Real-Time Kernel Architectures

Real-Time Industrial Systems

Marcello Cinque

Roadmap

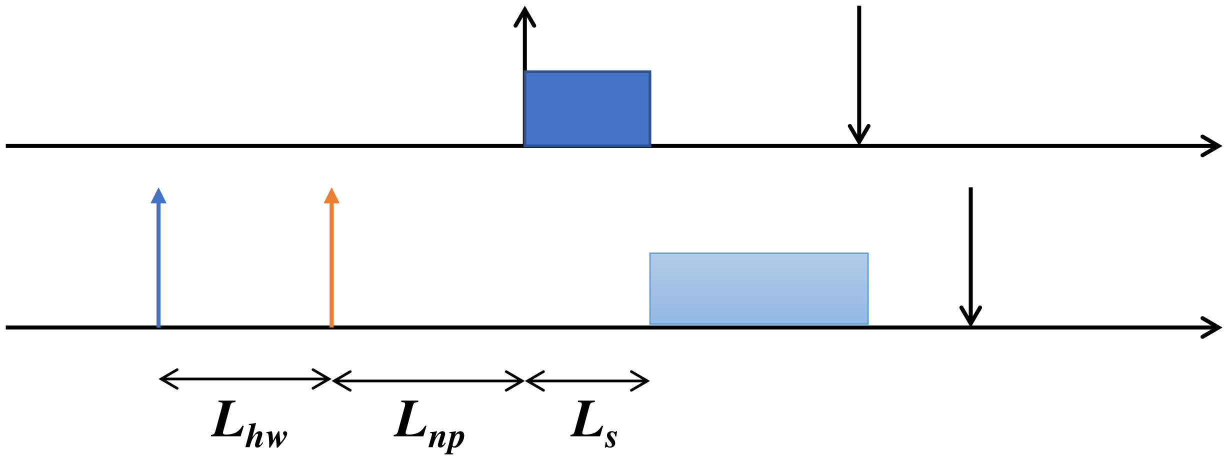
- Sources of Latency in real-time event handling
- Real-time Kernel architectural alternatives
    - Real-time executives
    - Monolithic kernels
    - Microkernels
    - Dual Kernels
    - Preemptable kernels

- References:
    - L. Abeni. “Real-Time OS Kernels”
    - Heiser, Elphistone. “ L4 Microkernels: The Lessons from 20 Years of Research and Deployment”, ACM Transactions on Computer Systems, 2016
    - F. Reghenzani et al. “The real-time Linux Kernel: a Survey on PREEMPT_RT”, ACM Computing Surveys

Latency

- Measure of the difference between the desired and the actual schedule
  - Task J expects to be scheduled at time *t*, but it is scheduled at *t'*
  - → Latency $L = t' - t$
- The latency can be modeled as a blocking time and be used in schedulability tests
- But, if not known, no schedulability tests are possible!!
- And if the maximum latency is known, but too high, only a few tasks sets will be schedulable!

Sources of Latency



- L_{hw} is due to delays in interrupt generation
- L_{np} is the non preemptable section latency
- L_s is the scheduling latency (mainly interference from high priority tasks), taken into account in schedulability tests

Questo lo abbiamo già considerato eventualmente in tutti i feasiblity test

Interrupt generation latency

- It is due to hardware issues
- it is generally small compared to other sources of latency
- But it might become large if the device is a **timer** with small resolution
  - → *timer resolution latency*
  - Kernel timers are generally implemented using devices that produce periodic interrupts
  - Can be small if high resolution timers are supported

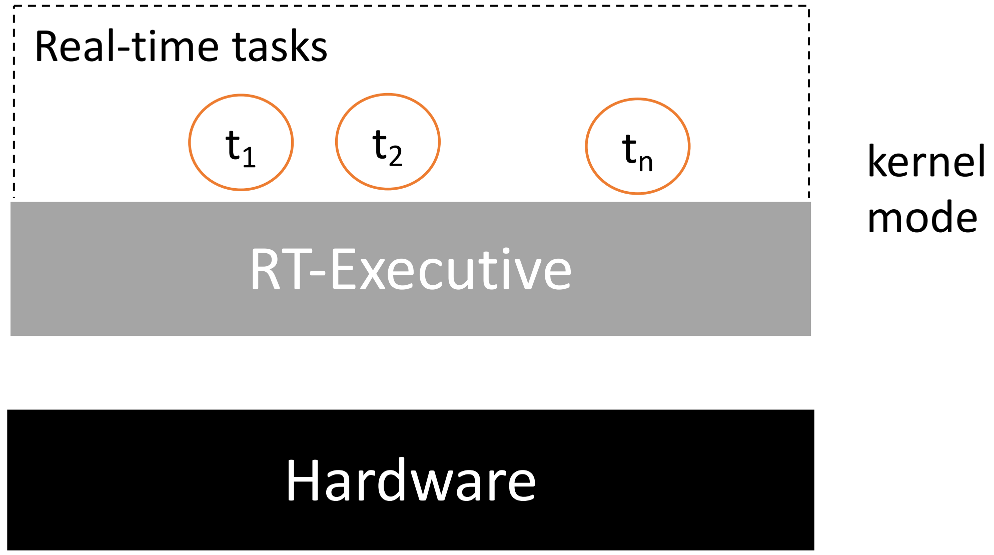
Non-Preemptable Section Latency

- Delay between an event is generated and the kernel handles it
- Due to non-preemptable sections in the kernel
  - Which are necessary to protect sensible kernel data structres

- ISRs and system calls may contain code sections that temporarily disable interrupts
  - Non-Real-Time Kernels usually adopt *spinlocks*: active wait on occupied resources!
- This is needed to avoid problems while the kernel is updating critical resources

Real-Time Executives

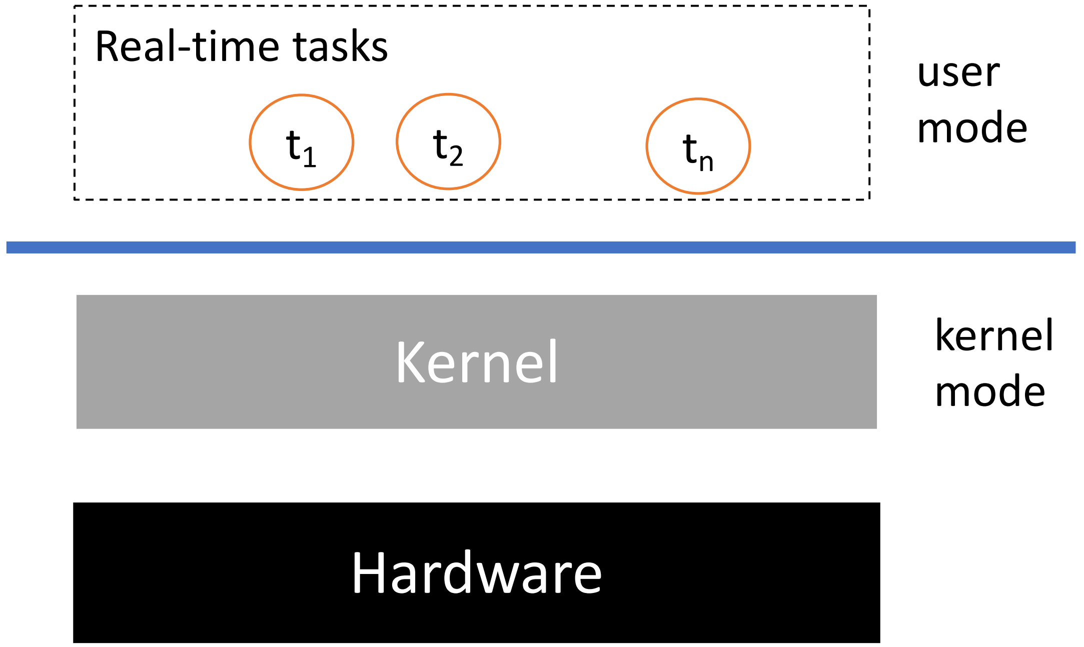
- Library OS: operating system's functions directly linked to tasks
- No distinction between user mode and kernel mode
- Applications can execute privileged code



- Simple, small, only the needed code is linked, good when small footprint is required
- BUT, dangerous as even applications can disable interrupts
- L_{np} is bounded by the maximum time the interrupts are disabled, but this also depends on applications!

Monolithic Kernels

- Traditional UNIX-like architectures
- Protection trough distinction of user mode and kernel mode
- Single-Thread model: only one execution flow in kernel space which simplify the management of consistency
  - Non-preemptable system calls
  - In SMP systems, system calls need to be executed in mutual exclusion



Single threaded kernels

- Also called ***non-preemptable kernels***
- Kernel-level synchronization on multi-processors easily solved with interrupt disabling or a unique kernel-level critical section protecting every system call with a spinlock (active wait…)
    - “**Big Kernel Lock**”
- L_{np} bounded by the maximum length of a system call or time spent serving an interrupt, but:
- Bad for real-time (and also for performance): what if a real-time task has to start while the kernel is locked?

Multi-threaded kernels

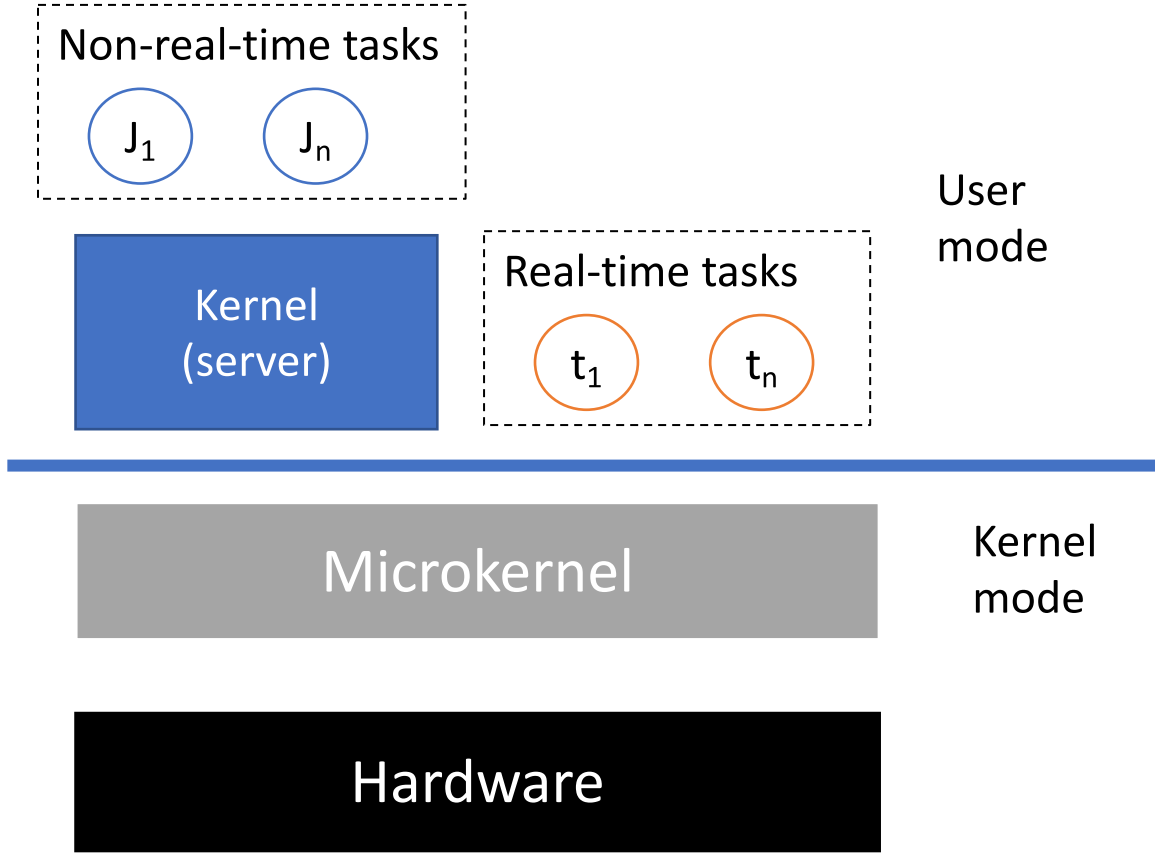
- Also called ***preemptable kernels***
- Remove the big kernel lock: allow kernel-level threads and to access sensible kernel structures using fine-grained (and non-preemptable) critical sections
- When the kernel is not in a critical section preemptions can occur
- L_{np} bounded by the maximum size of a kernel critical section
- **Non-preemptive protocol**: kernel critical sections are not preemptable!
  - If a low priority or non-real time task invokes a syscall with a long critical section this can affect the feasibility of a hard real-time task, even if it does not use that syscall!

Can we do better than NPP?

- Use HLP!
- But HLP needs to know in advance the resources used by tasks
  - How to identify such resources or tasks?
- Idea: separate real-time tasks (that do not use the kernel) from non real-time ones (that use the kernel)
- How to run a task without the kernel? → need for a low level rt-kernel
- Two approaches:
  - Microkernels
  - Dual kernels (also called co-kernels)

Microkernels

- Most of kernel functionalities are implemented in User space as "server" tasks
  - File server
  - Window server
  - Device drivers
  - ...
- The kernel implements only minimal core functionalities
  - Scheduling
  - (fast) IPC
  - Address space



HLP in microkernels

- Simple way to identify real-time tasks: they are native microkernel tasks!
- Native real-time tasks have higher priority than non-real-time tasks and OS servers and do not use kernel services
  - If an OS server needs to lock a resource, this will not affect real-time tasks

I Real-Time task devono avere ovviamente priorità maggiore di qualsiasi altro task anche durante l'esecuzione sul Kernel Server

Latency in microkernels

- The microkernel is small enough to remain non-preemptable
- L_{np} bounded by the maximum size of a microkernel critical section
- The idea is: system calls and ISRs should be shorter, so the latency in a microkernel is generally smaller than in a monolithic kernel

- … however, the first microkernels exhibited performance issues due to IPC, requiring to bring back large code portions back in the kernel
  - ”fat microkernels” → the Mach example
  - The microkernel is too big and it does not fit the cache memory!

IPC - Inter Process Comunication   Tutto viene passato con le message queue

Solutions to fat microkernels

- Keep the microkernel simple (only a few system calls)
- Keep it small (it must fit in the cache memory)
- Introduce optimized IPC (fast, efficient communication)
- Keep it non-preemptable (or introduce few strategic preemption points)

Example: **L4** microkernels family

L4

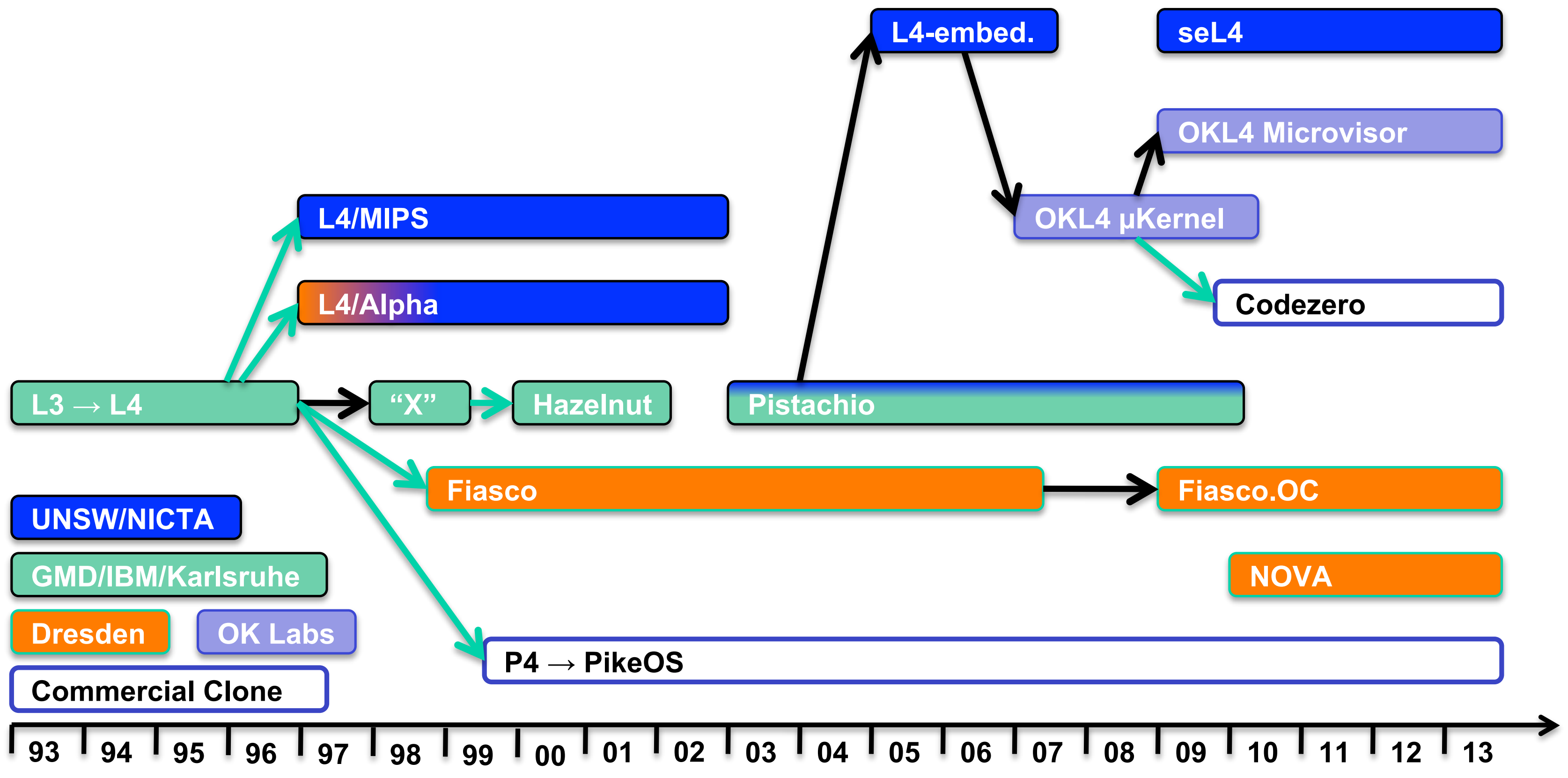
- In 1993 Liedtke demonstrated with his L4 kernel that microkernels IPC could be 20 times faster than contemporary microkernels, like Mach

- Liedtke's Minimality principle:

  «A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e. permitting competing implementations, would prevent the implementation of the system's required functionality»

The L4 genealogy

- A family of L4-based implementations have born

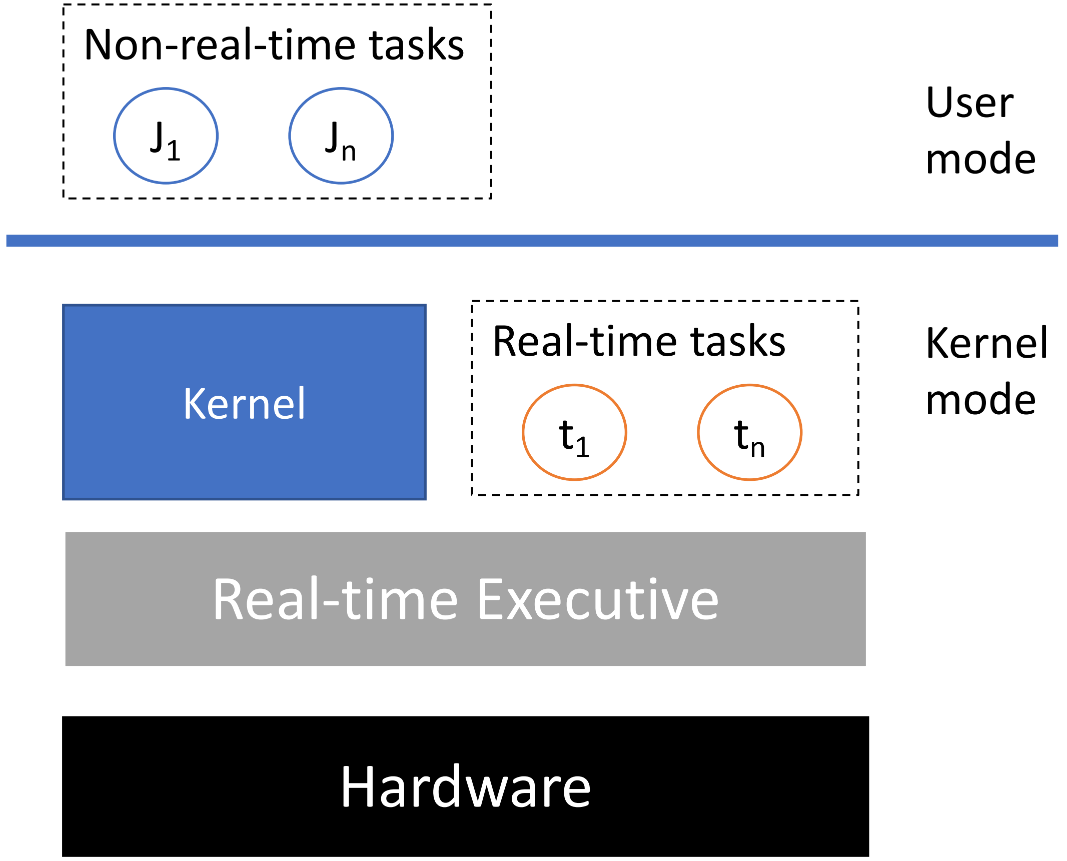


seL4

- A L4 variant designed from the beginning to support formal reasoning about security and safety, while maintaining the L4 tradition of minimality, performance, and portability
- Formal reasoning used to verify the correctness of the code, which however has to be as simple as possible (almost no preemption and revisited resource model)
- It is also the first protected-mode OS kernel in the literature with a complete and sound worst-case execution time (WCET) analysis

Dual kernels

- Mix the real-time executive approach with the monolithic kernel
- Low-level executive: directly manages the hw and handles interrupts
- Non-real time interrupts: forwarded to the Kernel only if they do not delay real-time tasks
- The kernel cannot disable interrupts
- Real-time task cannot use the kernel



HLP concept: user applications are non real-time and managed by the kernel, real-time task run at lower level with high priority than the kernel

Linux Dual Kernels: approach

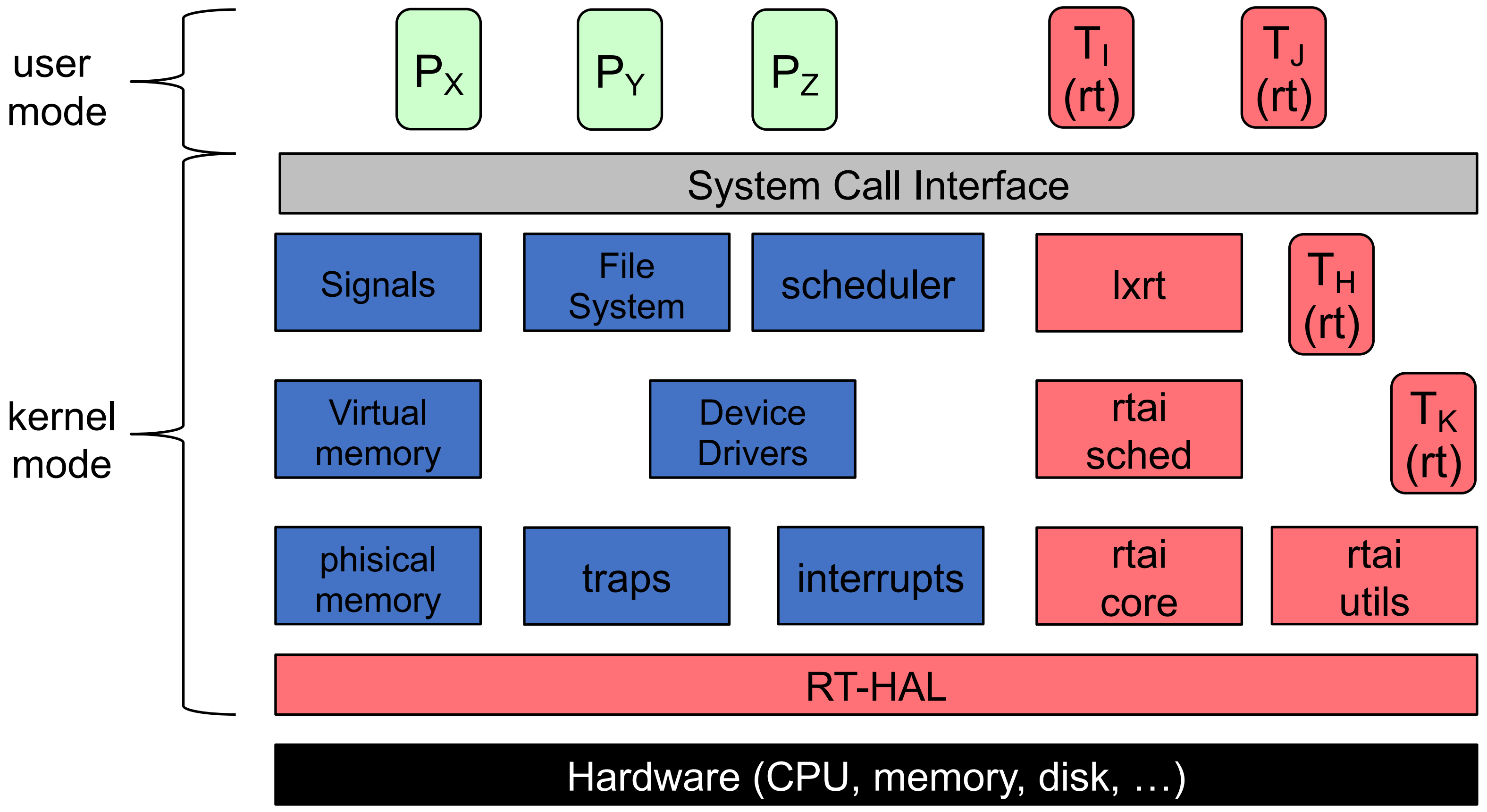
- The Linux kernel is modified (patched) for what concerning interrupt management and scheduling
    - Interrupts are redirected to the Executive
    - Clear Interrupts operations from the Linux Kernel becomes ”soft”: interrupts are not really disabled and pending interrupts for Linux are delayed by the Executive
- Linux becomes *fully preemtpable* by real-time operations
- Approach initially adopted by RT-Linux. Similar approach then used in RTAI and Xenomai.

Dual kernels: RTAI

- RTAI (Real Time Application Interface) is a patch of the Linux kernel that makes Linux *fully preemtpable*, introducing a dual kernel
- It also includes a library, coupled with a patch to the Linux scheduler, to implement real-time tasks in user space and map them to the real-time scheduler
- It has been impelemented at the Dipartimento di Ingegneria Aerospaziale of Politecnico di Milano
- It is an open source project, based on LGPL
- Might be subject to patent issues in USA with RT-Linux -> problem for adoption in the industry!

RTAI modular structure

- Modifies the Hardware Abstraction Layer
- Allows both user level and kernel level real-time tasks



RTAI: I-Pipes

- RTAI exploits I-Pipes (Interrupt Pipelines) to implement the interrupt filtering scheme in the RT-HAL
- I-Pipes is a small nanokernel to manage interrupts in pipelines of applications and kernel modules that manage them
- The trick is that real-time applications, and so real-time relevant interrupts, come first in the pipeline
- Same functionality of RT-Linux, with a different name, and I-Pipes has been published before the RT-Linux patent!

RTAI: Interrupt filtering

- When an interrupt arrives, it is handled by the real-time executive
  - If the interrupt is related to real-time activities, the real-time scheduler and corresponding real-time task is notified
  - If it is related to Linux non real-time activity, it is flagged and deferred… it will be managed by Linux eventually, when no more real-time tasks are active
- This way, Linux always executes in the background, as a low priority background task
- RTAI patches the Linux kernel in order to virtualize interrupt disabling
  - If the Linux kernel invokes a clear interrupt operation (CLI) to disable interrupts, this will become a «soft» operation which is forwarded to the RTAI executive which will not really disable interrupts (to not interfere, e.g., with real-time timers events)

Adeos and Xenomai

- Adeos: a nanokernel that implements the same mechanisms of I-Pipes, as a loadable Linux kernel module
- Xenomai is based on Adeos
- From Xenomai 3, both the dual kernel and a user-space approach are available…
  - … but how to do real-time in user space Linux??

Real-time in user space

- Not feasible with monolithic kernels: big kernel lock! → high latencies
- With classical preemptible kernels it is possible to bound L_{np} to the maximum duration of kernel critical sections, using NPP, but might be too high
- With microkernels and and dual kernels we use HLP instead of NPP, distinguishing non real-time tasks form real-time ones, but real-time tasks may require laborious programming
  - E.g. as low-level modules… in the case of Linux dual-kernels, they cannot use kernel services and existing drivers!
- Can we do better?? Can we use Priority Inheritance in the kernel to fully enable real-time in user space?

Why Linux real-time?

- Large community and developer base
- Large availability of library, drivers, …
- Support for the most used hardware architectures
- Kernel customization allows adoption also in hardware-constrained embedded systems, other than in cloud and edge devices

vanilla Linux è quello non
modificato, standard

Why is the vanilla Linux kernel not adequate for real-time?

- Linux is a multithreaded preemptable kernel, but
  - ISRs and drivers' bottom-halves (BHs) use spinlocks (active waits) for synchronization, that are implicitly equivalent to NPP

- What if:
  - ISRs and BHs are turned into kernel threads? (schedulable entities)
  - And if spinlocks are replaced with mutexes with a real-time synchronization protocol (as PI)?

PREEMPT_RT

- Instead of adding another kernel, the Linux kernel itself is modified to remove unbounded latency sources due to non-preemptive sections
- Main features:
  - ISRs and BHs run as kernel threads
    - They can be scheduled as any other task, according to priorities
    - User space real-time tasks can run at even a higher priority than ISRs!
  - Interrupt disabling is removed
    - Needed to allow the kernel be preempted from everywhere
  - Spinlocks (active waits) are turned into mutexes
  - Priority inversion problems avoided with priority inheritance at kernel level
  - Support to high-resolution timers

PREEMPT_RT vs Dual Kernels

- PROS:
  - The implementation of real-time tasks in user space is similar to non real-time ones, using standardized RT-POSIX APIs
    - Dual kernels introduce their own APIs that limit portability
  - Dual kernels require invasive modifications to the kernel code which cause instability and and introduce dependency to the kernel version
    - PREEMPT_RT is instead maintained and updated by the large Linux community!
- CONS:
  - The implementation or real-time tasks can be laborious
    - POSIX API are usually more complex than dual kernel ones
    - Other issues must be considered, such as, use of non-real time drivers, memory locking, …
  - The real-time perforamce is slightly worse than dual kernel
    - Task switch times in the order of tens of microseconds w.r.t. to one microsecond for RTAI and Xenomai