Making Linux do Hard Real-time

Jim Huang (黃敬群) <jserv.tw@gmail.com>

Jul 26, 2014

Rights to copy

Agenda

- General Concepts
  - Linux scheduler
  - Interrupt Handling
  - Latency in Linux
- Real-time Features in Linux
  - RT-Preempt
- Linux Realtime Extension: Xenomai

General Concepts

Real Time

Real Time != Real Fast

- is determinism
  - RTOS will deliver decent overall throughput (performance) but can sacrifice throughput for being deterministic (or predictable)
- Hard real-time
  - Missing a deadline is a total system failure
- Soft real-time
  - The usefulness of a result degrades after its deadline, thereby degrading the system's quality of service

Hard Real Time

A system is considered as a *hard real time* if it can answer to an internal or external stimulus **within a given maximum amount of time**. “Guaranteed worst case”

Hard real time systems are used wherever failing to react in time can cause a system failure or damage, or put its users in danger.

Typical examples:

- air traffic control
- vehicle subsystems control
- Medicine (pacemakers, etc.)

Soft Real Time

A system is considered as *soft real time* if it is built to react to stimuli as quickly as it can: “best effort”.

However, if the system loses events or fails to process them in time, there is no catastrophic consequence on its operation. There is just a degradation in quality.

Typical examples

- Voice over IP
- multimedia streaming
- computer games

Standard Linux and Real Time

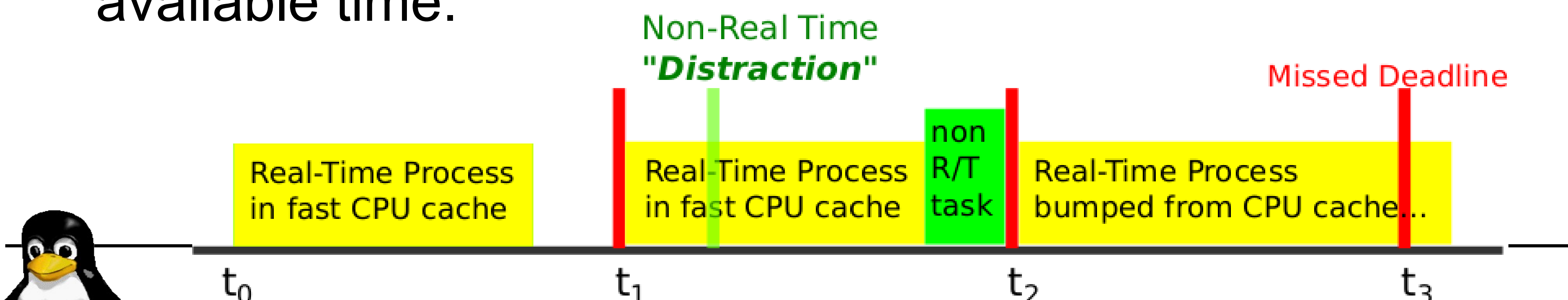
The vanilla Linux kernel was not designed as a hard real time system:

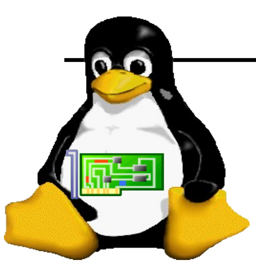
- Like Unix, it is a time sharing operating system designed to maximize throughput and give a fair share of the CPU in a multi-user environment.
  - scheduler avoids process starvation
- Non deterministic timing behavior of some kernel services: memory allocation, system calls...
- By default, processes are not preemptible when they execute system calls.

Throughput vs. Determinism

Linux as optimized for throughput rather than determinism.

- Linux CPUs typically utilize large, multi-layer memory caches
- Caches make CPUs run like a hare
    - but, in real-time systems, the tortoise wins!
- CPU memory caching prevents Hard RT processes from safely utilizing more than a small fraction of the available time.





Realtime Performance

Application program

100% source compatibility
better binary compatibility

API

Customized Realtime kernel

Real-time Linux

(Standard) Linux

Response time: < 10 µs

One (physical)
address space with n-tasks

Response time: ~1ms

One process address space
with n-threads

All tasks have
one common API
implemented by
one common library

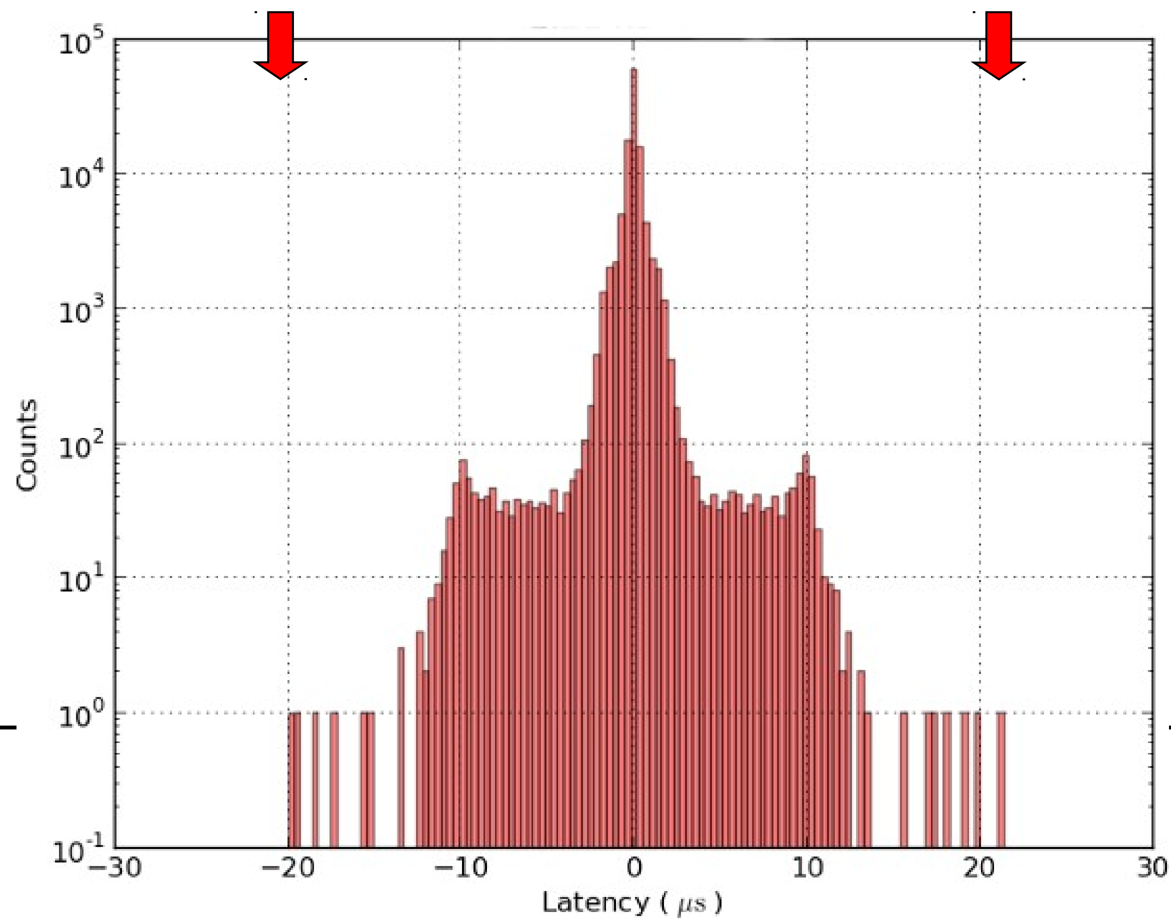
Response time: < 10 µs
same
homogeneity

All threads of a process have
one common API
implemented by
one common library

Jitter

It means the scatter of the control period of all the tasks defined in the system

- The total jitter is the time difference between the minimal response time and the maximal response time



Real-time Approaches

Two major approaches real time Linux

- rt-preempt (PREEMPT_RT patch)
  - Allows preemption, so minimize latencies
  - Execute all activities (including IRQ) in "schedulable/thread" context
  - Many of the RT patch have been merged
- Linux (realtime) extensions
  - Add extra layer between hardware and the Linux kernel to manage real-time tasks separately

Linux Scheduler

Linux Scheduler

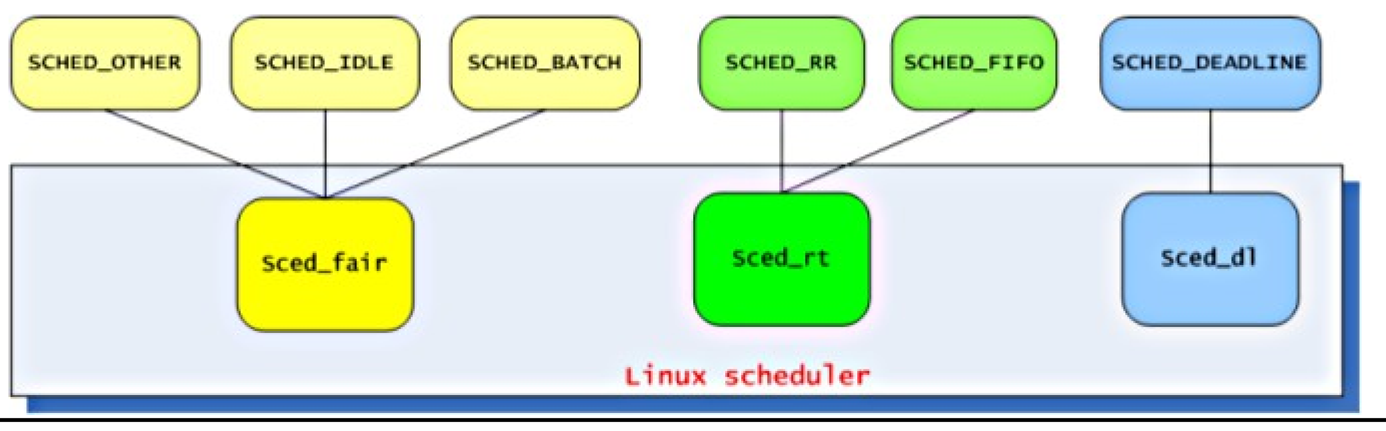
- Decides which process to run next
- Allocates processor(s)' time among run-able processes
- Realizes multi-tasking in a single processor machine
- Schedules SMP as well
- Affects how the system behaves, responsiveness

Linux Scheduler Framework

- Kernel supports various scheduling policies by plug-in
- Scheduling classes contains details of the scheduling policy
  - Each operation can be requested by the global scheduler;
  - Allows for creating of the generic scheduler without any knowledge about different scheduler classes

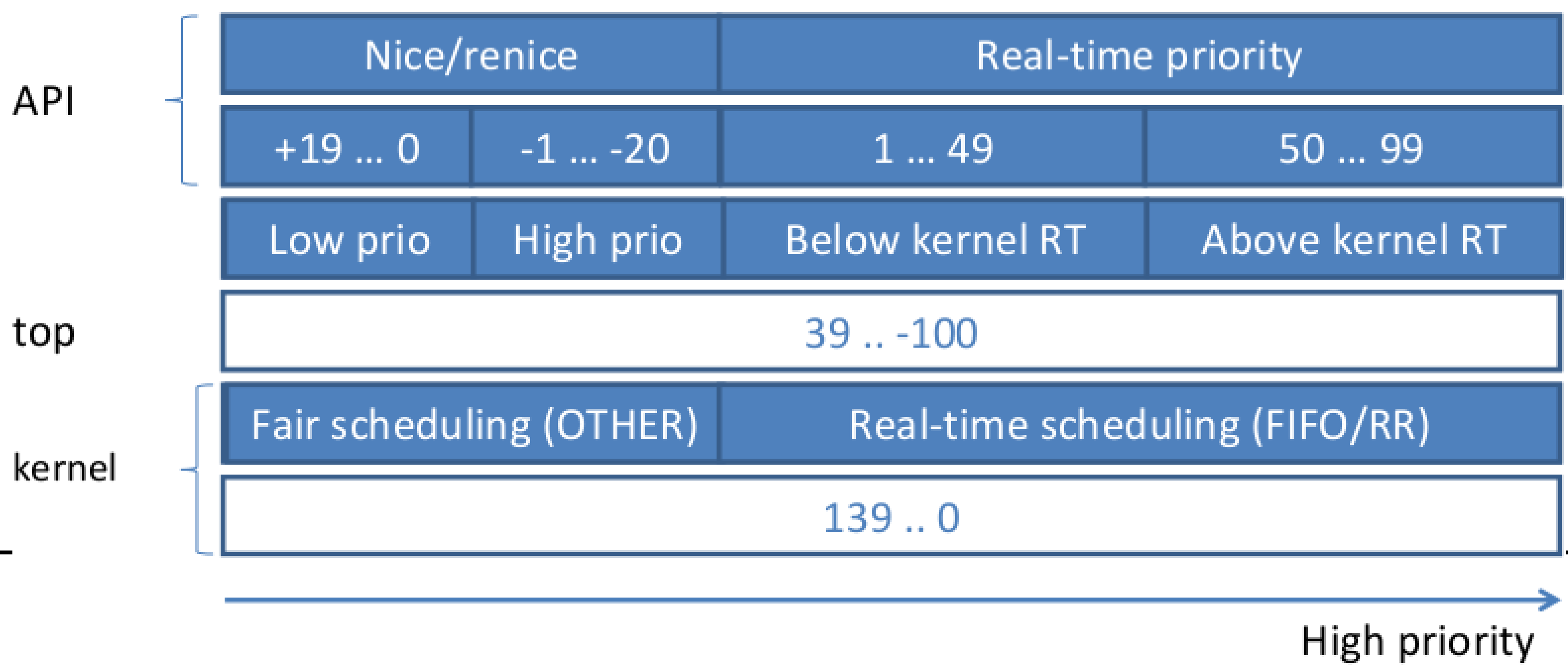
Linux Scheduler Classes

- SCHED_OTHER (default, non-RT)
- SCHED_FIFO
  - Use FIFO real-time scheduling
- SCHED_RR
  - Use round-robin real-time scheduling



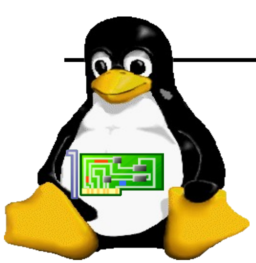
Linux Scheduler Priority

- Two separate priority ranges
  - Nice value: -20 ... +19 (19 being the lowest)
  - Real-time priority: 0 ... 99 (higher value is higher prio)
  - ps priority: 0 ... 139 (0: lowest and 139: highest)



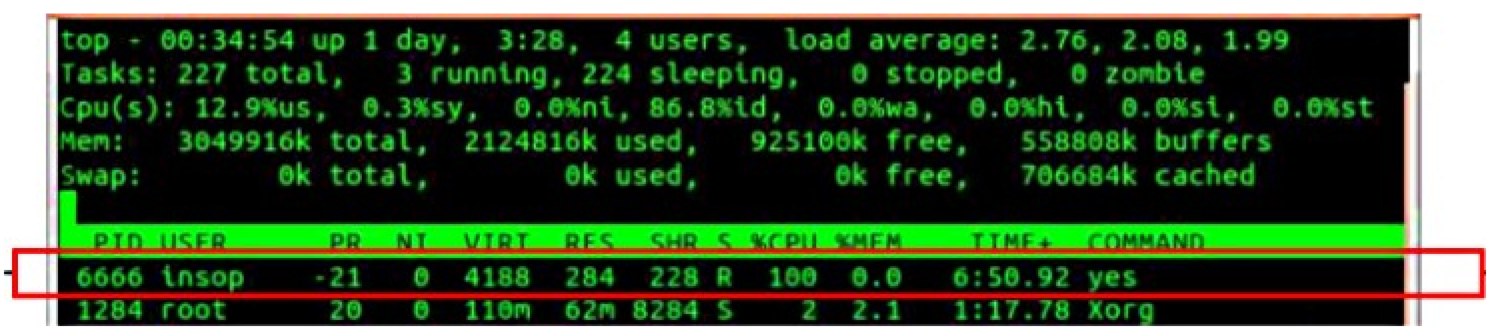
Linux Scheduler System Calls

System call	description
nice()	Sets a process' nice values
sched_setscheduler()	Sets a process' scheduling policy
sched_getscheduler()	Gets a process' scheduling policy
sched_setparam()	Sets a process' real-time policy
sched_getparam()	Gets a process' real-time policy
sched_get_priority_max()	Gets the max real-time priority
sched_get_priority_min()	Gets the min real-time priority
sched_rr_get_interval()	Gets a process' timeslice value
sched_setaffinity()	Sets a process' processor affinity
sched_getaffinity()	Gets a process' processor affinity
sched_yield()	Temporarily yields the processor



Linux Scheduler Utilities

- How to configure priority
  - **chrt** [option] **-p** [prio] pid
    - -r: SCHED_RR; -f: SCHED_FIFO
  - sched_setscheduler()
  - Checking priority, nice, rt-priority
    - ps -eo pid,class,pri,nice,rtprio,comm



Interrupt Handling

Interrupts (1)

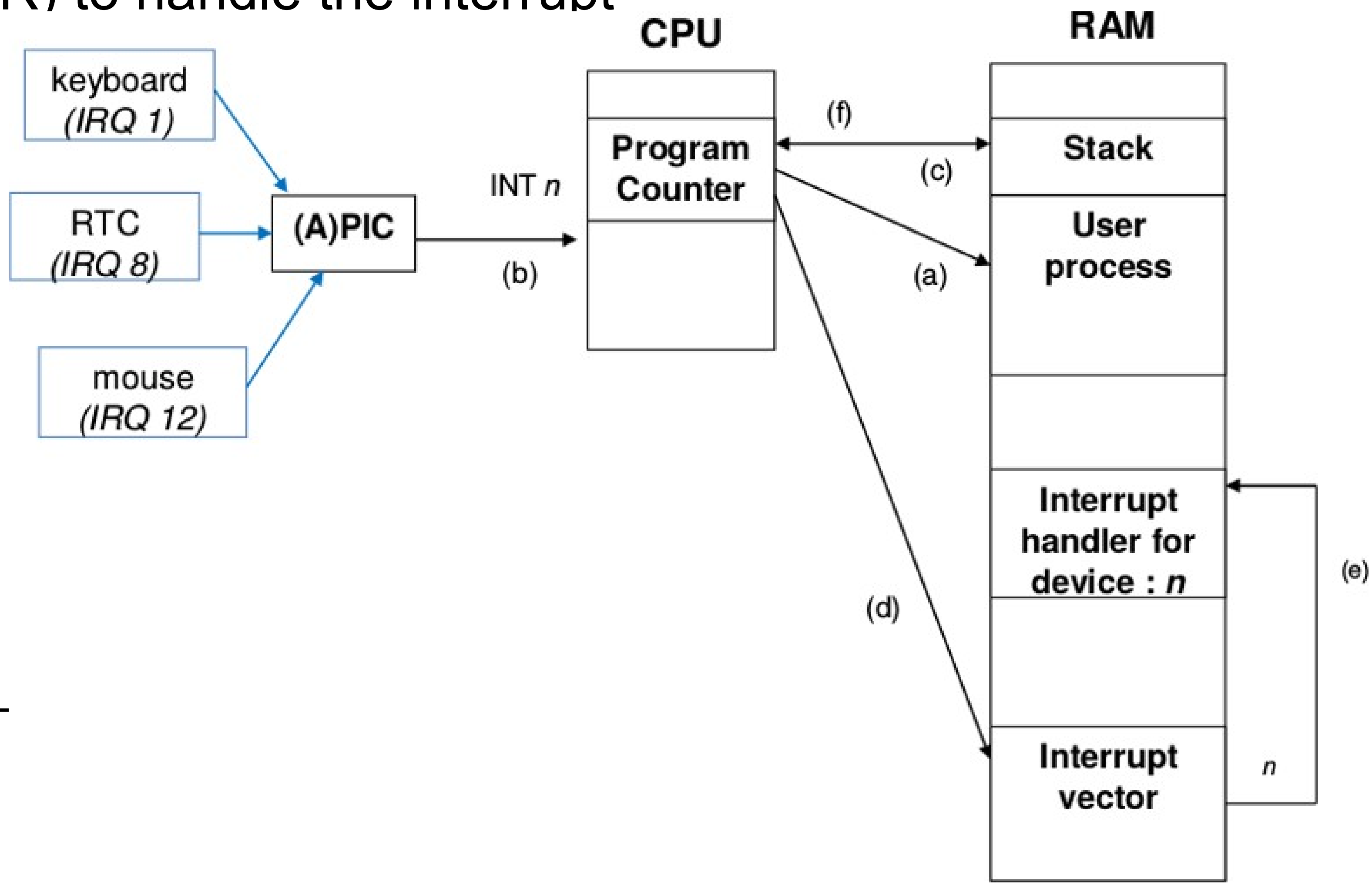
- I/O devices can be checked if ready to transfer data by pooling (ie. continuously checking device registers)

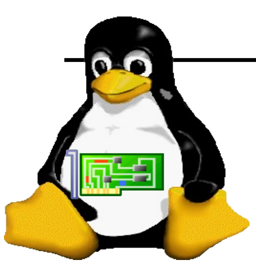
```
while (<Device is NOT ready for transfer data>)
    do_wait();
Transfer_Data();
```

  - This operation has a busy-waiting part which consumes CPU
- To minimize busy-waiting, checking registers can be done by a periodic task
  - If no data is ready to transfer the periodic task sleeps until the next period
- The busy waiting condition can be completely removed using interrupts

Interrupts (2)

- When an I/O device is ready to transfer data it signals the CPU (issues an interrupt)
- The CPU interrupts the current task an calls an Interrupt Service Routine (ISR) to handle the interrupt



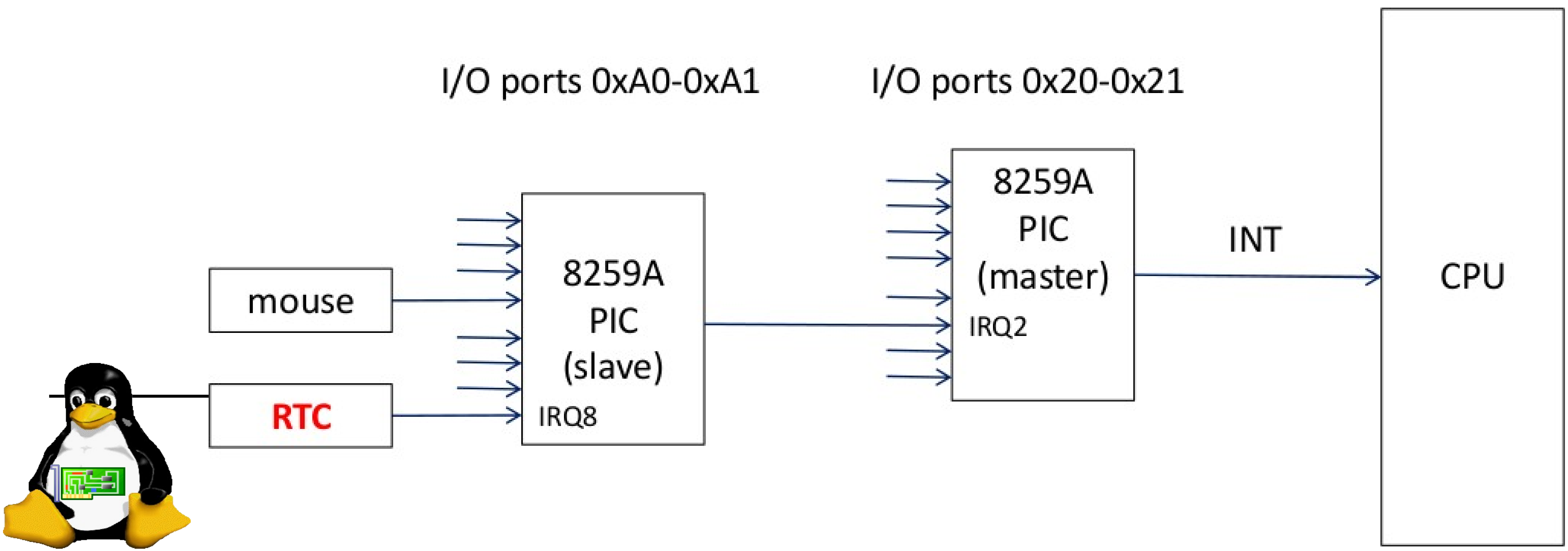


Interrupts (3)

- The length of Linux interrupt vector is 256
  - 32 entries are reserved to handle CPU exceptions
- The remaining 224 entries can point:
  - software interrupts' handlers (like int 0x80: system calls)
  - hardware interrupts' handlers (assigned to a IRQ line)
- On x86 architecture normally only 16 IRQ lines are available

Interrupts (4)

- Devices are connected to two Programmable Interrupt Controllers (intel 8259x) :
- A maximum of 8 8259s can be connected in cascade
- An ISR will be executed every time an interrupt occur in the IRQ line assigned
  - The RTC can be used to generate interrupts on IRQ 8.



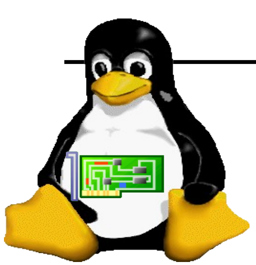
Interrupts (5)

- PICs can be accessed through I/O ports
- see how many PICs are available; IRQ lines reserved:

```
cat /proc/ioports
```

```
cat /proc/interrupts
           CPU0
  0:        163   IO-APIC-edge      timer
  1:       2901   IO-APIC-edge      i8042
  3:          1   IO-APIC-edge
  4:          1   IO-APIC-edge
  6:          5   IO-APIC-edge      floppy
  7:          0   IO-APIC-edge      parport0
  8:         16   IO-APIC-edge
  9:          0   IO-APIC-fasteoi   acpi
 12:       9848   IO-APIC-edge      i8042
 14:         93   IO-APIC-edge      ata_piix
 15:    2573571   IO-APIC-edge      ata_piix
 16:          0   IO-APIC-fasteoi   ehci_hcd:usb2
 17:      19463   IO-APIC-fasteoi   ioc0
 18:       6343   IO-APIC-fasteoi   eth0
 19:          0   IO-APIC-fasteoi   uhci_hcd:usb1, Ensoniq AudioPCI
NMI:          0
LOC:     378503 //Total interrupts on local processor
(…)
ERR:          0 //Errors on interrupts
MIS:          0
```

interrupts can be shared

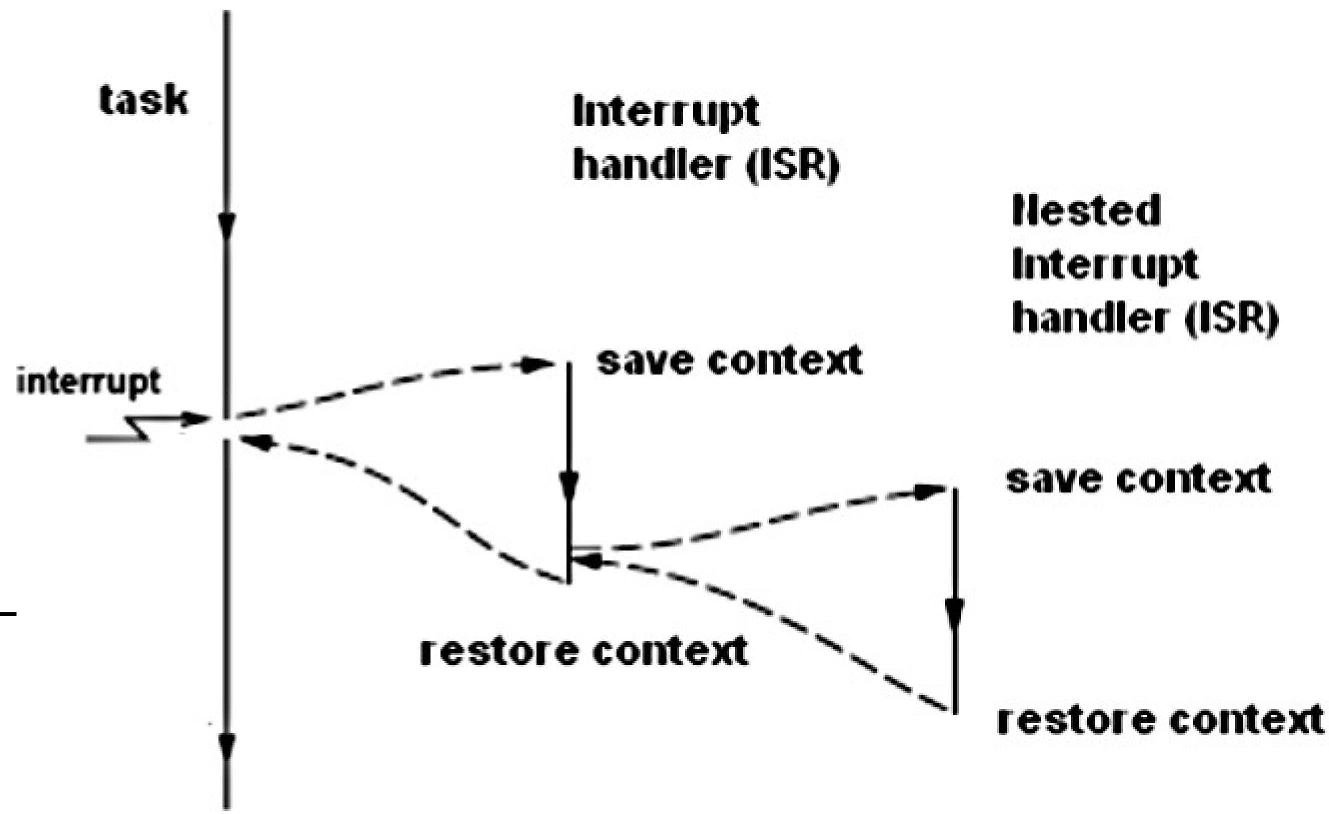


Context Switching (1)

- The context of the interrupted task (at least the values of the CPU registers at the time of the interrupt) must be preserved, so that it can be resumed with the same state
- Task context can be stored in:
  - Interrupted task local stack
  - system stack
  - in a set of specialized CPU registers
- Probably the most efficient is the last one
  - If the CPU itself has some registers to save context, it can be done in a few cycles (or just one)
  - Some DSPs and PowerPCs have a set of these registers

Context Switching (2)

- If the ISR itself can not be interrupted any one of those 3 strategies can be used to save task context
- However sometimes it is needed to interrupt an ISR (to handle a fast I/O device which issues an high priority interrupt)
- A Programmable interrupt controller PIC can have some levels (priorities of interrupts)

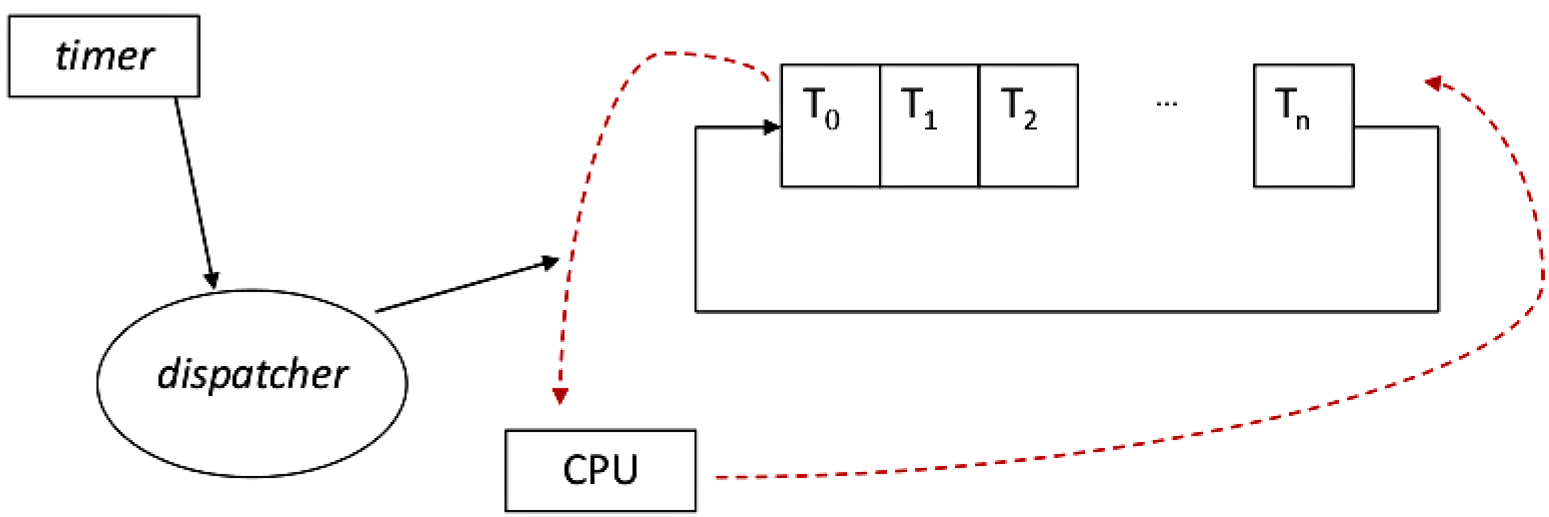


Context Switching (3)

- It is possible to store task context in any stack, even if several ISRs are nested
  - is just another set of data to put on top of the stack
  - stacks have limited capacity
- However CPUs have a restricted set of registers to store context, making impossible to store context of several ISRs …
  - or even one, it depends on the CPU

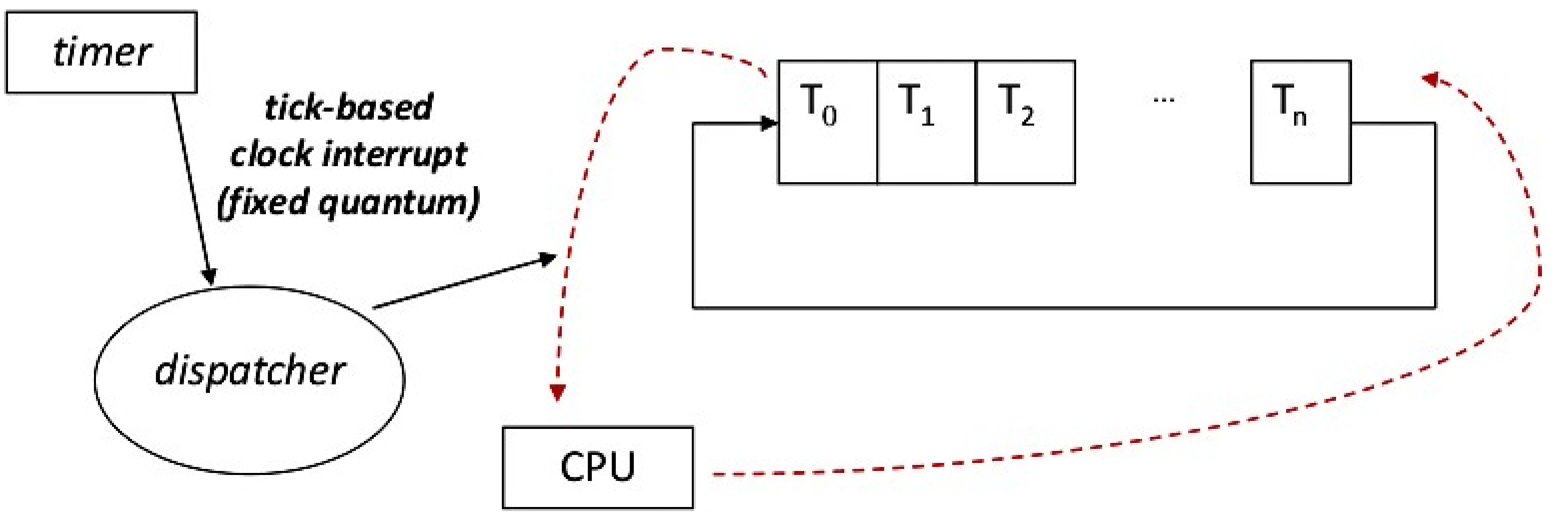
Clock Interrupts (1)

- In a scheduling mechanism, such as round-robin, after a time quantum or time-slice:
  - Dispatcher removes the task currently assigned to the CPU
  - and assigns the CPU to the first task in the ready tasks queue (switching task context)



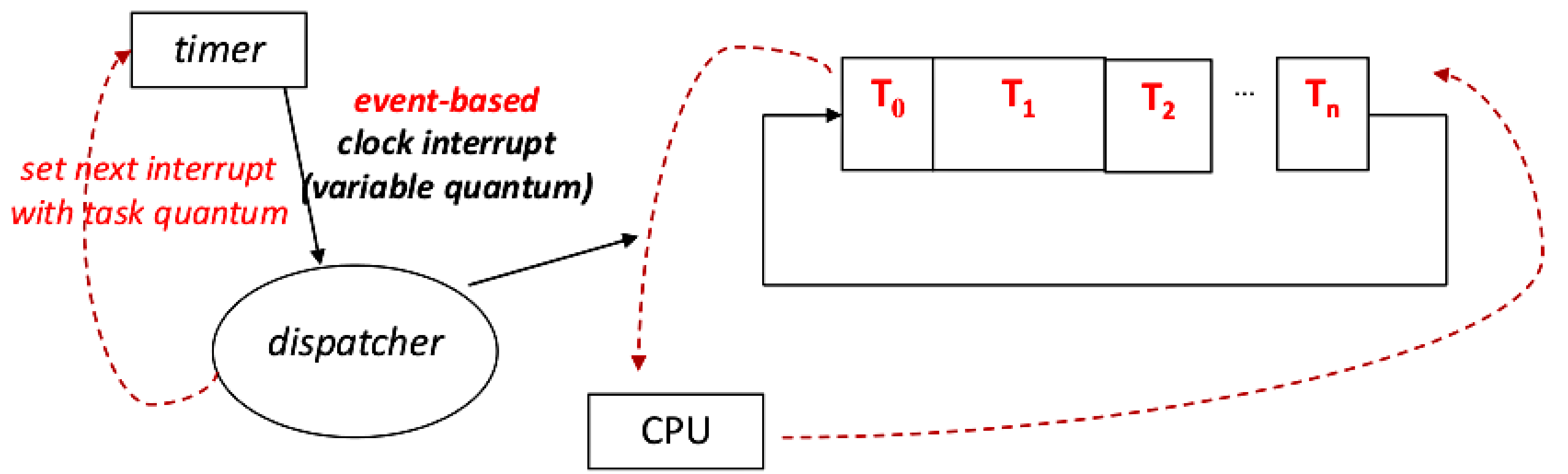
Clock Interrupts (2)

- The dispatcher can be an ISR, awaken by a periodic interrupt, generated by a timer, in each time quantum
- Assuming that the time-slice is fixed, the timer must generate an interrupt after a fixed number of cycles or ticks
- These interrupts are referred as tick-based clock interrupts



Clock Interrupts (3)

- If time-slice may be different for different tasks, the timer must be programmable
- When a new task is assigned to the CPU, the timer is programmed with the time-slice of this task
- These variable time interrupts are referred as event-based clock interrupts



Latency in Linux

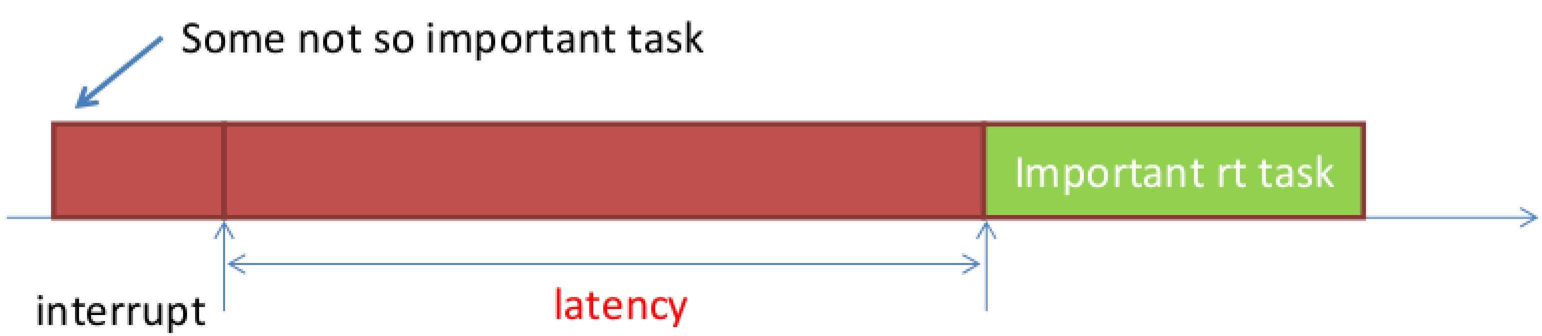
Latency in Kernel

Real time means external event should be handled within the bounded time

- Interrupt handler responds to the event and

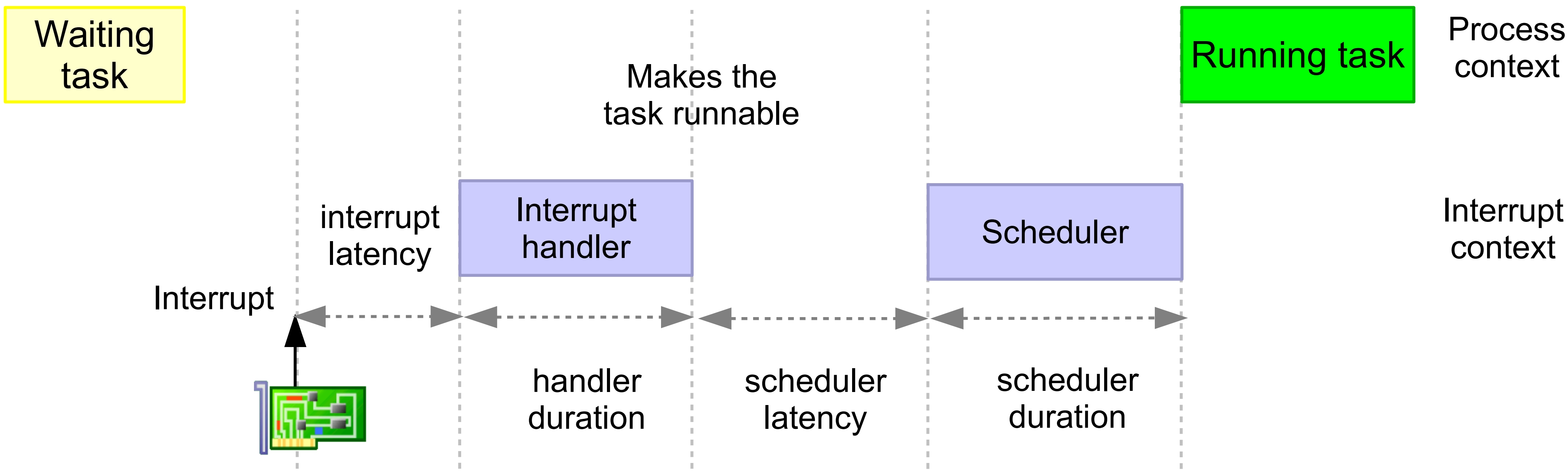
  inform user-space process

- Latency
  - Time taken from external interrupt till a user-space process to react to the interrupt



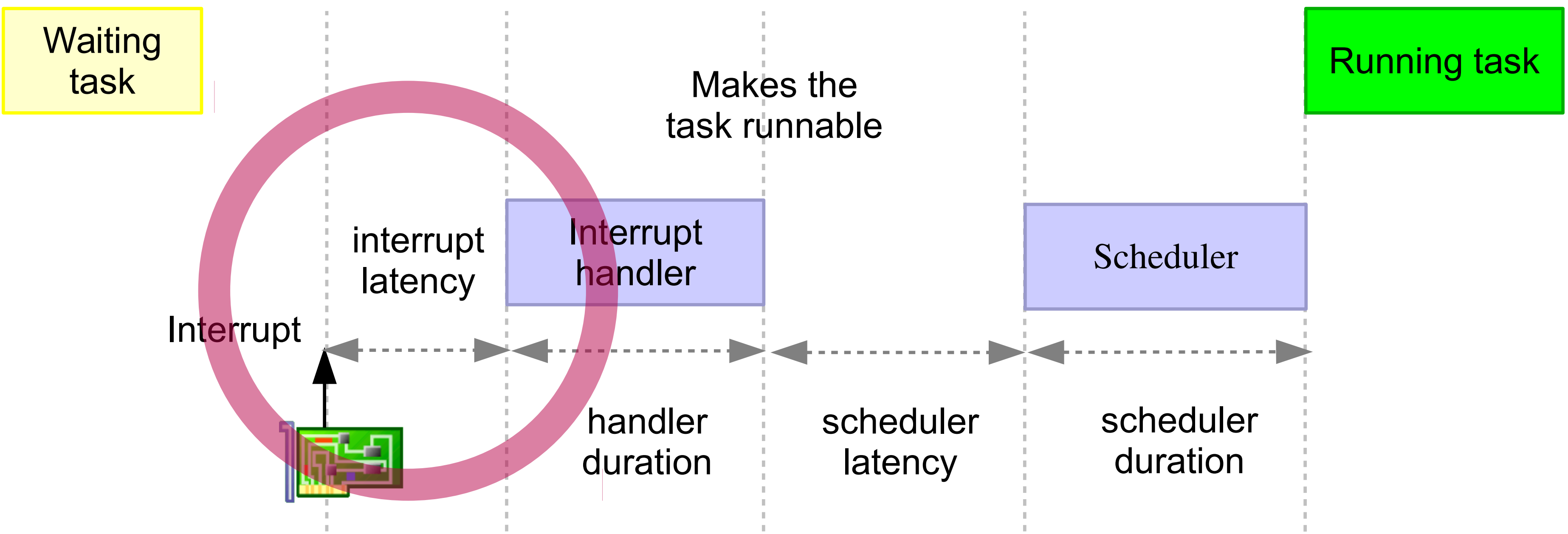
Linux kernel latency components

Typical scenario: process waiting for the completion of device I/O (signaled by an interrupt) to resume execution.



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

Interrupt latency



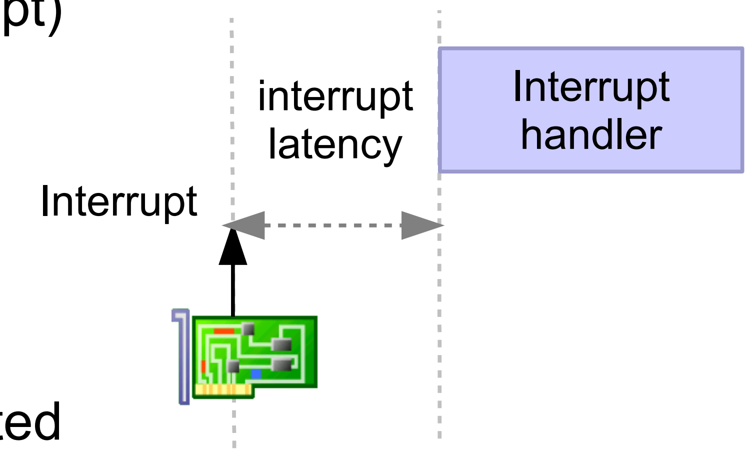
Time elapsed before executing the interrupt handler

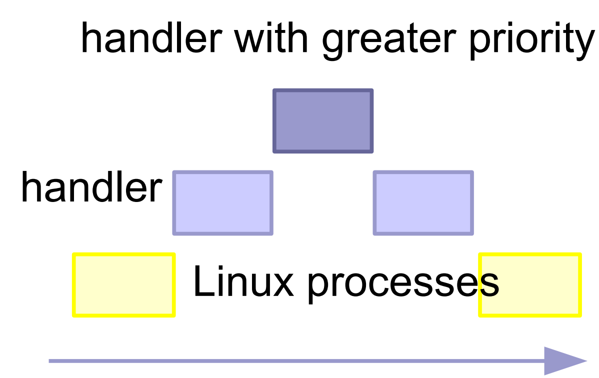
Sources of interrupt latency

Time between an event happens (raising an interrupt) and the handler is called. Sources of latency:

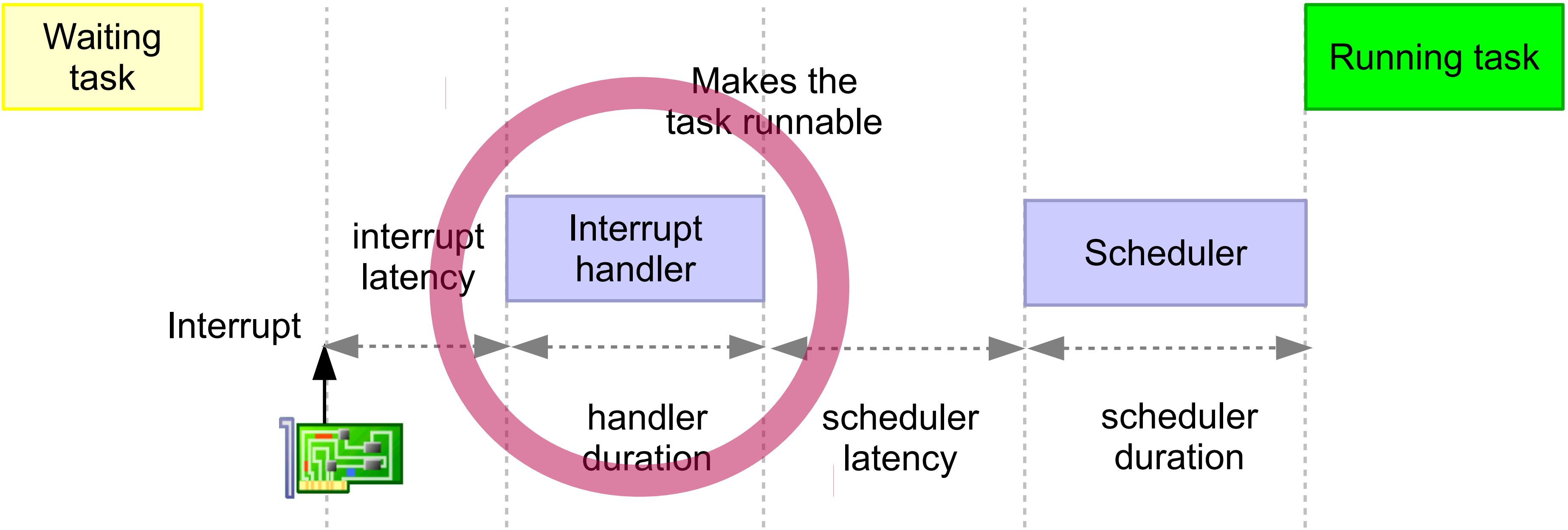
- Interrupts disabled by kernel code: spinlocks, evil drivers masking out interrupts.
- Other interrupts processed before:
  - Shared interrupt line: all handlers are executed (don't use shared IRQ lines for critical events)
  - Interrupts with higher priority can preempt lower priority ones (managed by the CPU, not by the scheduler). Not a problem in a correctly designed product: you use the top priority interrupt source.

Summary: the only real problem is disabled interrupts.

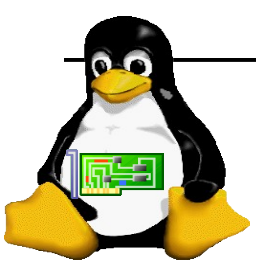




Interrupt handler duration



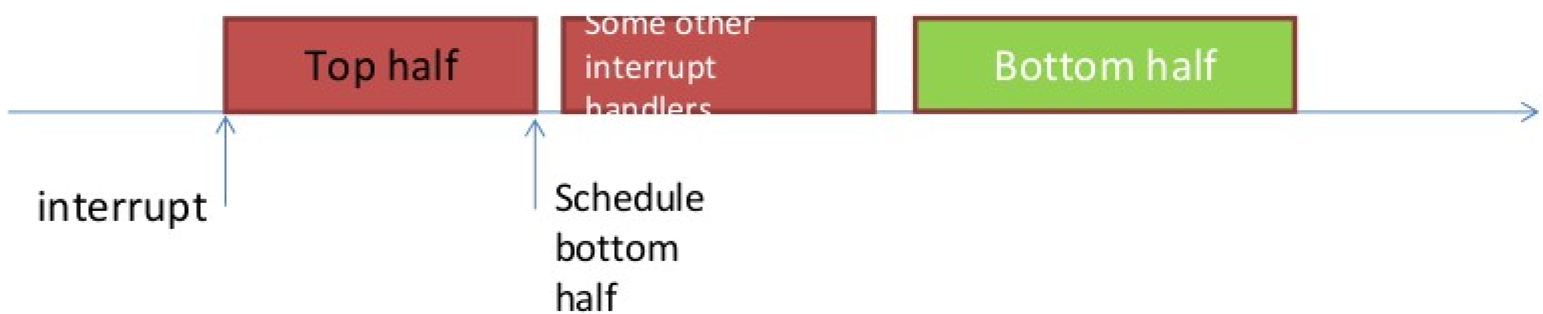
Time taken to execute the interrupt handler



Interrupt Handler

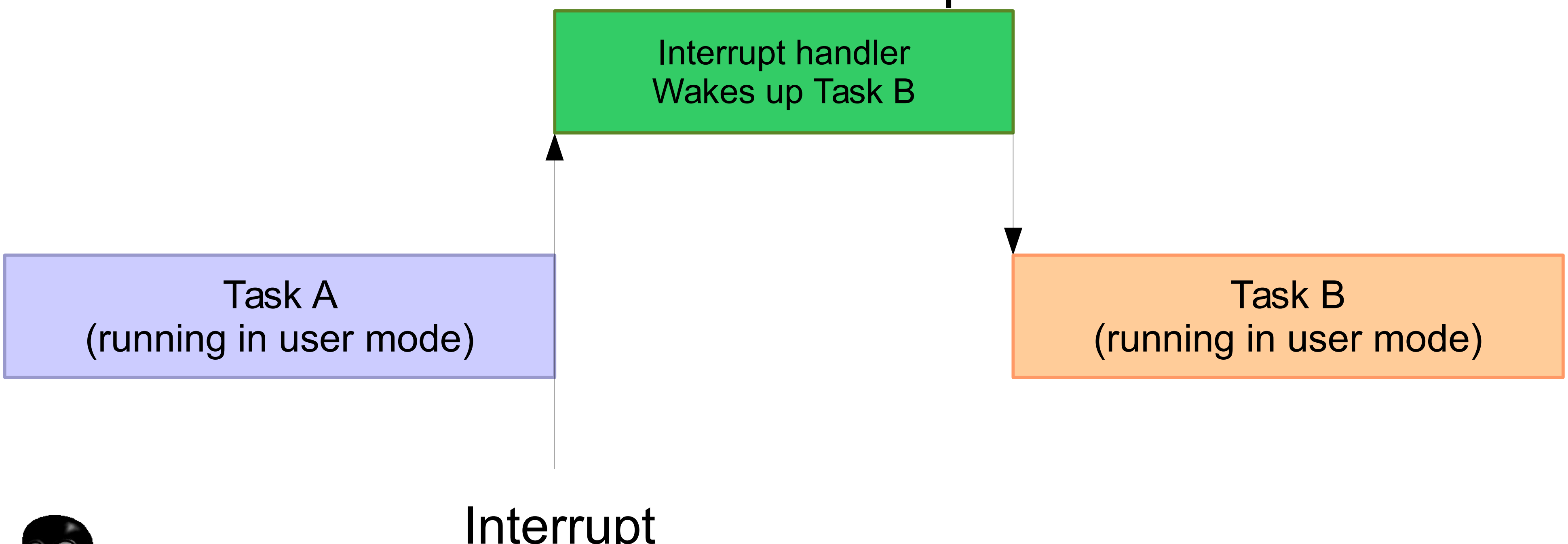
Interrupt handlers are split into two parts

- Top-half:
  - Process, as quickly as possible, the work during interrupt disabled, such as queue the information for the bottom-half
  - Schedule the bottom-half
- bottom-half
  - Process the required tasks from the triggered interrupt



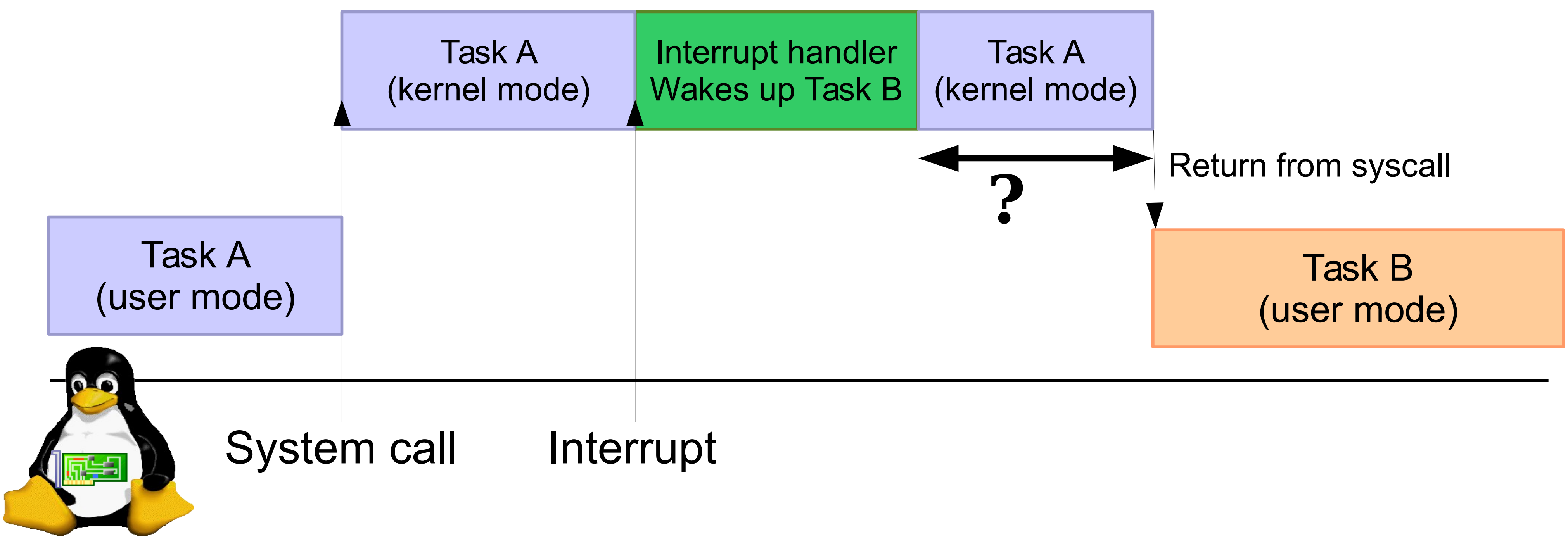
Kernel preemption (1)

- 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.

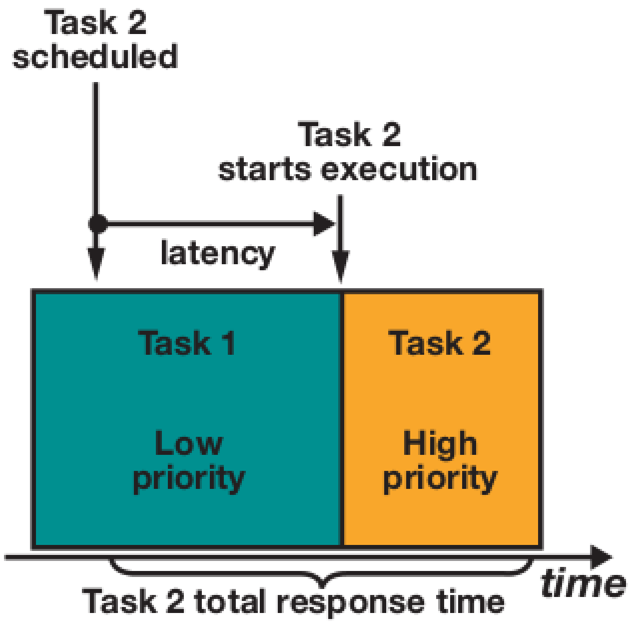


Kernel preemption (2)

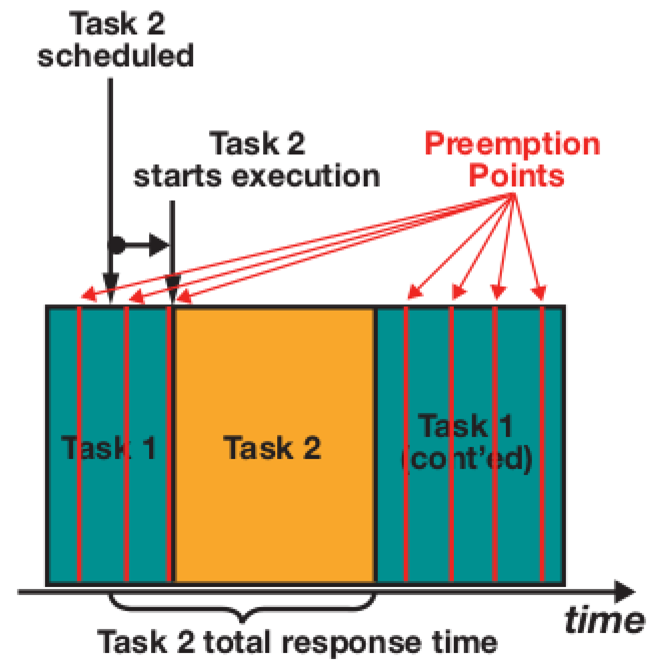
- However, when the interrupt comes while the task is executing a system call, this system call has to finish before another task can be scheduled.
- By *default,* the Linux kernel does not do *kernel preemption.*
- This means that the time before which the scheduler will be called to schedule another task is unbounded.



Preemptive Kernel



non-preemptive system

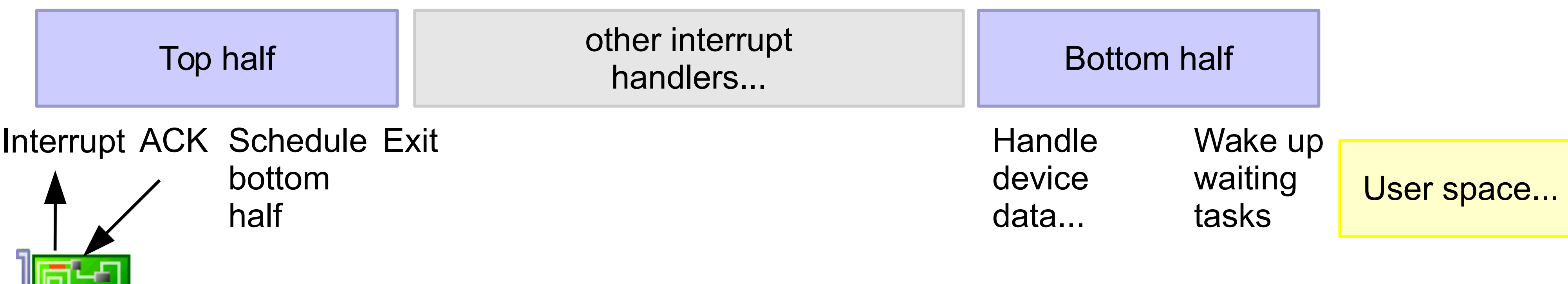


preemptive system

A concept linked to that of real time is preemption: the ability of a system to interrupt tasks at many "preemption points".The longer the non-interruptible program units are, the longer is the waiting time ('latency') of a higher priority task before it can be started or resumed. GNU/Linux is "user-space preemptible": it allows user tasks to be interrupted at any point. The job of real-time extensions is to make system calls preemptible as well.

Interrupt handler implementation

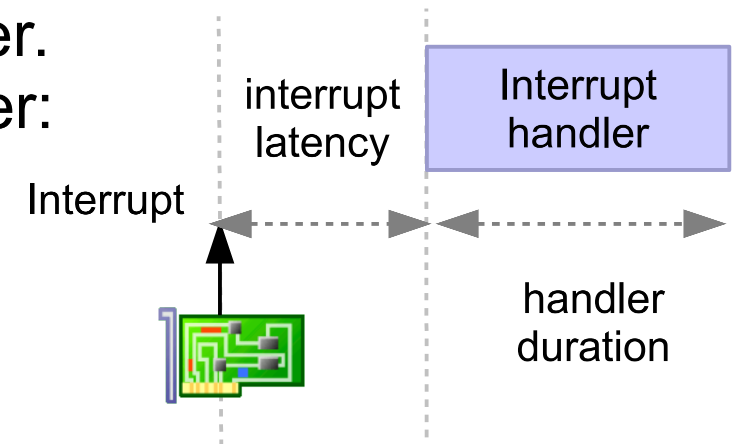
- Run with interrupts disabled (at least on the current line).
- Hence, they need to complete and restore interrupts as soon as possible.
- To satisfy this requirement, interrupt handlers are often implemented in 2 parts: top half and bottom half (soft IRQ)



Soft IRQs are handled in interrupt context (not by the scheduler)

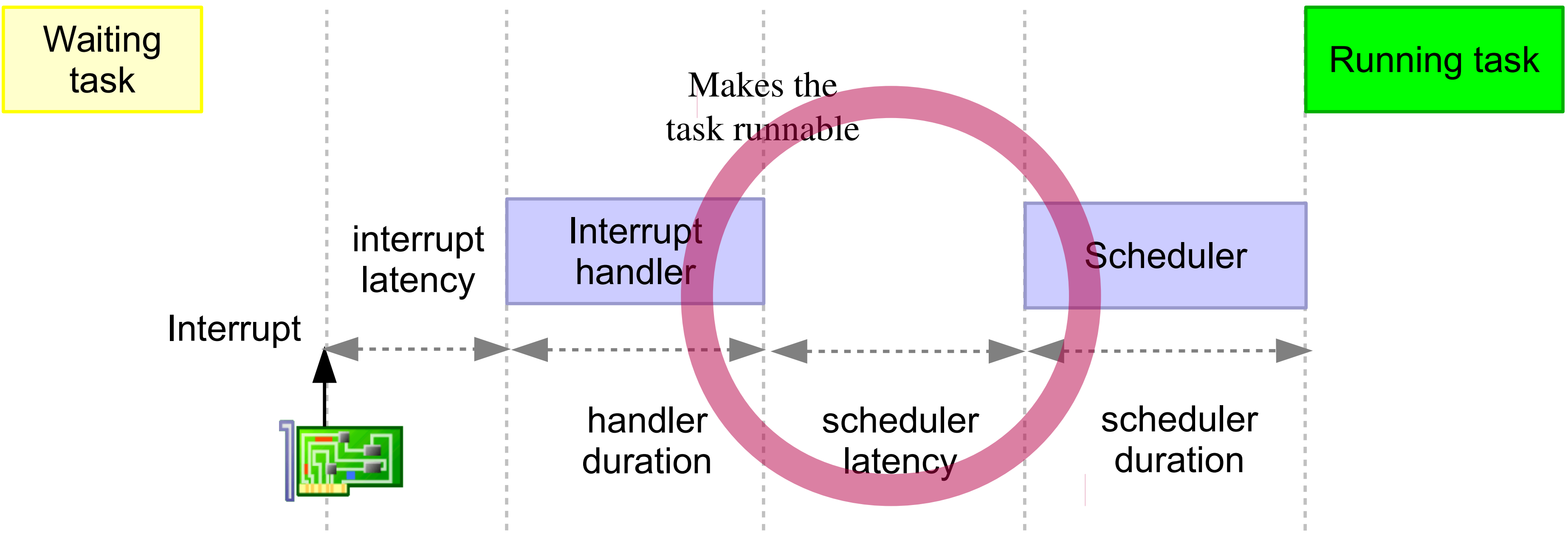
Sources of interrupt handler execution time

Time taken to execute your interrupt handler.
Causes which can make this duration longer:

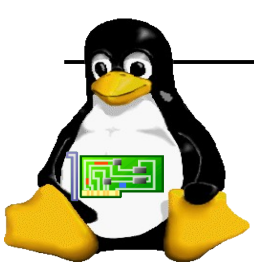


- Preemption by other interrupts with a greater priority. Again, not a problem if priorities are configured properly.
- Interrupt handler with a softirq (“bottom half”) component: run after all interrupts are serviced.
  => For critical processing, better not have a softirq
  (may require rewriting the driver if reusing an existing one).

Scheduler latency



Time elapsed before executing the scheduler



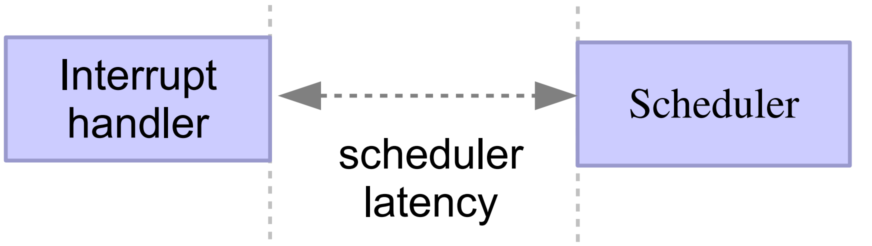
When is the scheduler run next?

- In the general case (still after a task is woken up)
- Upon return from interrupt context (our particular case), unless the current process is running a system call (system calls are not preemptible by default).
- After returning from system calls.
- When the current process runs an action which sleeps / waits (voluntarily or not, in kernel or user mode), or explicitly calls the scheduler.

Sources of scheduler latency

Possible sources of latency:

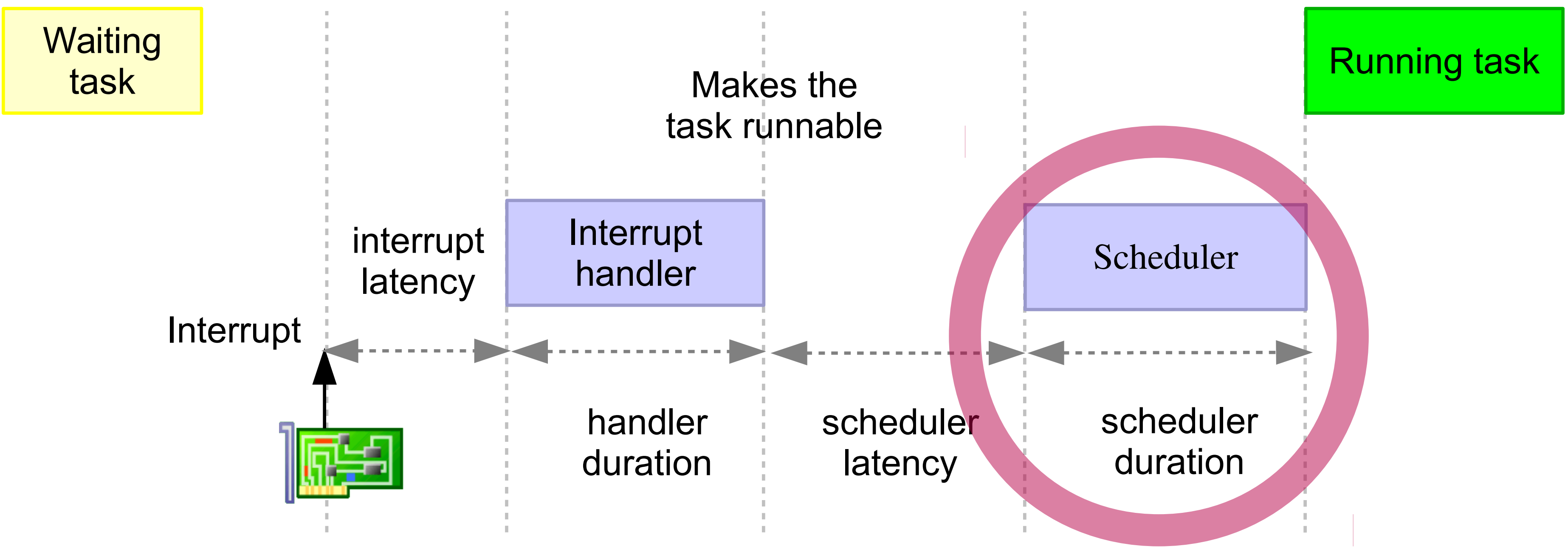
- Other interrupts coming in (even with a lower priority).



Back to our case

- We are returning from interrupt context. The scheduler is run immediately if the current task was running in userspace. Otherwise, we have to wait for the end of the current system call.

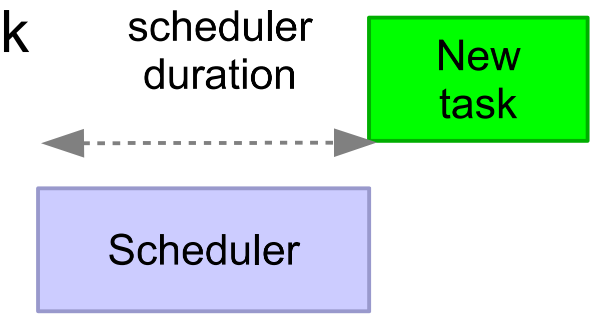
Scheduler duration



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

Sources of scheduler execution time

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

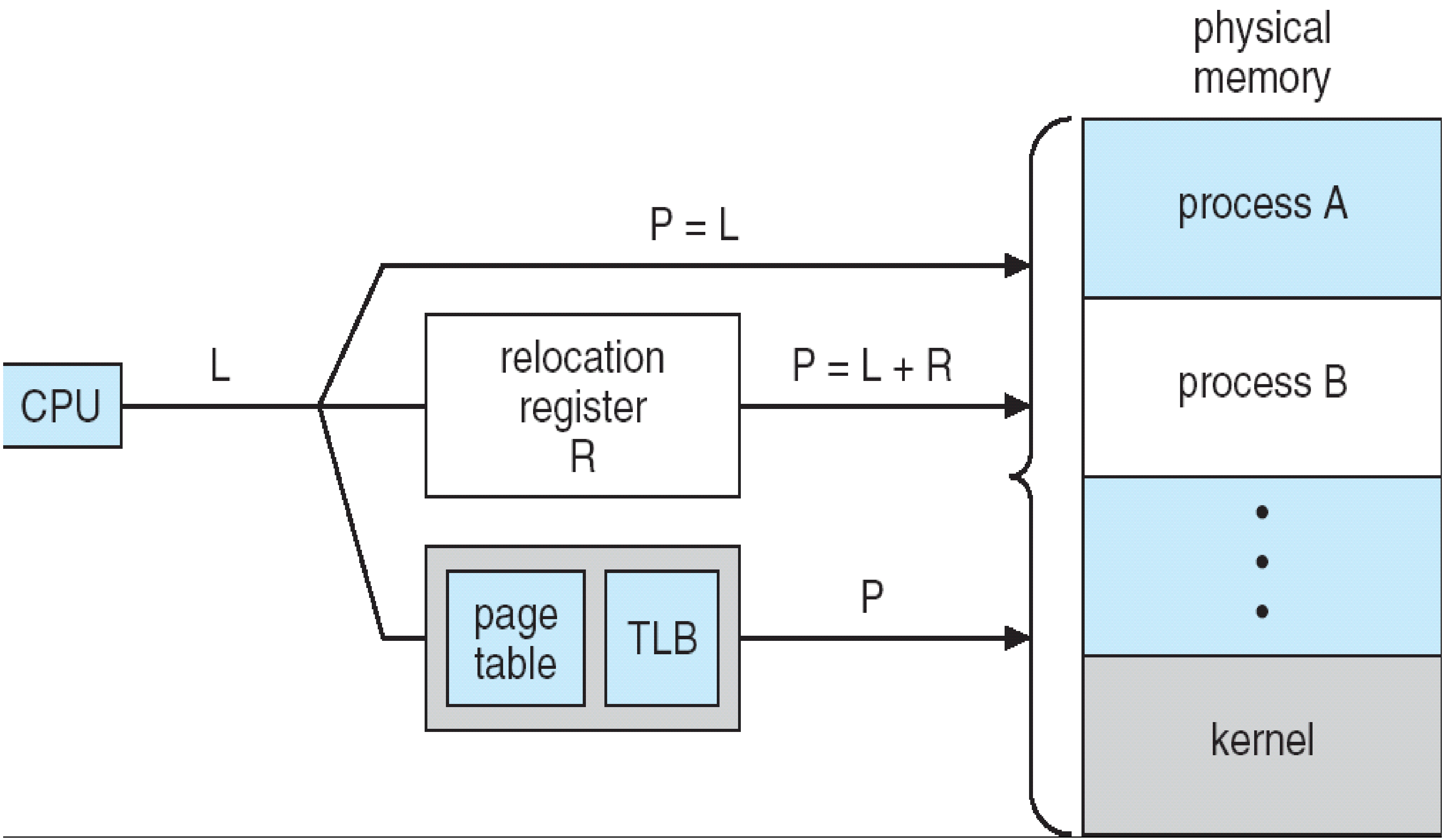
Other sources of latency (1)

Demand paging

- When it starts a process, the Linux kernel first sets up the virtual address space of the program (code, stack and data).
- However, to improve performance and reduce memory consumption, Linux loads the corresponding pages in RAM only when they are needed. When a part of the address space is accessed (like a particular part of the code) and is not in RAM yet, a page fault is raised by the MMU. This triggers the loading of the page in RAM.

This usually involves reading from disk: unexpected latencies!

Address Translation



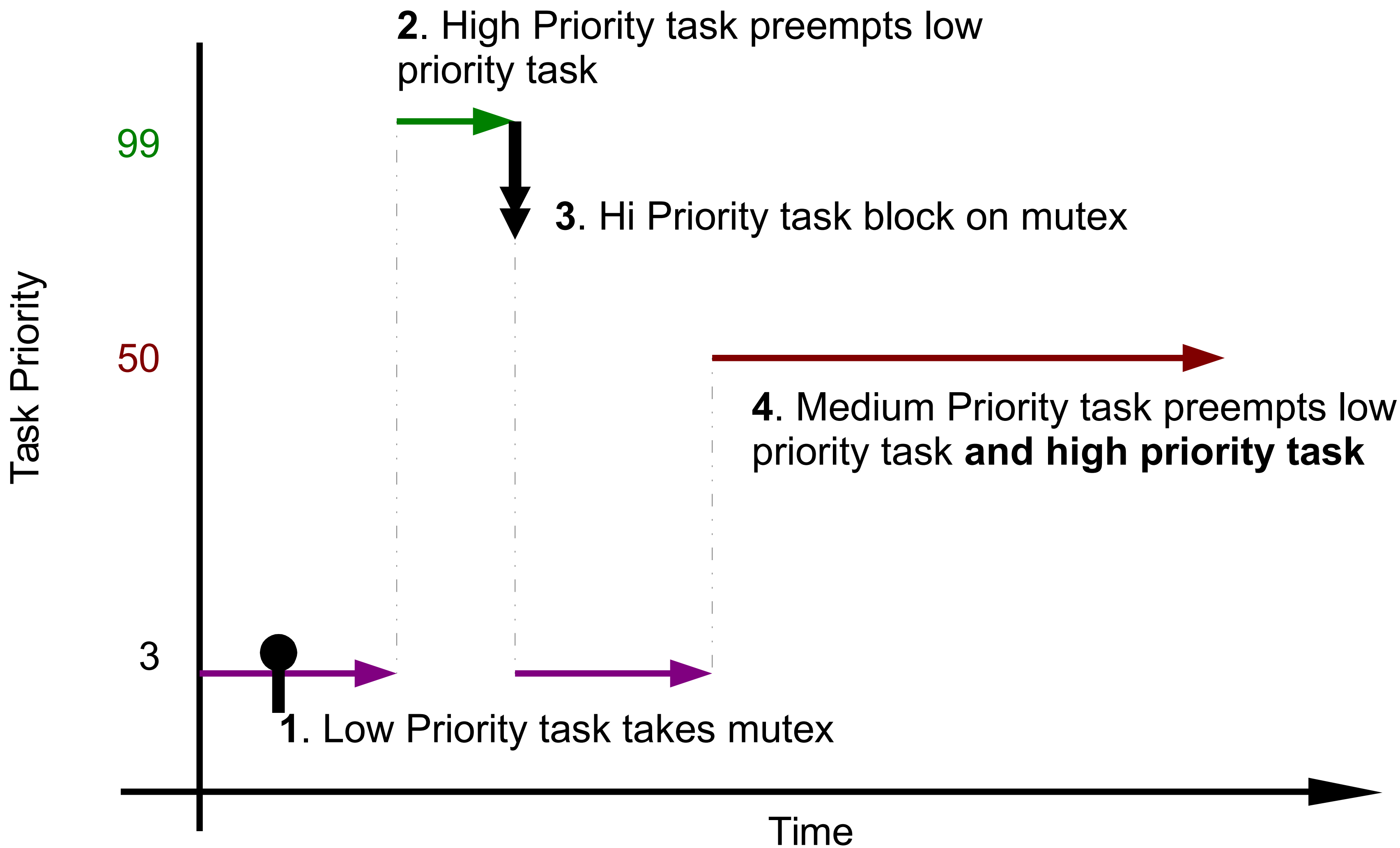
Other sources of latency (2)

- Linux is highly based on virtual memory provided by 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.
- Memory allocation
  The Linux allocator is fast,
  but does not guarantee a maximum allocation time.

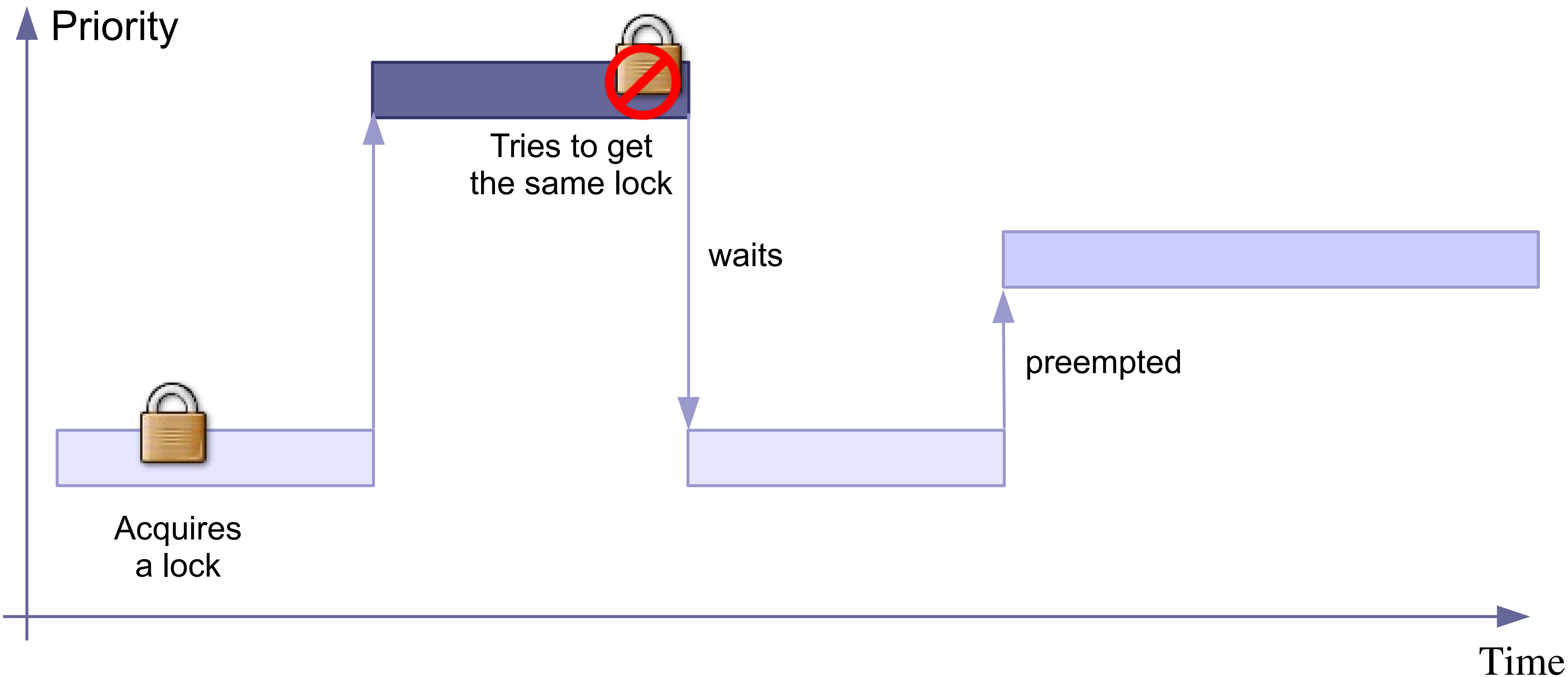
Other sources of latency (3)

- System calls
  Similarly, no guaranteed response time.
- Device drivers
  System calls handlers or interrupt handlers usually not implemented with care for real-time requirements.
  Expect uncertainties unless you implemented all kernel support code by yourself.

Issue: priority inversion

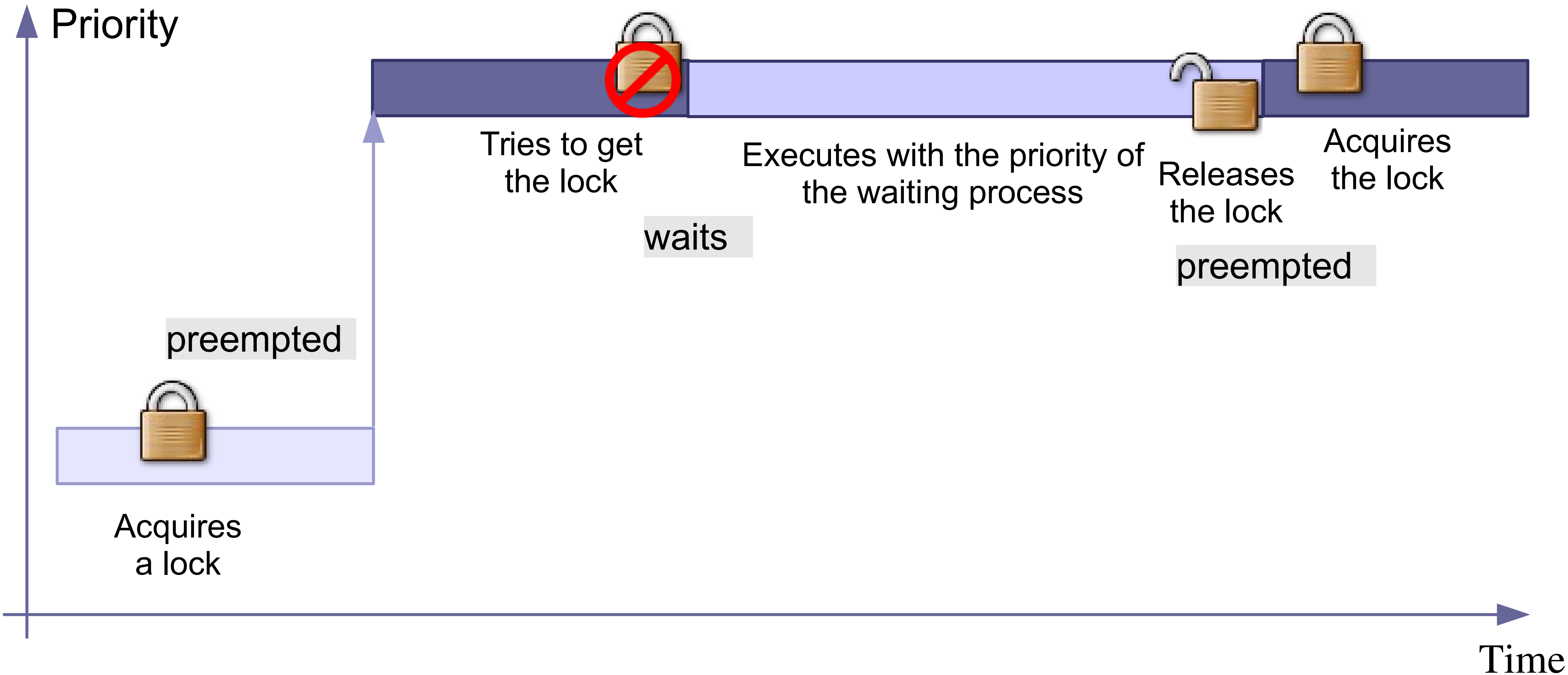


Issue: priority inversion



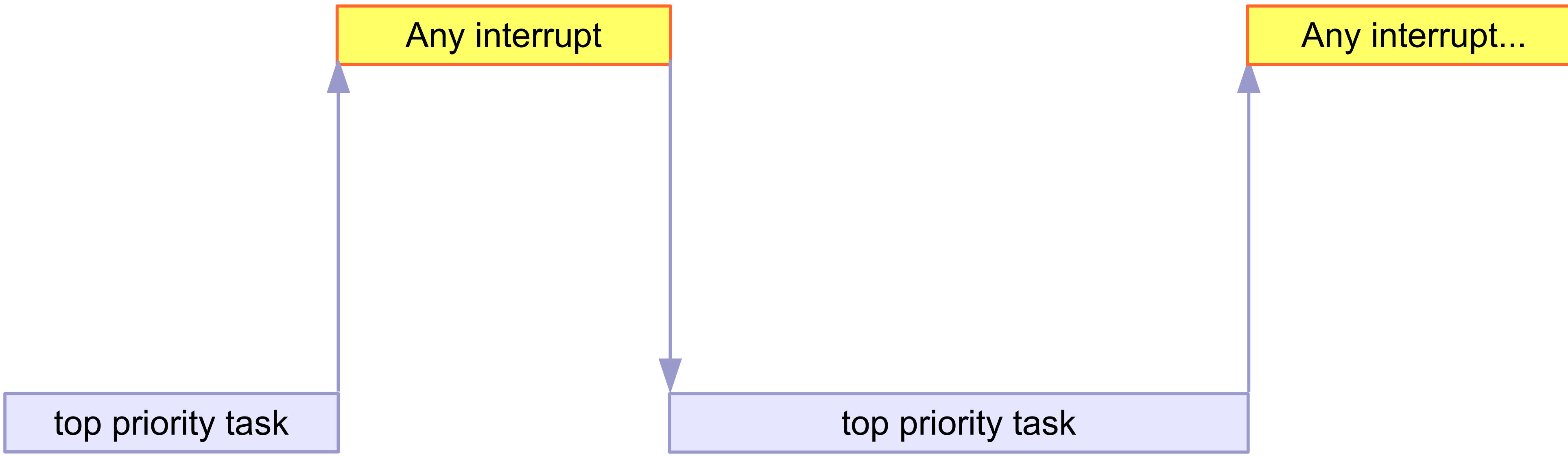
Issue: a process with more priority can preempt a process holding the lock. The top priority process could wait for a very long time.
This happened with standard Linux before version 2.6.18.

Solution: priority inheritance



Solution: *priority inheritance*. The process holding the lock inherits the priority of the process waiting for the lock with the greatest priority.

Issue: interrupt inversion



<u>Issue</u>: even your top priority task can be “preempted” by any interrupt handler, even for interrupts feeding tasks with lower priority.

<u>Solution</u>: threaded interrupts (scheduled for later execution).

Linux features for real-time

Linux 2.6 improvements (1)

- Scheduler: starting from 2.6.23, Linux implements CFS.
  - complexity of **O(log N)**, where N is the number of tasks in the runqueue.
  - Choosing a task can be done in constant time **O(1)**, but reinserting a task after it has run requires **O(log N)** operations, because the runqueue is implemented as a red-black tree.
- May be built with no virtual memory support (on supported platforms). Benefits: reduced context switching time (address space shared by all processes), better performance (CPU cache still valid after switching).
- Full POSIX real-time API support.
  - the standard Unix API to implement real-time applications.

POSIX 1003.1b Real-time extensions

- Priority Scheduling
- Real-Time Signals
- Clocks and Timers
- Semaphores
- Message Passing
- Shared Memory
- Asynchronous and Synchronous I/O
- Memory Locking

Linux 2.6 improvements (2)

- Threaded Interrupt
  Interrupt can be executed by a kernel thread, we can assign a priority to each thread
- RT-Mutexes
  RT-mutexes extend the semantics of simple mutexes by the priority inheritance protocol.
- BKL-free
  Since 2.6.37 kernel can be built completely without the use of Big Kernel Lock.

Linux 2.6 improvements (3)

- Increased determinism with improved POSIX RT API support.
- By default, system calls still not preemptible.
- New `CONFIG_PREEMPT` option
  making most of the kernel code preemptible.

However, even `CONFIG_PREEMPT` was not enough for audio users (need guaranteed latency in the milliseconds range), and of course for real-time users with stronger requirements.

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 (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).
- 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).
- Typically for desktop or embedded systems with latency requirements in the milliseconds range.
- Still a relatively minor impact on throughput.

Priority inheritance support

Available since Linux 2.6.18

- Implemented in kernel space locking
- Available in user space locking too,
  through fast user-space mutexes ("futex").
  Priority inheritance needs to be explicited for each mutex though.
- See the kernel documentation for details:
  `Documentation/pi-futex.txt`
  `Documentation/rt-mutex-design.txt`

High-resolution timers

- Make it possible for POSIX timers and `nanosleep()` to be as accurate as allowed by the hardware (typically 1 us).
- Together with tickless support, allow for “on-demand” timer interrupts.

Available in the vanilla kernel since 2.6.21.

Linux rt-preempt

Linux real-time preemption

http://www.kernel.org/pub/linux/kernel/projects/rt/

- led by kernel developers including Ingo Molnar, Thomas Gleixner, and Steven Rostedt
  - Large testing efforts at RedHat, IBM, OSADL, Linutronix
- Goal is to improve real time performance
- Configurable in the `Processor type and features` (`x86`), `Kernel Features` (`arm`) or `Platform options` (`ppc`)...

Preemption Mode	
○ No Forced Preemption (Server)	PREEMPT_NONE
○ Voluntary Kernel Preemption (Desktop)	PREEMPT_VOLUNTARY
○ Preemptible Kernel (Low-Latency Desktop)	PREEMPT_DESKTOP
◉ Complete Preemption (Real-Time)	PREEMPT_RT
Thread Softirqs	PREEMPT_SOFTIRQS
Thread Hardirqs	PREEMPT_HARDIRQS

Wrong ideas about real-time preemption

- *Myth (1): It will improve throughput and overall performance*
  Wrong: it will degrade overall performance.
- *Myth (2): It will reduce latency*
  Often wrong. The maximum latency will be reduced.

The primary goal is to make the system predictable and deterministic.

Current issues and limitations

- Linux RT patches not mainstream yet.
  Facing resistance to retain the general purpose nature of Linux.
- Though they are preemptible, kernel services (such as memory allocation) do not have a guaranteed latency yet! However, not really a problem: real-time applications should allocate all their resources ahead of time.
- Kernel drivers are not developed for real-time constraints yet (e.g. USB or PCI bus stacks)
- Binary-only drivers have to be recompiled for RT preempt
- Memory management (SLOB/SLUB allocator) still runs with preemption disabled for long periods of time
- Lock around per-CPU data accesses, eliminating the preemption problem but wrecking scalability

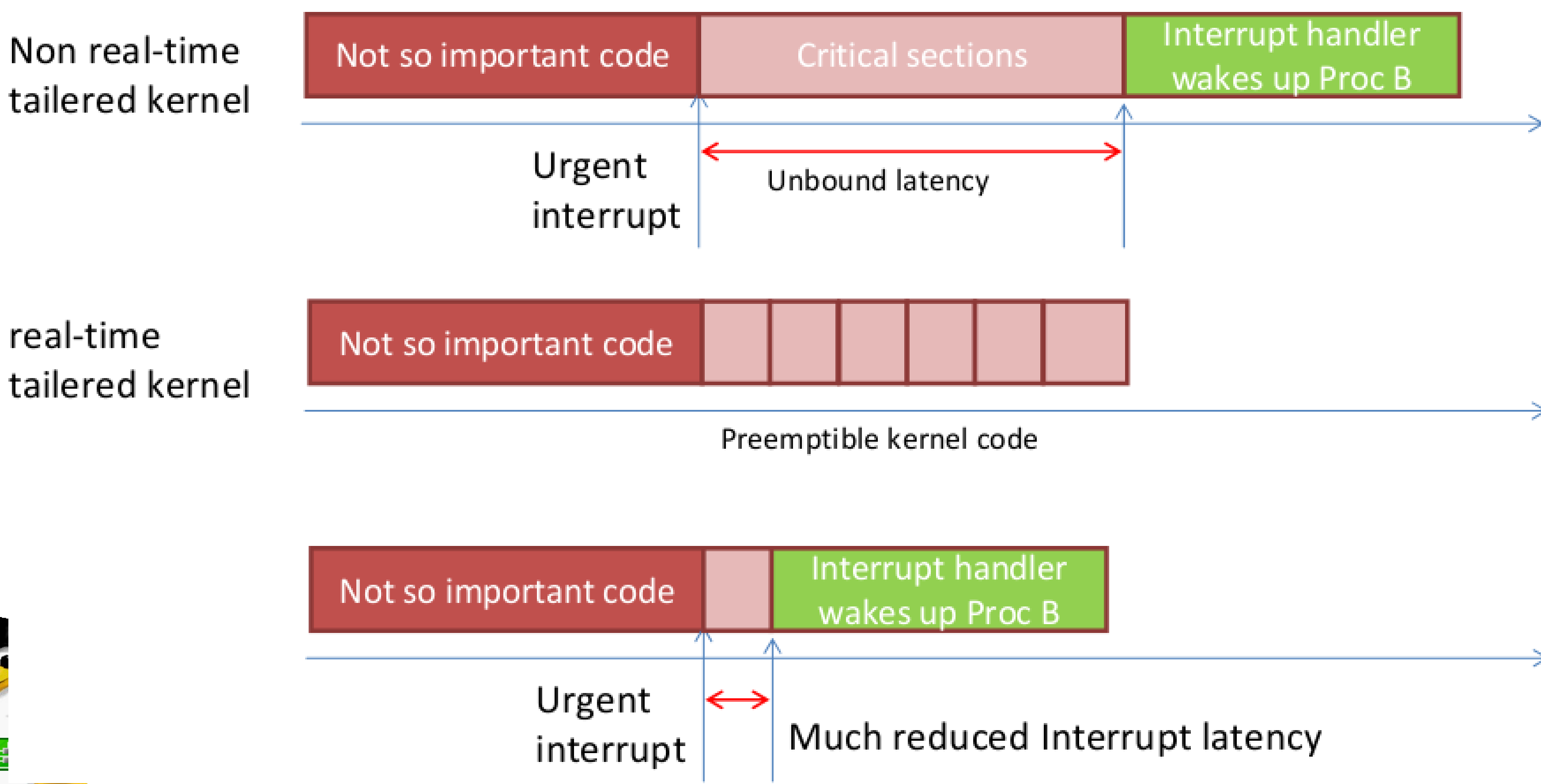
PREEMPT_RT: complete RT preemption

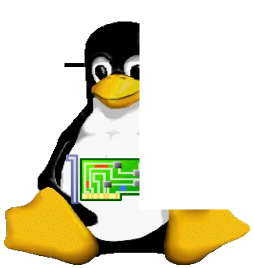
Replace non-preemptible constructs with preemptible ones

- Make OS preemptible as much as possible
  - except preempt_disable and interrupt disable
- Make Threaded (schedulable) IRQs
  - so that it can be scheduled
- spinlocks converted to mutexes (a.k.a. sleeping spinlocks)
  - Not disabling interrupt and allows preemption
  - Works well with threaded interrupts

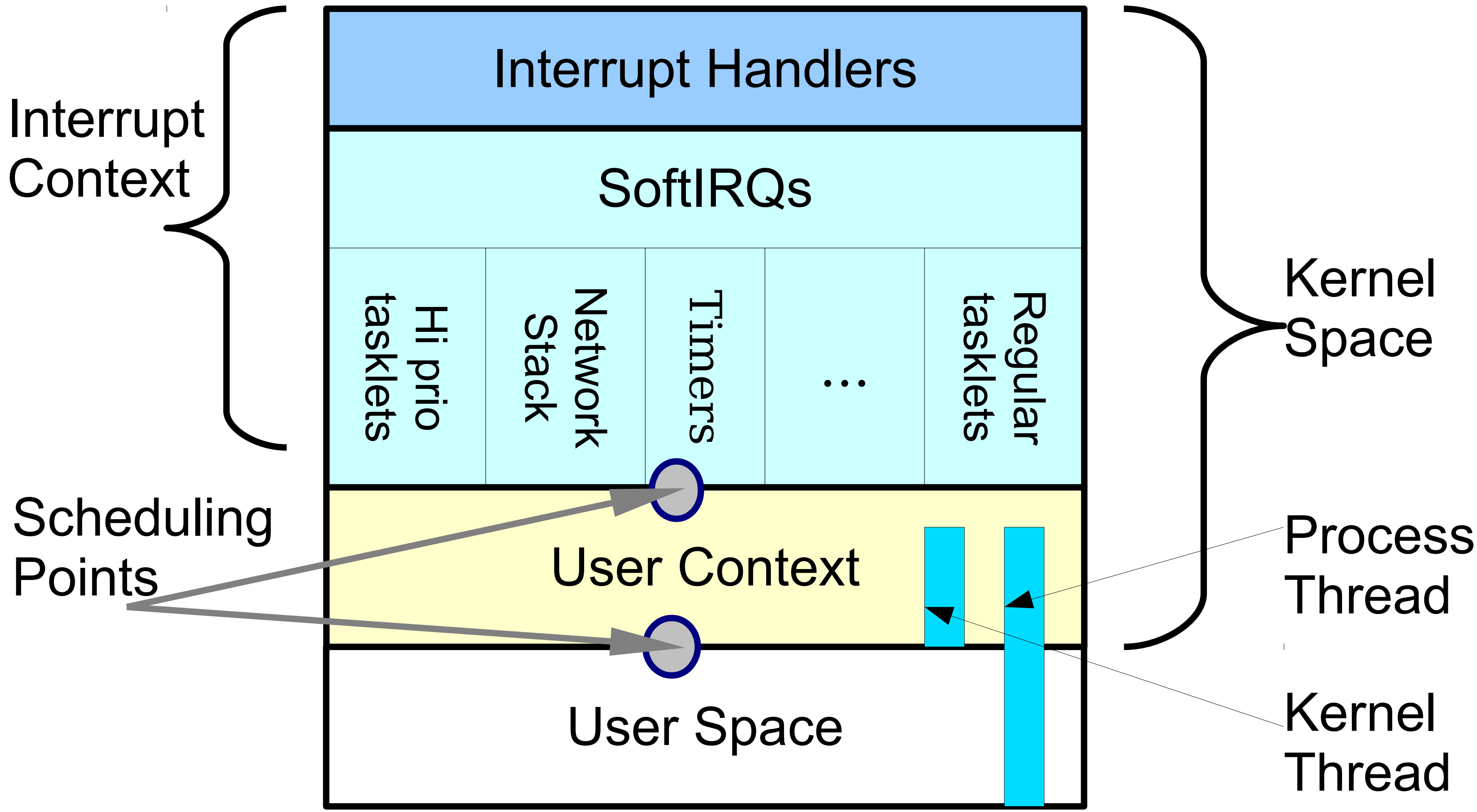
Toward complete RT preemption

- Most important aspects of Real-time
  - Controlling latency by allowing kernel to be preemptible everywhere

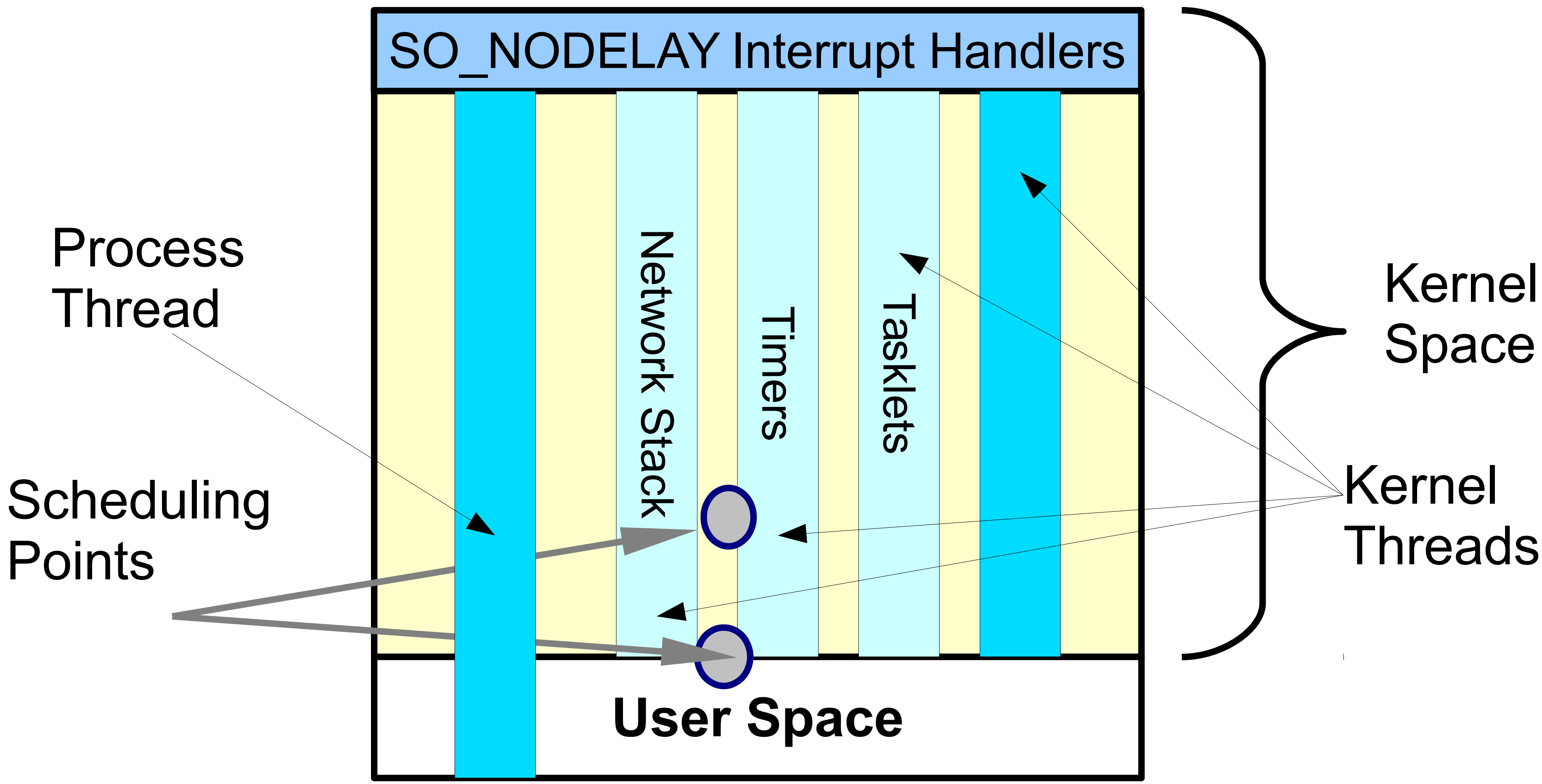




original Linux Kernel



PREEMPT_RT


SO_NODELAY Interrupt Handlers
Process Thread
Network Stack
Timers
Tasklets
Kernel Space
Kernel Threads
Scheduling Points
User Space

Interface changed by PREEMPT_RT

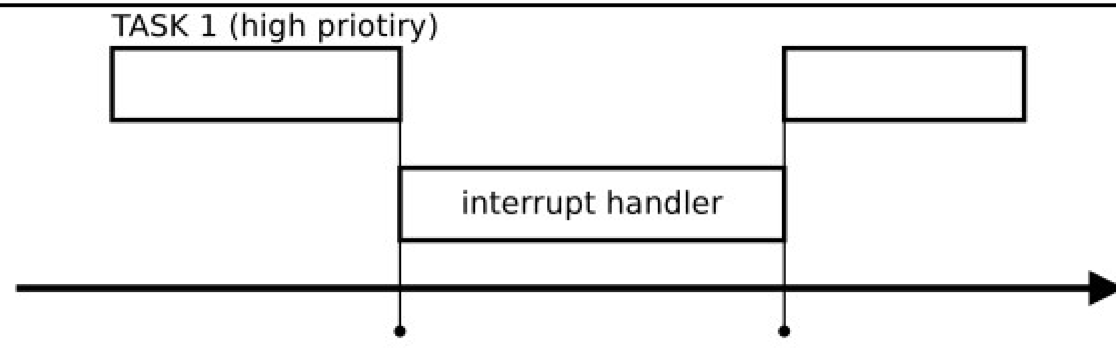
- spinlocks and local_irq_save() no longer disable hardware interrupts.
- spinlocks no longer disable preemption.
- Raw_ variants for spinlocks and local_irq_save() preserve original meaning for SO_NODELAY interrupts.
- Semaphores and spinlocks employ priority inheritance

Threaded Interrupts

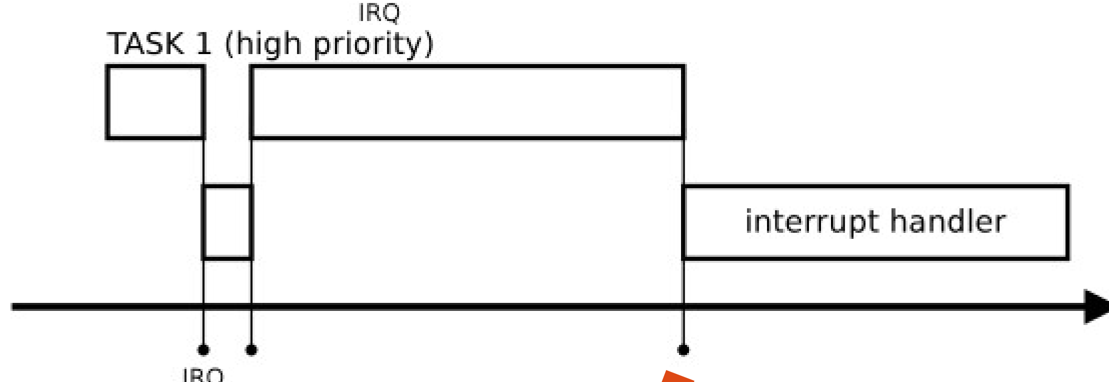
- Interrupt handlers run in normal kernel threads
  - Priorities can be configured
- Main interrupt handler
  - Do minimal work and wake-up the corresponding thread
- Thread interrupts allows to use sleeping spinlocks
- in PREEMPT_RT, all interrupt handlers are switched to threaded interrupt
  - drivers in mainline get gradually converted to the new threaded interrupt API that has been merged in Linux 2.6.30.

Threaded Interrupts

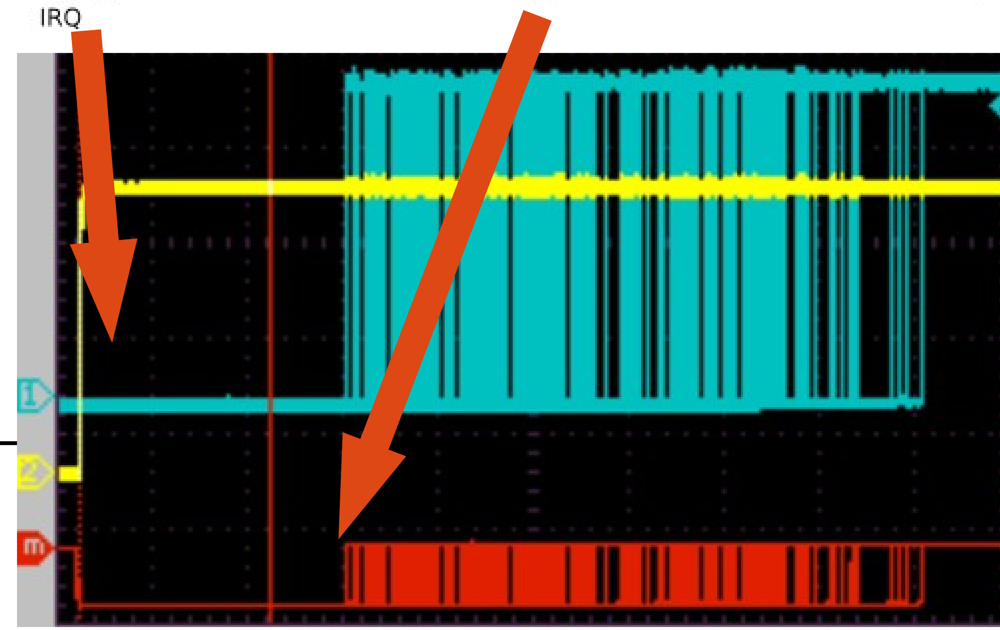
- The vanilla kernel



- Interrupts as threads



- Real world behavior



Threaded Interrupt handler

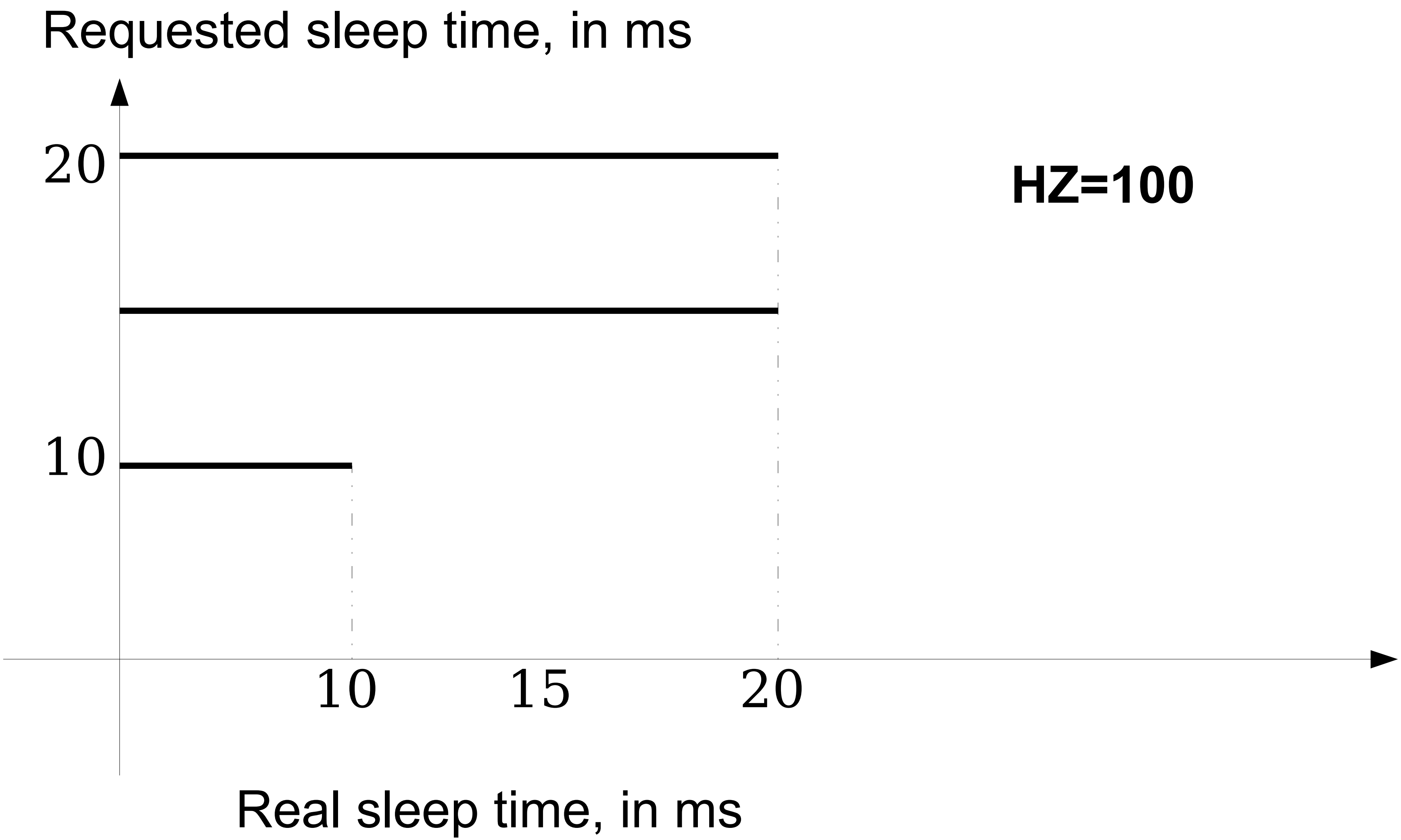
- Almost all kernel space is now preemptible
  - An interrupt can occur at any time
  - The woken up processes by interrupt can start immediately
- Treaded IRQs: Kernel thread per ISR
- Priority must be set
  - Interrupt handler threads
  - Softirq threads
  - Other kernel threads
  - Real time application processes/threads

Timer Frequency

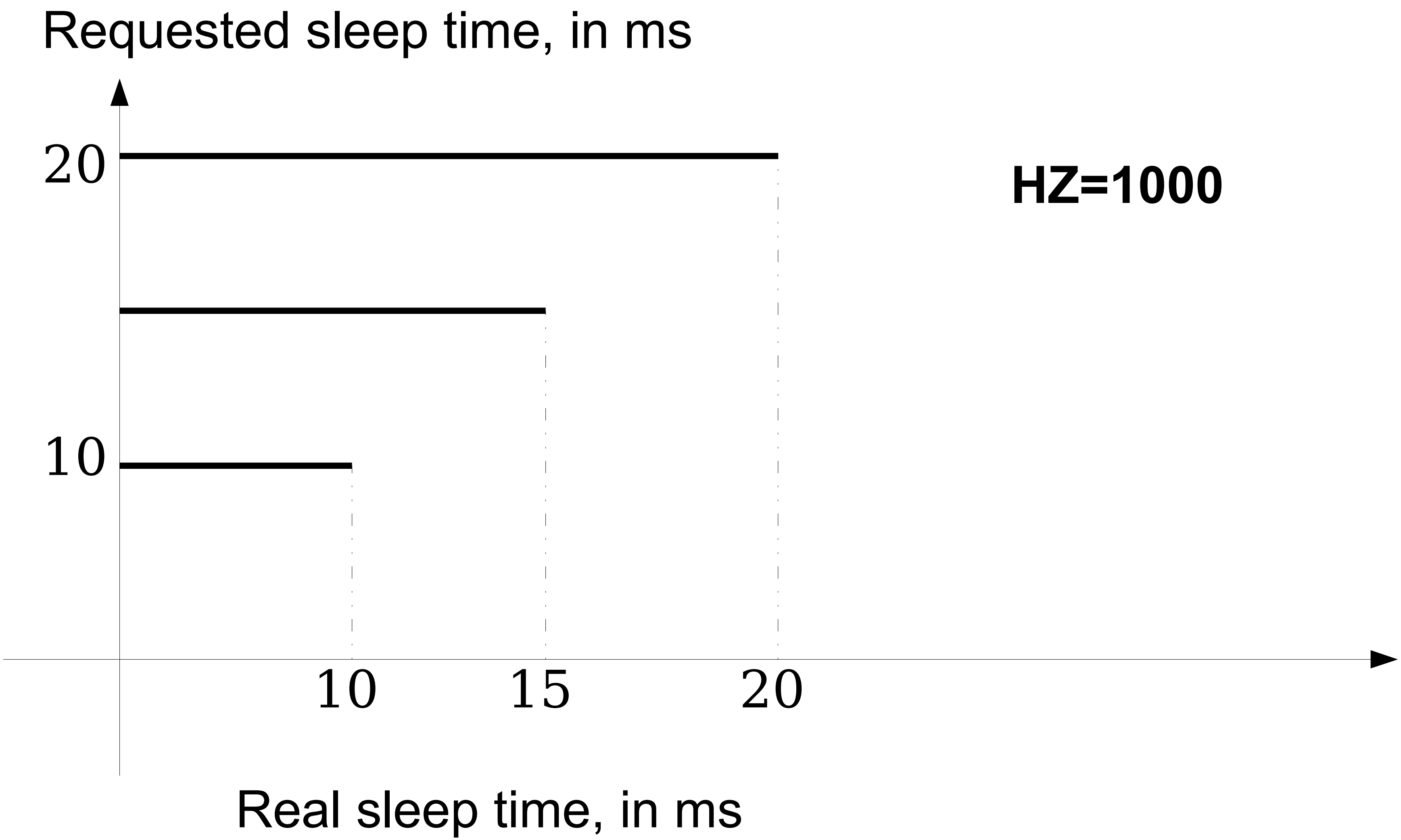
Linux timer interrupts are raised every HZ of second.

- Resolution of the system timer (HZ)
  - 100 HZ to 1000 HZ depends on arch and configuration
  - Resolution of 10 ms to 1 ms
  - Adds overheads → Compromise between system responsiveness and global throughput.

Effect of Timer Frequency



Effect of Timer Frequency



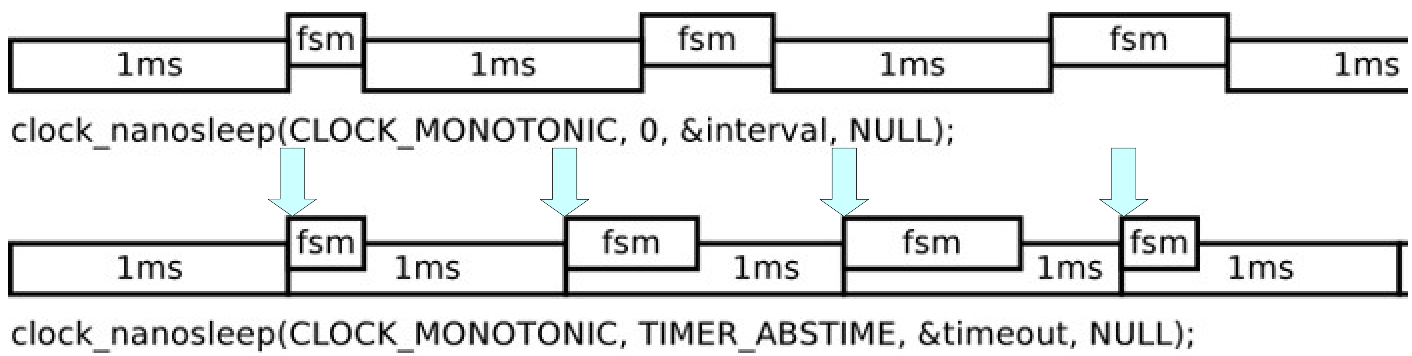
High Resolution Timers

Use non-RTC interrupt timer sources to deliver timer interrupt between ticks.

- Allows POSIX timers and nanosleep() to be as accurate as the hardware allows
- This feature is transparent.
  - When enabled it makes these timers much more accurate than the current HZ resolution.
  - Around 1usec on typical hardware.

Use POSIX Timer for Periodical Task

```
const int NSEC_IN_SEC = 1000000000l, INTERVAL = 1000000l;
struct timespec timeout;
clock_gettime(CLOCK_MONOTONIC, &timeout);
while (1) {
    do_some_work(&some_data);
    timeout.tv_nsec += INTERVAL;
    if (timeout.tv_nsec >= NSEC_IN_SEC) {
        timeout.tv_nsec -= NSEC_IN_SEC;
        timeout.tv_sec++;
    }
    clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &timeout, NULL);
}
```



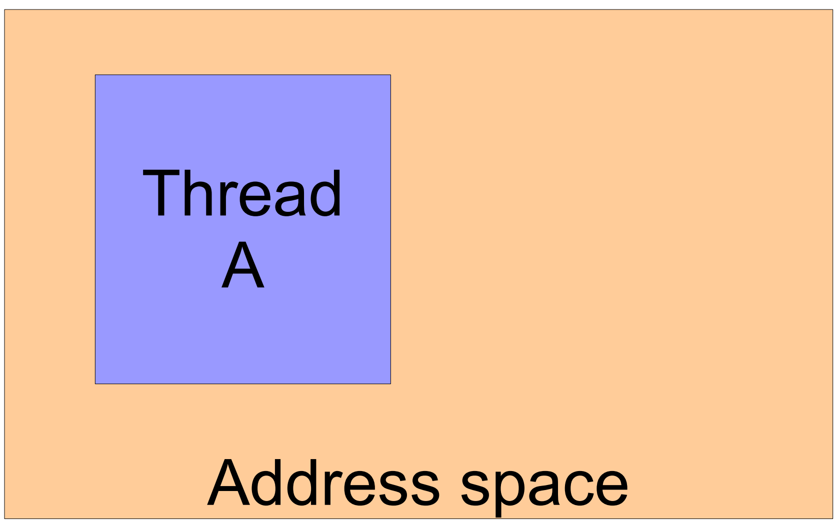
Implementing real-time constraints

Process vs. Thread

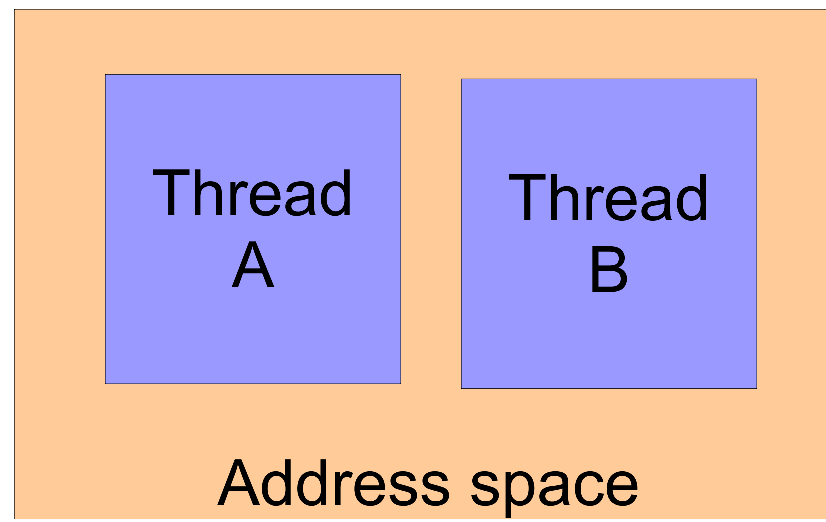
- In Unix, a process is created using `fork()` and is composed of
  - An address space, which contains the program code, data, stack, shared libraries, etc.
  - One thread, that starts executing the main() function.
  - Upon creation, a process contains one thread
- Additional threads can be created 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 vs. Thread: Kernel point of view

- 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 the termination of a thread
  - ```
    pthread_join(pthread_t *thread, void **value_ptr);
    ```
    ```

Scheduler policy

- SCHED_FIFO: the process runs until completion unless it is blocked by an I/O, voluntarily relinquishes the CPU, or is preempted by a higher priority process.
- SCHED_RR: the processes are scheduled in a Round Robin way. Each process is run until it exhausts a max time quantum. Then other processes with the same priority are run, and so and so...
- SCHED_BATCH: Does not preempt nearly as often as regular tasks would, thereby allowing tasks to run longer and make better use of caches but at the cost of interactivity. This is well suited for batch jobs.
- SCHED_IDLE: Lowest priority in the system
- SCHED_NORMAL: Used for regular processes with non-RT priority.

Scheduling classes (1)

- `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 (2)

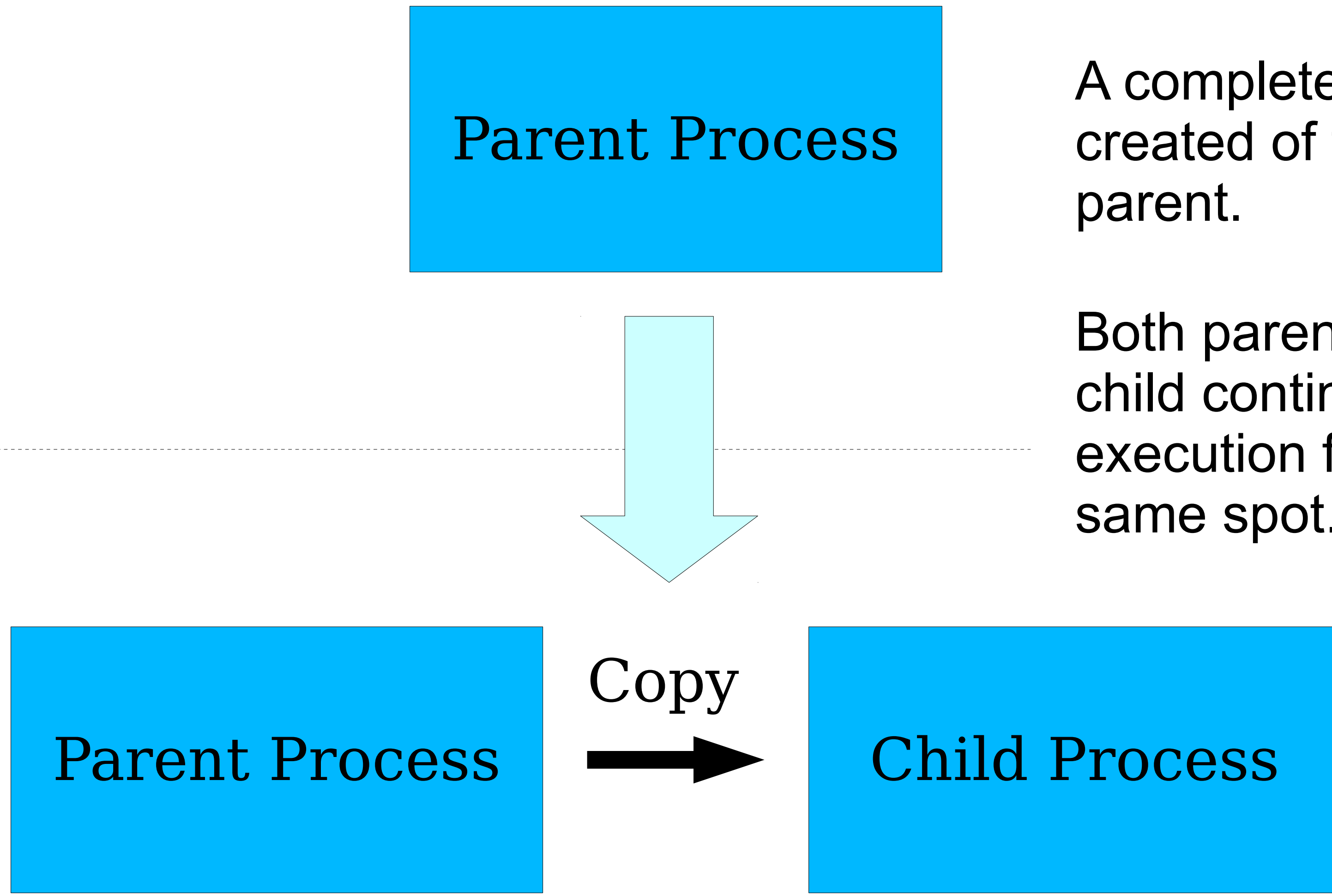
- Priority can be set on a per-thread basis when its creation:

```
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

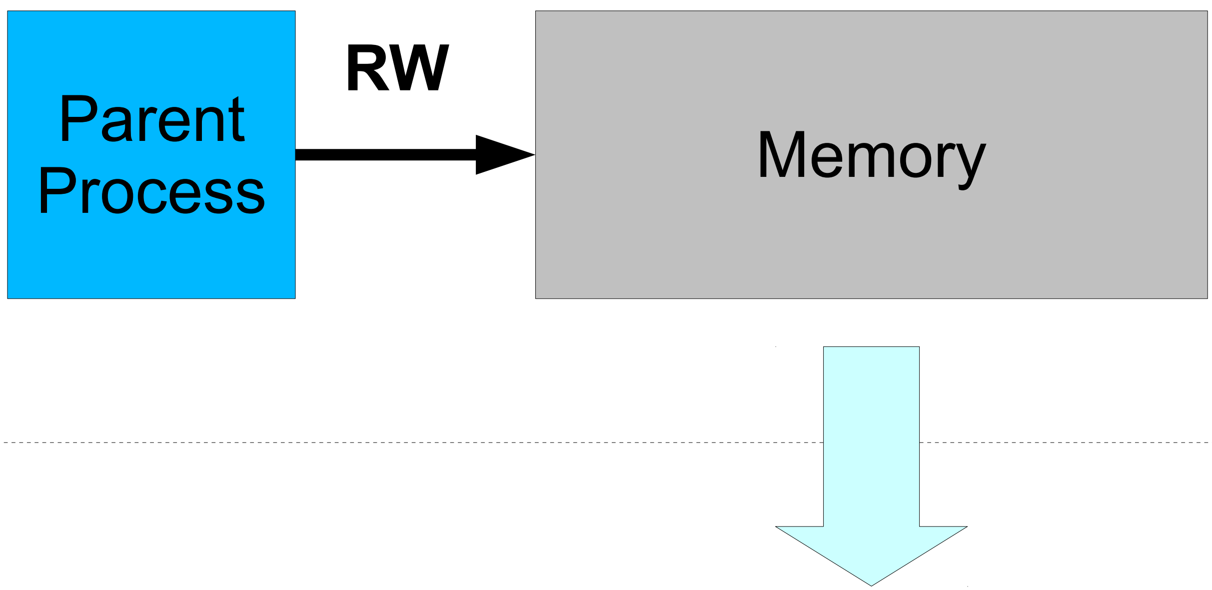
How fork() seems to work



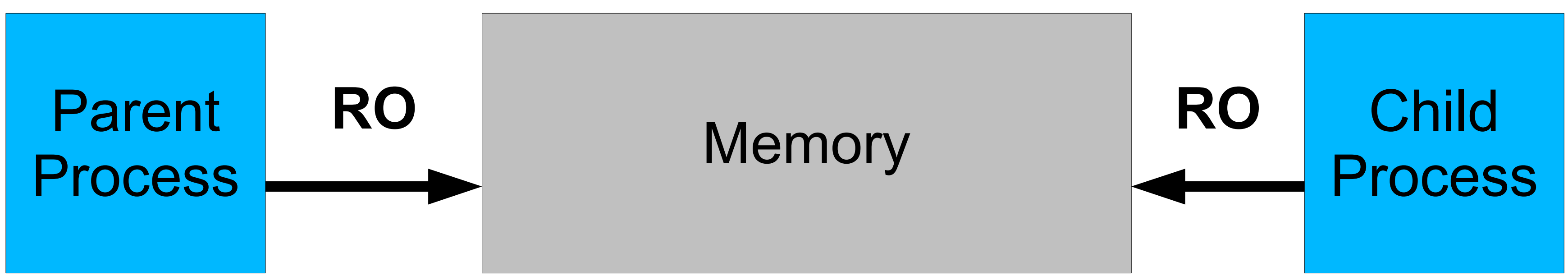
A complete copy is created of the parent.

Both parent and child continue execution from the same spot.

How fork() really works



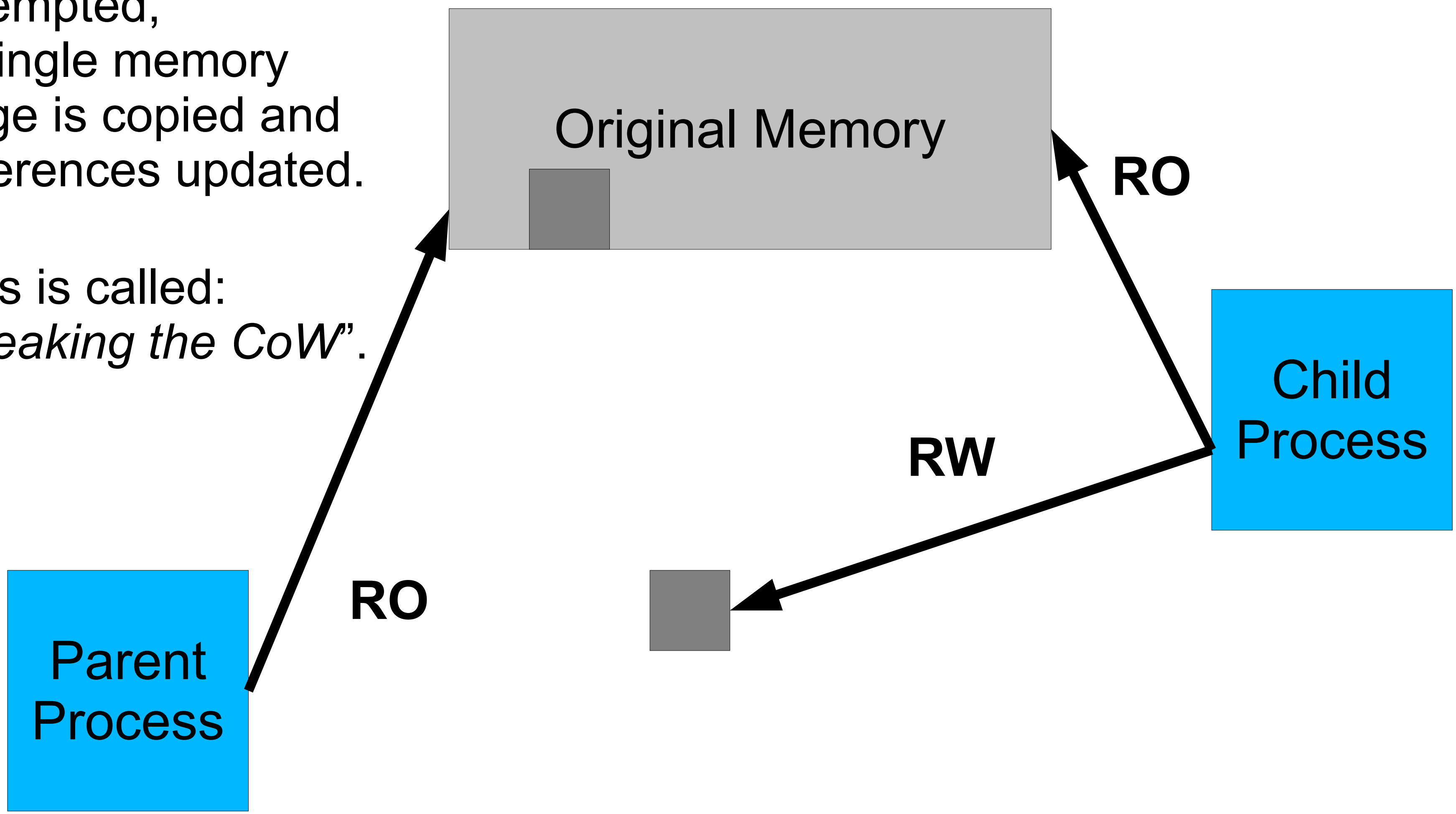
Child process gets a copy of stack and file descriptors. **Copy On Write** reference is used for the memory.



What happens during write?

When write is attempted,
a single memory page is copied and references updated.

This is called:
“*breaking the CoW*”.



Locking pages in RAM

A solution to latency issues with demand paging!

- Available through `<sys/mman.h>` (see `man mman.h`)
- `mlock`
  Lock a given region of process address space in RAM.
  Makes sure that this region is always loaded in RAM.
- `mlockall`
  Lock the whole process address space.
- `munlock`, `munlockall`
  Unlock a part or all of the process address space.

Realtime Hello World

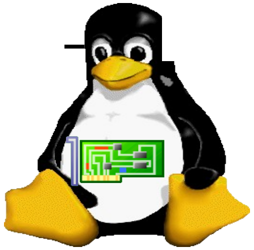
```
#include <stdlib.h>
#include <stdio.h>
#include <time.h>
#include <sched.h>
#include <sys/mman.h>
#include <string.h>

#define MY_PRIORITY (49) /* we use 49 as the PRREMPT_RT use 50
                    as the priority of kernel tasklets
                    and interrupt handler by default */

#define MAX_SAFE_STACK (8*1024) /* The maximum stack size
which is
                    guranteed safe to access without
                    faulting */
#define NSEC_PER_SEC    (1000000000) /* The noof nsecs per sec. */
void stack_prefault(void) {
    unsigned char dummy[MAX_SAFE_STACK];
    memset(dummy, 0, MAX_SAFE_STACK);
    return;
}
...
```

https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO

```
int main(int argc, char* argv[])
{
    struct timespec t;
    struct sched_param param;
    int interval = 50000; /* 50us*/
    param.sched_priority = MY_PRIORITY; /* Declare ourself as a real time task */
    if(sched_setscheduler(0, SCHED_FIFO, &param) == -1) {
        perror("sched_setscheduler failed");        exit(-1);
    }
    /* Lock memory */
    if(mlockall(MCL_CURRENT|MCL_FUTURE) == -1) {
        perror("mlockall failed");        exit(-2);
    }
    stack_prefault(); /* Pre-fault our stack */
    clock_gettime(CLOCK_MONOTONIC ,&t);
    /* start after one second */
    t.tv_sec++;
    while(1) {
        /* wait until next shot */
        clock_nanosleep(CLOCK_MONOTONIC, TIMER_ABSTIME, &t, NULL);
        /* do the RT stuff */
        t.tv_nsec += interval; /* calculate next shot */
        while (t.tv_nsec >= NSEC_PER_SEC) {
            t.tv_nsec -= NSEC_PER_SEC;    t.tv_sec++;
        }
    }
}
```



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 be activated explicitly
  ```
  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 asynchronous notification mechanisms
- 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()`
- The mask of blocked signals can be changed with `pthread_sigmask()`
- Delivery of a signal using `pthread_kill()` or `tgkill()`

Benchmarking (1)

cyclictest

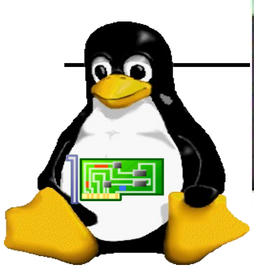
- measuring accuracy of sleep and wake operations of highly prioritized realtime threads
- https://rt.wiki.kernel.org/index.php/Cyclictest

```
insop@chai:~/Projects/rt-tests$ uname -a
Linux chai 3.2.0-24-generic-pae #39-Ubuntu SMP Mon May 21 18:54:21 UTC 2012 i686 i686 i386 GNU/Linux
insop@chai:~/Projects/rt-tests$ sudo ./cyclictest -a -t -n -p99
# /dev/cpu_dma_latency set to 0us
policy: fifo: loadavg: 0.54 0.69 0.67 6/417 3256

T: 0 ( 2772) P:99 I:1000 C:1008249 Min:      4 Act:   18 Avg:   11 Max:     701
T: 1 ( 2773) P:99 I:1500 C: 672166 Min:      4 Act:   35 Avg:   11 Max:     491
T: 2 ( 2774) P:99 I:2000 C: 504124 Min:      4 Act:    9 Avg:   11 Max:     363
T: 3 ( 2775) P:99 I:2500 C: 403299 Min:      4 Act:   14 Avg:   11 Max:    2013
T: 4 ( 2776) P:99 I:3000 C: 336082 Min:      4 Act:   14 Avg:   14 Max:     804
T: 5 ( 2777) P:99 I:3500 C: 288071 Min:      3 Act:   13 Avg:    9 Max:     190
T: 6 ( 2778) P:99 I:4000 C: 252062 Min:      3 Act:    9 Avg:    9 Max:     343
T: 7 ( 2779) P:99 I:4500 C: 224055 Min:      3 Act:   13 Avg:   10 Max:     224
```

vanilla kernel

Worst case latency: hundreds of usec



```
T: 0 ( 2995) P:99 I:1000 C:1030921 Min:      5 Act:    7 Avg:   12 Max:      32
T: 1 ( 2996) P:99 I:1500 C: 687280 Min:      5 Act:    7 Avg:   13 Max:      53
T: 2 ( 2997) P:99 I:2000 C: 515455 Min:      4 Act:    6 Avg:   13 Max:      34
T: 3 ( 2998) P:99 I:2500 C: 412364 Min:      5 Act:    7 Avg:   13 Max:      34
T: 4 ( 2999) P:99 I:3000 C: 343637 Min:      6 Act:   11 Avg:   16 Max:      31
T: 5 ( 3000) P:99 I:3500 C: 294546 Min:      3 Act:    4 Avg:   11 Max:     338
T: 6 ( 3001) P:99 I:4000 C: 257727 Min:      4 Act:    5 Avg:   11 Max:      24
T: 7 ( 3002) P:99 I:4500 C: 229091 Min:      3 Act:    5 Avg:   11 Max:      29
```

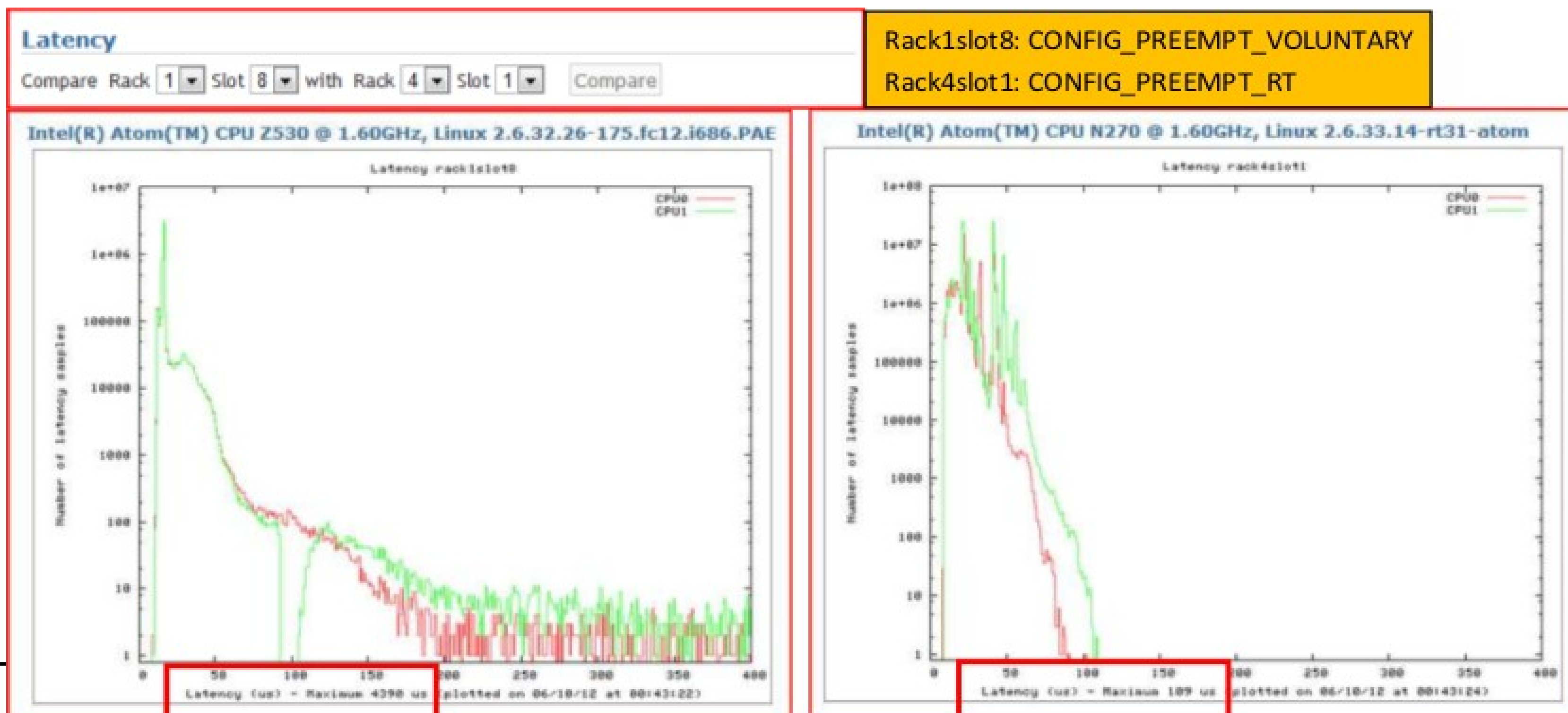
PREEMPT_RT

Worst case latency: tens of usec

Benchmarking (2)

QA farm running at OSADL.ORG

- worst case latency comparison between 2.6.32 and 2.6.33-rt
- https://www.osadl.org/Compare-systems.qa-farm-compare-latency.0.html
- Worst case latency in non-rt is over 4,000 usec, in rt around 100 usec



Again, Response time vs. Throughput

- Overhead for real-time preemption
  - Mutex instead of spin lock
  - Priority inheritance
  - Threaded interrupt handler
- Due to overhead of real-time preemption
  - Throughput is reduced
- Due to the flexibility of preemption
  - Better worst case latency

Making Linux do Hard Real-time

Linux hard real-time extensions

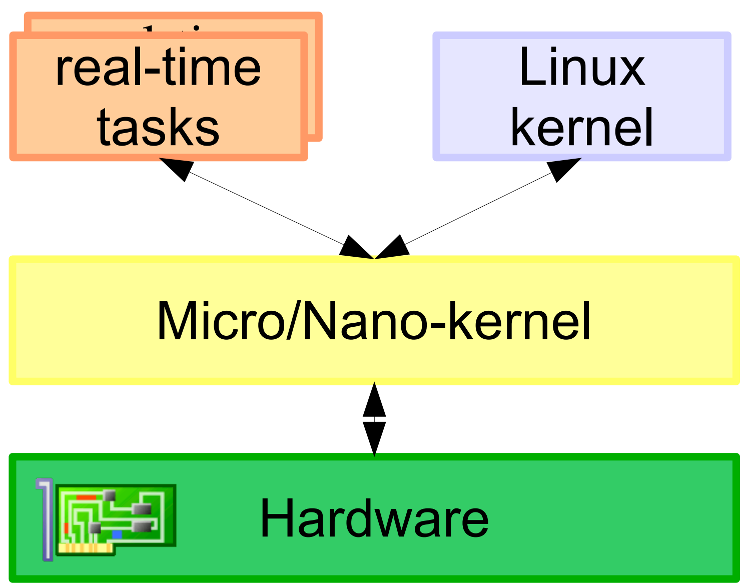
Linux hard real-time extensions

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.



Interrupt Response Time

PREEMPT: standard kernel with CONFIG_PREEMPT (“Preemptible Kernel (Low-Latency Desktop)) enabled
`cyclictest -m -n -p99 -t1 -i10000 -1360000`

XENOMAI: Kernel + Xenomai 2.6.0-rc4 + I-Pipe 1.18-03
`cyclictest -n -p99 -t1 -i10000 -1360000`

Configuration	Avg	Max	Min
XENOMAI	43	58	2
PREEMPT 1st run	88	415	27
PREEMPT 2nd run	81	1829	13

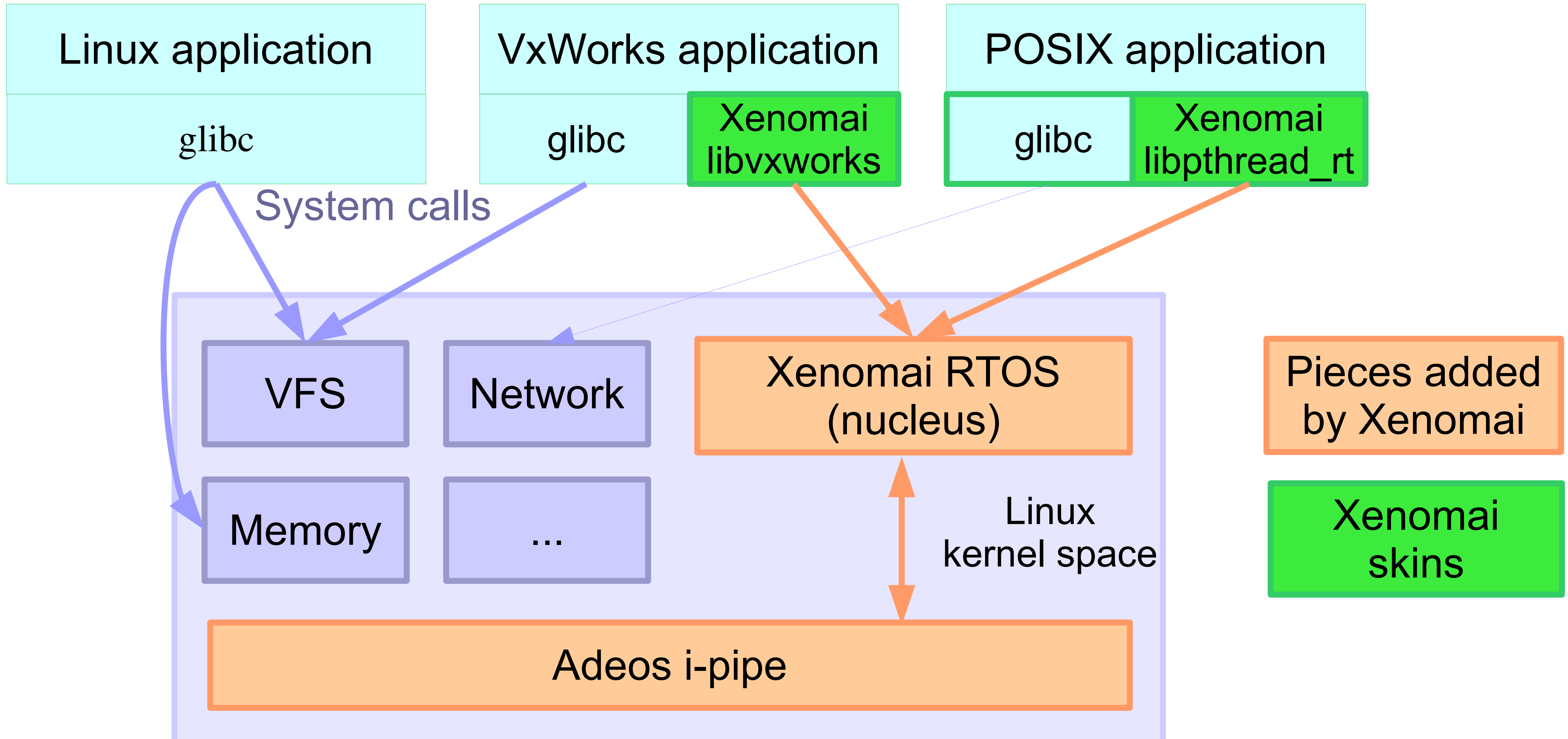
Hardware: Freescale i.MX53 ARM Cortex-A8 processor operating at 1GHz.
Time in micro second.

Xenomai project

http://www.xenomai.org/

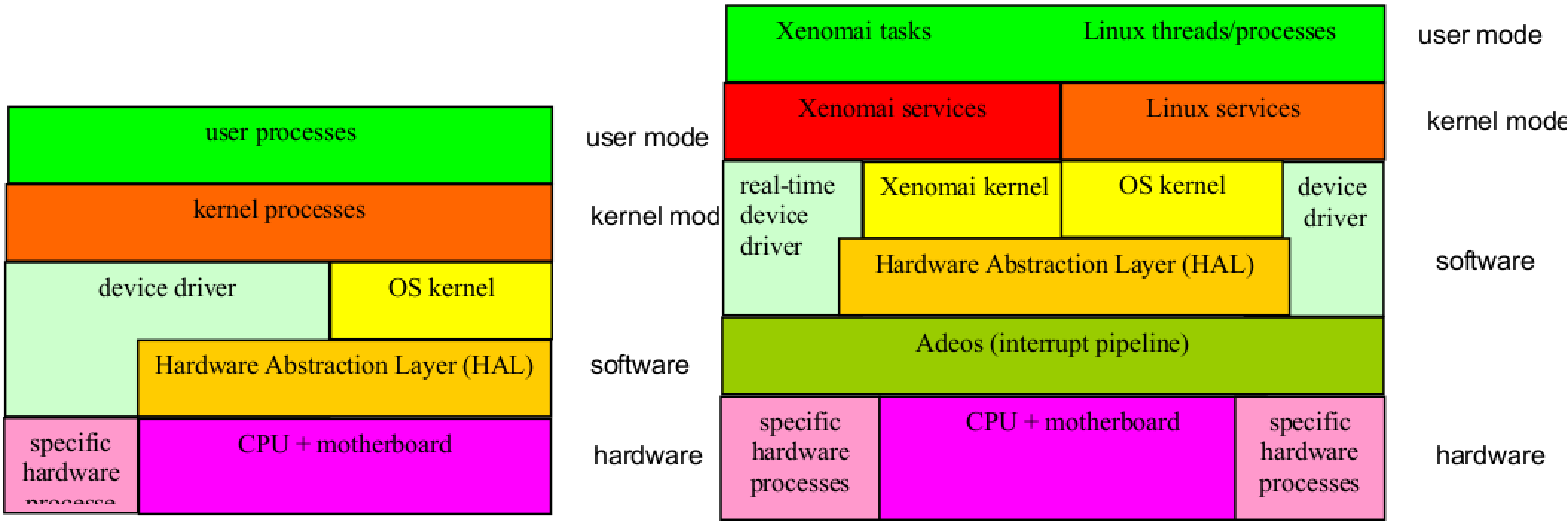
- Started in the RTAI project
  (called RTAI / fusion).
- Skins mimicking the APIs of traditional
  RTOS such as VxWorks, pSOS+, and VRTXsa.
- Initial goals: facilitate the porting of programs from traditional RTOS to RTAI on GNU / Linux.
- Now an independent project and an alternative to RTAI. Many contributors left RTAI for Xenomai, frustrated by its goals and development style.

Xenomai architecture

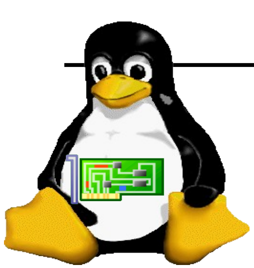


ipipe = interrupt pipeline

Linux vs. Xenomai

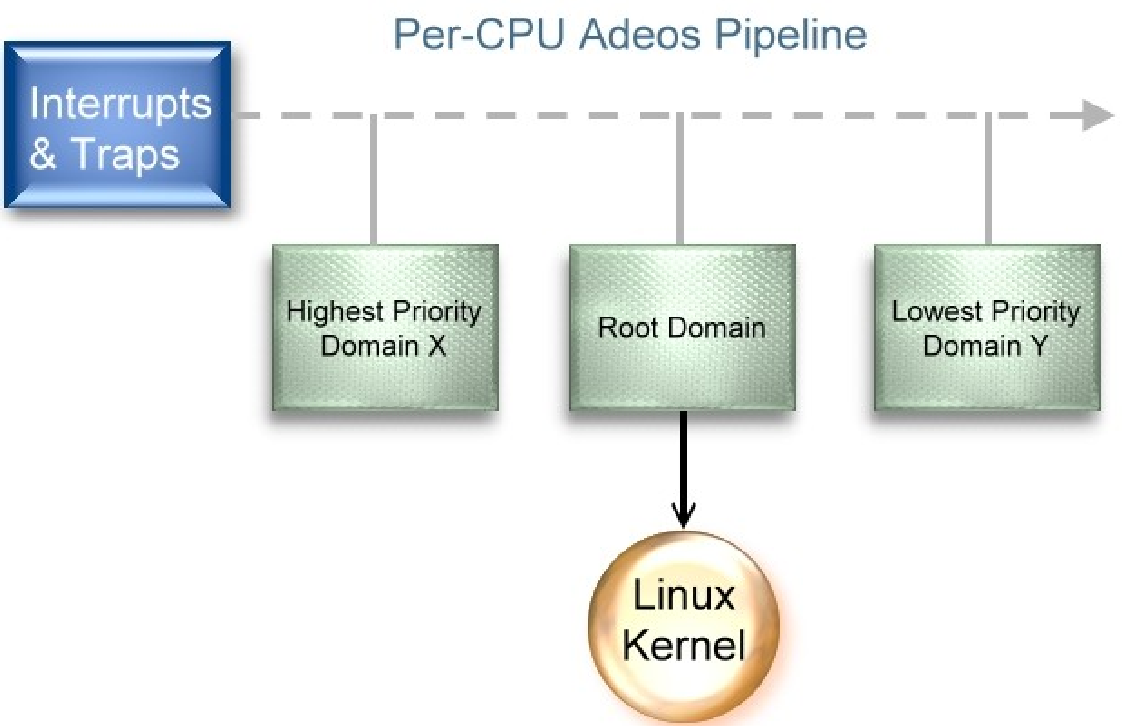


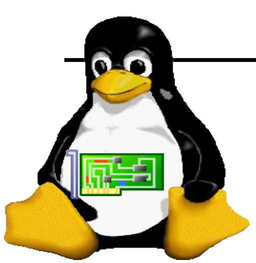
- A special way of passing of the interrupts between real-time kernel and the Linux kernel is needed.
- Each flavor of RT Linux does this is in its own way. Xenomai uses an interrupt pipeline from the Adeos project.



Interrupt pipeline abstraction

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



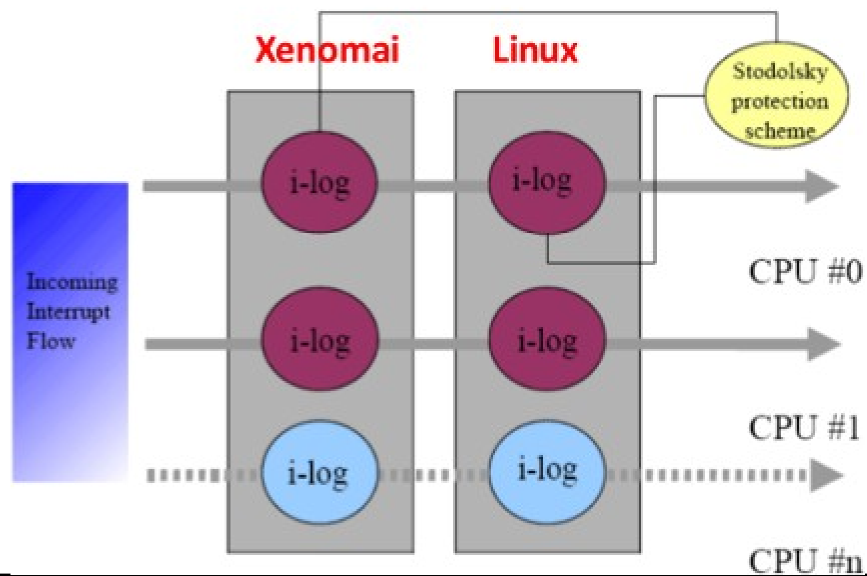


i-pipe: Optimistic protection scheme (1)

- Main idea
  - support interrupt priority
  - ensure that domains support segments of code that can not be interrupted (like critical sections)
- Adeos implements the Optimistic protection scheme (Stodolsky, 1993):
  - When a segment of code is executed in a domain with interrupts disabled, that stage of the pipeline stalls, so interrupts can not be serviced nor propagated further in the pipeline

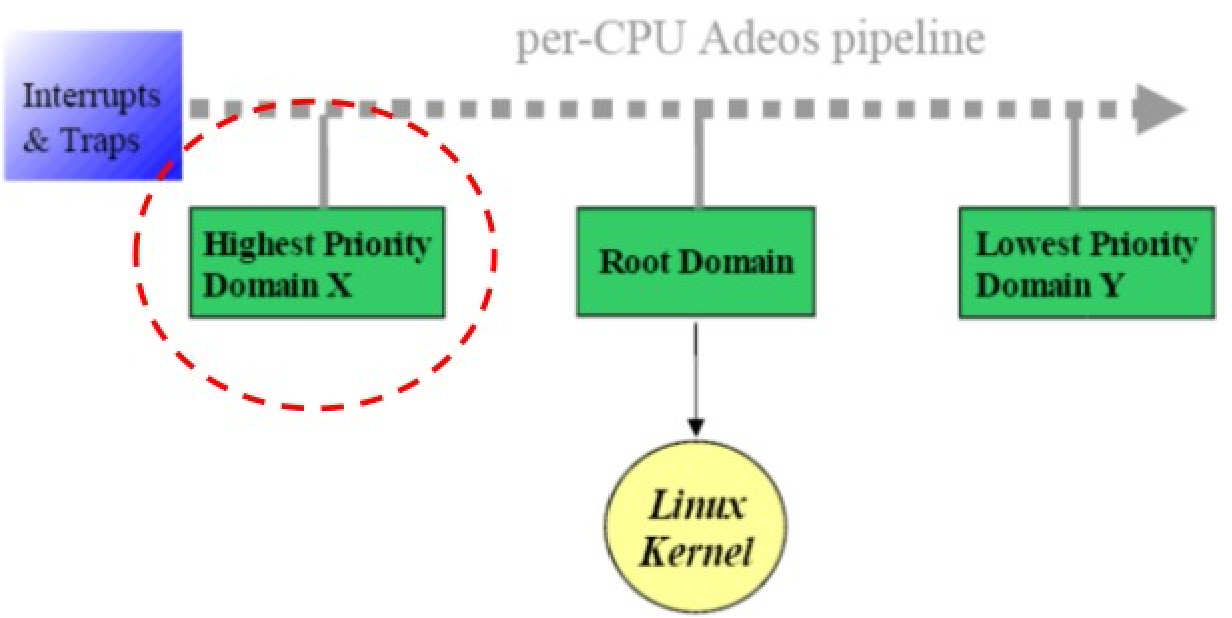
i-pipe: Optimistic protection scheme (2)

- If a real time domain (like Xenomai) has higher priority it is the first in the pipeline
  - It will receive interrupt notification first without delay (or at least with predictable latency)
  - Then it can be decided if interrupts are propagated to low priority domains (like Linux) or not



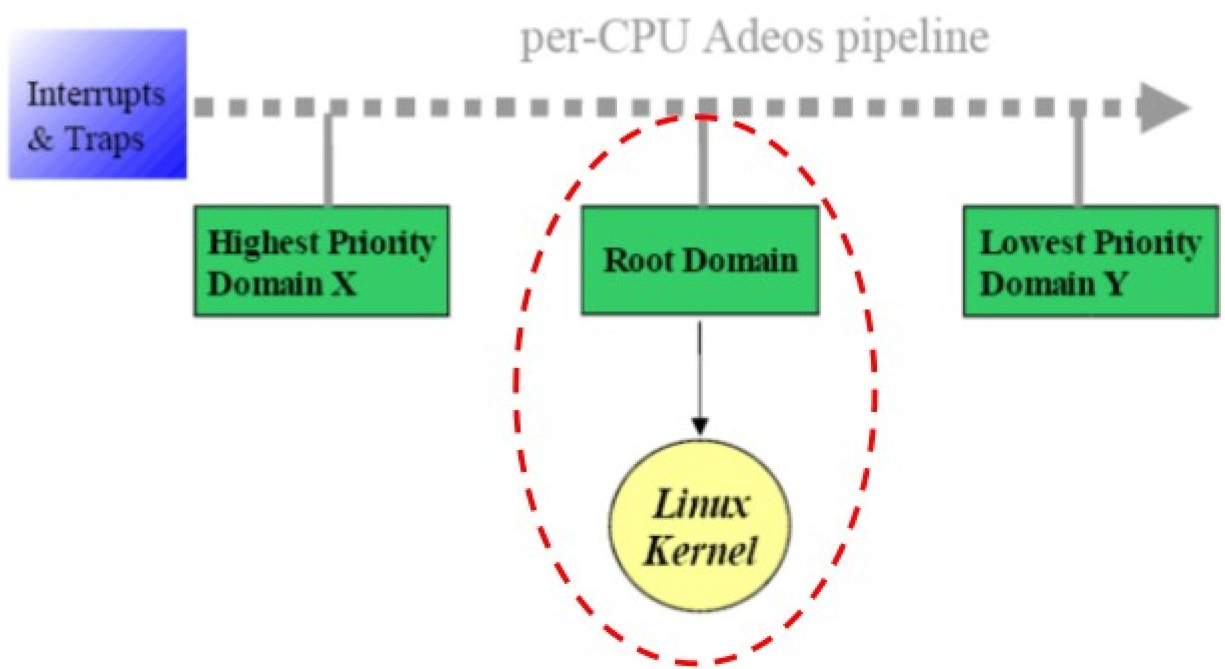
Interrupt pipeline (1)

- The high priority domain is at the beginning of the pipeline, so events are delivered first to it
- This pipeline is referred as interrupt pipeline or I-pipe
- There is a pipeline for each CPU



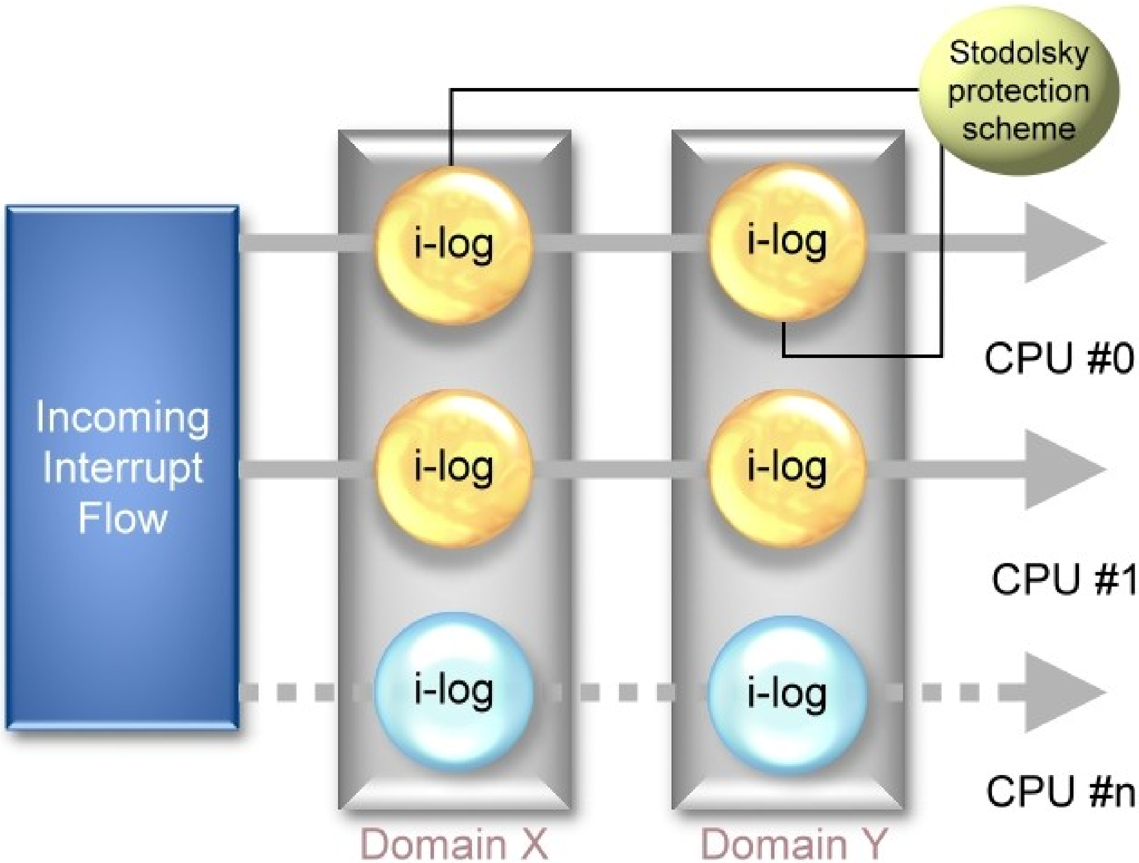
Interrupt pipeline (2)

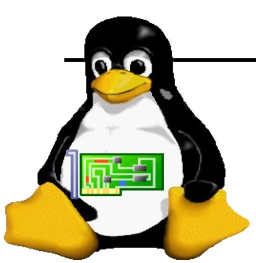
- The Linux domain is always the root domain, whatever is its position in the pipeline
- Other domains are started by the root domain
- Linux starts and loads the kernel modules that implement other domains



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.





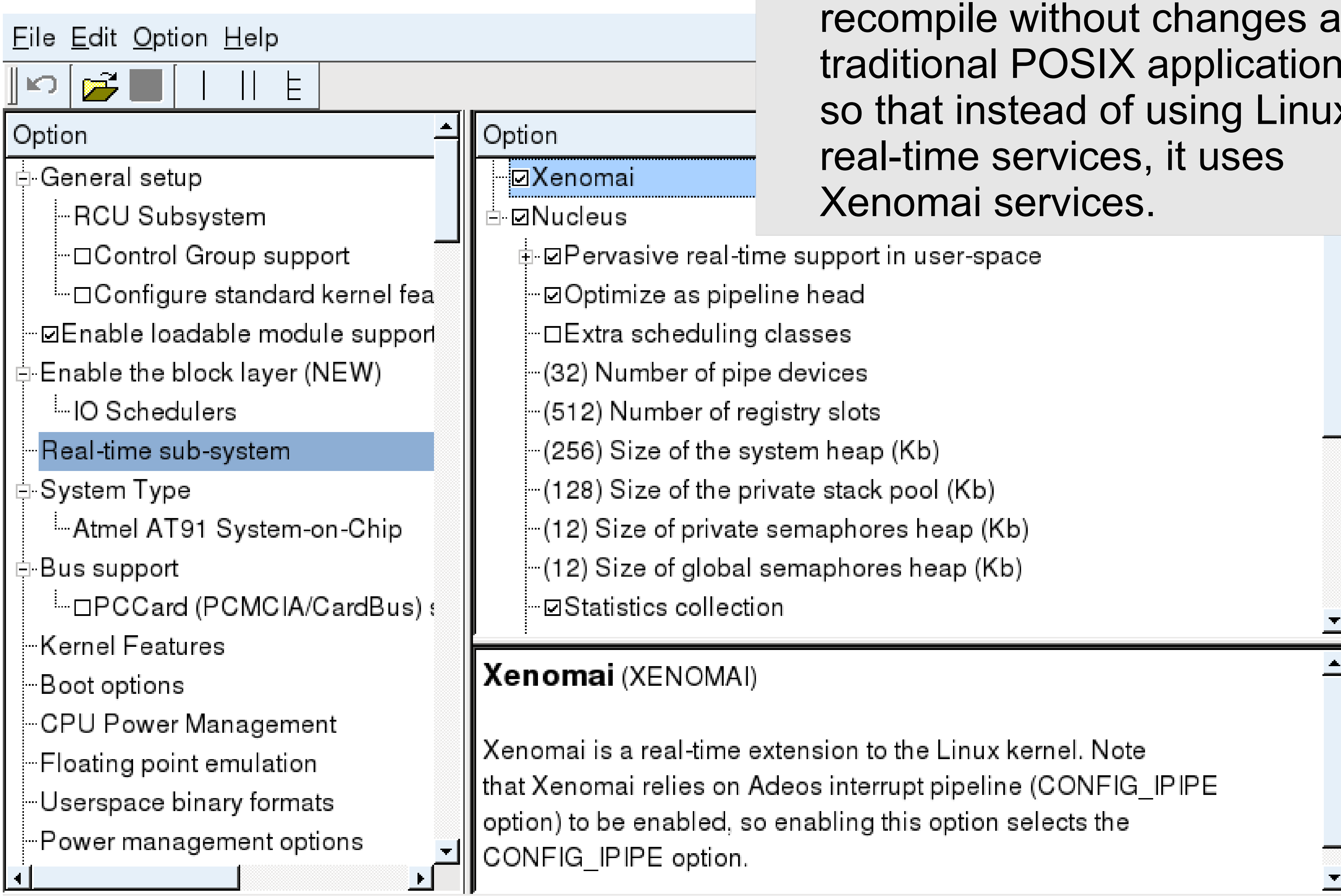
I-pipe observation through procfs

```
$ cat /proc/ipipe/Xenomai
        +----- Handling ([A]ccepted, [G]rabbed,
        |+---- Sticky    [W]ired, [D]iscarded)
        ||+--- Locked
        |||+-- Exclusive
        ||||+- Virtual
[IRQ]
 38: W..X.
418: W...V
```

```
$ cat /proc/ipipe/Linux
0: A....
1: A....
...
priority=100
```

Linux options for Xenomai configuration

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.



Xenomai Execution Model (1)

All tasks managed by Xenomai are known by the real time nucleus:

- Tasks (Xenomai tasks or threads) executed in the Xenomai domain are executed in primary mode (or primary domain)

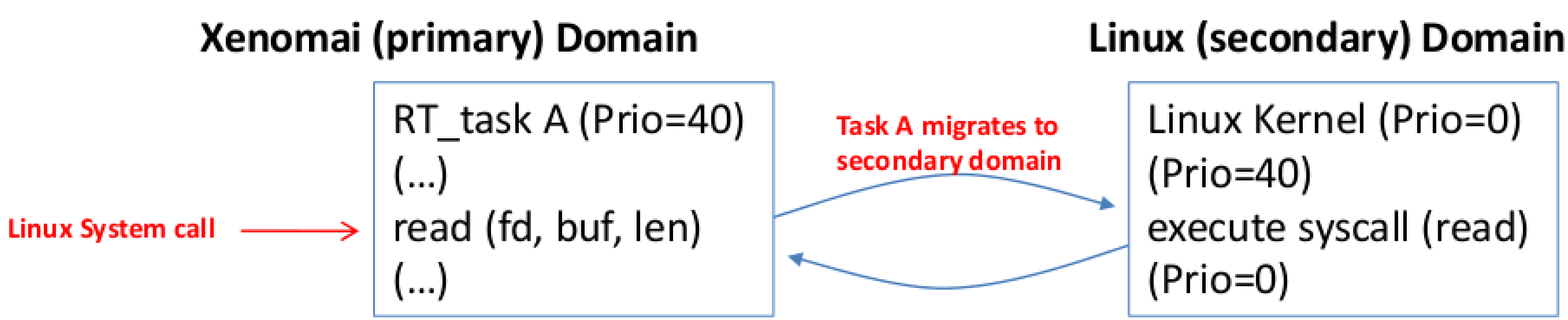
  [the highest priority domain; real time domain]

- Tasks (Linux threads or processes) executed in the Linux domain are executed in secondary mode (or secondary domain)

Xenomai Execution Model (2)

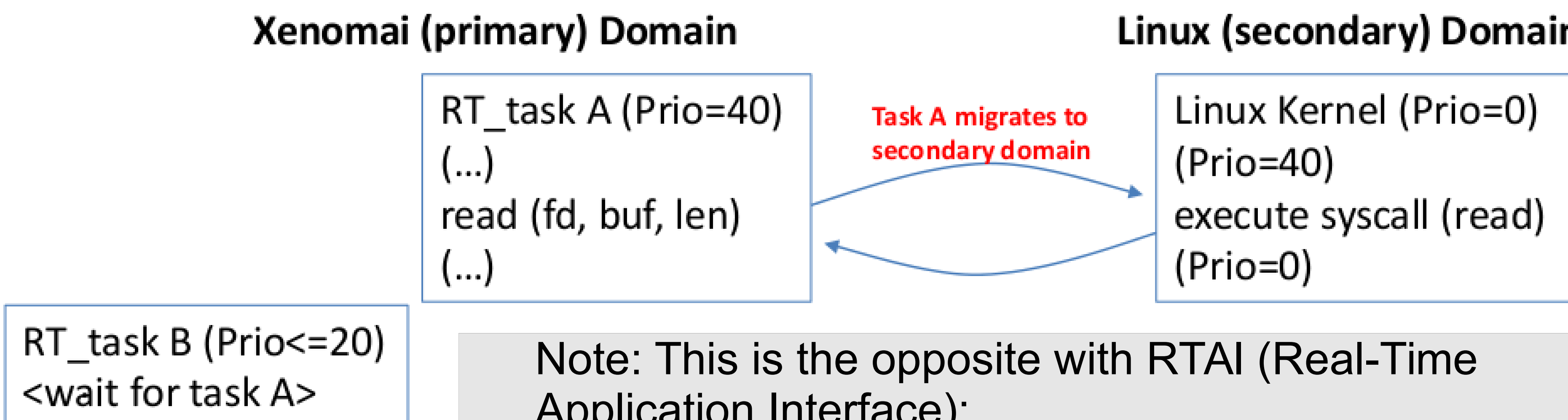
To ensure that tasks in secondary domain still have real time support, it is required:

- (A) Common priority scheme
  - When a Xenomai task migrates to Linux domain, the root thread's mutable priority technique is applied:
    - Linux kernel inherits the priority of the Xenomai thread that migrates



Xenomai Execution Model (3)

- Common priority scheme
  - This means that other threads in the Xenomai Domain can only preempt the migrated thread (ie. the linux kernel executing the syscall) if their priority is higher



Note: This is the opposite with RTAI (Real-Time Application Interface):

> A thread that enters the Linux Domain inherits the priority of the Linux kernel, which is the lowest in the RTAI scheduler

Xenomai Execution Model (4)

To ensure that tasks in secondary domain still have real time support, it is required:

- (B) Predictability of program execution time
  - When a Xenomai thread enters secondary domain (Linux) it should not be preempted by:
    - Linux domain interrupts
    - other low priority asynchronous activity at Linux kernel level
  - Note that it is assumed that when an interrupt is delivered to Linux domain, Linux Kernel code can be preempted whatever its priority
  - To avoid this, Linux kernel can be starved from interrupts while the Xenomai thread is in secondary mode

Xenomai Execution Model (5)

To ensure that a Xenomai tasks in secondary domain have still real time support (and predictable execution time)

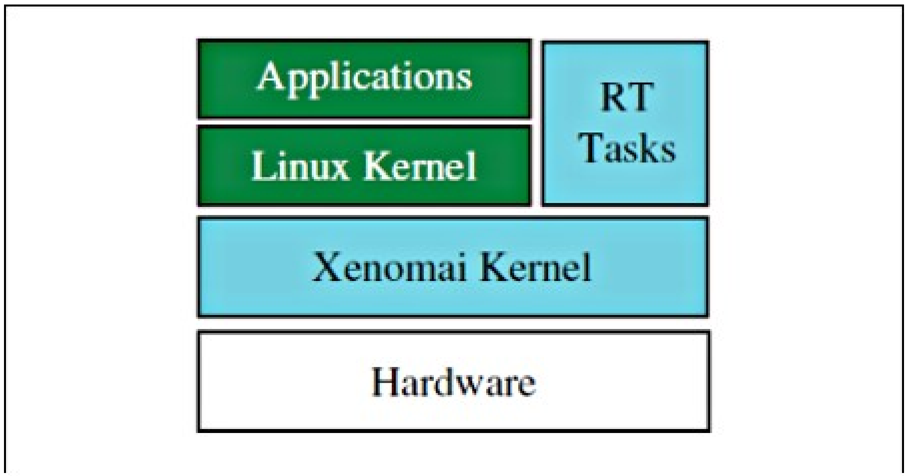
- While a Xenomai task migrates to Linux domain (secondary):
  - Linux kernel inherits the priority of the Xenomai task that migrates (normal Linux kernel priority is the lowest: 0)
  - Linux is starved from interrupts and other system events, which are blocked in an intermediate domain: The interrupt shield (to avoid that the Linux kernel interrupts the migrated Xenomai task when responding to an interrupt or other asynchronous event)

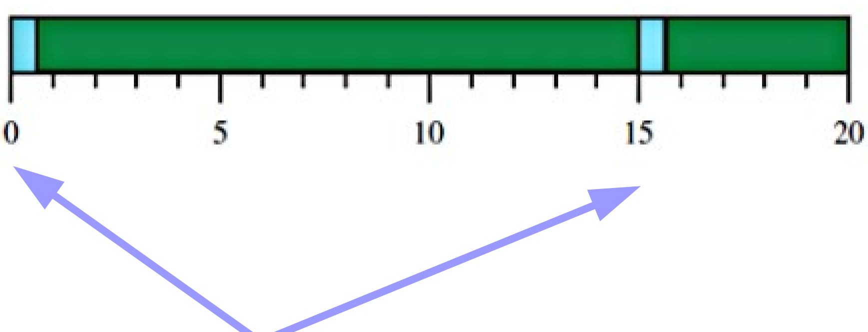
Adeos additional features

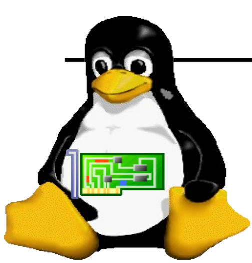
- The Adeos I-pipe patch implement additional features, essential for the implementation of the Xenomai real-time extension:
  - Disables on-demand mapping of kernel-space `vmalloc/ioremap` areas.
  - Disables copy-on-write when real-time processes are forking.
  - Allow subscribing to event allowing to follow progress of the Linux kernel, such as Linux system calls, context switches, process destructions, POSIX signals, FPU faults.
  - On ARMv5 architecture, utilizes FCSE to reduce the latency induced by cache flushes during context switches.

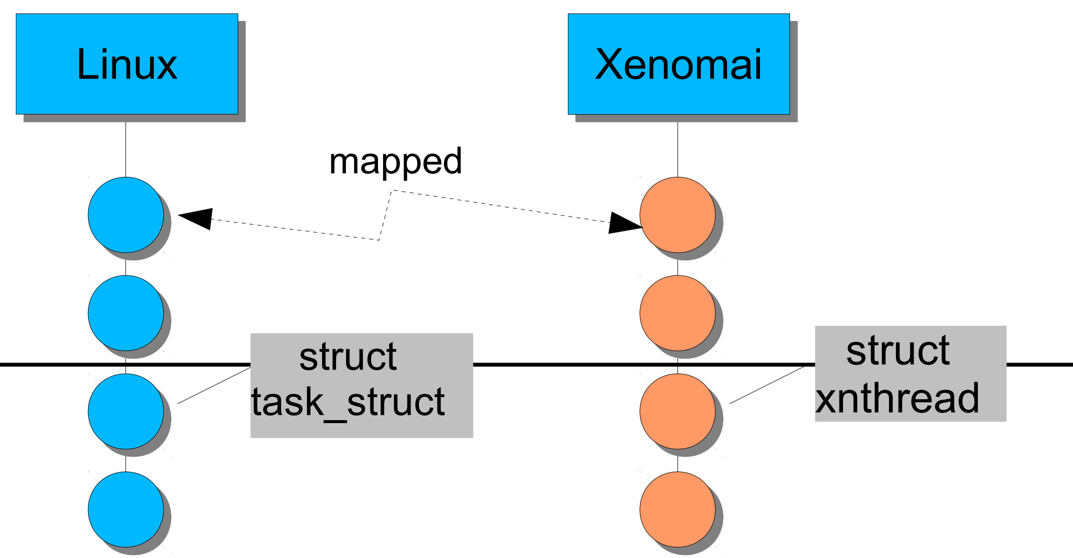
Real-Time Scheduler

- Xenomai extends the Linux kernel and is integrated as part of OS.
- A task with a period = 15 us, shown in light blue.
- While this real-time task is not being executed, Xenomai invokes the regular Linux scheduler which executes tasks as normal, shown in dark green.









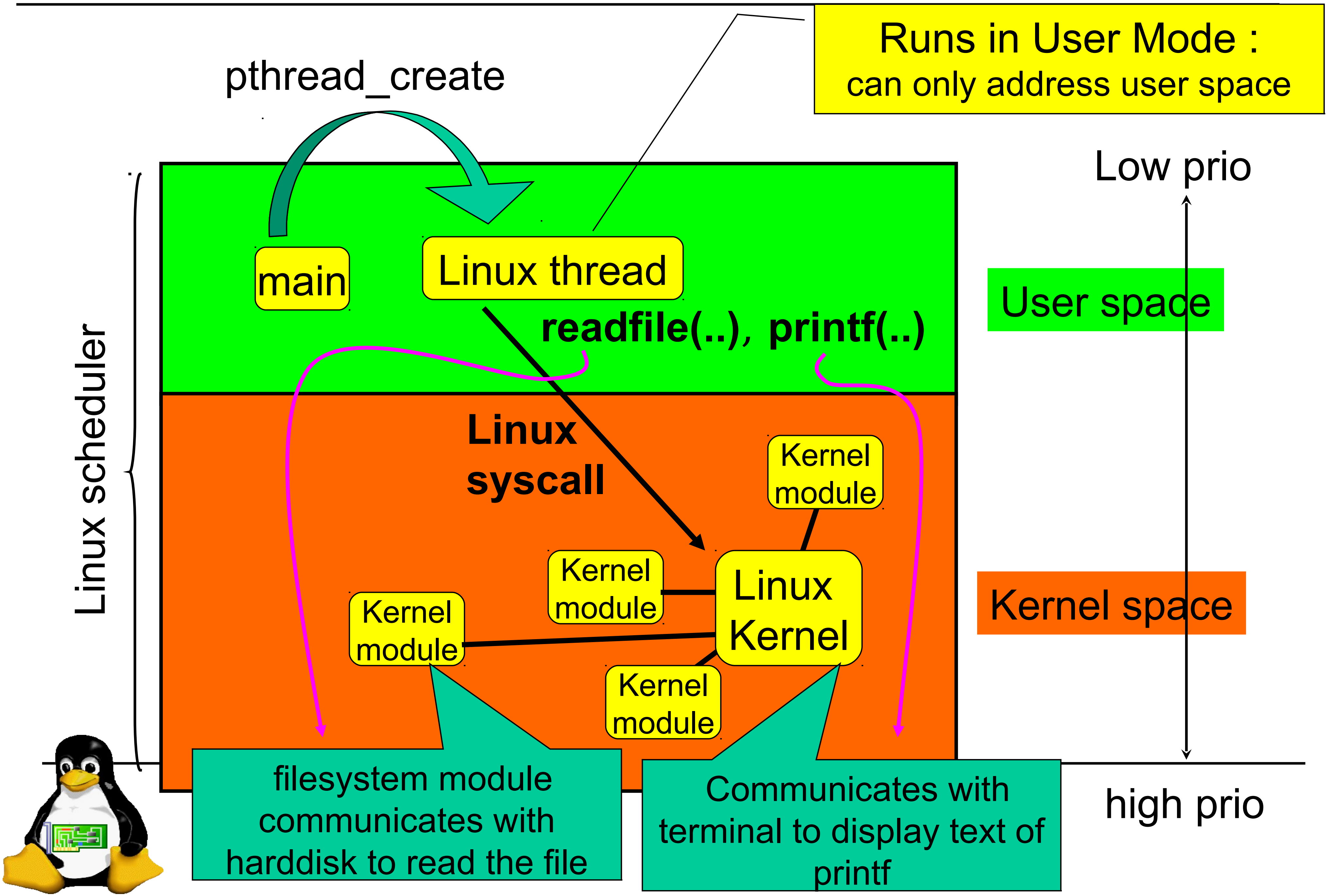
Xenomai features

- Factored RT core with skins implementing various RT APIs
- Seamless support for hard real-time in user-space
- No second-class citizen, all ports are equivalent feature-wise
- 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

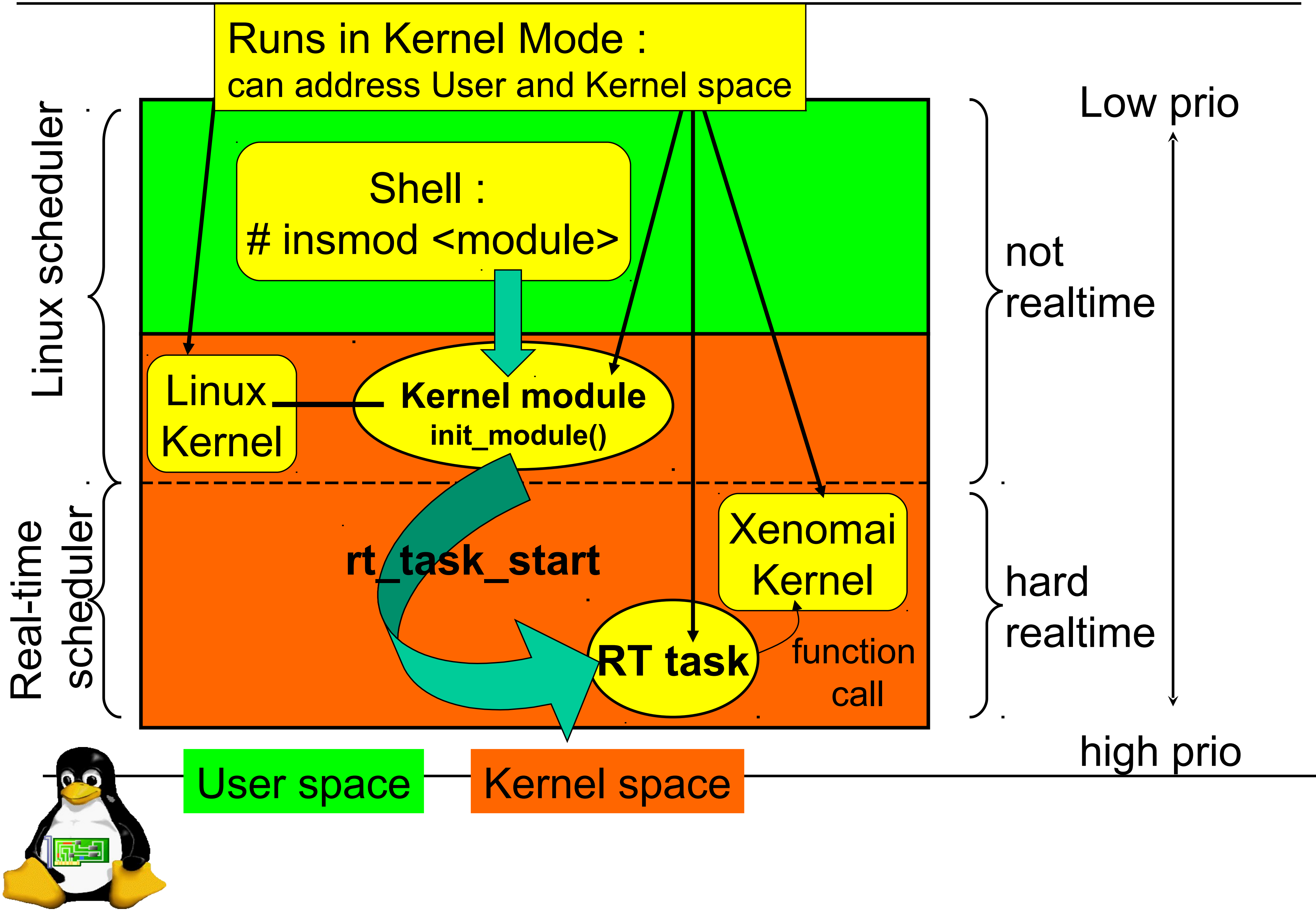
Xenomai real-time user-space support

- 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.

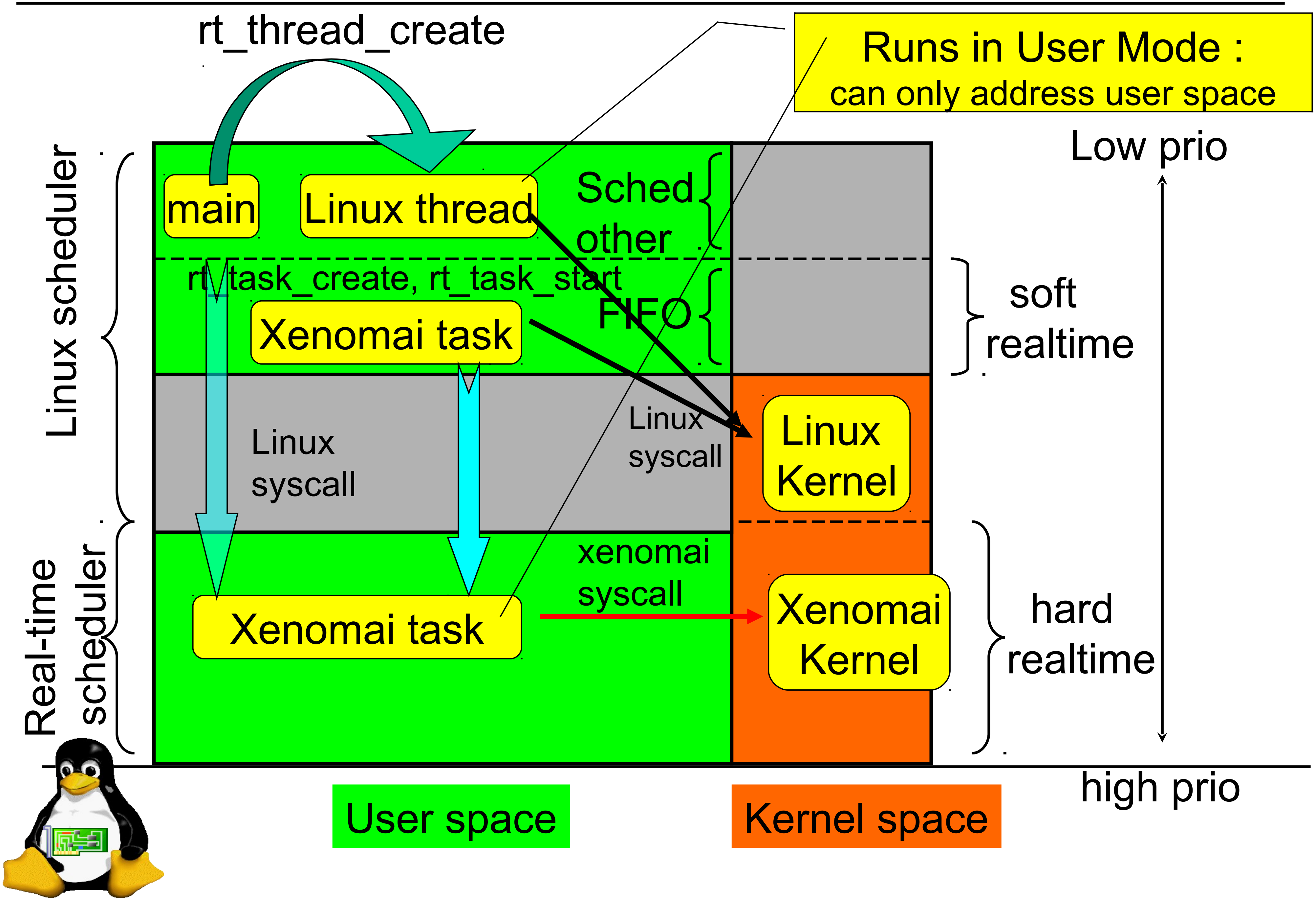
Original Linux



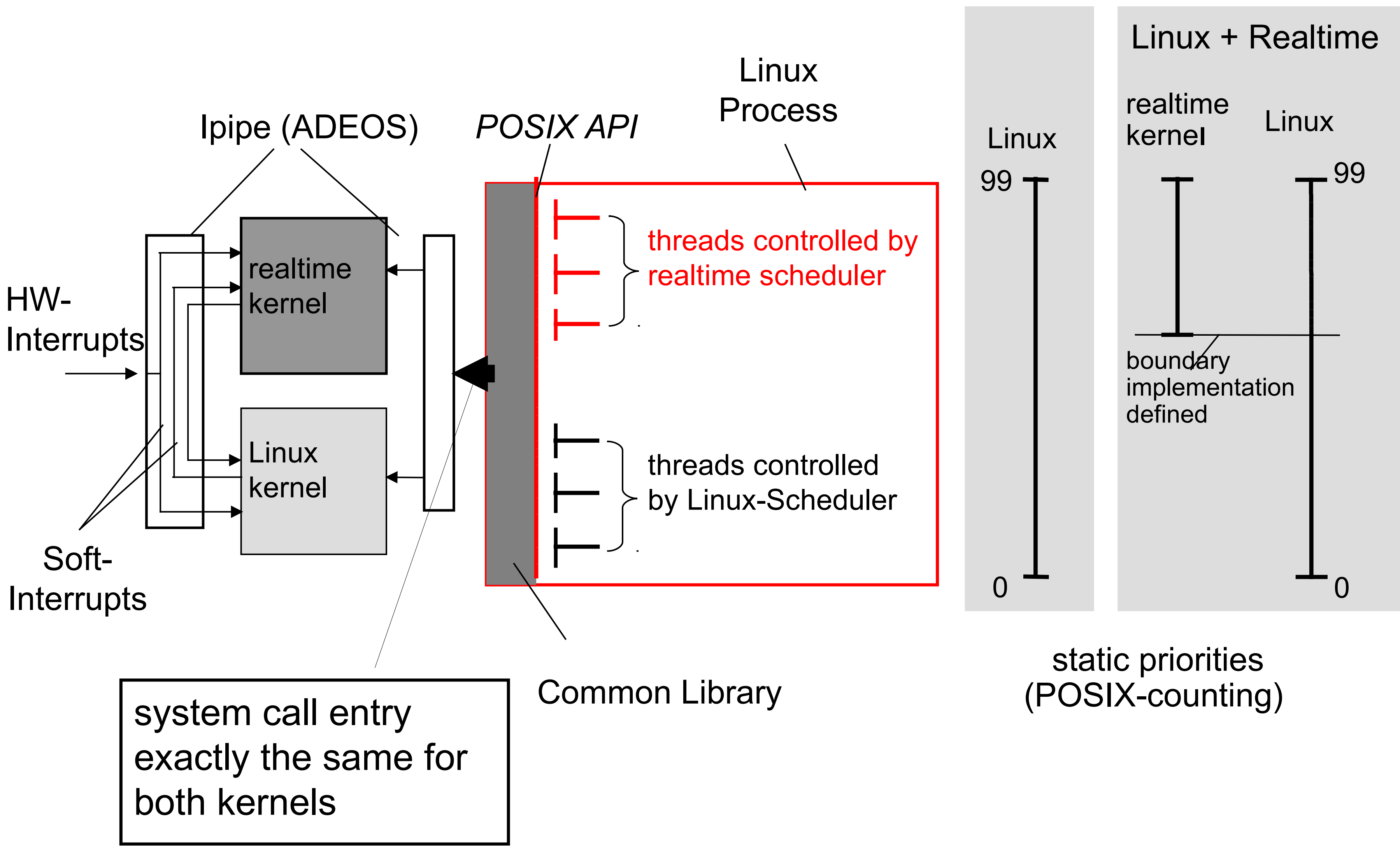
Xenomai (kernel space)



Xenomai (user space)



Concepts changed in Xenomai



Real-time Linux task

Observe the difference between real-time tasks and standard Linux programs:

- In real-time Linux, we can make a real-time program by programming real-time threads.
  - None-real-time programs in real-time Linux just use the conventional Linux threads.
- A real-time task can be executed in kernel space using a kernel module, but it can also be executed in user space using a normal C program.

Real-time tasks in Linux userspace

Running in user space instead of in kernel space gives a little extra overhead, but with the following advantages:

- Bugs in a user space program can only crash the program.
- In a kernel model, one can only use the real-time API and the limited kernel API, but in a real-time user space program the real-time API and the whole Linux API can be used.
  - However when using the Linux API in user space, the program cannot be scheduled by the real-time scheduler (HARD real-time) but must be scheduled by the Linux scheduler (SOFT real-time).
  - So calling a Linux system call from a real-time user space task will degrade its performance from HARD to SOFT real-time. After the call the task will return to the real-time scheduler.

RTDM

Real-Time Driver Model

- An approach to unify the interfaces for developing device drivers and associated applications under real-time Linux.
- Currently available for Xenomai and RTAI
  RTDM-native: a port over native Linux with the real-time preemption patch.
- See the whitepaper on
  http://www.xenomai.org/documentation/xenomai-2.6/pdf/RTDM-and-Applications.pdf

The POSIX skin

- 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

```
Export DESTDIR=/path/to/xenomai/
CFLAGS=`$DESTDIR/bin/xeno-config --posix-cflags`
LDFLAGS=`$DESTDIR/bin/xeno-config --posix-ldflags`
{CROSS_COMPILE}gcc ${CFLAGS} -o rttest rttest.c ${LDFLAGS}
```

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 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()`

Conclusion

- Linux was not designed as a RTOS
- However, you can make your system **hard real-time** by using one of the hard real-time extensions (e.g. Xenomai), and writing your critical applications and drivers with the corresponding APIs.
- You can get **soft real-time** with the standard kernel preemption mode. **Most** of the latencies will be reduced, offering better quality, but probably not all of them.
- However, using **hard real-time extensions will not guarantee that your system is hard real-time**. Your system and applications will also have to be designed properly (correct priorities, use of deterministic APIs, allocation of critical resources ahead of time...).

Reference

- Soft, Hard and Hard Real Time Approaches with Linux, Gilad Ben-Yossef
- A nice coverage of Xenomai (Philippe Gerum) and the RT patch (Steven Rostedt):http://oreilly.com/catalog/9780596529680/
- Real-time Linux, Insop Song
- Understanding the Latest Open-Source Implementations of Real-Time Linux for Embedded Processors, Michael Roeder