Real Time Linux patches: history and usage

Presentation first given at: FOSDEM 2006

Embedded Development Room

See www.fosdem.org

Klaas van Gend

Field Application Engineer for Europe

Why Linux in Real-Time Systems?

Not because of the Kernel's Real-Time Performance!

- UNIX-legacy Operating Systems were designed based operating principles focused on **throughput** and **progress**
- Fairness, progress and resource-sharing conflict with the requirements of time-critical applications
- UNIX systems (and Linux) are historically not Real-Time OS

In 2005, Linux RT Technology advanced dramatically

- Real-Time Linux *can* now be used a RTOS Kernel

Real-Time and Linux Kernel Evolution

montavista

- Gradual Kernel Optimizations over Time
  - SMP Critical sections (Linux 2.x)
  - Low-Latency Patches (Linux 2.2: Ingo Molnar/ Audio Community)
  - Preemption Points / Kernel Tuning (Linux 2.2 / 2.4)
  - Preemptible Kernel Patches (Linux 2.4) (Robert Love)
  - Fixed-time “O(1)” Scheduler (MontaVista -> Ingo Molnar)
  - Voluntary Preemption (Ingo Molnar)
  - Real–Time Preemption (MontaVista → Ingo Molnar)

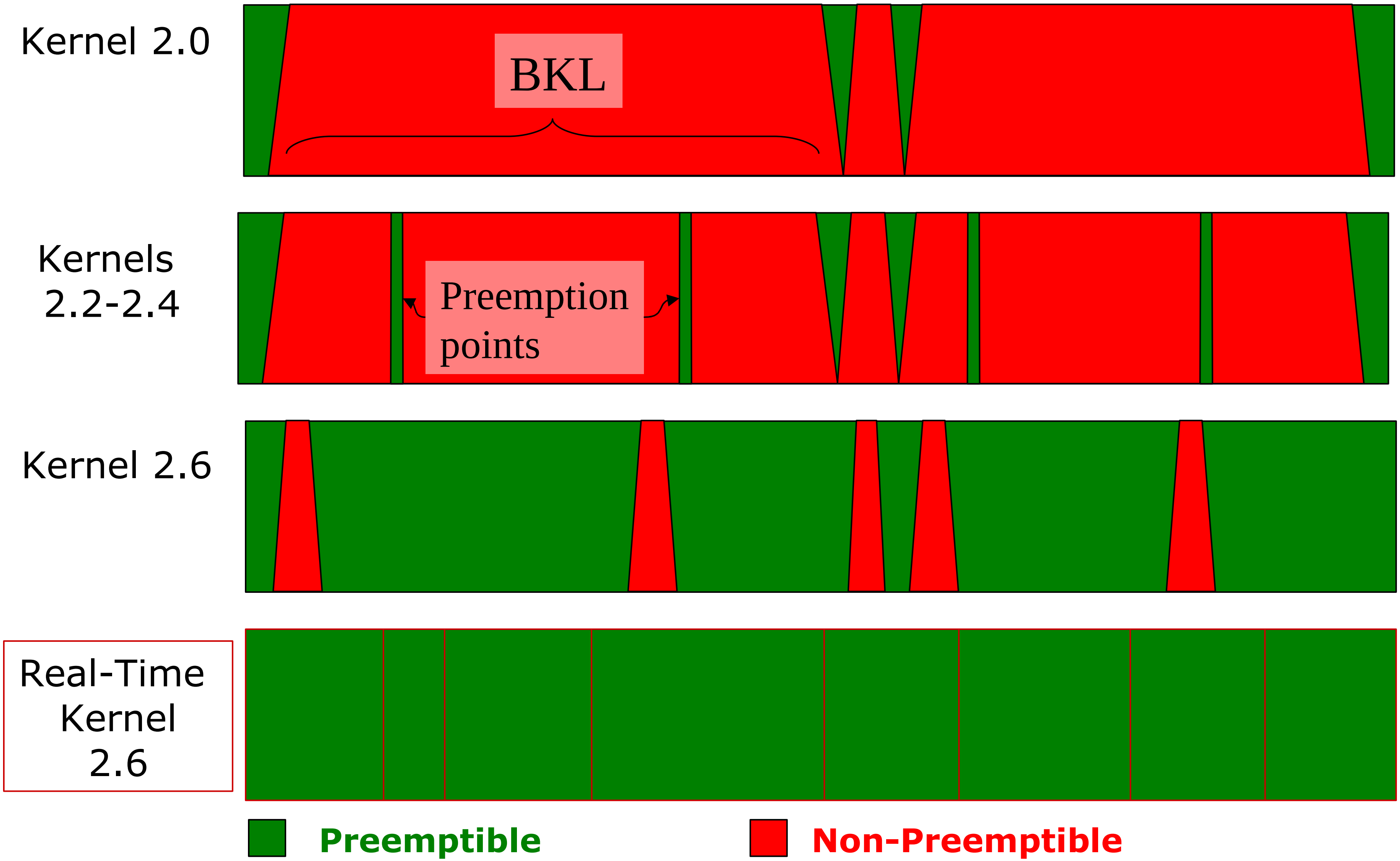
Real-Time and Linux Kernel Evolution

montavista

- Gradual SMP-Oriented Linux Kernel Optimizations

Early Kernel 1.x	No Kernel preemption
SMP Kernel 2.x	No Kernel preemption, “BKL” SMP Lock
SMP Kernel 2.2 - 2.4	No preemption, Spin-locked Critical Sections
“Preempt” Kernel 2.4	Kernel Preemption outside Critical Sections Spin-locked Critical Sections
Current Kernel 2.6	Kernel Preemption outside Critical Sections, Preemptible “BKL”, O(1) Scheduler
“RT-Preempt” Kernel	Kernel Critical sections Preemptible IRQ Subsystem Prioritized and Preemptible Mutex Locks with Priority Inheritance

Kernel Evolution: Preemptible Code



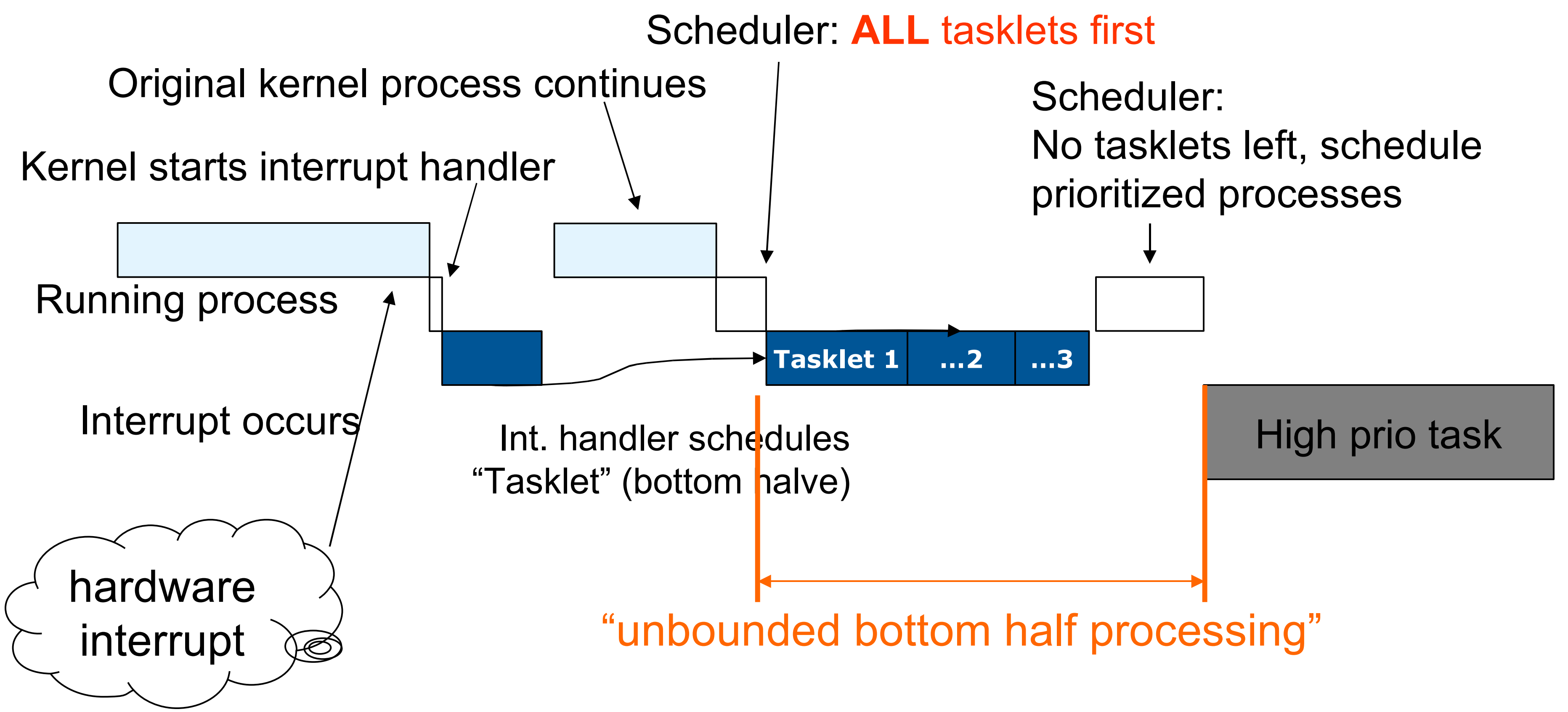
Linux Real-Time Technology Overview

montavista

- Linux 2.6 Kernel Real-Time Technology Enhancements
  - Preemptible Interrupt Handlers in Thread Context
  - Integrated Kernel Mutex with Priority Inheritance (PI)
    - Preemptible PI Mutex protects Kernel Critical Sections
  - PI Mutex Substituted for Non-Preemptible Kernel (SMP) Locks
    - Big Kernel Lock (BKL) converted to PI Mutex
    - Spin-Locks converted to PI Mutex
    - Read-Write Locks converted to PI Mutex
    - RCU Preemption Enhancements to support conversion to PI Mutex
  - Integrated High Resolution Timers (KTimers)
  - (Integrated User-Space Mutex)
    - Robustness / Dead-Owner
    - Priority Inheritance

Preemptible Interrupt Handlers in Thread Context

Standard Linux Interrupt Handler

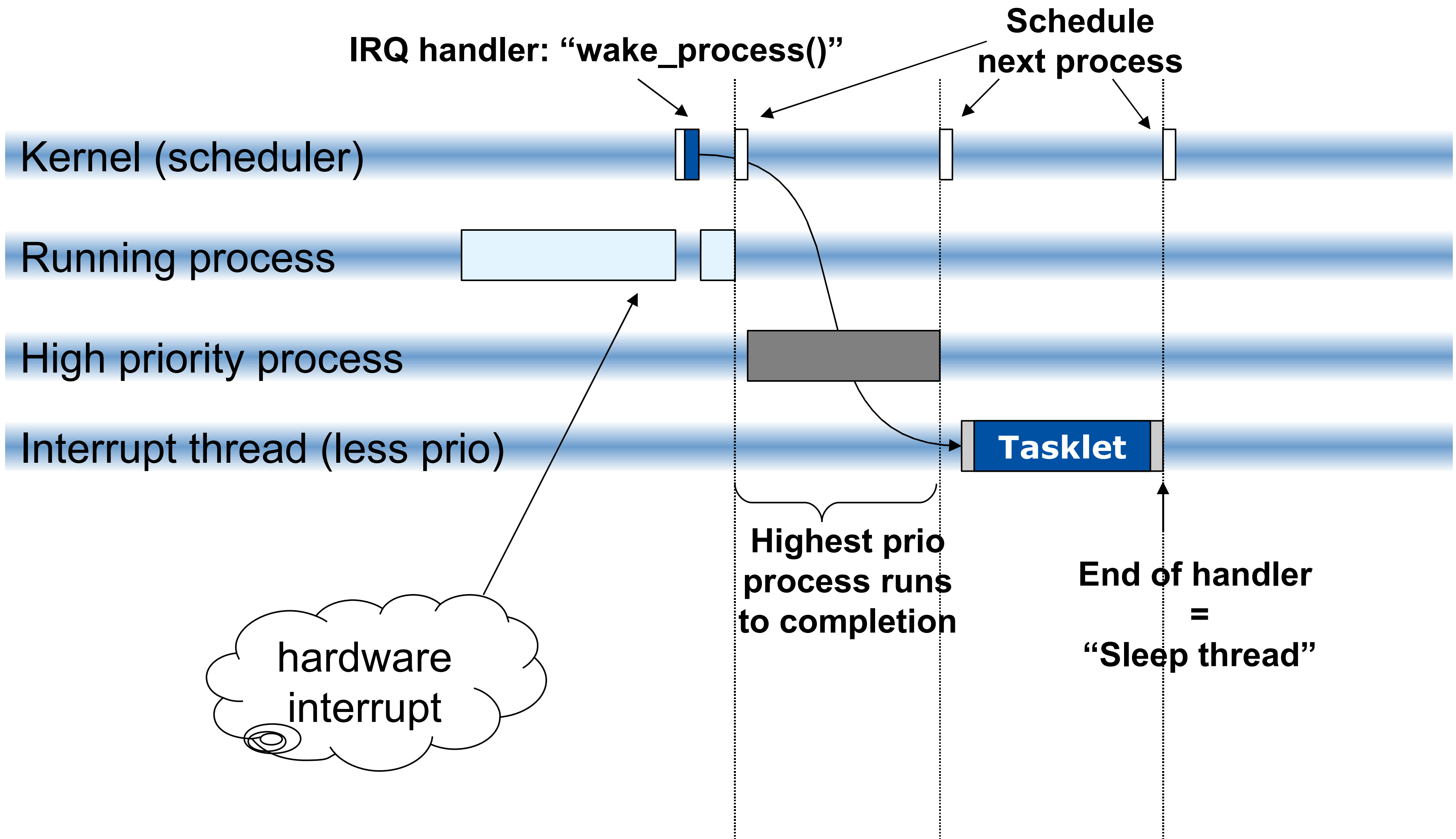


Thread-Context Interrupt Handlers

- Legacy Linux IRQ Subsystem Shortcomings
  - IRQ subsystem has unbounded latencies
  - SoftIRQ subsystem activated after IRQ handler
    - SoftIRQs can re-activate themselves holding off task execution
    - SoftIRQ daemon already defers SoftIRQ activity to task space
  - No Priorities for Interrupts
- Solution: Interrupts in Thread Context
  - Demote top- and bottom-halves to Priority Task-space
  - Real-Time tasks at Higher Priority than IRQ handlers
  - Inter-leaving of RT and IRQ tasks
  - Vacated IRQ execution-space for RT IRQ functions
    - RT IRQs do not contend with common IRQs, achieve minimal Response-time & Latency-variation

New: Thread Context Interrupt Handlers (2)

montavista



System designers now have the choice!

Thread-Context Interrupt Handlers

- Threaded IRQs Pros
  - ☑ IRQ Processing does not interfere with Task Scheduling
  - ☑ Priority Assignment Flexibility
    - Developer can create Real-Time tasks at Higher Priority than IRQ handlers
  - ☑ RT IRQs do not contend with common IRQs
    - RT IRQs see minimal Response-time & Latency-variation
  - ☑ Fully Preemptible

- Threaded IRQs Cons
  - ✖ IRQ-Thread Overhead
    - Scheduler must run to activate IRQ Threads
  - ✖ IRQ Thread Latency
    - IRQs no longer running at the highest priority
    - Full task switch required to handle IRQ
    - Response-Time / Throughput tradeoff

PI Mutex in kernel space

Kernel Locking and Preemption

- Spinlock protected code is non-preemptible
  - Linux 2.6 Kernel has 11,000 critical sections
  - Exhaustive testing of Kernel to identify worst-case
  - Labor-intensive cleanup of critical sections
  - Worst-case after cleanup still not acceptable
  - No control over 3rd party drivers
  - Maintenance

Priority-Inheriting Kernel Mutex

New Kernel (+Userspace) Synchronization Primitive

- Fundamental RT Technology
  - Preemptible alternative to spin-locked / non-preemptible regions
  - Expands on "Preemptible Kernel" Concept
  - Spinlock typing preserved *(maps spin_lock to RT or non-RT function)*
- Enabler for User-space Real-Time Condition Variables & Mutexes
- Priority Inheritance
  - Eliminate Priority Inversion Delays
- Priority-ordered O(1) Wait Queues
  - Constant-time Waiter-list Processing
  - Minimize Task Wake-Up Latencies
- Deadlock Detect
  - Identify Lock-Ordering Errors
  - Reveal Locking Cycles

π

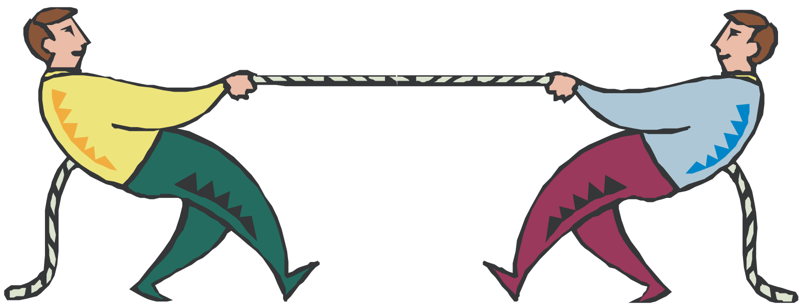
Real-Time Response vs. Throughput

montavista

Efficiency and Responsiveness are Inversely Related

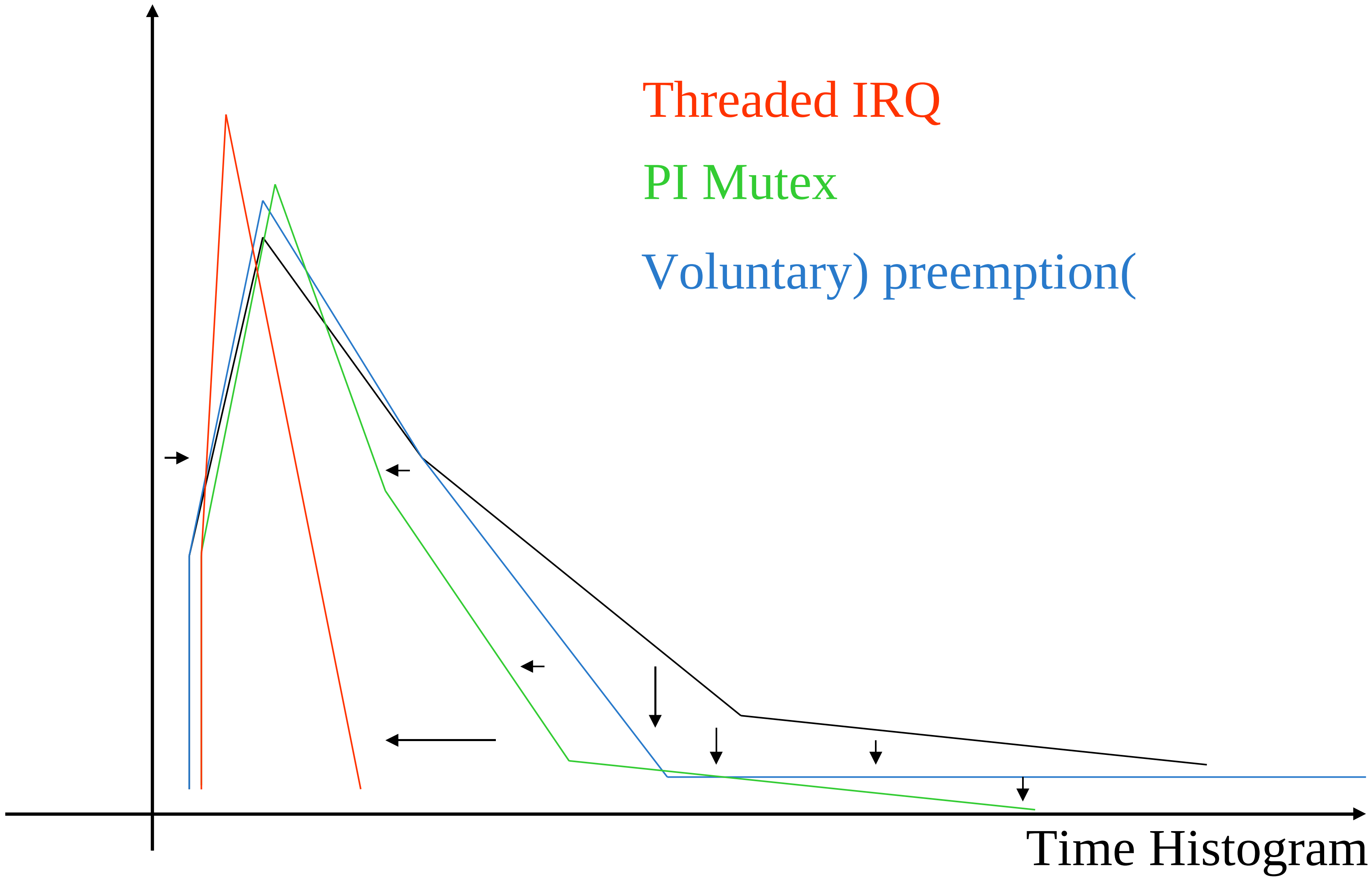
- Overhead for Real-Time Preemption
  - Mutex Operations more complex than Spinlock Operations
  - Priority Inheritance on Mutex increases Task Switching
  - Priority Inheritance increases Worst-Case Execution Time

- Design flexibility allows much better worst case scenarios
  - Real-time tasks are designed to use kernel resources in managed ways then delays can be eliminated or reduced



What does that mean?

Process preemption



Performance

Real-time Linux 2.6 Performance

- Real-Time Linux 2.6 Kernel Performance
  - Far exceeds most stringent Audio performance requirements
  - Enables sub-millisecond control-loop response
  - Enables Hard Real Time for RT-aware Applications

Linux 2.6 IRQ Latency – Hard RT IRQ Handling

montavista

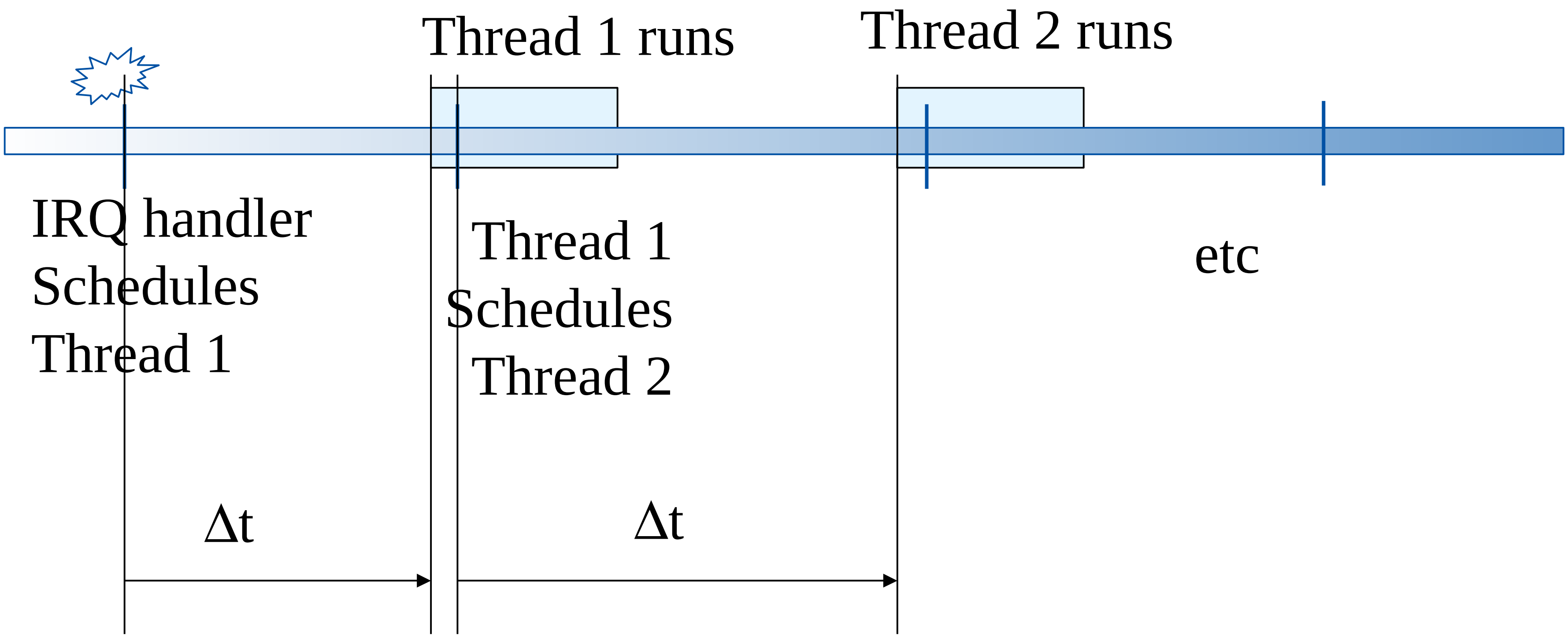
Linux-2.6.12-rc6-RT vs. Adeos / I-Pipe

Kernel	sys load	Aver	Max	Min	StdDev
Vanilla-2.6.12-rc6	None	13.9	55.5	13.4	0.4
	Ping	14.0	57.9	13.3	0.4
	lm. + ping	14.3	171.6	13.4	1.0
	lmbench	14.2	150.2	13.4	1.0
	lm. + hd	14.7	191.7	13.3	4.0
with RT-V0.7.48-25	None	13.9	53.1	13.4	0.4
	Ping	14.4	56.2	13.4	0.9
	lm. + ping	14.7	56.9	13.4	1.1
	lmbench	14.3	57.0	13.4	0.7
	lm. + hd	14.3	58.9	13.4	0.8
with Ipipe-0.4	None	13.9	53.3	13.5	0.8
	Ping	14.2	57.2	13.6	0.9
	lm.+ ping	14.5	56.5	13.5	0.9
	lmbench	14.3	55.6	13.4	0.9
	lm. + hd	14.4	55.5	13.4	0.9

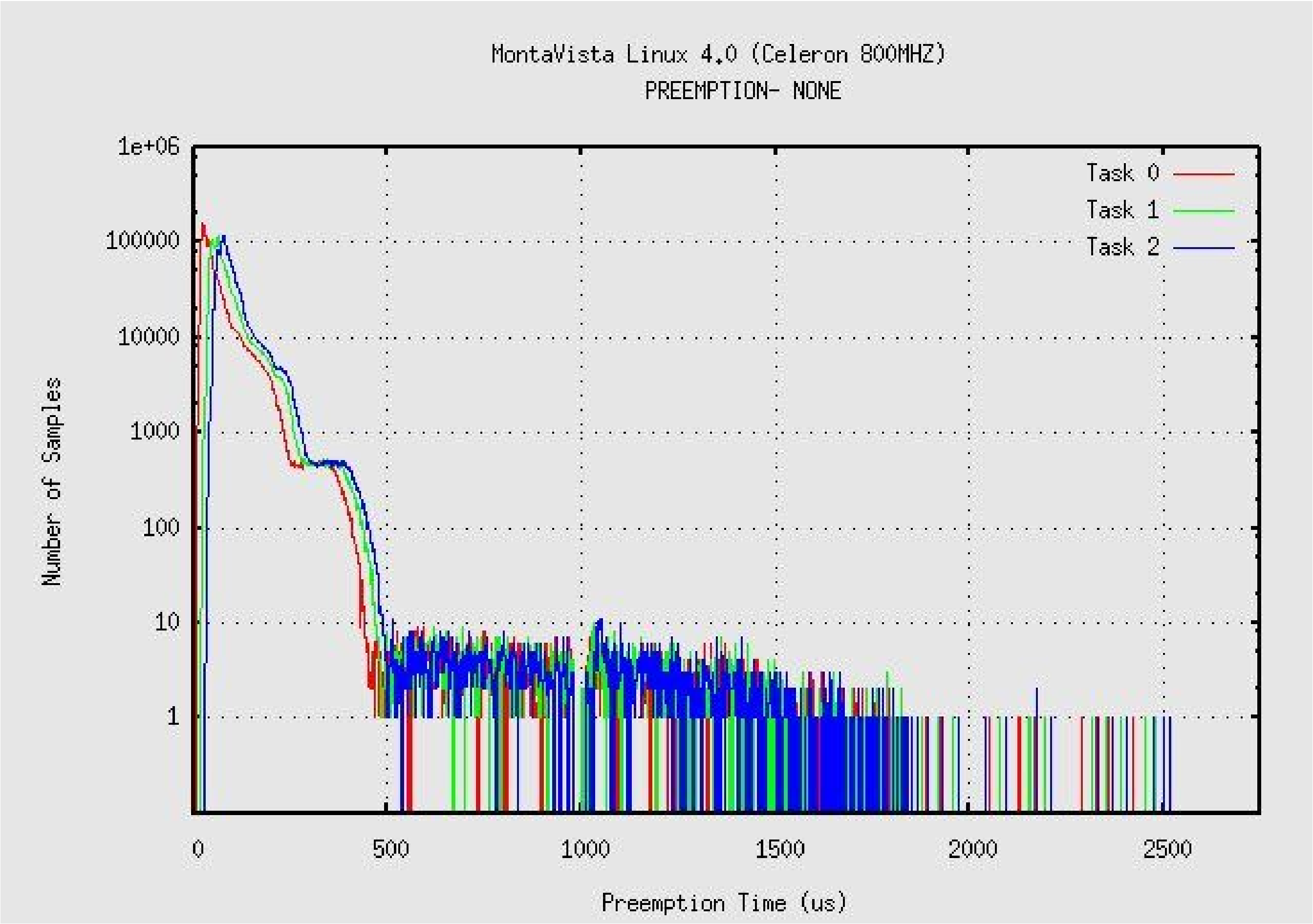
Benchmarks

- **Target machine:**
  - Intel® Celeron® 800 MHz
- **Workload applied to the target system:**
  - Lmbench
  - Netperf
  - Hackbench
  - Dbench
  - Video Playback via MPlayer
- **CPU utilization during test:**
  - 100% most of the time
- **Test Duration:**
  - 20 hours

Fast Real-time Domain Measurement tool

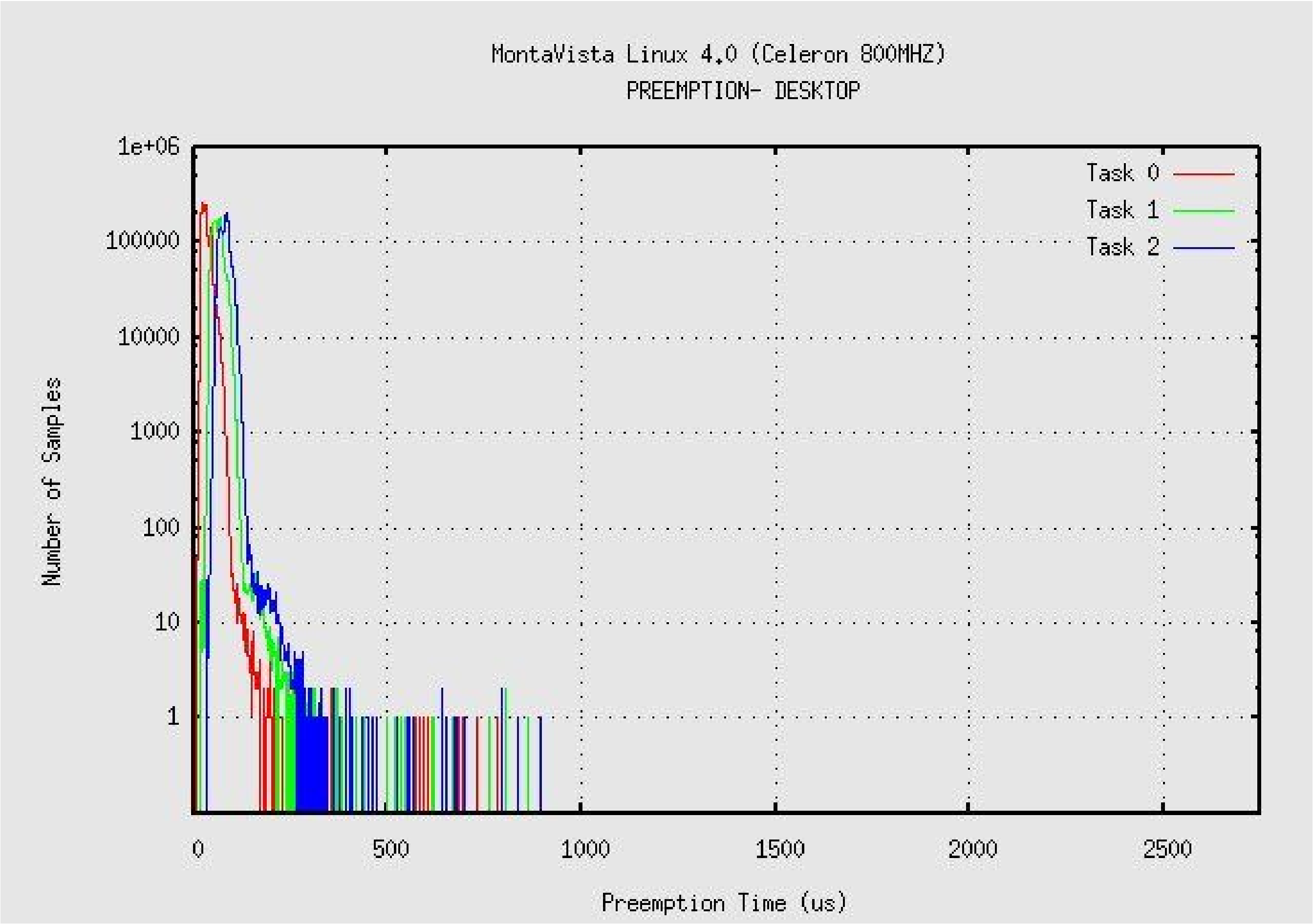


Linux 2.6 Kernel – No Preemption



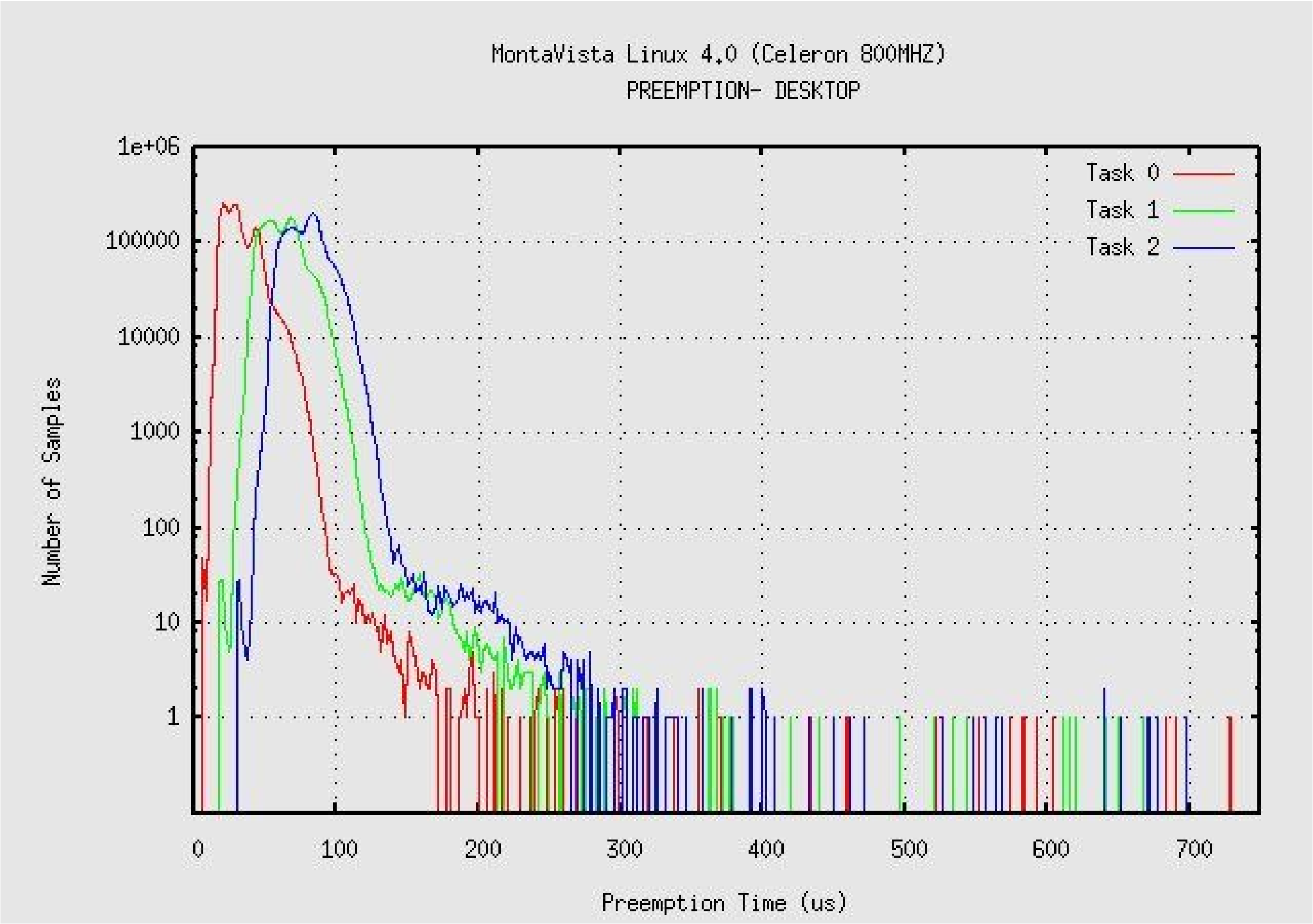
Source:

Linux 2.6 Kernel – Preemption



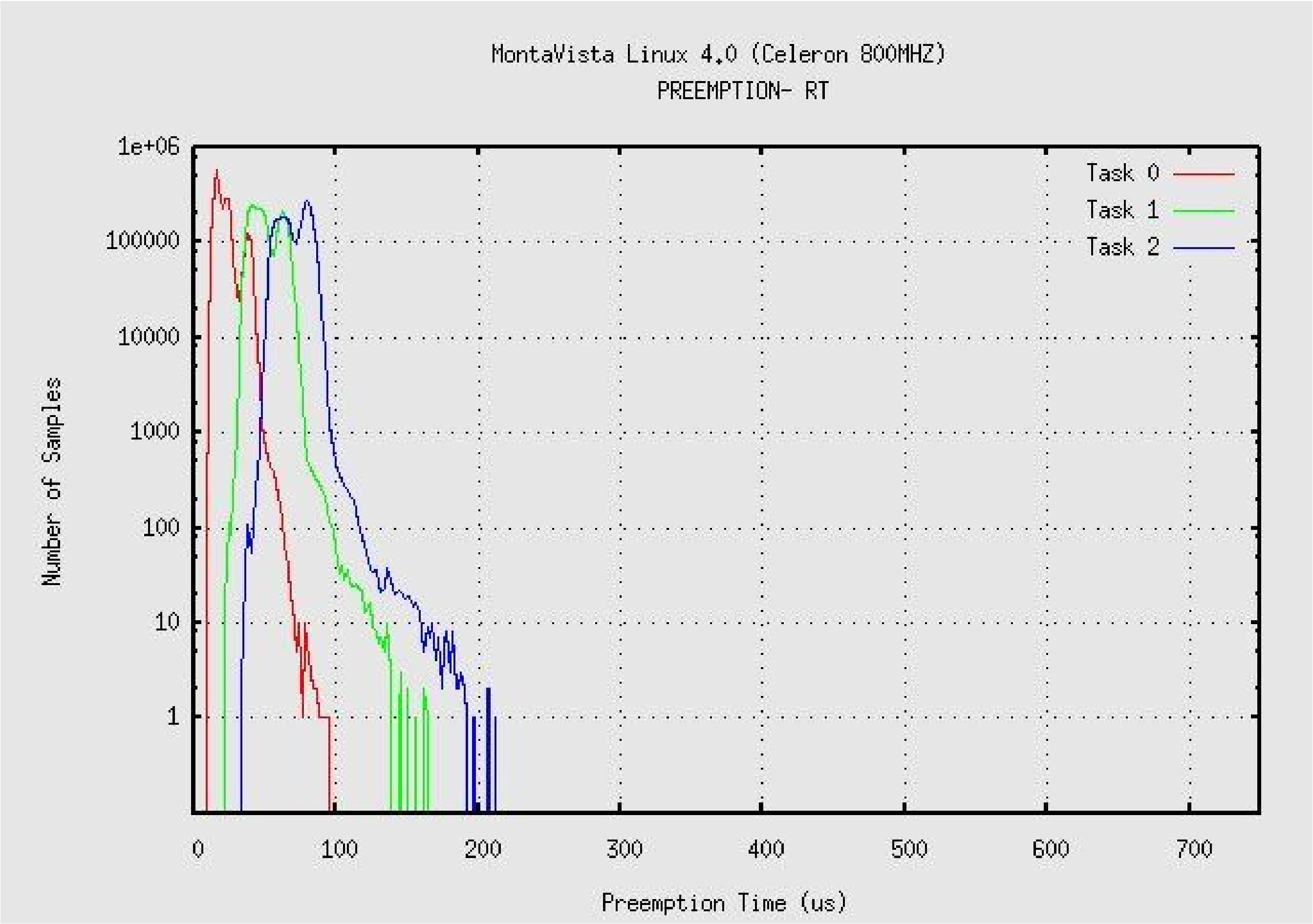
Source:

Linux 2.6 Kernel – Preemption (scaled)



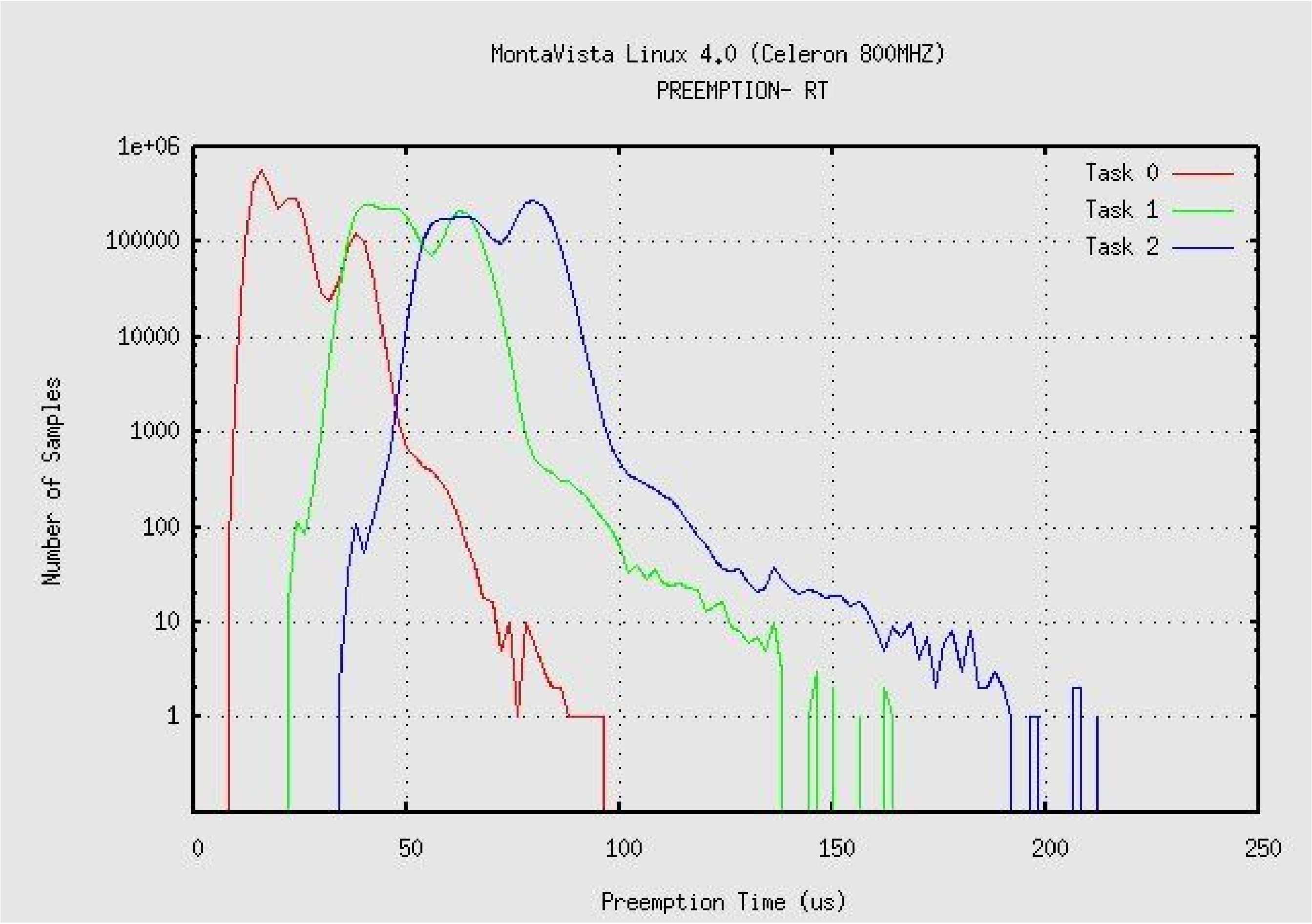
Source:

Linux 2.6 Kernel - RT Preemption



Source:

Linux 2.6 Kernel - RT Preemption (scaled)



Source:

Userspace mutex

Requirements on user space mutex

montavista

A cool new user space mutex should have:

- Priority inheritance (PI)
  - Protect user space against priority inversion
  - Preferably same mechanism as in kernel
- Robustness
  - If a mutex is held by a process that died, the mutex will be released again
- Priority Queuing (PQ)
  - If multiple threads are waiting, wake up the highest priority thread
  - Instead of “the first one” or “the first we come across”
- Deadlock Detect

π

Both PI and PQ require the current mutex owner to be known.

- Thus process lists need to be maintained

Real Time Mutex

The new RT kernel mutex already features:

- **Priority Inheritance**
- **Priority Queuing**
- **Deadlock Detect**

Missing is:

- **Robustness**
  - Since Robustness is only needed in userspace it would make sense in a kernel mutex.

Existing code - fusyn

- “Dead” project
- Unfortunately, used by most carrier grade linuxes
- No link with kernel mutex

Existing Interface – Futex

- **Simple Mutex with no RT functionality**
- **Complete userspace interface**
- **Already leveraged by glibc**
- **Robustness add on from Todd Kneisel**
  - The robustness add on also gave Futex a mutex owner concept which is needed for PI and PQ

Status

Linux Real-Time Technology Status

- Recent Real-Time Development
  - IRQ-Disable Virtualization (Walker) (partial, but including all drivers)
  - Enhanced APIC Support
  - Robust User-Space PI Mutexes (Kneisel / Singleton)
  - High Resolution Timers Integrated (Ktimers: Gleixner)
  - Arm Generic IRQ Subsystem Integration (King / Gleixner)
  - Mainstream Arm RT Extensions (Thomas Gleixner)
- Future Innovation
  - RT “awareness” extensions to Power-management subsystem
  - Quick CPU Power+Freq Ramp-UP when RT Task Scheduled

Real-Time Linux 2.6 Acceptance

- Community Status
  - RT Kernel Stable Development in Community
    - Steady stream of RT Patches into “–mm” and “-rc” Kernels
    - Including KTimers and new mutex implementation
  - Generic Implementation Facilitates Portability, Stability
    - Intel, AMD 32-bit and 64-bit
    - Arm
    - PPC
- Real-Time Linux 2.6 Technology Confidence
  - RT Preemption can Identify Hard-to-find SMP Bugs
    - Concurrency bugs easier to trace on UP Systems
    - Sanctioned by Kernel Summit as Constructive R & D
    - Voluntary Preemption Merged into 2.6.13
  - Growing Community awareness of Performance Issues
  - Audiophile Linux Distributions Shipping RT Kernel

Real world usage

Questions?

Platform to Innovate

Backup slides

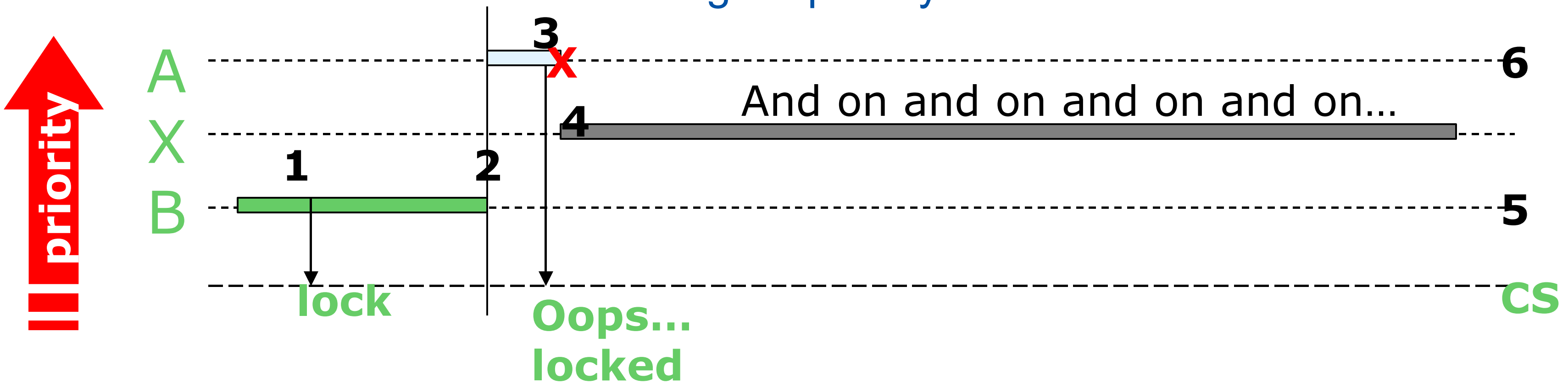
New Kernel Preemption Modes

- No forced preemption (server mode)
  - Traditional Linux non-preemptible kernel for best throughput
  - No Guarantees and long delays can occur for High Priority Tasks
- Voluntary Kernel preemption (Desktop)
  - Add explicit Preemption check-points to reduce locking time
  - Reduces maximum preemption latency, slightly lower throughput
- Preemptible Kernel (Low latency Desktop)
  - Kernel preemptible unless task is executing in SMP Critical Section
  - Best-available preemption performance in Community 2.6 kernel
- Complete Preemption (Real Time)
  - Kernel preemptible in SMP Critical Sections
  - Interrupt threads and IRQ priorities
  - Preemption Performance comparable to Sub-Kernel Performance.

(What is Priority Inversion?)

- Priority Inversion in FIFO Scheduling
  1. Process B is running and locks critical section CS_1
  2. Process B is preempted with critical section CS_1 locked
  3. Process A is scheduled and attempts to lock critical section CS_1
     1. Process A checks lock status and finds it locked by B
     2. Process A blocks and releases the CPU
  4. Process X is scheduled and becomes CPU-bound (does not block)
  5. Process B is does not get Scheduled and is starved by Process X
  6. Process A is blocked by process B holding critical section CS_1

The priority of process A > priority of X, but A does not run because X is CPU bound and higher priority than B



(What is Priority Inheritance?)

- Priority Inversion and Priority Inheritance in FIFO Scheduling
  1. Process B is running and locks critical section CS_1
  2. Process B is preempted with critical section CS_1 locked
  3. Process A is scheduled and attempts to lock critical section CS_1
     a. Process A checks lock status and finds it locked by B
     b. Process A finds priority of B < priority of A
     c. Process A saves priority of B and increases it to the priority of A
     d. Process A blocks and releases the CPU
  4. Process B is scheduled and completes its operation in critical section CS_1
     i. Process B checks lock status and finds it has inherited priority from A
     ii. Process B unlocks critical section CS_1 and resets its priority to the saved priority
  5. Process B is preempted and Process A gains access to critical section CS_1
  6. Process X is scheduled after Process A releases the CPU