



Understanding Linux real-time with PREEMPT_RT training

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Latest update: November 05, 2025.

Document updates and training details:
<https://bootlin.com/training/preempt-rt>

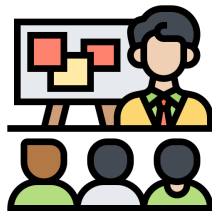
Corrections, suggestions, contributions and translations are welcome!
Send them to feedback@bootlin.com





Understanding Linux real-time with PREEMPT_RT training

- ▶ These slides are the training materials for Bootlin's *Understanding Linux real-time with PREEMPT_RT* training course.
- ▶ If you are interested in following this course with an experienced Bootlin trainer, we offer:
 - **Public online sessions**, opened to individual registration. Dates announced on our site, registration directly online.
 - **Dedicated online sessions**, organized for a team of engineers from the same company at a date/time chosen by our customer.
 - **Dedicated on-site sessions**, organized for a team of engineers from the same company, we send a Bootlin trainer on-site to deliver the training.
- ▶ Details and registrations:
<https://bootlin.com/training/preempt-rt>
- ▶ Contact: training@bootlin.com



Icon by Eucalyp, Flaticon



About Bootlin

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





Bootlin introduction

- ▶ Engineering company
 - In business since 2004
 - Before 2018: *Free Electrons*
- ▶ Team based in France and Italy
- ▶ Serving **customers worldwide**
- ▶ **Highly focused and recognized expertise**
 - Embedded Linux
 - Linux kernel
 - Embedded Linux build systems
- ▶ **Strong open-source** contributor
- ▶ Activities
 - **Engineering** services
 - **Training** courses
- ▶ <https://bootlin.com>

bootlin



Bootlin engineering services

Bootloader / firmware development

U-Boot, Barebox,
OP-TEE, TF-A, .../

Linux kernel porting and driver development

Linux BSP development, maintenance and upgrade

Embedded Linux build systems

Yocto, OpenEmbedded,
Buildroot, ...

Embedded Linux integration

Boot time, real-time,
security, multimedia,
networking

Open-source upstreaming

Get code integrated
in upstream
Linux, U-Boot, Yocto,
Buildroot, ...



Bootlin training courses

Embedded Linux system development

On-site: 4 or 5 days
Online: 7 * 4 hours

Linux kernel driver development

On-site: 5 days
Online: 7 * 4 hours

Yocto Project system development

On-site: 3 days
Online: 4 * 4 hours

Buildroot system development

On-site: 3 days
Online: 5 * 4 hours

Embedded Linux networking

On-site: 3 days
Online: 4 * 4 hours

Understanding the Linux graphics stack

On-site: 2 days
Online: 4 * 4 hours

Embedded Linux audio

On-site: 2 days
Online: 4 * 4 hours

Real-Time Linux with PREEMPT_RT

On-site: 2 days
Online: 3 * 4 hours

Linux debugging, tracing, profiling and performance analysis

On-site: 3 days
Online: 4 * 4 hours

All our training materials are freely available
under a free documentation license (CC-BY-SA 3.0)
See <https://bootlin.com/training/>



Bootlin, an open-source contributor

- ▶ Strong contributor to the **Linux** kernel
 - In the top 30 of companies contributing to Linux worldwide
 - Contributions in most areas related to hardware support
 - Several engineers maintainers of subsystems/platforms
 - 9000 patches contributed
 - <https://bootlin.com/community/contributions/kernel-contributions/>
- ▶ Contributor to **Yocto Project**
 - Maintainer of the official documentation
 - Core participant to the QA effort
- ▶ Contributor to **Buildroot**
 - Co-maintainer
 - 6000 patches contributed
- ▶ Significant contributions to U-Boot, OP-TEE, Barebox, etc.
- ▶ Fully **open-source training materials**



Bootlin on-line resources

- ▶ Website with a technical blog:
<https://bootlin.com>
- ▶ Engineering services:
<https://bootlin.com/engineering>
- ▶ Training services:
<https://bootlin.com/training>
- ▶ LinkedIn:
<https://www.linkedin.com/company/bootlin>
- ▶ Elixir - browse Linux kernel sources on-line:
<https://elixir.bootlin.com>



Icon by Freepik, Flaticon

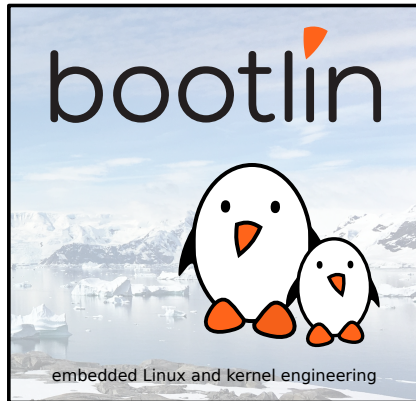


Generic course information

© Copyright 2004-2025, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Supported hardware

Discovery Kits from STMicroelectronics: STM32MP157A-DK1, STM32MP157D-DK1, STM32MP157C-DK2 or STM32MP157F-DK2

- ▶ STM32MP157 (Dual Cortex-A7 + Cortex-M4) CPU from STMicroelectronics
- ▶ 512 MB DDR3L RAM
- ▶ Gigabit Ethernet port
- ▶ 4 USB 2.0 host ports, 1 USB-C OTG port
- ▶ 1 Micro SD slot
- ▶ On-board ST-LINK/V2-1 debugger
- ▶ Misc: buttons, LEDs, audio codec
- ▶ LCD touchscreen (DK2 only)



DK1 Discovery Kit

Board and CPU documentation, design files, software: [A-DK1](#), [D-DK1](#), [C-DK2](#), [F-DK2](#)



Training quiz and certificate

- ▶ You have been given a quiz to test your knowledge on the topics covered by the course. That's not too late to take it if you haven't done it yet!
- ▶ At the end of the course, we will submit this quiz to you again. That time, you will see the correct answers.
- ▶ It allows Bootlin to assess your progress thanks to the course. That's also a kind of challenge, to look for clues throughout the lectures and labs / demos, as all the answers are in the course!
- ▶ Another reason is that we only give training certificates to people who achieve at least a 50% score in the final quiz **and** who attended all the sessions.



Participate!

During the lectures...

- ▶ Don't hesitate to ask questions. Other people in the audience may have similar questions too.
- ▶ Don't hesitate to share your experience too, for example to compare Linux with other operating systems you know.
- ▶ Your point of view is most valuable, because it can be similar to your colleagues' and different from the trainer's.
- ▶ In on-line sessions
 - Please always keep your camera on!
 - Also make sure your name is properly filled.
 - You can also use the "Raise your hand" button when you wish to ask a question but don't want to interrupt.
- ▶ All this helps the trainer to engage with participants, see when something needs clarifying and make the session more interactive, enjoyable and useful for everyone.



Collaborate!

As in the Free Software and Open Source community, collaboration between participants is valuable in this training session:

- ▶ Use the dedicated Matrix channel for this session to add questions.
- ▶ If your session offers practical labs, you can also report issues, share screenshots and command output there.
- ▶ Don't hesitate to share your own answers and to help others especially when the trainer is unavailable.
- ▶ The Matrix channel is also a good place to ask questions outside of training hours, and after the course is over.

embedded-linux-nov2020 Channel for

Srinath

michael What should be CROSS_COMPILE variable set to in case of the Xplained board? I ran into some issues with my USB hub so doing the u-boot again

michael

Srinath you should look at the name of the cross-compiler in the toolchain's bin/ directory. CROSS_COMPILE should be set to what's before "gcc" in the name, including the trailing "-". Like if the compiler is arm-buildroot-linux-gcc, CROSS_COMPILE should be arm-buildroot-linux-

2 messages deleted.

Srinath

Will ask them here since I am going to do labs after the session is over! Thank!

Srinath changed their display name to **srinath.t**

@srinath.t

I tried to finalize Kernel - Cross-compiling task, but my system is not able to restart the new kernel. Does anyone know what can be the root cause?

20:47

Decrypt image.png (109.2 KB)

arnaud.a

I had the same because I accidentally renamed the `membase` from the kernel

Send an encrypted message...



Prepare your lab environment

- ▶ Download and extract the lab archive

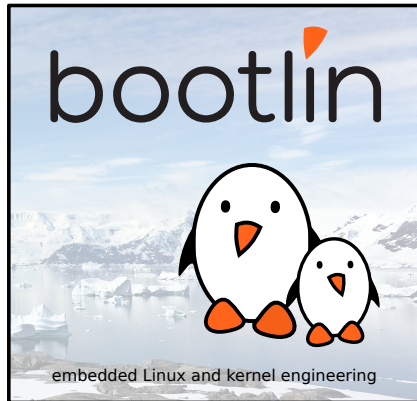


Realtime Systems

© Copyright 2004-2025, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





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



The same input must always yield the same output

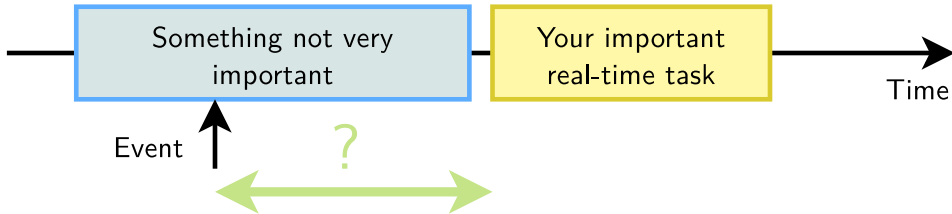
- ▶ In an Realtime system, the timing for event and data processing must be consistent
- ▶ This is trivial on single-task single-core systems
- ▶ On multi-tasking systems, critical tasks should be deterministic
- ▶ The influence of CPU sharing and external interrupts must be fully predictable



Latencies

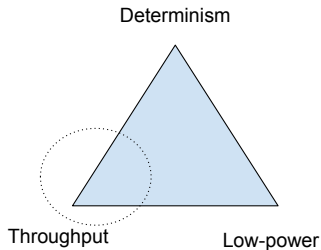
Time elapsed between an event and the reaction to the event

- ▶ 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





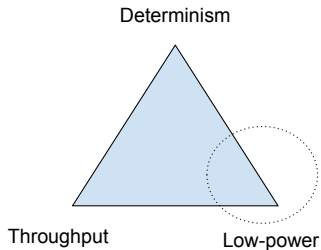
Design constraints - Throughput



- ▶ 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



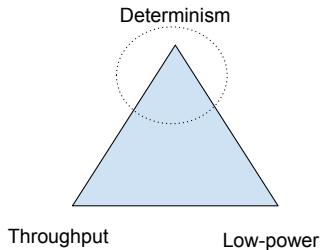
Design constraints - Low Power



- ▶ Opportunistic sleeping modes
- ▶ Dynamic Frequency Scaling
- ▶ Only go fast when required
- ▶ Long wakeup latencies
- ▶ Power-Management firmware 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**



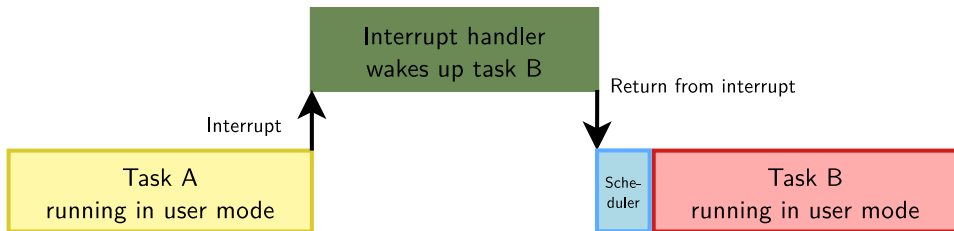
Preemption

- ▶ 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)

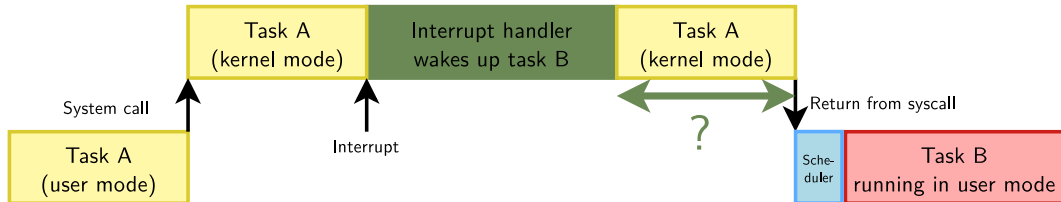
- ▶ Most multi-tasking OSes are preemptive operating systems, including Linux
- ▶ When a task runs in user space mode and gets interrupted by an interrupt, if the interrupt handler wakes up another task, this task can be scheduled as soon as we return from the interrupt handler.





Understanding preemption (2)

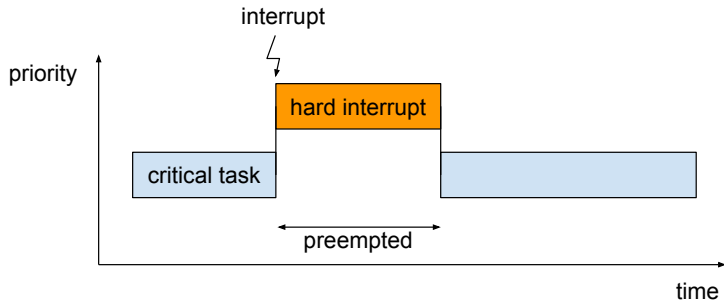
- ▶ However, when the interrupt comes while the task is executing a system call, this system call has to finish before another task can be scheduled.
- ▶ By default, the Linux kernel does not do kernel preemption.
- ▶ This means that the time before which the scheduler will be called to schedule another 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 synchronize (or serialize) accesses
- ▶ This needs to be looked at closely in RT context
- ▶ A low-priority task might 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.



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

- ▶ **Mutually exclusive**
- ▶ 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

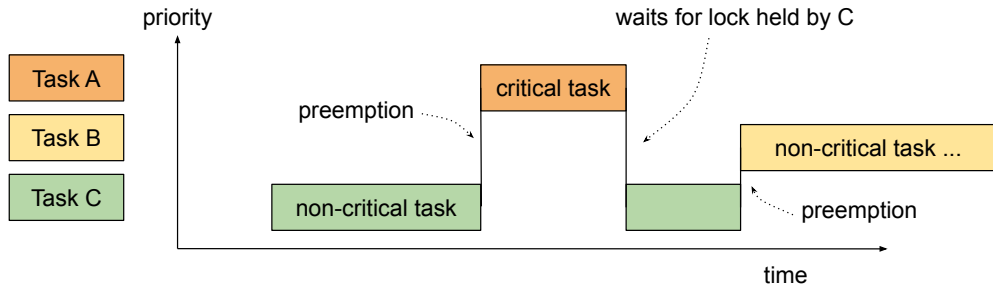


Priority inversion

- ▶ Priority inversion arises when strict priority-based scheduling interferes with locking
- ▶ It creates a scenario where a critical task is prevented from running by a lower priority task



Priority inversion

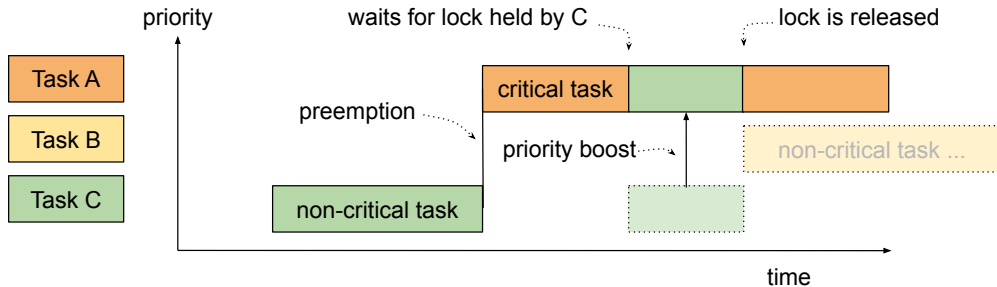


- ▶ Task A (high priority) needs to access a lock, hold by task C (low priority)
- ▶ The scheduler runs task C so that it can release the lock
- ▶ Task B has a higher priority than C, but lower than A, preempts task C



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!





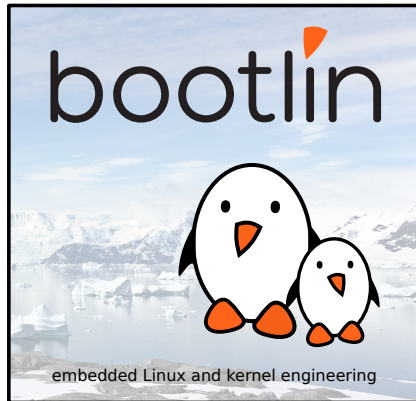
Priority Inheritance (2)

- ▶ **Priority Inheritance (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

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





The PREEMPT_RT preemption level

- ▶ 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
- ▶ Preempt-RT aims at making all code running in kernel mode preemptible
- ▶ PREEMPT_RT is a **compile-time** configuration option
 - It may only be accessible after applying a patch, the **Preempt RT Patch**
 - The patch is being included as part of the Linux kernel



PREEMPT_RT mainlining status

- ▶ The project has made steady progress since it got funding from the Linux Foundation in 2015 (Linux version 4.1 at that time).
- ▶ The PREEMPT_RT option no longer requires applying the "Preempt RT patch" on some architectures, starting from v6.12
 - x86_64, arm64 and RISC-V are supported so far
 - Other architectures still require the patch on v6.12.
- ▶ Stable versions of the patch are still maintained for older kernels



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?

- ▶ 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
- ▶ Using the `PREEMPT_RT` mode has performance impacts
 - Use it when you know your requirements



Getting the patch

- ▶ Starting from **v6.12**, applying a patch isn't needed on **x86**, **arm64** and **RiscV**
- ▶ For other architectures, or older kernels, the patch is still needed.
- ▶ 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
- ▶ Tracing
- ▶ Printk



- ▶ Code running in kernel mode can do so in different contexts :
 - **NMI** context
 - **hardirq** context
 - **softirq** context
 - **task** context
- ▶ Some operations are forbidden depending on the context :
 - Blocking (sleeping) operations are forbidden in **atomic** context
- ▶ **atomic** context means preemption is forbidden :
 - When not in **task** mode
 - When `preempt_disable()` was called
 - e.g. holding a spinning lock



Locking inside the Linux Kernel

- ▶ 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



Sleeping locks

- ▶ Sleeping locks will sleep and schedule while waiting
- ▶ There are several types of sleeping locks:
 - `mutex`
 - `rt_mutex`
 - `semaphore`
 - `rw_semaphore`
- ▶ Can't be used in **NMI**, **hardirq** and **softirq** contexts



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
- ▶ They can be used in **any** context, but with precaution
- ▶ With `PREEMPT_RT`, `spinlock_t` and `rwlock_t` will become sleeping 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



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()` is one of the main logging mechanism in the kernel
- ▶ It works in all execution context
- ▶ It was the last item preventing the RT-patch from being fully upstream
- ▶ The last kernel task who printed was in charge of printing the full buffer
- ▶ A low priority task could block a high priority task by printing lots of data



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
- ▶ Exceptions: `IRQF_NOTHREAD`, `IRQF_PERCPU`, `IRQS_ONESHOT`



Hard interrupts vs. Threaded interrupts

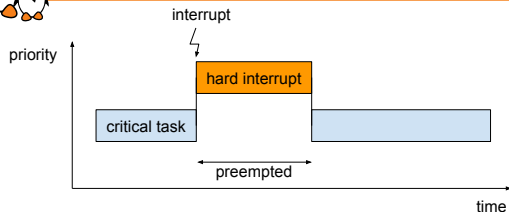


Figure: Hardware interrupt processing

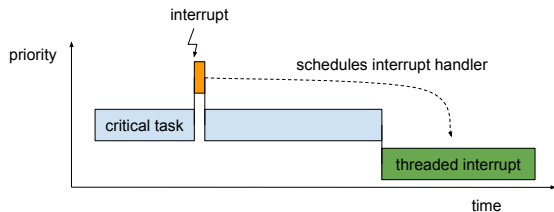


Figure: Threaded interrupt processing

- ▶ 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

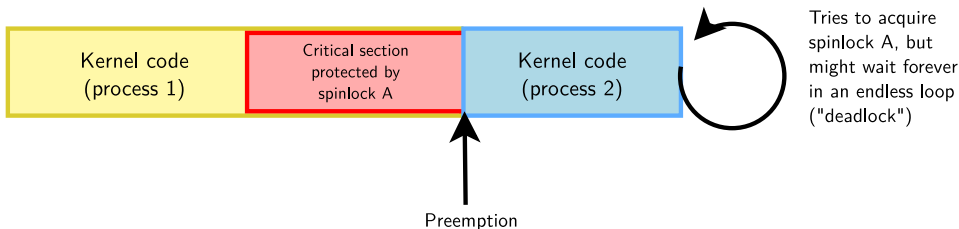


3rd option: preemptible kernel

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 explicitly 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



Testing and Benchmarking

© Copyright 2004-2025, Bootlin.
Creative Commons BY-SA 3.0 license.
Corrections, suggestions, contributions and translations are welcome!





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**



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.
- ▶ Traces **events**, defined within the kernel code, in **per-cpu buffers**
- ▶ Events have associated context:
 - `sched:sched_switch` : Context switch, indicates `prev_pid`, `next_pid`
- ▶ Can also be used to trace any kernel function with the **function tracer**



Using ftrace

- ▶ Tracing information available through the tracefs virtual fs
- ▶ Mount this filesystem as follows:
`mount -t tracefs nodev /sys/kernel/tracing`
- ▶ On some systems, it can also be found in `/sys/kernel/debug/tracing`
- ▶ Check available tracers:
 - `cat /sys/kernel/tracing/available_tracers`
- ▶ Select the interesting events:
 - `echo 1 > /sys/kernel/tracing/events/[X/Y/]enable`
- ▶ Start/stop tracing:
 - `echo [0|1] > /sys/kernel/tracing/tracing_on`
- ▶ Retrieve the trace:
 - `/sys/kernel/tracing/trace`: The trace buffers merged
 - `/sys/kernel/tracing/trace_pipe`: Stream the trace buffers, consuming data
 - `/sys/kernel/tracing/per-cpu/cpuX/trace`: Per-cpu traces



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
```



trace-cmd

- ▶ Wrapper around the ftrace interface
- ▶ Trace only during a program execution:
 - `trace-cmd record <opts> <cmd>`
 - `trace-cmd report`
- ▶ Start, stop and show the trace buffer:
 - `trace-cmd start <opts>`
 - `trace-cmd stop`
 - `trace-cmd show`
- ▶ Save the content of the trace buffer for further analysis:
 - `trace-cmd extract`
- ▶ Options
 - Events: `-e sched, -e sched:sched_switch`
 - Plugins: `-p function`
 - Tracers: `-t osnoise`
 - Functions: `-f netif_tx_wake_queue`

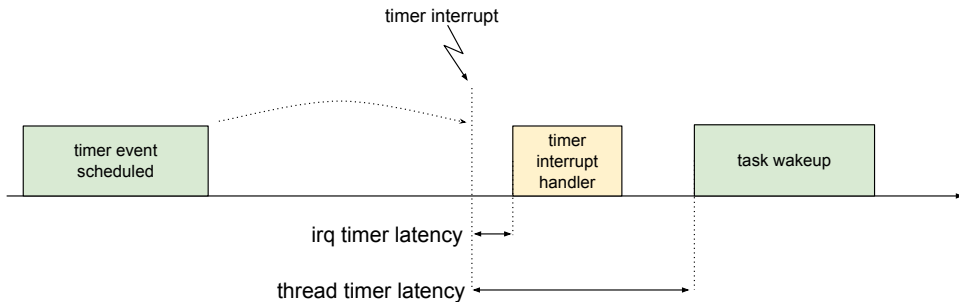


RealTime Linux Analysis tool

- ▶ Developed by Daniel Bristot de Oliveira
- ▶ High-level interface to the `timerlat` and `osnoise` tracers
- ▶ `rtla osnoise|timerlat top|hist` gives high-level view of noise and latencies
- ▶ Can generate histograms, that can then be visualized



- ▶ Gives an overview of "noise" sources from the Kernel and the Hardware
- ▶ Uses a similar measurement loop as `hwlatdetect`
- ▶ Uses `tracepoints` to detect the source of noise:
 - Thread Latency: Latencies due to the measuring thread being preempted
 - SoftIRQ Latency: Latencies due to softIRQ processing
 - IRQ Latency: Latencies introduced by IRQs
 - NMI Latency: Latencies introduced by NMIs
 - Hardware Latency: Latencies that aren't explained by any of the above
- ▶ `trace-cmd start -p osnoise`: Start recording os noise events
- ▶ `trace-cmd start -p osnoise -e osnoise`: Start recording noise events and trace their cause



- ▶ Periodic, per-cpu wakeup latency measurement in-kernel
- ▶ Can differentiate the IRQ wakeup time from the scheduling wakeup time
- ▶ Can use `osnoise` tracepoints to analyze the delay sources
- ▶ Can also trace **User return** latency, to benchmark custom applications



- ▶ Running `timerlat -a <us>` will trigger the auto-analysis mode
- ▶ A threshold is set with `-a`
- ▶ Measurement will stop if a latency higher than the threshold is detected
- ▶ Timerlat then prints a stack trace with the cause of the latency
- ▶ Can identify if the latency comes from a **blocking** or an **interference**
- ▶ Can identify the **task** of **interrupt** at the origin of the latency
- ▶ Can identify if the **hardware** itself is the culprit



- ▶ Focus on Hardware-induced latencies
- ▶ Similar to osnoise, but runs with **interrupts disabled**
- ▶ Only the hardware or non-maskable interrupts can interfere with measurements
- ▶ Tries to assign the noise to `NMI` when it can
- ▶ **NMI-based** watchdogs and **Hyperthreading** can cause such latencies



- ▶ 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`



Tool provided by `rt-tests`, relying on a dedicated `ftrace` tracer.

- ▶ Predecessor to **rtla hwnoise**
- ▶ Runs a tight loop on all CPU cores with local interrupts disabled
- ▶ Only NMIs and Hardware Latencies can interrupt the loop
- ▶ Samples a high precision timer and looks for large gaps between samples
- ▶ Useful to benchmark and validate a hardware platform
- ▶ Must **not** be used in production environment, introduces huge latencies



- ▶ 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



Stress and benchmark the Linux Kernel Scheduler

- ▶ Stresses the scheduler by creating lots of processes or 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



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

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



- ▶ Allows orchestrating stressing tools and testing tools
- ▶ Loads are **hackbench**, **kcompile** and **stress-ng**
- ▶ Testing tools are **cyclicttest** and **timerlat**
- ▶ Describe a full test through configuration files
- ▶ Generates a test report
- ▶ Ideal to integrate in a CI environment

<https://git.kernel.org/pub/scm/utils/rteval/rteval.git/>



Benchmarking an application



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

- ▶ Help analyze 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 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`
- ▶ `powercat`: Display the CPU usage
- ▶ `cat /sys/kernel/realtime`: Indicates if the RT Patch is applied
- ▶ `htop`: Displays running tasks, including kernel tasks



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



Hardware

© Copyright 2004-2025, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





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 can be 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 controlled
- ▶ Modern CPUs can expose a NMI counter
- ▶ Kernel-processed NMIs can be instrumented with `ftrace`
- ▶ `hwlatdetect` can help measure the NMIs on a given system



On modern hardware, several firmware 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
- ▶ Such firmware can disable interrupts while they run
- ▶ Depending on the SoC, we might or might not have control over these firmware
- ▶ `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

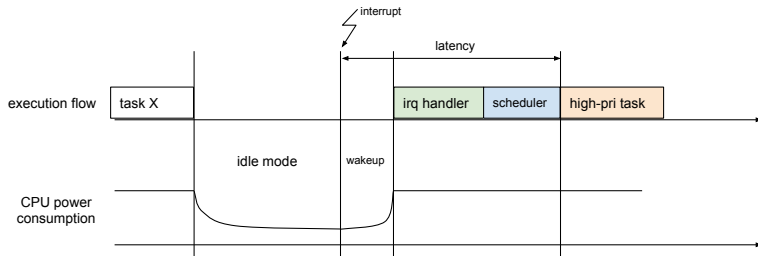


- ▶ **Non Uniform Memory Access**
- ▶ For high-end machines and servers, there are several banks of memory called **nodes**
- ▶ Typically, nodes are closer to some CPUs than others
- ▶ The Kernel can migrate pages from one node to another if need be
- ▶ Access latency will be longer for distant nodes
- ▶ 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 exist, 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 μ s	0 μ s
C1	2 μ s	2 μ s
C1E	10 μ s	20 μ s
C3	70 μ s	100 μ s
C6	85 μ s	200 μ s
C7s	124 μ s	800 μ s
C8	200 μ s	800 μ s
C9	480 μ s	5000 μ s
C10	890 μ s	5000 μ s

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-States 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 understand 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 (cc3)	3,2%	C3	3,0%	0,2 ms 2,3%	0,2 ms
C3 (pc3)	1,9%	C6 (cc6)	5,2%	C6	7,5%	0,5 ms 5,1%	0,3 ms
C6 (pc6)	0,4%	C7 (cc7)	28,0%	C7s	0,0%	0,1 ms 0,0%	0,1 ms
C7 (pc7)	0,0%			C8	16,0%	0,8 ms 8,2%	0,4 ms
C8 (pc8)	0,0%			C9	1,5%	1,1 ms 2,3%	0,2 ms
C9 (pc9)	0,0%						
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

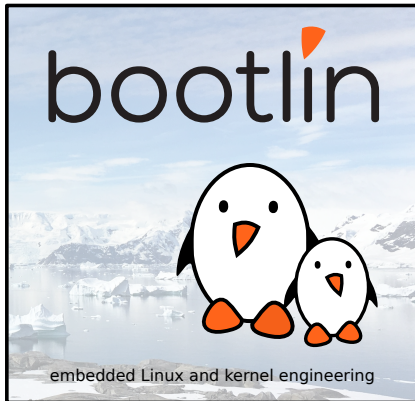


Configuring the system

© Copyright 2004-2025, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





- ▶ 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`



CPU Isolation - cpusets

- ▶ cpuset is a mechanism allowing to subdivide the CPU scheduling pool
- ▶ They are created at runtime, through the `cpusetfs`
 - `mount -t cpuset none /dev/cpuset`
- ▶ cpusets are created at will in the `cpuset` main directory
 - `mkdir /dev/cpuset/rt-set`
 - `mkdir /dev/cpuset/non-rt-set`
- ▶ Each cpuset is assigned a pool of cpu cores
 - `/bin/echo 2,3 > /dev/cpuset/rt-set`
 - `/bin/echo 0,1 > /dev/cpuset/non-rt-set`
- ▶ We can then select which task gets to run in each cpuset
 - `while read i; do /bin/echo $i; done < /dev/cpuset/tasks > /dev/cpuset/nonrt-set/tasks`
 - `/bin/echo $$ > /dev/cpuset/rt-set/tasks`
- ▶ You can run tasks in a given set with `cgexec -g cpuset:rt-set ...`



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 affinity to spread the load across CPUs
- ▶ Use the `IRQBALANCE_BANNED_CPUS` environment variable to make `irqbalance` ignore some CPUs
- ▶ The `irqaffinity` cmdline parameter can also be used



RCU

- ▶ **Read Copy Update**
- ▶ Synchronisation mechanism that can deferred object reclamation
- ▶ Deferred reclamation can be executed on any CPU RCU callbacks
- ▶ We can prevent CPU cores from running RCU callbacks with
`rcu_nocbs=<cpus> rcu_nocb_poll`

Workqueues

- ▶ Deferred execution mechanism
- ▶ Can be pinned to CPUs in `/sys/devices/virtual/workqueue/cpumask`



- ▶ Tool to easily setup **cpu isolation** and **irq affinities**
- ▶ Written in **python**
- ▶ `tuna isolate -c 3-5`
 - Removes CPUs 3,4 and 5 from every tasks's affinity list
 - Removes CPUs 3,4 and 5 from every IRQ's affinity list
- ▶ `tuna run -c 4 -p fifo:3 <cmd>`
 - Runs `cmd` on CPU core **4**, using `SCHED_FIFO` with a priority of **3**.



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 is tightly coupled to CPUidle and CPUFreq
- ▶ It schedules both **userspace** tasks and **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
 - This policy is actually called **SCHED_NORMAL** by the kernel
- ▶ **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

- ▶ 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
- ▶ Both `SCHED_FIFO` and `SCHED_RR` can be assigned a priority between 1 and 99
- ▶ `SCHED_DEADLINE`: For tasks doing recurrent jobs, extra attributes are attached to a task
 - A computation time, which represents the time the task 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
- ▶ Using one of these classes is necessary but not sufficient to get real-time behavior



Changing the Scheduling Class

- ▶ The Scheduling Class is set per-task, and defaults to `SCHED_OTHER`
- ▶ The `man 2 sched_setscheduler` syscall allows changing the class of a task
- ▶ The `chrt` tool uses it to allow changing the class of a running task:
 - `chrt -f/-b/-o/-r/-d -p PRIO PID`
- ▶ It can also be used to launch a new program with a dedicated class:
 - `chrt -f/-b/-o/-r/-d PRIO CMD`
- ▶ To show the current class and priority:
 - `chrt -p PID`
- ▶ New processes will inherit the class of their parent except if the `SCHED_RESET_ON_FORK` flag is set with `man 2 sched_setscheduler`
- ▶ See `man 7 sched` for more information



Scheduling order

SCHED_DEADLINE



SCHED_FIFO
SCHED_RR



SCHED_OTHER
SCHED_BATCH

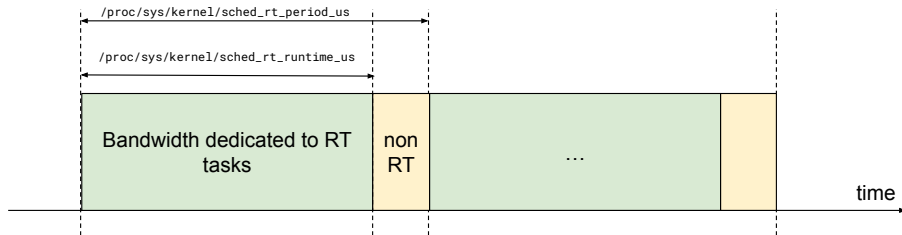


SCHED_IDLE

- ▶ When invoked, the scheduler looks for runnable tasks in a specific order
- ▶ SCHED_FIFO and SCHED_RR share the same **runqueue**
- ▶ However when a SCHED_RR task yields, any same prio SCHED_FIFO will run until done
- ▶ until **v6.6**, SCHED_OTHER and SCHED_BATCH uses the **Completely Fair Scheduler**
- ▶ It was then replaced by **EEVDF** (Earliest Eligible Virtual Deadline First) Scheduler, improved in **v6.12**.



Realtime Throttling



- ▶ An infinite loop in a high-priority task will starve everything else
- ▶ `/proc/sys/kernel/sched_rt_period_us` : Time sharing period (-1 to disable)
- ▶ `/proc/sys/kernel/sched_rt_runtime_us` : Amount of time in each period dedicated to RT tasks
- ▶ The default values allocates 95% of the CPU time to RT tasks
- ▶ Can be used **per-cgroup**



Deadline Server

- ▶ RT Throttling can be overkill, as it fires even when no non-RT tasks needs to run
- ▶ The Deadline Server is a more precise replacement mechanism
- ▶ Uses a dedicated `SCHED_DEADLINE` task, the **deadline server**
- ▶ Available in **v6.12**, configurable in **debugfs**
- ▶ Won't trigger if there's no starving non-realtime task
- ▶ The starving task will only use the bandwidth it needs
- ▶ Replaces the old throttling mechanism, except for **grouping**

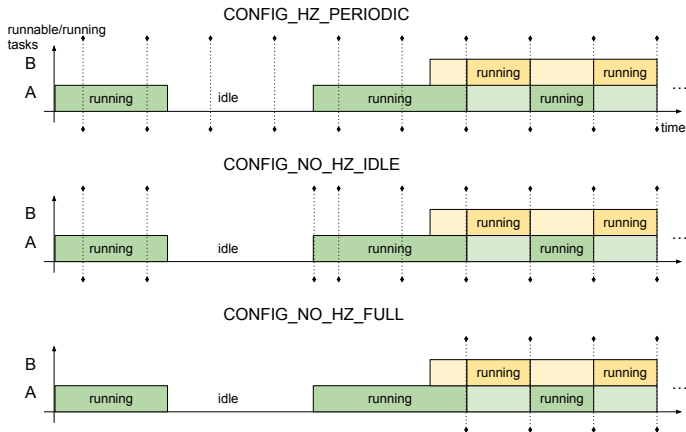


System timer

- ▶ 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
- ▶ Several policies regarding system ticks are available:
- ▶ `CONFIG_HZ_PERIODIC`: Always tick at a given rate. This introduces small interferences but is deterministic.
- ▶ `CONFIG_NO_HZ_IDLE`: Disable the tick when idle, for powersaving
 - Longer wakeup from Idle, replay the missed ticks (**jiffies** housekeeping)
- ▶ `CONFIG_NO_HZ_FULL`: Actively disables ticking even when not idle.
 - Per-CPU setting, through the `nohz_full=<range>` boot parameter
 - Only relevant on multi-core systems, one core must stay in `nohz_idle` mode
 - `nohz_full` cores will automatically offload their **RCU**
 - Slightly more expensive Kernel to User transitions



System timer



► Tick always enabled

► Tick disabled in Idle

► Tick disabled in Idle and when only one runnable task



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



Application development

© Copyright 2004-2025, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Real-time application development

- ▶ A few best-practices should be followed when developing 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 initialization 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`



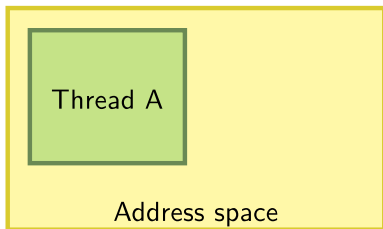
Process, thread?

- ▶ 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()`

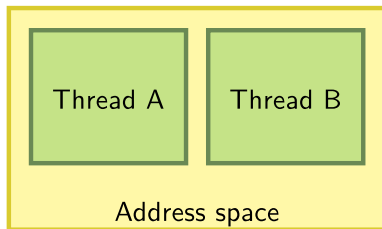


Process, thread: kernel point of view

- ▶ 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)

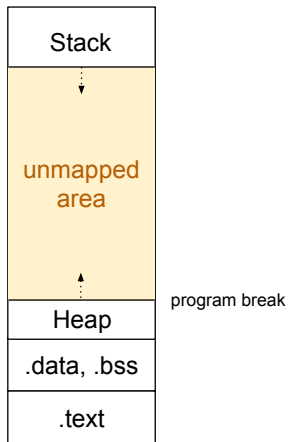
- ▶ The priority can be set on a per-thread basis when a thread is created

```
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.



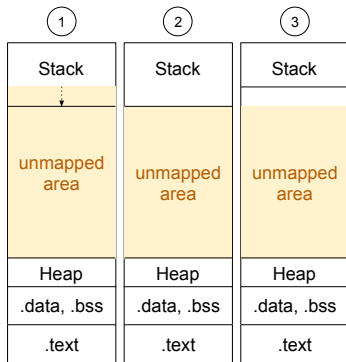
Memory layout and allocation



- ▶ The **stack** grows down from high addresses
- ▶ The **heap** is used for dynamic allocations, grows up to the **program break**
- ▶ `mmap()` can also be used to allocate memory in-between
- ▶ `malloc` uses both `brk` and `mmap`
- ▶ Allocated memory needs to be **mapped** to physical memory
- ▶ Mapping is done per **page**, usually 4 Kilobytes
- ▶ It is created when the page is **first accessed** through a **page fault**
- ▶ Each page fault will raise an exception, preempt the process and introduce latency



Pre-faulting the stack



1. Allocate huge buffers on the stack in a sub-function
2. Access (read or write) the whole buffer
3. return from the sub-function

stack prefault

```
#define BUFSZ (1024 * 1024 * 8)
static void stack_prefault() {
    volatile char buff[BUFSZ];
    long page_sz;
    int i;

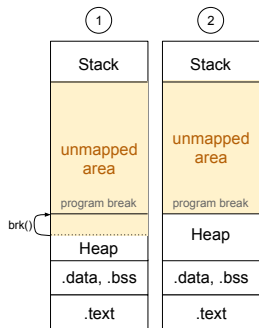
    page_sz = sysconf(_SC_PAGESIZE);

    for (i = 0; i < BUFSZ; i += page_sz)
        buff[i] = 0;
}
```

Each thread's stack must be pre-faulted



Pre-faulting the heap



- ▶ The **program break** defines the size of the `.data` segment
- ▶ `malloc` uses the `.data` segment as a pool for small buffers
- ▶ Prevent the program break from shrinking
 - `mallopt(M_TRIM_THRESHOLD, -1);`
- ▶ Prevent `malloc` from using `mmap`
 - By default, `malloc` uses `mmap` for large buffers
 - Once free'd, `mmap`'d pages won't be re-used
 - `mallopt(M_MMAP_MAX, 0);`
- ▶ allocate a huge buffer to move the program break
- ▶ Access the whole buffer
- ▶ `free()` the buffer
- ▶ Subsequent calls to `malloc` will re-use pre-faulted memory
- ▶ For multi-threaded applications, pre-fault each **arena**
 - see `M_ARENA_TEST` and `M_ARENA_MAX` `mallopt` options



Memory management

- ▶ Call `mlockall(MCL_CURRENT | MCL_FUTURE)` at init to lock all memory regions
- ▶ This prevents mapping from being removed
- ▶ Beware of `fork()`, since the child will copy-on-write pages
- ▶ `malloc`'s implementation is libC specific
- ▶ **glibc** implements all the necessary options
- ▶ **musl** doesn't implement `mallopt` and may use `mmap`
- ▶ **fttrace** can be used to trace pagefaults :
 - `trace-cmd record -e page_fault* <cmd>`



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 explicitly enabled:
`pthread_mutexattr_setprotocol(&mutexattr, 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



- ▶ 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

© Copyright 2004-2025, Bootlin.

Creative Commons BY-SA 3.0 license.

Corrections, suggestions, contributions and translations are welcome!





Thank you!
And may the Source be with you



Rights to copy

© Copyright 2004-2025, Bootlin

License: Creative Commons Attribution - Share Alike 3.0

<https://creativecommons.org/licenses/by-sa/3.0/legalcode>

You are free:

- ▶ to copy, distribute, display, and perform the work
- ▶ to make derivative works
- ▶ to make commercial use of the work

Under the following conditions:

- ▶ **Attribution.** You must give the original author credit.
- ▶ **Share Alike.** If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.
- ▶ For any reuse or distribution, you must make clear to others the license terms of this work.
- ▶ Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

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