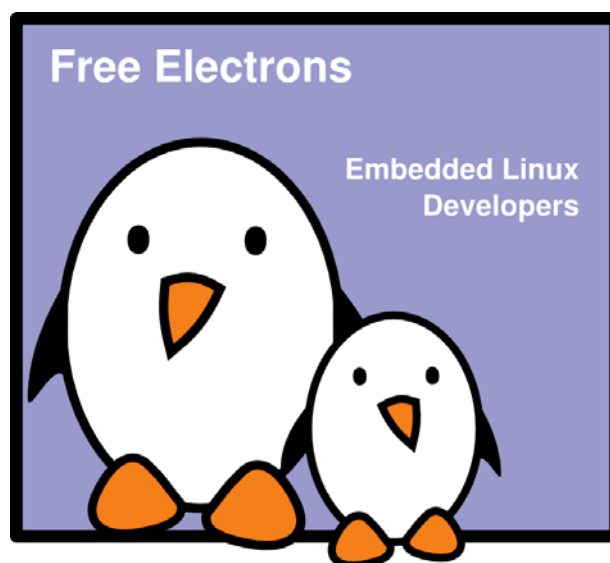




## Real-time in embedded Linux systems

Michael Opdenacker  
Thomas Petazzoni  
Gilles Chantepedrix  
**Free Electrons**



© Copyright 2004-2011, Free Electrons.  
Creative Commons BY-SA 3.0 license  
Latest update: Nov 2, 2011,  
Document sources, updates and translations:  
<http://free-electrons.com/docs/realtime>  
Corrections, suggestions, contributions and translations are welcome!

1



## Real Time in Embedded Linux Systems

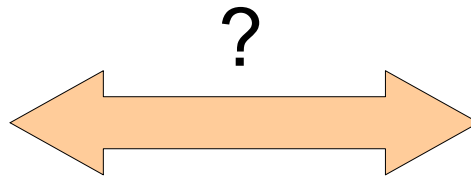
### Introduction

2



# Embedded Linux and real time

- ▶ Due to its advantages, Linux and the open-source softwares 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



3



# Embedded Linux and real time

- ▶ Linux is an operating system part of the large Unix family
- ▶ It was originally designed as a time-sharing system
  - ▶ The main goal is to get the best throughput from the available hardware, by making the best possible usage of resources (CPU, memory, I/O)
  - ▶ Time determinism is 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

4



- ▶ 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.
  - ▶ 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 behaviour of Linux doesn't affect real-time tasks.
  - ▶ Approach taken by RTLinux, RTAI and Xenomai

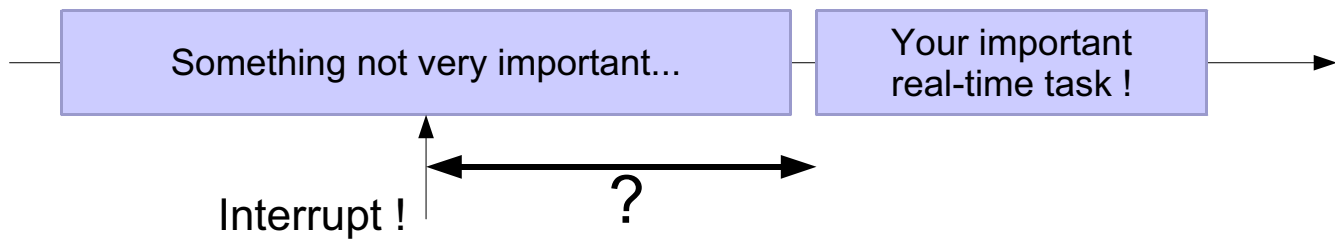


## **Approach 1** Improving the main Linux kernel with PREEMPT\_RT



# Understanding latency

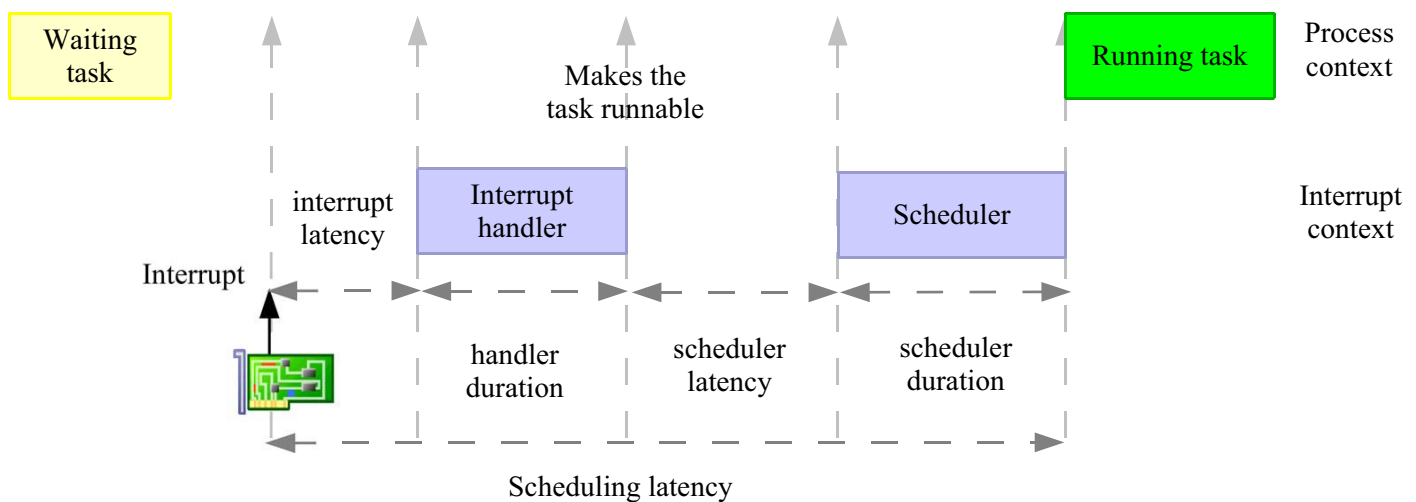
- ▶ When developing real-time applications with a system such as Linux, the typical scenario is the following
  - ▶ An event from the physical world happens and gets notified to the CPU by means of an interrupt
  - ▶ The interrupt handler recognizes and handles the event, and then wake-up the user-space 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*



7



# Linux kernel latency components



$$\text{kernel latency} = \text{interrupt latency} + \text{handler duration} + \text{scheduler latency} + \text{scheduler duration}$$

8



## Other non-deterministic mechanisms

- ▶ Outside of the critical path detailed previously, other non-deterministic mechanisms of Linux can affect the execution time of real-time tasks
- ▶ 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 create huge delays.
- ▶ Many C library services or kernel services are not designed with real-time constraints in mind.

9

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



## The PREEMPT\_RT project

- ▶ Long-term project lead by Linux kernel developers Ingo Molnar, Thomas Gleixner and Steven Rostedt
  - ▶ <https://rt.wiki.kernel.org>
- ▶ The goal is to gradually improve the Linux kernel regarding real-time requirements and to get these improvements merged into the mainline kernel
  - ▶ PREEMPT\_RT development works very closely with the mainline development
- ▶ 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

10

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



# Improvements in the mainline kernel

- ▶ Coming from the PREEMPT\_RT project
- ▶ Since the beginning of 2.6
  - ▶ O(1) 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
- ▶ Since 2.6.30
  - ▶ Threaded interrupts
- ▶ Since 2.6.33
  - ▶ Spinlock annotations

11



# New preemption options in Linux 2.6

2 new preemption models offered by standard Linux 2.6:

Preemption Model	
○ No Forced Preemption (Server)	PREEMPT_NONE
⦿ Voluntary Kernel Preemption (Desktop)	PREEMPT_VOLUNTARY
○ Preemptible Kernel (Low-Latency Desktop)	PREEMPT

12



## 1<sup>st</sup> 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).

13

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



## 2<sup>nd</sup> 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.

14

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



### 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), but a rescheduling point occurs when exiting the outer critical section, in case a preemption opportunity would have been signaled while in the critical section.
- ▶ Typically for desktop or embedded systems with latency requirements in the milliseconds range.
- ▶ Still a relatively minor impact on throughput.



## 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 much 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



# 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 (see later)
- ▶ 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

17

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



## PREEMPT\_RT specifics

18

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



## CONFIG\_PREEMPT\_RT (1)

- ▶ 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

19

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



## CONFIG\_PREEMPT\_RT (2)

- ▶ 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 !

20

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



# 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.

21

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



## Setting up PREEMPT\_RT

22

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



## PREEMPT\_RT setup (1)

- ▶ PREEMPT\_RT is delivered as a patch against the mainline kernel
  - ▶ Best to have a board supported by the mainline kernel, otherwise the PREEMPT\_RT patch may not apply and may require some adaptations
- ▶ Many official kernel releases are supported, but not all. For example, 2.6.31 and 2.6.33 are supported, but not 2.6.32.
- ▶ Quick set up
  - ▶ Download and extract mainline kernel
  - ▶ Download the corresponding PREEMPT\_RT patch
  - ▶ Apply it to the mainline kernel tree

23

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



## PREEMPT\_RT setup (2)

- ▶ In the kernel configuration, be sure to enable
  - ▶ CONFIG\_PREEMPT\_RT
  - ▶ High-resolution timers
- ▶ Compile your kernel, and boot
- ▶ You are now running the real-time Linux kernel
- ▶ Of course, some system configuration remains to be done, in particular setting appropriate priorities to the interrupt threads, which depend on your application.

24

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



## Real-time application development

25

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



## Development and compilation

- ▶ No special library is needed, the POSIX realtime API is part of the standard C library
- ▶ The glibc or eglibc C libraries are recommended, as the support of some real-time features is not available yet in uClibc
  - ▶ Priority inheritance mutexes or NPTL on some architectures, for example
- ▶ Compile a program
  - ▶ ARCH-linux-gcc -o myprog myprog.c -lrt
- ▶ To get the documentation of the POSIX API
  - ▶ Install the manpages-posix-dev package
  - ▶ Run `man functionname`

26

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



# Process, thread ?

- ▶ Confusion about the terms «process», «thread» and «task»
- ▶ In Unix, a process is created using `fork()` and is composed of
  - ▶ An address space, which contains the program code, data, stack, shared libraries, etc.
  - ▶ One thread, that starts executing the `main()` function.
  - ▶ Upon creation, a process contains one thread
- ▶ Additional threads can be created inside an existing process, using `pthread_create()`
  - ▶ They run in the same address space as the initial thread of the process
  - ▶ They start executing a function passed as argument to `pthread_create()`

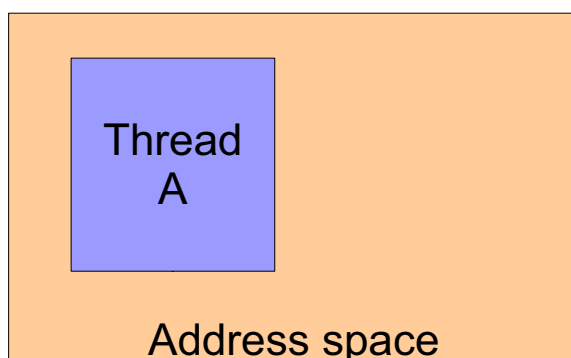
27

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>

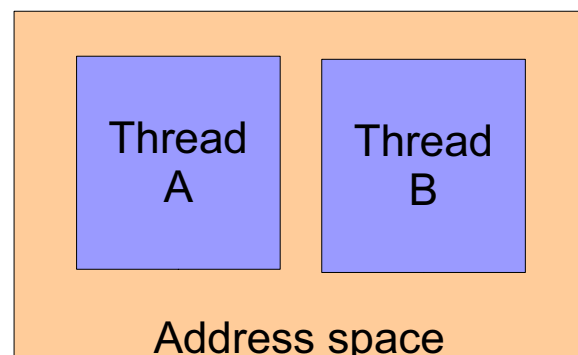


## Process, thread: kernel point of view

- ▶ The kernel represents each thread running in the system by a structure of type `task_struct`
- ▶ From a scheduling point of view, it makes no difference between the initial thread of a process and all additional threads created dynamically using `pthread_create()`



Process after `fork()`



Same process after `pthread_create()`

28

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



# Creating threads

- ▶ Linux support the POSIX thread API
- ▶ To create a new thread
  - ▶ **pthread\_create**(pthread\_t \*thread,  
pthread\_attr\_t \*attr,  
void \*(\*routine)(\*void\*),  
void \*arg);
  - ▶ The new thread will run in the same address space, but will be scheduled independently
- ▶ Exiting from a thread
  - ▶ **pthread\_exit**(void \*value\_ptr);
- ▶ Waiting for a thread termination
  - ▶ **pthread\_join**(pthread\_t \*thread, void \*\*value\_ptr);

29

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



# Scheduling classes (1)

- ▶ The Linux kernel scheduler support different scheduling classes
- ▶ The default class, in which processes are started by default is a time-sharing class
  - ▶ All processes, regardless of their priority, get some CPU time
  - ▶ The proportion of CPU time they get is dynamic and affected by the nice value, which ranges from -20 (highest) to 19 (lowest). Can be set using the nice or renice commands
- ▶ The real-time classes **SCHED\_FIFO** and **SCHED\_RR**
  - ▶ The highest priority process gets all the CPU time, until it blocks.
  - ▶ In SCHED\_RR, round-robin scheduling between the processes of the same priority. All must block before lower priority processes get CPU time.
  - ▶ Priorities ranging from 0 (lowest) to 99 (highest)

30

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



## Scheduling classes (2)

- ▶ An existing program can be started in a specific scheduling class with a specific priority using the `chrt` command line tool
  - ▶ Example: `chrt -f 99 ./myprog`
- ▶ The `sched_setscheduler()` API can be used to change the scheduling class and priority of a process
  - ▶ `int sched_setscheduler(pid_t pid, int policy, const struct sched_param *param);`
  - ▶ `policy` can be `SCHED_OTHER`, `SCHED_FIFO`, `SCHED_RR`, etc.
  - ▶ `param` is a structure containing the priority

31

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



## Scheduling classes (3)

- ▶ The priority can be set on a per-thread basis when a thread is created :

```
struct sched_param parm;
pthread_attr_t attr;

pthread_attr_init(&attr);
pthread_attr_setinheritsched(&attr,
                             PTHREAD_EXPLICIT_SCHED);
pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
parm.sched_priority = 42;
▶ pthread_attr_setschedparam(&attr, &parm);
```

the `attr` structure.

- ▶ Several other attributes can be defined this way: stack size, etc.

32

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



# Memory locking

- ▶ In order to solve the non-determinism introduced by virtual memory, memory can be locked
  - ▶ Guarantee that the system will keep it allocated
  - ▶ Guarantee that the system has pre-loaded everything into memory
- ▶ `mlockall(MCL_CURRENT | MCL_FUTURE);`
  - ▶ Locks all the memory of the current address space, for currently mapped pages and pages mapped in the future
- ▶ Other, less useful parts of the API: `munlockall`, `mock`, `munlock`.
- ▶ Watch out for non-currently mapped pages
  - ▶ Stack pages
  - ▶ Dynamically-allocated memory

33

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



# Mutexes

- ▶ Allows mutual exclusion between two threads in the same address space
  - ▶ Initialization/destruction
    - `pthread_mutex_init`**(`pthread_mutex_t *mutex, const pthread_mutexattr_t *mutexattr`);
    - `pthread_mutex_destroy`**(`pthread_mutex_t *mutex`);
  - ▶ Lock/unlock
    - `pthread_mutex_lock`**(`pthread_mutex_t *mutex`);
    - `pthread_mutex_unlock`**(`pthread_mutex_t *mutex`);
- ▶ Priority inheritance must explicitly be activated
  - `pthread_mutexattr_t attr`;
  - `pthread_mutexattr_init`** (&attr);
  - `pthread_mutexattr_getprotocol`**  
(&attr, PTHREAD\_PRIO\_INHERIT);

34

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



# Timers

- ▶ **timer\_create**(clockid\_t clockid,  
                  struct sigevent \*evp,  
                  timer\_t \*timerid)
- ▶ Create a timer. **clockid** is usually CLOCK\_MONOTONIC. **sigevent** defines what happens upon timer expiration : send a signal or start a function in a new thread. **timerid** is the returned timer identifier.
- ▶ **timer\_settime**(timer\_t timerid, int flags,  
                  struct itimerspec \*newvalue,  
                  struct itimerspec \*oldvalue)
- ▶ Configures the timer for expiration at a given time.
- ▶ **timer\_delete**(timer\_t timerid), delete a timer
- ▶ **clock\_getres**(), get the resolution of a clock
- ▶ Other functions: **timer\_getoverrun**(), **timer\_gettime**()

35



# Signals

- ▶ Signals are an asynchronous notification mechanism
- ▶ Notification occurs either
  - ▶ By the call of a signal handler. Be careful with the limitations of signal handlers!
  - ▶ By being unblocked from the **sigwait**(), **sigtimedwait**() or **sigwaitinfo**() functions. Usually better.
- ▶ Signal behaviour can be configured using **sigaction**()
- ▶ Mask of blocked signals can be changed with **pthread\_sigmask**()
- ▶ Delivery of a signal using **pthread\_kill**() or **tgkill**()
- ▶ All signals between **SIGRTMIN** and **SIGRTMAX**, 32 signals under Linux.

36



## ▶ Semaphores

- ▶ Usable between different processes using named semaphores
- ▶ **`sem_open()`**, **`sem_close()`**, **`sem_unlink()`**, **`sem_init()`**, **`sem_destroy()`**, **`sem_wait()`**, **`sem_post()`**, etc.

## ▶ Message queues

- ▶ Allows processes to exchange data in the form of messages.
- ▶ **`mq_open()`**, **`mq_close()`**, **`mq_unlink()`**, **`mq_send()`**, **`mq_receive()`**, etc.

## ▶ Shared memory

- ▶ Allows processes to communicate by sharing a segment of memory
- ▶ **`shm_open()`**, **`ftruncate()`**, **`mmap()`**, **`munmap()`**, **`close()`**, **`shm_unlink()`**

37



## Approach 2

### Real-time extensions to the Linux kernel

38



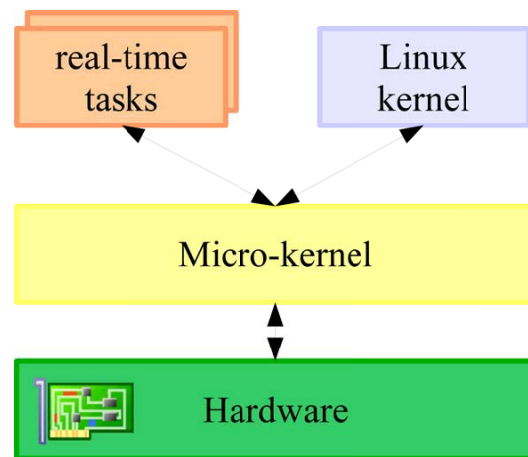
# Linux real-time extensions

## 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.



39

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



# 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.

40

Free Electrons. Kernel, drivers and embedded Linux development, consulting, training and support. <http://free-electrons.com>



<http://www.rtai.org/> - Real-Time Application Interface for Linux

- ▶ Created in 1999, by Prof. Paolo Montegazza (long time contributor to RTLinux), Dipartimento di Ingegneria Aerospaziale Politecnico di Milano (DIAPM).
- ▶ Community project. Significant user base. Attracted contributors frustrated by the RTLinux legal issues.
- ▶ Only really actively maintained on x86
- ▶ May offer slightly better latencies than Xenomai, at the expense of a less maintainable and less portable code base
- ▶ Since RTAI is not really maintained on ARM and other embedded architectures, our presentation is focused on Xenomai.



## 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 the upcoming 3.0 branch.
- ▶ Will never be merged in the mainline kernel.