Linux Realtime Response

Challenges in Making Linux Ready for Real Time Computing

Paul E. McKenney
Distinguished Engineer
IBM Linux Technology Center
Overview

- Goals, Non-Goals, and Corollaries
- Overview of Linux Realtime Approaches
- Priority Inversion and Reader-Writer Lock
- Case Study: Signal-Delivery Latency
- Summary and Conclusions
Goals, Non-Goals, And Corollaries

- **Goals**
  - Realtime response on commodity mid-range multiprocessors
  - Common Linux-kernel code base
  - Merciless application of the 80-20 rule: do the 20% of the work required by 80% of the realtime applications now, more later

- **Non-Goals**
  - Provable “diamond-hard” realtime response (not yet, anyway)
  - Realtime response from *all* services: incrementalism instead

- **Corollaries**
  - Normal locking (priority inheritance)
  - Full POSIX semantics
  - Scalability and performance *in addition to* realtime response
Linux Realtime Goals

- SMP
- STRATEGY
- TACTICS
- COORDINATION
- ACTUATION
- SENSING
- MODULATION
- SIGNALING
- CUSTOM HARDWARE

TIMEFRAME:
- 10 s
- 1 s
- 100 ms
- 10 ms
- 1 ms
- 100 μs
- 10 μs

PERCEPTION → REACTION → COGNITION
# Linux Realtime Approaches (Violently Abbreviated)

<table>
<thead>
<tr>
<th>Project</th>
<th>Quality of Service</th>
<th>Inspection</th>
<th>API</th>
<th>Complexity</th>
<th>Fault Isolation</th>
<th>HW/SW Configs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vanilla Linux Kernel</td>
<td>10s of ms all services</td>
<td>All</td>
<td>POSIX + RT extensions</td>
<td>N/A</td>
<td>None</td>
<td>All</td>
</tr>
<tr>
<td>PREEMPT</td>
<td>100s of us Schd, Int</td>
<td>All spinlock critsect, preempt- &amp; int-disable</td>
<td>POSIX + RT extensions</td>
<td>N/A</td>
<td>None</td>
<td>All</td>
</tr>
<tr>
<td>Nested OS</td>
<td>~10 us RTOS svcs</td>
<td>RTOS + int-disable</td>
<td>RTOS</td>
<td>Dual environment</td>
<td>Good</td>
<td>All</td>
</tr>
<tr>
<td>Dual-OS / Dual-Core</td>
<td>&lt;1 us RTOS svcs</td>
<td>All RTOS</td>
<td>RTOS</td>
<td>Dual environment</td>
<td>Excellent</td>
<td>Specialized</td>
</tr>
<tr>
<td>PREEMPT_RT</td>
<td>10s of us Schd, Int</td>
<td>All preempt- &amp; int-disable (most ints in process ctxt)</td>
<td>POSIX + RT extensions</td>
<td>“Modest” patch</td>
<td>None</td>
<td>All (except some drivers)</td>
</tr>
<tr>
<td>Migration Between OSes</td>
<td>? us RTOS svcs</td>
<td>All RTOS + int-disable</td>
<td>RTOS (can be POSIX)</td>
<td>Dual env. (Fusion)</td>
<td>OK</td>
<td>All?</td>
</tr>
<tr>
<td>Migration Within OS</td>
<td>? us RTOS svcs</td>
<td>Scheduler + RT syscalls</td>
<td>POSIX + RT extensions</td>
<td>Small patch</td>
<td>None</td>
<td>All?</td>
</tr>
</tbody>
</table>

[http://lwn.net/Articles/143323/](http://lwn.net/Articles/143323/) for additional detail.
Other Features That Might Appear. Someday.

- **Deterministic I/O**
  - Disk I/O – or, more likely, Flash memory
  - Network protocols
    - Datagram protocols (UDP) relatively straightforward
    - “Reliable” protocols (TCP, SCTP) more difficult
    - Maintaining low network utilization is key workaround
    - Possible contender: Van Jacobson's lock-free Linux TCP/IP work

- **Priority Inheritance Beyond Locking**
  - Reader-writer locks with concurrent readers
    - Writer-to-reader boosting problematic
  - Across RCU, especially when low on memory
  - Across memory allocation
    - Boost priority of someone who is about to free?
  - Across networks
In Some Cases, Priority Boosting is Undesirable... 

...Or At Least Uncomfortable!!!
Priority Boosting 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!
  - Further degrading P1's realtime response latency
- Linux -rt approach: only one reading task...

![Diagram]

- High-Priority Process P1
- Medium-Priority Processes (One Per CPU)
- Lock 1
  - Read Hold
  - Preempt
- Lock 2
  - Read Acquire
  - Read Hold
- Lock 3
  - Write Acquire
  - Read Hold
- Lock 4
  - Write Acquire
  - Read Hold
Priority Inversion and RCU: What is RCU?

- Analogous to reader-writer lock, but readers acquire no locks
  - Readers therefore cannot block writers
  - Reader-to-writer priority inversion is therefore impossible
- 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)
- IBM recently adapted RCU for realtime use in Linux
RCU Example: Removal From Linked List

Determine when RCU readers are done by observing states forbidden to RCU readers
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...

Diagram:
- High-Priority Process P1
  - Acquire Lock 1
- Medium-Priority Processes (One Per CPU)
  - Preempt
- Low-Priority Process P2
  - Acquire
- Low-Priority Process P3
  - Acquire
- Low-Priority Process P4
  - Acquire
- Lock 1
  - RCU
- Lock 2
  - RCU
- Lock 3
  - RCU
- Lock 4
  - RCU

Acquire
Write
Acquire
Write
Acquire
Write
## RCU Realtime Scorecard

<table>
<thead>
<tr>
<th>Feature</th>
<th>Reliable</th>
<th>Callable From IRQ</th>
<th>Preemptible Read Side</th>
<th>Small Memory Footprint</th>
<th>Sync-Free Read Side</th>
<th>Indpt of Memory Blocks</th>
<th>Nestable Read Side</th>
<th>Uncond R-W Upgrade</th>
<th>Compatible API</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classic RCU</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rcu-preempt</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Jim Houston Patch</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reader-Writer Locking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>n</td>
</tr>
<tr>
<td>Unconditional Hazard Pointers</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hazard Pointers: Failure</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>Hazard Pointers: Panic</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Hazard Pointers: Blocking</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>N</td>
</tr>
<tr>
<td>Reference Counters</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rcu_done_reference()</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Lock-Based Deferred Free</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read-Side Counter GP Suppression</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Read-Side Counters w/ “Flipping”</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Case Study: kill() System-Call Latency

- Current concern: Latency of signal transmission
  - Reduce latency effect on sending process
  - Transmission-to-reception latency not yet a problem
- kill() read-holds on tasklist_lock for mutual exclusion
  - Prevent processes and threads from changing state
- Updates to process/thread state write-hold tasklist_lock
  - fork(), exec(), exit(), change process group, setuid, ...
- But most state-changes do not affect signal delivery
  - Traditional approach: fine-grained locking or non-blocking synchronization
  - But these approaches introduce high complexity
- Alternative: use RCU instead of read-acquiring tasklist_lock
  - 2x-3x reduction in latency, small code change
  - Now in Linus's mainline kernel source tree
Summary

- Linux is making great progress in realtime latency
- Modest technical goals, striving for widespread usefulness
  - Tens of microseconds scheduling/interrupt latency
  - Similar latencies for selected operations and system calls
  - Single source base (this may take awhile)
  - Simplicity, scalability, and performance minimally degraded
  - No provable latencies – perhaps SW tools will help?
- Using old (preemption) and new (RCU) techniques
  - Preemption of RCU read-side critical sections requires innovation in RCU implementation (ongoing work)
  - Replacement of reader-writer locks with RCU requires care due to RCU readers not blocking updates (ongoing work)
- No obvious *technological* barrier to scalable realtime Linux...
- But can the Linux community handle RCU?
Can the Linux Community Handle RCU?

Seems to be doing so!!!
The views expressed in this paper are the author's only, and should not be attributed to IBM.

Linux is a trademark of Linus Torvalds in the United States, other countries, or both.

Other company, product, and service names may be trademarks or servicemarks of others.
Resources

- Discussion of realtime measures and goals
- Different approaches to Linux realtime
  - http://lwn.net/Articles/143323/
- Description of PREEMPT_RT patchset
  - http://lwn.net/Articles/146861/
- PREEMPT_RT patchset
- Victor Yodaiken dislikes priority inheritance; Doug Locke disagrees
  - http://www.linuxdevices.com/articles/AT7168794919.html
  - http://www.linuxdevices.com/articles/AT5698775833.html
BACKUP
Why Realtime Response???

- Moore's Law Now Generating Multithread/Multicore CPUs
- Consolidate Realtime Market: Improve software portability
- Customer Demand: DoD, Digital Media/Gaming, Financial
- “Nintendo Generation”
  - Grew up with sub-reflex response time from computers
  - Now are entering jobs controlling computer purchases
- Human-computer interaction changes when response time drops below about 100 milliseconds
  - Much more natural and fluid, much more productive
  - And can developed countries afford to continue to pay their people to stare at hourglasses???
    - But this problem extends far above the operating system...
- Delays accumulate across networks of machines
Isn't Realtime a Single-CPU Thing?

Today's Systems

Historical Realtime:
• Few CPUs
• Latency Guarantees
• Non-Standard

Emerging Systems

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

Convergence

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

But Not Both!!!

OR

Historical SMP:
• Many CPUs
• No Guarantees
• Standard (and OSS)
What Does Realtime Entail?

- Quality of Service (Beyond “Hard”/“Soft”)
  - Services Supported
    - Probability of meeting deadline absent HW failure
    - Deadlines supported
  - Performance/Scalability for RT & non-RT Code
- Amount of Global Knowledge Required
- Fault Isolation
- HW/SW Configurations Supported

“But Will People Use It?”
Classic RCU

- May hold reference
- Can't hold reference to old version, but RCU can't tell
- Can't hold reference to old version

CPU 0
- Context Switch
- RCU Read-Side Critical Section
- Call rcu()
- Context Switch
- RCU Read-Side Critical Section
- Context Switch
- RCU Read-Side Critical Section

CPU 1
- Remove
- Call rcu()
- Context Switch
- RCU Read-Side Critical Section
- Context Switch
- RCU Read-Side Critical Section

Can't hold reference to old version
Simple Solution: Lock-Based Defer

```c
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);
}
```
Lock-Based Defer: Grace Periods

CPU 0
- RCU Read-Side Critical Section
- Wait for CPU 0...
- Delete Element
- Acquire Lock...
- Acquired
- May hold reference

CPU 1
- RCU Read-Side Critical Section
- Can't hold reference to old version

- RCU Read-Side Critical Section
- RCU Read-Side Critical Section
- RCU Read-Side Critical Section
Problems With Lock-Based Deferral

- Latency can “bleed” from one reader to another via updater
  - Reader 1 read-holds lock
  - Updater blocked attempting to write-acquire lock
  - Reader 2 blocked attempting to read-acquire lock
    - Allowing Reader 2 to precede Updater results in starvation
- Use of RCU in interrupt handlers can result in self-deadlock
  - These deadlocks could be avoided by masking interrupts
  - But that would defeat the whole purpose: preemptible RCU read-side critical sections

- Solution: Counter-based scheme
## Counter-Based Realtime RCU

<table>
<thead>
<tr>
<th></th>
<th>Current Count</th>
<th>Previous Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>CPU 0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>CPU 1</td>
<td>2</td>
<td>0</td>
</tr>
<tr>
<td>CPU 2</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>CPU 3</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>CPU 4</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>CPU 5</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>CPU 6</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>CPU 7</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Wow. Just when I thought Linux was getting good enough, that it has all the features I need for the foreseeable future, along comes something like this that makes me say, I want I want I want!

From http://lwn.net/Articles/129511/