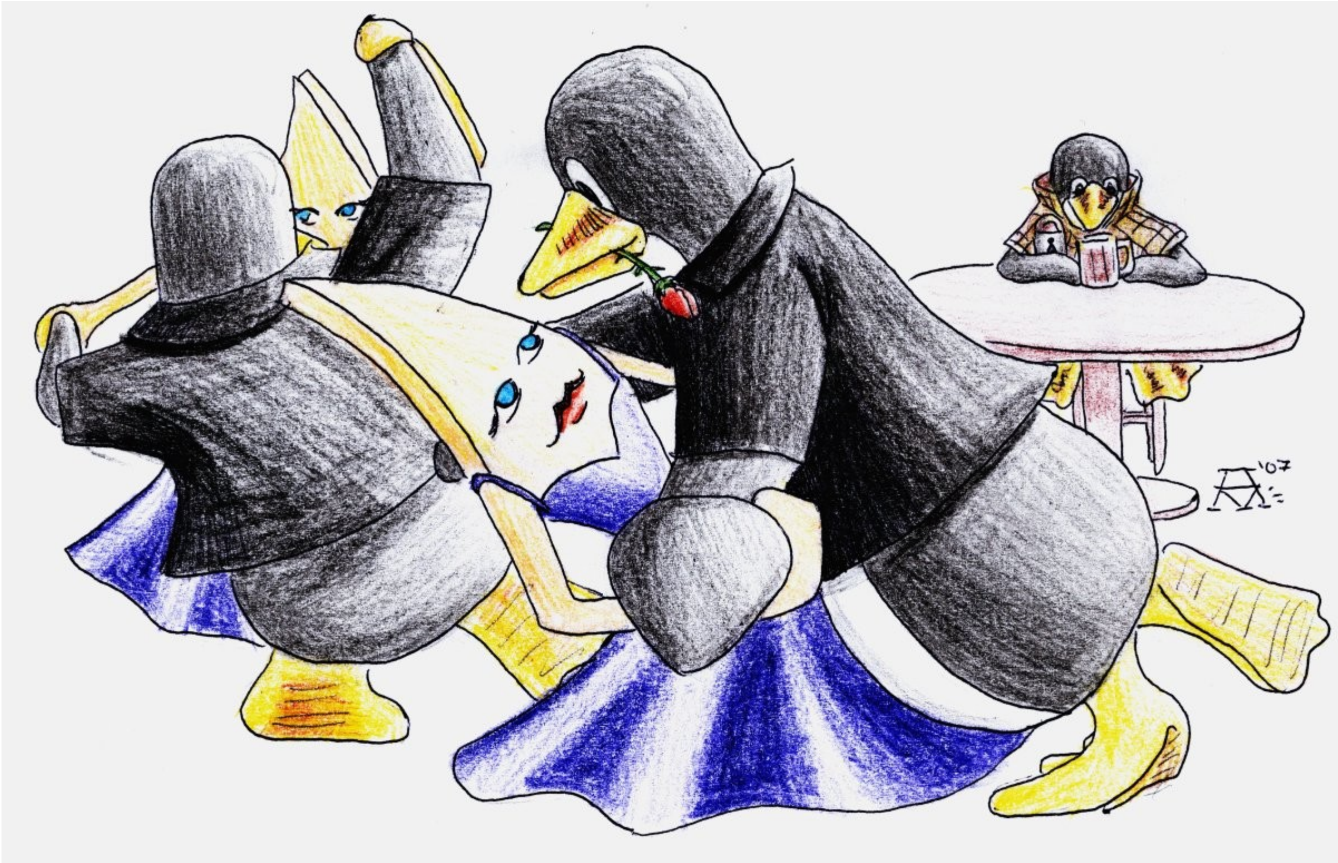
Priority Inversion and -rt Patchset

“Trapdoor” Metaphor for Priority Inheritance

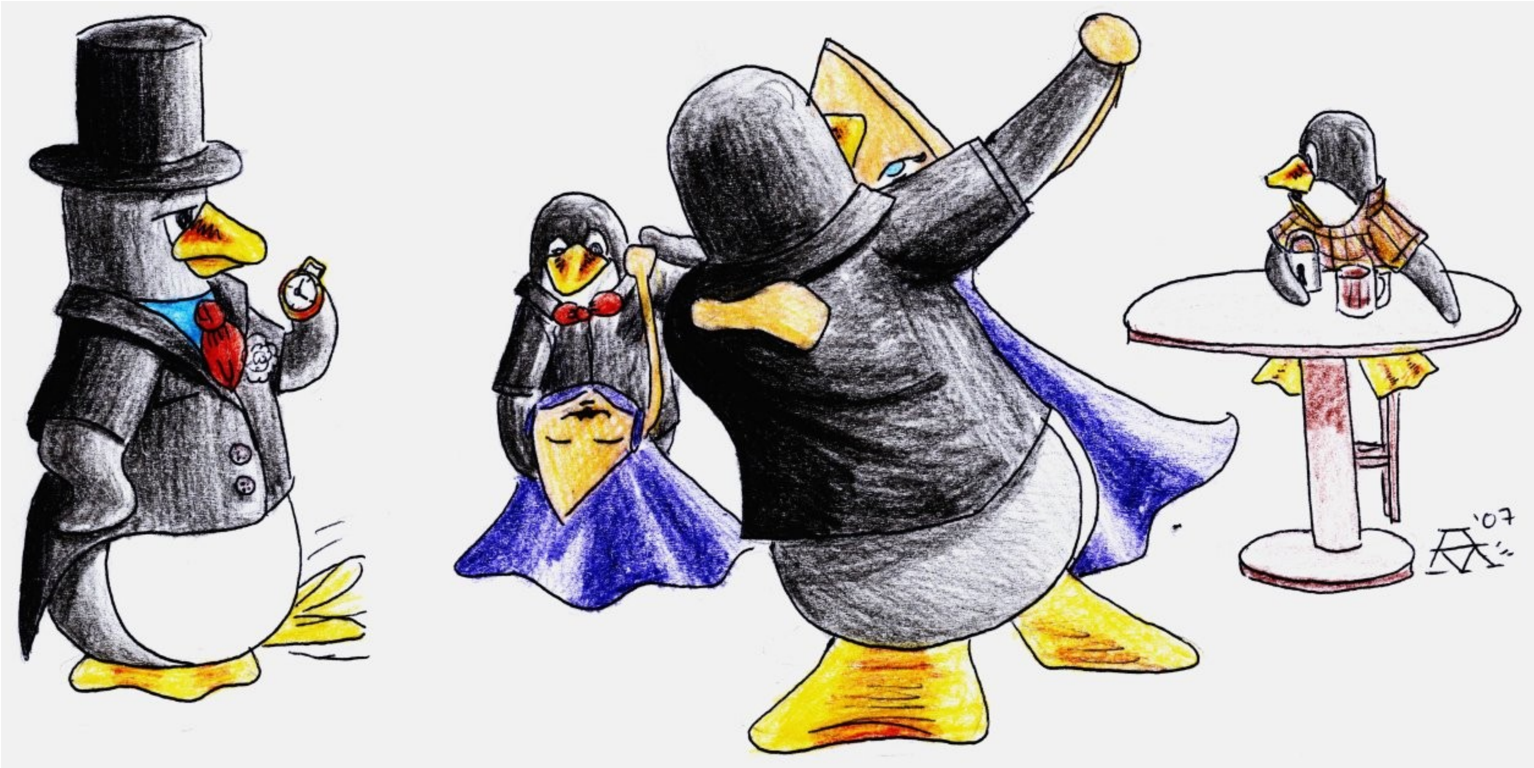
- A dance floor…
  - CPUs dance with highest priority tasks (Tuxes)
- Warning: any attempt to apply this metaphor in reverse will probably not end well…

Priority Inheritance

Priority Inheritance



Priority Inheritance

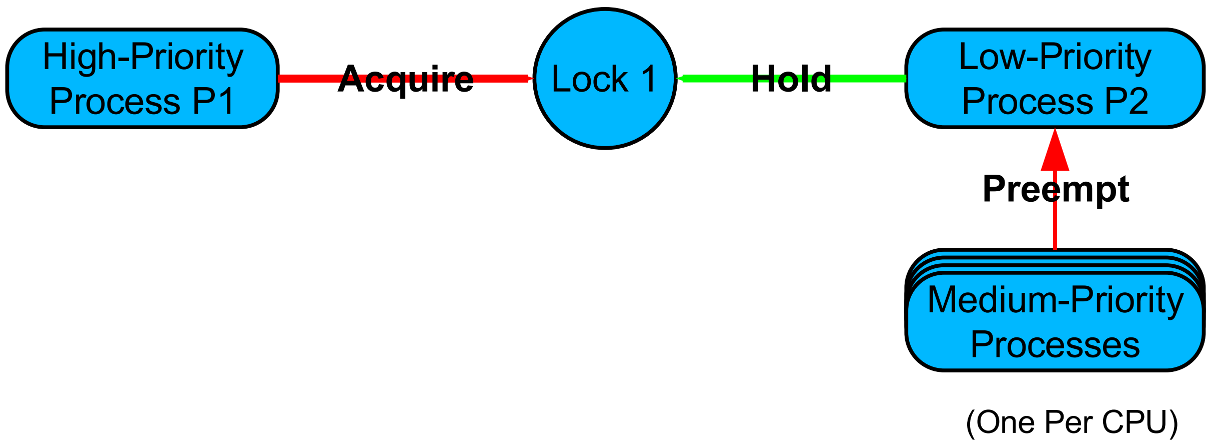


Priority Inheritance

Priority Inheritance

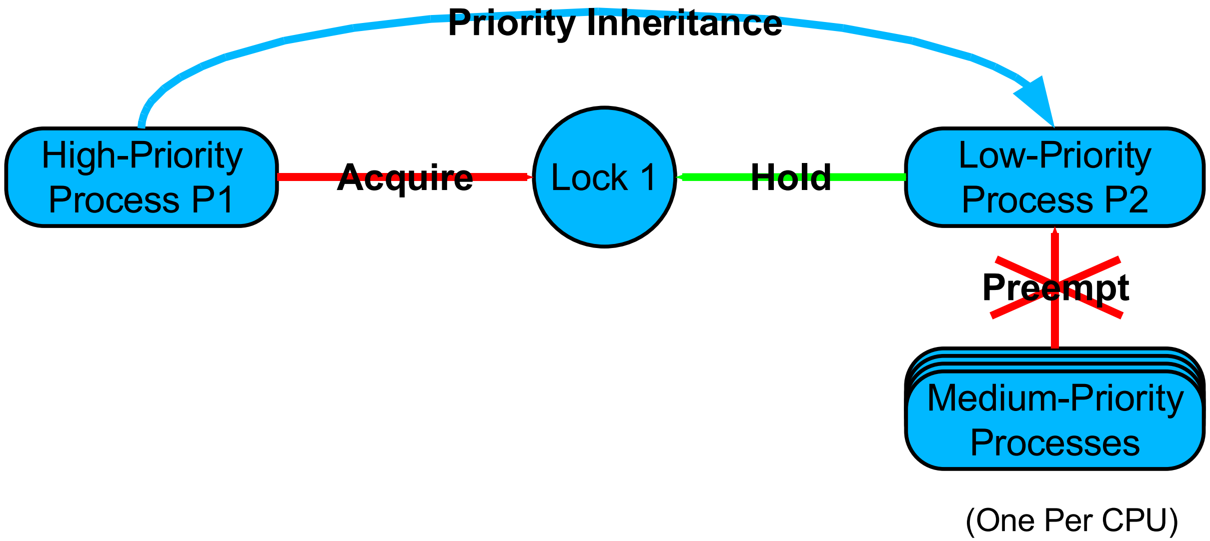
Priority Inversion Outside the Dance Hall

- Process P1 needs Lock L1, held by P2
- Process P2 has been preempted by medium-priority processes
  - Consuming all available CPUs
- Process P1 is blocked by lower-priority processes



Preventing Priority Inversion Outside the Dance Hall

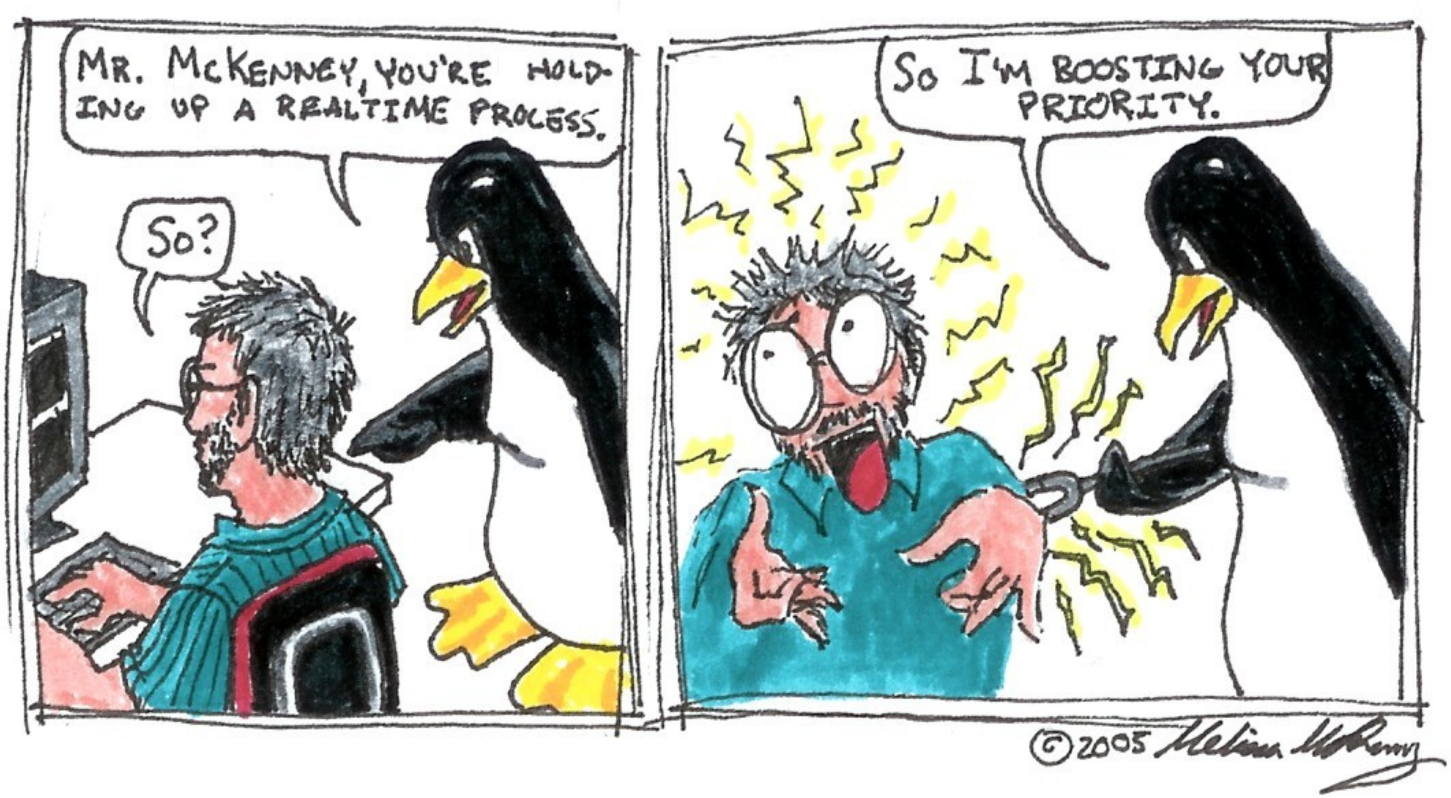
- Trivial solution: Prohibit preemption while holding locks
  - But degrades latency!!! Especially for sleeplocks!!!!
- Simple solution: “Priority Inheritance”: P2 “inherits” P1's priority
  - But only while holding a lock that P1 is attempting to acquire
  - Standard solution, very heavily used
- Either way, prevent the low-priority process from being preempted



Limits to Priority Boosting

- Inappropriate for ultimate in responsiveness
  - Then again, the same is true for digital hardware
- Does not work for events – who will raise the event?
- Does not work for memory exhaustion – who will free memory?
- Does not work for mass storage – make the disk spin faster???
- Does not work for network receives – boostee on other machine!
  - ***Could*** do cross-system boosting
  - But there are limits (see next slide)
- Does not work for reader-writer locking
  - At least not very well (see following slides)

In Some Cases, Priority Boosting is Undesirable...



...Or At Least Uncomfortable!!!

Priority-Inheritance API

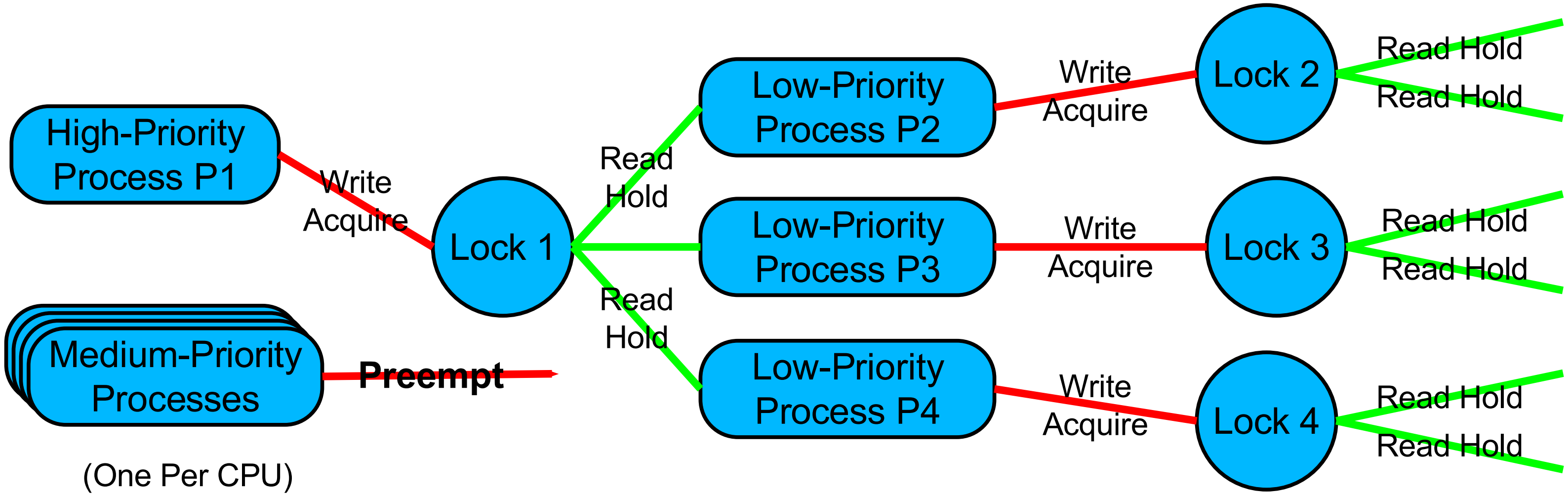
- All spinlock_t primitives do priority inheritance
- All struct semaphore primitives do priority inheritance
  - Use compat_semaphore to avoid priority inheritance (events)
- All struct mutex (and struct rt_mutex) primitives do priority inheritance
  - struct rt_mutex does priority inheritance in mainline as well
  - As of 2.6.22, used only by futexes

Priority Inheritance and Reader-Writer Locking

Priority Inheritance and Reader-Writer Locking

Priority Inversion and Reader-Writer Locking

- Process P1 needs Lock L1, held by P2, P3, and P4
  - Each of which is waiting on yet another lock
    - read-held by yet more low-priority processes
    - preempted by medium-priority processes
- Process P1 will have a long wait, despite its high priority
  - ***Even given priority inheritance: many processes to boost!***
- And a great many processes might need to be priority-boosted
  - Further degrading P1's realtime response latency

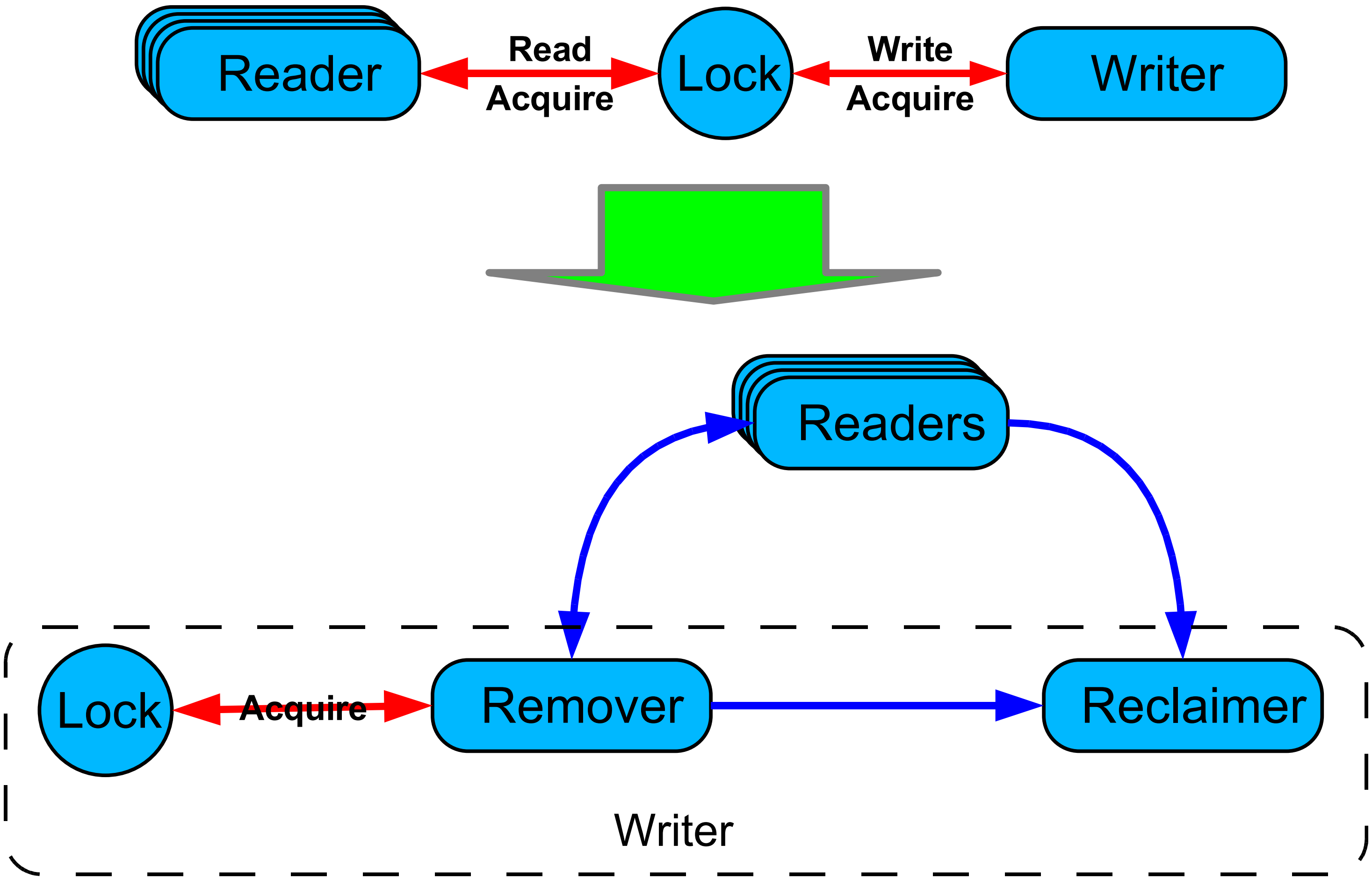


Priority Inheritance and Reader-Writer Locking

- Real-time operating systems have taken the following approaches to writer-to-reader priority boosting:
  - Boost only one reader at a time
    - Reasonable on a single-CPU machine, except in presence of readers that can block for other reasons.
    - Extremely ineffective on an SMP machine, as the writer must wait for readers to complete serially rather than in parallel
  - Boost a number of readers equal to the number of CPUs
    - Works well even on SMP, except in presence of readers that can block for other reasons (e.g., acquiring other locks)
  - Permit only one task at a time to read-hold a lock (PREEMPT_RT)
    - Very fast priority boosting, but severe read-side locking bottlenecks
- All of these approaches have heavy bookkeeping costs
  - Priority boost propagates transitively through multiple locks
  - Processes holding multiple locks may receive multiple priority boosts to different priority levels, actual boost must be to maximum level
  - Priority boost reduced (perhaps to intermediate level) when locks released
- So -rt patchset permits only one reading task at a time on a given lock
  - How to deal with this scalability limitation???

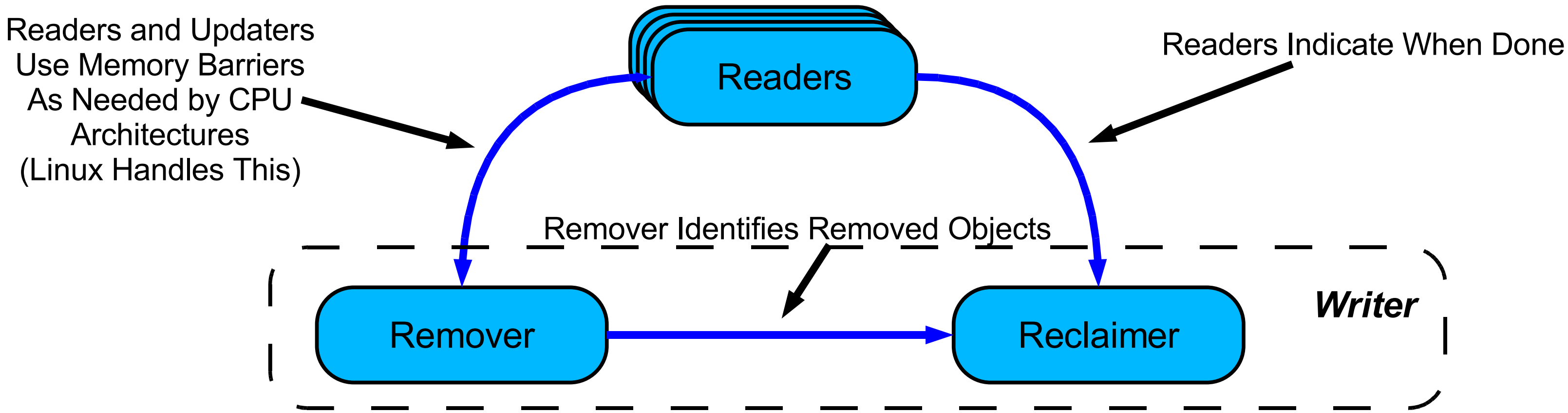
RCU

Reader-Writer Lock vs. RCU



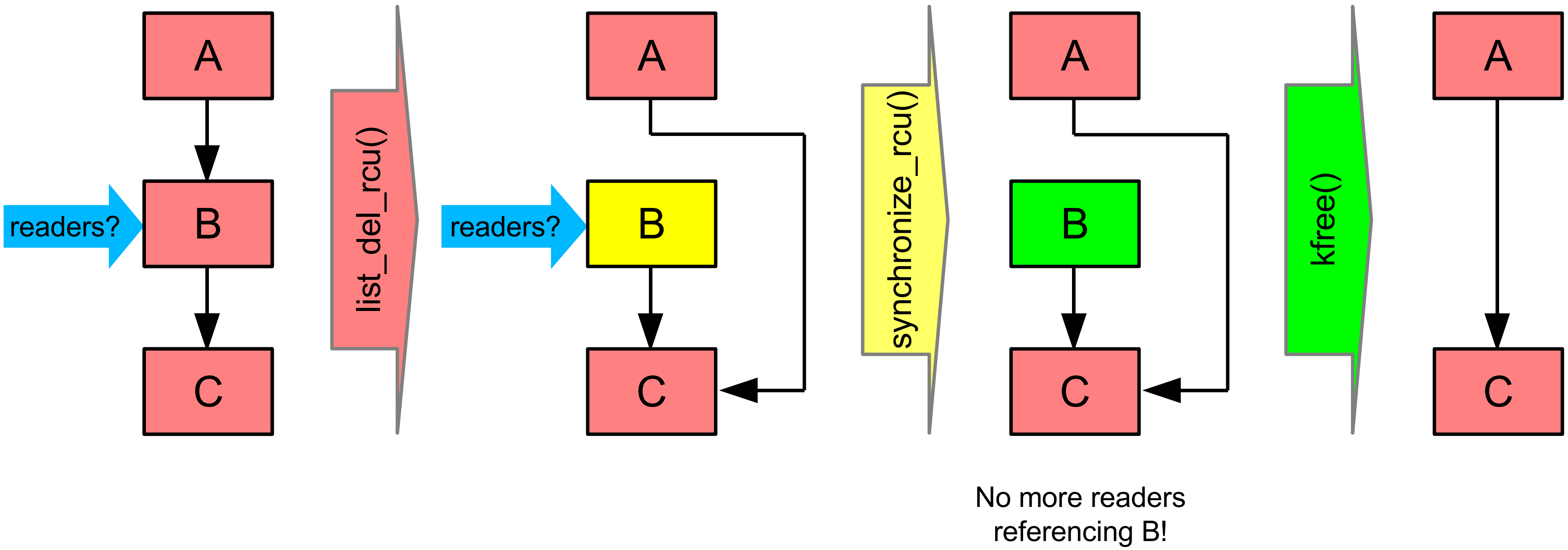
What is RCU?

- Analogous to reader-writer lock, but readers acquire no locks
  - Readers therefore cannot block writers
  - Readers cannot be involved in deadlock cycles
- Writers break updates into “removal” and “reclamation” phases
  - Removals do not interfere with readers
  - Reclamations deferred until all readers drop references
    - Readers cannot obtain references to removed items
- RCU used in production for over a decade by IBM (and Sequent)
  - RCU API best suited for read-intensive situations

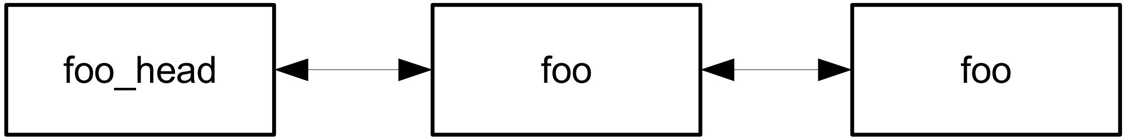


Example: RCU Removal From Linked List

- 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())



Code For RCU Removal From Linked List

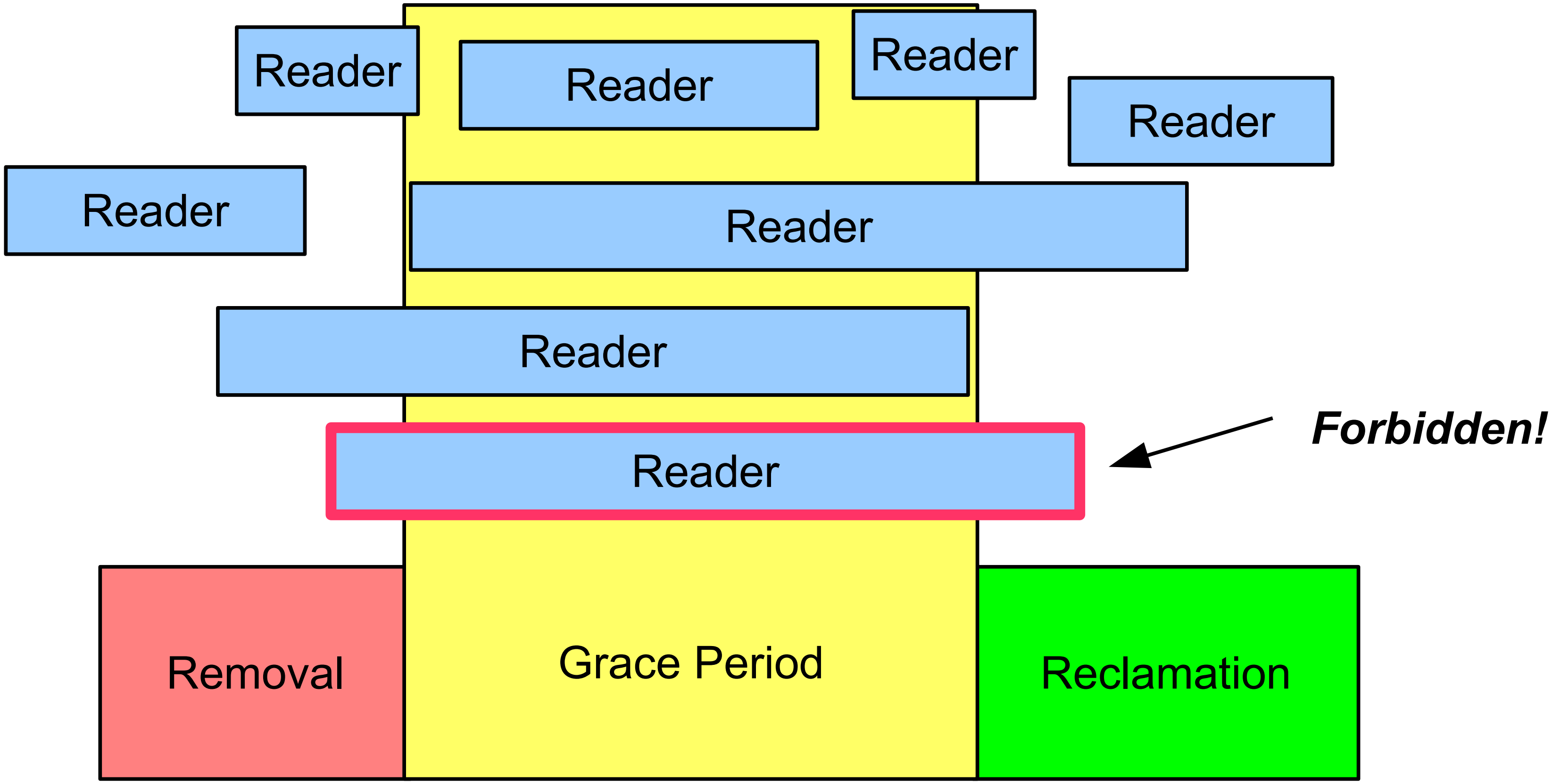



```
struct foo_head {
   struct list_head list;
   spinlock_t mutex;
};

struct foo {
   struct list_head list;
   int key;
};
```

```
int search(struct foo_head *fhp, int k)
{
   struct foo *p;
   struct list_head *head = &fhp->list;

   rcu_read_lock();
   list_for_each_entry_rcu(p, head, list) {
      if (p->key == k) {
         rcu_read_unlock();
         return 1;
      }
   }
   rcu_read_unlock();
   return 0;
}
```

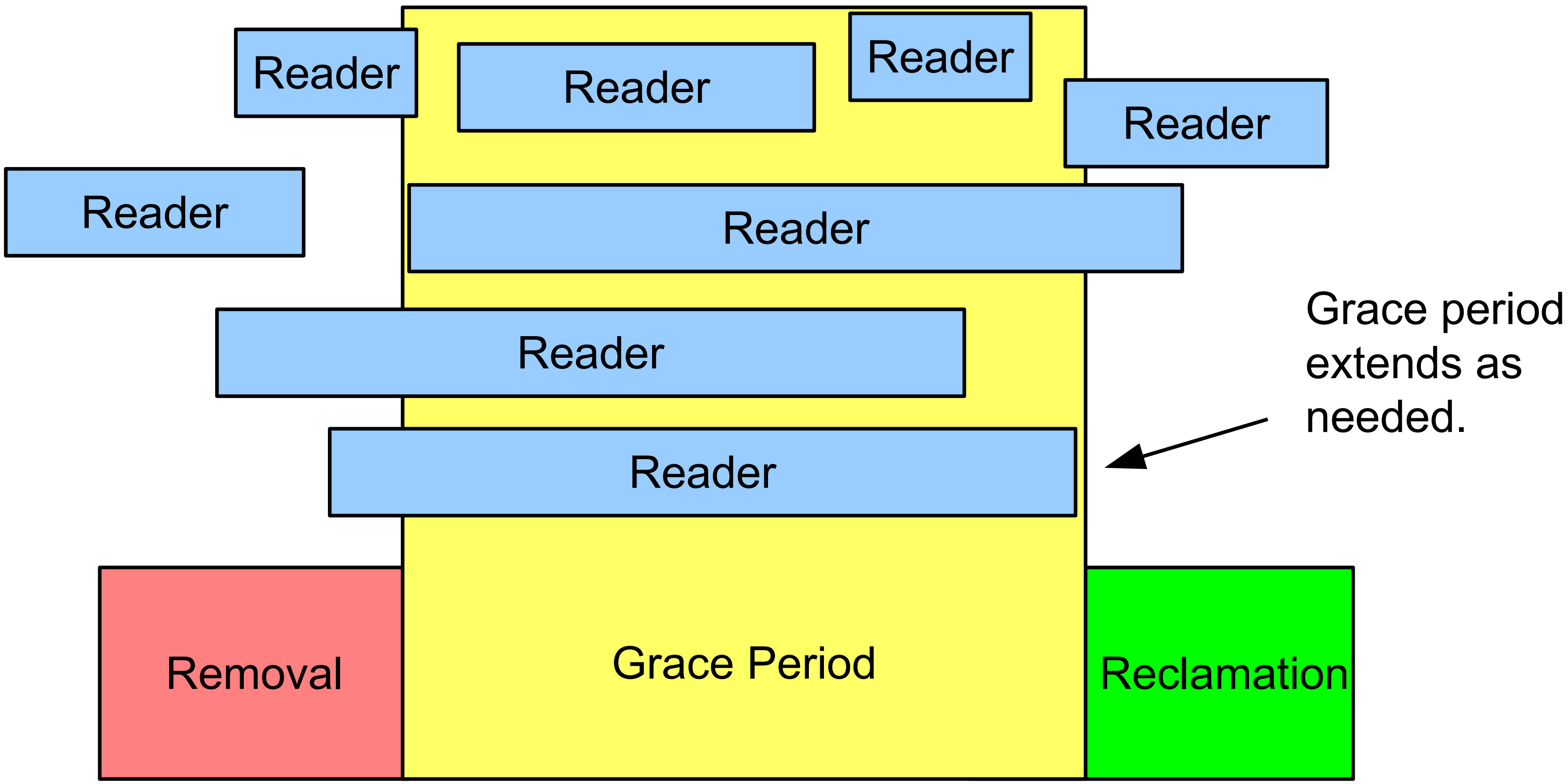
```
int delete(struct foo_head *fhp, int k)
{
   struct foo *p;
   struct list_head *head = &fhp->list;

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

Relation of Grace Period to Readers



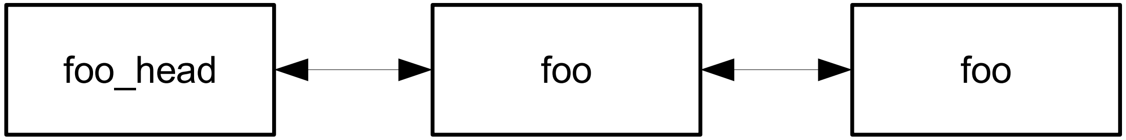
So what happens if you try to extend an RCU read-side critical section across a grace period?

Relation of Grace Period to Readers



So what happens if you try to extend an RCU read-side critical section across a grace period?

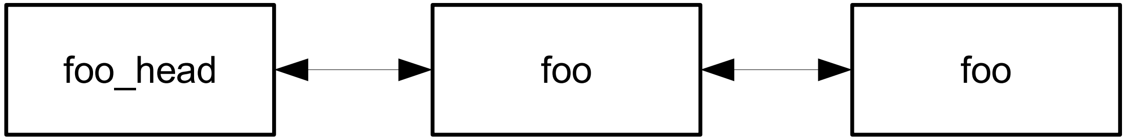
Code For RCU Removal From Linked List




```
struct foo_head {
   struct list_head list;
   spinlock_t mutex;
};

struct foo {
   struct list_head list;
   int key;
};


int search(struct foo_head *fhp, int k)
{
   struct foo *p;
   struct list_head *head = &fhp->list;

   rcu_read_lock();
   list_for_each_entry_rcu(p, head, list) {
      if (p->key == k) {
         rcu_read_unlock();
         return 1;
      }
   }
   rcu_read_unlock();
   return 0;
}
```

```
int delete(struct foo_head *fhp, int k)
{
   struct foo *p;
   struct list_head *head = &fhp->list;

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

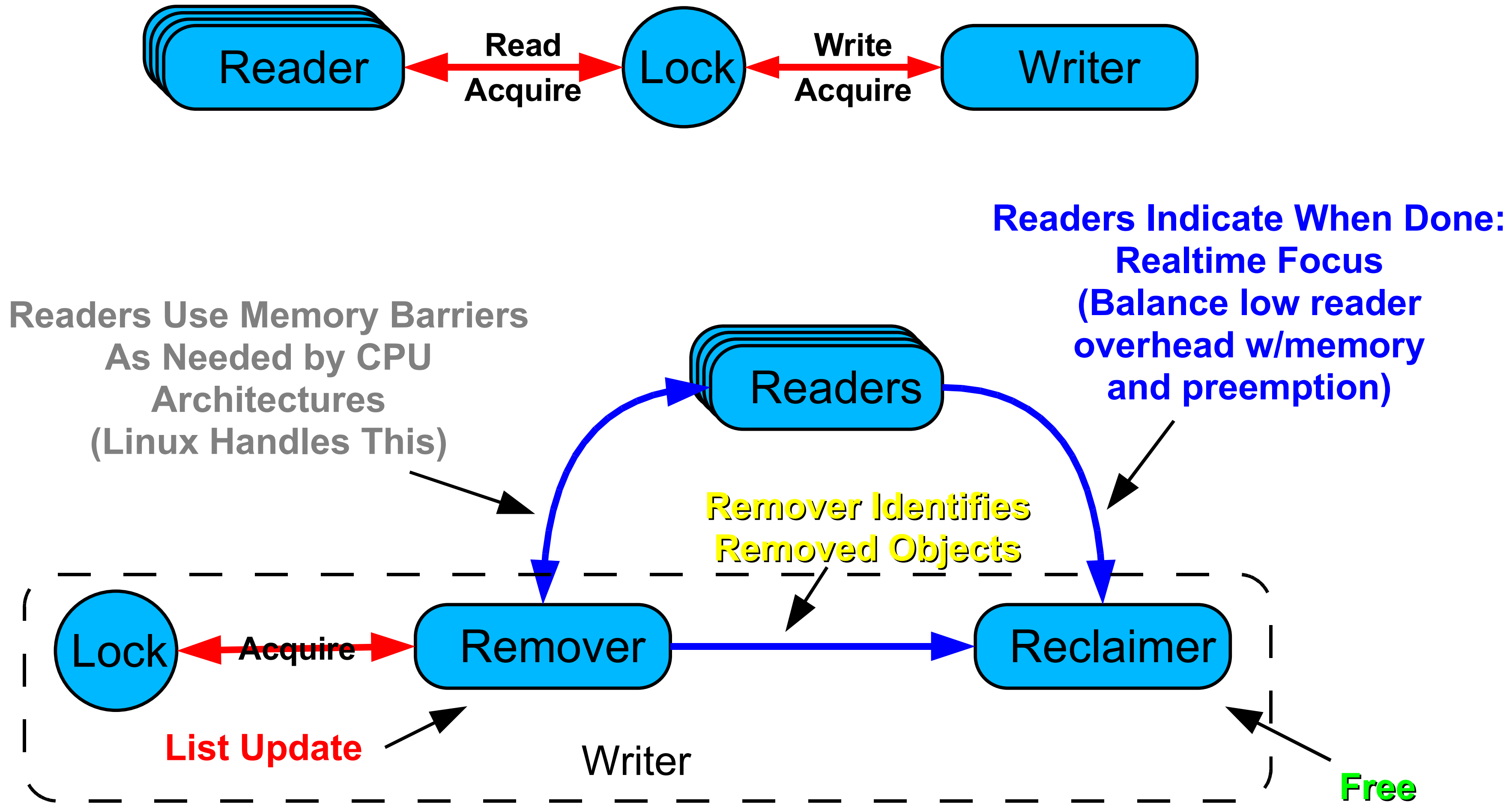
Code For Reader-Writer Removal From Linked List




```
struct foo_head {
   struct list_head list;
   rwlock_t mutex;
};

struct foo {
   struct list_head list;
   int key;
};


int search(struct foo_head *fhp, int k)
{
   struct foo *p;
   struct list_head *head = &fhp->list;

   read_lock(&fhp->mutex);
   list_for_each_entry(p, head, list) {
      if (p->key == k) {
         read_unlock(&fhp->mutex);
         return 1;
      }
   }
   read_unlock(&fhp->mutex);
   return 0;
}
```

```
int delete(struct foo_head *fhp, int k)
{
   struct foo *p;
   struct list_head *head = &fhp->list;

   write_lock(&fhp->mutex);
   list_for_each_entry(p, head, list) {
      if (p->key == k) {
         list_del(p);
         write_unlock(&fhp->mutex);
         /* */
         kfree(p);
         return 1;
      }
   }
   write_unlock(&fhp->mutex);
   return 0;
}
```

Reader-Writer Lock vs. RCU



Guide to RCU API

`rcu_read_lock()`
`rcu_read_unlock()`
	`synchronize_rcu()`
	`synchronize_net()`
	`call_rcu()`
		General purpose

`rcu_read_lock_bh()` `rcu_read_unlock_bh()`	`call_rcu_bh()`	Bottom-half context (networking)
`preempt_disable()` `preempt_enable()`	`synchronize_sched()`	Hard IRQs, NMIs, ...
`srcu_read_lock()` `srcu_read_unlock()`	`synchronize_srcu()`	When readers must sleep

```
rcu_dereference()
list_for_each_entry_rcu()
hlist_for_each_entry_rcu()
[list_for_each_rcu()]
[list_for_each_continue_rcu()]
```

Pointer dereferencing, including list traversal

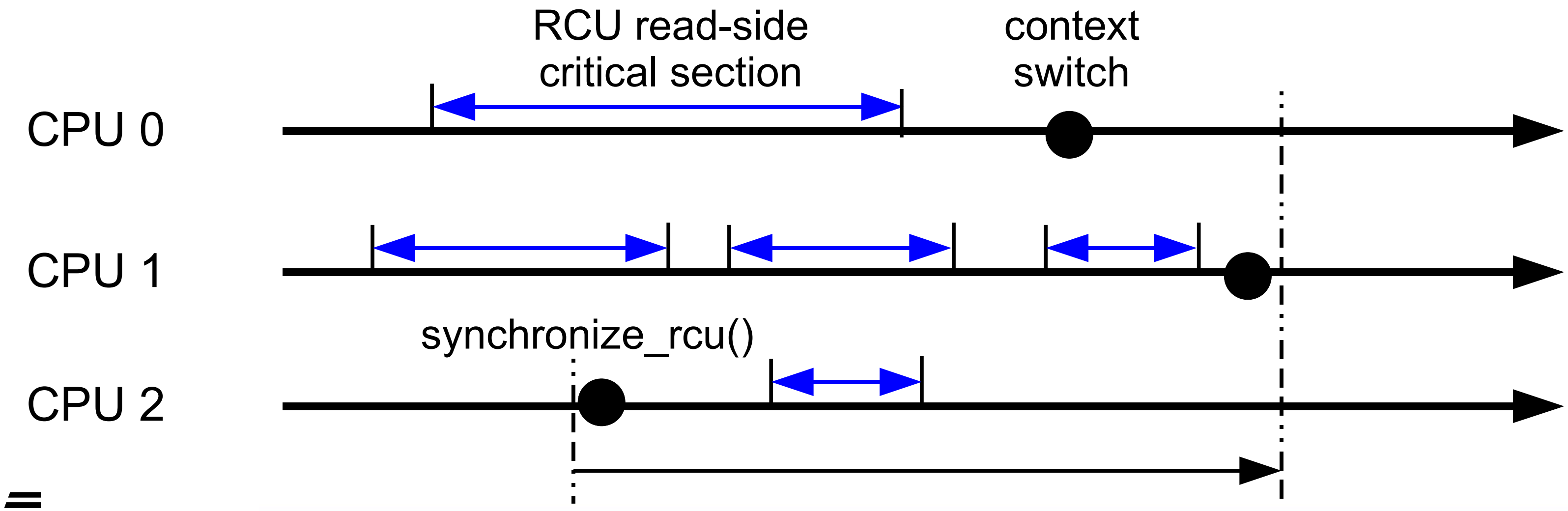
List update

```
rcu_assign_pointer()
list_add_rcu()
list_add_tail_rcu()
list_del_rcu()
list_replace_rcu()
hlist_del_rcu()
hlist_add_head_rcu()
```

```
kfree()
kmem_cache_free()
```

Example RCU Infrastructure Implementation

- 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 uses counter-based algorithm
    - Permits preemption of RCU read-side critical sections



RCU Read-Side Primitives: How Lightweight?

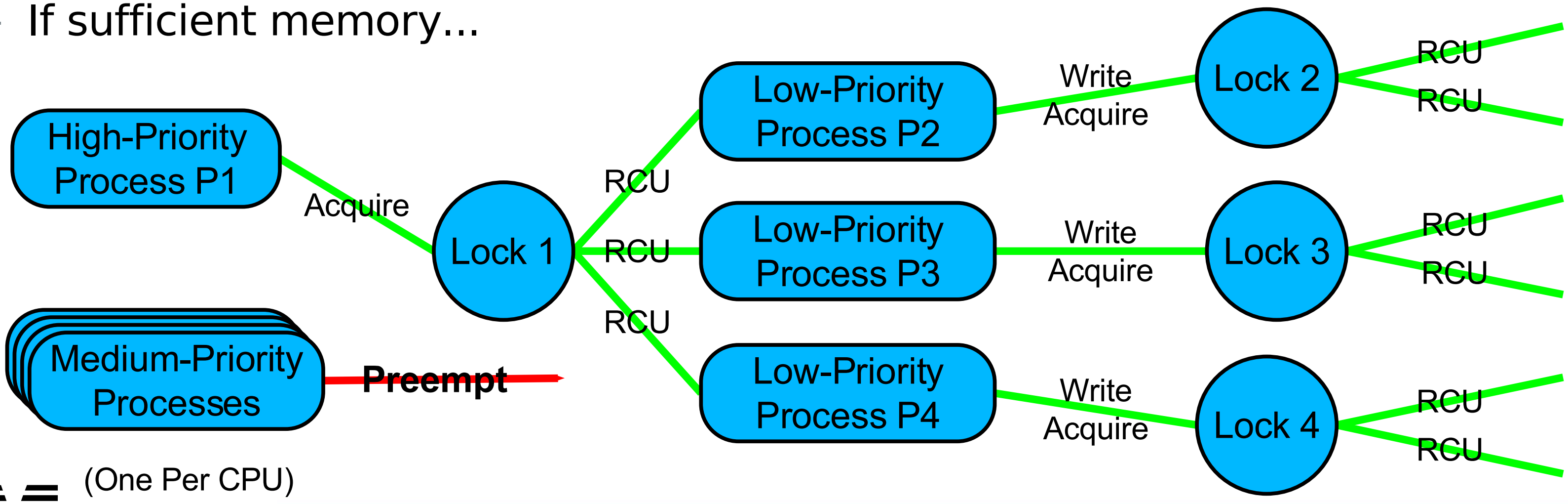
- RCU
  - “Classic RCU” (non-CONFIG_PREEMPT)
    - `#define rcu_read_lock()`
    - `#define rcu_read_unlock()`
  - CONFIG_PREEMPT RCU
    - `#define rcu_read_lock() preempt_disable()`
    - `#define rcu_read_unlock() preempt_enable()`
  - CONFIG_PREEMPT_RT RCU on following slide
- RCU BH
  - `#define rcu_read_lock_bh() {rcu_read_lock(); local_bh_disable(); }`
  - `#define rcu_read_unlock_bh() { local_bh_enable(); rcu_read_unlock(); }`
- synchronize_sched() RCU
  - `#define preempt_disable() {inc_preempt_count(); barrier(); }`
  - `#define preempt_enable() { barrier(); dec_preempt_count(); }`

But What About The Update Side?

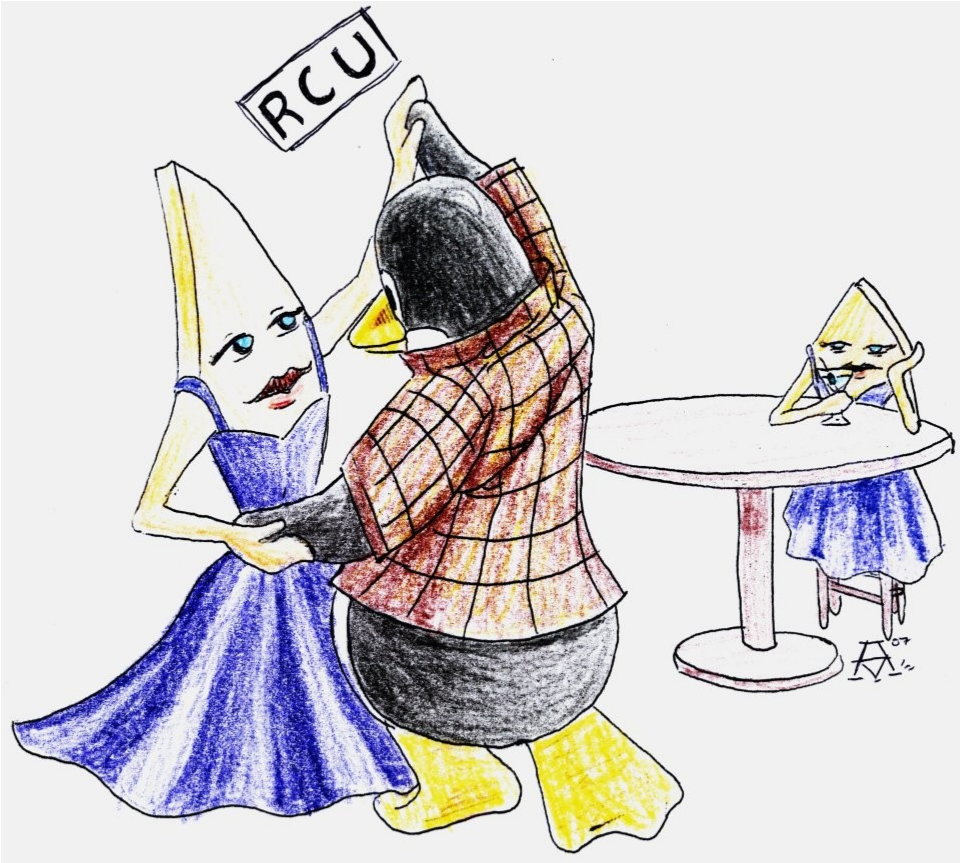
- Updates can be ***quite*** expensive, despite numerous optimizations
- Which is why RCU should be used for read-mostly situations
  - Use the right tool for the job!!!
- The important thing is ***overall*** performance:
  - System V IPC: 12x at system call level, >5% DB benchmark
  - dcache: 10-30% improvement SDET, SPECweb99, kernbench
  - FD array: Up to 30% improvement in chat
  - SELinux avc: More than 2 orders of magnitude on 32 CPUs
  - IP route cache: 2x reduction in lookup overhead

Priority Inversion and RCU

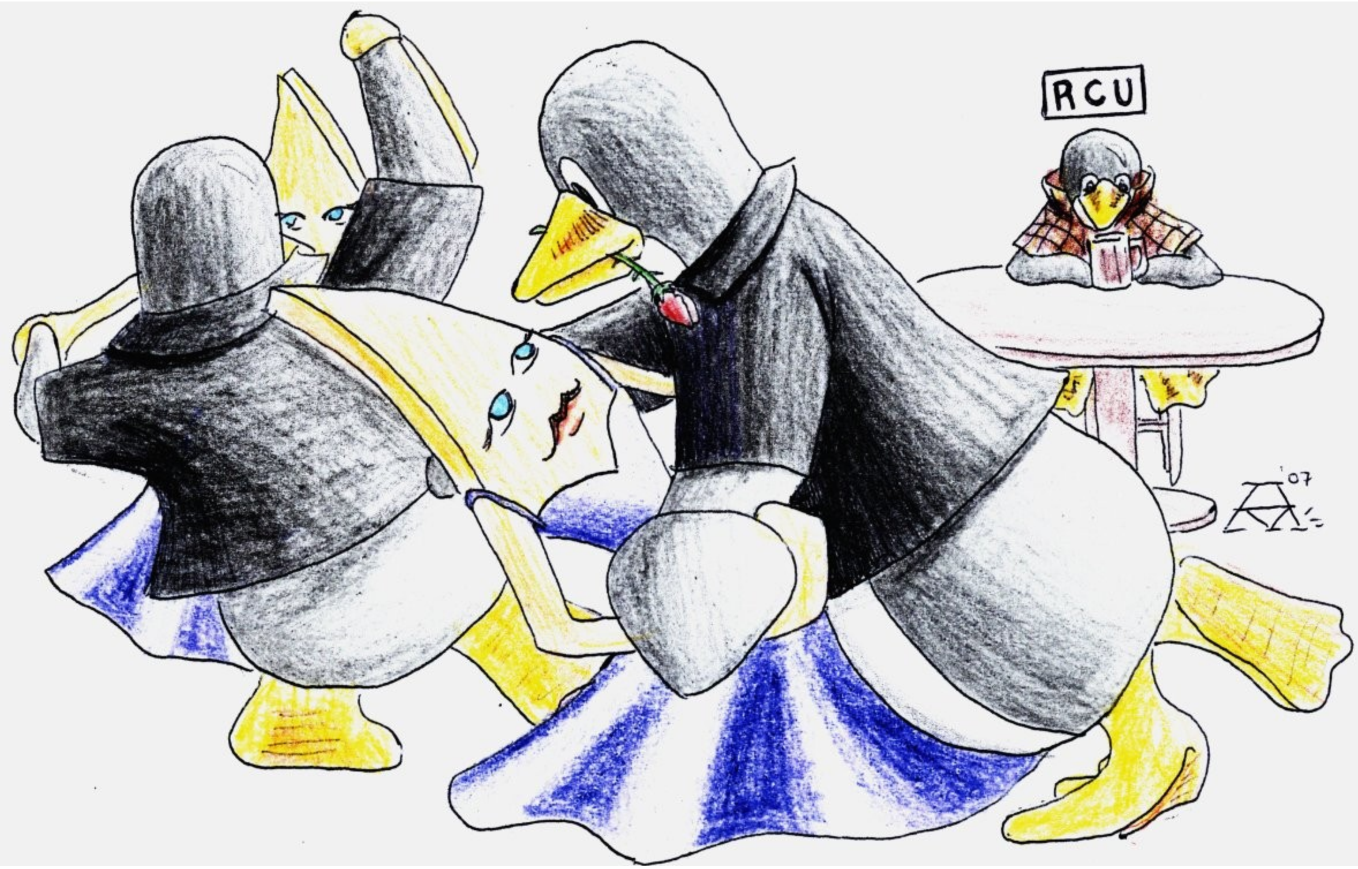
- Process P1 needs Lock L1, but P2, P3, and P4 now use RCU
  - P2, P3, and P4 therefore need not hold L1
  - Process P1 thus immediately acquires this lock
  - Even though P2, P3, and P4 are preempted by the per-CPU medium-priority processes
- No priority inheritance required
  - Except if low on memory: permit reclaimer to free up memory
- Excellent realtime latencies: medium-priority processes can run
  - High-priority process proceeds despite low-priority process preemption
  - If sufficient memory...



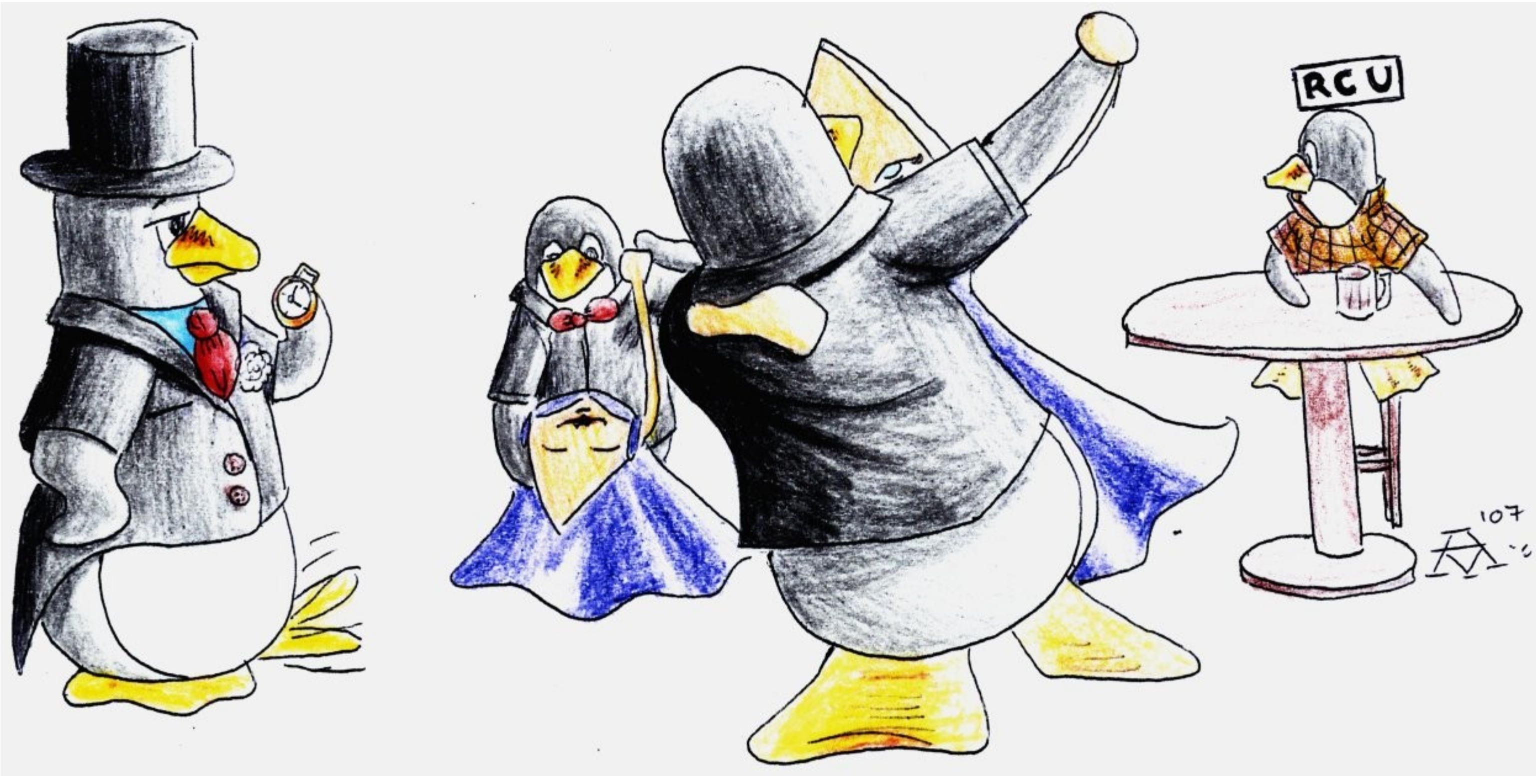
Priority Inversion and RCU



Priority Inversion and RCU



Priority Inversion and RCU



Priority Inversion and RCU

Realtime and RCU

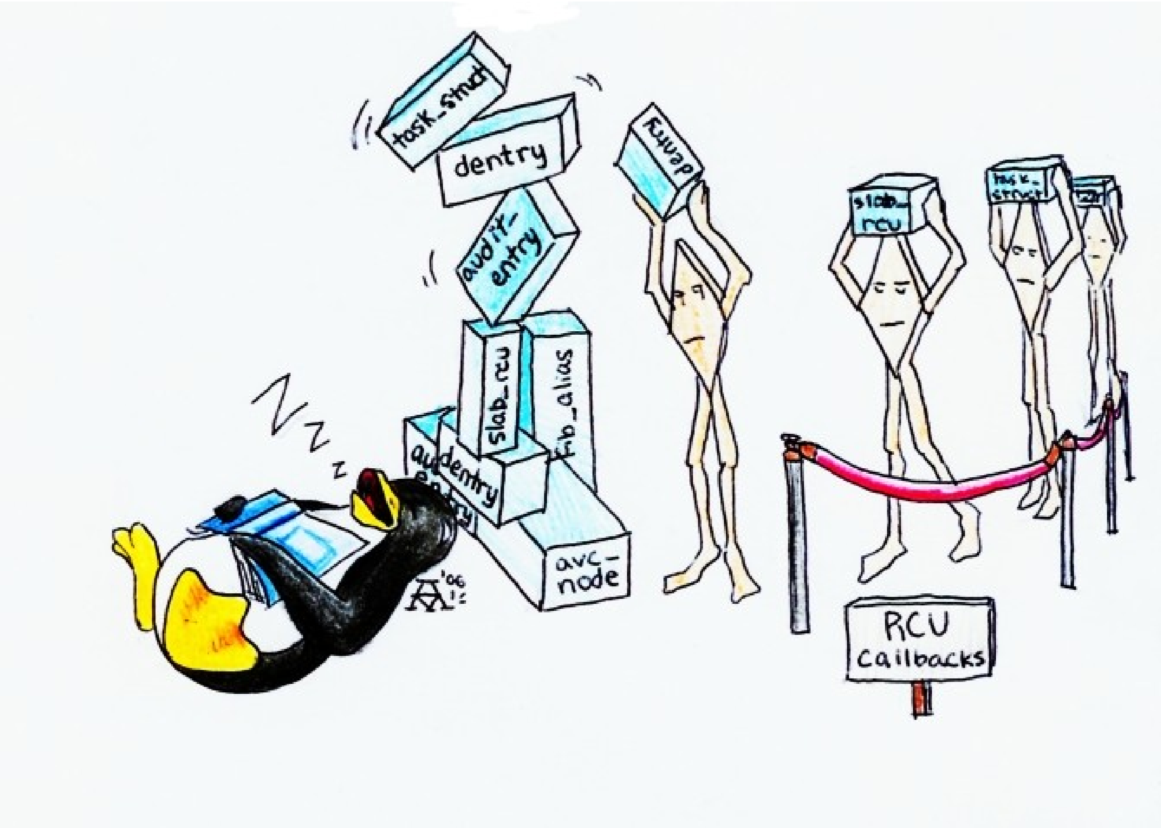
- RCU exploited in PREEMPT_RT patchset to reduce latencies
  - "kill()" system-call RCU provided large reduction in latency
  - Expect similar benefits for pthread_cond_broadcast() and pthread_cond_signal()
- Current PREEMPT_RT realtime Linux provides relatively few realtime services
  - Process scheduling, interrupts, some signals
- Increasing the number of realtime services will likely require additional exploitation of RCU
  - And will likely require that RCU readers be priority-boosted when low on memory
- But "Classic RCU" has realtime-latency problems of its own!!!
  - Classic RCU disables preemption across read-side critical sections...

What is Needed From Realtime RCU

- Reliable
- Callable from IRQ
- ***Preemptible read-side critical sections***
- ***Small memory footprint***
- Synchronization-free read side
- Independent of memory-allocator data structures
- Freely nestable read side
- Unconditional read-to-write upgrade
- API compatible with “Classic RCU”

Why small memory footprint???

But Can't *Just* Make RCU Preemptible...



Small memory footprint means timely grace-period processing...

Overhead of RT RCU Read-Side....

- Heavier weight than the classic RCU implementations
- But still:
  - No locks
  - No loops
  - In out-of-tree patch:
    - No atomic instructions
    - No memory barriers
  - So still lightweight with *O(1)* worst-case execution time
    - And many implementations have *fixed* execution time

Real-Time rcu_read_lock()

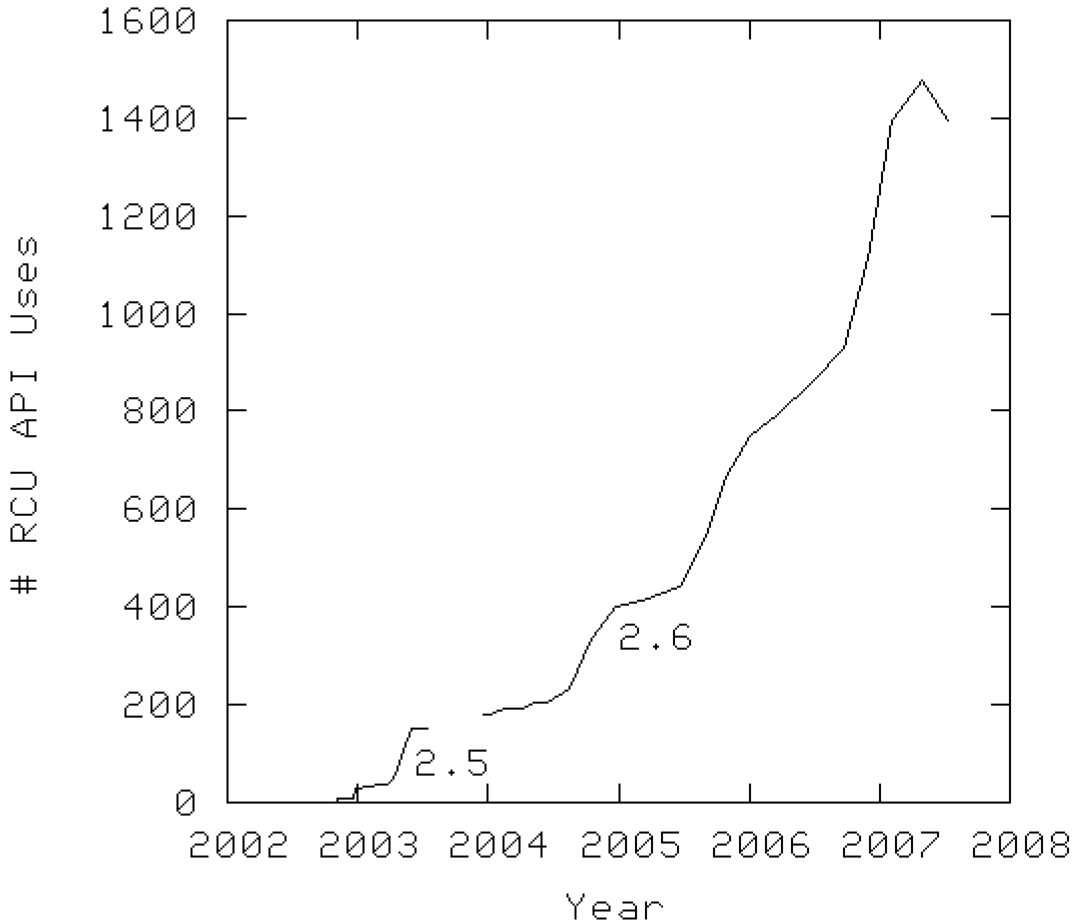
```
void rcu_read_lock(void)
{
        int idx;
        int nesting;

        nesting = current->rcu_read_lock_nesting;
        if (nesting != 0) {
                current->rcu_read_lock_nesting = nesting + 1;
        } else {
                unsigned long oldirq;

                local_irq_save(oldirq);
                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);
        }
}
```

Real-Time rcu_read_unlock()

```
void __rcu_read_unlock(void)
{
        int idx;
        int nesting;

        nesting = current->rcu_read_lock_nesting;
        if (nesting > 1) {
                current->rcu_read_lock_nesting = nesting - 1;
        } else {
                unsigned long oldirq;

                local_irq_save(oldirq);
                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);
        }
}
```

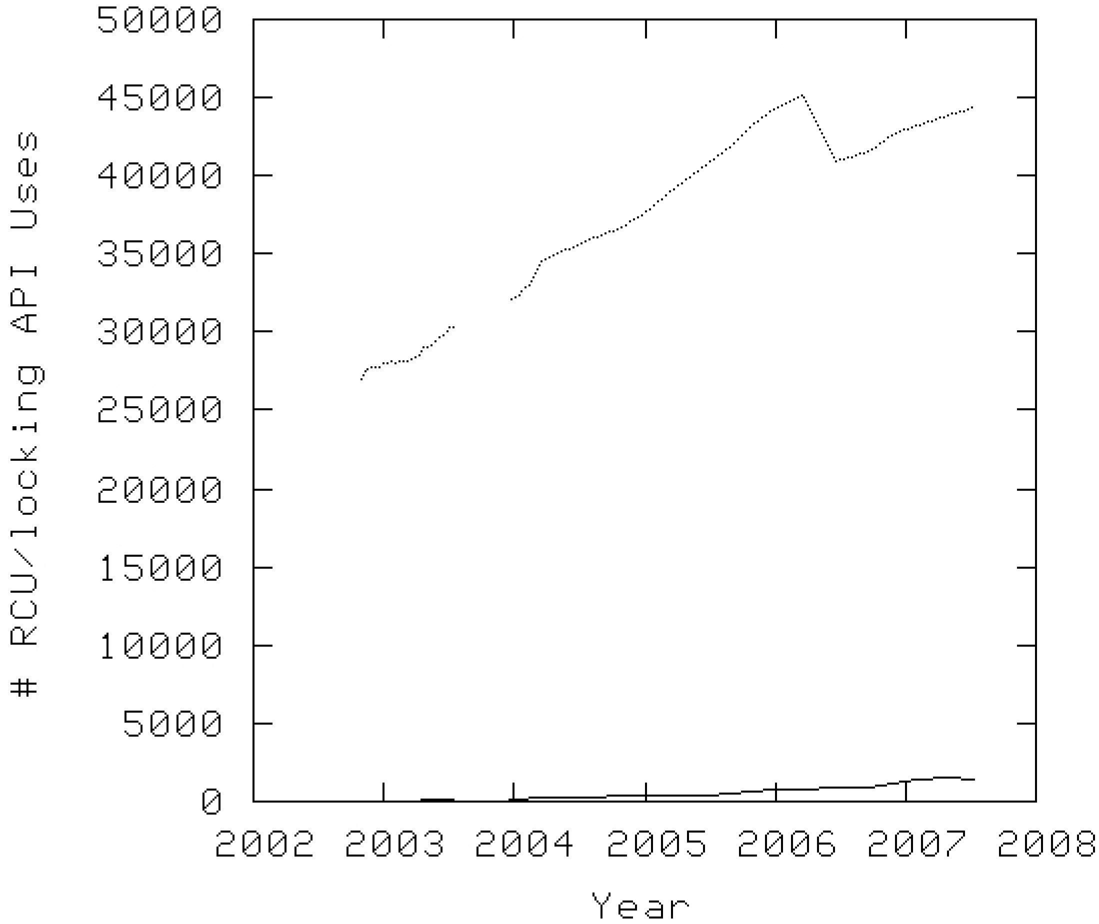
Can the Linux Community Handle RCU?

Linux Usage of RCU APIs

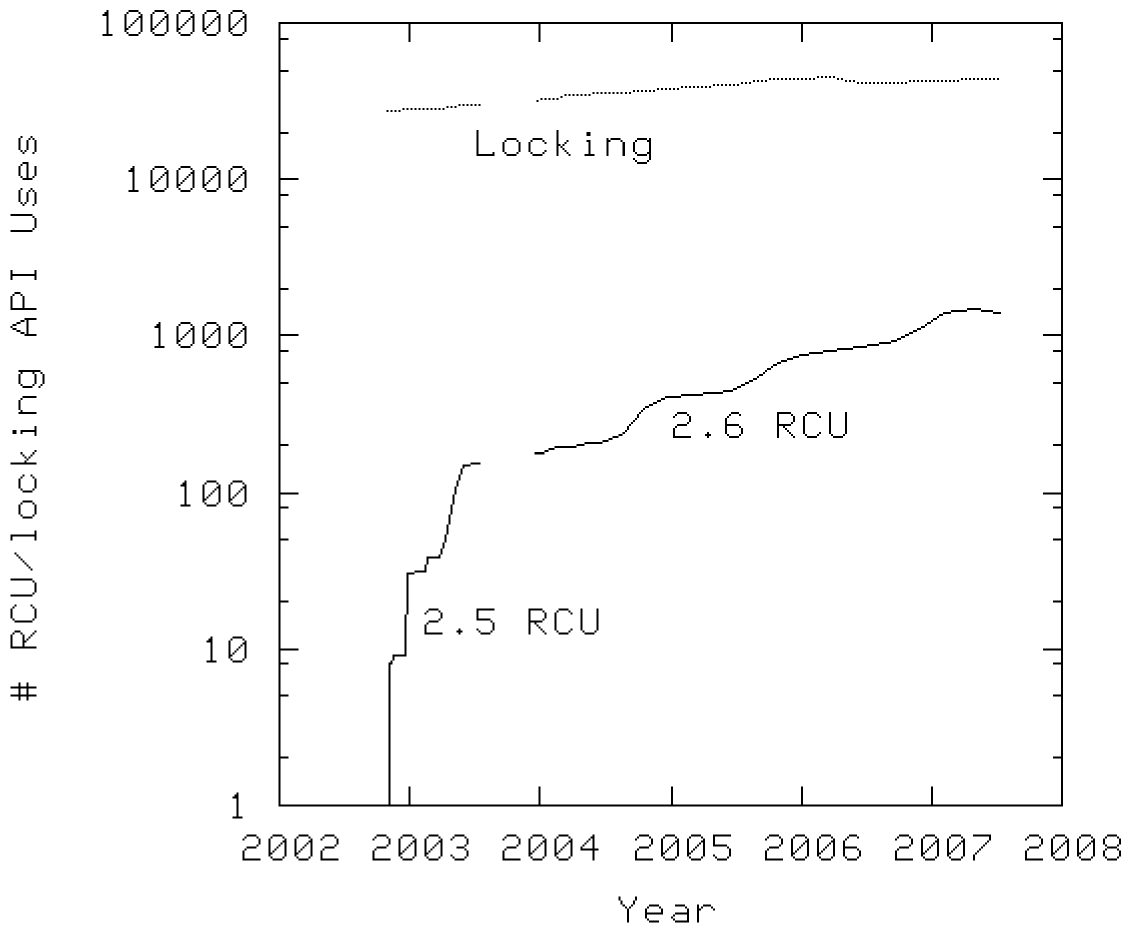


http://www.rdrop.com/users/paulmck/RCU/linuxusage.html

Linux Usage of RCU APIs – In Perspective



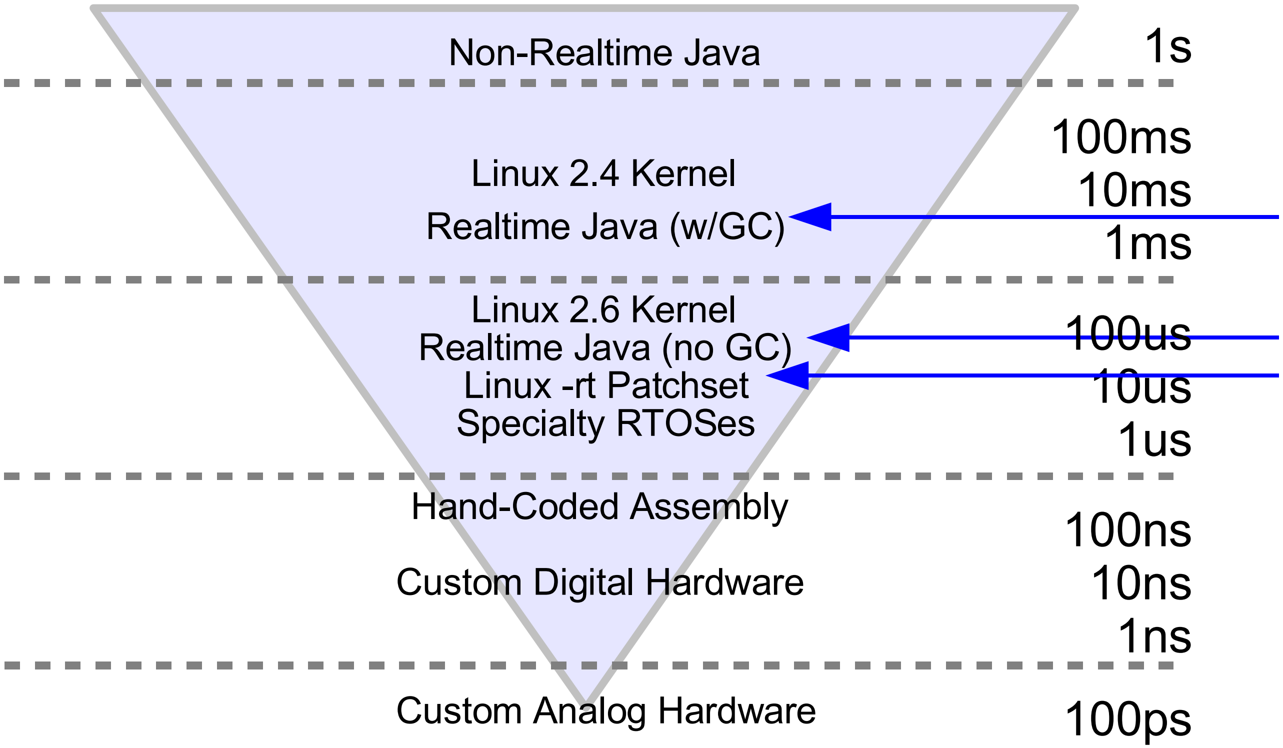
Linux Usage of RCU APIs – Perspective II



To Probe Deeper

- http://en.wikipedia.org/wiki/RCU
- http://lwn.net/Articles/128228/ (early realtime-RCU attempt)
- http://www.rdrop.com/users/paulmck/RCU/OLSrtRCU.2006.08.11a.pdf (realtime-RCU OLS paper)
- http://www.rdrop.com/users/paulmck/RCU/ (More RCU papers)
- http://www.rdrop.com/users/paulmck/RCU/linuxusage.html (Graphs)
- http://lwn.net/Articles/201195/ (Jon Corbet realtime-RCU writeup)
- http://lwn.net/Articles/220677/ (RCU priority boosting)
- http://lwn.net/Articles/220677/ (patch for higher-performance RCU)

Summary: Realtime Regimes Redux



Summary

Use
the right tool
for the job!!!

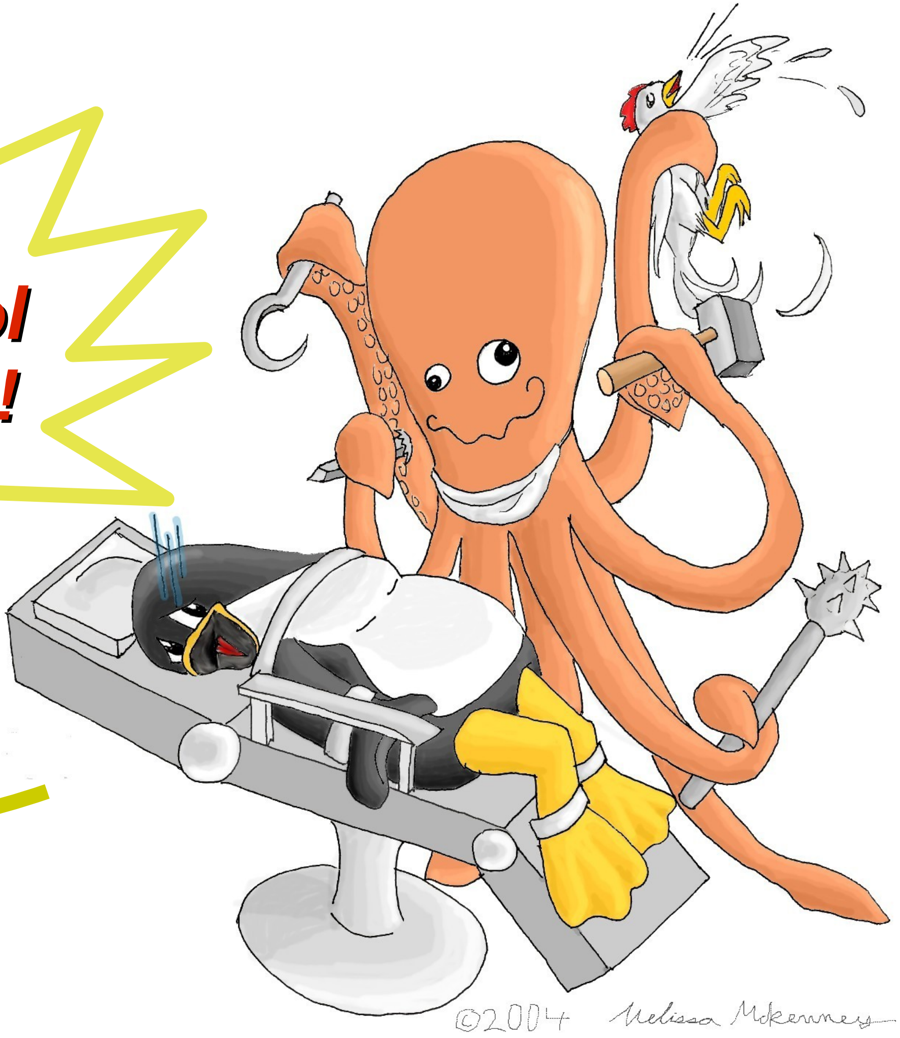


Image copyright © 2004 Melissa McKenney

To Probe Deeper

- http://rt.wiki.kernel.org/index.php/Main_Page
- http://people.redhat.com/mingo/realtime-preempt/
  - But now: http://www.kernel.org/pub/linux/kernel/projects/rt/
- http://www.linuxjournal.com/article/9361 (Linux Journal article)
- http://www.ibm.com/common/ssi/fcgi-bin/ssialias?subtype=ca&infotype=an&appname=iSource&supplier=877&letternum=ENUSZP06-0365
- http://www.linutronix.de/
- http://www.mvista.com/products/realtime.html
- Hollis Blanchard's “Virtualization – Not Just for Servers”
- My “Real Time Linux Technology: A Deeper Dive” (shameless plug)

"Controlling a laser with Linux is crazy, but everyone in this room is crazy in his own way. So if you want to use Linux to control an industrial welding laser, I have no problem with your using PREEMPT_RT." -- Linus Torvalds, July 2006

Legal Statement

This work represents the view of the author and does not necessarily represent the view of IBM.

IBM, IBM (logo), e-business (logo), pSeries, e (logo) server, and xSeries 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?