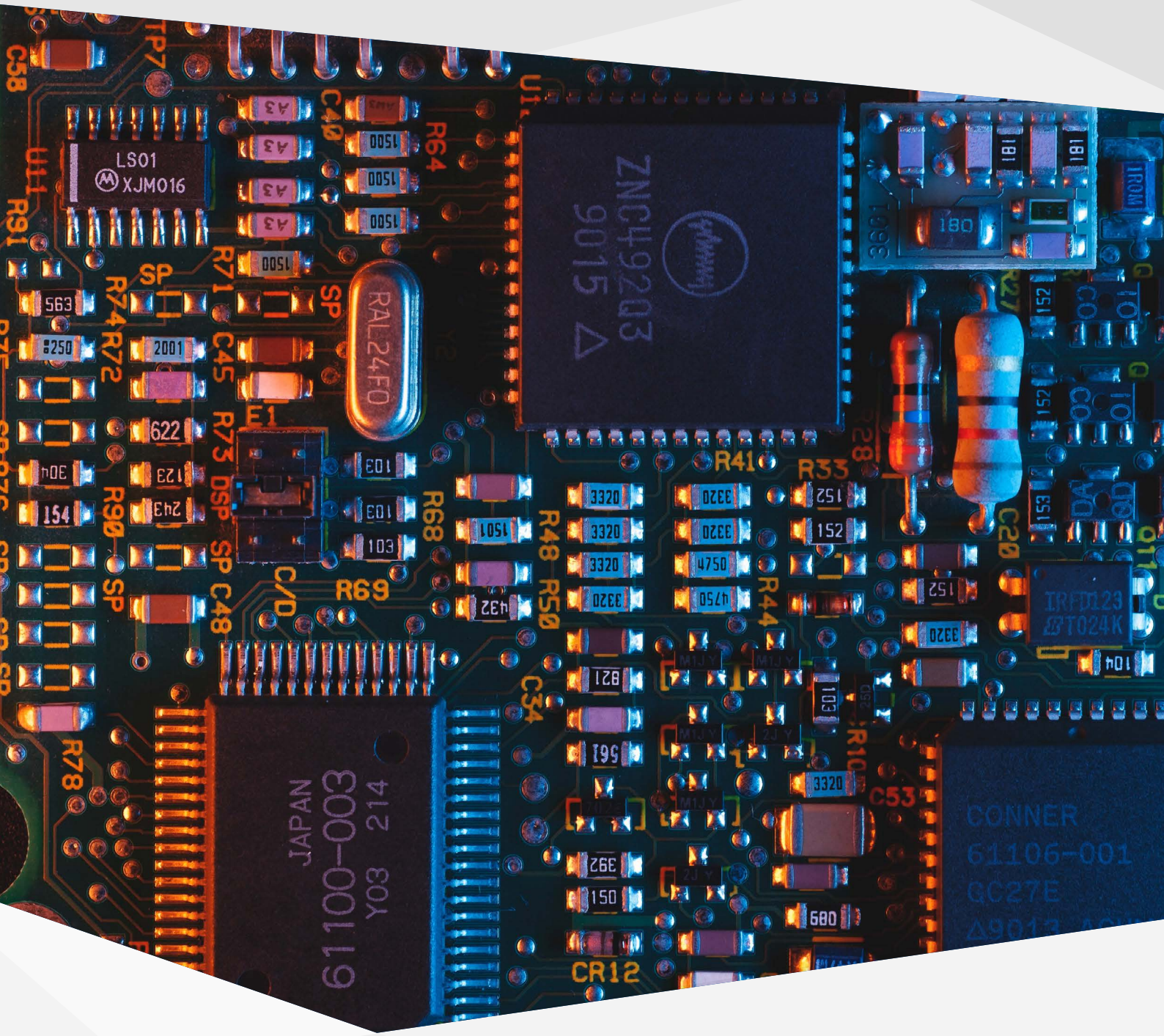
Canonical

A CTO's guide to real-time Linux

Understanding real-time systems, their use cases and inner workings



Executive Summary

30% of the world's data will need real-time processing by 2025.

According to recent forecasts, nearly 30% of the world's data will need real-time processing by 2025.[1] Demand for industrial PCs that drive control systems, industrial edge servers, PLCs, robots and drones with real-time capabilities is rising. While workloads demand varying levels of real-time performance and desired cycle times differ, analysts expect market adoption of real-time computing to continue and accelerate in the years ahead.

On the software side, real-time compute via the Linux kernel is emerging as a valuable solution thanks to the rich support for hardware devices and peripherals. Among various approaches to bring real-time capabilities to the kernel, the PREEMPT_RT patch aims to implement a priority scheduler and other supporting real-time mechanisms. As the de-facto Linux real-time implementation, PREEMPT_RT increases predictability and reduces latencies by modifying the existing kernel code.

With time-bound responses for mission-critical latency requirements, real-time Linux strives to provide deterministic processing to the most demanding workloads in industrial, telco, automotive, aerospace and defence. This in-depth guide covers real-time Linux with PREEMPT_RT, including considerations for adoption, common misconceptions, key market applications and use cases.

While real-time Linux is ideal for latency-sensitive use cases by ensuring it executes high-priority processes first, a real-time kernel on its own will not necessarily make a system real-time. This guide will introduce you to the complete real-time stack, and highlight the performance trade-offs you should consider when choosing a real-time versus, for instance, a low-latency Linux kernel.

1 https://www.readkong.com/page/the-digitization-of-the-world-from-edge-to-core-8666239

Contents

Introduction

In this guide, we discuss how a Linux OS with support for real-time compute delivers industrial-grade performance to meet stringent latency requirements.

The report structure follows the considerations an executive or engineering manager may go through when evaluating the adoption of real-time Linux in their enterprise. Each of the subsequent chapters gradually builds upon the knowledge from the previous, starting with the building blocks and concluding with a technical assessment.

Chapter 1 presents the common misconceptions behind a real-time OS and clarifies its definition, and its target market and industry verticals, with a specific focus on applications in the industrial sector and telecommunications.

Chapter 2 introduces you to the concept of preemption and the PREEMPT_RT patches, the cornerstone behind delivering deterministic response times in the Linux kernel.

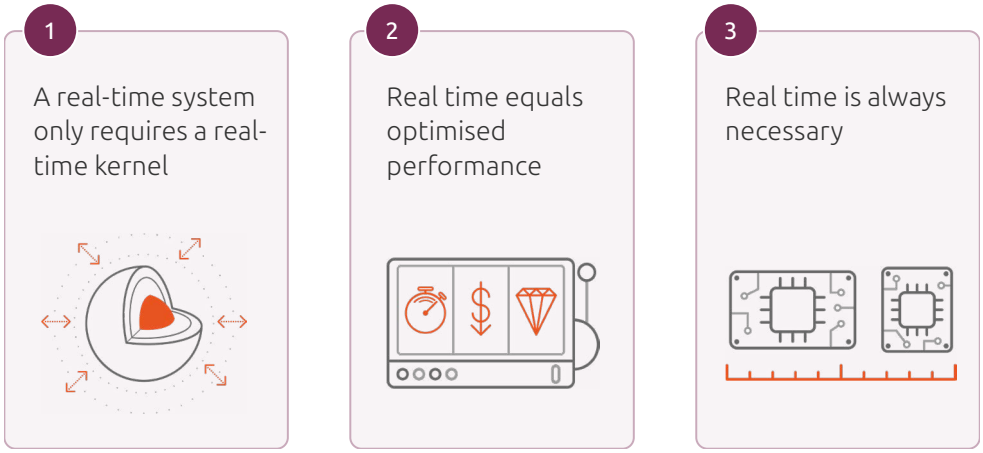
A real-time kernel on its own will not necessarily make a system real-time, as even the most efficient Real Time Operating System (RTOS) can be useless in the presence of other latency sinks. Each layer of a real-time stack must support deterministic processing, from the hardware to the kernel. Chapter 3 covers the silicon optimisations and enablements required to deliver an out-of-the-box real-time solution with Ubuntu on Intel SoCs.

Finally, chapter 4 covers technical considerations. From networking to cache partitioning, every shared resource can affect cycle times and be a source of jitter. We place special emphasis on the tuning aspect of setting up a real-time system to meet stringent low-latency requirements.

An introduction to real-time Linux

There is no unanimous agreement on the precise terminology and definition of real-time Linux. In this chapter, we explore two definitions of real time and clarify the concept by dispelling common misconceptions. Additionally, we examine various verticals where real time capabilities are crucial, highlighting their specific use cases. This chapter will lay the groundwork necessary to delve into the technicalities of real-time Linux in later sections. In particular, we introduce concepts that are necessary to properly comprehend the mechanisms implemented by the PREEMPT_RT patchset to reduce kernel latencies.

Common misconceptions



1. A real-time system only requires a real-time kernel

It is essential to grasp that a real-time Linux kernel alone does not guarantee real-time behaviour in a system. The kernel is just one component of a real-time system, and even the most efficient RTOS can prove ineffective if latency issues exist elsewhere. Achieving real-time compute demands meticulous tuning of the entire system stack, from the underlying hardware to the operating system, networking layer, and applications.

2. Real time equals optimised performance

Another common misconception is that real time implies optimised performance. This misunderstanding often arises from video applications described as real time due to their low perceived latency. However, these are typically best-effort systems that prioritise reducing noticeable delays. A real-time Linux kernel does not focus on achieving optimised performance but rather on providing a deterministic response to external events and minimising response latency. In fact, a real-time Linux kernel will often exhibit inferior performance compared to schedulers like CFS, except in specific task-scheduling scenarios.[2]

3. Real time is always necessary

Contrary to popular belief, a real-time operating system is not always necessary. The desire for “real time” stems from the performance connotation, but it is crucial to assess the actual consequences of missing a deadline and determine

2 https://docs.kernel.org/scheduler/sched-design-CFS.html

whether real-time requirements are warranted. For instance, if a deadline falls within a timeframe of seconds, a properly tuned multi-GHz CPU is likely to meet it effectively.

Real-time Linux definitions

Real-time operating systems and real-time systems have been a subject of confusion and disagreement, making it essential to shed light on their precise meanings. In this section, we explore two distinct definitions.

"In a real-time system, the correctness of a computation depends not only upon the logical correctness of the result but upon the time at which it is produced. If the timing constraints of the system are not met, system failure is said to have occurred."[3]

The above definition is often used to describe a real-time system. We prefer this definition because it clearly outlines the purpose of this type of system: establishing timing constraints. Real-time systems operate within well-defined and fixed time boundaries. Failure occurs when these time constraints are not met and the system is unable to complete its real-time tasks within the specified upper boundary. Hence, to ensure the proper functioning of real-time applications, it is imperative to determine the worst-case time for a given response time (i.e. real-time applications require a deterministic operating environment).

Now, let's look at a common definition of real-time operating systems.

"Real time in operating systems: the ability of the operating system to provide a required level of service in a bounded response time"[4]

As the definition explains, a real-time operating system should be capable of responding to an event within a known and limited time frame. The response time of an application is the time interval between receiving a signal or stimulus, typically through a hardware interrupt in the Linux kernel, and producing a result based on that input. This means that the processing of time-critical tasks relies on the system's capability to respond to an event within a known and limited time frame.

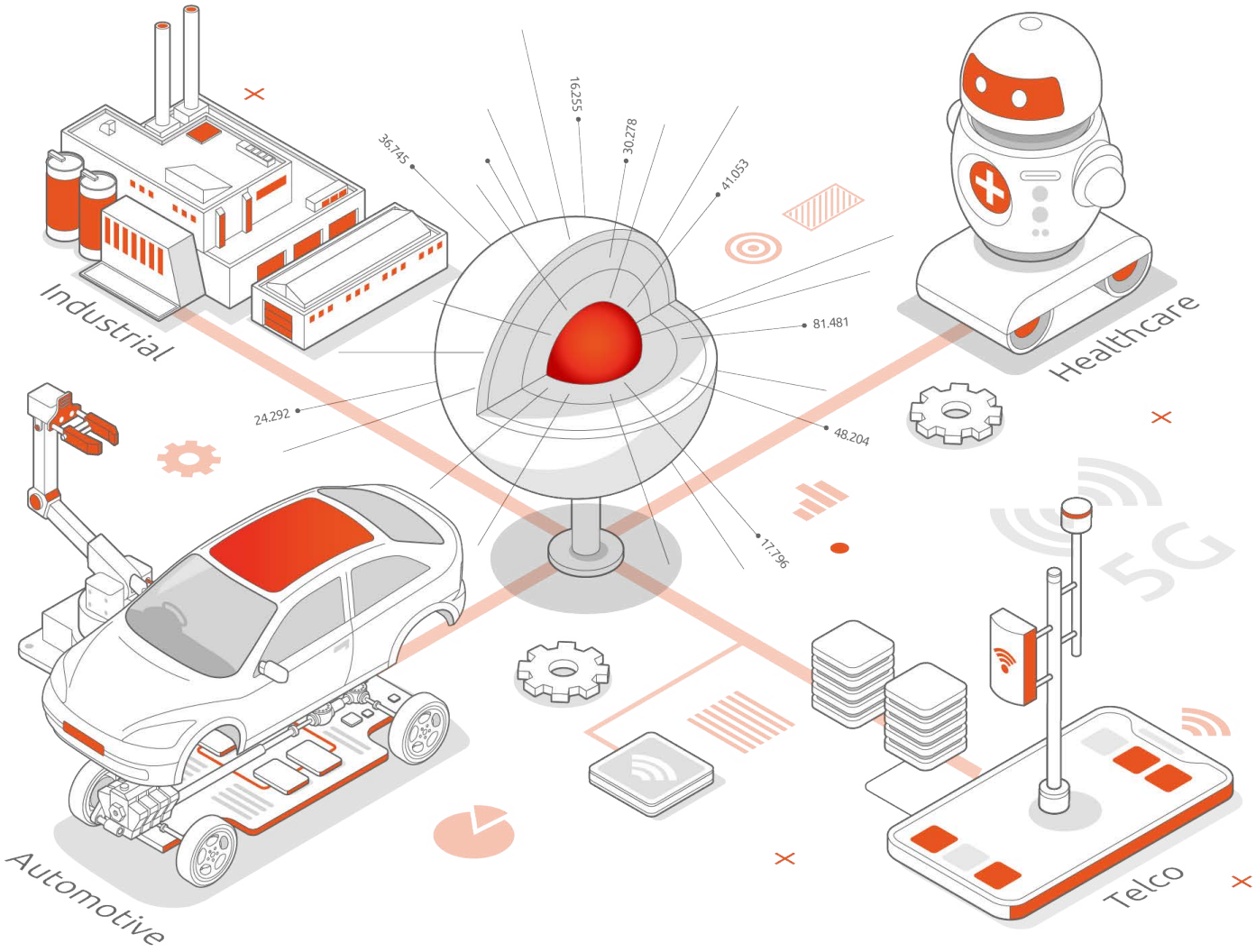
Real-time Linux encompasses both the concept of a real-time system with well-defined timing constraints and an OS that attempts to provide deterministic responses to external events. In the context of the kernel, real-time signifies a deterministic response to an external event. The objective is to guarantee an upper bound on response time, ensuring the timely execution of critical tasks.

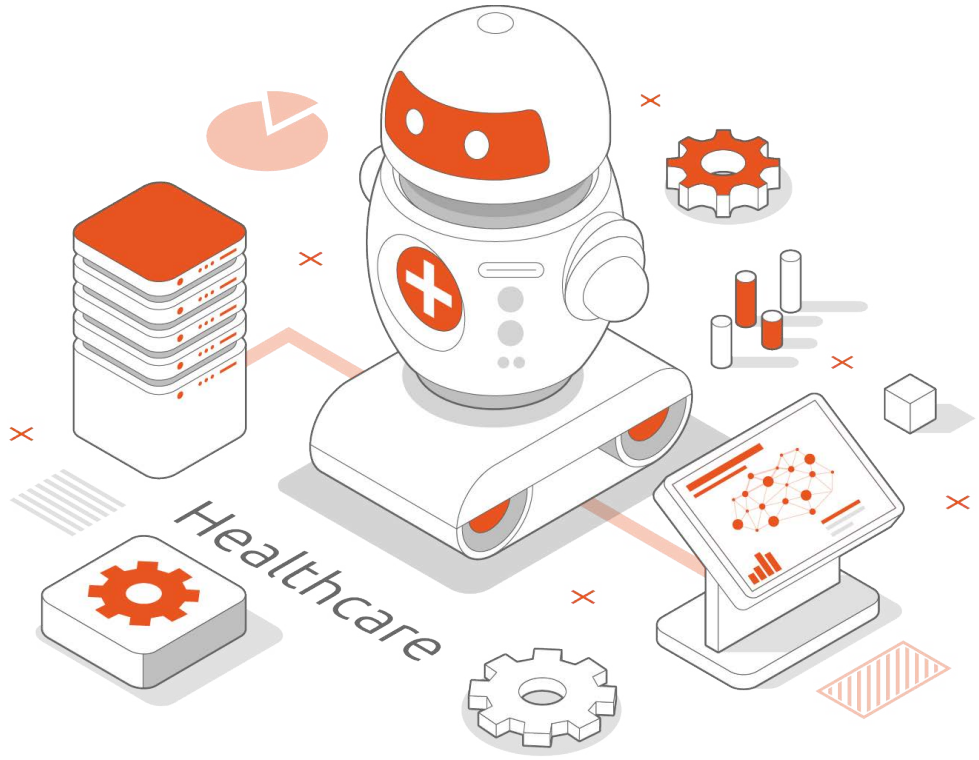
Now that we have dispelled the common myths around real-time Linux systems and defined them, let's look at their typical use cases and market applications.

3 https://groups.google.com/g/comp.realtime/c/BuXZqYnm3tg/m/iwtllaqGyHIJ
4 https://ieeexplore.ieee.org/document/8277153

Target verticals and applications

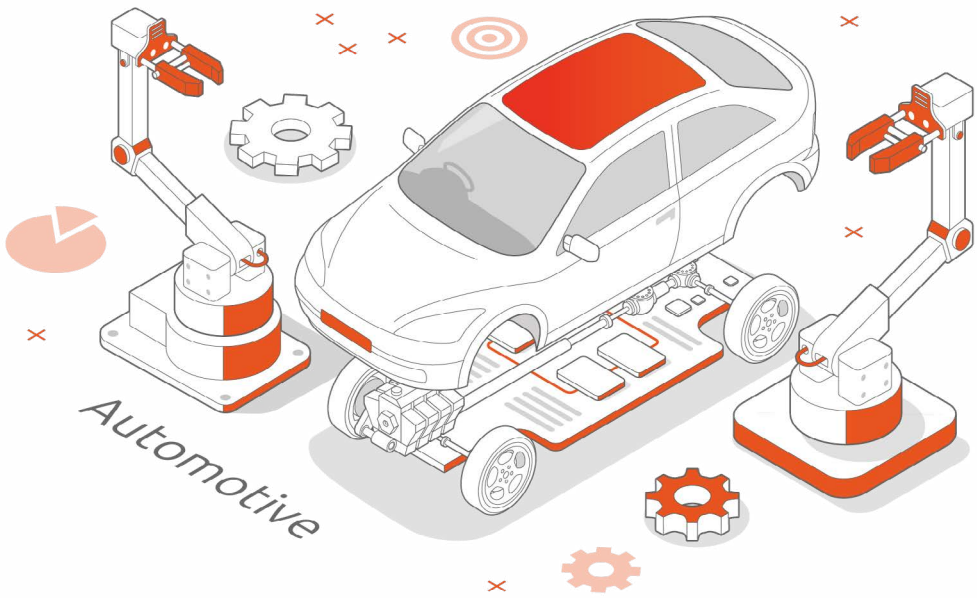
This section identifies scenarios where relying on a real-time Linux kernel is appropriate. Real-time Linux finds applications across a wide range of industries, including process automation (energy sector, petroleum, refineries) and discrete automation (car manufacturing). Furthermore, healthcare, factories, telco networks, automotive, aviation, and more, often demand real-time compute capabilities.





Healthcare

Dedicated devices like life-support medical equipment can require real-time processing. Operators expect computing systems to complete tasks within specified deadlines, as failure to meet these deadlines can have catastrophic consequences.



Automotive

Real-time systems find application in the automotive industry, particularly in latency-sensitive scenarios where a missed deadline could have catastrophic consequences, such as car brakes failing to prevent a crash.[5]

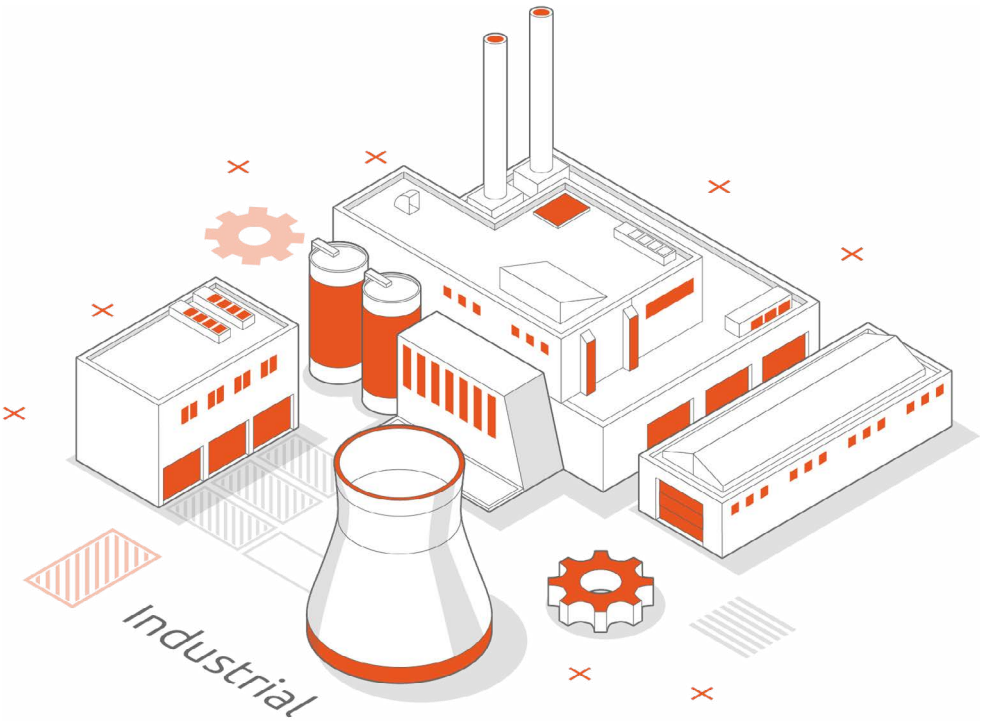
5 https://canonical.com/blog/real-time-ubuntu-aws

While RTOSs have traditionally been used to meet these strict latencies, automotive manufacturers and Tier 1 suppliers are increasingly relying on the real-time Linux kernel for applications like infotainment systems and Human-Machine Interfaces (HMIs).[6]

Beyond the aforementioned verticals, real-time Linux finds application in a wide range of use cases, including product quality assurance, safety, oil and gas, and transportation systems with strict precision requirements for automation.

We will now zoom into two key verticals where real-time compute and Linux are in demand: industrial automation, and telecommunications.

Real-time Linux in the industrial sector



In this section, we explore the underlying technology shifts affecting industrial companies, the business benefits of this transformation, and the role of real-time Linux in industrial automation.

The term “Industry 4.0” is often used to describe the journey undertaken by industrial companies to digitally transform their value chains, both within their vertical operations and across their horizontal partners.[7] This transformation entails digitising essential functions and embracing connectivity throughout the industrial ecosystem. It marks a shift from traditional dedicated hardware to software-defined systems capable of delivering a diverse range of functionalities. By doing so, companies gain more dynamic systems with enhanced flexibility, facilitating implementation within factories and other industrial environments.

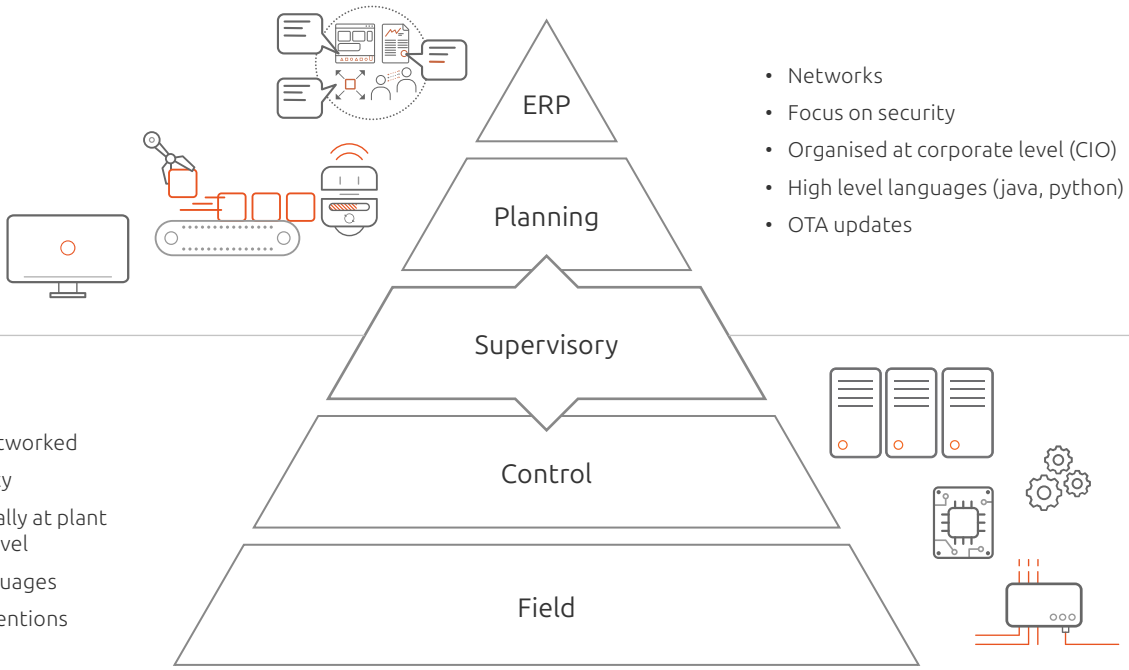
By adopting a general-purpose OS like Linux with support for real-time compute via the PREEMPT_RT patches, enterprises can navigate the transition towards Industry 4.0. They can achieve digitisation and modernisation while driving flexible production. Let’s explore how.

6 https://www.youtube.com/watch?v=IWdQETXN-0k&ab_channel=CanonicalUbuntu
7 https://canonical.com/blog/industry-4

Bridging the gap with real-time Linux

Traditional industrial platforms are characterised by isolated networks, often comprising proprietary and industrial field bus systems. These networks connect various dedicated setups, each performing a specific (real-time) function. However, this legacy infrastructure poses barriers to data sharing and impedes two-way communication between devices.

The automation pyramid



- Networks
- Focus on security
- Organised at corporate level (CIO)
- High level languages (java, python)
- OTA updates

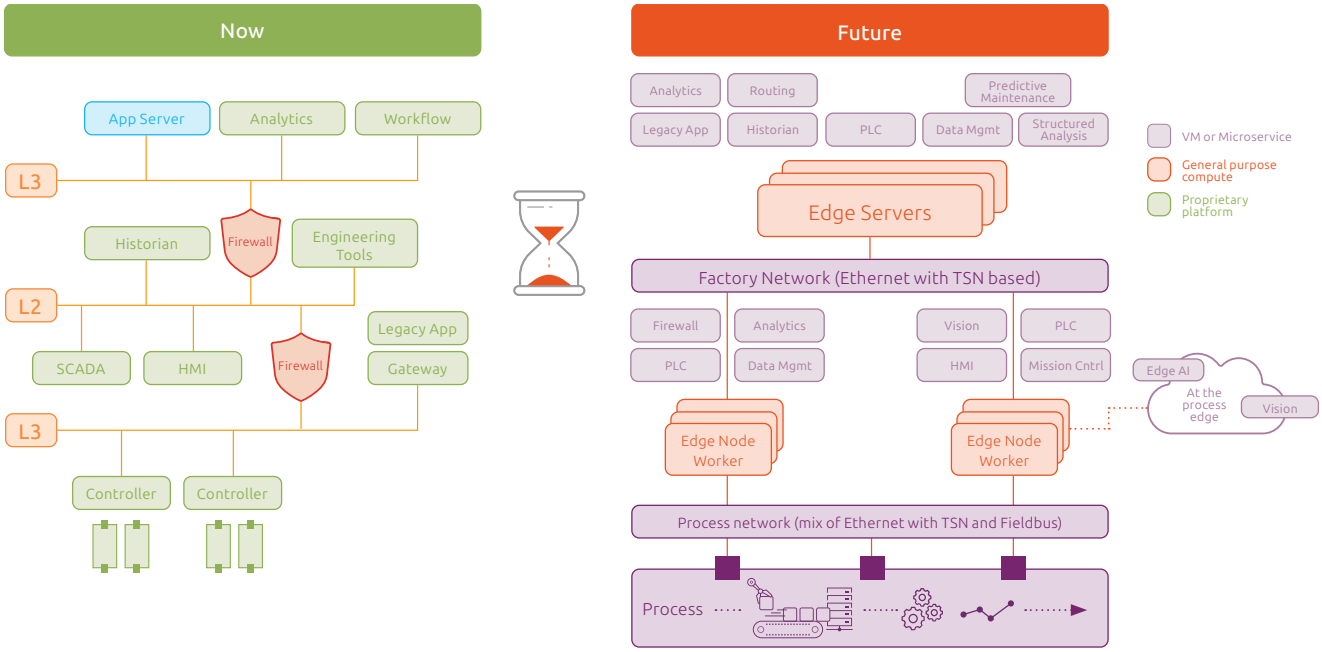
- Not always networked
- Lack of security
- Organized locally at plant engineering level
- Specialist languages
- On-site interventions at shop floor

The transition towards Industry 4.0 and the realisation of autonomous industrial systems cannot happen without infrastructure digitisation and modernisation. These changes enable visibility, transparency, and control over physical devices and processes in manufacturing operations.[8] To achieve this, a transition from closed standards and interfaces to open source software and modern IT solutions is crucial.

An open source OS like Linux, with support for real-time capabilities, can play a role in bridging this gap.[9] Enterprises can make use of real-time capabilities to facilitate quick reconfiguration and enable the exchange and utilisation of data for analytics from various nodes.

8 https://ubuntu.com/engage/bridge-the-IT-OT-gap-Industry4
9 https://ubuntu.com/engage/linux-embedded-applications-whitepaper

Industry Transformation



Real-time Linux in industrial environments

Real-time Linux operates at different levels within the industrial landscape, consolidating historically separate components into industrial PCs (IPCs) . Traditionally, IPCs were primarily used for HMIs and data gathering, lacking support for real-time compute. However, we are now witnessing increased demand for IPCs with real-time capabilities to drive control systems. Moreover, determinism and real-time response are essential on the factory floor. Programmable Logic Controllers (PLCs) play a vital role in this transition, with industrial servers running real-time Linux for factory-level control. PLCs in assembly lines must deliver and process data in real time to ensure system integrity and continuous production. Missing a deadline in such cases could jeopardise the entire production line.

Real-time Linux plays a key role in the software-defined transition to industry 4.0, underpinned by open standards, open interfaces, and the adoption of open source Linux. The adoption of a Linux OS with real-time capabilities can enable connectivity, flexibility, and enhanced control over industrial processes. As this progression will occur gradually rather than overnight, it is essential to initiate the transition as early as possible.

Real-time Linux in telco

Real-time Linux can bring multiple advantages to the telecom sector, from processing real-time data, to ensuring ultra-low latency and providing enhanced security for critical infrastructure. Real-time Linux support for Radio Access Network (RAN) solutions can enable telecom operators to build efficient and high-performing 5G network architectures. Similarly, scalable and cost-effective 5G networks can benefit from real-time Linux in conjunction with a virtualised Radio Access Network (vRAN).[10]

10 https://ubuntu.com/engage/intel-flexran-ubuntu-real-time-kernel

Meeting the transformation needs of 5G networks

The virtualised approach addresses the challenges of traditional RAN architectures by abstracting hardware from core network functions and enabling optimal resource utilisation.

As telecom operators evolve their networks to handle high-bandwidth, real-time data, OpenRAN implementations play a vital role in the access network transformation.[11] By using virtualised OpenRAN architectures and the latest advancements in Linux, mobile operators can deploy network functions with greater flexibility and efficient resource utilisation.

Real-time LInux support for OpenRAN implementations like FlexRAN empowers telecom operators to build high-performance 5G networks with optimal performance and ultra-low latency.[12] As the telco industry continues to evolve, the combination of real-time Linux and FlexRAN holds the potential to set the stage for transformative advancements in telecommunications networks, paving the way for future innovations and services.[13]

In the following chapter, we explore how real-time Linux addresses the challenges posed by various use cases and verticals, including the ones mentioned above, from a technical perspective. Before delving into technicalities, we need to understand the concept of preemption.

11 https://ubuntu.com/blog/what-is-openran?
12 https://www.intel.com/content/www/us/en/developer/videos/an-overview-of-flexran-sw-wireless-access-solutions.html
13 https://ubuntu.com/blog/canonical-announces-ubuntu-22-04-lts-support-for-flexran-reference-software

The role of preemption in real-time Linux

"Preemption" and real-time scheduling lie at the core of real-time Linux.[14] Without kernel preemption, there can be no real-time compute in Linux. Preemption consists of temporarily interrupting the current thread of execution so that a higher-priority event can be processed in a timely manner. Increasing the preemptible code surface within the Linux kernel dramatically improves the capability to provide a deterministic response time to an external event.

In the following sections, we will sketch the mechanics of preemption by looking at an example transition between User and Kernel Mode.

Introduction to Linux kernel preemption

Before delving into the transition, it is important to recall that a process, i.e. an instance of a program in execution, operates its sequence of instructions within the specific set of memory address spaces it is allowed to reference. Furthermore, the system memory in the Linux kernel is segregated into kernel space and user space. The kernel space is the access-protected memory area reserved for executing the core of the OS in Kernel Mode, whereas user space is the system memory where user processes execute in User Mode.

When a process executes in User Mode, it cannot directly access the kernel data structures or programs. On the other hand, these restrictions no longer apply when an application executes in so-called Kernel Mode.

Usually, a program executes in User Mode and switches to Kernel Mode only when requesting a service provided by the kernel. The Linux kernel then puts the program back in User Mode after satisfying the program's request. To this end, each CPU model provides special instructions to switch from User Mode to Kernel Mode and vice versa.

User space to kernel space transition

In the example Figure 1 below, Process A starts its execution in User Mode until a hardware device raises an interrupt. Process A then gets preempted out, as interrupts are preemptive in user code. The execution on that core thus switches to Kernel Mode to service the interrupt. After the kernel handles the interrupt, possibly running some user code, it then schedules the next process to run. Hence, Process A can now resume its execution.

The process continues in User Mode until a timer interrupt occurs, and the scheduler is activated in Kernel Mode. Now, a process switch occurs, and Process B starts its execution in User Mode.

Process B stays in User space until it issues a system call to access the kernel data structures: the process switches to Kernel Mode, and the system call can be serviced.

14 https://www.brighttalk.com/webcast/6793/571067

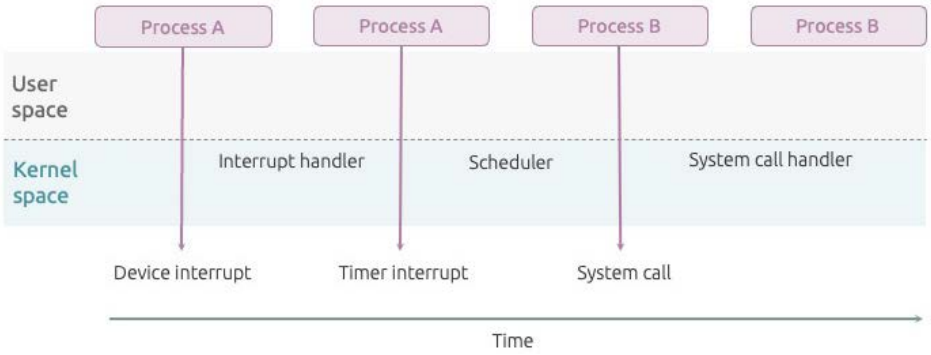


Figure 1: Non-preemptible kernel

With no kernel preemption, like in the scenario depicted above, tasks can't be interrupted once they start executing code in the kernel or a driver. When a user space process requests a kernel service, no other task can be scheduled to run until that process either goes to sleep or until the kernel request is completed. Scheduling can only take place after the task voluntarily suspends itself in the kernel, or preemption can occur when the process exits it. What this means is that there can be no deterministic response time, as all processes, including those with the highest priority, have to wait for the completion of the running kernel request.

On the other hand, making the kernel preemptible means that while one lower-priority process is running in the kernel, a higher-priority process can interrupt it and be allowed to run. Hence, introducing the flexibility to preempt tasks executing within the kernel is the key to guarantee an upper time boundary.

Let's now look at a process switch in a Linux kernel with preemption capabilities to illustrate the advantages.

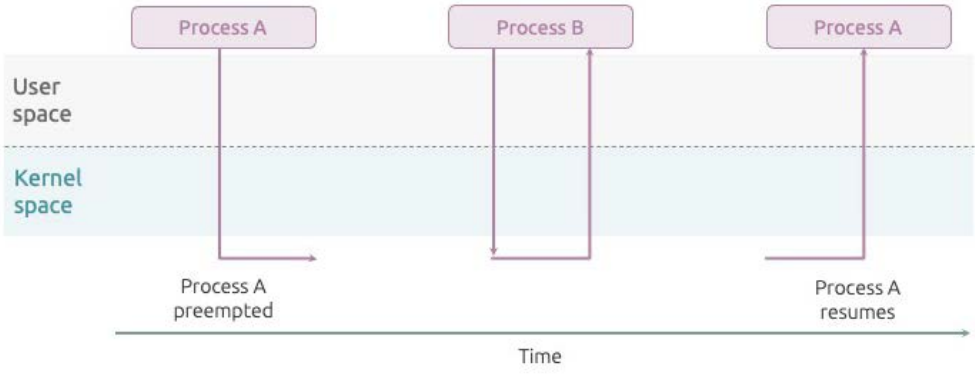


Figure 2: Making the kernel preemptible

In Figure 2, Process A enters the kernel. At this point, a higher priority Process B is woken up. The kernel then preempts the lower-priority Process A and assigns the CPU to Process B, even though Process A hasn't yet serviced its request.

The simple scenario aims to illustrate how the interrupt which caused Process B to become ready, triggered the scheduler to run as opposed to just waiting for the next timer tick. In perhaps overly simplistic terms, one can think of higher-priority processes being able to interrupt lower-priority ones (already running in Kernel Mode) in a preemptive kernel. While in the early Linux days of Linux 1.x kernel

preemption did not exist, several attempts to bring preemption into the kernel have been proposed through the years and made their way into mainline. Linus Torvalds introduced kernel preemption in Linux only with version 2.5.4.[15]

Making the kernel preemptible means that while one lower-priority process is running in the kernel, a higher-priority process can interrupt it and be allowed to run.

The challenge in making a preemptible kernel lies in identifying all critical sections within the kernel that must be protected from preemption. The next section illustrates the challenges involved in bringing preemption into the Linux kernel. In the remainder of the chapter, we will then study the different preemption models currently available in mainline and introduce the PREEMPT_RT real-time patches.

Challenges to implementing preemption

In the example scenario in Figure 1 below, two processes with different priority levels operate on shared variables and make decisions based on their values. Process A is accessing and working on some shared data in the kernel. At this point, a higher-priority Process B preempts the running, lower-priority Process A. Process B then changes the value of the shared data structure. A second process switch then occurs, with Process A resuming its operations after the prior interruption. Once Process A gets to run again, however, it will now make a decision based on a value determined by Process B.

In such a scenario, where two processes access the same data structures in critical sections, kernel preemption must be disabled. Alternatively, the operations on said data structures must be changed to become transactional/atomic in ways other than disabling preemption.

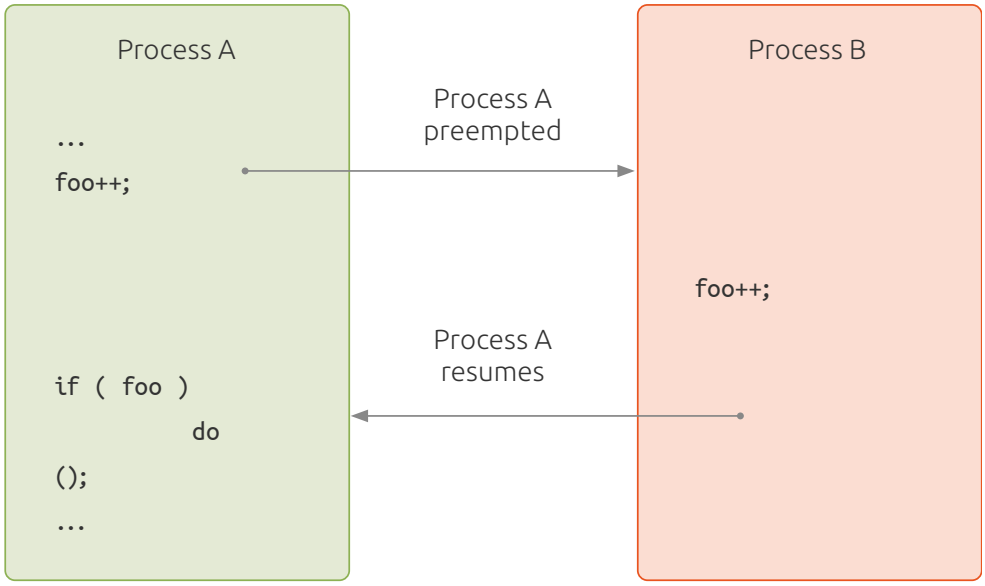


Figure 1: Two processes operating on common variables in a preemptible kernel.

For real-time processing in Linux to be a reality, then, the kernel must be able to preempt the current thread of execution while placing checks at strategic locations. The difficult tradeoff is to make the Linux kernel preemptible to reduce latency and achieve real-time compute capabilities, while disabling preemption around critical sections to avoid resulting in corrupted data.

Such a balance is hard to strike, and it involves poring over the entire kernel source code base to assess which data to protect from so-called concurrency.

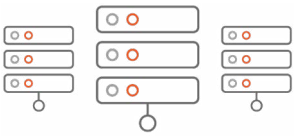
15 https://cdn.kernel.org/pub/linux/kernel/v2.5/ChangeLog-2.5.4

Intuitively, there are different “degrees” of determinism one can reach on the preemption spectrum. The next subsection will clarify how the different preemption models available in Linux directly affect the real-time compute capabilities of the kernel.

Preemption options in the Linux kernel

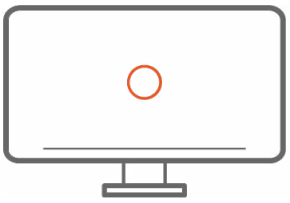
Following the introduction in kernel version 2.5.4, a few preemption models are now available in the mainline Linux kernel’s scheduler.[16] In this section, we’ll go over the key features and target applications for each.[17] Decision-makers should carefully evaluate their latency needs, the potential impact of missed deadlines, and the resource utilisation trade-offs when determining the most suitable option for their systems.

PREEMPT_NONE



The traditional Linux preemption model maximises the kernel’s raw processing power, irrespective of scheduling latencies.
As the default behaviour in standard kernels, the no-preemption case for server workloads is optimised for overall throughput for systems making intense computations like a server. Whereas PREEMPT_NONE can still provide decent latencies, there are no guarantees, and occasional longer delays can occur, as interrupts, exceptions, and system calls in the kernel code are never preempted.

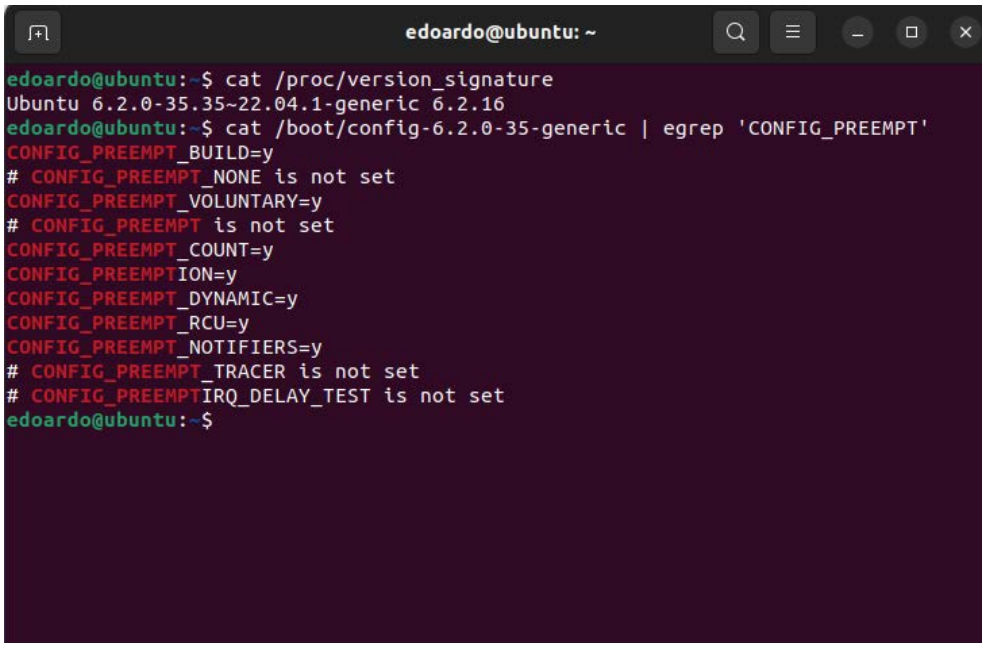
PREEMPT_VOLUNTARY



The PREEMPT_VOLUNTARY mode provides quicker application reactions to user input and targets desktop use. When operating in this mode, a low-priority process can “voluntarily” preempt itself even when executing a system call in kernel code. By enabling voluntary preemption points, PREEMPT_VOLUNTARY reduces the maximum latency of rescheduling at the price of slightly lower throughput. As a result, faster application reactions, even when under load, are perceived to run more smoothly.

A standard Linux desktop machine is likely to have PREEMPT_VOLUNTARY as the default preemption model. As per the screen capture below, the Linux machine is running the 22.04 LTS release of Ubuntu Desktop with version 5.19 of the Linux kernel. PREEMPT_VOLUNTARY is selected, as the kernel is for a desktop system.

16 https://www.kernel.org/
17 https://github.com/torvalds/linux/blob/master/kernel/Kconfig.preempt



From the above, CONFIG_PREEMPT (to enable the PREEMPT mode) is not set.

PREEMPT

Just like PREEMPT_VOLUNTARY, the PREEMPT mode enables voluntary preemption points in the Linux kernel. Furthermore, it makes kernel code preemptible when not executing in a critical section. With the addition of non-critical-section preemption, higher-priority processes can interrupt execution threads in low-priority processes even when executing a system call in kernel mode and when not about to reach a natural preemption point.

The reduced kernel latencies are suitable for desktop or embedded systems with latency requirements in the milliseconds range and they result in slightly lower throughput and runtime overhead to kernel code.

PREEMPT is the configuration flavour of mainline that provides the highest preemption level.[18] For extreme low-latency requirements and higher levels of determinism, one must look beyond mainline Linux.

PREEMPT_RT

As we mentioned earlier, PREEMPT_RT is the name of the patchset that implements a priority scheduler and other supporting real-time mechanisms.[19] PREEMPT_RT is nowadays the de-facto Linux real-time implementation but various mechanisms to bring real-time compute into the Linux kernel have been attempted through the years. Examples include Xenomai and RTAI. Other approaches to bringing real-time in Linux often implement a 'co-kernel' running concurrently with a soft Linux kernel; PREEMPT_RT differs by affecting the kernel itself.

The upstream project is maintained by the Linux Foundation and its development closely resembles the standard procedure for the mainline kernel. Developers send patches to add new functionalities or fix bugs to the upstream community via the mailing list.[20]

18 https://www.youtube.com/watch?v=cZUzc0U1jJ4&t=14444s&ab_channel=LinuxPlumbersConference
19 https://wiki.linuxfoundation.org/realtime/start
20 https://wiki.linuxfoundation.org/realtime/communication/mailinglists

If approved, the maintainers apply them to the real-time patchset in the Git repo.[21]

While most of the real-time logic is in mainline, not all the patches are upstream yet. A relevant portion, especially for locking, is still in a patch set form. The PREEMPT_RT locking code, representing a bulk of the outstanding real-time patches, was merged in Linux 5.15, but there is still more work to be upstreamed.

Although PREEMPT_RT doesn't necessarily result in the lowest latencies possible, an unbounded latency would represent a bug. PREEMPT_RT provides more deterministic response times by making the kernel more preemptible. The PREEMPT_RT patchset strives for full preemption in the Linux kernel, including in critical sections (except for very low-level, critical code paths like entry code and interrupt handling). The goal with PREEMPT_RT is to make all code running in kernel mode involuntarily preempted at any time by bringing most execution contexts under scheduler control. PREEMPT_RT does so by replacing locking primitives like spinlocks with variants aware of priority inheritance, enforcing interrupt threading and introducing mechanisms to break up long non-preemptible sections.

The PREEMPT_RT patch can also adopt other mechanisms to make the kernel deterministic. For example, it can be used to replace mutexes in the mainline Linux kernel with rt_mutexes. Chapter 4 in this guide will explore this approach in more detail.[22]

As argued earlier, a real-time kernel on its own does not necessarily make a system real-time. Furthermore, a mainline Linux kernel may be less costly to maintain than an invasive set of patches. Is PREEMPT_RT then always necessary to reach so-called real-time behaviour, or would a kernel with e.g. PREEMPT suffice for some applications? There are distinct advantages and trade-offs in terms of latency, responsiveness, and resource utilisation that one must be aware of. The next section discusses some of those considerations.

Real-time or low-latency kernel?

A real-time Linux kernel with the PREEMPT_RT patchset is specifically designed for applications with tight latency constraints. It aims to provide deterministic and predictable response times, making it suitable for critical systems where missed deadlines can have severe consequences.

However, it is important to note that real-time performance can come at the expense of optimised throughput. Real-time Linux kernels prioritise minimising response latency to events, which may negatively affect throughput and resource utilisation. These kernels may consume a significant portion of system resources, potentially impacting overall performance. On the other hand, low-latency kernels can offer a more balanced solution, favouring throughput and user space CPU access.

A low-latency kernel, for example low-latency Ubuntu, can reduce overhead while maintaining responsiveness.[23] Low-latency Ubuntu efficiently handles real-world, low-latency and low-jitter workloads.[24] The low-latency Ubuntu kernel incorporates the maximum preemption available in the mainline kernel

21 https://git.kernel.org/pub/scm/linux/kernel/git/rt/linux-stable-rt.git/
22 https://lwn.net/Articles/146861/
23 https://packages.ubuntu.com/search?suite=all&searchon=all&keywords=lowlatency&_ga=2.8278002.1788117502.1692006599-1471687884.1676892496
24 https://ubuntu.com/engage/kernel-telco-nfv

(PREEMPT), coupled with a timer interrupt frequency of 1000 Hz (higher than the timer granularity of a generic Ubuntu Linux kernel).

The choice between a low-latency Linux kernel and a real-time Linux kernel with the PREEMPT_RT patchset depends on the specific requirements of the system. By understanding these factors, decision-makers can make informed choices based on their needs. The latency requirements of the system and the potential impact of missed deadlines are crucial. If the latency needs are exceptionally tight and missing a deadline can lead to system failure, a real-time Linux kernel might be necessary. Conversely, if the latency requirements are less stringent, and the consequences of missed deadlines are less severe, a low-latency kernel may suffice. Audio/video systems for entertainment are a typical example of this category of latency requirements. The delays inherent to human perception are large enough that a “proper” real-time system isn’t necessary.

If your use case requires a real-time system and you’re considering a Linux distribution with real-time capabilities, Real-time Ubuntu offers support for real-time compute with PREEMPT_RT with a 10 year security update commitment. The next chapter will explain how you can guarantee bounded execution times for business-critical applications with the real-time kernel built into Ubuntu.

Using Real-time Ubuntu

Real-time Ubuntu is an Ubuntu 22.04 LTS kernel with the PREEMPT_RT patches integrated by Canonical. It relies on the Linux Foundation's upstream project's 5.15-rt patch set, maintained by Canonical's Joseph Salisbury.[25]

Real-time Ubuntu with the out-of-tree PREEMPT_RT patches entered general availability in February 2023.[26] By minimising the non-preemptible critical sections in kernel code, the kernel is more preemptive than mainline. Real-time Ubuntu reduces kernel latencies as required by the most exacting workloads by prioritising critical tasks, and helps ensure a time-predictable task execution.

By minimising response latency, Real-time Ubuntu aims to ensure time-sensitive applications receive prompt processing by providing deterministic responses to external events.

How to enable Real-time Ubuntu

Real-time Ubuntu is available via Ubuntu Pro, Canonical's comprehensive enterprise security and compliance subscription, covering all aspects of open infrastructure.[27] A free tier is available for personal and small-scale commercial use, in line with the company's community commitment and mission to ease open-source access and consumption.

With an Ubuntu Pro subscription, launching the kernel is as easy as:[28]

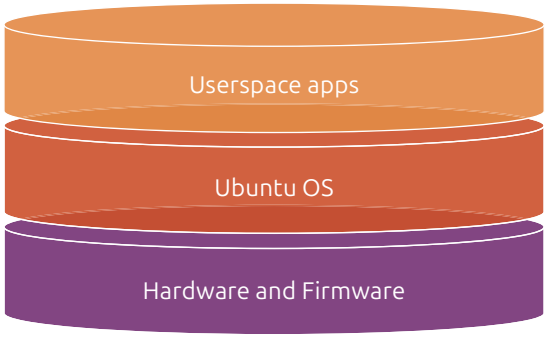
```
pro attach <token>
pro enable realtime-kernel
```

Canonical's enterprise-grade software support and long-term maintenance for the real-time Ubuntu kernel ensure product longevity. Device manufacturers can focus on their business drivers, shortening time-to-market by relying on an enterprise-grade, real-time Linux kernel supported over 10 years.

Silicon-optimised Real-time Ubuntu

Besides the OS, each layer of a stack must prioritise real-time workloads and allocate resources to deliver the low-latencies and tight time synchronisation needed to support deterministic processing. From optimised hardware and other components, building a real-time system encompasses every layer from the silicon to the OS, encompassing networking and userspace applications.

25 https://cdn.kernel.org/pub/linux/kernel/projects/rt/5.15/
26 https://ubuntu.com/blog/real-time-ubuntu-is-now-generally-available
27 https://ubuntu.com/pro
28 https://ubuntu.com/pro/dashboard



Since standalone hardware or software components are not sufficient, Canonical and Intel have joined forces to deliver an out-of-the-box real-time solution, now generally available on Intel Core processors.[29]

In this chapter, we will delve deeper into how this combined Intel and Canonical stack delivers enterprise-grade performance for the time-bound workloads of industrial systems. The technologies we integrate with are Intel® Time Coordinated Computing (Intel® TCC) and IEEE 802.1 Time Sensitive Networking (TSN). [30,31]

At the end of this chapter, we introduce a scalable testbed to emulate a real-world industrial usage scenario, using Real-time Ubuntu and Intel's technologies.

Launching silicon-optimised Real-time Ubuntu on Intel SoCs

Canonical and Intel's integrated approach offers a pre-integrated stack with enabled and supported technologies, ensuring ease of use for developers.[32] With the addition of TCC and TSN, enterprises can achieve enhanced performance, time synchronisation, and temporal isolation at the silicon layer.

Optimised real-time Ubuntu on Intel hardware meets stringent release criteria, having undergone extensive manual, automated and stress testing.[33] Dedicated teams at Canonical will continue testing the images end-to-end with the support of Intel engineers, executing rigorous management of all Linux kernel CVEs, and reviewing and applying all relevant patches. With long-term support and security maintenance, customers can enjoy extended product availability and reliability for their embedded Linux deployments in large-scale production environments.

Empowering industrial systems with Intel® TCC, TSN and real-time Ubuntu

TSN primarily focuses on the network space, ensuring that time-sensitive applications and workloads receive the necessary processing and network priorities. On the other hand, Intel® TCC is the equivalent solution designed specifically for the latest Intel processors. It optimises the entire System-on-Chip (SoC) to deliver the time-sensitive and deterministic needs of real-time workloads.

29 https://ubuntu.com/blog/optimised-real-time-ubuntu-is-now-generally-available-on-intel-socs
30 https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/real-time/overview.html
31 https://1.ieee802.org/tsn/
32 https://ubuntu.com/download/iot/intel-iot
33 https://ubuntu.com/engage/realtime-webinar-ga

Intel's TSN and TCC capabilities are at the forefront of real-time systems with cutting-edge features and optimisations that help address challenges unique to industrial systems. For instance, by enabling deterministic and low-latency communication over Ethernet, TSN ensures real-time data exchange between various components. Sensor readings and control signals can be prioritised and delivered with minimal delay, allowing coordination and the smooth functioning of production processes in a factory environment.

Furthermore, mixed-criticality systems are common in industrial settings, where different applications can have varying levels of criticality. Whereas some tasks might be safety-critical, others could be more time-flexible.

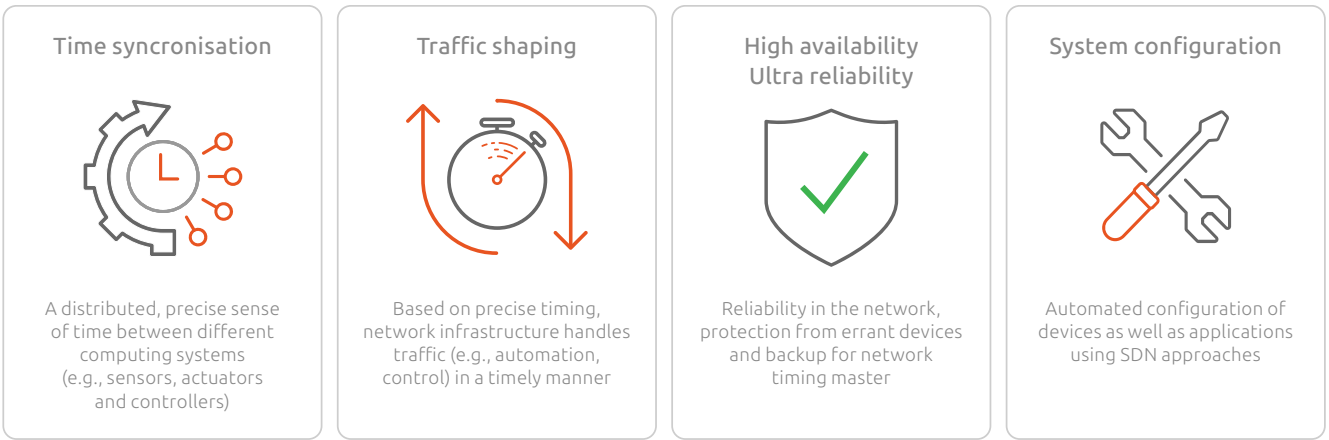
With time awareness and traffic shaping, TSN allows you to segregate the various data stream types from the factory floor based on their priority levels. Similarly, Intel® TCC enhances the security of such mixed-criticality systems by providing hardware-based protection and isolation, shielding critical operations from potential security threats.[34]

In the following two subsections, we will delve deeper into the two technologies.

Deterministic and time-sensitive networking

An integral part of industrial systems' real-time capabilities is the connection to the network. TSN provides a network infrastructure capable of supporting both general workloads and real-time demands with deterministic and time-sensitive characteristics.

To achieve this, time synchronisation is crucial, ensuring that different network nodes are properly synchronised. Similarly, timeliness in terms of workload and data scheduling is essential. By providing time synchronisation across the network, traffic shaping, and continuous availability, optimised real-time Ubuntu support for TSN on Intel platforms ensures the uninterrupted and reliable operation of real-time workloads.



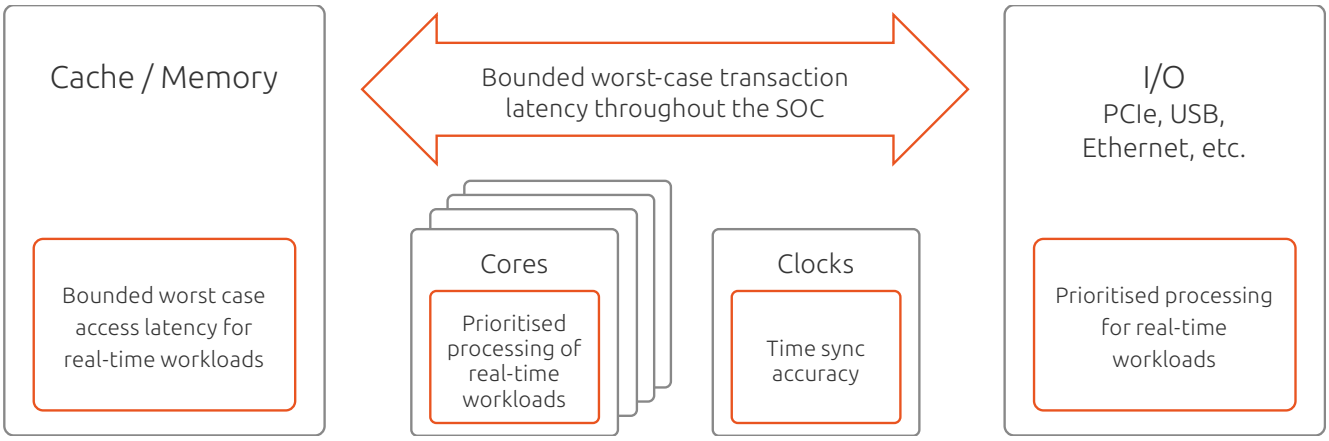
Core Elements of TSN

34 https://www.intel.com/content/www/us/en/developer/articles/technical/real-time-systems-measurement-library.html

Timeliness and temporal isolation

Intel® TCC encompasses a comprehensive set of optimisations throughout the entire SoC to ensure real-time workloads are processed correctly.

Intel® TCC helps meet the stringent requirements of real-time workloads while concurrently running multiple workloads on the same SoC, offering an efficient and isolated environment. Intel® TCC includes cache and memory optimisations to allocate adequate resources, manage interrupt priorities, and synchronise various clocks throughout the SoC. Furthermore, it enables temporal isolation, protecting real-time applications from the impact of other processes and events within the system.



Intel® TCC: Real-time Optimisations throughout the SoC

By allocating sufficient cache and memory resources, and with TCC's temporal isolation and time synchronisation, Intel SoCs running real-time Ubuntu are well-equipped to handle industrial systems' stringent real-time requirements.

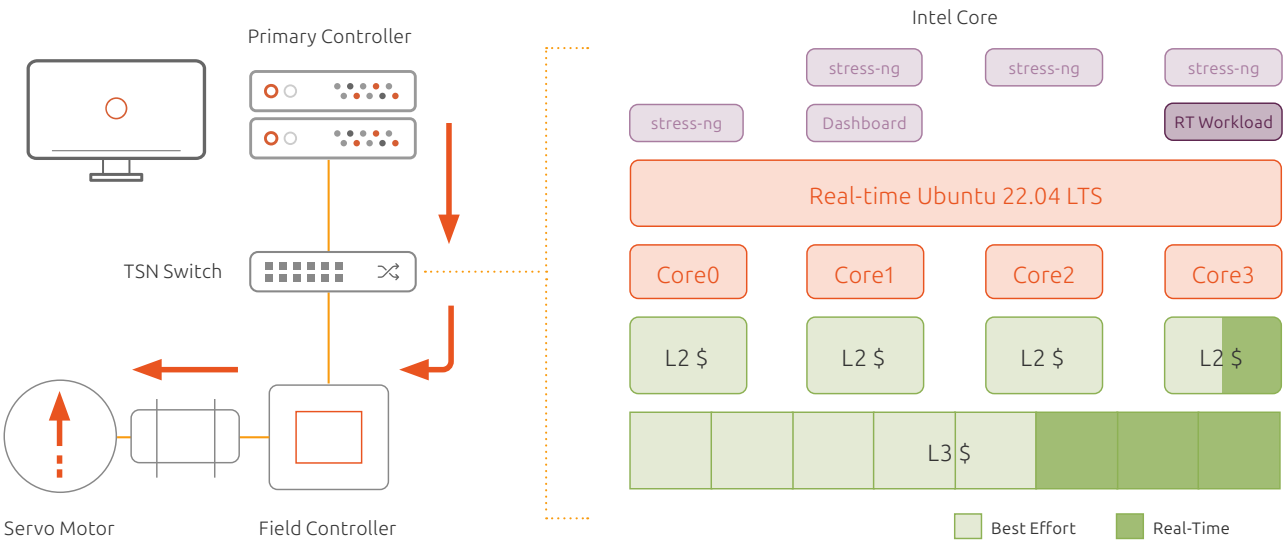
You can deploy real-time Ubuntu on Intel processors via Ubuntu Pro:

```
pro attach <token>
pro enable realtime-kernel --variant intel-iotg
```

Emulating real-world scenarios with real-time Ubuntu on Intel SoCs

Intel and Canonical collaborated to prepare a scalable testbed to emulate a real-world industrial usage scenario, using Real-time Ubuntu and Intel's technologies. The demo showcases a servo controlled by an industrial-grade PC and a display showing what's happening on the machine.

The primary controller (Kontron KBox C-103-BB-TGL with Intel® Core) emulates a workload consolidation scenario with both best-effort and real-time workloads. The real-time workload calculates motor parameters in defined cycle time and publishes them via OPC UA pub/sub over TSN. Best-effort workloads like graphic rendering on a UI run on all cores. The field controller (Cyclone® V SOC Dev Kit Intel® SOC FPGA Embedded Development Suite) controls the motor based on the parameters from the primary controller acquired over TSN.



Without real-time optimisations in the system, best-effort workloads could interfere with real-time workloads and may lead to missed deadlines and non-deterministic motor stepping.

Canonical and Intel's joint solution, on the other hand, increases the temporal isolation between best effort and real-time workloads by applying Intel® TCC features (e.g. TCC BIOS Option, and Cache QoS), and real-time Linux kernel configuration guidance to guarantee high determinism and controlled motor stepping.[35]

As soon as the CPU is getting more utilisation, more traffic and coordination are needed. Intel® TCC on the hardware enables the isolation of real-time workloads in multiple ways. For instance, Intel® TCC ensures a predictable response time by isolating the cache access, or the execution from non-real-time workloads.

The test demonstrates real-world mixed-criticality real-time scenarios. By enabling real-time Ubuntu and isolating the workloads through Intel's TCC features, enterprises can get predictable results for industrial-grade use cases.

By now, the reader should have a solid grasp of the basics of real-time Linux, its applications, and its relevance in various scenarios. The concluding chapter will dive into some technical details—specifically, the ins and outs of configuring Real-time Ubuntu, fine-tuning its performance, and selecting boot parameters.

35 https://ubuntu.com/blog/real-time-kernel-tuning

Technical deep-dive in a real-time Linux OS

The following sections function as a real-time Linux toolbox. While you might not require every tool now, it's good to be aware of their existence for when the need arises. Whether you're planning to get hands-on with the technical aspects or want to know what's under the hood, this chapter has got you covered.

Feel free to skim through, focus on the relevant sections that catch your eye, or bookmark it for future reference. The chapter aims to provide valuable resources, ranging from technical details to configuration options and test suites that may be useful when deploying Real-time Ubuntu.

Overall, the objective is to provide technically-minded readers and their development teams with a solid understanding of the engineering side and a reliable resource to turn to when encountering technical challenges.

Testing Canonical's real-time kernel

Canonical's real-time Ubuntu relies on two primary test suites, rt-tests and stress-ng.[36,37]

rt-tests, maintained upstream in a Git repository, includes oslat and Cyclictest, the primary test suite upstream used to establish a baseline and determine if there is regression.[38,39,40] Canonical routinely runs stress-ng every SRU cycle to check for regression and changes in kernel stability as well.[41,42]

The real-time Ubuntu kernel relies on extensive testing, often in combination. For example, stress-ng to put a load on the system and Cyclictest to measure its latency. Furthermore, Canonical also tests the real-time kernel via partner-provided programs, like Intel's Jitter Measurement Tool (provided as a package and not upstream).

The role of the scheduler in a real-time kernel

The scheduler is a key component of a real-time system. In the Linux kernel, a few scheduling classes, like Early Deadline First, Real-Time, and the Completely Fair Scheduler, are available, with different scheduling policies within each class, as per the table below:

36 https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/rt-tests
37 https://manpages.ubuntu.com/manpages/jammy/man1/stress-ng.1.html
38 https://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git/
39 https://manpages.ubuntu.com/manpages/jammy/man8/oslat.8.html
40 https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start
41 https://github.com/ColinIanKing/stress-ng
42 https://wiki.ubuntu.com/StableReleaseUpdates

Scheduling Class	Scheduling Policy
EDF	SCHED_DEADLINE
RT	SCHED_RR
	SCHED_FIFO
CFS	SCHED_OTHER
	SCHED_BATCH
	SCHED_IDLE
IDLE	-

The runqueue contains per-processor scheduling information and is the basic data structure in the scheduler. It lists the runnable processes for a given processor and is defined as a struct in `kernel/sched.c`.

The scheduler can be run via a call to the `schedule()` function. The Linux kernel will then sequentially check the EDF, RT and CFS runqueue for waiting tasks. Alternatively, the system will do an idle loop if no tasks ready to run are present.

Early Deadline First in the kernel scheduler

EDF's scheduling policy, SCHED_DEADLINE, is deadline-based.[43] Hence, after calling the `schedule()` function, the scheduler will run whichever task in the runqueue is closest to the deadline. Whereas in the POSIX (and RT class) approach, the highest-priority task gets the CPU, the runqueue's process nearest its deadline is the next one for execution in EDF. A notable advantage of EDF over priority-driven scheduling is that it can employ a larger share of the processing power in a single-processor system. While the limit depends on the nature of the task, a rough estimate for the simplest case in a priority-driven scheme is up to 70%. Theoretically, EDF can go up to 100%.

The unintended consequence and potential issue with the SCHED_DEADLINE policy is that if a task misses its deadline, it would keep running, causing a negative domino effect with the follow-on processes. Whereas continuing the work anyway is often desirable, different applications will have different needs. In most scenarios, one can get around this with interrupt-driven killing of the task overrunning its deadline. Overall then, one must pay close attention to the system and application requirements when using SCHED_DEADLINE.

Real-time in the kernel scheduler

Real-time Ubuntu relies on the RT class, a POSIX fixed-priority scheduler, which provides the FIFO and RR scheduling policies, first-in-first-out and round-robin, respectively.[44] In particular, real-time Ubuntu uses the SCHED_RR policy. SCHED_RR and SCHED_FIFO are both priority-based: the higher priority task will run on the processor by preempting the lower priority ones.

43 https://docs.kernel.org/scheduler/sched-deadline.html
44 https://man7.org/linux/man-pages/man7/sched.7.html

The difference between the FIFO and RR schedulers is evident when two tasks share the same priority. In the FIFO scheduler, the task that arrived first will receive the processor, running until it goes to sleep. On the other hand, in the RR scheduler, the tasks with the same priority will share the processor in a round-robin fashion.

The danger with the round-robin scheduling policy is that the CPU may spend too much time in context switching because the scheduler assigns an equal amount of runtime to each task. One can remediate such downsides by properly tuning a real-time kernel and focusing on how long tasks will run and the type of work they will do.

Completely Fair Scheduler and Idle

Finally, the generic kernel uses the CFS by default, whereas IDLE comes in handy when the system is not performing any action.[2]

Assigning scheduling policies in code

Let's now get our hands dirty and dive into the code directly.
Assigning a task to a specific policy type is relatively straightforward. If using POSIX threads, one can set the policy when calling the `pthread_attr_setschedpolicy function`, as per the example below with SCHED_FIFO:[45]

```
int main(int argc, char* argv[])
{
        struct sched_param param;
        pthread_attr_t attr;
        pthread_t thread;
        int ret;

…
…
…

ret = pthread_attr_setschedpolicy(&attr, SCHED_FIFO);
        if (ret) {
                printf("pthread setschedpolicy failed\n");
                goto out;
        }

…
…
…
```

45 https://docs.kernel.org/scheduler/sched-design-CFS.html

An alternative code snippet with comments:

```
 if (pthread_attr_setschedpolicy(&attr, SCHED_FIFO) != 0) {
           printf("pthread_attr_setschedpolicy: %s\n", strerror(errno));
           exit(1);
       }
    /* Always assign a SCHED_FIFO priority below 50-99.
 * Kernel threads run in that range.
 * It's never a good idea to use fifo:99 for a realtime application; the
 * migration thread uses fifo:99 and all the interrupt threads run at
 * fifo:50.  Refer to mail on rt mailing list
 */
       struct sched_param param;
       memset(&param, 0, sizeof(param));
       param.sched_priority = MIN(5, sched_get_priority_max(SCHED_FIFO));
```

Another way is to set the policy, whether SCHED_DEADLINE, SCHED_RR or SCHED_FIFO, in a `sched_attr` structure:

```
struct sched_attr attr = {
    .size = sizeof (attr),
    .sched_policy = SCHED_DEADLINE,
    .sched_runtime = 10 * 1000 * 1000,
    .sched_period = 2 * 1000 * 1000 * 1000,
    .sched_deadline = 11 * 1000 * 1000
};
```

In the above, the task will repeat every two seconds and can be up to 11 µs late. The thread function would then be:

```
sched_setattr(0, &attr, 0);
for (;;) {
    printf("sensor\n");
    fflush(0);
    sched_yield();
};
```

Another practical piece of code assigns priority to a thread. This can be done by directly passing a priority number:

```
param.sched_priority = 49;
        ret = pthread_attr_setschedparam(&attr, &param);
        if (ret) {
                printf("pthread setschedparam failed\n");
                goto out;
```

Assigning a sensible priority number is particularly important when working with the priority-based round-robin and FIFO policies.

An application should never run at priority 90 or higher, as that is where critical kernel threads run. Similarly, watchdogs and migration run at priority 99. Running a task at priority 99 will likely result in the overall system locking up. Hence, one should strive to set a priority below the range of 50-99 when writing a program.

Locks in a real-time kernel

There are two primary types of locks: blocking locks and spinning locks.

Blocking Locks

The primary characteristic of blocking locks is that the tasks holding them can be put to sleep. Among examples of blocking locks there are counting semaphores, (per-CPU) Reader/Writer semaphores, mutexes and WW-mutexes and RT-mutexes. Of those, RT-Mutex is the only blocking lock that will not lead to priority inversion, covered in the following section.

These lock types are then converted to sleeping locks, e.g. local_lock (often used to protect CPU-specific critical data), spinlock_t and rwlock_t, when enabling preemption in a real-time Linux kernel. Further details on locking primitives and their rules are available in the Linux kernel's documentation.[46]

Spinning Locks

Let's now consider spinning locks. To understand their advantages, it's worth remembering classical spin locks can't sleep and implicitly disable preemption and interrupts.[47] In turn, this can cause unbounded latencies, which are undesirable for real-time applications, as there is no guaranteed upper boundary of execution time. Furthermore, the lock function may have to disable soft or hardware interrupts depending on the context.

In a real-time kernel, classical spin locks convert to sleepable spinlocks and are renamed raw_spinlocks. Hence, a developer may have to recode their applications and drivers to use raw spinlocks in a kernel with PREMPT_RT, depending on whether or not a spin lock is allowed to sleep.

Among spinning locks, reader/writer locks are also available. In particular, rwlock_t is a multiple reader and single writer lock mechanism. Non-PREEMPT_RT kernels implement rwlock_t as a spinning lock, with the suffix rules of spinlock_t applying accordingly.

Processes and threads

Among the reasons why PREEMPT_RT is not in mainline yet is that much of the locking within the kernel has to be updated to prevent priority inversion from occurring in a real-time environment.[48] The present and the following section will introduce unbounded priority inversion and the need for priority inheritance.

Unbounded priority inversion

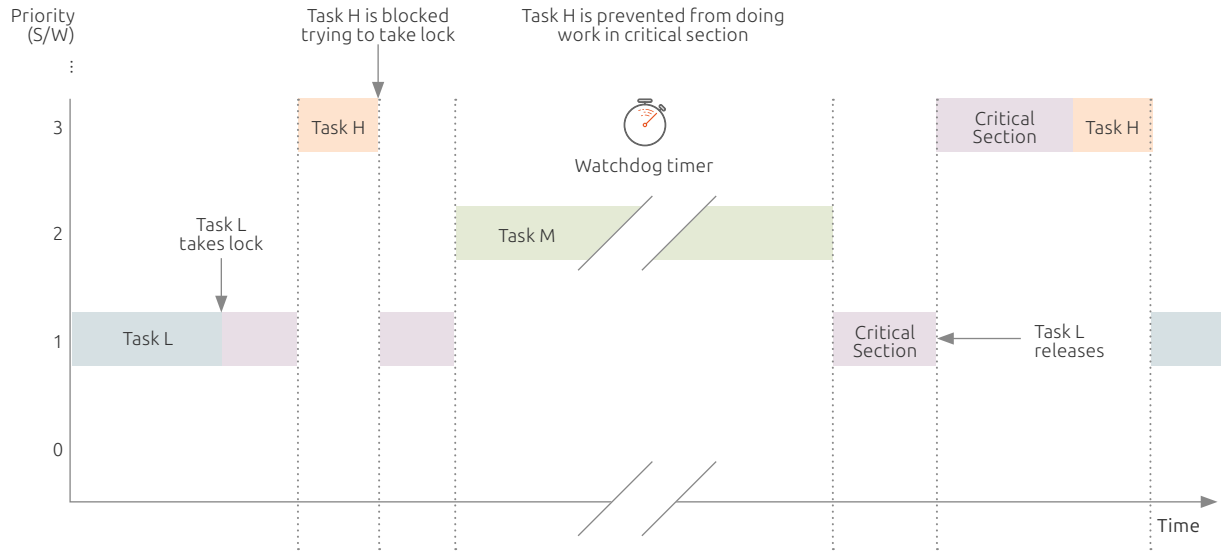
Let's begin with priority inversion by looking at the diagram sketched below. Three tasks, L, M, and H, with varying priority levels, low, medium and high, are present in the kernel and about to contest for CPU access.

The low-priority task L runs until it takes a lock; in the diagram below, the blue bar turns red. After acquiring it, task L holds the lock and begins entering some critical sections within the kernel.

46 https://docs.kernel.org/locking/locktypes.html
47 https://ubuntu.com/blog/what-is-real-time-linux-part-iii
48 https://www.youtube.com/watch?v=cZUzc0U1jJ4&t=14444s&ab_channel=LinuxPlumbersConference

Once the higher-priority task H appears, it preempts task L and starts to run. At this point, task H would like to acquire the same lock task L is holding.[49] As it can't do so, the higher-priority task H goes to sleep and waits for the lower-priority task L to release the lock. Task L thus continues running while Task H is sleeping. In such a setup, priority inversion can occur if a medium-priority task M comes along and preempts task L. Once task M starts running, the high-priority task H will potentially wait for an unbounded amount of time, preventing it from doing work in a critical kernel section. Improving the flexibility to preempt tasks executing within the kernel would thus help guarantee an upper time boundary.

Unbounded Priority Inversion



Unbounded Priority Inversion [50]

In this specific example, task M finishes running and releases the CPU - where the horizontal bar turns from green to red in the drawing - allowing task L to start running again while still holding the lock. Only once task L releases it, task H will wake up and acquire the lock, starting its work within the critical section.

Priority inversion occurred on the Mars Rover, and it is a critical challenge for developers and engineers working with real-time systems.[51] With unbounded priority inversion, the need for priority inheritance becomes clear.

Priority Inheritance

A real-time Linux kernel resolves the unbounded latencies of priority inversion via priority inheritance.

The diagram below helps illustrate the mechanism. As before, the low-priority task L starts running and acquires the lock. Similarly to the previous scenario, task H wakes up and starts running, but it is soon blocked while attempting to get the lock.

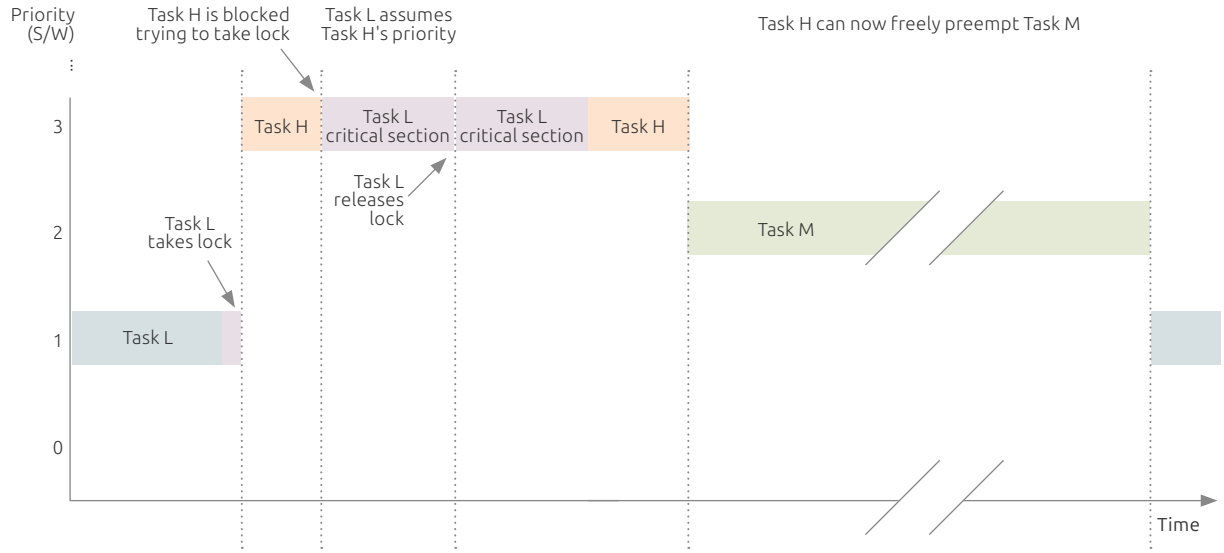
49 https://ubuntu.com/blog/what-is-real-time-linux-ii
50 https://www.digikey.com/en/maker/projects/introduction-to-rtos-solution-to-part-11-priority-inversion/abf4b8f7cd4a4c70bece35678d178321
51 https://www.youtube.com/watch?v=pr9J0pxHI6c&ab_channel=PhilKoopman

The high-priority task H wants to take the same lock held by the low-priority task L. Differently than in the priority inversion's case, and instead of H going to sleep and waiting, priority inheritance occurs, with L acquiring H's priority. The low-priority task L can now run with the same priority as task H, enabling it to finish its work in the critical section and then release the lock. The inheritance mechanism centres around boosting the lower task's priority, giving it one higher than the upcoming medium priority task M, which would cause unbounded latencies.

Once task L finishes its critical section work, task H acquires the lock, where the red bar turns orange. Whenever task H completes, it will, in turn, release the lock. Only now can the medium-priority task M come along and start running.

If needed, the higher-priority task H could further preempt task M to finish its processing. Priority inheritance in a real-time kernel solves the issue of task M starting to run between tasks H and L, which would give rise to unbounded latencies and priority inversion.

Priority Inheritance



Priority Inheritance [50]

Tuning a real-time kernel

Once you launch a real-time kernel, you can start tuning. Tuning a real-time kernel is a complex endeavour, as each layer of a real-time stack must support deterministic processing.[52] From the hardware to the kernel, and finally through to the application, every level can be a source of latency. A real-time kernel on its own will not necessarily make a system real-time, as even the most efficient Real Time Operating System (RTOS) can be useless in the presence of other latency sinks. Specific tuning for each use case is required, and an optimal combination of tuning configs for one particular hardware platform may still lead to poor results in a different environment. Real-time Ubuntu does not guarantee a maximum latency as performance strictly depends on the system at hand. From networking to cache partitioning, every shared resource can affect cycle times and be a source of jitter. Setting up a real-time configuration to meet stringent low-latency requirements takes careful understanding and tuning.

52 https://ubuntu.com/blog/real-time-kernel-technical

What follows are considerations that may prove helpful in some real-time environments. Rather than recommended configurations leading to optimal performance, they are intended as starting points for a subsequent, iterative tuning process. Only the engineering team developing a real-time stack controls the deployment environment and can decide on the best tuning configuration options. The real-time developers architecting the overall hardware and software system are responsible for end-application tuning and optimisation of individual drivers for specific workloads.

Metrics and tools for tuning

The three primary metrics to monitor when tuning a real-time kernel are jitter, average latency and max latency.

The maximum latency is the key metric, and it is fundamental to know its value before running in production. A preemptive kernel aims to provide a deterministic response time to service events, with system failure in case of missed deadlines regardless of the system load. For instance, if the maximum latency for an airbag is 10 µs, any reaction time higher than the specified upper boundary results in system failure. When tuning, one must ensure the real-time system can process threads and processes within the maximum latency measured during each task period or the real-time application lifetime. Jitter is the difference between average and max latency over time.

It is worth mentioning one should first evaluate the isolated run-time of each user task in the system. These are probability curves, typically derived statically in critical cases (civilian aeronautics certification), and experimentally in more general cases. In less critical scenarios, typical tools to capture system resource statistics which are useful for tuning are ps, perf, irqtop, stress-ng, Cyclictest, irqstat, dstat, and watch_interupts.[53,54,55] One can monitor by, for instance, watching the interrupts at `/proc/interrupts` via the `watch` command as per:

```
watch -n1 -d “cat /proc/interrupts”
```

Similarly, the `ps` command can help determine information like the processors on which a task is running on and its priority level via:

```
ps -eo psr,tid,pid,comm,%cpu,priority,nice -T | sort -g | grep irq
```

TuneD, RTLA and ftrace are alternative tools that may assist when tuning. TuneD is a userspace tool to set parameters and tuning profiles without manually modifying the grub configuration file. It helps isolate CPUs and set iirqs. The real-time Linux Analysis (RTLA) tool, merged into upstream 5.17, can trace kernel latencies.[56,57] It is not available in Ubuntu 22.04 but can be backported; it requires libtracefs v1.3.0. Finally, ftrace to profile applications is also worthy of mention, and it helps assess latencies and performance issues occurring outside of user-space.[58]

53 https://www.man7.org/linux/man-pages/man1/ps.1.html
54 https://www.man7.org/linux/man-pages/man1/perf.1.html
55 https://www.man7.org/linux/man-pages/man1/irqtop.1.html
56 https://docs.kernel.org/tools/rtla/index.html
57 https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4d66020dcef83314092f2c8c89152a8d122627e2
58 https://www.kernel.org/doc/Documentation/trace/ftrace.txt

BIOS options

Whereas the previous paragraphs mentioned jitter, average latency and max latency as the primary metrics to monitor, this section lists some BIOS options to look for when tuning a real-time kernel.

In a real-time system, the kernel is not the only possible source of latency. The applications, the hardware, firmware and BIOS for the hardware can also influence this.

In addition to tuning parameters and applications, a real-time developer needs to review and consider these BIOS options when setting up a low-latency environment:

- SMIs
- C state
- Intel speed step
- Turbo mode
- VTx
- VTd
- Hyperthreading

Identifying and tweaking kernel config options are arguably the most time-consuming, iterative activities to reduce latency when tuning. The following paragraphs will highlight and explain key tuning parameters and boot options. Starting with `CONFIG_NO_HZ_FULL` to omit scheduling-clock ticks and `CONFIG_RCU_NOCB_CPU` to enable callback offloading, we will cover relevant boot parameters, followed by an overview of kthread_cpus, to specify which CPU to use for kernel threads. Finally, we will provide reference code snippets for assigning real-time threads and processes to specific cores at runtime, as well as an example of adding config options `/etc/default/grub`.

Config options

This section highlights some tuning configs set at compile time when using a real-time kernel.

Scheduling-clock ticks

As `nohz=on` for tickless CPUs disables the timer tick on the specified CPUs, the `CONFIG_NO_HZ_FULL` config option indicates how the system will generate clock checks and will cause the kernel to avoid sending scheduling-clock interrupts to a CPU that is idle or has a single runnable task.

`nohz_full=<list of isolated cpus>` reduces the number of scheduling-clock interrupts, improving energy efficiency and reducing OS jitter.

When tuning a real-time system, it is customary to assign one task per CPU. By setting `CONFIG_NO_HZ_FULL= y` it will omit scheduling-clock ticks on CPUs that are either idle or that have only one runnable task.

Please note the boot CPU cannot run in `nohz_full` mode, as at least one CPU needs to continue to receive interrupts and do the necessary housekeeping. Further information can be found in the kernel.org documentation.[59]

59 https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt

RCU callbacks

Another config option is Read-copy-update (RCU) with no callbacks: `CONFIG_RCU_NOCB_CPU = y`.[60]

RCU carries out significant processing in Softirqs contexts, during which preemption is disabled, causing unbounded latencies.[61] These callbacks often free memory with memory allocators imposing large latencies when taking slow paths.

For example, `rcu_nocbs=1,3-4` would enable callback offloading on CPUs 1, 3, and 4. The real-time Ubuntu kernel sets it to all, as per `CONFIG_RCU_NOCB_CPU_ALL = y`

`rcu_nocb_poll` makes kthreads poll for callbacks instead of explicitly awakening the corresponding kthreads. The RCU offload threads will be periodically raised by a timer to check if there are callbacks to run. In this way, `rcu_nocb_poll` helps improve the real-time response for the offloaded CPUs by relieving them of the need to wake up the corresponding kthread. The above tuning configs are done at compile time. Once set, they can't be changed unless the kernel is recompiled. Preemptiveness can, on the other hand, be dynamically changed at run time.

The discussion so far exposed parameters for scheduling clock ticks and RCU config options. The remainder of this section will introduce further boot parameters, like `timer_migration` and `sched_rt_runtime`, as well as common approaches to handle IRQ interrupts.

Boot parameters

This section lists a few additional parameters worth monitoring when tuning a real-time Linux system with PREEMPT_RT, like `kthread_cpus`, `domain`, `isolcpus`, `timer_migration` and `sched_rt_runtime`. For a more thorough discussion, please refer to the kernel's documentation.[62]

Kernel threads like ksoftirqd, kworker and migration need to run on every CPU regardless of isolation, and `kthread_cpus` to define CPUs for kernel threads, can be particularly handy.

Further helpful config options to assess are `domain`, which removes the CPUs from the scheduling algorithm and `isolcpus`. The latter is used specifically for userspace or real-time processing. It isolates CPUs so they only run specific tasks, and will have a limited number of kernel threads to run. This way, housekeeping threads won't run on these CPUs which prevents them from being targeted by managed interrupts.

In a real-time system with multiple sockets, it helps to turn the `timer_migration` parameter off so as to prevent the timer from migrating between them. By setting `timer_migration = 0` in a multi socket machine, the time will stay assigned to a core.

The `echo 0 > /proc/sys/kernel/timer_migration` command will set the relevant information.

`sched_rt_runtime` is an important kernel parameter to specify the number of µs during which a real-time process can dominate a CPU. When running tasks on `isolcpus`, setting `kernel.sched_rt_runtime_us = -1` turns off throttling, allowing a process or real-time task to dominate the CPU indefinitely.

60 https://www.kernel.org/doc/html/latest/RCU/
61 https://github.com/torvalds/linux/blob/master/kernel/softirq.c
62 https://www.kernel.org/doc/html/v5.15/admin-guide/kernel-parameters.html

Setting it to -1 via: `echo -1 > /proc/sys/kernel/sched_rt_runtime_us` is often desired when tuning a system as all real-time processes or threads will be on cores 1-95 (and can thus be allowed to dominate the CPU). Please note limiting the execution time of real-time tasks per period can be dangerous on a generic system without isolated CPUs and is only advised on a real-time kernel.

IRQ Affinity

IRQ interrupts, often originating from device drivers, can cause issues with the tuning of real-time processes.[63] For instance, pushing a button on a keyboard or moving the mouse can cause interrupts which have to be processed by the CPU.

Common parameters to handle IRQ interrupts are `kthread_cpus`, to specify which CPU to use for all kernel threads, `irqaffinity` and `isolcpus`.
Every IRQ can be seen in `/proc/interrupts` and `irqaffinity=<list of non-isolated cpus>` assigns all IRQs to the specified core.

`/etc/systemd/system.conf` defines the IRQ affinity, which can be set as `CPUAffinity=0`. Checking the current IRQ affinity per interrupt is easy via a for loop through the IRQs listed in `/proc/irq`, followed by the `$i` IRQ number:

```
for i in {1..54} ; do cat /proc/irq/$i/smp_affinity; done
```

The file `smp_affinity` will then clarify which core or CPU the IRQs are assigned to (0 in this case).

Assigning threads to cores

This section provides examples of assigning real-time threads and processes to specific cores at runtime.

The `taskset` and `cset` commands, the POSIX function calls, or other software using the CPU affinity syscalls specify which CPUs to run drivers and real-time applications. Otherwise tasks will be assigned to any of the CPUs defined by `isolcpus`.

At runtime you can use the t`askset` command to pin a process to a specific CPU by specifying the `CPU_NUM` (cpu number) and PID as:

```
taskset -pc CPU_NUM[s] PID
```

The code snippet below exemplifies how to assign CPU 0 to 7 to a specific `cpuset`:

```
/* Set affinity mask to include CPUs 0 to 7. */
           CPU_ZERO(&cpuset);
           for (int j = 0; j < 8; j++)
               CPU_SET(j, &cpuset);
```

Communicating existing threads to only run on CPUs 0 to 7 can be done by setting the affinity:

```
s = pthread_setaffinity_np(thread, sizeof(cpuset), &cpuset);
/* Sets the CPU affinity mask for the given thread to the given set of
CPUs*/
```

63 https://www.kernel.org/doc/html/latest/power/suspend-and-interrupts.html

Adding params to grub

As an example to shed some further light, the tuning parameters discussed so far can be set by adding the following line to `/etc/default/grub`:

```
GRUB_CMDLINE_LINUX_DEFAULT=”console=tty0 console=ttyS0,115200 skew_tick=1
rcu_nocb_poll rcu_nocbs=1-95 nohz=on nohz_full=1-95 kthread_cpus=0
irqaffinity=0 isolcpus=managed_irq,domain,1-95 intel_pstate=disable
nosoftlockup tsc=nowatchdog”
```

The above refers to a 96-cores machine, where CPU core 0 is for housekeeping and kernel work, and will handle all IRQs. 1-95 is the list of CPUs where the isolation options are applied for real-time userspace applications. Furthermore, `nohz` disables the timer tick on the specified CPUs and `domain` removes the CPUs from the scheduling algorithms.

Tuning example

A generic Linux kernel will often lead to spikes in latency. Similarly, while the average latency will decrease, an out-of-the-box real-time kernel with no tuning can have maximum latency values higher than desired.

Performance strictly depends on the system at hand. However, a good rule of thumb for the maximum latency of a real-time Linux system is around 100 µs. On the other hand, real-time Ubuntu with a few changes to boot parameters can result in an average latency of 2-3 µs and max latency dropping considerably, with all values well under 100 µs.

The easiest way to get a feel for the improved results is to compare a generic Linux kernel with the non-tuned, default real-time Ubuntu and a tuned version.

Start by creating pre-defined grub files in which to put boot parameters. Those strictly depend on the specific use case - for instance, after a bit of tuning, two housekeeping CPUs may be the most effective. A real-time developer must figure this out on their system.

Assuming the below example combination leads to optimal results, add these boot parameters:

```
rcu_nocb_poll rcu_nocbs=2-95 nohz=on nohz_full=2-95 kthread_cpus=0,1
irqaffinity=0,1 isolcpus=managed_irq,domain,2-95
```

Then, set the IRQ affinity in /etc/systemd/system.conf and disable throttling:

```
sudo sysctl kernel.sched_rt_runtime_us=-1
echo -1 > /proc/sys/kernel/sched_rt_runtime_us
```

You can now disable timer migration:

```
sudo sysctl kernel.timer_migration=0
echo 0 > /proc/sys/kernel/timer_migration
```

Finally, update grub and reboot. At this point, you can run Cyclictest to get baseline values for average and max latency:

```
sudo cyclictest --mlockall --smp --priority=80 --interval=30 --distance=0
```

Considerations after tuning

Tuning is an iterative process, and it is advisable to tweak one parameter at a time while measuring the results. Depending on the environment and how stringent the latency requirements are, tuning and testing a real-time system to evaluate and check its performance can take multiple days and potentially weeks. Despite the effort, tuning can bring beneficial effects and tangible improvements in latency results.

When building your real-time applications, this material from the Linux Foundation may also be helpful:

- How to build a simple real-time application
- How to build a basic cyclic application
- Memory for real-time applications
- CPU idle power saving methods for real-time workloads
- Real-time latency debugging
- Real-time tools and utilities

Conclusion

With forecasts suggesting that nearly 30% of the world's data will require real-time processing by 2025, the demand for real-time computing solutions across the technology landscape is rising.

Real-time systems must operate within well-defined time boundaries, and their inability to meet these constraints will result in a system failure. A real-time Linux kernel aims to provide deterministic responses to external events and plays a critical role in diverse industries. Real-time Linux is a key enabler across sectors with stringent precision requirements, from industrial to telecommunications, where timely execution of tasks can have critical consequences.

By bringing long-term support to the workloads of modern enterprises, Real-time Ubuntu by Canonical supports industries on their journey to meet ever-demanding latency requirements. Integrating the PREEMPT_RT patches into Ubuntu increases the kernel's preemptiveness, reducing latencies and ensuring time-predictable task execution. This commitment to minimising response latency aligns perfectly with the needs of time-sensitive applications, providing deterministic responses to external events.

Despite the high demand, it is important to realise a real-time Linux kernel alone does not guarantee real-time behaviour. Achieving real-time performance required careful tuning of the entire system stack, from hardware to the kernel, networking layer, and applications. This is why Canonical works across its rich and vibrant technology ecosystem and partners with key market leaders like Intel to deliver an out-of-the-box real-time solution. Real-time Ubuntu optimised on Intel silicon offers a pre-integrated stack for the time-bound workloads of industrial systems.

As companies shift their workloads to the edge and navigate the accelerating adoption of real-time computing, Real-time Ubuntu is ready to deliver the precision and predictability demanded by mission-critical workloads.

Author @Canonical:
Edoardo Barbieri, Product Manager
edoardo.barbieri@canonical.com

References

1. https://www.readkong.com/page/the-digitization-of-the-world-from-edge-to-core-8666239
2. https://docs.kernel.org/scheduler/sched-design-CFS.html
3. https://groups.google.com/g/comp.realtime/c/BuXZqYnm3tg/m/iwtllaqGyHIJ
4. https://ieeexplore.ieee.org/document/8277153
5. https://canonical.com/blog/real-time-ubuntu-aws
6. https://www.youtube.com/watch?v=IWdQETXN-0k&ab_channel=CanonicalUbuntu
7. https://canonical.com/blog/industry-4
8. https://ubuntu.com/engage/bridge-the-IT-OT-gap-Industry4
9. https://ubuntu.com/engage/linux-embedded-applications-whitepaper
10. https://ubuntu.com/engage/intel-flexran-ubuntu-real-time-kernel
11. https://ubuntu.com/blog/what-is-openran?
12. https://www.intel.com/content/www/us/en/developer/videos/an-overview-of-flexran-sw-wireless-access-solutions.html
13. https://ubuntu.com/blog/canonical-announces-ubuntu-22-04-lts-support-for-flexran-reference-software
14. https://www.brighttalk.com/webcast/6793/571067
15. https://cdn.kernel.org/pub/linux/kernel/v2.5/ChangeLog-2.5.4
16. https://www.kernel.org/
17. https://github.com/torvalds/linux/blob/master/kernel/Kconfig.preempt
18. https://www.youtube.com/watch?v=cZUzc0U1jJ4&t=14444s&ab_channel=LinuxPlumbersConference
19. https://wiki.linuxfoundation.org/realtime/start
20. https://wiki.linuxfoundation.org/realtime/communication/mailinglists
21. https://git.kernel.org/pub/scm/linux/kernel/git/rt/linux-stable-rt.git/
22. https://lwn.net/Articles/146861/
23. https://packages.ubuntu.com/search?suite=all&searchon=all&keywords=lowlatency&_ga=2.8278002.1788117502.1692006599-1471687884.1676892496
24. https://ubuntu.com/engage/kernel-telco-nfv
25. https://cdn.kernel.org/pub/linux/kernel/projects/rt/5.15/
26. https://ubuntu.com/blog/real-time-ubuntu-is-now-generally-available
27. https://ubuntu.com/pro
28. https://ubuntu.com/pro/dashboard
29. https://ubuntu.com/blog/optimised-real-time-ubuntu-is-now-generally-available-on-intel-socs
30. https://www.intel.com/content/www/us/en/developer/topic-technology/edge-5g/real-time/overview.html
31. https://1.ieee802.org/tsn/
32. https://ubuntu.com/download/iot/intel-iot
33. https://ubuntu.com/engage/realtime-webinar-ga
34. https://www.intel.com/content/www/us/en/developer/articles/technical/real-time-systems-measurement-library.html
35. https://ubuntu.com/blog/real-time-kernel-tuning
36. https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/rt-tests
37. https://manpages.ubuntu.com/manpages/jammy/man1/stress-ng.1.html
38. https://git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git/
39. https://manpages.ubuntu.com/manpages/jammy/man8/oslat.8.html
40. https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start
41. https://github.com/ColinIanKing/stress-ng
42. https://wiki.ubuntu.com/StableReleaseUpdates
43. https://docs.kernel.org/scheduler/sched-deadline.html
44. https://man7.org/linux/man-pages/man7/sched.7.html
45. https://docs.kernel.org/scheduler/sched-design-CFS.html
46. https://docs.kernel.org/locking/locktypes.html
47. https://ubuntu.com/blog/what-is-real-time-linux-part-iii
48. https://www.youtube.com/watch?v=cZUzc0U1jJ4&t=14444s&ab_channel=LinuxPlumbersConference

49. https://ubuntu.com/blog/what-is-real-time-linux-ii
50. https://www.digikey.com/en/maker/projects/introduction-to-rtos-solution-to-part-11-priority-inversion/abf4b8f7cd4a4c70bece35678d178321
51. https://www.youtube.com/watch?v=pr9J0pxHI6c&ab_channel=PhilKoopman
52. https://ubuntu.com/blog/real-time-kernel-technical
53. https://www.man7.org/linux/man-pages/man1/ps.1.html
54. https://www.man7.org/linux/man-pages/man1/perf.1.html
55. https://www.man7.org/linux/man-pages/man1/irqtop.1.html
56. https://docs.kernel.org/tools/rtla/index.html
57. https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=4d66020dcef83314092f2c8c89152a8d122627e2
58. https://www.kernel.org/doc/Documentation/trace/ftrace.txt
59. https://www.kernel.org/doc/Documentation/timers/NO_HZ.txt
60. https://www.kernel.org/doc/html/latest/RCU/
61. https://github.com/torvalds/linux/blob/master/kernel/softirq.c
62. https://www.kernel.org/doc/html/v5.15/admin-guide/kernel-parameters.html
63. https://www.kernel.org/doc/html/latest/power/suspend-and-interrupts.html
64. https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/application_base
65. https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/cyclic
66. https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/memory
67. https://wiki.linuxfoundation.org/realtime/documentation/howto/applications/cpuidle
68. https://wiki.linuxfoundation.org/realtime/documentation/howto/debugging/start
69. https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/start

Canonical is the publisher of Ubuntu, the OS for most public cloud workloads as well as the emerging categories of smart gateways, self-driving cars and advanced robots. Canonical provides enterprise security, support and services to commercial users of Ubuntu. Established in 2004, Canonical is a privately held company.

Canonical