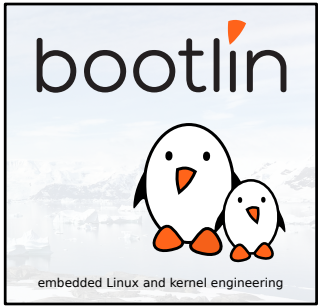
Understanding Linux real-time with PREEMPT_RT

Latest update: January 17, 2022.

Document updates and sources:
https://bootlin.com/doc/training/preempt-rt

Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@bootlin.com



Rights to copy

Document sources: https://github.com/bootlin/training-materials/

Hyperlinks in the document

There are many hyperlinks in the document

- Regular hyperlinks:
  https://kernel.org/
- Kernel documentation links:
  dev-tools/kasan
- Links to kernel source files and directories:
  drivers/input/
  include/linux/fb.h
- Links to the declarations, definitions and instances of kernel symbols (functions, types, data, structures):
  platform_get_irq()
  GFP_KERNEL
  struct file_operations

Company at a glance

- Engineering company created in 2004, named "Free Electrons" until Feb. 2018.
- Main locations: Toulouse, Lyon (France)
- Serving customers all around the world
- Head count: 13 - Only Free Software enthusiasts!
- Focus: Embedded Linux, Linux kernel, build systems and low level Free and Open Source Software for embedded and real-time systems.
- Feb. 2021: Bootlin is the 20th all-time Linux kernel contributor
- Activities: development, training, consulting, technical support.
- Added value: get the best of the user and development community and the resources it offers.

bootlin

No.1	Unknown	140019(15.26%)
No.2	Intel	94806(10.33%)
No.3	Red Hat	78140(8.52%)
No.4	Hobbyists	73603(8.02%)
No.5	Novell	39218(4.27%)
No.6	IBM	35085(3.82%)
No.7	Linaro	28288(3.08%)
No.8	AMD	22426(2.44%)
No.9	Google	20489(2.23%)
No.10	Renesas Electronics	18443(2.01%)
No.11	Oracle	17729(1.93%)
No.12	Samsung	17514(1.91%)
No.13	Texas Instruments	16372(1.78%)
No.14	HuaWei	13377(1.46%)
No.15	Mellanox Technologies	11477(1.25%)
No.16	ARM	8919(0.97%)
No.17	Academics	8560(0.93%)
No.18	Consultants	8073(0.88%)
No.19	Broadcom	8011(0.87%)
No.20	Bootlin	7611(0.83%)
No.21	NXP	7549(0.82%)
No.22	Linutronix	7430(0.81%)
No.23	NVIDIA	6951(0.76%)
No.24	Canonical	6855(0.75%)
No.25	Linux Foundation	6369(0.69%)
No.26	Code Aurora Forum	6280(0.68%)
No.27	Pengutronix	6201(0.68%)
No.28	VISION Engraving and Routing Systems	6045(0.66%)
No.29	Analog Devices	5944(0.65%)
No.30	Fujitsu	5120(0.56%)
No.31	QUALCOMM	4903(0.53%)
No.32	Freescale	4694(0.51%)
No.33	Wolfson Microelectronics	4180(0.46%)
No.34	Marvell	4178(0.46%)
No.35	Nokia	4097(0.45%)
No.36	Cisco	4071(0.44%)
No.37	Parallels	3841(0.42%)
No.38	Imagination Technologies	3774(0.41%)
No.39	Facebook	3484(0.38%)
No.40	QLogic	3394(0.37%)
No.41	ST Microelectronics	3188(0.35%)
No.42	Astaro	2981(0.32%)
No.43	NetApp	2860(0.31%)

Top Linux contributors since `git` (2005)

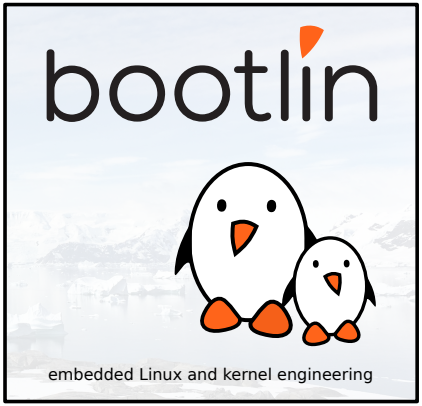
Bootlin on-line resources

- All our training materials and technical presentations: https://bootlin.com/docs/
- Technical blog: https://bootlin.com/
- Quick news (Mastodon): https://fosstodon.org/@bootlin
- Quick news (Twitter): https://twitter.com/bootlincom
- Quick news (LinkedIn): https://www.linkedin.com/company/bootlin
- Elixir - browse Linux kernel sources on-line: https://elixir.bootlin.com

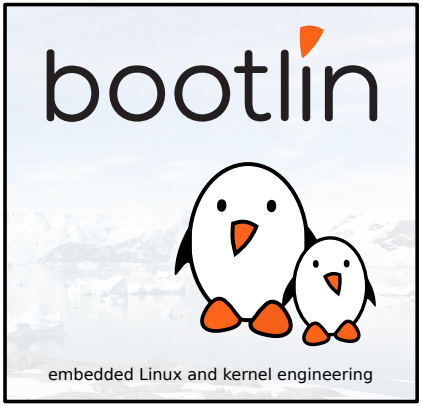
Mastodon is a free and decentralized social network created in the best interests of its users.

Image credits: Jin Nguyen - https://frama.link/bQwcWHTP

Generic course information



Realtime Systems



Realtime Operating System

A real-time system is a time-bound system which has well-defined, fixed time constraints. Processing must be done within the defined constraints or the system will fail.

Wikipedia

- The correctness of the program's computations is important
- The time taken to perform the computation is equally important

Determinism

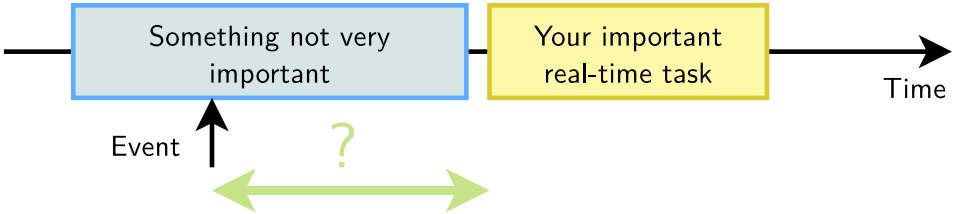
The same input must always yield the same output

- In an Realtime system, the timing for event and data processing must be consistent
- This is trivial on single-task single-core systems
- On multi-tasking systems, critical tasks should be deterministic
- The influence of CPU sharing and external interrupts must be fully predictable

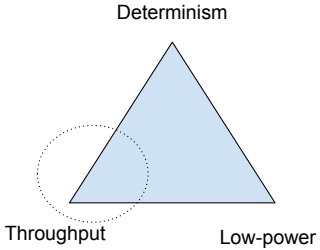
Latencies

Time elapsed between an event and the reaction to the event

- The Worst Case Execution Time is very difficult to predict
- We therefore want bounds on the Worst Case Reaction Time
- Latencies are the main focus for Realtime Operating Systems

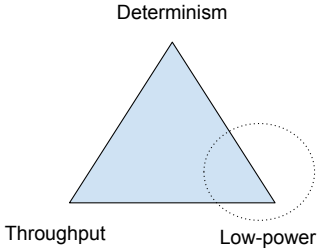


Design constraints - Throughput



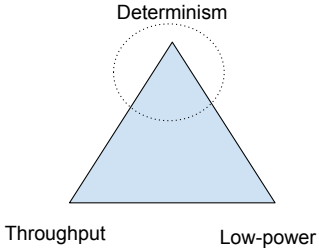
- Optimize most-likely scenario
- Might have a fast path and a slow path
- Use Hardware Offloading and caches
- Use Branch-Prediction and Speculative execution
- Latencies are acceptable for cold-start
- Most modern hardware implement such features

Design constraints - Low Power



- Oportunistic sleeping modes
- Dynamic Frequency Scaling
- Only go fast when required
- Long wakeup latencies
- Power-Management firmwares can preempt the whole system

Design constraints - Determinism



- Avoid unpredictable effects
- Caches, Hardware Offload are hard to predict
- Avoid sleeping too deep, to wakeup fast
- Make the system fully preemptible
- Try to keep control over every aspect of the system

Security Features and Fixes

- Hardware security flaws are discovered quite often
- Spectre, Meltdown, Foreshadow, Rowhammer
- Some can only be mitigated through software fixes...
- ... that can introduce some latencies
- In other cases, security features are actually beneficial for Realtime
- To mitigate timing-based attacks, making the system predictable is crucial
- **core scheduling** is also a good example, to deal with Hyperthreading issues

Multi-tasking

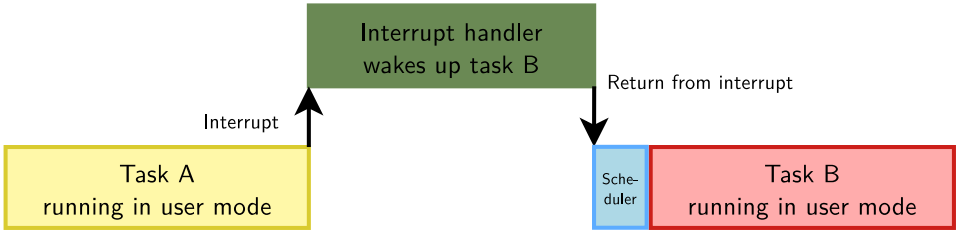
- Modern OSes are designed to be multi-task
- The CPU time is shared between applications
- The Scheduler decides who runs at any given time
- The scheduler is invoked at several occasions :
  - When an application waits for external data or events
  - When external data or events needs to be processed
  - Periodically, at every **System Tick** (between 300Hz and 1KHz)
  - Tickless systems are throughput oriented, but can also be useful for RT
- Switching between tasks is called **context switching**

Preemption

- Ability to stop whatever the CPU is running to run another task
- Useful for general-purpose OS, to share execution time
- Critical for an RTOS, to run critical tasks
- Any task should be preemptible, both in userspace and kernelspace

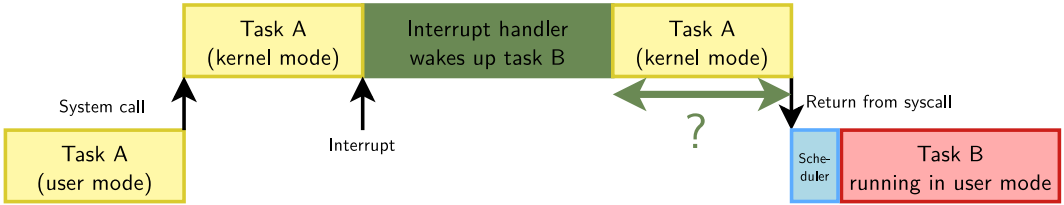
Understanding preemption (1)

- Most multi-tasking OSses are preemptive operating systems, including Linux
- 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.



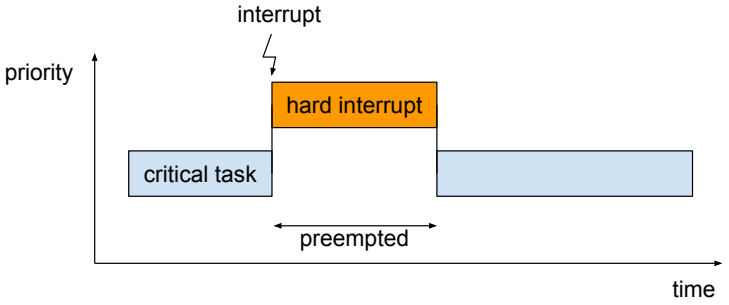
Understanding preemption (2)

- 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.



Interrupts and events

- Hardware interrupts are a common source of latencies
- Interrupts run in a dedicated context
- Other interrupts are disabled while the interrupt handler runs
- Non-important interrupts can preempt critical tasks



Scheduling and proritizing

- The Scheduler is a key component in guaranteeing RT behaviour
- There exist realtime and non-realtime scheduling algorithms
- Most realtime OSes rely on task prioritization
- Tasks with the same priority can be handled in a FIFO or Round-Robin manner

Locking

- Multitasking implies concurrent accesses to resources
- Critical resources must be protected by a dedicated mechanism
- Mutexes and semaphores help synchronise (or serialize) accesses
- This needs to be looked at closely in RT context
- A low-priority task migh hold a lock, blocking a high-priority task

- **mutex** : Two states (taken, free). The task that has taken the mutex is the **owner**
- **semaphore** : Shared variable that is incremented by multiple users.

Lock Families

Semaphores

- Semaphores rely on a **counter** that is positive or null
- A task trying to access the critical section decrements a counter
- A task is blocked if the counter can't be decremented
- Multiple tasks can be in a critical section, hence there's no single owner

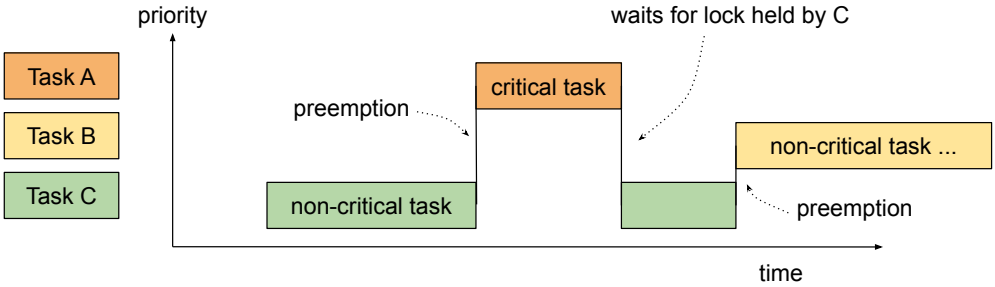
Mutexes

- **Mut**ually **ex**clusive
- Have two states : Taken, Free
- The task that has taken the mutex is the **owner**
- Other tasks wait for the mutex to be free before taking it
- A Mutex is a semaphore with a counter that can only be incremented once

Priority inversion

- Priority inversion arises when strict priority-based scheduling interfers with locking
- It creates a scenario where a critical task is prevented to run by a lower priority task

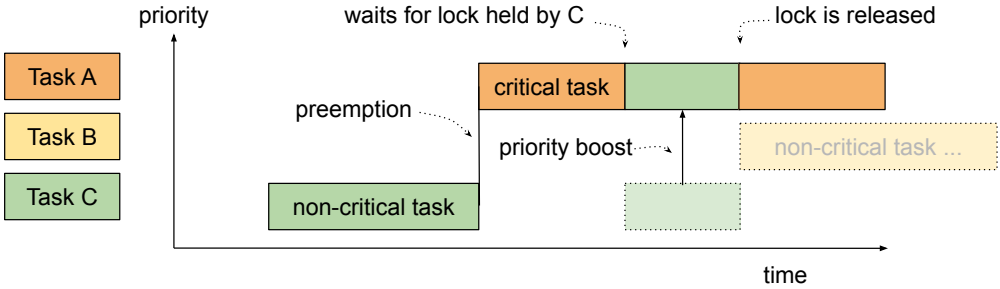
Priority inversion



- Task A (high priority) needs to access a lock, hold by task C (low priority)
- The scheduler runs task C so that it can release the lock
- Task B has a higher priority than C, but lower than A, preempts task C

Priority Inheritance

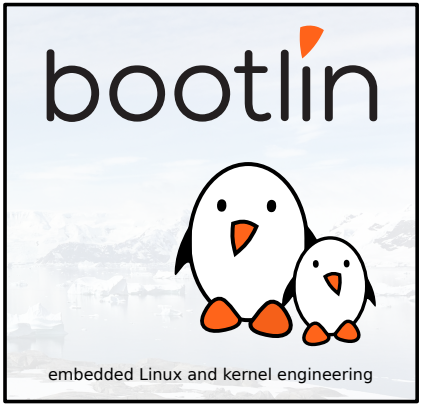
- The solution for the Priority Inversion issue is **priority inheritance**
- The scheduler detects that task C holds a lock needed by task A
- Task C's priority is boosted to A's priority until it releases the lock
- Task B can no longer preempt task C !



Priority Inheritance (2)

- **P**riority **I**nheritance (PI) only works with Mutexes
- Semaphores don't have owners, so we can't apply this mechanism
- Another way to prevent Priority Inversion is by careful design
- Limit critical section accesses only to tasks with the same priority
- PI support exists for `pthread_mutex_t` in Linux

Preempt RT

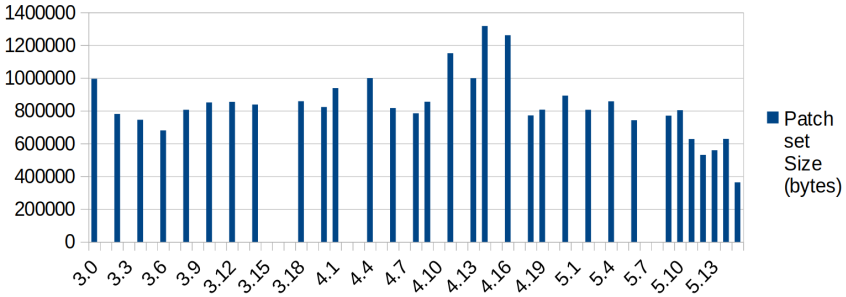


The PREEMPT_RT Patch

- One way to implement a multi-task Real-Time Operating System is to have a preemptible system
- Any task can be interrupted at any point so that higher priority tasks can run
- Userspace preemption already exists in Linux
- The Linux Kernel also supports real-time scheduling policies
- However, code that runs in kernel mode isn't fully preemptible
- The Preempt-RT patch aims at making all code running in kernel mode preemptible

PREEMPT_RT mainlining status (1)

- The project is making good progress since it got funding from the Linux Foundation in 2015 (Linux version 4.1 at that time).
- We can now observe a slow but steady reduction of the size of the PREEMPT_RT patchset:



PREEMPT_RT mainlining status (2)

- However, the mainline Linux kernel is a moving target too, introducing new issues for real-time (such as disabling preemption in BPF... see https://lwn.net/Articles/802884/).
- In 5.12, new CONFIG_PREEMPT_DYNAMIC switch to change the preemption model at boot time: preempt=none, preempt=voluntary or preempt=full
- In 5.15, the realtime preemption locking core is merged in mainline. This was the biggest part to mainline. A few pending changes are still left. See https://lwn.net/Articles/867919/.
- printk() is one of the remaining pieces to be upstreamed
- See the latest news on https://wiki.linuxfoundation.org/realtime/

Why use PREEMPT_RT ?

- Allow using the POSIX/Linux API, which is portable and familiar
- Benefit from the huge hardware support provided by Linux
- Run common software in non-RT mode, and custom critical software in parallel
- Benefit from the community support and help

Why not to use PREEMPT_RT ?

- Linux will never be a formally proven RTOS
- The hardware Linux typically runs on isn't designed with RT in mind
- The RT patch makes the Kernel deterministic and preemptible...
- ... but the goal is not to have the lowest latencies possible
- Linux will never be suitable for safety critical applications

Getting the patch

- The RT Patchset is released **every other** kernel release
- It can be downloaded as a set of patches or a single patch :
  https://cdn.kernel.org/pub/linux/kernel/projects/rt/
- Kernel trees with the patch applied are also available :
  - https://git.kernel.org/cgit/linux/kernel/git/rt/linux-rt-devel.git
  - https://git.kernel.org/cgit/linux/kernel/git/rt/linux-stable-rt.git
- Most build-systems like Buildroot or Yocto Project support building an RT Kernel

The legacy of the RT patch

Many current features in the Linux Kernel originated from the RT patch :

- Kernel-side preemption
- High Resolution Timers
- Threaded interrupts
- Priority-Inheritance support for locking primitives
- Tickless operation
- Earliest-Deadline First scheduler
- Realtime locks - Locking primitives conversion

Locking

- Locks are synchronisation primitives that arbitrate concurrent accesses to a resource
- Several locking primitives exist in the Kernel and in Userspace
- Kernel lock families are :
  - Sleeping locks
  - CPU Local locks
  - Spinning locks

Critical sections

- Spinning locks can be taken with interrupts constraints
- Spinlock functions have variants with some suffixes :
  - `_bh()`
    - Disable / Enable soft interrupts (bottom halves)
  - `_irq()`
    - Disable / Enable interrupts
  - `_irqsave()` / `_irqrestore()`
    - Save and disable or restore interrupt state (if previously disabled)
- Dedicated functions also exist, but should be used only for the Kernel core
- `preempt_disable()` / `preempt_enable()`
  - Disable / Enable preemption, to protect per-CPU data
- `migrate_disable()` / `migrate_enable()`
  - Disable / Enable migration, also to protect per-CPU data

Sleeping locks

- Sleeping locks will sleep and schedule while waiting
- There are several types of sleeping locks :
  - `mutex`
  - `rt_mutex`
  - `semaphore`
  - `rw_semaphore`

Spinlocks

- Spinlocks will busy-wait until the lock is freed
- There exist several types of spinlocks :
  - `spinlock_t`
  - `rwlock_t`
  - `raw_spinlock_t`
- Spinlocks will disable preemption when taken
- With PREEMPT_RT, `spinlock_t` and `rwlock_t` will become sleeping locks

High resolution timers

- The resolution of the timers used to be bound to the resolution of the regular system tick
  - Usually 100 Hz or 250 Hz, depending on the architecture and the configuration
  - A resolution of only 10 ms or 4 ms.
  - Increasing the regular system tick frequency is not an option as it would consume too many resources
- The high-resolution timers infrastructure 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

Printk

- `printk()` is one of the main logging mechanism in the kernel
- It works in all execution context
- It's currently one of the last items preventing the RT-patch from being fully upstream
- Currently, the last kernel task who printed must print the full buffer
- A low priority task can block a high priority task by printing lots of data
- A low log-level can help remove these issues for now

Interrupt handlers

- Interrupt handlers run with interrupts disabled
- In PREEMPT_RT, almost all interrupt handlers are **threaded**
- Very small hardware interrupt handlers are used, that have a well-defined execution time
- They acknowledge the interrupts, and enqueues the "real" interrupt handler
- The interrupt handler runs in a dedicated Kernel thread
- Threaded interrupts are well established in the mainline kernel

Hard interrupts vs. Threaded interrupts

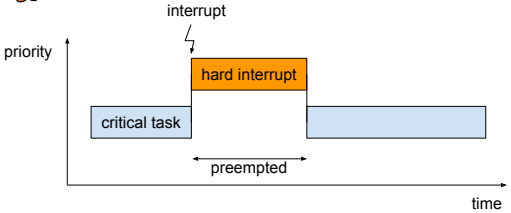


Figure: Hardware interrupt processing

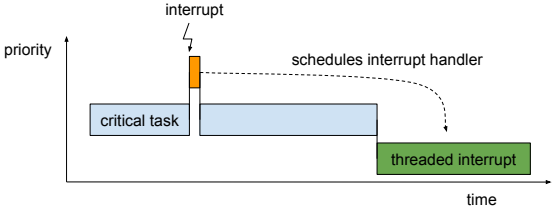


Figure: Threaded interrupt processing

- Small, well-defined hard irq handlers
- Irq handlers run in a dedicated task
- It has a PID, and can be assigned a priority
- Critical tasks can run regardless of interrupts
- Use `ps -e` to list tasks
- Use `chrt -p <prio> <pid>` to change the priority

Uncompatible options

- Some configuration options don't play well with realtime
- CONFIG_LOCKUP_DETECTOR and CONFIG_DETECT_HUNG_TASK
  - Kernel tasks with a priority of 99, can introduce latencies
- CONFIG_DEBUG_*
  - Debugging options are very useful
  - Most of them introduce latencies due to heavy logging
  - Some options adds security checks and verifiers (lockdep)

Preemption models

The Linux kernel Scheduler has several preemption models available :

- `CONFIG_PREEMPT_NONE` - No Forced Preemption (server)
- `CONFIG_PREEMPT_VOLUNTARY` - Voluntary Kernel Preemption (Desktop)
- `CONFIG_PREEMPT` - Preemptible Kernel (Low-latency Desktop)
- `CONFIG_PREEMPT_RT` - Fully Preemptible Kernel (Real-Time)

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 real-time 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 (several possible values between 100 Hz and 1000 Hz).

2nd option: voluntary kernel preemption

`CONFIG_PREEMPT_VOLUNTARY`

Kernel code can preempt itself

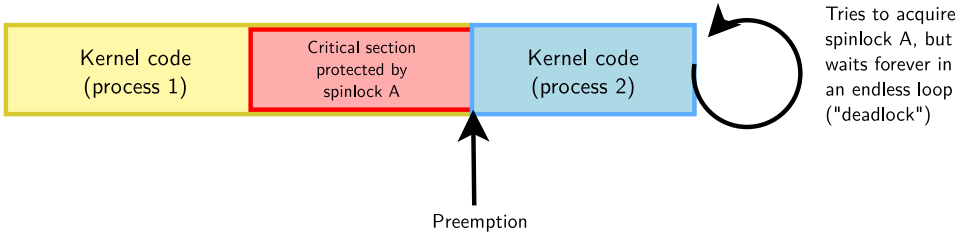
- Typically for desktop systems, for quicker application reaction to user input.
- Adds explicit rescheduling points (`might_sleep()`) throughout kernel code.
- Minor impact on throughput.
- Still used in: Ubuntu Desktop 20.04

3rd option: preemptible kernel

CONFIG_PREEMPT

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

- Exception: kernel critical sections (holding spinlocks):



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

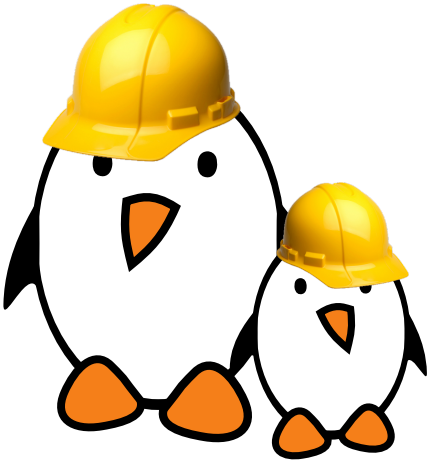
4th option: fully preemptible kernel

`CONFIG_PREEMPT_RT`

Almost all kernel code can be involuntarily preempted at any time.

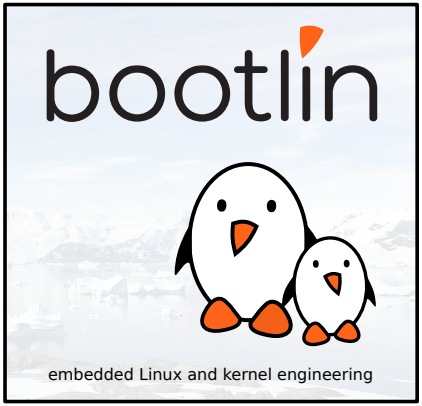
- spinlocks are turned into sleeping locks
- Only `raw_spinlock_t` remains a real spinning lock
- All interrupt handlers are threaded, except for a few that explicitely need hard irq
  - This is the case for drivers involved in interrupt dispatching
  - cpufreq and cpuidle drivers too
- For use on systems with Realtime requirements
- If you find a kernel-side unbounded latency, **this is a bug**

Practical lab - Download and Boot a RT Kernel



- Generate a Linux Image with Buildroot
- Configure the PREEMPT_RT Scheduling Model
- Check that we boot a Realtime Linux Kernel

Hardware



Hardware latencies

The hardware itself can be the source of latencies :

- Power-management features uses sleep states that introduce latencies
- Throughput features introduce lots of cache levels that are hard to predict
- Even CPU features like branch-prediction introduce latencies
- Hardware latencies are nowadays unavoidable, but some can be mitigated
- It's important to benchmark the hardware platform early during development

Non-Maskable Interrupts

Non-Maskable Interrupts can't be disabled, and are often transparent to the OS

- Common NMIs are System Management Interrupts (SMIs)
- SMIs run in a dedicated context, the System Management Mode
- It often runs low-level firmware code, like BIOS and EFI
- Used for thermal management, remote configuration, and are very opaque
- Can't be prevented, predicted, monitored or controled
- Modern CPUs can expose a NMI counter
- `hwlatdetect` can help measure the NMIs on a given system

Deep Firmwares

On modern hardware, several firmwares can run outside the control of the Kernel

- The ARM TrustZone can run firmware like OP-TEE, to handle sensitive tasks
- Low-level firmware calls can be made by drivers through SMC calls or similar
- These firmwares can disable interrupts while they run
- Depending on the SoC, we might or might not have control over these firmwares
- `hwlatdetect` and careful driver analysis can help identify these issues

Memory access

Accessing a virtual address can trigger a Page Fault if :

- The page isn't mapped yet
- The address is invalid

Programs should first access all pages to populate the page-table, then pin them with `mlockall()`

On a smaller scale, caching can also impact memory access time, and be affected by processes running on other CPUs

Hyperthreading

Some CPU cores have 2 pipelines feeding the same ALU. This is known as **Hyperthreading**

- The ALU executes instructions for one pipeline while the other fetches instructions
- This maximizes CPU usage and throughput
- Hyperthreads are very sensitive to what the co-thread runs
- Usually we recommend that hyperthreading is disabled for RT

Core Scheduling can help with hyperthreading

- Recent kernel development introduced **core scheduling**, in v5.14
- The scheduler is aware of hyperthreads, mostly for security reasons
- RT tasks won't have sibling processes scheduled on the same core
- Non-RT process can still benefit from hyperthreading

IO Memory and DMA

- When writing drivers that are RT-critical, several considerations should be taken
- Some memory busses can buffer accesses and create latency spikes
- PCI accesses can buffer writes until the next read
- Some control busses such as `i2c` can be shared between devices
- DMA accesses can also introduce latencies due to bus mastering

NUMA

- **N**on **U**niform **M**emory **A**ccess
- For high-end machines and servers, there are several banks of memory called **nodes**
- Typically, nodes are closer to some CPUs than others
- The Kernel can migrate pages from one node to another if need be
- Access latency will be longer for distant nodes
- Critical applications must have their memory locked to avoid migration
- Use `numactl` to pin nodes to CPU cores, and pin your application on the CPU

CPU Idle

- Modern CPUs have several Idle States for better power management
- The deeper the CPU core is sleeping, the longer it takes to wake it up
- Idle states are often called **C-States**
- Other Idle states can also exists, such as **PC-States** on a per-package basis
- We can limit the CPU idle states to lightweight states
- This can have a big impact on power consumption and thermal management

Idle States

C-States are defined by :

- `latency` : The time it takes to wake-up
- `residency` : Expected sleeping time for which the state can be used
- `power` : The power consumption in this C-State

The `POLL` state means that the CPU stays in a busy-loop instead of sleeping

Idle states, Intel i7-8550U

state	latency	residency
POLL	0 µs	0 µs
C1	2 µs	2 µs
C1E	10 µs	20 µs
C3	70 µs	100 µs
C6	85 µs	200 µs
C7s	124 µs	800 µs
C8	200 µs	800 µs
C9	480 µs	5000 µs
C10	890 µs	5000 µs

C-States can be controlled in `/sys/devices/system/cpu/cpuX/cpuidle/`

Limiting the Idle states

- Limiting the idle states can be done at runtime
  - `echo 1 > /sys/devices/system/cpu/cpu0/cpuidle/stateX/disable`
- C-States can be also limited at boot-time with boot options :
  - `processor.max_cstates=1` : Limits the deepest sleep state
  - `idle=poll` : Only use polling, never go to sleep
- C-States can also be temporarily limited by an application :
  - While `/dev/cpu_dma_latency` is opened, deep C-Staets won't be used
  - Writing `0` to this file and maintaining it opened emulates `idle=poll`
- **Be careful**, using the `POLL` idle state can overheat and destroy your CPU !

CPU Frequency scaling

The CPU frequency can also be dynamically changed through DVFS

- Dynamic Voltage and Frequency Scaling
- The frequency can be controlled by the kernel by picking a governor
- The governor selects one of the available **Operating Performance Points**
- An **OPP** defines a frequency and voltage at which a core can run
- The `performance` governor always uses the highest frequency
- The `powersave` governor uses the lowest frequency
- Other governors can adjust the frequency dynamically
- Adjusting the frequency causes non-deterministic execution times
- `cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor`
- The governor can also be picked in the Kernel Configuration

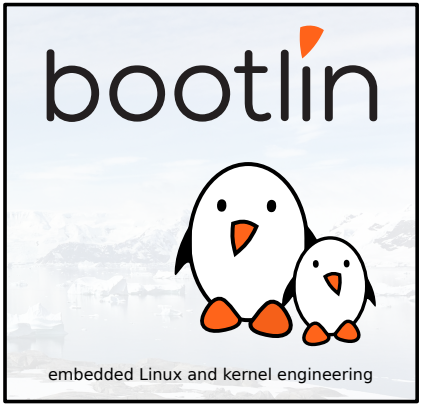
Powertop is a tool to monitor the CPU idle states and frequency usage

- It is designed to optimize the power usage of a system
- Useful to undertstand which C-States and OPP are being used

PowerTOP 2.13 Overview **Idle stats** Frequency stats Device stats Tunables WakeUp

Pkg(HW)		Core(HW)		CPU(OS) 0	CPU(OS) 4		
			C0 active	16,8%	18,1%		
			POLL	0,0%	0,0 ms	0,1%	0,0 ms
			C1	0,3%	0,1 ms	15,7%	0,1 ms
C2 (pc2) 3,3%							
C3 (pc3) 1,9%	C3 (cc3)	3,2%	C3	3,0%	0,2 ms	2,3%	0,2 ms
C6 (pc6) 0,4%	C6 (cc6)	5,2%	C6	7,5%	0,5 ms	5,1%	0,3 ms
C7 (pc7) 0,0%	C7 (cc7)	28,0%	C7s	0,0%	0,1 ms	0,0%	0,1 ms
C8 (pc8) 0,0%			C8	16,0%	0,8 ms	8,2%	0,4 ms
C9 (pc9) 0,0%			C9	1,5%	1,1 ms	2,3%	0,2 ms
C10 (pc10) 0,0%							
			C10	50,7%	1,8 ms	42,6%	0,2 ms
			C1E	4,7%	0,1 ms	4,4%	0,1 ms

Testing and Benchmarking



Benchmarking vs Testing

- **Benchmarking** will give you system-wide metrics for :
  - Your hardware platform
  - Your Kernel configuration
  - Your Non-critical userspace stack
- **Testing** will ensure that your business application behaves correctly
- Stressing tools used for benchmarking can also be used for testing
- It's important to always consider the **Worst Case Scenario**

ftrace - Kernel function tracer

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

- Very well documented in `trace/ftrace`
- Negligible overhead when tracing is not enabled at run-time.
- Can be used to trace any kernel function!

Using ftrace

- Tracing information available through the `tracefs` virtual fs
- Mount this filesystem as follows:
  `mount -t tracefs nodev /sys/kernel/tracing`
- Check available tracers in `/sys/kernel/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 us.
- Check that `wakeup` is listed in `/sys/kernel/tracing/available_tracers`
- To select, reset and enable this tracer:

```
echo wakeup > /sys/kernel/tracing/current_tracer
echo 0 > /sys/kernel/tracing/tracing_max_latency
echo 1 > /sys/kernel/tracing/tracing_enabled
```

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

```
echo 0 > /sys/kernel/tracing/tracing_enabled
```

- Read the maximum recorded latency and the corresponding trace:

```
cat /sys/kernel/tracing/tracing_max_latency
```

kernelshark

- Kernelshark is a graphical interface for processing ftrace reports
- It's better used with `trace-cmd`, an interface to ftrace
- Useful when a deep analysis is required for a specific bug
- `trace-cmd list`
- `trace-cmd record -e <event> [<command>]`
- `kernelshark`

hwlatdetect

Tool provided by `rt-tests`, relying on a dedicated kernel feature

- Allows detecting NMIs, that the kernel is normally unaware of
- Runs a loop on all CPU cores with local interrupts disabled
- Only NMIs can interrupt the loop
- Samples a high precision timer and looks for large latencies
- Useful to benchmark and validate a hardware platform
- Must **not** be used in production environment, introduces huge latencies

cyclictest

- Tool that tests the System and Kernel Latencies
- Provided by the `rt-test` suite
- Schedules timer events and compares the expected and actual wakeup time
- Measures the kernel-induced latencies, but also hardware-induced latencies
- Can create graphs, and be used with tracing subsystems
- Best used in conjunction with various stressing workloads
- Should be run for long amounts of time

hackbench

Stress and benchmark the Linux Kernel Scheduler

- Stresses the scheduler by creating lots of processes of threads
- They communicate with each-other through sockets or pipes
- This generates lots of context switches and scheduling events
- Useful to check how a RT program behaves when heavy workloads run in parallel

stress-ng

Very feature-full stressing utility, with more than 260 stressors

- Can stress very specific aspects of the sytem :
  - Specific syscalls
  - CPU instructions and computations
  - Caches, Memory access, Page-faults
  - Network and Filesytem stacks
- Very useful to accurately simulate known workloads

Benchmarking an application

strace

strace is a userspace tool that trace system-calls and signals

- Help analyse how an application interacts with the kernel
- Some syscalls can be detrimental to RT behaviour
- strace can help understand what an application does
- Helpful for external libraries

perf

perf is a performance analysis tool that gathers kernel and hardware statistics

- Uses Hardware Counters and monitoring units
- Uses Kernel Counters and the tracing infrastructure
- Can profile the whole system, an application, a CPU core, etc.
- Very versatile, but tied to the kernel version
- Perf relies on various **events** reported by the kernel

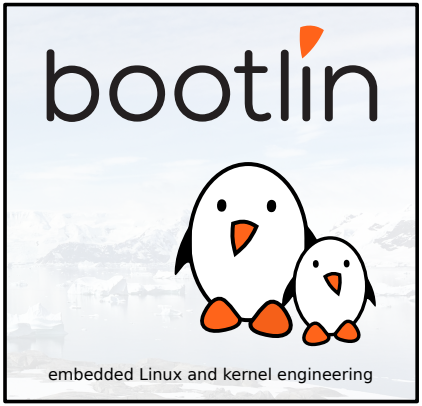
Using perf, examples

- Lots of events measurable : hardware events, software events, cache misses, power management
  - `perf list`
- Display statistics in real-time
  - `perf top`
  - `perf top -e cache-misses`
  - `perf top -e context-switches`
- Investigate scheduling latencies
  - `perf sched record`
  - `perf sched latency`

Other useful tools

- `vmstat` : Displays the system state, interrupts, context switches...
  - `vmstat -w 0`
- `powertop` : Display the CPU usage
- `cat /sys/kernel/realtime` : Indicates if the RT Patch is applied
- `htop` : Displays running tasks, including kernel tasks

Configuring the system



Configuration

- The Linux Kernel has a lot of available configurations
- Some are dedicated to performance
  - Focus on improving the most-likely scenario
  - Have the best average performance
  - Optimize throughput and low-latency
  - Usually have a slow-path, which is non-deterministic
- Some are dedicated to security and hardening
- Some can be useful for Deterministic Behaviour

CPU Pinning

- The Linux Kernel Scheduler allows setting constraints about the CPU cores that are allowed to run each task
- This can be useful for lots of purposes :
  - Make sure that a process won't be migrated to another core
  - Dedicate cores for specific tasks
  - Optimize the data-path if a process deals with data handled by a specific CPU core
  - Ease the job of the scheduler's CPU load-balancer, whose complexity grows non-linearly with the number of CPUs
- This mechanism is called the `cpu affinity` of a process
- The `cpuset` subsystem and the `sched_setaffinity` syscall are used to select the CPUs
- Use `taskset -p <mask> <cmd>` to start a new process on the given CPUs

CPU Isolation

- Users can pin processes to CPU cores through the cpu affinity mechanism
- But the kernel might also schedule other processes on these CPUs
- `isolcpus` can be passed on the kernel commandline
- Isolated CPUs will not be used by the scheduler
- The only way to run processes on these CPUs is with cpu affinity
- Very useful when RT processes coexist with non-RT processes

```
isolcpus=0,2,3
```

IRQ affinity

- Interrupts are handled by a specific CPU core
- The default CPU that handles interrupts is the CPU 0
- On Multi-CPU systems, it can be good to balance interrupt handling between CPUs
- Similarly, we might also want to prevent CPUs from handling external interrupts
- IRQs can be pinned to CPUs by tweaking `/proc/irq/XX/smp_affinity`
- The `irqbalance` tool monitors and distributes the irq affinty to spread the load across CPUs
- Use the `IRQBALANCE_BANNED_CPUS` environment variable to make `irqbalance` ignore some CPUs

The Linux Kernel Scheduler

- The Linux Kernel Scheduler is a key piece in having a real-time behaviour
- It is in charge of deciding which **runnable** task gets executed
- It also elects on which CPU the task runs, and it tightly coupled to CPUidle and CPUFreq
- It schedules both **userspace** tasks, but also **kernel** tasks
- Each task is assigned one **scheduling class** or **policy**
- The class determines the algorithm used to elect each task
- Tasks with different scheduling classes can coexist on the system

Non-Realtime Scheduling Classes

There are 3 **Non-RealTime** classes

- `SCHED_OTHER` : The default policy, using a time-sharing algorithm
- `SCHED_BATCH` : Similar to `SCHED_OTHER`, but designed for CPU-intensive loads that affect the wakeup time
- `SCHED_IDLE` : Very low priority class. Tasks with this policy will run only if nothing else needs to run.
- `SCHED_OTHER` and `SCHED_BATCH` use the **nice** value to increase or decrease their scheduling frequency
- A higher nice value means that the tasks gets scheduled **less** often

Realtime Scheduling Classes

There are 3 **Realtime** classes

- Tasks under a Realtime class can be assigned a priority between 0 and 98
- Priority 99 is **reserved** for critical housekeeping tasks
- Runnable tasks will preempt any other lower-priority task
- `SCHED_FIFO` : All tasks with the same priority are scheduled **First in, First out**
- `SCHED_RR` : Similar to SCHED_FIFO but with a time-sharing round-robin between tasks with the same priority
- `SCHED_DEADLINE` : For tasks doing recurrent jobs, extra attributes are attached to a task
  - A computation time, which represents the time the tasks needs to complete a job
  - A deadline, which is the maximum allowable time to compute the job
  - A period, during which only one job can occur

Changing the Scheduling Class

- The Scheduling Class is set per-task, and defaults to `SCHED_OTHER`
- The `sched_setscheduler` syscall allows changing the class of a task
- The tool `chrt` uses it to allow changing the class of a running task :
  - `chrt -f/-b/-o/-r -p PRIO PID`
- It can also be used to launch a new program with a dedicated class :
  - `chrt -f/-b/-o/-r PRIO CMD`
- New processes will inherit the class of their parent except if the `SCHED_RESET_ON_FORK` flag is set with `sched_setscheduler`

Realtime policies and CPU hogging

- A bug or an infinite loop in a high-priority realtime task will cause the system to become unresponsive
- This can be prevented by creating an emergency shell with a higher priority to kill the faulty programs
- The Kernel also provides a mechanism to make sure non-RT tasks get to run at one point
- `/proc/sys/kernel/sched_rt_period_us` defines a window in microseconds that the scheduler will share between RT and non-RT tasks
- `/proc/sys/kernel/sched_rt_runtime_us` defines how-much of that window is going to be dedicated to RT tasks.
- The default values allocates 95% of the CPU time to RT tasks

System timer

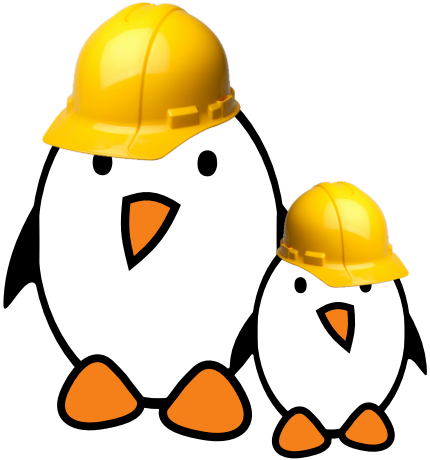
- The Scheduler is invoked on a regular basis to perform time-sharing activities
- This is sequenced through the **system ticks**, generated by a high resolution timer
- Typically run at 250 Hz (x86) or 100 Hz (arm), but can be configured
- Several policies regarding system ticks are available :
- `CONFIG_HZ_PERIODIC` : Always tick at a given rate. This can introduce small latencies but keep the system responsive.
- `CONFIG_NO_HZ_IDLE` : Disable the tick when idle, for powersasving
- `CONFIG_NO_HZ_FULL` : Actively disables ticking even when not idle. Introduces long latencies during context switches.

Writing a driver

A few considerations can be taken when writing a driver

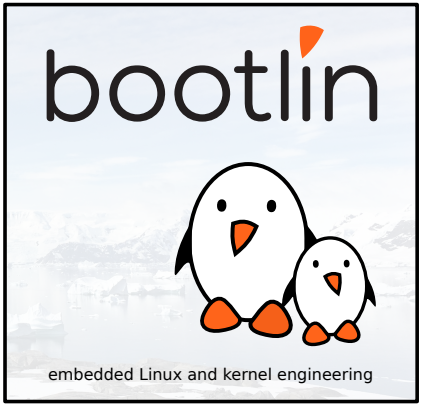
- Avoid using `raw_spinlock_t` unless really necessary
- Avoid forcing non-threaded interrupts, unless writing a driver involved in interrupt dispatch
  - irqchip, gpio-irq drivers
  - cpufreq and cpuidle drivers due to scheduler interaction
- Beware of DMA bus mastering and other serialized IO buffering
  - Certain register writes are buffered until the next register read

Practical lab - Benchmark and tweak your system



- Measure the system latencies
- Stress the system and try to improve the latencies
- Tweak the system to get the best behaviour

Application development



Real-time application development

- A few best-practices should be followed when developping a real-time application
- Some POSIX APIs weren't designed with RT behaviour in mind
- Some syscalls and memory-access patterns will lead to kernel-side latencies
- Following the good practises is important
- But benchmarking the application is also crucial

Initialization

- Usually, the initialisation section of the application doesn't need to be RT
- This init section will configure various settings :
  - Allocate and lock the memory
  - Start the threads and configure them
  - Initialize the locks
  - Configure the scheduling parameters (priority, deadlines)
  - Configure the CPU affinity

Development and compilation

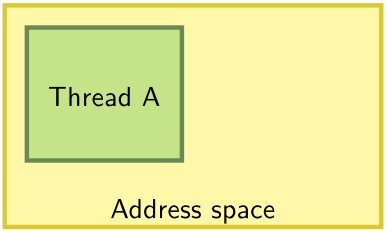
- No special library is needed, the POSIX real-time API is part of the standard C library
- The glibc C library is recommended, as support for some real-time features is not mature in other C libraries
  - Priority inheritance mutexes or NPTL on some architectures, for example
- Compile a program
  -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

Process, thread?

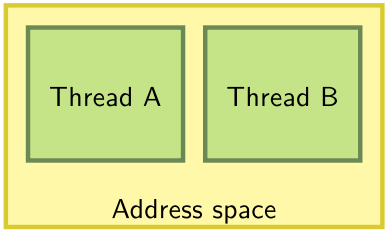
- Confusion about the terms *process*, *thread* and *task*
- In UNIX, a process is created using `fork()` and is composed of
  - An address space, which contains the program code, data, stack, shared libraries, etc.
  - One thread, that starts executing the `main()` function.
  - Upon creation, a process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



Process after pthread_create()

Creating threads

- Linux supports 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);
```

Using scheduling classes (1)

- 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`
    `-f`: `SCHED_FIFO`
    `-r`: `SCHED_RR`
    `-d`: `SCHED_DEADLINE`
- The `sched_setscheduler()` API can be used to change the scheduling class and priority of a threads

```
int sched_setscheduler(pid_t pid, int policy,
                       const struct sched_param *param);
```

  - `policy` can be `SCHED_OTHER`, `SCHED_FIFO`, `SCHED_RR`, `SCHED_DEADLINE`, etc. (others exist).
  - `param` is a structure containing the priority

Using scheduling classes (2)

- 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 management

- When asking the Kernel to allocate some memory, a virtual address range is given to the application
- This virtual range is mapped to a physical memory range **upon accessing it**
- This will introduce a **page fault** latency for the application
- All memory should be **locked** and **accessed** at application initialization
- Accessing memory at init is called **prefaulting** the memory
- This concerns the heap memory, but also the stack

Memory management

- Call `mlockall()` at init to lock all memory regions
- Allocate all memory and acces it at initialization
- Don't call `fork()`, since the child will copy-on-write pages
- Avoid using mmap'd memory, since ranges aren't reused after free
- malloc's behaviour can be tuned not to use mmap : `mallopt(M_MMAP_MAX, 0)`

Locking

- When creating multi-threaded applications, use `pthread_mutex_t`
- Avoid using semaphores, which don't have an owner
- These are POSIX mutexes, which have a notion of **ownership**
- Ownership allows to handle **Priority Inheritance** (PI)
- PI needs to be explicitely enabled :
  `pthread_mutexattr_setprotocol(&mattr, PTHREAD_PRIO_INHERIT);`

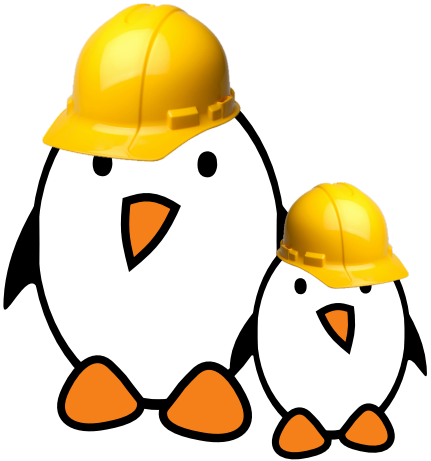
Synchronizing and signaling

- Application might need to wait or react to external events
- Inter-thread signaling should be done with `pthread_cond_wait()`
- Conditions can be attached to mutexes
- Avoid using UNIX Signals

timekeeping

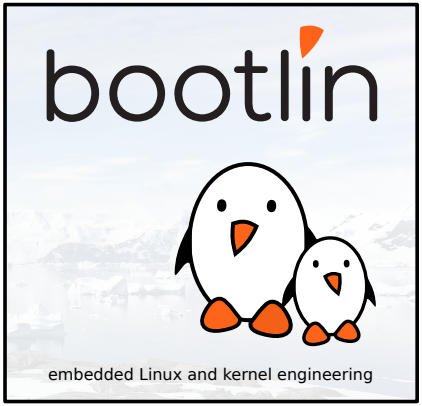
- Usually, real-time applications will need timing information
- This can be done by using `clock_gettime(clk_id, &ts)`
- Although counter-intuitive, don't use the `CLOCK_REALTIME` clock id
- `CLOCK_REALTIME` gives the current time, which can be adjusted and is non-consistent
- Instead, use `CLOCK_MONOTONIC` which is never adjusted and strictly increasing

Practical lab - Build, Analyse and Improve a Realtime Application



- Build and run an example application
- Investigate possible problems
- Use the best coding practices
- Adapt the system configuration for a specific use-case

Last slides



Last slide

Thank you!

And may the Source be with you