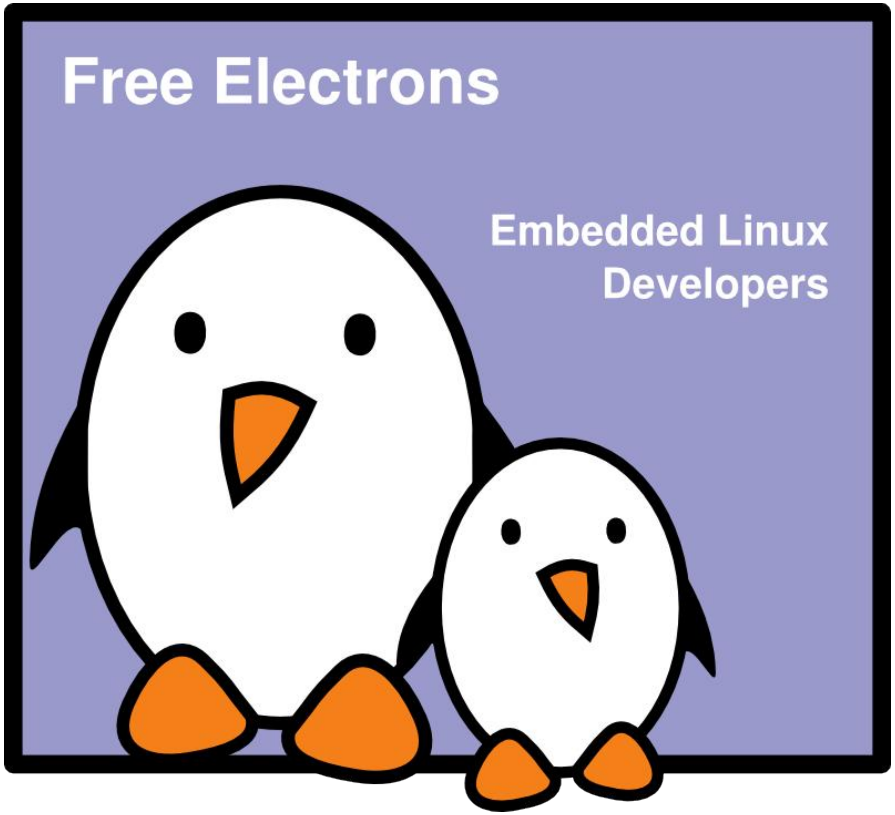
Real-time in embedded Linux systems

Michael Opdenacker
Thomas Petazzoni
Gilles Chanteperdrix
Free Electrons

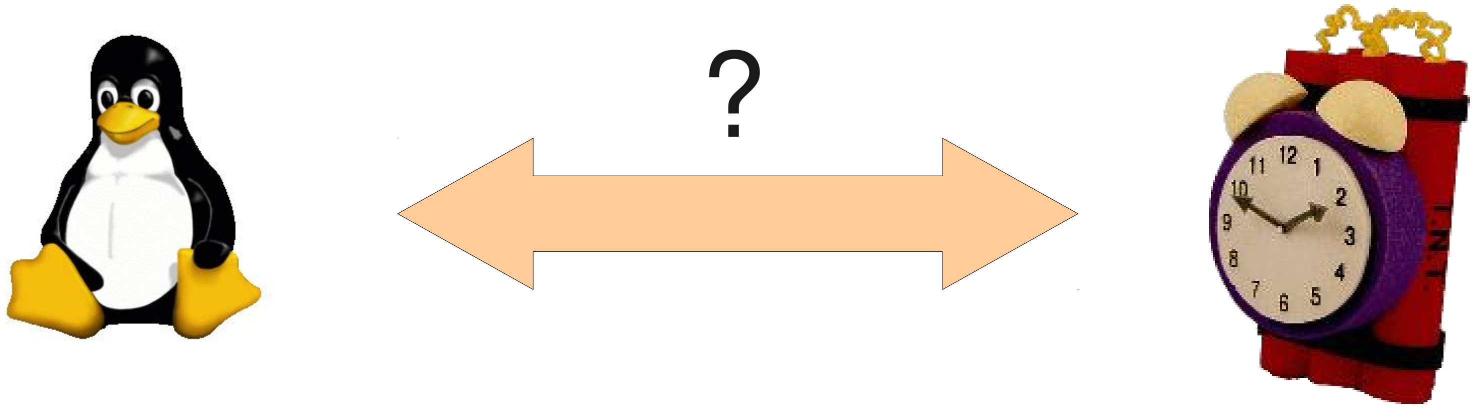


Latest update: Feb 21, 2011,
Document sources, updates and translations:
http://free-electrons.com/docs/realtime
Corrections, suggestions, contributions and translations are welcome!

Introduction

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



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

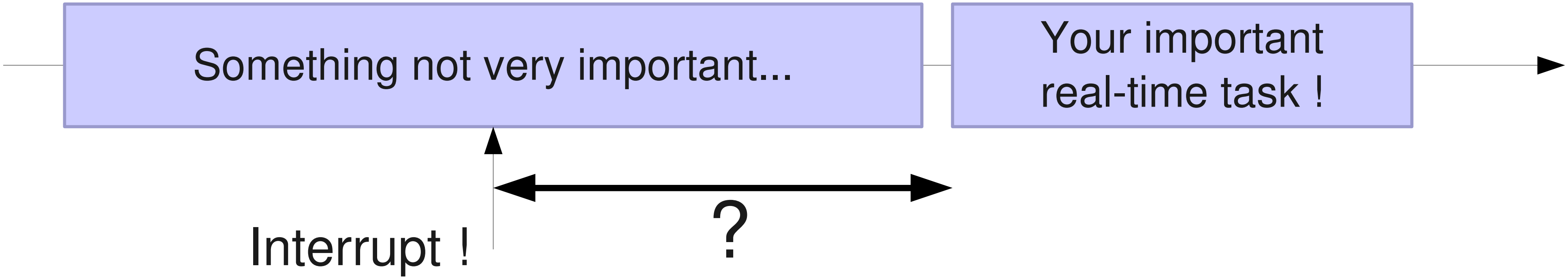
Linux and real-time approaches

- 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.
  - 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

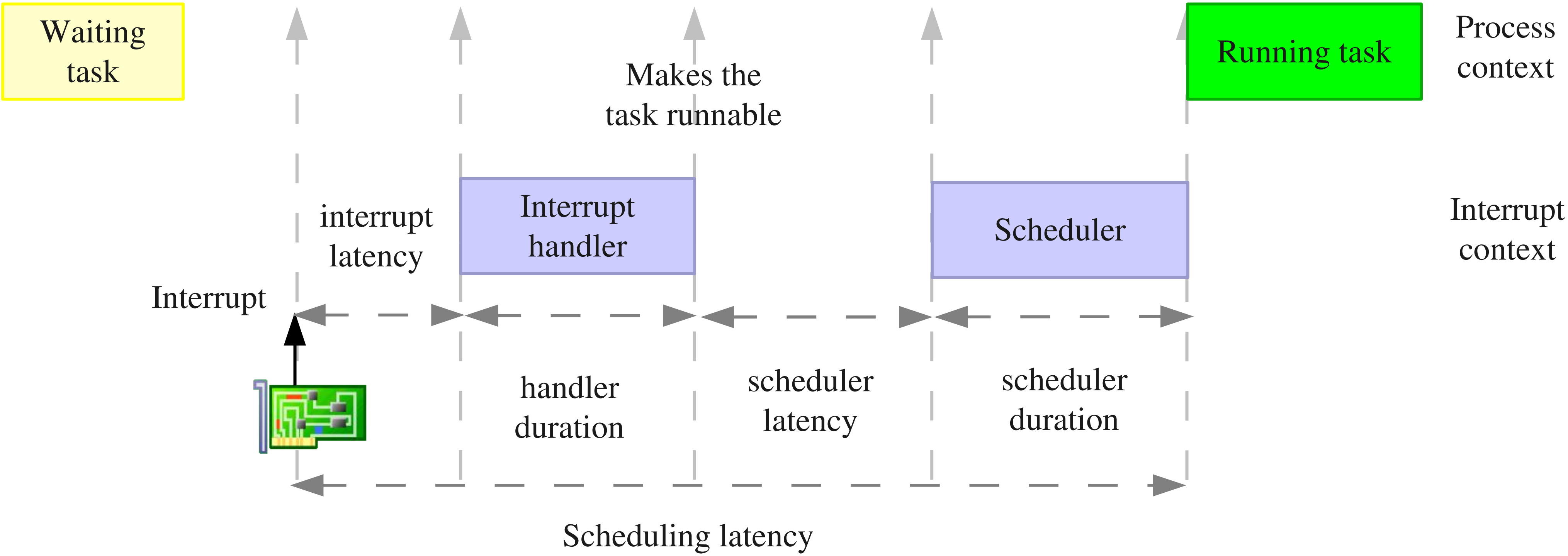
Approach 1
Improving the main Linux kernel with PREEMPT_RT

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*

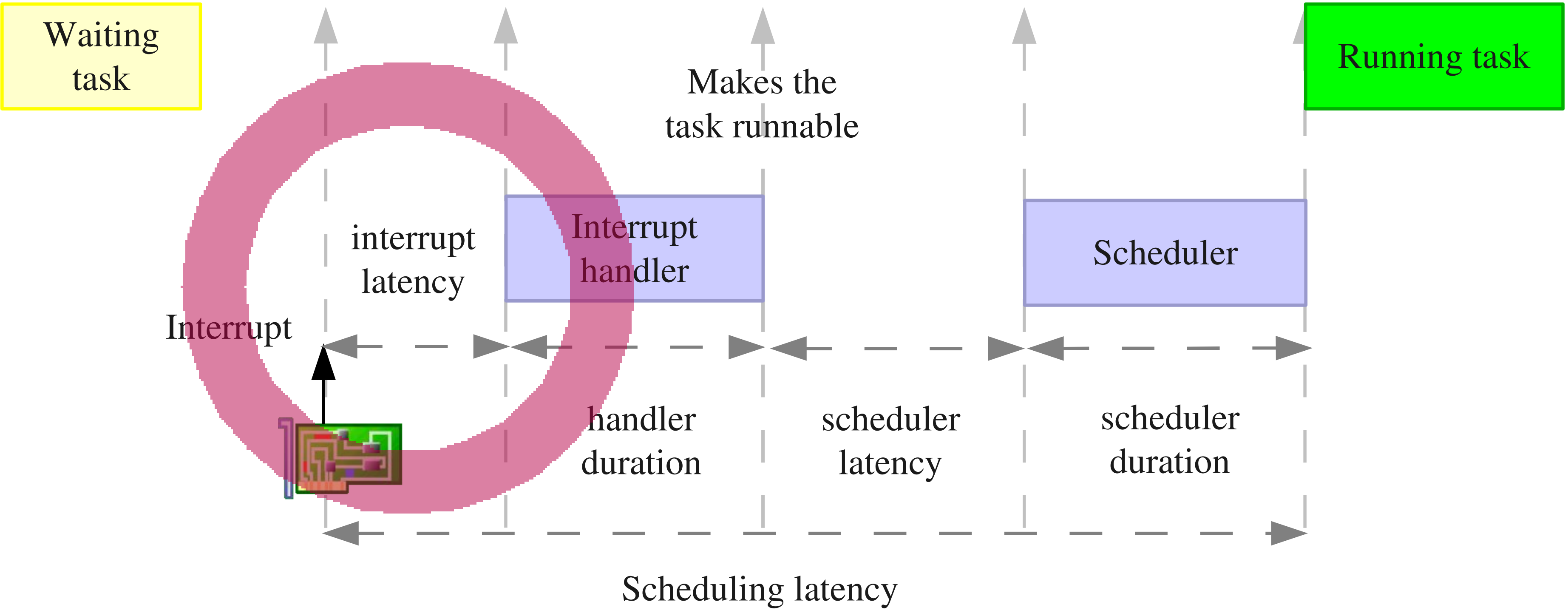

Something not very important...
Your important real-time task !
Interrupt !
?

Linux kernel latency components



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

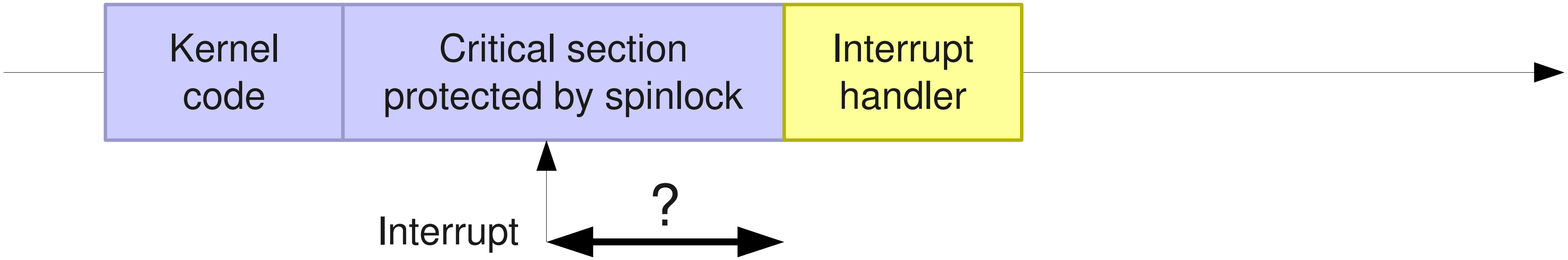
Interrupt latency



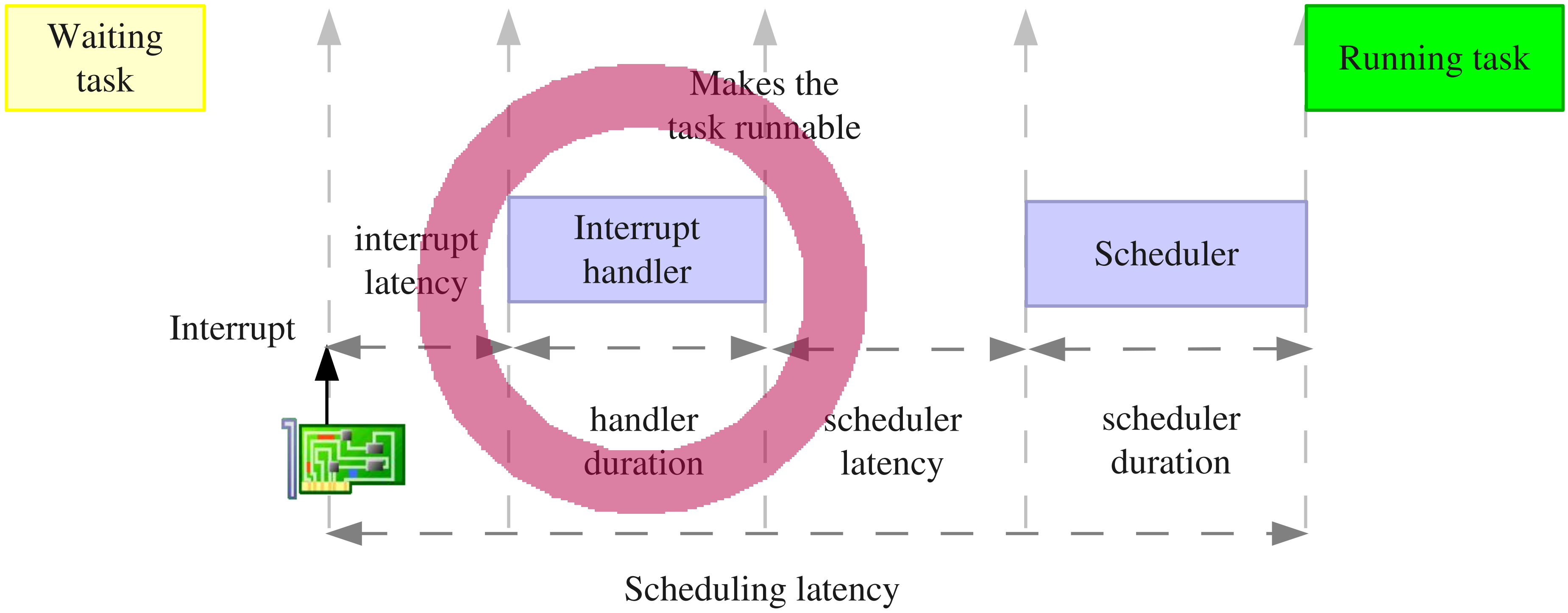
Time elapsed before executing the interrupt handler

Source of interrupt latency

- One of the concurrency prevention mechanism used in the kernel is the **spinlock**
- It has several variants, but one of the variant commonly used to 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 explictly 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



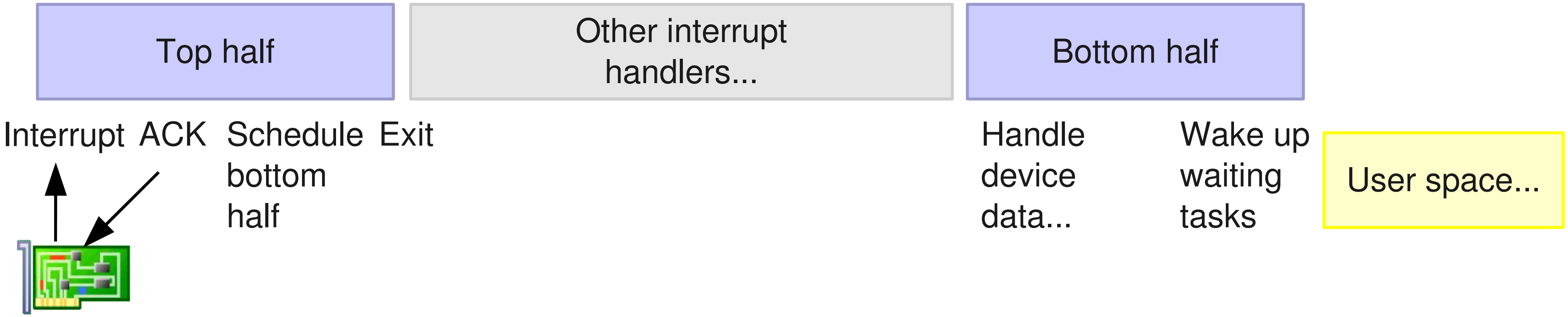
Interrupt handler duration



Time taken to execute the interrupt handler

Interrupt handler implementation

- In Linux, many interrupt handlers are split in two parts
  - A top-half, started by the CPU as soon as interrupt are enabled. It 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-half have completed their execution.
- Therefore, for real-time critical interrupts, bottom-half shouldn't be used: their execution is delayed by all other interrupts in the system.



Top half

Other interrupt handlers...

Bottom half

Interrupt

ACK

Schedule bottom half

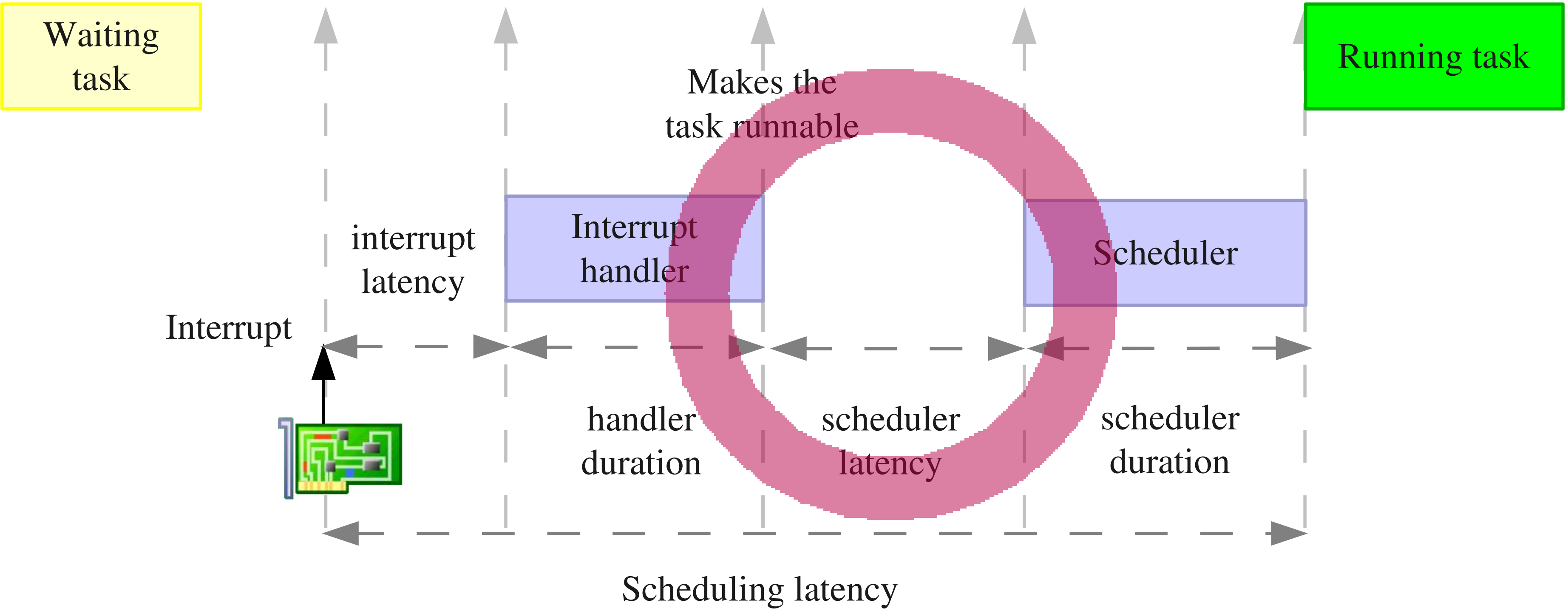
Exit

Handle device data...

Wake up waiting tasks

User space...

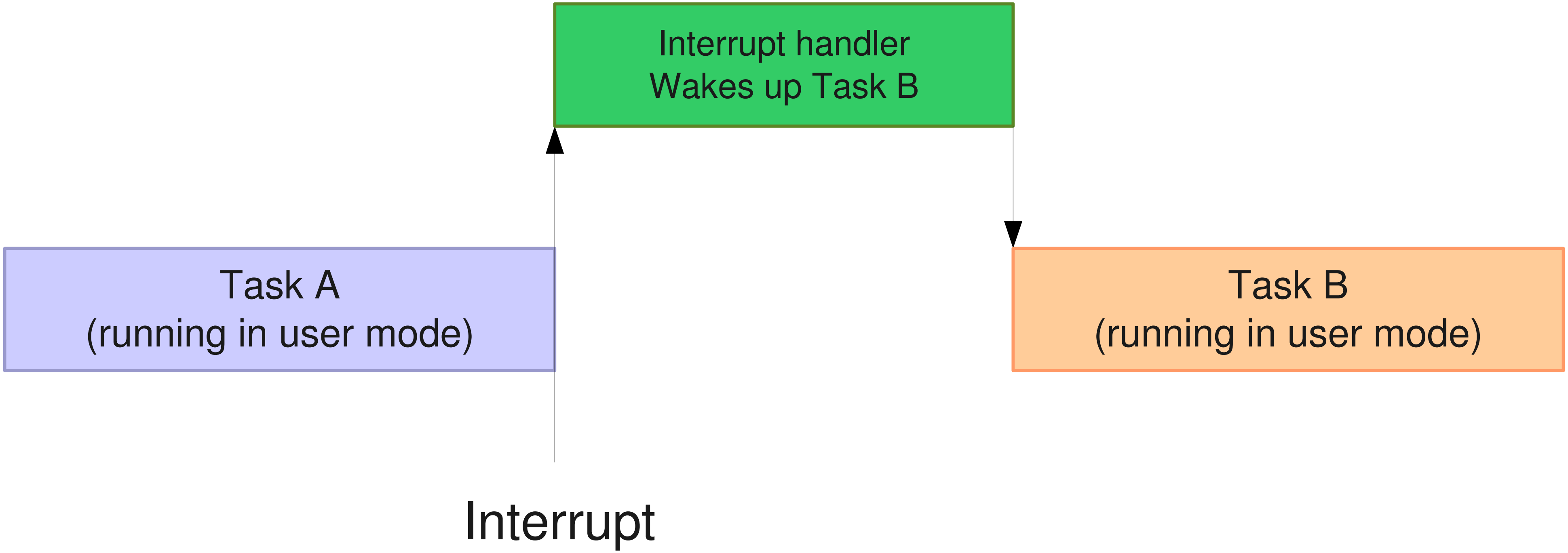
Scheduler latency



Time elapsed before executing the scheduler

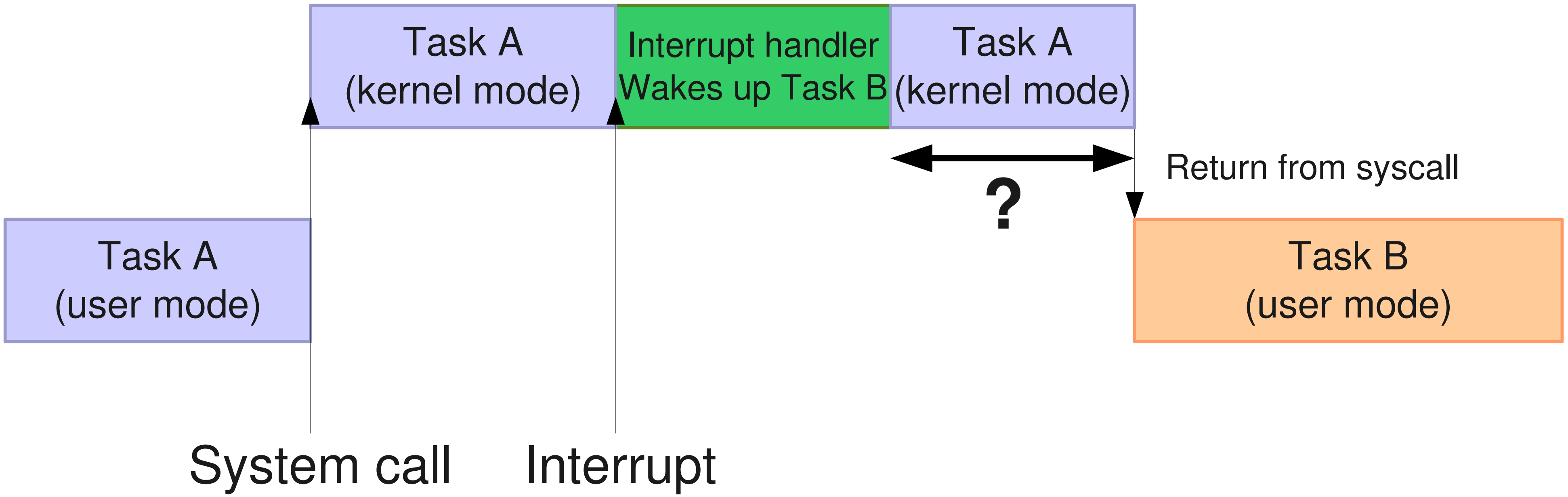
Understanding preemption (1)

- 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.

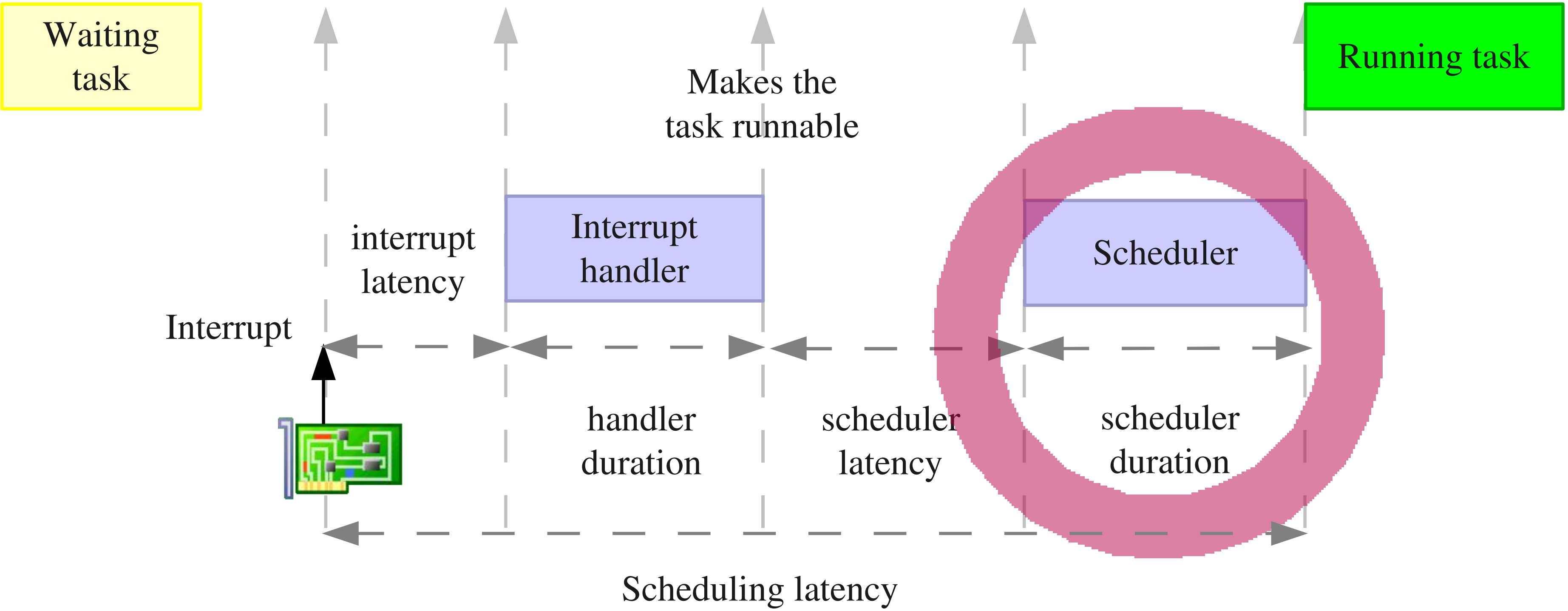


Understanding preemption (2)

- 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.*
- This means that the time before which the scheduler will be called to schedule another task is unbounded.



Scheduler duration



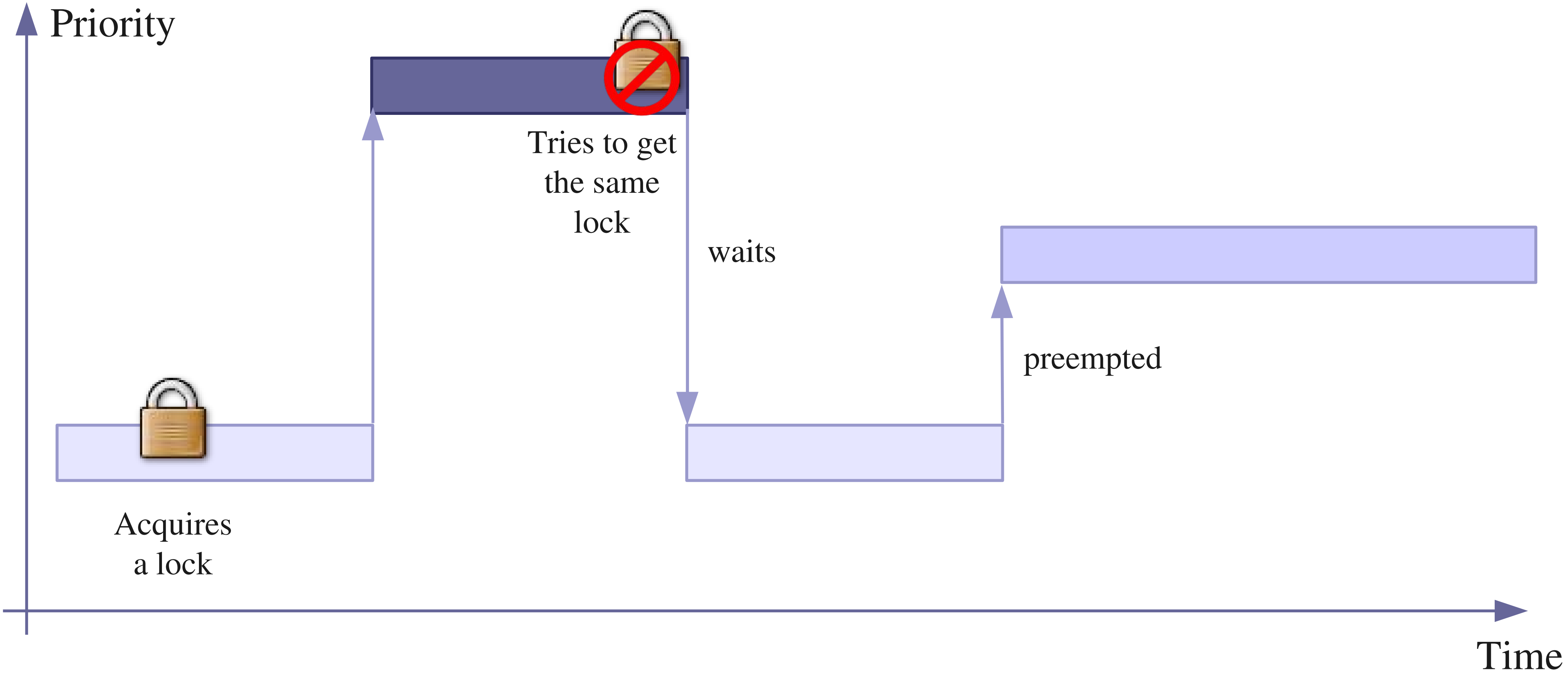
Time taken to execute the scheduler and switch to the new task.

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 creates huge delays.
- Many C library services or kernel services are not designed with real-time constraints in mind.

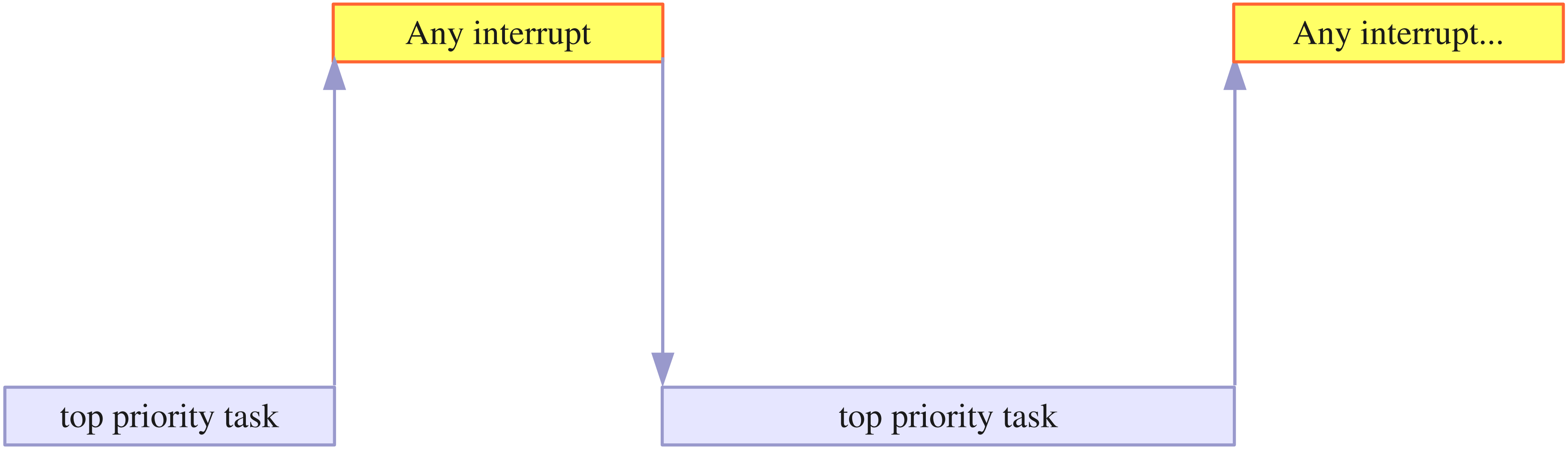
Priority inversion

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.



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 an higher priority than all tasks running on the system.



The PREEMPT_RT project

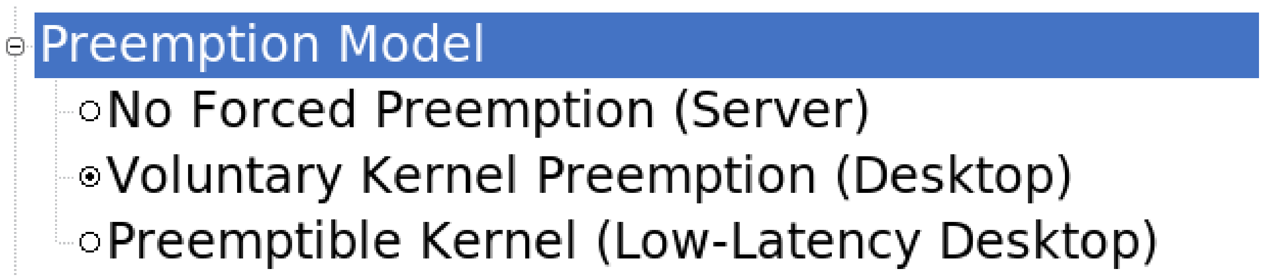
- Long-term project lead by Linux kernel developers Ingo Molnar, Thomas Gleixner and Steven Rostedt
  - https://rt.wiki.kernel.org
- The goal is to gradually improve the Linux kernel regarding real-time requirements and to get these improvements merged into the mainline kernel
  - PREEMPT_RT development 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

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

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

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.

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), 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.

Priority inheritance

- One classical solution to the priority inversion problem is called priority inheritance
  - The idea is that when a task of a low priority holds a lock requested by an higher priority task, the priority of the first task gets temporarly 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 explictly enabled on a per-mutex basis.

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

PREEMPT_RT specifics

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

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 !

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.

Setting up PREEMPT_RT

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

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.

Real-time application development

Development and compilation

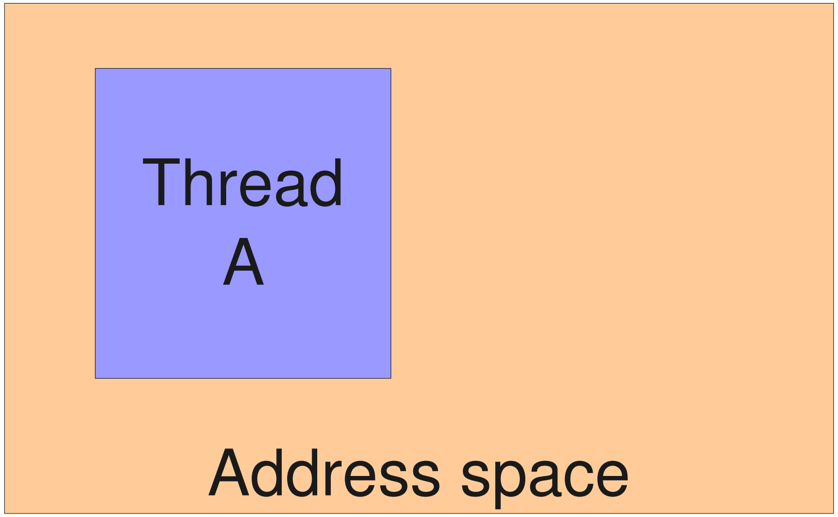
- No special library is needed, the POSIX realtime API is part of the standard C library
- The glibc or eglibc C libraries are recommended, as the support of some real-time features is not available yet in uClibc
  - Priority inheritance mutexes or NPTL on some architectures, for example
- Compile a 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 functioname`

Process, thread ?

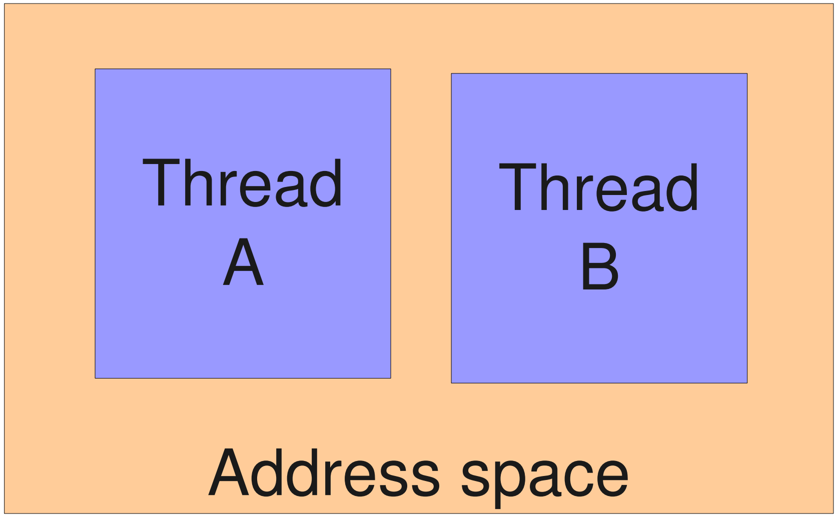
- Confusion about the terms «process», «thread» and «task»
- In Unix, a process is created using `fork()` and is composed of
  - An address space, which contains the program code, data, stack, shared libraries, etc.
  - One thread, that starts executing the main() function.
  - Upon creation, a process 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()`

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()`

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);
    ```

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)

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

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);
```

- Then the thread can be created using `pthread_create()`, passing the `attr` structure.
- Several other attributes can be defined this way: stack size, etc.

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

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 explictly be activated
  ```
  pthread_mutexattr_t attr;
  pthread_mutexattr_init (&attr);
  pthread_mutexattr_getprotocol
        (&attr, PTHREAD_PRIO_INHERIT);
  ```

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()

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.

Inter-process communication

- 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()`**

Debugging real-time latencies

ftrace - Kernel function tracer

New infrastructure that can be used for debugging or analyzing latencies and performance issues in the kernel.

- Developed by Steven Rostedt. Merged in 2.6.27.
  For earlier kernels, can be found from the rt-preempt patches.
- Very well documented in `Documentation/ftrace.txt`
- Negligible overhead when tracing is not enabled at run-time.
- Can be used to trace any kernel function!
- See our video of Steven's tutorial at OLS 2008:
  http://free-electrons.com/community/videos/conferences/

Using ftrace

- Tracing information available through the debugfs virtual fs (`CONFIG_DEBUG_FS` in the `Kernel Hacking` section)
- Mount this filesystem as follows:
  ```
  mount -t debugfs nodev /debug
  ```
- When tracing is enabled (see the next slides), tracing information is available in `/debug/tracing`.
- Check available tracers in `/debug/tracing/available_tracers`

Scheduling latency tracer

`CONFIG_SCHED_TRACER` (`Kernel Hacking` section)

- Maximum recorded time between waking up a top priority task and its scheduling on a CPU, expressed in µs.
- Check that `wakeup` is listed in `/debug/tracing/available_tracers`
- To select, reset and enable this tracer:

```
echo wakeup > /debug/tracing/current_tracer
echo 0 > /debug/tracing/tracing_max_latency
echo 1 > /debug/tracing/tracing_enabled
```

- Let your system run, in particular real-time tasks. Example: `chrt -f 5 sleep 1`
- Disable tracing:

```
echo 0 > /debug/tracing/tracing_enabled
```

- Read the maximum recorded latency and the corresponding trace:

```
cat /debug/tracing/tracing_max_latency
```

Useful reading

About real-time support in the standard Linux kernel

- Internals of the RT Patch, Steven Rostedt, Red Hat, June 2007
  http://www.kernel.org/doc/ols/2007/ols2007v2-pages-161-172.pdf
  Definitely worth reading.
- The Real-Time Linux Wiki: http://rt.wiki.kernel.org
  “The Wiki Web for the `CONFIG_PREEMPT_RT` community, and real-time Linux in general.”
  Contains nice and useful documents!
- See also our books page.

Approach 2

Real-time extensions to the Linux kernel

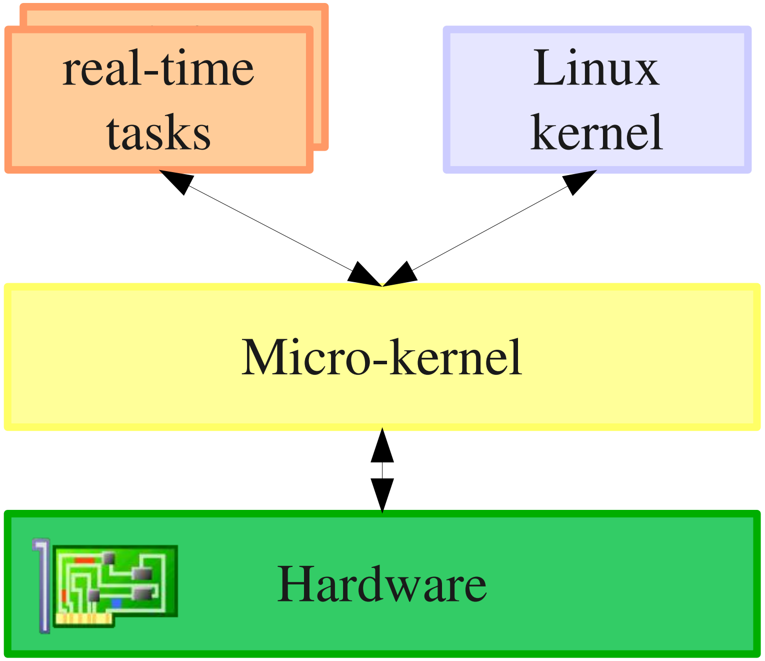
Linux real-time extensions

Three generations

- RTLinux
- RTAI
- Xenomai

A common principle

- Add a 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

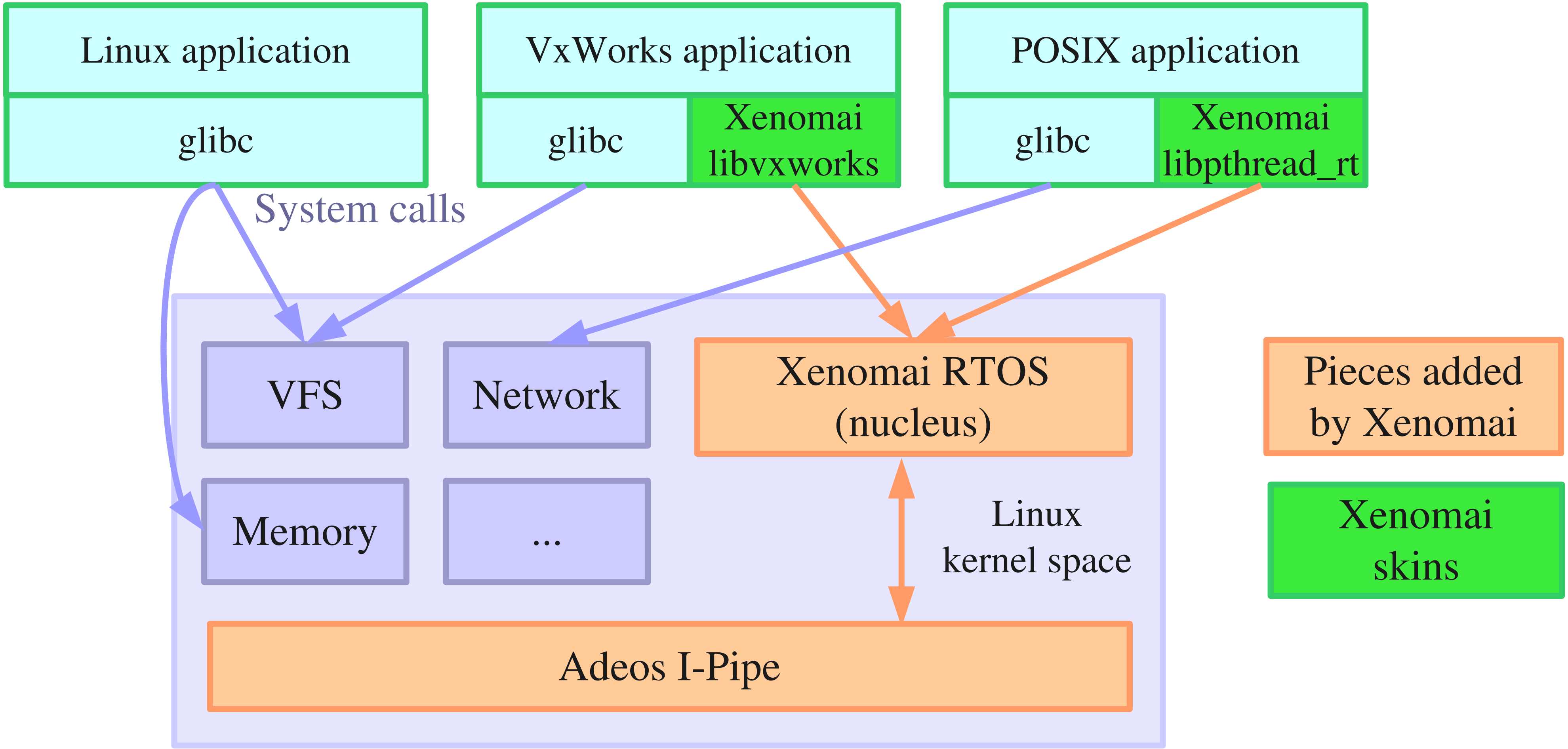
- Created in 1999, by Prof. Paolo Montegazza (long time contributor to RTLinux), Dipartimento di Ingegneria Aerospaziale Politecnico di Milano (DIAPM).
- Community project. Significant user base. Attracted contributors frustrated by the RTLinux legal issues.
- Only really actively maintained on x86
- 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/

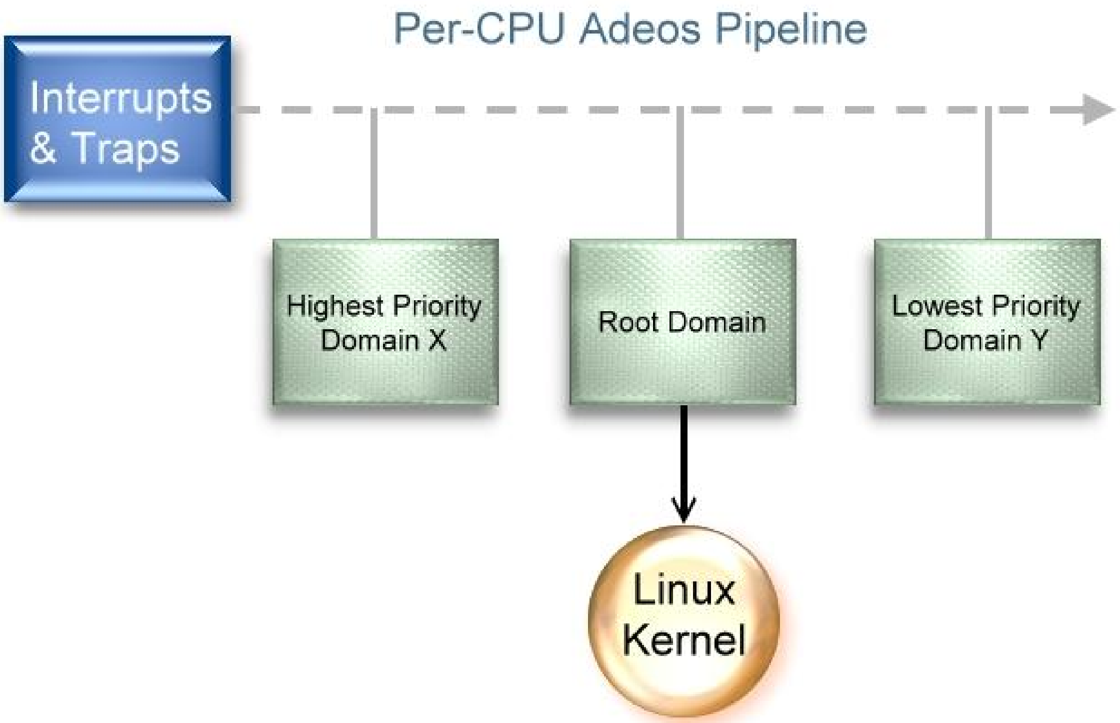
- Started in 2001 as a project aiming at emulating traditional RTOS.
- Initial goals: facilitate the porting of programs to GNU / Linux.
- Initially related to the RTAI project (as the RTAI / fusion branch), now independent.
- Skins mimicking the APIs of traditional RTOS such as VxWorks, pSOS+, and VRTXsa 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.

Xenomai architecture



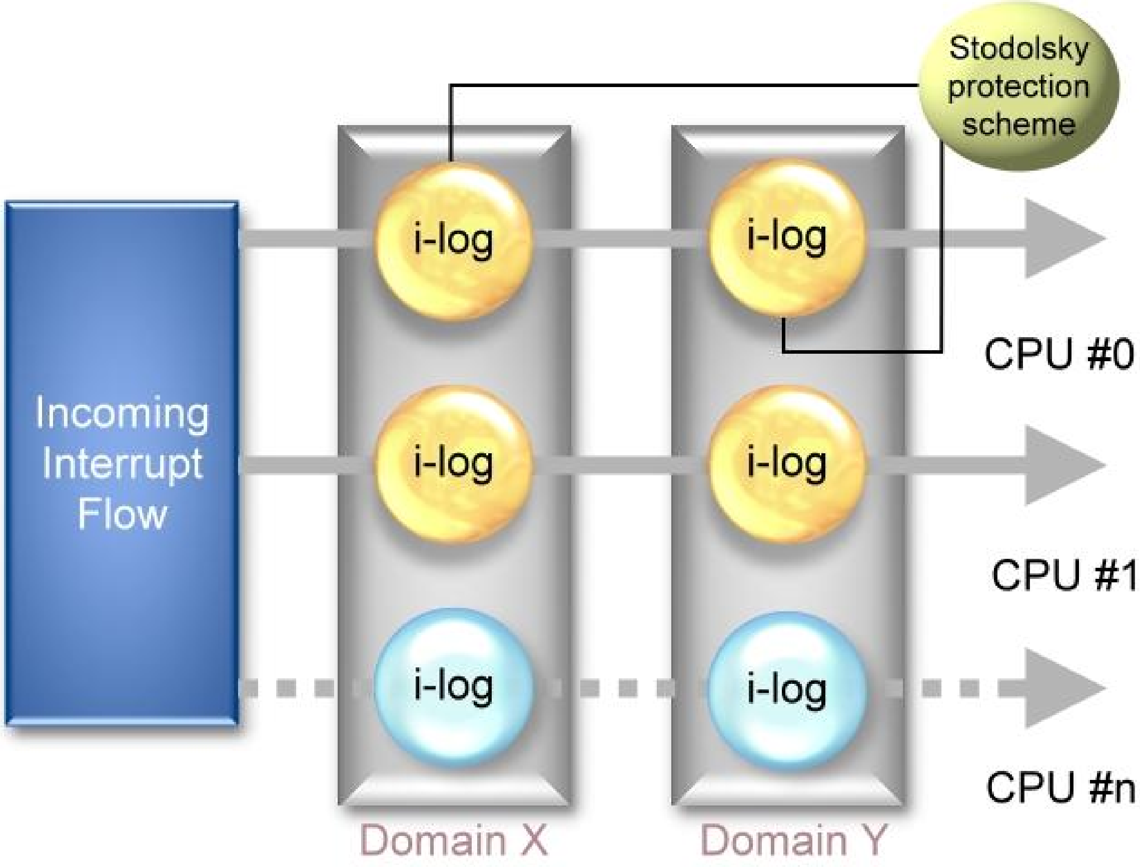
The Adeos interrupt pipeline abstraction

- From 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.



Adeos virtualized interrupts disabling

- Each domain may be “stalled”, meaning that it does not accept interrupts.
- Hardware interrupts are not disabled however (except for the domain leading the pipeline), instead the interrupts received during that time are logged and replayed when the domain is unstalled.



Adeos additional features

- The Adeos I-pipe patch implement additional features, essential for the implementation of the Xenomai real-time extension:
  - Disables on-demand mapping of kernel-space vmalloc/ioremap areas.
  - Disables copy-on-write when real-time processes are forking.
  - Allow subscribing to event allowing to follow progress of the Linux kernel, such as Linux system calls, context switches, process destructions, POSIX signals, FPU faults.
  - On the ARM architectures, integrates the FCSE patch, which allows to reduce the latency induced by cache flushes during context switches.

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* application, 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 looses 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)

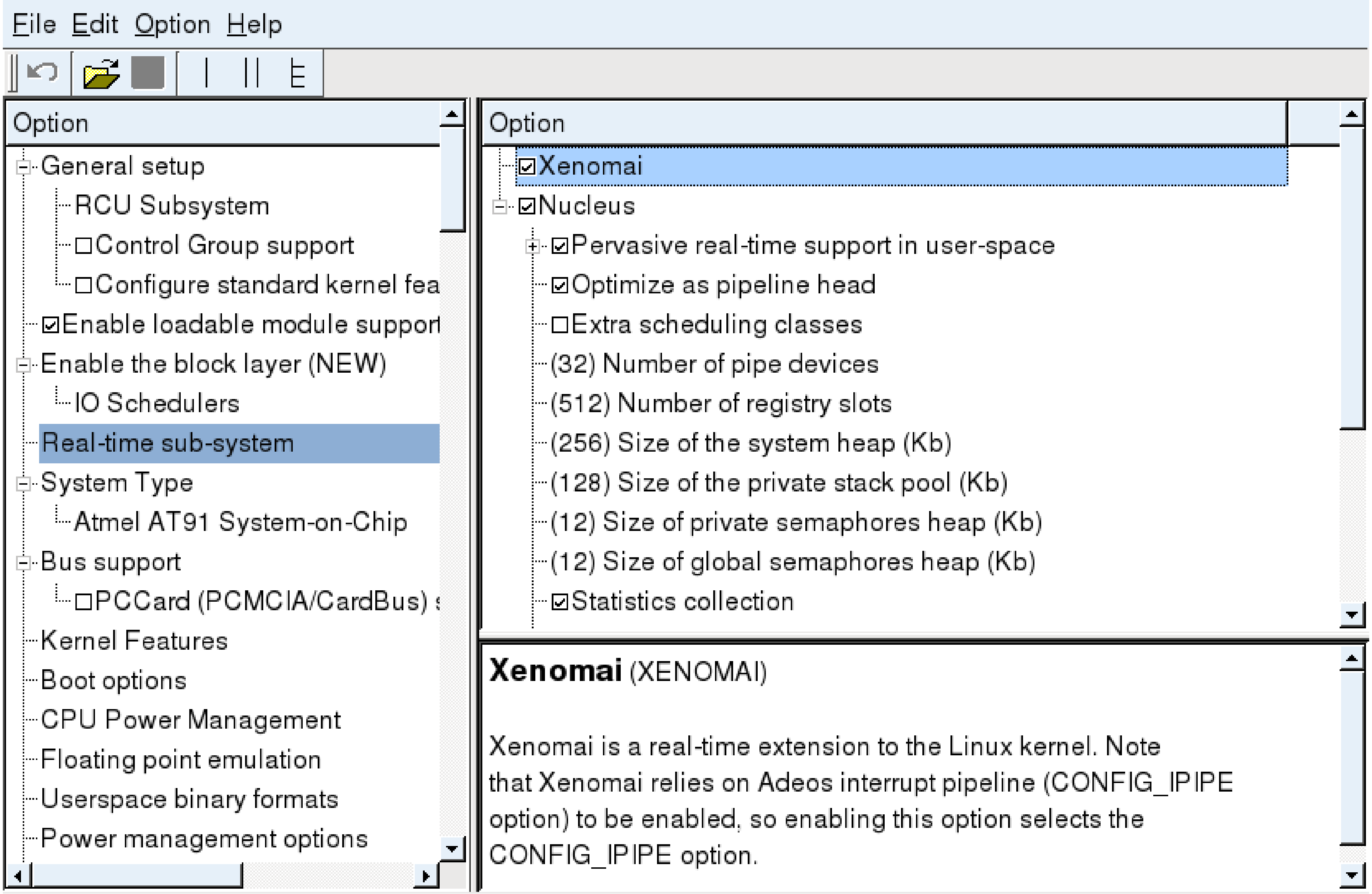
- An approach to unify the interfaces for developing device drivers and associated applications under real-time Linux
  - An API very similar to the native Linux kernel driver API
- Allows the development, in kernel space, of
  - Character-style device drivers
  - Network-style device drivers
- See the whitepaper on http://www.xenomai.org/documentation/xenomai-2.4/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.

Setting up Xenomai

How to build Xenomai

- Download Xenomai sources at http://download.gna.org/xenomai/stable/
- Download one of the Linux versions supported by this release (see `ksrc/arch/<arch>/patches/`)
- Since version 2.0, split kernel/user building model.
- Kernel uses a script called script/prepare-kernel.sh which integrates Xenomai kernel-space support in the Linux sources.
- Run the kernel configuration menu.

Linux options for Xenomai configuration



- General setup
  - RCU Subsystem
  - □Control Group support
  - □Configure standard kernel fea
- ☑Enable loadable module support
- Enable the block layer (NEW)
  - IO Schedulers
- Real-time sub-system
- System Type
  - Atmel AT91 System-on-Chip
- Bus support
  - □PCCard (PCMCIA/CardBus) s
- Kernel Features
- Boot options
- CPU Power Management
- Floating point emulation
- Userspace binary formats
- Power management options

- ☑Xenomai
- ☑Nucleus
  - ☑Pervasive real-time support in user-space
  - ☑Optimize as pipeline head
  - □Extra scheduling classes
  - (32) Number of pipe devices
  - (512) Number of registry slots
  - (256) Size of the system heap (Kb)
  - (128) Size of the private stack pool (Kb)
  - (12) Size of private semaphores heap (Kb)
  - (12) Size of global semaphores heap (Kb)
  - ☑Statistics collection

Xenomai (XENOMAI)

Xenomai is a real-time extension to the Linux kernel. Note that Xenomai relies on Adeos interrupt pipeline (CONFIG_IPIPE option) to be enabled, so enabling this option selects the CONFIG_IPIPE option.

Xenomai user-space support

- User-space libraries are compiled using the traditional autotools
  - `./configure --target=arm-linux && make && make DESTDIR=/your/rootfs/ install`
- The xeno-config script, installed when installing Xenomai user-space support helps you compiling your own programs.
- See Xenomai's `examples` directory.
- Installation details may be found in the README.INSTALL guide.
- For an introduction on programming with the native API, see: http://www.xenomai.org/documentation/branches/v2.3.x/pdf/Native-API-Tour-rev-C.pdf
- For an introduction on programming with the POSIX API, see: http://www.xenomai.org/index.php/Porting_POSIX_applications_to_Xenomai

Developing applications on Xenomai

The POSIX skin

- The POSIX skin allows to recompile without changes a traditional POSIX application so that instead of using Linux real-time services, it uses Xenomai services
  - Clocks and timers, condition variables, message queues, mutexes, semaphores, shared memory, signals, thread management
  - Good for existing code or programmers familiar with the POSIX API
- Of course, if the application uses any Linux service that isn't available in Xenomai, it will switch back to secondary mode
- To link an application against the POSIX skin

```
DESTDIR=/path/to/xenomai/
export DESTDIR
CFL=`$DESTDIR/bin/xeno-config --posix-cflags`
LDF=`$DESTDIR/bin/xeno-config --posix-ldflags`
ARCH-gcc $CFL -o rttest rttest.c $LDF
```

Communication with a normal task

- If a Xenomai real-time application using the POSIX skin wishes to communicate with a separate non-real-time application, it must use the *rtipc* mechanism
- In the Xenomai application, create an `IPCPROTO_XDDP` socket

```
socket(AF_RTIPC, SOCK_DGRAM, IPCPROTO_XDDP);
setsockopt(s, SOL_RTIPC, XDDP_SETLOCALPOOL,&poolsz,
sizeof(poolsz));
memset(&saddr, 0, sizeof(saddr));
saddr.sipc_family = AF_RTIPC;
saddr.sipc_port = MYAPPIDENTIFIER;
ret = bind(s, (struct sockaddr *)&saddr, sizeof(saddr));
```

  - And then the normal socket API `sendto()` / `recvfrom()`
- In the Linux application
  - Open `/dev/rtpX`, where X is the XDDP port
  - Use `read()` and `write()`

The native API (1)

- A Xenomai-specific API for developing real-time tasks
  - Usable both in user-space and kernel space. Development of tasks in user-space is the preferred way.
  - More coherent and more flexible API than the POSIX API. Easier to learn and understand. Certainly the way to go for new applications.
- Applications should include `<native/`*`service`*`.h>`, where service can be `alarm, buffer, cond, event, heap, intr, misc, mutex, pipe, queue, sem, task, timer`
- To compile applications :

```
DESTDIR=/path/to/xenomai/
export DESTDIR
CFL=`$DESTDIR/bin/xeno-config --xeno-cflags`
LDF=`$DESTDIR/bin/xeno-config --xeno-ldflags`
ARCH-gcc $CFL -o rttest rttest.c $LDF -lnative
```

The native API (2)

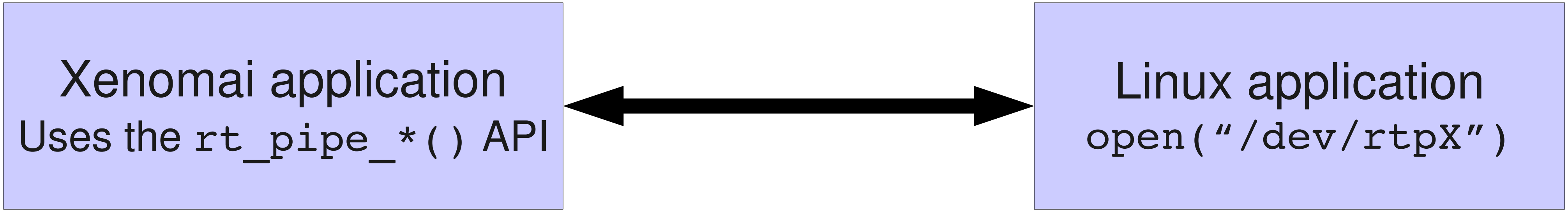
- Task management services
  - `rt_task_create(), rt_task_start(), rt_task_suspend(), rt_task_resume(), rt_task_delete(), rt_task_join(), etc.`
- Counting semaphore services
  - `rt_sem_create(), rt_sem_delete(), rt_sem_p(), rt_sem_v(), etc.`
- Message queue services
  - `rt_queue_create(), rt_queue_delete(), rt_queue_alloc(), rt_queue_free(), rt_queue_send(), rt_queue_receive(), etc.`
- Mutex services
  - `rt_mutex_create(), rt_mutex_delete(), rt_mutex_acquire(), rt_mutex_release(), etc.`

The native API (3)

- Alarm services
  - `rt_alarm_create(), rt_alarm_delete(), rt_alarm_start(), rt_alarm_stop(), rt_alarm_wait(), etc.`
- Memory heap services
  - Allows to share memory between processes and/or to pre-allocate a pool of memory
  - `rt_heap_create(), rt_heap_delete(), rt_heap_alloc(), rt_heap_bind()`
- Condition variable services
  - `rt_cond_create(), rt_cond_delete(), rt_cond_signal(), rt_cond_broadcast(), rt_cond_wait(), etc.`

Xenomai and normal task communication

- Using *rt_pipes*
- In the native Xenomai application, use the Pipe API
  - `rt_pipe_create()`, `rt_pipe_delete()`, `rt_pipe_receive()`, `rt_pipe_send()`, `rt_pipe_alloc()`, `rt_pipe_free()`
- In the normal Linux application
  - Open the corresponding `/dev/rtpX` file, the minor is specified at `rt_pipe_create()` time
  - Then, just `read()` and `write()` to the opened file



Xenomai application

Uses the `rt_pipe_*()` API

Linux application

`open(“/dev/rtpX”)`

Real-time approaches

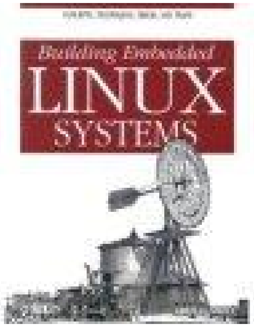
The following table is Paul Mac Kenney's summary of his own article describing the various approaches for real-time on Linux:

Approach	Quality	Inspection	API	Complexity	Fault isolation	HW/SW Configs
Vanilla Linux	10s of ms all services	All	POSIX + RT	N/A	None	All
PREEMPT	100s of us Schd, Int	preempt or irq disable	POSIX + RT	N/A	None	All
Nested OS (co-kernel)	~10us RTOS svcs	RTOS, hw irq disable	RTOS (can be POSIX RT)	Dual env.	Good	All
Dual-OS/Dual-Core (ASMP)	<1us RTOS svcs	RTOS	RTOS (can be POSIX RT)	Dual env.	Excellent	Specialized
PREEMPT_RT	10s of us Schd, Int	preempt and irq disable (most ints in process ctx), (mostly drivers)	POSIX + RT	"Modest" patch (careful tuning)	None	All (except some drivers)
Migration between OSes	? us RTOS svcs	RTOS, hw irq disable	RTOS (can be POSIX RT)	Dual env. (easy mix)	OK	All
Migration within OS	? us RTOS svcs	Sched, RTOS svcs	POSIX + RT	Small patch	None	All?

(additions in blue)

Full story at http://lwn.net/Articles/143323

Books



- Building Embedded Linux Systems, O'Reilly
  By Karim Yaghmour, Jon Masters,
  Gilad Ben-Yossef, Philippe Gerum and others
  (including Michael Opdenacker), August 2008

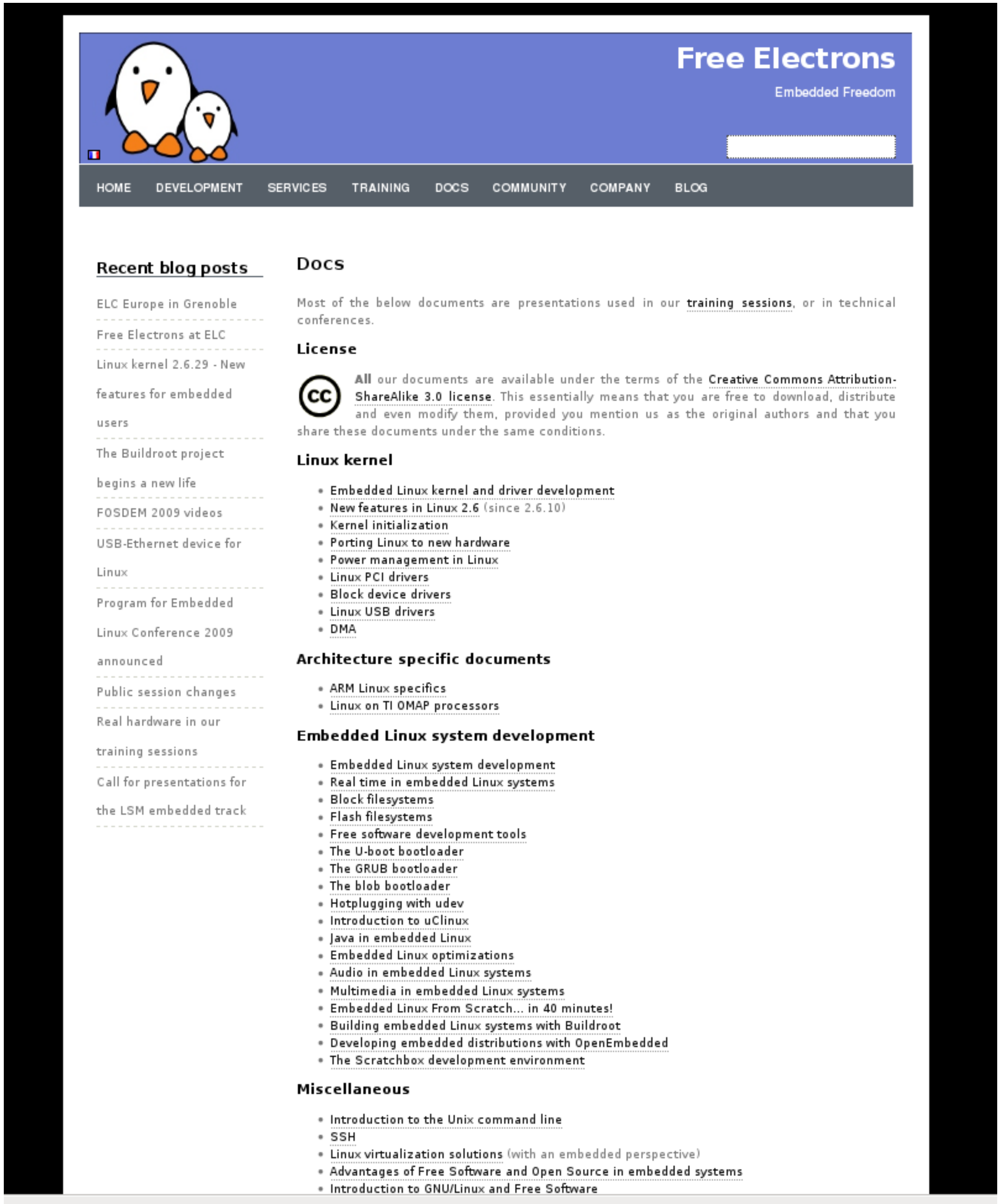
  A nice coverage of Xenomai (Philippe Gerum)
  and the RT patch (Steven Rostedt)
  http://oreilly.com/catalog/9780596529680/

★ ★ ★

Organizations

- http://www.realtimelinuxfoundation.org/
  Community portal for real-time Linux.
  Organizes a yearly workshop.
- http://www.osadl.org
  Open Source Automation Development Lab (OSADL)
  Created as an equivalent of OSDL for machine and plant control systems. Member companies are German so far (Thomas Gleixner is on board). One of their goals is to supports the development of RT preempt patches in the mainline Linux kernel (HOWTOs, live CD, patches).

Related documents



Free Electrons

Embedded Freedom

Recent blog posts

- ELC Europe in Grenoble
- Free Electrons at ELC
- Linux kernel 2.6.29 - New features for embedded users
- The Buildroot project begins a new life
- FOSDEM 2009 videos
- USB-Ethernet device for Linux
- Program for Embedded Linux Conference 2009 announced
- Public session changes
- Real hardware in our training sessions
- Call for presentations for the LSM embedded track

Docs

Most of the below documents are presentations used in our training sessions, or in technical conferences.

License

All our documents are available under the terms of the Creative Commons Attribution-ShareAlike 3.0 license. This essentially means that you are free to download, distribute and even modify them, provided you mention us as the original authors and that you share these documents under the same conditions.

Linux kernel

- Embedded Linux kernel and driver development
- New features in Linux 2.6 (since 2.6.10)
- Kernel initialization
- Porting Linux to new hardware
- Power management in Linux
- Linux PCI drivers
- Block device drivers
- Linux USB drivers
- DMA

Architecture specific documents

- ARM Linux specifics
- Linux on TI OMAP processors

Embedded Linux system development

- Embedded Linux system development
- Real time in embedded Linux systems
- Block filesystems
- Flash filesystems
- Free software development tools
- The U-boot bootloader
- The GRUB bootloader
- The blob bootloader
- Hotplugging with udev
- Introduction to uClinux
- Java in embedded Linux
- Embedded Linux optimizations
- Audio in embedded Linux systems
- Multimedia in embedded Linux systems
- Embedded Linux From Scratch... in 40 minutes!
- Building embedded Linux systems with Buildroot
- Developing embedded distributions with OpenEmbedded
- The Scratchbox development environment

Miscellaneous

- Introduction to the Unix command line
- SSH
- Linux virtualization solutions (with an embedded perspective)
- Advantages of Free Software and Open Source in embedded systems
- Introduction to GNU/Linux and Free Software

All our technical presentations
on http://free-electrons.com/docs

- Linux kernel
- Device drivers
- Architecture specifics
- Embedded Linux system development

How to help

You can help us to improve and maintain this document...

- By sending corrections, suggestions, contributions and translations
- By asking your organization to order development, consulting and training services performed by the authors of these documents (see http://free-electrons.com/).
- By sharing this document with your friends, colleagues and with the local Free Software community.
- By adding links on your website to our on-line materials, to increase their visibility in search engine results.

Free Electrons

Our services

Linux kernel

Linux device drivers
Board support code
Mainstreaming kernel code
Kernel debugging

Embedded Linux Training

All materials released with a free license!

Unix and GNU/Linux basics
Linux kernel and drivers development
Real-time Linux, uClinux
Development and profiling tools
Lightweight tools for embedded systems
Root filesystem creation
Audio and multimedia
System optimization

Custom Development

System integration
Embedded Linux demos and prototypes
System optimization
Application and interface development

Consulting and technical support

Help in decision making
System architecture
System design and performance review
Development tool and application support
Investigating issues and fixing tool bugs