A Comparison of Real-Time Operating Systems and Their Schedulers

Hannah Ashton
School of Engineering and Computer Science
Syracuse University
Syracuse, New York
hcashton@syr.edu

***Abstract*—This paper explores the real-time operating system, starting with the distinction of a real-time versus a general-purpose operating system. We then explore the key characteristics and functionalities of real-time operating systems: their presence in embedded systems, their architectures, their timing classifications, and their essentials and variations. The latter half of this paper explores one of these essentials: the operating system's scheduler. Background information about schedulers is discussed, distinguishing between static and dynamic scheduling, preemptive and non-preemptive scheduling, and acknowledging the cost of task switching. The paper then takes a deeper look at the schedulers of several of the previously introduced real-time operating systems, and concludes with a proposal for a new real-time operating system scheduler.**

Keywords—real-time, operating system, architecture, kernel, scheduler, process, embedded system, preemptive, priority-based

I. Introduction to Real-Time Operating Systems

Operating systems exist in many different realms and applications and can be categorized as either general-purpose or real-time. General-purpose (or non-real-time) operating systems are used in personal computing, but real-time operating systems come into play when the response time of the operating system must be extremely fast [1]. A deadline of a fraction of a second must be adhered to in each and every response to some external events, typically from the environment [2].

As would be expected from their name similarities, real-time operating systems are typically used to run real-time applications where the timing is also critical. They handle the scheduling and need to be predictable. In order to maximize the speed of the response, real-time operating systems are typically compact and only focus on executing a few tasks very well. This prioritization makes them ideal candidates for use in embedded systems [2]. For example, a self-driving vehicle must have a compact yet extremely efficient and reliable operating system to react to environmental triggers safely.

A. Real-Time and General-Purpose Operating Systems

A general-purpose operating system, as stated previously, is known for its application in personal computing. These systems must be able to run a wide array of applications and be able to support running any number of these at any given time [3]. They are complex systems and can be thought of as a "jack of all trades" for their diverse abilities.

The real-time operating system, if continuing with the previous analogy, is the "master of one". They are designed to be simpler and more nimble; real-time operating system typically "do not support disk, network, keyboard, monitor, or mouse by default" [4]. Their goal is to conduct a few number of procedures and to do so extremely well.

Another key difference between a real-time operating system and a general-purpose operating system is in the access to the kernel. The kernel is responsible for handling system calls, including input/output access, memory access, and more. General-purpose operating systems typically follow a layered architecture in which the kernel can only be accessed by specific layers. In a real-time operating system, the kernel can be accessed by applications, middleware, and device drivers [4]. Figure 1 visualizes the connection between the application and the kernel in a real-time operating system, as opposed to the separated user applications in the general-purpose model.

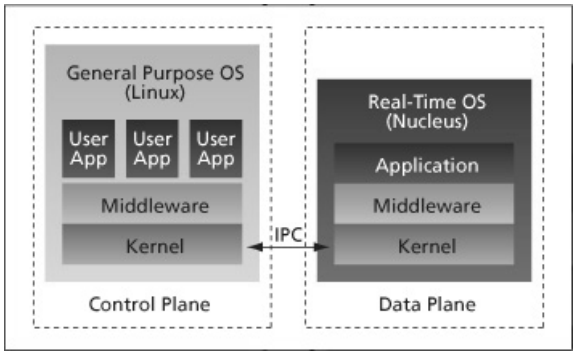

Fig. 1. A comparison of General Purpose OS and Real-Time OS [5].

B. Embedded Systems

Because of their intertwined nature with real-time operating systems, it's worth discussing the embedded system. An embedded system operates within some larger system and although they often rely on a real-time operating system, they do not always. Embedded systems sometimes do not even utilize an operating system, such as the Arduino system. In fact, this used to be the case more often in older microprocessors [6]. However, in order for a system to run multiple applications to run at once, an operating system is required to allocate memory and handle scheduling, filing, and so on.

II. Variations of Real-Time Operating Systems

A. Essential Components

Now that the purpose of a real-time operating system is clear, let's discuss what essential characteristics are necessary to achieve this. Because of the requirement to handle stimuli within a specific deadline, "the fundamental service provided by an RTOS is task management" [7]. Task management ensures that the real-time operating system executes the highest priority processes first, seemingly instantaneous from the process being created.

The characteristics that a real-time operating system must provide include determinism, high-performance, safety and

security, priority-based scheduling, and small size [3]. Determinism refers to the consistency of results: given the same input or stimuli, the same output or response must result. The speed of these results is captured by the high-performance characteristic. Safety and security is essential because many of these embedded systems are in safety- or mission-critical environments. Priority-based scheduling, as we will dive deeper into later, is responsible for the task management and handling the most urgent processes first. Finally, a small size is a mark of a real-time operating system; it is estimated that some real-time operating system are up to 20,000 times smaller than their general-purpose operating system counterparts [7].

B. Architectures

The real-time operating system typically utilizes either a microkernel or a monolithic kernel [3]. In a microkernel, the operations and the kernel each have their own dedicated space in memory; in a monolithic kernel, everything is done in the kernel. Each of these architectures has advantages and disadvantages.

Because user services are run in a separate address space in a microkernel, the kernel is not only smaller and more compact, but it is also more protected and secure [8]. If an error occurs with a user process, the kernel is unimpacted. However, because the address spaces are separate, a microkernel requires more time and CPU energy to switch between the kernel and user processes [3].

In a monolithic kernel, the kernel space is larger and is susceptible to system failures, since every process must exist in the same address space [8]. These kernels are also much more challenging to update, because the address space is so intertwined and "making a change in one area could have ramifications for the entire system" [3]. A benefit to the monolithic kernel, though, is its speed; messages do not have to be passed across different address spaces, thus increasing efficiency [8].

C. Timing Classifications

Another variation in real-time operating system is the designation of being hard, firm, or soft. Each of these classifications indicates how strictly the timing deadlines must be adhered to. In a hard real-time operating system, if the deadline is not met, the entire system fails and the consequences can be catastrophic. A firm real-time operating system can tolerate an occasional missed deadline with degraded performance, but if deadlines are missed regularly, the system fails. A soft real-time operating system is able to continue operating through missed deadlines, though performance may decline if deadlines are repeatedly missed [9].

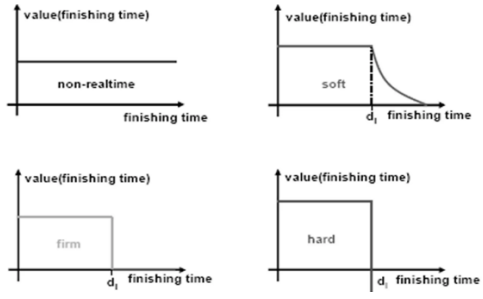

Fig. 2. Value in relation to finishing time, various time classifications [10].

Figure 2 illustrates these classifications (as well as a non-real-time system) as a correlation between finishing time and value. While "soft" refers to more flexibility in the operating system response time than "hard", it is worth noting that even soft real-time operating system are still extremely responsive. In fact:

> Soft real-time systems operate within a few hundred milliseconds, at the scale of a human reaction. Hard real-time systems, however, provide responses that are predictable within tens of milliseconds or less [3].

Clearly, timing is a crucial aspect of the real-time operating system. A similar classification is used by the real-time operating system for tasks. Tasks labeled as critical must be completed within the deadline, dependent tasks are best if completed within the deadline, and flexible tasks can be completed at any time [4].

III. Examples of Real-Time Operating Systems

Because of their popular presence in embedded systems, real-time operating systems have a wide range of applications. Figure 3 lists just a few examples of these industries and use-cases. Different applications necessitate different options of real-time operating systems, some of which we will discuss here.

A&D	Telecom	Transportation	Medical	Manufacturing
· Flight display controller · Engine turbine · Drones · Extraterrestrial rovers	· 5G modem · Satellite modem · Base station	· Functional safety systems · Emergency braking systems · Engine warning systems	· Magnetic resonance imaging · Surgery equipment · Ventilators	· Factory robotics systems · Safety systems · Oil and gas vibration monitors

Fig. 3. Applications of real-time operating systems [3].

First, some industries or users may opt for an in-house development of their own real-time operating system, which allows the system to be crafted to perfectly match the demands and flexibilities of the product or system they will be a part of [3]. A drawback to this approach is the time and money required to develop an entire operating system. The company may not have operating system experts on staff, so the research, development, and testing may stretch out beyond the scope of the project.

A. VxWorks

Perhaps the most visible real-time operating system is VxWorks, created in the 1980s [12]. Applications for this C++-based system can be designed with two compatible IDEs, both developed by Wind River Systems, the creators of VxWorks itself [9]. This operating system is similar to Unix and can function with either a single core or multicore, although it only offers a fixed priority scheduling system [13]. Though system does include security features, these features have been called into question [9].

B. Linux Adapted / Embedded Linux

Modified versions of Linux are also commonly used for real-time operating system, piggy-backing off of the general-purpose operating systems benefits. Linux allows for dynamic priorities, a feature that promotes time sharing, potentially too much. Instead, a solution is proposed:

Considering also that swapping can be disabled and that, for given level ranges, priorities can be made fixed, we can state that Linux can currently be considered a soft real-time operating system and can therefore be used for many applications which tolerate occasional delays in system response [13].

All that is required to consider this a real-time operating system is that interrupts are disabled when real-time performance is necessary. There are several versions and extension of Linux that serve this purpose, including Embedded Linux [9], RTAI, and Xenomai [13].

C. QNX

Another example of a real-time operating system is QNX. Similar to VxWorks, this system is similar to Unix, but in contrast, it is offered in more programming languages and in multiple architectures. The microkernel operating system comes at a financial cost, thus its usage in high level systems such as the Jaguar Land Rover [9].

IV. Real-Time Operating System Schedulers

Now that we have reviewed the general characteristics, applications, and variations in real-time operating systems, we will dive into one of the most crucial aspects of these systems: the scheduler. As mentioned before, the scheduler is responsible for determining which process to run when, ultimately ensuring that the critical tasks are executed within their deadlines. When multiple processes need to be run at once, simultaneity is achieved by the scheduler "rapidly switching between programs" [14].

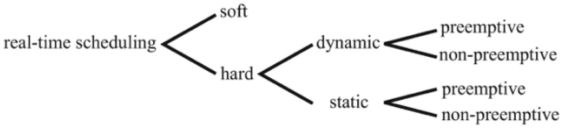

Fig. 4. Taxonomy of real-time operating systems [15].

A. Static and Dynamic Schedulers

The scheduling for a system can either be determined statically or dynamically. Static scheduling is done prior to run-time based on the initially-known characteristics of the tasks. Dynamic scheduling is determined at run-time based on the evolving information known about the tasks [15]. An example of a dynamic scheduling policy would be the Earliest-Deadline-First Algorithm, which updates real-time based on the existing time and deadlines in the system [15].

B. Preemptive and Nonpreemptive Schedulers

Schedulers in general can be divided into preemptive and non-preemptive. Preemptive scheduling means that the current process will be interrupted and replaced if a higher priority process becomes ready. Non-preemptive scheduling, on the other hand, allows each process to run as long as it needs or wants, until self-blocked as in the case of waiting for input or output, sleeping, or completing. This is usually best utilized when all processes are relatively quick to run completely or interrupt themselves frequently [15].

Without preemption, the current process cannot be interrupted by another process. Imagine a case where a low priority process begins execution and does not block for any reason. If this process runs for an extended period, it would be detrimental to a higher priority process that enters the ready queue during this time. The higher priority process will not have a chance to execute until after the current process terminates, which could be after the deadline for the high-priority process has passed.

Real-time operating systems, especially hard real-time operating systems, cannot tolerate high priority processes missing their deadlines. To avoid the above described scenario, real time operating systems all use at least some version of preemptive scheduling. We will discuss later a few of the versions adapted in common real-time operating systems.

Scheduling Methods	Advantages	Disadvantages
Preemptive Scheduling	① the outstanding character of real-time; the quick reaction; ② simple scheduling algorithm can ensure the time constraint of high-priority tasks	the context switches are too many
Non-preemptive Scheduling	the context switches are less	① The low utilization rate of processor's effective resources; ② poor schedulability
Static Table-driven Strategy	specified in the offline case	the weak function of scheduler and only the function of dispatcher left

Fig. 5. Advantages and disadvantages of various scheduler types [11].

C. Task Switching

An important consideration with scheduling is the cost required to switch between tasks. Each time the operating system changes the current task, a context switch is required, but context switches are expensive in terms of the CPU's time and energy [2]. At the same time, we have already discussed the importance of running the right process at the right time in order to stay within the process deadlines. These completing priorities call for a delicate balance between minimizing task switching and ensuring the highest priority process is running. For this reason, round robin scheduling is typically used minimally in real-time operating system [7].

V. Examples of RTOS Schedulers

Now, we return to the real-time operating systems that we discussed previously, as well as a few additional operating systems, in order to explore their schedulers and the variation among them.

A. VxWorks

In the VxWorks scheduler, there are 255 priorities with 0 being the highest [12]. This real-time operating system uses preemption, so if a higher priority process enters the ready queue, the current process will interrupt and allow the new, higher-priority process to run. If two processes have the same priority level in the ready queue, VxWorks offers either a First-Come-First-Serve policy or a Round Robin policy [12].

First-Come-First-Serve, as it implies, runs whichever process was added to the queue first. Once that process finishes or interrupts, then the other process (that had the same priority level but entered the ready queue second) will run.

In a Round-Robin policy, processes of equal priority will alternate running, each given a specified amount of time to run. The Round-Robin policy is best for sharing resources

between the processes, but would require more context switches, using valuable CPU resources [12].

B. Linux Adapted or Embedded Linux

Linux typically runs with either a normal, First-Come-First-Serve, or Round-Robin scheduling policy. When adapted for real-time applications, Linux utilizes the Round-Robin policy to ensure high priority processes complete within their deadline, and a preemptive approach is almost always used [16]. A visual comparison of these two scheduling scenarios is outlined below; in the real-time version, Thread1 is permitted to run to completion before Thread2 is given any time, since Thread2 is of lower priority than Thread1.

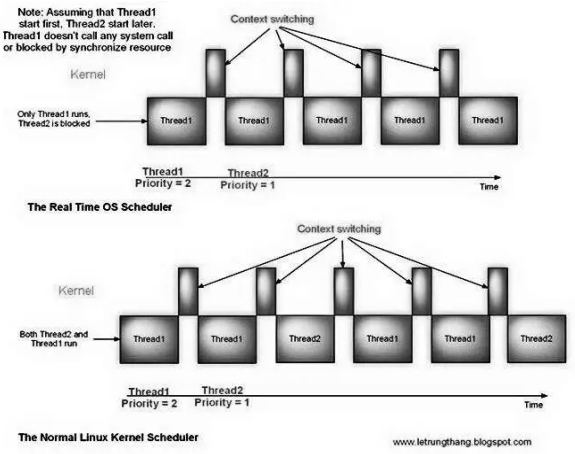

Fig. 6. Real Time OS vs Normal Linux Scheduler [16].

C. LynxOS

A real-time operating system that we did not yet discuss is the LynxOS. Similar to an adapted real-time Linux, LynxOS offers First-In-First-Out and Round Robin policies, but LynxOS has an additional option referred to as "Quantum". The Quantum policy mirrors Round-Robin in that each process is given a certain amount of time to run before switching to another equal-priority process, but in Quantum, "the length of the time-slice is not fixed; it is a variable for each priority level" [12]. For instance, tasks of higher priority may be given a shorter quantum to ensure fairness and that all can execute by their deadline, but lower priority processes may have shorter quantums to minimize context switching.

Another point worth noting about the LynxOS is that the scheduler handles both kernel and user tasks. Any interrupts are at the highest priority level, so that they take precedence over all other processes and can be handled quickly [12]. In this system, there are 512 possible priority levels; the even number priority levels are only for user processes, and the odd number priority levels are only for kernel processes. This set up allows there to always be a kernel priority level higher than any given user priority but lower than a higher priority user process [12].

LynxOS also utilizes semaphores with an approach called priority inheritance. Priority inheritance means that whichever process is holding the semaphore will inherit the priority level of the highest priority waiting process [12]. This ensures that when the current process signals the semaphore, it signals the process with the next highest priority. Without priority inheritance, the result is priority inversion where a low priority holding the semaphore might signal a slightly higher priority process, but not signal the highest priority processes that has been waiting [12].

D. QNX

Last but certainly not least, we will discuss the QNX scheduler. The QNX scheduler uses threads instead of processes [17], and allows each thread to adjust their own priority level from whatever they may have inherited from their parent thread [18]. First-In-First-Out, Round Robin, and Sporadic (which we shall explain further below) scheduling policies are all options, though QNX applies these scheduling techniques in a unique manner:

> Each thread in the system may run using any method. Scheduling methods are effective on a per-thread basis, not on a global basis for all threads and processes on a node [18].

The sporadic scheduling policy operates similar to a round robin, but the priority levels are dynamic. Whichever process has the highest priority first will run first, but after a certain amount of time, it is considered to have "exhausted its budget" and its priority level decreases [19]. Eventually, the priority level of this process will increase again to allow it the chance to run once again, but the even temporary drop in priority level promotes fairness. A lower priority process like L in Figure 7 below would normally have to wait until the priority N processes completed its execution, but now L only has to wait until N has exhausted its budget [19]. In other words, sporadic scheduling "ensures that a thread running at priority N will consume only C/T of the system's resources if there are other ready threads with a priority higher than L" [19].

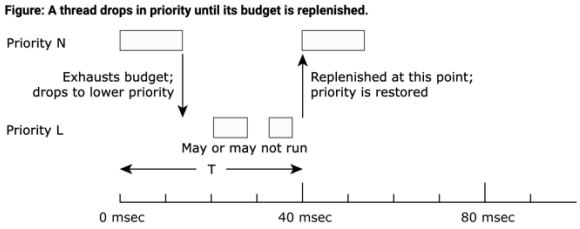

Fig. 7. QNX sporadic scheduling policy [19].

In the QNX operating system, the higher priorities are represented by higher numbers [17]. Similar to LynxOS, it offers priority inversion, as well as a number of other benefits such as CPU time partitions, "critical thread priorities, critical budgets in adaptive partitions, sporadic scheduling, and processor affinity—or even how the number of CPU cores on our board might affect our design" [17].

VI. Author's Perspective

There is no debate that the real-time operating system is a crucial piece of software with vast applications. From smart refrigerators to self-driving cars, the real-time operating system thrives in time-sensitive environments where only a small number of applications or processes may ever exist. Because the world of smart appliances is only continuing to expand, I believe embedded systems and thus real-time operating systems will also continue to develop and push boundaries.

As discussed, there are already soft, firm, and hard real-time operating systems, though these classifications are somewhat subjective. I anticipate that these lines will become increasingly blurry, with the soft-to-hard classifications becoming more of a spectrum. I also anticipate that the line between general-purpose and real-time operating systems will blur; the speed of general purpose systems may become more predictable and efficient, and the number of applications that can be run on a real time system will increase.

The scheduler is arguably the most crucial aspect of a real-time operating system because it determines when to switch the current process and what to switch it to. As seen with the previous examples, it is common for a real-time operating system to offer a few scheduling policies, including First-Come-First-Serve, Round Robin, and potentially a third variation. I think having multiple options offered by the operating system increases its potential. A potential customer can choose the scheduling policy that matches their application and needs most appropriately, and may even use the same operating system in a variety of applications.

A. Author's Suggestion and Implementation

Because of the crucial nature of the scheduler in a real-time operating system, and the variations that already exist with scheduling policies, my proposal for a new or existing system would be a new scheduling policy. This scheduler would be a hybrid of the LynxOS Quantum and the QNX Sporadic scheduling policies with dynamic priorities.

Essentially, each process will have a starting priority, and minimum priority, and a time period. The starting priority is what the process will initially be set to. While the process is in the ready queue, there is no change to its priority. Once the process is the highest priority in the ready queue, it will begin executing. This ensures that high priority processes are given the chance to execute.

However, in order to promote fairness, after the process has been running for the amount of time specified in its time period, its priority will be decremented by one level. Now that its priority has decreased, the scheduler will check to see if it should remain as the current process. If it does remain as the current process, it will remain running, but after the next time period passes, the priority will decrease again and the scheduler will reassess again.

When the current process's priority drops low enough that another process has a higher priority, it will be preempted or replaced as the current process. The original process will go into the ready queue with its current priority level and will have a chance to be resumed once the new current process has its time period exhausted.

One key characteristic of this proposed implementation is the minimum priority for each process. Once the current process reaches its minimum priority level, it will no longer decrease, even after each time period passes. This ensures that a very low priority process will not take over for a process that has a higher starting priority but has been decremented repeatedly, so long as its minimum priority is higher than the lower priority process's starting priority.

The minimum priority also allows for a case where a process must continue executing until it has completed, as long as no higher priority process enters the queue. The minimum priority could be set to be equal to the starting priority, meaning that it will never decrement and thus will not be re-added to the ready queue. Another less drastic way to help crucial processes execute would be to give them a very long time period, so that it is rare (but still possible) for their priority to decrease.

```
struct process {
    int procID
    int startPrio; // starting or maximum priority
    int minPrio; // minimum priority
    int prioTime; // priority time period at which it decrements
};

// Global variables
currProcTime = 0; // time the current process has been executing
currPrio = 0; // the current priority of the process executing

// called at each clock tick to check if preemption is needed
int schedule(int procID) {
    // check if needs to decrement priority
    if (currProcTime >= prioTime) && (currPrio >= minPrio) {
          currProcTime = 0; // reset time
          currPrio--; // decrement priority
          resched(); // check if preemption is needed
    } else { // still allowed to run at current priority
          currProcTime++;
    }
    return 0;
}
```

Fig. 8. Psuedocode for the author's proposed scheduling policy.

REFERENCES

[1] E. Hash, "Difference between the real time operating system and non-real time operating system in Embedded…," *Medium*, Apr. 24, 2024. https://medium.com/@embeddedhash.in/difference-between-the-real-time-operating-system-and-non-real-time-operating-system-in-embedded-705fedb7675f (accessed Jun. 02, 2024).

[2] T. I. Academy, "What is RTOS — Real-Time Operating System in Embedded System," *Medium*, Sep. 28, 2023. https://theiotacademy.medium.com/what-is-rtos-real-time-operating-system-in-embedded-system-1f1b9acb4040 (accessed Jun. 02, 2024).

[3] "Intro to Real-Time Operating Systems (RTOS)," Wind River. https://www.windriver.com/solutions/learning/rtos

[4] B. I. Morshed, "Software Design," Springer eBooks, pp. 101–166, Jan. 2021, doi: https://doi.org/10.1007/978-3-030-66808-2_4.

[5] "Linux Nucleus... Or both - Tech Design Forum Techniques," *www.techdesignforums.com*. https://www.techdesignforums.com/practice/technique/linux-nucleus-or-both/

[6] "What is an Embedded System? Definition and FAQs | HEAVY.AI," www.heavy.ai. https://www.heavy.ai/technical-glossary/embedded-systems

[7] I. Ungurean, "Timing Comparison of the Real-Time Operating Systems for Small Microcontrollers," Symmetry, vol. 12, no. 4, p. 592, Apr. 2020, doi: https://doi.org/10.3390/sym12040592.

[8] "Difference between Microkernel and Monolithic Kernel - javatpoint," www.javatpoint.com. https://www.javatpoint.com/microkernel-vs-monolithic-kernel#:~:text=The%20microkernel%20runs%20user%20and

[9] C. Feldbacher, "Top 4 Embedded Operating Systems of 2020 with Examples," blog.felgo.com, Jan. 06, 2022. https://blog.felgo.com/embedded/embedded-operating-systems

[10] “presentation on real time operating system(RTOS’s),” *SlideShare*, Feb. 13, 2014. https://www.slideshare.net/slideshow/ppt-on-rtos/31155721 (accessed Jun. 02, 2024).

[11] “EZproxy | Syracuse University Libraries,” *login.libezproxy2.syr.edu*. https://ieeexplore-ieee-org.libezproxy2.syr.edu/document/5376693 (accessed Jun. 02, 2024).

[12] H. Carlgren and R. Ferej, “Comparison of CPU scheduling in VxWorks and LynxOS.” Accessed: Jun. 02, 2024. [Online]. Available: https://class.ece.iastate.edu/cpre584/ref/embedded_OS/vxworks_vs_lynxOS.pdf

[13] “EZproxy | Syracuse University Libraries,” login.libezproxy2.syr.edu. https://ieeexplore-ieee-org.libezproxy2.syr.edu/document/4448543 (accessed Jun. 02, 2024).

[14] “FreeRTOS vs Linux for Embedded Systems,” www.bytesnap.com, Jan. 30, 2023. https://www.bytesnap.com/news-blog/freertos-vs-linux-embedded-systems/# (accessed Jun. 02, 2024).

[15] H. Kopetz and W. Steiner, “Real-Time Scheduling,” Springer eBooks, pp. 247–267, Jan. 2022, doi: https://doi.org/10.1007/978-3-031-11992-7_10.

[16] E. Staff, “Comparing real-time scheduling on the Linux kernel and an RTOS,” Embedded.com, Apr. 25, 2012. https://www.embedded.com/comparing-real-time-scheduling-on-the-linux-kernel-and-an-rtos/

[17] devblog.blackberry.com, “Thread Scheduling and Time Partitioning in a QNX Neutrino RTOS System,” devblog.blackberry.com. https://devblog.blackberry.com/en/2021/02/thread-scheduling-and-time-partitioning-in-a-qnx-neutrino-rtos-system

[18] Qnx.com, 2024. https://www.qnx.com/developers/docs/7.1/#com.qnx.doc.neutrino.prog/topic/overview_SCHEDS.html (accessed Jun. 02, 2024).

[19] “Sporadic scheduling,” www.qnx.com. https://www.qnx.com/developers/docs/8.0/com.qnx.doc.neutrino.sys_arch/topic/kernel_Sporadic_scheduling.html (accessed Jun. 02, 2024).