Linux Realtime Response:
The \texttt{CONFIG\_PREEMPT} Patch Set
Overview

- Production Systems and Realtime Response
- Isn't Realtime a Single-CPU Thing?
- What Does Realtime Entail?
- Linux Approaches to Realtime Response
- CONFIG_PREEMPT_RT Patch
- Priority Inversion and Reader-Writer Locking
- Administrative Tools
- Summary
System Administrators Must:

- 1960: Keep system running
- 1970: Control user access to system
- 1980: Keep network running
- 1990: Keep system performing and scaling
- 2000: Keep cluster/datacenter running
- **2010: Keep system responding in real time**
- 2020: Keep Internet responding in real time?
  - Or maybe just cluster/datacenter...
Why Realtime Response??

- Moore's Law: AKA “because we can”
  - Cell phones are more powerful than 1970s mainframes, and therefore can support “real” operating systems (see next slide)
- Software “network effects”: common platform & software
- “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
Moore's Law as Illustrated by Sequent Computers
Isn't Realtime a Single-CPU Thing?

Today's Systems

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

OR

Historical SMP:
- Many CPUs
- No Guarantees
- Standard (and OSS)

Emerging Systems

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

Convergence

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

But Not Both!!!
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?”
# 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 services</td>
<td>all services</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>
Examples of Linux Approaches

- Nested OS:
  - RTLinux, L4Linux, I-pipe (latency from RTLinux)

- Dual-OS/Dual-Core:
  - Huge numbers of real products, e.g., cell phones

- Migration Between OSes:
  - RTAI-Fusion

- Migration Within OS:
  - ARTiS (Asymmetric Real-Time Scheduling)
**Related Patches & Components**

- **High-Resolution Timers (HRT)**
  - Avoids “three-millisecond shuffle”
  - Additional code provides fine-grained timers
  - “ktimers” seems to be superseding HRT
- **Variable idle Sleep Time (VST)**
  - Suppress unneeded timer ticks, CONFIG_VST
  - Also helps virtualization/consolidation
- **Robust Mutexes / “fusyn”**
  - Priority inheritance for user-level mutexes
    - Such as pthread_mutex
- **Isolcpus + interrupt-shielding patches & config options**
Other Patches 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

- **Other Priority Inheritance**
  - Across memory allocation
    - Boost priority of someone who is about to free...
  - Reader-writer locks with concurrent readers
    - Writer-to-reader boosting problematic
  - Across networks (automated cattle prod for users???)
  - Across RCU when OOM (this one is straightforward!)
Leverage Linux Kernel's SMP Capability

- Any code segment must be able to tolerate interference from some other CPU
  - That is what SMP locking is all about, after all!!!
- This property can be leveraged to support "macho preemption"

But no need to actually remove a CPU
- No high-overhead CPU-hotplug events, please!
CONFIG_PREEMPT_RT Patch: Philosophy

Happy coincidence: that which helps scalability usually also helps realtime latency!!!
CONFIG_PREEMPT_RT Patch: Caveats

- Some Changes Were Required
  - Spinlocks can now sleep
    - “Raw” spinlock facility for the few locks that cannot tolerate sleeping (e.g., scheduler locks)
  - Must now explicitly protect per-CPU variables
  - Explicitly disable preemption or interrupts
  - Use get_cpu_var() API
  - Use DEFINE_PER_CPU_LOCKED() facility
    - Avoids realtime latency degradation
  - Interrupt handlers can now be preempted
    - As can “interrupt disable” code sequences
  - But Numerous SMP Bugs Were Located!
Preempting Interrupt Handlers: IRQ Threads

1. Mainline Code → Interrupt → IRQ Handler → Return From Interrupt → Mainline Code


  _IRQ Thread_
Preempting Interrupt Handlers: IRQ Threads

Mainline Code \rightarrow \text{IRQ} \rightarrow \text{Mainline Code}

Return From Interrupt

Preempting Realtime Thread

IRQ Handler
In-Kernel Primitives

- So what does it mean to disable interrupts???
  - Disabling preemption will do the trick
    - And so local_irq_disable() and friends disable preemption
  - But disabling preemption degrades latency, so use of locks is usually preferable
  - Except that the scheduling-clock interrupt is still a “real” interrupt
    - Marked with SA_NODELAY
  - So raw_local_irq_disable() and friends disable “real” interrupts
- Per-CPU variables prone to preemption, so “locked” per-CPU variables
  - DEFINE_PER_CPU_LOCKED, DECLARE_PER_CPU_LOCKED, get_per_cpu_locked, put_per_cpu_locked, per_cpu_lock, per_cpu_locked
More In-Kernel Primitives

- spinlock_t is preemptible and participates in priority inheritance
  - But the runqueue spinlocks cannot be preempted (why?)
  - So there is raw_spinlock_t for “pure spinlock”
- Ditto for rwlock_t and raw_rwlock_t
- seqlock_t is preemptible, and participates in priority inheritance on the update side
- struct semaphore participates in priority inheritance
  - But priority inheritance does not make sense in event mechanisms (why?)
  - So there is a struct compat_semaphore with no inheritance
- Ditto for struct rw_semaphore and struct compat_rwlock_semaphore
Semaphores as Event Mechanisms

- Semaphores have associated “count”, initialize to “1” for sleeplock
  - First task's “down()” proceeds
  - Second task's “down()” blocks until first task does “up()”
  - Any task doing a “down()” must eventually do an “up()”
  - So if blocked on down(), give priority to whoever succeeded on last “down()” so that they get to their “up()” more quickly
- Initialize count to “0” for event
  - First task's “down()” blocks: wait for event
  - Task that detects event does “up()”
  - How to tell which task will detect event?
  - And why would raising that task's priority make the event happen more quickly???
    - “Are we there yet?”
- Thus: priority-inheritance-immune compat_semaphore for events
Priority Inversion

- 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

- 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
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

![Diagram showing lock acquisition and priority inheritance](image)
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

Need something better...

Linux provides RCU!
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

Readers and Updaters Use Memory Barriers As Needed by CPU Architectures (Linux Handles This)

Remover Identifies Removed Objects

Readers Indicate When Done

Remover

Reclaimer
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
- Low-Priority Processes P2, P3, P4
  - RCU
  - Acquire Locks 2, 3, 4
  - Write

Medium-Priority Processes (One Per CPU)
Realtime and RCU

- RCU exploited in PREEMPT_RT patchset to reduce latencies
  - “kill()” system-call RCU prototype 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
Provable Realtime Guarantees

- Linux approaches to realtime reduce amount of code that must be inspected in order to derive realtime guarantees
  - In PREEMPT_RT patchset, only need to inspect code with:
    - Interrupts disabled
    - Preemption disabled
    - High-latency hardware interactions
- However, commercial market is primarily soft realtime rather than hard realtime
  - Needed soft-realtime guarantees established via testing
Tools and Systems Administration

- Linux has plenty of fault-isolation tools
  - “ps”, “top”, network monitoring, memory consumption, resource limits, error logging, ...
  - Intent: find functional and performance problems

- Linux will need latency-isolation tools
  - Determine what is imposing poor latency
    - Report and/or fix problem
    - Avoid using problematic part of system
  - These are starting to appear...
Tools & Systems Administration: CONFIG Options

- CRITICAL_PREEMPT_TIMING: measure maximum time that preemption is disabled
- CRITICAL_IRQSOFF_TIMING: measure maximum time that hardware interrupts are disabled
- DETECT_SOFTLOCKUP: dump stack of any process spending more than 10 seconds in kernel without rescheduling
- LATENCY_TRACE: record function-call traces of long-latency events
- RT_DETECT_DEADLOCK: find deadlock cycles
- RTC_HISTOGRAM: generate latency histograms
- WAKEUP_TIMING: measure maximum time from when high-priority task is awakened until it actually starts running
Summary

- Realtime requirements will start appearing more widely
- SMP systems starting to support realtime, courtesy of commodity realtime (multicore, multithreaded) SMP hardware
- Systems administrators will start needing to worry about realtime latency
  - Just as they started worrying about users, networks, performance, clustering, and so on...
- Tools to measure and manage latency are starting to appear, but are in their infancy

- Computing will continue to be exciting!!!