Rights to copy

- © Copyright 2004-2019, Bootlin
- **License: Creative Commons Attribution - Share Alike 3.0**
- https://creativecommons.org/licenses/by-sa/3.0/legalcode
- You are free:
  - to copy, distribute, display, and perform the work
  - to make derivative works
  - to make commercial use of the work
- Under the following conditions:
  - Attribution. You must give the original author credit.
  - Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only
- under a license identical to this one.
  - For any reuse or distribution, you must make clear to others the license terms of this work.
  - Any of these conditions can be waived if you get permission from the copyright holder.
- Your fair use and other rights are in no way affected by the above.
- **Document sources:** https://git.bootlin.com/training-materials/

11. Real-Time in Embedded Linux Systems

Embedded Linux and real time

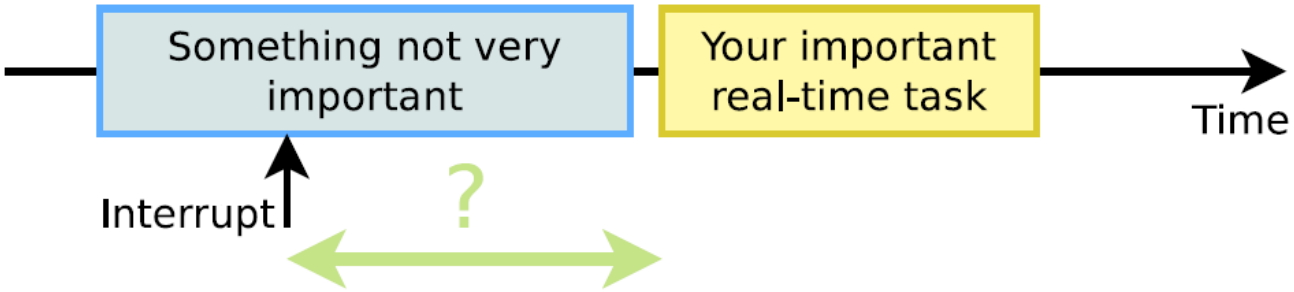
- Due to its advantages, Linux and open-source software are more and more commonly used in embedded applications
- However, some applications also have real-time constraints
- They, at the same time, want to
  - Get all the nice advantages of Linux: hardware support, components re-use, low cost, etc.
  - Get their real-time constraints met
- Linux was originally designed as a time-sharing system
  - The main goal was to get the best throughput from the available hardware, by making the best possible usage of resources (CPU, memory, I/O)
  - Time determinism was not taken into account
- On the opposite, real-time constraints imply time determinism, even at the expense of lower global throughput
- Best throughput and time determinism are contradictory requirements

Linux and real-time approaches

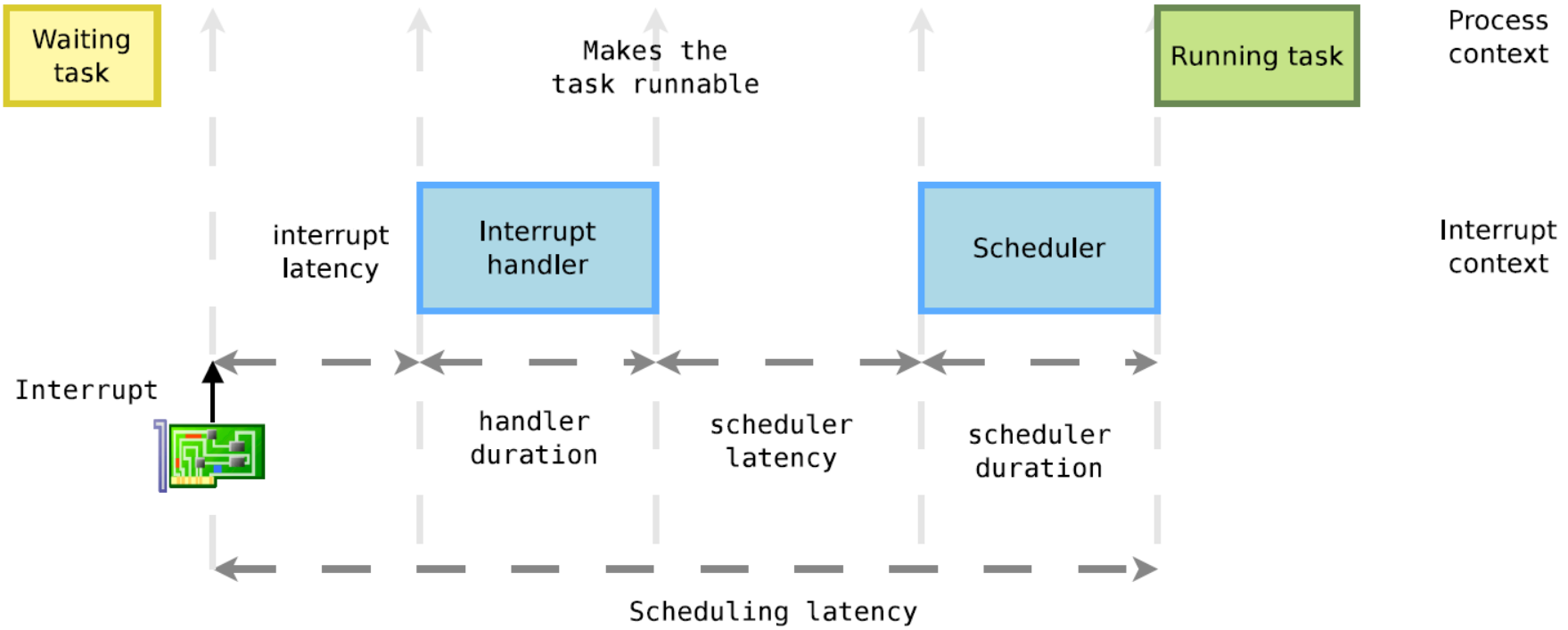
- Over time, two major approaches have been taken to bring real-time requirements into Linux
- Approach 1
  - Improve the Linux kernel itself so that it matches real-time requirements, by providing bounded latencies, real-time APIs, etc.
  - Allows preemption, so minimize latencies
  - Execute all activities (including IRQ) in "schedulable/thread" context
  - Approach taken by the mainline Linux kernel and the PREEMPT_RT project.
- Approach 2
  - Add a layer below the Linux kernel that will handle all the real-time requirements, so that the behavior of Linux doesn't affect real-time tasks.
  - RTLinux, RTAI and Xenomai

Understanding latency

- When developing real-time applications with a Linux, the typical scenario is the following
  - An event from the physical world happens and gets notified to the CPU by means of an interrupt
  - The interrupt handler recognizes and handles the event, and then wake-up the user-space task that will react to this event
  - Some time later, the user-space task will run and be able to react to the physical world event
- Real-time is about providing guaranteed worst case latencies for this reaction time, called latency



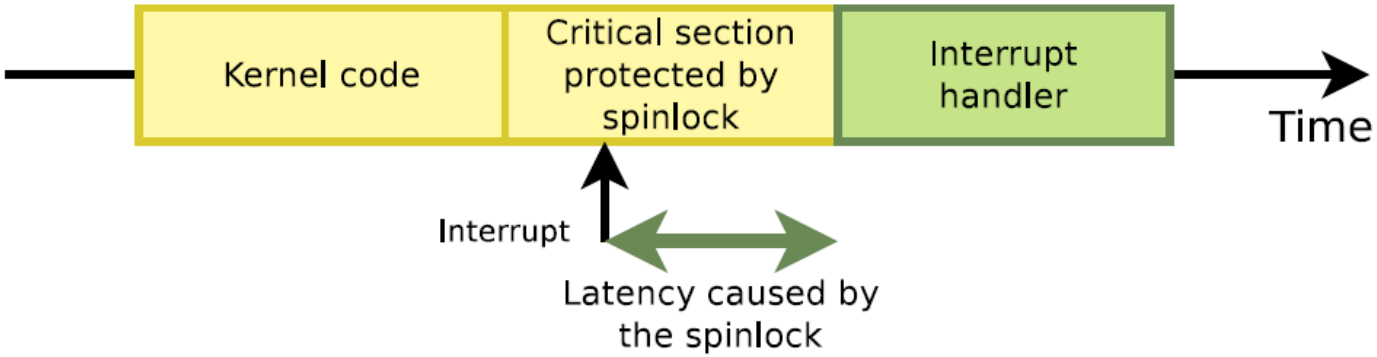
Linux kernel latency components



kernel latency = interrupt latency + handler duration
+ scheduler latency + scheduler duration

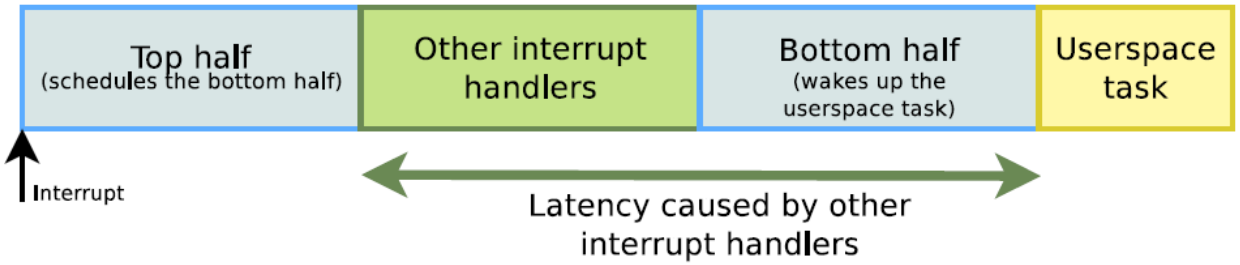
Sources of interrupt latency

- One of the concurrency prevention mechanism used in the kernel is the spinlock
  - prevent concurrent accesses between a process context and an interrupt context works by disabling interrupts
  - Critical sections protected by spinlocks, or other section in which interrupts are explicitly disabled will delay the beginning of the execution of the interrupt handler
    - The duration of these critical sections is unbounded
- Other possible source: shared interrupts



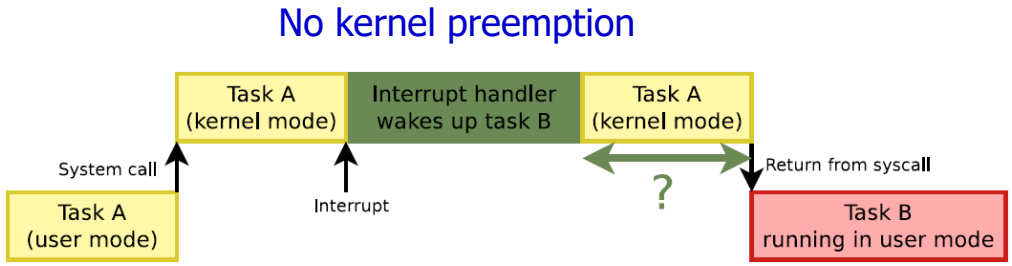
Interrupt handler duration

- In Linux, many interrupt handlers are split in two parts
  - A top-half, started by the CPU as soon as interrupts are enabled.
    - runs with the interrupt line disabled and is supposed to complete as quickly as possible.
  - A bottom-half, scheduled by the top-half, which starts after all pending top-halves have completed their execution.
- Therefore, for real-time critical interrupts, bottom-halves shouldn't be used
  - their execution is delayed by all other interrupts in the system.



Scheduler latency

- The Linux kernel is a preemptive operating system
- When a task runs in user-space mode and gets interrupted by an interruption
  - if the interrupt handler wakes up another task, this task can be scheduled as soon as we return from the interrupt handler.
- However, when the interrupt comes while the task is executing a system call
  - this system call has to finish before another task can be scheduled.
  - By default, the Linux kernel does not do kernel preemption.
  - the time before which the scheduler will be called to schedule another task is unbounded.



Scheduler Duration

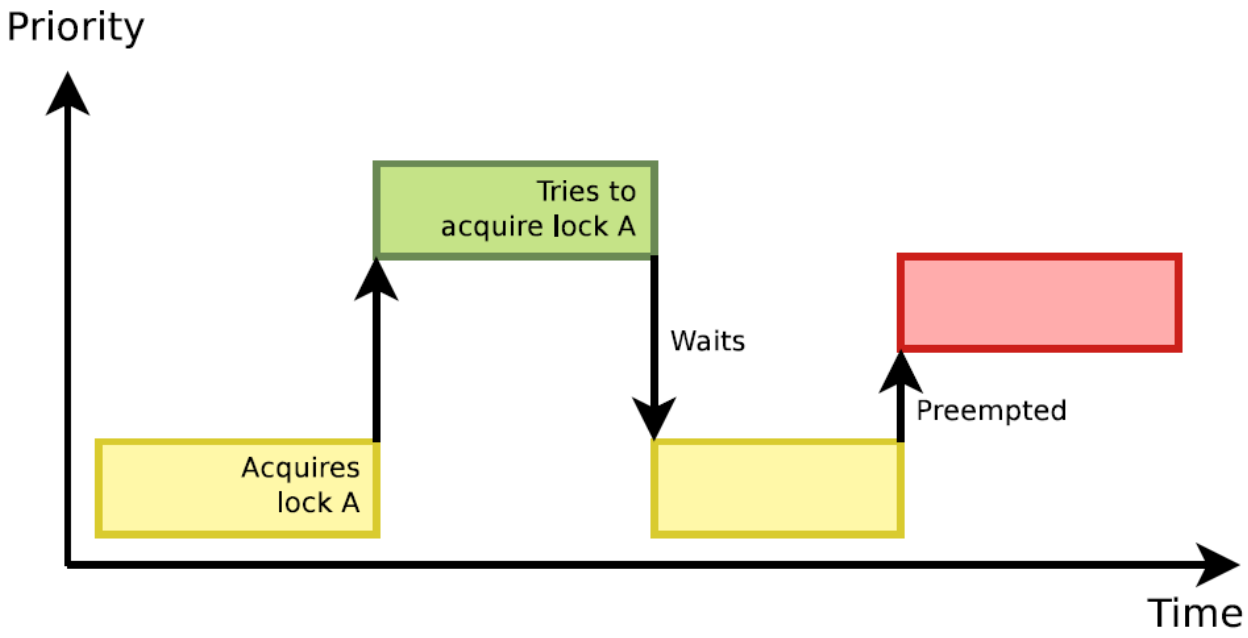
- Time to execute the scheduler and switch to a new task
  - Time to execute the scheduler: depended on the number of running processes.
  - Context switching time: time to save the state of the current process (CPU registers) and restore the new process to run.
    - This time is constant, so if not an issue for real-time.
  - SCHED_FIFO & SCHED_RR use bit map to find out the next task
  - CFS (Completely Fair Scheduler) uses Red-black tree as a sorted queue

Other non-deterministic mechanisms

- Linux is highly based on virtual memory, as provided by an MMU, so that memory is allocated on demand.
  - Whenever an application accesses code or data for the first time, it is loaded on demand, which can creates huge delays.
- Many C library services or kernel services are not designed with real-time constraints in mind.

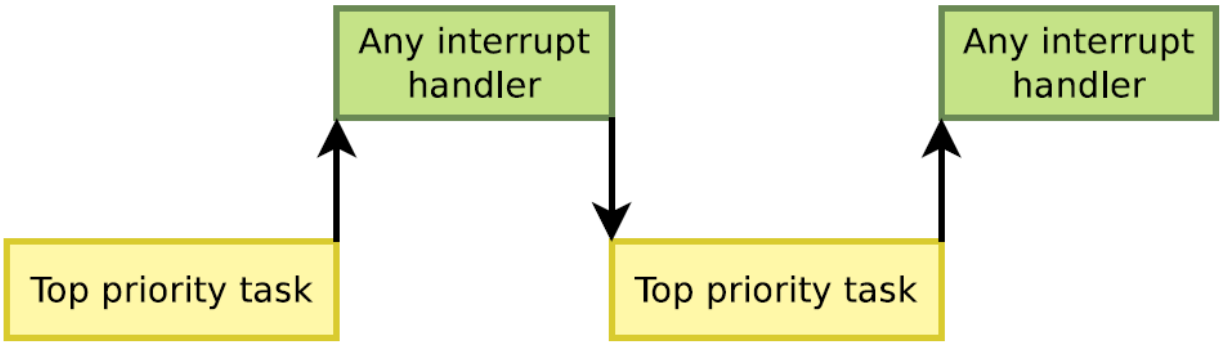
Priority inversion

- A process with a low priority might hold a lock needed by a higher priority process, effectively reducing the priority of this process.
- Things can be even worse if a middle priority process uses the CPU.
- RTOSes provide priority inheritance protocols



Interrupt handler priority

- In Linux, interrupt handlers are executed directly by the CPU interrupt mechanisms, and not under control of the Linux scheduler.
- Therefore, all interrupt handlers have a higher priority than all tasks running on the system.



The PREEMPT_RT project

- Long-term project lead by Linux kernel developers
  Ingo Molnar, Thomas Gleixner and Steven Rostedt
  - https://rt.wiki.kernel.org

Paul McKenney and Ingo Molnar
at the 2013 Kernel Summit

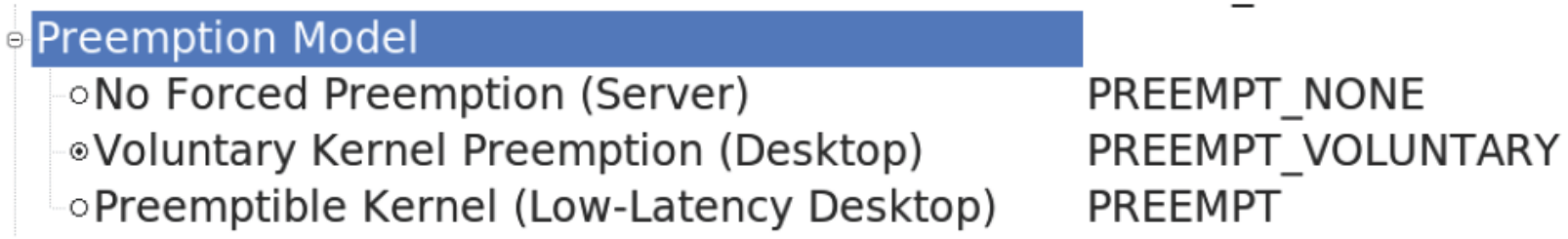
- The goal is to gradually improve the Linux kernel regarding real-time requirements and to get these improvements merged into the mainline kernel
- Many of the improvements designed, developed and debugged inside PREEMPT_RT over the years are now part of the mainline Linux kernel
  - The project is a long-term branch of the Linux kernel that ultimately should disappear as everything will have been merged

Improvements in the mainline kernel

- Coming from the PREEMPT_RT project
- Since the beginning of 2.6
  - O(1) scheduler ➔ Now, CFS scheduler
  - Kernel preemption
  - Better POSIX real-time API support
- Since 2.6.18
  - Priority inheritance support for mutexes
- Since 2.6.21
  - High-resolution timers (HRT)
- Since 2.6.30
  - Threaded interrupts
- Since 2.6.33
  - Spinlock annotations

New preemption options in Linux 2.6

2 new preemption models offered by standard Linux 2.6:



1st option: no forced preemption

- CONFIG_PREEMPT_NONE
  - Kernel code (interrupts, exceptions, system calls) never preempted.
  - Default behavior in standard kernels.
- Best for systems making intense computations, on which overall throughput is key.
- Best to reduce task switching to maximize CPU and cache usage (by reducing context switching).
- Still benefits from some Linux 2.6 improvements: O(1) scheduler, increased multiprocessor safety (work on RT preemption was useful to identify hard to find SMP bugs).
- Can also benefit from a lower timer frequency (100 Hz instead of 250 or 1000).

2nd option: voluntary kernel preemption

- CONFIG_PREEMPT_VOLUNTARY
  - Kernel code can preempt itself
- Typically for desktop systems, for quicker application reaction to user input.
- Adds explicit rescheduling points throughout kernel code.
- Minor impact on throughput.
- Used in: Ubuntu Desktop 13.04, Ubuntu Server 12.04

3rd option: preemptible kernel

- CONFIG_PREEMPT
  - Most kernel code can be involuntarily preempted at any time.
  - When a process becomes runnable, no more need to wait for kernel code (typically a system call) to return before running the scheduler.

- Exception: kernel critical sections (holding spinlocks)
  - In a case you hold a spinlock on a uni-processor system, kernel preemption could run another process, which would loop forever if it tried to acquire the same spinlock.
- Typically for desktop or embedded systems with latency requirements in the milliseconds range.
- Still a relatively minor impact on throughput.

Priority inheritance

- One classical solution to the priority inversion problem
- When a task of a low priority holds a lock requested by a higher priority task, the priority of the first task gets temporarily raised to the priority of the second task: it has inherited its priority.
- In Linux, since 2.6.18, mutexes support priority inheritance
- In userspace, priority inheritance must be explicitly enabled on a per-mutex basis.

```
#include <linux/rtmutex.h>

    void rt_mutex_init(struct rt_mutex *lock);
    void rt_mutex_destroy(struct rt_mutex *lock);

    void rt_mutex_lock(struct rt_mutex *lock);
    int rt_mutex_lock_interruptible(struct rt_mutex *lock,
                                    int detect_deadlock);
    int rt_mutex_timed_lock(struct rt_mutex *lock,
                            struct hrtimer_sleeper *timeout,
                                                int detect_deadlock);
    int rt_mutex_trylock(struct rt_mutex *lock);
    void rt_mutex_unlock(struct rt_mutex *lock);
    int rt_mutex_is_locked(struct rt_mutex *lock);
```

High resolution timers

- The resolution of the timers used to be bound to the resolution of the regular system tick
  - Usually 100 Hz or 250 Hz, depending on the architecture and the configuration
  - A resolution of only 10 ms or 4 ms.
  - Increasing the regular system tick frequency is not an option as it would consume too many resources
- The high-resolution timers infrastructure, merged in 2.6.21
  - allows to use the available hardware timers to program interrupts at the right moment.
  - Hardware timers are multiplexed, so that a single hardware timer is sufficient to handle a large number of software-programmed timers.
  - Usable directly from user-space using the usual timer APIs
    - `void hrtimer_init(struct hrtimer *timer, clockid_t which_clock);`
    - `int hrtimer_start(struct hrtimer *timer, ktime_t time, enum hrtimer_mode mode);`

Threaded interrupts

- To solve the interrupt inversion problem, PREEMPT_RT has introduced the concept of threaded interrupts
- The interrupt handlers run in normal kernel threads, so that the priorities of the different interrupt handlers can be configured
- The real interrupt handler, as executed by the CPU, is only in charge of masking the interrupt and waking-up the corresponding thread
- The idea of threaded interrupts also allows to use sleeping spinlocks
- Merged since 2.6.30, the conversion of interrupt handlers to threaded interrupts is not automatic: drivers must be modified
- In PREEMPT_RT, all interrupt handlers are switched to threaded interrupts

CONFIG_PREEMPT_RT

- The PREEMPT_RT patch adds a new level of preemption, called CONFIG_PREEMPT_RT
- This level of preemption replaces all kernel spinlocks by mutexes (or so-called sleeping spinlocks)
  - Instead of providing mutual exclusion by disabling interrupts and preemption, they are just normal locks: when contention happens, the process is blocked and another one is selected by the scheduler.
  - Works well with threaded interrupts, since threads can block, while usual interrupt handlers could not.
  - Some core, carefully controlled, kernel spinlocks remain as normal spinlocks.

CONFIG_PREEMPT_RT

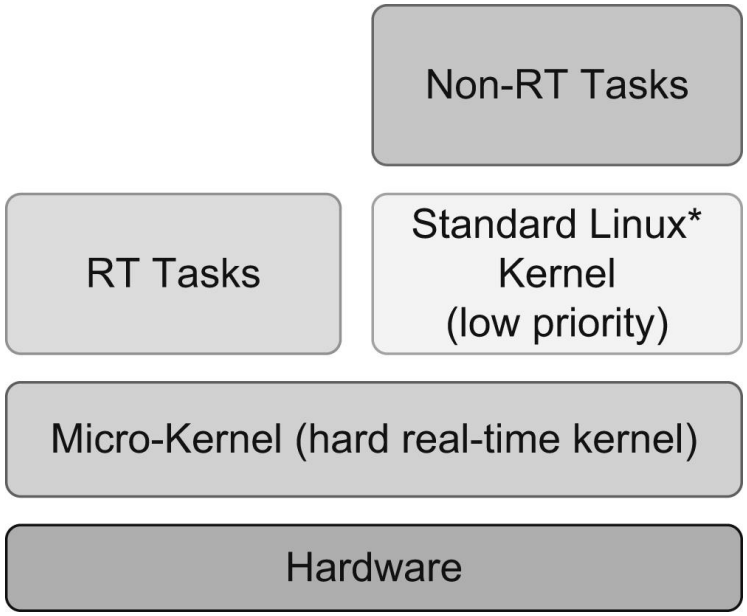
- With CONFIG_PREEMPT_RT, virtually all kernel code becomes preemptible
  - An interrupt can occur at any time, when returning from the interrupt handler, the woken up process can start immediately.
- This is the last big part of PREEMPT_RT that isn't fully in the mainline kernel yet
  - Part of it has been merged in 2.6.33: the spinlock annotations.
  - The spinlocks that must remain as spinning spinlocks are now differentiated from spinlocks that can be converted to sleeping spinlocks.
  - This has reduced a lot the PREEMPT_RT patch size!

Threaded interrupts

- The mechanism of threaded interrupts in PREEMPT_RT is still different from the one merged in mainline
- In PREEMPT_RT, all interrupt handlers are unconditionally converted to threaded interrupts.
- This is a temporary solution, until interesting drivers in mainline get gradually converted to the new threaded interrupt API that has been merged in 2.6.30.

Linux real-time extensions

- Three generations
  - RTLinux
  - RTAI
  - Xenomai
- A common principle
  - Add an extra layer between the hardware and the Linux kernel, to manage real-time tasks separately.



RTLinux

- First real-time extension for Linux, created by Victor Yodaiken.
  - Nice, but the author filed a software patent covering the addition of real-time support to general operating systems as implemented in RTLinux!
  - Its Open Patent License drew many developers away and frightened users. Community projects like RTAI and Xenomai now attract most developers and users.
  - February, 2007: RTLinux rights sold to Wind River. Now supported by Wind River as "Real-Time Core for Wind River Linux."
  - Free version still advertised by Wind River on http://www.rtlinuxfree.com, but no longer a community project.

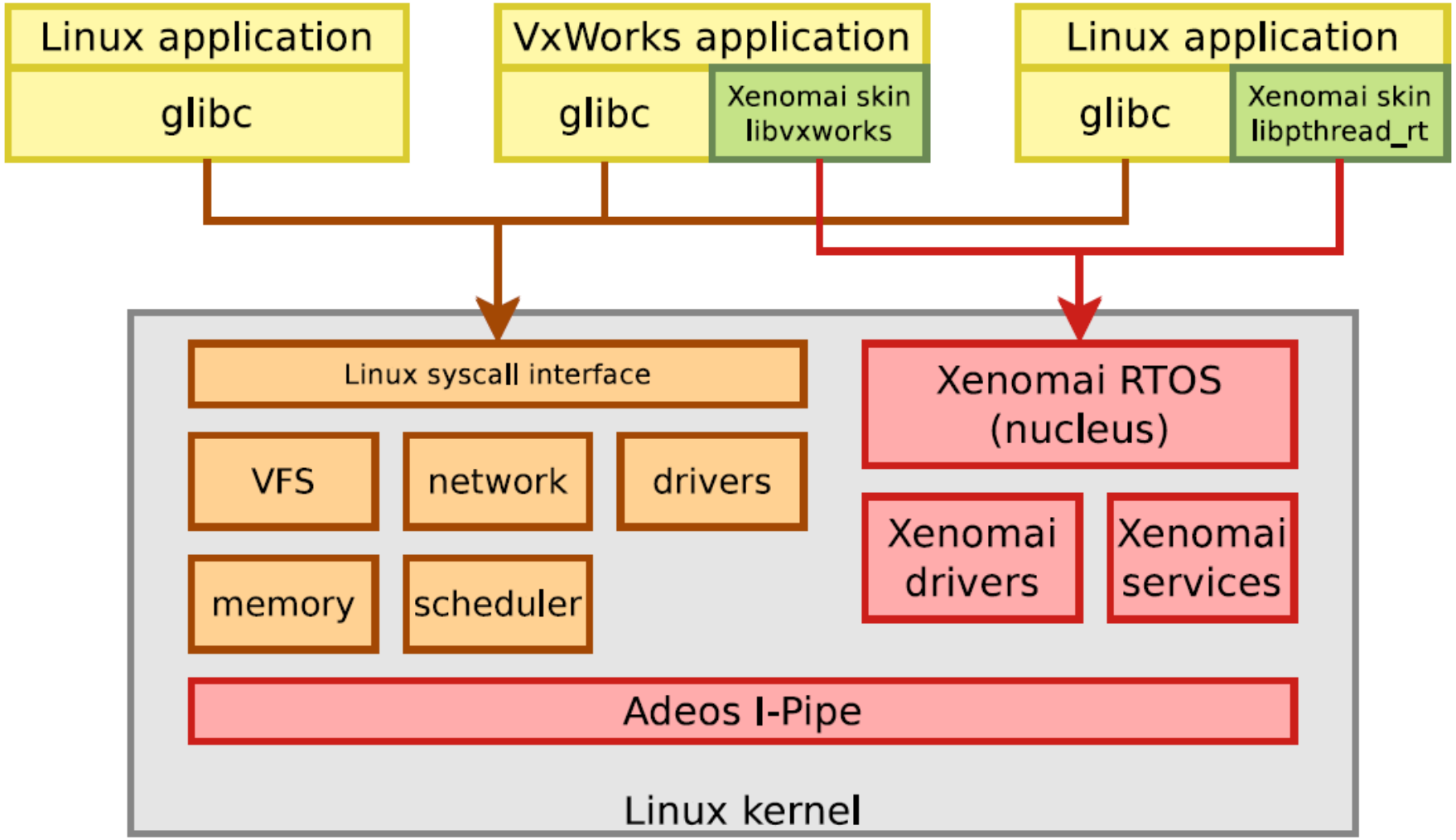
RTAI

- http://www.rtai.org/ - Real-Time Application Interface for Linux
  - Created in 1999, by Prof. Paolo Mantegazza (long time contributor to RTLinux)
  - Community project. Significant user base. Attracted contributors frustrated by the RTLinux legal issues.
  - supports several architectures:
    - x86 (with and without FPU and TSC)
    - x86_64
    - PowerPC
    - ARM (StrongARM; ARM7: clps711x-family, Cirrus Logic EP7xxx, CS89712, PXA25x)
    - m68k (supporting both MMU and NOMMU cpus)
  - May offer slightly better latencies than Xenomai, at the expense of a less maintainable and less portable code base

Xenomai project

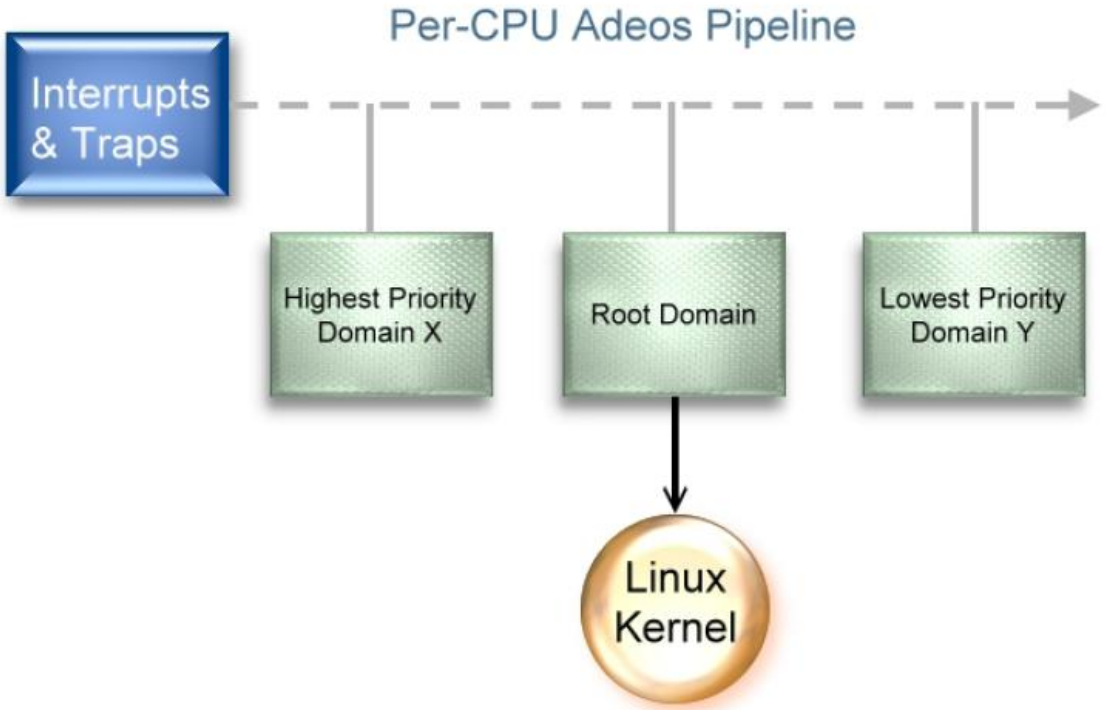
- http://www.xenomai.org/
  - Started in 2001 as a project aiming at emulating traditional RTOS.
  - Initial goals: facilitate the porting of programs to GNU / Linux.
  - Initially related to the RTAI project (as the RTAI / fusion branch), now independent.
  - Skins mimicking the APIs of traditional RTOS such as VxWorks, pSOS+, and VRTXsa as well as the POSIX API, and a "native" API.
  - Aims at working both as a co-kernel and on top of PREEMPT_RT in future upstream Linux versions.
  - Will never be merged in the mainline kernel.

Xenomai architecture



The Adeos interrupt pipeline abstraction

- From the Adeos point of view, guest OSes are prioritized domains.
- For each event (interrupts, exceptions, syscalls, etc...), the various domains may handle the event or pass it down the pipeline.



Xenomai features

- Factored real-time core with skins implementing various real-time APIs
- Seamless support for hard real-time in user-space
- No second-class citizen, all ports are equivalent feature-wise
- Xenomai support is as much as possible independent from the Linux kernel version (backward and forward compatible when reasonable)
- Each Xenomai branch has a stable user/kernel ABI
- Timer system based on hardware high-resolution timers
- Per-skin time base which may be periodic
- RTDM skin allowing to write real-time drivers

Xenomai user-space real-time support

- Xenomai supports real-time in user-space on 5 architectures, including 32 and 64 bits variants.
- Two modes are defined for a thread
  - the primary mode, where the thread is handled by Xenomai scheduler
  - the secondary mode, when it is handled by Linux scheduler.
- Thanks to the services of the Adeos I-pipe service, Xenomai system calls are defined.
  - A thread migrates from secondary mode to primary mode when such a system call is issued
  - It migrates from primary mode to secondary mode when a Linux system call is issued, or to handle gracefully exceptional events such as exceptions or Linux signals.

Life of a Xenomai application

- Xenomai applications are started like normal Linux processes, they are initially handled by the Linux scheduler and have access to all Linux services
- After their initialization, they declare themselves as real-time applications, which migrates them to primary mode. In this mode:
  - They are scheduled directly by the Xenomai scheduler, so they have the real-time properties offered by Xenomai
  - They don't have access to any Linux service, otherwise they get migrated back to secondary mode and loose all real-time properties
  - They can only use device drivers that are implemented in Xenomai, not the ones of the Linux kernel
- Need to implement device drivers in Xenomai, and to split real-time and non real-time parts of your applications.

Real Time Driver Model (RTDM)

- An approach to unify the interfaces for developing device drivers and associated applications under real-time Linux
  - An API very similar to the native Linux kernel driver API
- Allows to develop in kernel space:
  - Character-style device drivers
  - Network-style device drivers
- See the whitepaper on

  http://www.xenomai.org/documentation/xenomai-2.6/pdf/RTDM-and-Applications.pdf
- Current notable RTDM based drivers:
  - Serial port controllers;
  - RTnet UDP/IP stack;
  - RT socket CAN, drivers for CAN controllers;
  - Analogy, fork of the Comedy project, drivers for acquisition cards.