

Real-Time OS Kernels

Advanced Real Time Operating Systems

Luca Abeni

luca.abeni@santannapisa.it

Latency

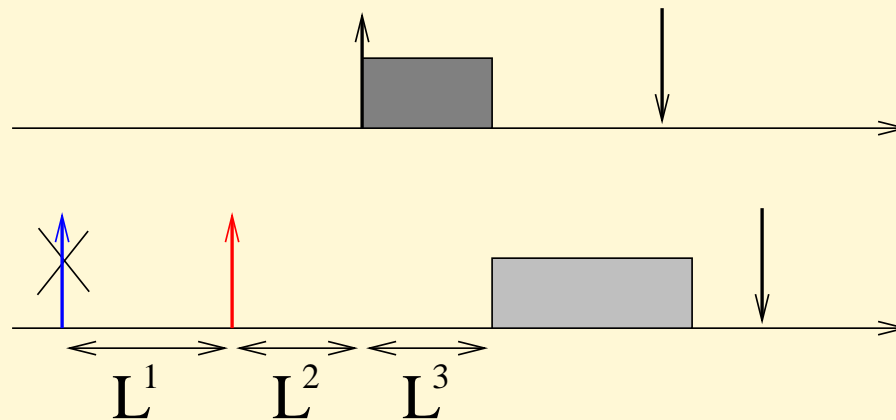
- Latency: measure of the difference between the **theoretical** and **actual** schedule
 - Task τ **expects** to be scheduled at time t ...
 - ... but **is actually scheduled** at time t'
 - \Rightarrow Latency $L = t' - t$
- The latency L can be modelled as a blocking time \Rightarrow affects the guarantee test
 - Similar to what done for shared resources
 - Blocking time due to latency, not to priority inversion

Effects of the Latency

- Upper bound for L ? If not known, no schedulability tests!!!
 - The latency must be *bounded*: $\exists L^{max} : L < L^{max}$
- If L^{max} is too high, only few task sets result to be schedulable
 - Large blocking time **experienced by *all tasks!***
 - The worst-case latency L^{max} cannot be too high

Sources of Latency

- A task τ_i is a stream of jobs $J_{i,j}$ arriving at time $r_{i,j}$
- Job $J_{i,j}$ is scheduled at time $t' > r_{i,j}$
 - $t' - r_{i,j}$ is given by:
 1. $J_{i,j}$'s arrival is signalled at time $r_{i,j} + L^1$
 2. Such event is served at time $r_{i,j} + L^1 + L^2$
 3. $J_{i,j}$ is actually scheduled at $r_{i,j} + L^1 + L^2 + L^3$

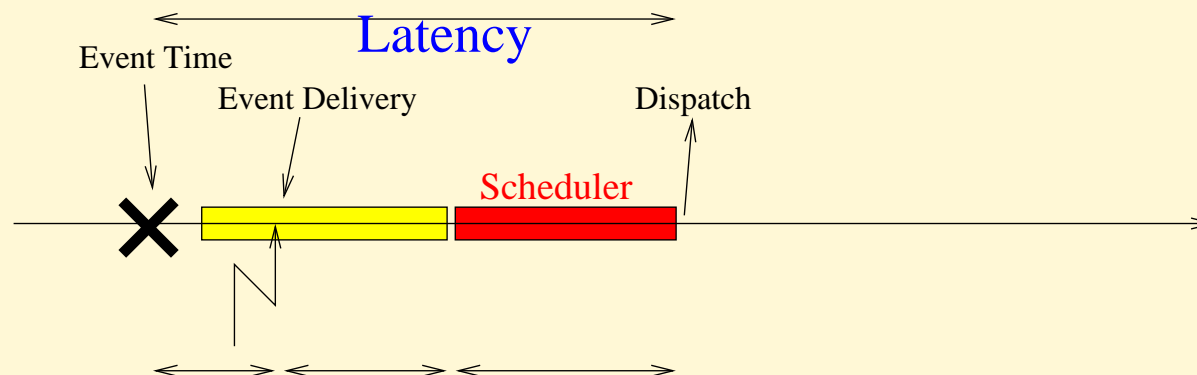


Analysis of the Various Sources

- $L = L^1 + L^2 + L^3$
- L^3 is sometimes called *scheduler latency*
 - But it is not really a latency!!!
 - Interference from higher priority tasks
 - Already accounted for by RTA / TDA or similar → let's not consider it
- L^2 is the *non-preemptable section latency* (L^{np})
- L^1 is due to the delayed interrupt generation

Non-Preemptable Section Latency

- Delay between time when an event is generated and when the kernel handles it
 - Due to non-preemptable sections in the kernel, which delay the response to hardware interrupts
 - Composed by various parts: *interrupt disabling*, *bottom halves delaying*, ...
- Depends on how the kernel handles the various events...
- Will talk about it later!



Interrupt Generation Latency

- Hardware interrupts: generated by devices
- Sometimes, an interrupt **should be generated** at time $t \dots$
- \dots but it is **actually generated** at time $t' = t + L^{int}$
- L^{int} is the *Interrupt Generation Latency*
 - It is due to hardware issues
 - It is *generally* small compared to L^{np}
 - Exception: if the device is a timer device, the interrupt generation latency can be quite high
 - *Timer Resolution Latency* L^{timer}

The Timer Resolution Latency

- Interrupt generation latency for a hw timer device
- L^{timer} can often be much larger than the non-preemptable section latency L^{np}
- Where does it come from?
 - Kernel timers are generally implemented by using a hardware device that produces periodic interrupts
- Can we do anything about it?

Example: Data Structures Consistency

- HW interrupt: *breaks* the regular execution flow
 - If the CPU is executing in US, switch to KS
- If execution is already in KS, possible problems:
 1. The kernel is updating a linked list
 2. IRQ While the list is in an inconsistent state
 3. Jump to the ISR, that needs to access the list...
- Must *disable interrupts* while updating the list!
- Similar interrupt disabling is also used in spinlocks and mutex implementations...

Real-Time Executives

- Executive: Library code that can be directly linked to applications
- Implements functionalities generally provided by kernels
- Generally, no distinction between US and KS
 - No CPU privileged mode, or application executes in privileged mode
 - “kernel” functionalities are invoked by direct function call
 - Applications can execute privileged instructions

Real-Time Executives - 2

- Advantages:
 - Simple, small, low overhead
 - Only the needed code is linked in the final image
- Disadvantages:
 - No protection
 - Applications can even disable interrupts $\rightarrow L^{np}$
risks to be unpredictable

Real-Time Executives - 3

- Consistency of the internal structures is generally ensured by disabling interrupts
 - L^{np} is bounded by the maximum amount of time interrupts are disabled
 - ...Disabled by the executive or by applications!!!
- Generally used only when memory footprint is important, or when the CPU does not provide a privileged mode
 - Example: TinyOS <http://www.tinyos.net>

Monolithic Kernels

- Traditional Unix-like structure
- Protection: distinction between Kernel (running in KS) and User Applications (running in US)
- The kernel behaves as a single-threaded program
 - One single execution flow in KS at each time
 - Simplify consistency of internal kernel structures
- Execution enters the kernel in two ways:
 - Coming from upside (system calls)
 - Coming from below (hardware interrupts)

Single-Threaded Kernels

- Only one single execution flow (thread) can execute in the kernel
 - It is not possible to execute more than 1 system call at time
 - Non-preemptable system calls
 - In SMP systems, syscalls are critical sections (execute in mutual exclusion)
 - **Interrupt handlers execute in the context of the interrupted task**

Bottom Halves

- Interrupt handlers split in two parts
 - Short and fast ISR
 - “Soft IRQ handler”
- Soft IRQ handler: *deferred* handler
 - Traditionally known as Bottom Half (BH)
 - AKA Deferred Procedure Call - DPC - in Windows
 - Linux: distinction between “traditional” BHs and Soft IRQ handlers

Synchronizing System Calls and BHs

- Synchronization with ISRs by disabling interrupts
- Synchronization with BHs: is almost automatic
 - BHs execute atomically (a BH cannot interrupt another BH)
 - BHs execute at the end of the system call, before invoking the scheduler for returning to US
- Easy synchronization, but large non-preemptable sections!
 - Achieved by reducing the kernel parallelism
 - Can be bad for real-time

Latency in Single-Threaded Kernels

- Kernels working in this way are often called *non-preemptable kernels*
- L^{np} is upper-bounded by the maximum amount of time spent in KS
 - Maximum system call length
 - Maximum amount of time spent serving interrupts

Evolution of the Monolithic Structure

- Monolithic kernels are single-threaded: how to run then on multiprocessor?
 - The kernel is a critical section: Big Kernel Lock protecting every system call
 - This solution does not scale well: a more fine-grained locking is needed!
- Tasks cannot block on these locks → not mutexes, but *spinlocks*!
 - Remember? When the CS is busy, a mutex **blocks**, a spinlock **spins**!
 - Busy waiting... Not that great idea...

Removing the Big Kernel Lock

- Big Kernel Lock → huge critical section **for everyone**
 - Bad for real-time...
 - ...But also bad for throughput!
- Let's split it in multiple locks...
- Fine-grained locking allows more execution flows in the kernel simultaneously
 - More parallelism in the kernel...
 - ...But tasks executing in kernel mode are still non-preemptable

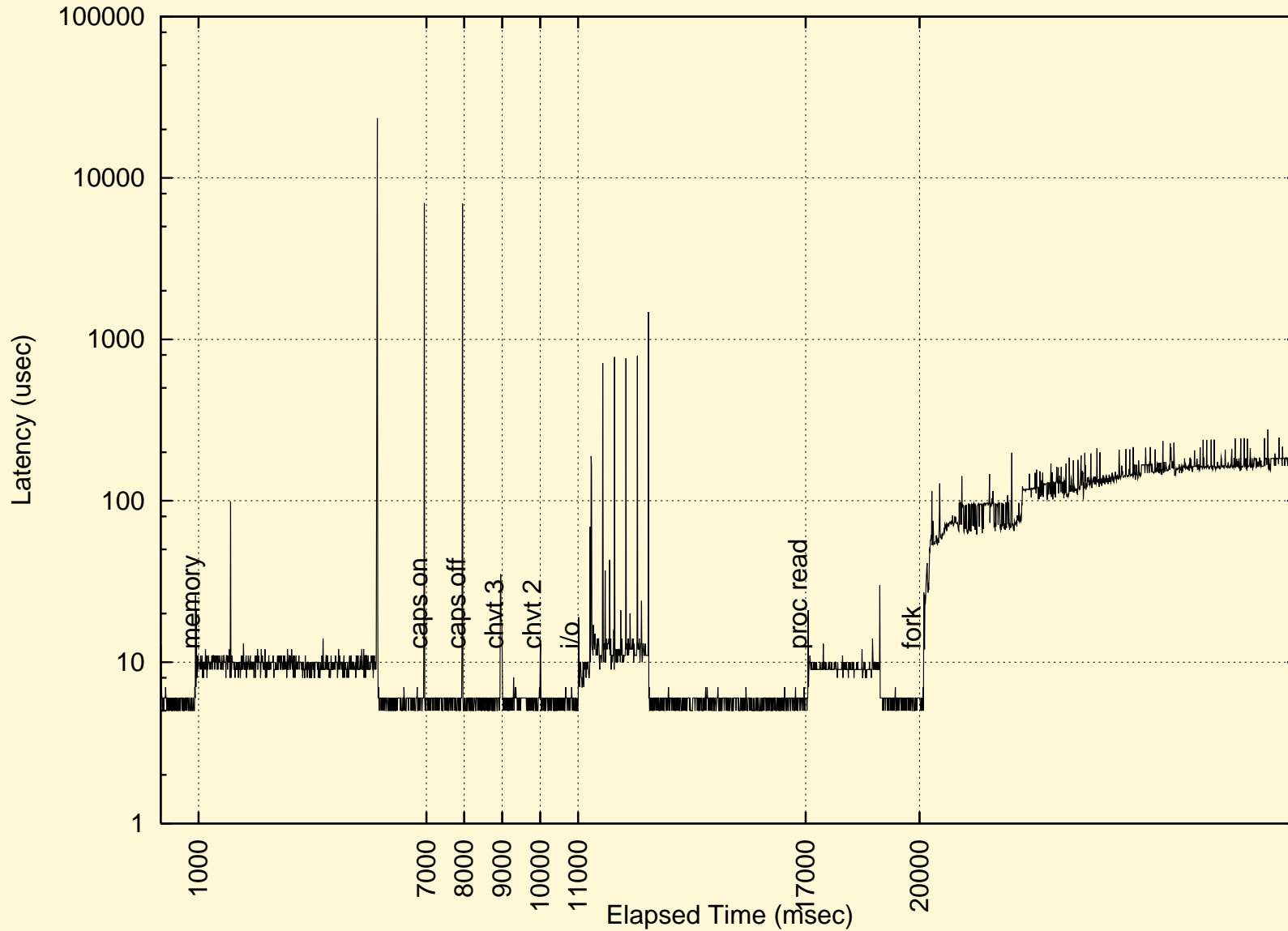
Preemptable Kernels

- Multithreaded kernel
 - Fine-grained critical sections inside the kernel
 - Kernel code is still non-preemptable
- Idea: When the kernel is not in critical section, preemptions can occur
 - Check for preemptions when exiting kernel's critical sections

Linux Kernel Preemptability

- Check for preemption when exiting a kernel critical section
 - Implemented by modifying spinlocks
 - Preemption counter: increased when locking, decreased when unlocking
 - When preemption counter == 0, check for preemption
- In a preemptable kernel, L^{np} is upper bounded by the maximum size of a kernel critical section
- Critical section == non-preemptable... **This is NPP!!!**

Latency in a Preemptible Kernel



NPP Drawbacks

- Preemptable Kernel: use NPP for kernel critical sections
- NPP is known to have issues
 - Low-priority tasks with large critical sections can affect the schedulability of high-priority tasks not using resources!
 - In this context: low-priority (or NRT) tasks invoking system calls with long critical sections can compromise the schedulability of high priority real-time tasks
 - Even if they do not use those syscalls or critical sections!
- Can we do better???

Doing Better than NPP

- Possible alternatives: **HLP** and **PI**
- HLP: easy to implement, but requires to know which resources the tasks will use
 - Possible to avoid high latencies on tasks not using the “long critical sections”, but...
 - ...Those tasks must be identified somehow!
- PI: does not impose restrictions or require a-priori knowledge of the tasks behaviour, but requires more changes to the kernel!

Using HLP

- Simple idea: distinction between RT tasks (**do not use the kernel!**) and NRT tasks (can use the kernel)
 - Do not use the kernel: simple way to avoid long critical sections!
- How the hell can we **execute a task without using the OS kernel???**
- Some “lower level RT-kernel” is needed
 - Running **below** the kernel!
 - Two possibilities: μ kernels or dual-kernel systems

- Basic idea: simplify the kernel
 - Reduce to the number of abstractions exported by the kernel
 - Address Spaces
 - Threads
 - IPC mechanisms (channels, ports, etc...)
 - Most of the “traditional” kernel functionalities implemented in user space
 - Even device drivers can be in user space!

μ Kernels - 2

- Interactions via IPC (IRQs to drivers as messages, ...)
- Servers: US processes implementing OS functionalities
 - OS kernel as a single user-space process:
Single-server OSs
 - Multiple user-space processes (a server per driver, FS server, network server, ...):
Multi-server OSs

μ Kernels vs Multithreaded Kernels

- μ Kernels are known to be “more modular” (servers can be stopped / started at run time)
- All the modern monolithic kernels provide a *module* mechanism
- Modules are linked into the kernel, servers are separate programs running in US
- Key difference between μ Kernels and traditional kernels: each server runs in its own address space
- In some “ μ Kernel systems”, some servers share the same address space for some servers to avoid the IPC overhead

Latency in μ Kernel-Based Systems - 1

- Non-preemptable sections latency is similar to monolithic kernels
 - L^{np} is upper-bounded by the maximum amount of time spent in the μ Kernel...
 - ...But μ Kernels are simpler than monolithic kernels!
 - System calls and ISRs should be shorter \Rightarrow the latency in a μ Kernel is generally smaller than in a monolithic kernel

Latency in μ Kernel-Based Systems - 2

- Unfortunately, the latency reduction achieved by the μ Kernel structure is often not sufficient for real-time systems
- Even μ Kernels **have to be modified** like monolithic kernels for obtaining good real-time performance
 - (μ) kernel preemptability, ...

2nd Generation μ Kernels

- Problems with Mach-like “fat μ Kernels”
 - The kernel is too big \rightarrow does not fit in cache memory
 - Unefficient IPC mechanisms
- Second generation of μ Kernels (“MicroKernels Can and Must be Small”): L4
 - Very simple kernel (only few syscalls)
 - Small (fits in cache memory)
 - Super-optimized IPC (designed to be efficient, not powerful)

2nd Generation μ Kernels: Performance

- L4 μ kernel: optimised for performance
 - Impact on global OS performance?
 - Real-Time performance?
- Linux ported to L4: `l4linux`
 - Single-Server OS
 - Only 10% performance penalty!
- Real-time performance: **not so good**. L4 heavily modified (introducing preemption points) to provide low latencies (Fiasco)

L4Linux

- l4linux: single-server OS, providing the Linux ABI
 - Linux applications run unmodified on it
 - Actually the server is the Linux kernel (ported to a new “l4” architecture)
- Idea: a μ Kernel is so simple and small that it does not need to be preemptable
 - False: Fiasco needed some special care to obtain good real-time performance

L4Linux and Real-Time

- Real-Time OS: DROPS
 - Non real-time applications run on l4linux (regular Linux applications)
 - Real-time applications directly run on L4
 - The l4linux server should not disable interrupts, or contain non-preemptable sections
- Use HLP instead of NPP
 - Easy to identify RT tasks: native L4 tasks!
 - The l4linux server **must never** have a priority higher than RT applications

“Tamed” L4Linux - 1

- The Linux kernel often disables interrupts (example: `spin_lock_irq()`) or preemption...
- ...So, I4linux risks to increase the latency for L4...
- Solution: in the “L4 architecture”, interrupt disabling can be remapped to a *soft interrupt disabling*
 - I4linux disables interrupts → no real `cli`
 - IPCs notifying interrupts to I4linux are disabled
 - When I4linux re-enables interrupts, pending interrupts can be notified to the I4linux server via IPC

“Tamed” L4Linux - 2

- I4linux does not really disable hw interrupts
 - L^{np} is high for the I4linux server (and for Linux applications)...
 - ...But is **very low** for L4 applications!
- I4linux cannot affect the latency experienced by L4 applications
 - HLP requires to know which applications use the resource...
 - ...In this context, it means “which applications use I4linux”

Dual Kernel Approach

- HLP idea: Linux applications are non real-time; real-time applications run at lower level
- Instead of using μ kernels, mix the real-time executive approach with the monolithic approach
 - Low-level real-time kernel: directly handles interrupts and manage the hardware
 - Non real-time interrupts: forwarded to Linux only when they do not interfere with RT activities
 - Linux cannot disable interrupts (no `cli`)
 - can only disable (or delay) interrupt forwarding
- Real-time applications cannot use the Linux kernel

RTLinux

- Dual kernel approach: initially used by RTLinux
 - Patch for the Linux kernel to intercept the interrupts
 - Small kernel module implementing a real-time executive
 - Handle real-time interrupts (low latency)
 - Forward non real-time interrupts to Linux
 - Provide real-time functionalities (POSIX API)
 - Real-time applications are kernel modules
- There is a patent on interrupt forwarding ???

RTLinux & RTAI

- RTAI: “Free” implementation of a dual-kernel approach
- Better maintained than RTLinux
- Real-time applications are Linux modules: must have an (L)GPL compatible license
- No problem in Europe, maybe subject to RTLinux patent in the US
 - Big problem for adoption in the industry
 - Would you use something that might be infringing a patent?

RTAI & Friends

- I-Pipes: Interrupt Pipelines
 - A small *nanokernel* handles interrupts by sending them to pipelines of applications / kernels that actually manage them
 - Real-time application come first in the pipeline
 - Same functionalities as RTLinux interrupt forwarding, but **different naming!**
- Described in a paper that has been **published before** the RTLinux patent → patent free

I-Pipes Implementation

- Adeos nanokernel: implements interrupt pipelines
 - Same functionalities as RTLinux, but patent-free!
 - Can be optionally used by RTAI
- Xenomai: similar to RTAI; based on Adeos
 - Provides different real-time APIs
- Xenomai 3: both dual-kernel and user-space

Summing Up...

- Monolithic kernel: high latencies (no real-time)
- Preemptible kernel: kernel critical sections → Use NPP to protect them
 - Upper bound for L^{np} , but might be too high
- μ kernel and dual-kernel: use HLP instead of NPP
 - HLP requires to know in advance which tasks will use a resource
 - Distinction between RT and NRT tasks!
- Can we do better? How to use PI???

Real-Time in Linux User Space

- HLP Idea: do not care about Linux kernel latencies, but make sure that they do not affect RT tasks
 - RT tasks: not Linux tasks!
- Real-Time performance to Linux processes \Rightarrow need to reduce L^{np} for the Linux kernel, not for low-level applications running under it
- How to reduce L^{np} ? Using PI directly is not easy...
 - There is a reason for using NPP
 - In some situations, the kernel cannot block!
 - But PI is a blocking protocol...

RT in User Space: Requirements

- Linux is a multithreaded kernel \Rightarrow need:
 1. Fine-grained locking
 2. Preemptable kernel
 3. Schedulable ISRs and BHs \Rightarrow threaded interrupt handling
 4. Replacing spinlocks with mutexes
 5. A real-time synchronisation protocol (PI) for these mutexes
- Remember Linux already provides high-resolution timers (since 2.6.21)

Using Threads for BHs and ISRs

- Using threads for serving BHs and ISRs, it is possible to schedule them
- The priority of interrupts not needed by real-time applications can be decreased, to reduce L^{np}
- Non-threaded handlers: ISRs and BHs **always preempt** all tasks!!!
 - NRT tasks can trigger high latencies by just doing a lot of I/O!!!
- Threaded handlers: if an interrupt is not needed by RT tasks, its priority can be lower than all the RT tasks priorities

Threaded Interrupt Handlers and PI

- Non-threaded ISRs \Rightarrow use spinlocks to protect data structures accessed by the ISR
 - The ISR executes in the interrupted process context \Rightarrow it cannot block
- Using threaded ISRs, spinlocks can be replaced with mutexes
- Spinlocks implicitly use NPP, mutexes can use PI!!!

The Preempt-RT Patch

- The features presented in the previous slides can surprisingly be implemented with a fairly small kernel patch
- Preempt-RT patch, started by Ingo Molnar and other Linux developers; now maintained by Thomas Gleixner
- <https://www.kernel.org/pub/linux/kernel/projects/rt>
 - Core RT patch: about 700KB of code
 - Larger patches because of added features (tracing, ...)
- Most of the code just changes spinlocks in mutexes
- Various real-time features can be enabled / disabled at kernel configuration time

Preempt-RT: Performance

- Continuous Integration and testing:

<https://www.osadl.org/QA-Farm-Realtime.qa-farm-about.0.html>

- On a standard PC, **Worst Case** kernel latency less than $50\mu s$
 - Remember: it was more than $10ms$ on a vanilla kernel!
- Much more tested than many other “RT” kernels
 - Long (continuous!) runs
 - Multiple CPUs / architectures