EECS 3221 Report

A Comparison of Real Time Operating Systems and the Linux Operating System

Daniel Di Giovanni — 218204818

February 14, 2024

My signature below attests that this submission is my original work:

Following professional engineering practice, I bear the burden of proof for original work. I have read the York University Senate Policy on Academic Integrity and the EECS Academic Honesty Guidelines and confirm that this work is in accordance with the Policy.

Signature	Date
	February 14, 2024

Contents

1 Introduction and Background

An operating system is a computing layer that separates the hardware of the computer from the programs that run on it. It provides the *environment* for other programs to do useful work [1, p. 4]. The fundamental tasks of an operating system include allocating resources (such as memory and CPU time), handling the control of input/output (I/O) devices, and ensuring proper usage of the computer and preventing errors [1, pp. 3-5].

The most important part of an operating system is the *kernel*. It is the first program loaded into memory on startup and is the one program that is always running on the computer [1, pp. 6-7, 22]. Along with the kernel, operating systems also include *middleware frameworks* that ease application development, and *system programs* that help the system run but are not part of the ever-running kernel. All of this supports the execution of *application programs*, which are the programs that provide functionality to the end user [1, p. 4, 7]. A diagram of system organization with the kernel is shown in Figure 1.

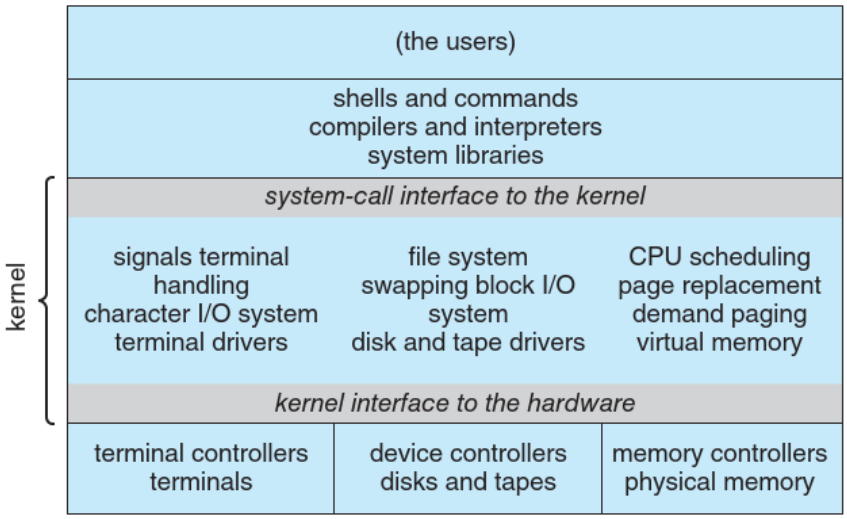

Figure 1: Diagram of a kernel within the operating system [1, p. 82].

In industrial and commercial computing applications, the choice of an operating system is crucial. It affects the performance, security, and maintainability of the system. As an example, consider the secure boot of an embedded system. Secure boot, an important security technique to ensure that the kernel code has not been modified, is often neglected in embedded systems. The absence of secure boot allows the system to boot faster with less memory and energy consumption—at the cost of leaving the boot process and internal software vulnerable. However, it was discovered that the introduction of secure boot software caused boot-up time to increase by only 4%, whereas a hardware implementation of secure boot caused a 36% increase [2, pp. 11-12]. Clearly, the operating system has a significant impact on the overall quality of the system.

Two important classes of operating systems/kernels will be discussed here: the Linux

operating system and real-time operating systems (RTOSs). The Linux kernel is a free and open source implementation of an operating system kernel. It is used ubiquitously not only for desktop computers, but also for servers and embedded devices with a broad range of commercial and industrial applications [3]. The Linux kernel is a tried-and-tested system with high flexibility and extendability. RTOSs are more vague, being defined not by a specific implementation, but by the ability to manage systems with complex time and resource constraints [4]. RTOSs need to be able to meet strict deadlines associated with external events using limited resources. In short, "a real-time system is one whose correctness involves both the logical correctness of the outputs and their timeliness" [5].

The objective of this report is to provide a thorough comparison of the Linux operating system/kernel with RTOS/real-time kernels to aid in the decision of which operating system to use.

2 Overview of Linux

When "Linux" is referred to, an entire operating system is often being referenced. However, "Linux" is just the kernel. The Linux kernel is used in combination with other software to make a complete operating system. The entire operating system (with the Linux kernel inside) is called a "Linux distribution" (for example, Ubuntu and Debian for PCs) [6]. The ability to extend and modify a Linux operating system is where its flexibility originates.

Since Linux is an open source kernel, anyone can read, use, and modify the code. And since it is just a kernel, it *requires* additional software to be useful. This leads to a wide variety of adaptations of the Linux-based operating systems to fit many different needs [7].

Many Linux distributions are desktop-focused, creating an easy user-interface, similar to Microsoft's Windows and Apple's MacOS, with the ability to run virtually all of the programs expected from a desktop computer. Linux operating systems are also a popular choice for web servers and have become the backbone of enterprise computing [8]. According to the Linux Foundation, Linux powers the majority of the public cloud [9], and some companies, like Amazon Web Services, have developed their own Linux distribution for use in their products [10].

Most pertinent to this report, however, is the use of Linux in embedded applications. While some embedded devices do not employ an operating system (these are called *bare-metal* systems, and they forgo an operating system to conserve resources [11]), most are complex enough to require an operating system. And when an embedded system requires an operating system, Linux is an appropriate choice for its kernel.

The Linux Foundation estimates that 62% of embedded systems use a Linux-based operating system [9]. This is possible because of the high degree of modularity within the Linux kernel, making it easy to configure the kernel to specific hardware [12]. Further, developers of embedded Linux distributions have the ability to exclude packages specific

to desktops, like user systems and GUI environments, opting instead for packages suited for embedded development, like cross-development tools, different types of drivers, and debugging and profiling tools [12]. This extensibility and flexibility makes embedded Linux a great choice.

3 Overview of Real-Time Operating Systems

Real-time systems are systems with rigid timing requirements [1, pp. 46]. RTOSs can be used to provide a layer of abstraction between the software handling real-time events and the underlying hardware. These operating systems must be high-performance with predictable and deterministic behavior [13].

Real-time systems can be categorized into *soft* and *hard* systems. According to Intel, hard real-time systems refer to systems where missing a deadline means failure of the system [14]. In soft real-time systems, however, the focus in on optimizing quality of service. Missing a deadline would result lower-quality service, not in a system failure [15]. As an example, a robotic arm needs to be instructed to stop *before* it hits a wall, and failure to meet this deadline will cause the arm to break. This means the robotic arm is a hard real-time system [1, pp. 45-46]. A consumer smartphone would be a soft real-time system because while a lagging phone is frustrating to the user, it is not catastrophic.

There are many strategies for implementing a RTOS. The focus is on responding to external (real-time) events in a time period that is minimal and predictable. The time taken to respond to an event is called *dispatch latency*. As shown in Figure 2, dispatch latency consists of many aspects, like interrupt processing, preempting and managing conflicts with other processes, and executing the actual process. Thus, the operating system's process scheduling algorithm is of utmost importance.

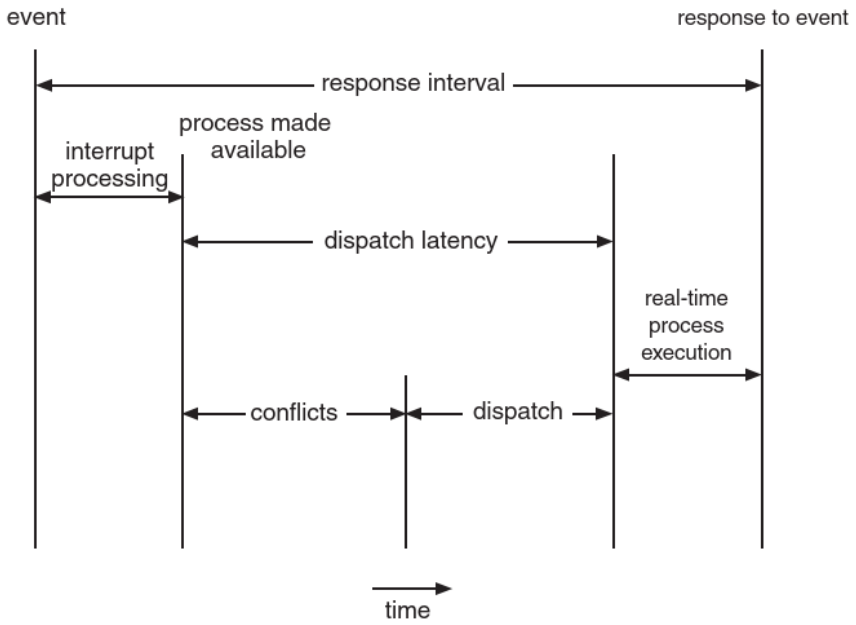

Figure 2: Dispatch latency of an event [1, p. 229].

To meet strict deadline requirements, a priority-based scheduler is necessary. Such a scheduler enables higher-priority processes to preempt lower-priority ones [1, p. 229]. For hard real-time systems, both the CPU burst time of the process and its deadline must be known. When a process is ready to execute, the scheduler can decide whether or not it is possible to meet the process's deadline, and choose to schedule it accordingly. This is called an *admission-control algorithm* [1, pp. 229-230].

Rate-monotonic scheduling can be used to schedule periodic real-time tasks. In this algorithm, processes with shorter periods are assigned higher priorities, since they have a shorter deadline. For this type of scheduling to work its CPU burst time must be the same every time it executes, and must be shorter than its deadline and period. A pitfall of rate-monotonic scheduling is that it is not always able schedule processes even if there is enough CPU time for them [1, pp. 230-232].

Another real-time scheduling technique is called *earliest-deadline-first (EDF)*. In this algorithm, higher priority is given to processes whose deadline is sooner. This way, a process near its deadline can preempt a process that has more time to spare. EDF scheduling can achieve a CPU usage near 100%, thus being able to schedule more processes than rate-monotonic scheduling [1, pp. 232-233].

Implementing an RTOSs is more complex than using a Linux-based operating system, but the complexity is the cost for performance. For hard real-time systems, an RTOS is necessary to meet performance requirements. For soft real-time systems, tradeoffs must be balanced to determine whether the improved service quality of the system is worth the added complexity and cost.

4 Comparing Linux-Based and Real-Time Operating Systems

RTOSs provide safety and predictability to the system they support. The ability to meet deadlines when responding to external events is necessary for hard real-time systems and crucial for soft ones. The price for this assurance is increased complexity, cost, and development time. Alternatively, Linux-based operating systems feature a suite of tools for handling soft real-time requirements. And, by virtue of the Linux kernel being open source, it can be extended to further meet real-time requirements by developers with expert Linux knowledge. Deciding whether to switch from a Linux distribution to a RTOS involves analyzing the benefits of a RTOS and balancing them with their associated costs. These benefits and tradeoffs between will be discussed here. But first, it must be acknowledged that there is a high degree of overlap between the two, especially regarding soft real-time systems.

4.1 The Overlap Between Linux-Based and Real-Time Operating Systems

When utilizing Linux for real-time systems, there are generally two options. The first is to use the real-time capabilities built into the Linux kernel. One such capability is a patch to the kernel that makes it fully preemptible. The second option is to use a real-time framework for the Linux kernel. Three such frameworks are discussed, all of which function by adding a *co-kernel* in a addition to the Linux kernel.

4.1.1 The Fully Preemptible Linux Kernel

The real-time features of the Linux kernel revolve around *preemption*—pausing a running process to run a higher-priority process instead. In user-mode, any process can be preempted, transferring control to the kernel for scheduling [16]. The preemption strategy for kernel mode is where the potential for real-time Linux lies. Mainline Linux (the vanilla version of the Linux kernel not associated with any specific distribution) offers three preemption strategies for the kernel:

- **PREEMPT_NONE** forbids preemption when in kernel mode; system call returns and interrupts are the only preemption points [17].
- **PREEMPT_VOLUNTARY** allows low-priority processes to voluntarily preempt themselves when executing a system call in kernel mode [18].
- **PREEMPT** makes all kernel code preemptible, except for critical sections [17].

The fully preemptible kernel (the **PREEMPT_RT** patch) takes preemption further. This patch aimed to give Linux even more real-time capabilities. Although not part of mainline Linux, this patch is backed by the Linux Foundation and is in use for real-time systems

[19]. This preemption strategy further increases the number of preemption points in kernel code and uses more real-time data structures when handling interrupts and threads [17].

The fully preemptible Linux kernel can be utilized in high-performance computing and embedded industrial environments. For complex real-time systems this kernel configuration provides real-time capabilities within the familiarity of a Linux-based operating system. The primary caveat is that there is no formal guarantee of worst case execution times, and thus cannot be used for safety-critical real-time systems [20].

4.1.2 Real-Time Linux Frameworks

Other than the fully preemptible Linux kernel, the most common approach to adapting the Linux kernel to real-time environments involves using a real-time framework that adds a co-kernel to the Linux kernel. The idea is to have the co-kernel working as a layer between the hardware and the Linux kernel. The co-kernel is responsible for catching hardware interrupts, scheduling them as either real-time tasks or Linux tasks, and guaranteeing that real-time tasks meet their deadline. Leftover CPU time is given back to the Linux kernel [20].

The most common open-source implementations of a real-time co-kernel Linux framework are RTLinux, Xenomai, and the Real Time Application Interface for Linux (RTAI).

- **RTLinux** runs the Linux kernel as a fully-preemptible process [20]. It then intercepts all hardware interrupts and schedules tasks. Shown in Figure 3, hardware interrupts not related to real-time events are passed to the Linux kernel as software interrupts, whereas real-time event interrupts are handled by the appropriate real-time interrupt service routines [21].

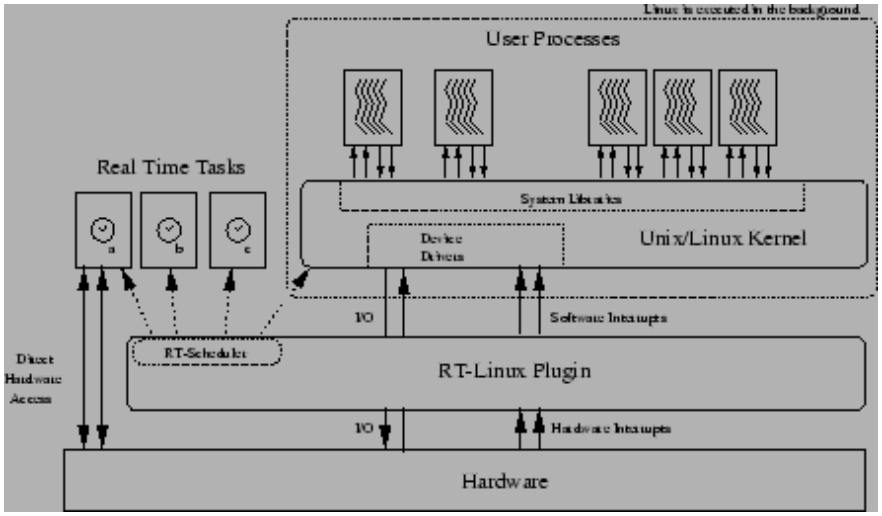

Figure 3: Structure of RTLinux [21].

- **Xenomai** works by supplementing the Linux kernel with a real-time kernel running

side-by-side with it [23]. The real-time core deals with all time-critical tasks and has a higher priority than the Linux kernel. The Xenomai kernel and the Linux kernel communicate via the *Adaptive Domain Environment for Operating System (ADEOS)*, which separate domains for both kernels [20]. This is shown in Figure 4.

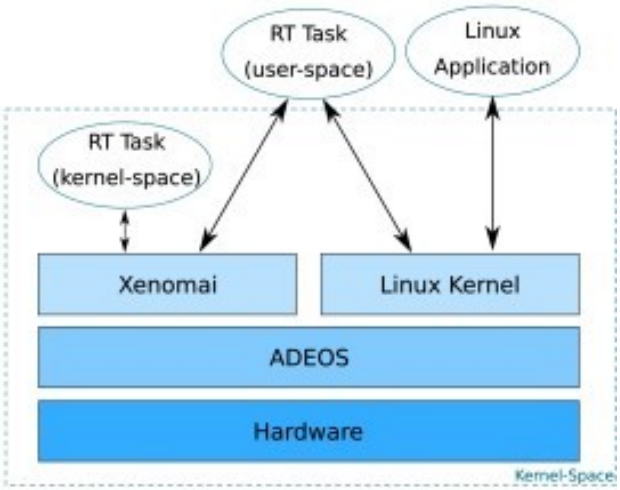

Figure 4: Structure of Xenomai [20].

- **RTAI** uses a hardware abstraction layer (RTHAL) to get information from Linux and dispatch interrupts [23]. This is displayed in Figure 5. It has few dependencies to Linux, making it easy to switch between versions of the Linux kernel [23].

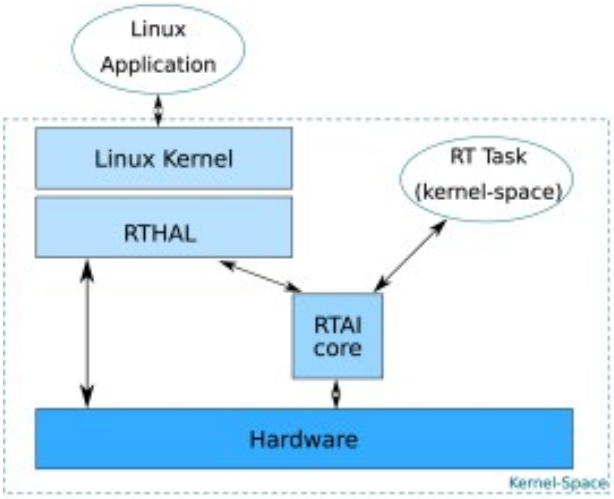

Figure 5: Structure of RTAI [20].

4.1.3 Choosing Between the Preemptible Kernel and Real-Time Frameworks

The fully preemptible Linux kernel and real-time Linux kernel frameworks both add real-time capabilities to the Linux kernel. The fully preemptible kernel achieves this by allowing

the kernel to be preempted in kernel mode and utilizing real-time data structures [17]. However, these modifications are not enough to accommodate the needs of a hard real-time system [20]. At its core, the PREEMPT_RT patch is still the Linux kernel, offering no guarantees on worst case execution time [20]. This approach should only be taken for soft real-time systems. In this use case, the most important advantage of using the fully preemptible Linux kernel is the cost and speed of development. PREEMPT_RT Linux does not differ much from mainline Linux. This allows the wealth of code for drivers and libraries in Linux to be reused—although they may need to be adapted to fit real-time needs. Further, many software engineers are competent with Linux and can handle developing with the PREEMPT_RT patch [20].

When tighter deadlines are required, a co-kernel Linux framework can be used. These frameworks do have the ability to handle hard real-time requirements [20]. The cost of this functionality is development time and complexity. Co-kernel approaches require modifications of the Linux kernel code, none of which are fully backed by the Linux community. This also introduces a dependency on the specific Linux kernel version being modified, often being an older version [20]. The complex interaction between the two kernels introduces a level of complexity that needs to be handled by skilled real-time developers, increasing development efforts [20]. As a result, real-time Linux frameworks should be used only when necessary, preferring the fully preemptible kernel when timing requirements allow it.

4.2 Non-Linux-Based Real Time Operating Systems

There are many RTOSs that are not associated with the Linux kernel. Some of these operating systems are open source, others are commercially available. They have different use cases when compared with each other, and when compared with Linux-based alternatives. These use cases will be presented here, along with examples of non-Linux RTOSs.

4.2.1 Analysis of Non-Linux-Based Real-Time Operating Systems

The main advantage of using the Linux kernel for real time systems is the ease of development. Many software engineers are familiar with Linux and can quickly learn how to use the real-time capabilities of the kernel. Soft real-time systems can make use of the fully preemptible kernel, while hard real-time systems require a framework that modifies the kernel. However, real-time Linux-based operating systems, whether it is the fully preemptible kernel or a real-time kernel framework, require the memory space and processing power to run a full Linux operating system. This is a luxury that many embedded systems simply do not have [4]. Apart from very large systems that can take support a full Linux kernel, another solution is needed.

For hard real-time embedded systems with limited resources, a non-Linux RTOS is the only option (aside from writing bare-metal code for the hardware without an operating system). Commercial RTOSs are the most popular choice for this scenario. These are

RTOSs developed and maintained by a private company. While they have a high up-front cost, and may even charge royalties for deployment of the system, they alleviate in-house development costs and provide a high level of trust [4]. Open-source RTOSs also exist, allowing access to the source code of the operating system. Although the source code is freely obtained, long-term support is often needed and must be purchased [4]. This support can come from the maintainers and user community of the operating system, or from hired software engineers. To further add to the cost, low-level development tools are often needed to work with these RTOSs and must be purchased for the engineers [4].

Choosing an RTOS is a technical decision that must take into account the system requirements and timing deadlines that the operating system must be able to handle. This information must be balanced with the cost of purchasing and maintaining the system, which will introduce a cost whether the engineers are hired or not. Ultimately, the decision is very specific to the situation being addressed and requires expertise and careful thought.

4.2.2 Examples of Non-Linux-Based Real-Time Operating Systems

Before concluding, some examples of non-Linux RTOSs will be discussed to provide a more detailed view of the real-time software ecosystem.

- **VxWorks** is a commercial RTOS that is the most popular in the embedded industry [4]. Some notable uses of VxWorks include the Mars Curiosity rover [24] and the International Space Station [4]. VxWorks supports many development hosts and can run on all popular CPU platforms.
- **Embedded Configurable Operating System (eCos)** is the most popular open source RTOS. It provides a configuration tool available in both graphical and command-line formats to specify the system's requirements [4]. The focus of eCos is on configurability. It can scale from extremely small memory-constrained systems to large systems with complex functionality [25]. There are fewer supported development hosts and CPU platforms with eCos than with VxWorks, but it still offers the expected operating system functionality and real-time capabilities [4].
- **HartOs** is a research project that is attempting to implement a RTOS at the hardware level [26]. HartOs's goal is to sidestep the computational overhead and memory footprint associated with software RTOSs while increasing the flexibility and capabilities of hardware RTOSs [26]. While not fully complete, HartOs is an innovation in the development of hardware-based RTOSs.

5 Conclusion

Choosing to migrate from a Linux distribution to a real-time operating system is a complex and technical decision. One option is to stay with Linux and utilize the preemption strategies provided by the kernel. The fully preemptible Linux kernel offers the closest capability

to real-time, but still cannot support hard real-time systems. Still staying with Linux, the kernel can be modified to handle hard real-time deadlines. The most widely-used way of implementing this is with a co-kernel approach.

Leaving the domain of Linux, there are both commercial and open source RTOSs that are not based on the Linux kernel. Choosing between these involves a careful processes of weighing the costs and efforts of development, maintenance, and system requirements. This is a very specific decision that should be guided by technical efforts to determine exactly what the real-time requirements are, whether or not they can be handled by a soft real-time system, and a balancing of the associated costs.

References

[1] A. Silberschatz, P. B. Galvin, G. Gagne, *Operating System Concepts*, 10th ed., John Wiley and Sons, Inc., 2018.

[2] J. Ingelhag, “How to choose an operating system for an embedded system”, Örebro Universitet, 2023. Accessed February 9, 2024. [Online]. Available: https://www.diva-portal.org/smash/get/diva2:1773441/FULLTEXT01.pdf.

[3] The Linux Foundation, “What is Linux?,” *The Linux Foundation*. Accessed February 7, 2024. [Online]. Available: https://www.linux.com/what-is-linux/.

[4] W. Cedeño, P.A. Laplante. “An Overview of Real-Time Operating Systems,” JALA: Journal of the Association for Laboratory Automation, 2007, ch. 12, sec. 1, pp. 40-45. Accessed February 7, 2024. [Online]. Available: https://doi.org/10.1016/j.jala.2006.10.016.

[5] P. A. Laplante, “Real-Time Systems Design and Analysis,” 3rd ed., Hoboken, NJ, Wiley, 2004, p. 505. Accessed February 7, 2024. [Online]. Available: https://doi.org/10.1002/0471648299.fmatter.

[6] R. Stallman, “Linux and the GNU System,” *Free Software Foundation*. Accessed February 9, 2024. [Online]. Available: https://www.gnu.org/gnu/linux-and-gnu.en.html.

[7] A. Adekotujo, A. Odumabo, A. Adedokun, O. Aiyeniko, “A Comparative Study of Operating Systems: Case of Windows, UNIX, Linux, Mac, Android and iOS,” International Journal of Computer Applications, 2020. Accessed February 10, 2024. [Online]. Available: https://www.researchgate.net/profile/Adedoyin-Odumabo/publication/343013056_A_Comparative_Study_of_Operating_Systems_Case_of_Windows_UNIX_Linux_Mac_Android_and_iOS/links/61f2b50a9a753545e2fe8300/A-Comparative-Study-of-Operating-Systems-Case-of-Windows-UNIX-Linux-Mac-Android-and-iOS.pdf.

[8] Grand View Research, “Server Operating System Market Size, Share & Trends Analysis Report By Operating System (Windows, Linux), By Virtualization (Virtual Machine, Physical), By Deployment, By Region, And Segment Forecasts, 2022 - 2030,” *Grand View Research*, 2020. Accessed February 10, 2024. [Online]. Available: https://www.grandviewresearch.com/industry-analysis/server-operating-system-market-report.

[9] The Linux Foundation, “Linux Runs All of the World’s Fastest Supercomputers,” *The Linux Foundation*, November 20, 2017. Accessed February 10, 2024. [Online]. Available: https://www.linuxfoundation.org/blog/blog/linux-runs-all-of-the-worlds-fastest-supercomputers.

[10] L. Clark, “How Amazon Web Services Uses Linux and Open Source,” *The Linux*

Foundation, September 8, 2014. Accessed February 10, 2024. [Online]. Available: https://www.linux.com/news/how-amazon-web-services-uses-linux-and-open-source/.

[11] M. Salehi, D. Hughes, B. Crispo, "MicroGuard: Securing Bare-Metal Microcontrollers against Code-Reuse Attacks," *2019 IEEE Conference on Dependable and Secure Computing (DSC)*, Hangzhou, China, IEEE, 2019, pp. 1-8, doi: 10.1109/DSC47296.2019.8937667. Accessed February 10, 2024. [Online]. Available: https://doi.org/10.1109/DSC47296.2019.8937667.

[12] P. Raghavan, A. Lad, S. Neelakandan, *Embedded Linux System Design and Development*, Boca Raton, FL, Taylor and Francis Group, LLC, 2006.

[13] A. S. Gillis, "DEFINITION real-time operating system (RTOS)," TechTarget Accessed February 11, 2024. [Online]. Available: https://www.techtarget.com/searchdatacenter/definition/real-time-operating-system

[14] Intel, "Real-Time Systems Overview," Intel. Accessed February 11, 2024. [Online]. Available: https://www.intel.com/content/www/us/en/robotics/real-time-systems.html

[15] G. Lipari, L. Palopoli, "Real-Time scheduling: from hard to soft real-time systems," arXiv, 2015. Accessed February 11, 2024. [Online]. Available: https://doi.org/10.48550/arXiv.1512.01978

[16] E. Barbieri, "What is real-time Linux? Part II," *Ubuntu*, February 28, 2023. Accessed February 13, 2024. [Online]. Available: https://ubuntu.com/blog/what-is-real-time-linux-ii.

[17] The Linux Foundation, "Preemption Models," *The Linux Foundation*, October 3, 2023. Accessed February 13, 2024. [Online]. Available: https://wiki.linuxfoundation.org/realtime/documentation/technical_basics/preemption_models.

[18] E. Barbieri, "What is real-time Linux? Part III," *Ubuntu*, March 7, 2023. Accessed February 13, 2024. [Online]. Available: https://ubuntu.com/blog/what-is-real-time-linux-part-iii.

[19] J. Markus, "Evaluation of Real-Time Linux on RISC-V processor architecture," Tampere University, March 2022. Accessed February 14, 2024. [Online]. Available: https://trepo.tuni.fi/handle/10024/138547

[20] F. Reghenzani, G. Massari, W. Fornaciari, "The Real-Time Linux Kernel: A Survey on PREEMPT_RT," New York, NY, USA, Association for Computing Machinery, January 2020. Accessed February 14, 2024. [Online]. Available: https://doi.org/10.1145/3297714

[21] Michael Barabanov, "Getting Started with RTLinux," *FSM Labs, Inc.*, July 26, 2001. Accessed February 14, 2024. [Online]. Available: http://cs.uccs.edu/ cchow/pub/rtl/-

doc/html/GettingStarted/.

[22] Xenomai, “Xenomai 3 Overview,” *Xenomai*. Accessed February 14, 2024. [Online]. Available: https://v3.xenomai.org/overview/.

[23] RTAI, “RTAI Beginner’s guide,” *RTAI*. Accessed February 14, 2024. [Online]. Available: https://www.rtai.org/Beginner’s-guide.html.

[24] VxWorks, “VxWorks Powers NASA JPL’s Mars Rover Curiosity,” *VxWorks*. Accessed February 14, 2024. [Online]. Available: https://www.windriver.com/success-stories/nasa-jpl.

[25] eCos, “eCos Overview,” *eCos*. Accessed February 14, 2024. [Online]. Available: https://www.ecoscentric.com/ecos/index.shtml.

[26] A. B. Lange, K. H. Andersen, U. P. Schultz, A. S. Sørensen, “HartOS - a Hardware Implemented RTOS for Hard Real-time Applications,” University of Southern Denmark, 2012. Accessed February 14, 2024. [Online]. Available: https://doi.org/10.3182/20120523-3-CZ-3015.00041