Tampere University

Oskari Heinonen

CONCURRENCY CONTROL TECHNIQUES AND THEIR PERFORMANCE IN REAL-TIME OPERATING SYSTEMS

Bachelor's thesis
Faculty of Information Technology and Communication Sciences
Examiner: Joonas Multanen
September 2025

ABSTRACT

Oskari Heinonen: Concurrency control techniques and their performance in real-time operating systems
Bachelor's thesis
Tampere University
Bachelor's Programme in Computing and Electrical Engineering
September 2025

Real-time operating systems (RTOS) are often used in embedded systems projects where deterministic behavior and precise timing are important. This thesis explores how different RTOS implementations manage concurrency, analyzes the performance overhead introduced by concurrency primitives, and compares some common RTOSes in terms of their implementation and performance. A literature review was conducted, primarily focusing on peer-reviewed research and technical documentation. Concurrency mechanisms of primarily FreeRTOS, uC/OS-II/uC/OS-III, and Zephyr were analyzed.

The findings indicate that no single RTOS excels in all performance aspects; rather, each has trade-offs depending on the use case. For example, the popular FreeRTOS suffers from significant jitter and limited priority inheritance support. In contrast, uC/OS-II and uC/OS-III demonstrate more consistent real-time behavior with lower jitter, particularly when using semaphores for synchronization. Studies also show that the choice of concurrency primitives, such as mutexes versus semaphores, significantly impacts system responsiveness.

One challenge in comparing RTOS performance is the diversity of benchmarking methodologies as well as hardware and software configurations across different studies. The results suggest that a standardized benchmarking framework could improve comparability and help system designers make informed decisions when selecting an RTOS. Further research with controlled experiments on modern embedded hardware is recommended to validate and expand upon these findings.

Keywords: Embedded systems, RTOS, real-time operating systems, concurrency, performance

TIIVISTELMÄ

Oskari Heinonen: Rinnakkaisuuden hallinnan tekniikat ja niiden suorituskyky reaaliaikakäyttöjärjestelmissä
Kandidaatin tutkielma
Tampereen yliopisto
Tieto- ja sähkötekniikan kandidaattiohjelma
Syyskuu 2025

Reaaliaikakäyttöjärjestelmiä (RTOS) hyödynnetään usein sulautetuissa järjestelmissä, joissa deterministinen toiminta ja tarkka ajastus ovat tärkeitä vaatimuksia. Tämä opinnäytetyö tarkastelee, kuinka eri RTOS-toteutukset hallitsevat rinnakkaisuutta, analysoi rinnakkaisuusmekanismien aiheuttamaa lisäkuormaa suorituksessa sekä vertailee yleisesti käytettyjä RTOS-järjestelmiä niiden toteutuksen ja suorituskyvyn näkökulmasta. Työssä toteutettiin kirjallisuuskatsaus, joka perustui ensisijaisesti vertaisarvioituun tutkimukseen ja tekniseen dokumentaatioon. Rinnakkaisuuden hallintamekanismeja analysoitiin erityisesti FreeRTOS-, uC/OS-II/uC/OS-III- ja Zephyr-käyttöjärjestelmien osalta.

Tulosten perusteella mikään RTOS ei ole paras kaikilla suorituskyvyn osa-alueilla, vaan jokaisella on omat vahvuutensa ja rajoituksensa käyttökohteesta riippuen. Suosittu FreeRTOS soveltuu tiettyihin tarkoituksiin, mutta sen heikkouksiin kuuluvat merkittävä ajoitusvaihtelu sekä rajoitettu prioriteettiperinnän tuki, mikä voi tehdä siitä sopimattoman erittäin tarkkaa ajoitusta vaativiin järjestelmiin. Sen sijaan uC/OS-II ja uC/OS-III tarjoavat johdonmukaisempaa reaaliaikaista käyttäytymistä ja pienempää vaihtelua ajoituksessa, erityisesti semaforeja käytettäessä synkronointiin. Tutkimuksissa havaittiin myös, että rinnakkaisuusmekanismin, kuten muteksin tai semaforin, valinta vaikuttaa merkittävästi järjestelmän reagointikykyyn.

Yksi keskeinen haaste RTOS-järjestelmien suorituskyvyn vertailussa on erilaisten mittausmenetelmien sekä vaihtelevien laitteisto- ja ohjelmistoalustojen käyttö eri tutkimuksissa. Tulokset viittaavat siihen, että standardoidun vertailukehyksen käyttöönotto voisi parantaa tulosten vertailtavuutta ja tukea järjestelmäsuunnittelijoita tekemään perusteltuja valintoja RTOS:n suhteen. Jatkossa suositellaan lisätutkimusta kontrolloiduilla kokeilla modernilla sulautetulla laitteistolla tulosten vahvistamiseksi ja laajentamiseksi.

Avainsanat: Sulautetut järjestelmät, RTOS, reaaliaikakäyttöjärjestelmät, rinnakkaisuus, suorituskyky

USE OF ARTIFICIAL INTELLIGENCE IN THIS WORK

Artificial intelligence (AI) has been used in generating this work:

☐ No
☒ Yes

I hereby declare, that the AI-based applications used in generating this work are as follows:

Application	Version
ChatGPT	GPT-4o

Purpose of the use of AI

The most common use case was to prompt the AI to rephrase something I had written, or something that had been written in a paper or a book, such as: "Rephrase this to make it clearer" or "Write this in one sentence." AI was also used for fixing grammar, often through Overleaf's Writefull integration, which uses a GPT LLM. Paragraphs from literature were given for AI to summarize to determine if they would be worth reading. AI was also used for generating drafts for sections in the text by prompting it with ideas for what I wanted to write about and the context of the thesis. This generated text was later heavily modified as well as updated with correct information and citations.

Parts of this work, where AI was used

AI was used for grammar and spelling in all chapters. The abstract was initially generated with AI by prompting it with everything I had written until then. Similarly, the conclusion was drafted initially by giving AI the RTOS comparison chapter as context. The abstract was translated with AI from English to Finnish.

Acknowledgement of risks

I hereby acknowledge, that as the author of this work, I am fully responsible for the contents presented in this thesis. This includes the parts that were generated by an AI, in part or in their entirety. I therefore also acknowledge my responsibility in the case, where use of AI has resulted in ethical guidelines being breached.

CONTENTS

LIST OF ABBREVIATIONS

ADC	Analog-to-digital converter
API	Application programming interface
CPU	Central processing unit
GPOS	General purpose operating system
I/O	Input/Output
ISR	Interrupt service routine
MMU	Memory management unit
OS	Operating system
RTOS	Real-time operating system

1. INTRODUCTION

Embedded systems are computers integrated into devices which are typically not classified as computers, and they often use microcontrollers which are low-power and low-cost microprocessors. Real-time embedded systems are a subset of this, where correct timing of the actions performed are critical. In the context of this thesis, "real-time" can be defined as follows:

> "A real-time system is one in which the correctness of the computations not only depends on the logical correctness of the computation but also on the time at which the result is produced. If the timing constraints of the system are not met, system failure is said to have occurred."[1, chap. 1]

Real-time systems can be categorized into soft and hard real-time systems. A soft real-time system is able to meet the desired deadlines most of the time, and a missed deadline might result in degradation of the service quality that the system provides. Hard real-time systems are able to, and they must meet them always. [2, pp. 401]. As an example, consider a modern automobile that is equipped with a wide range of real-time embedded systems that provide various functionalities. Systems such as airbags, anti-lock braking systems, and autonomous driving technologies can be classified as hard real-time systems, as failure to meet their computational deadlines could result in catastrophic consequences. In contrast, the multimedia and the audio system can be categorized as soft real-time systems. While responsiveness to user input and smooth sounding audio is desirable, a certain degree of latency in these systems is unlikely to lead to safety hazards.

Real-time operating systems (RTOS) are commonly used in embedded systems' software stack to help developers implement software where timely and predictable task execution is critical, such as automotive systems and medical devices. These systems often need to perform multiple tasks at the same time and share resources while meeting strict timing constraints. Inefficient handling of concurrent tasks can introduce performance overhead, potentially affecting system responsiveness and reliability. To add to the challenge, embedded systems are often limited by hardware in terms of memory, computing power, and energy consumption.

This thesis explores how different RTOS implementations manage simultaneous execution of multiple tasks, also referred to as concurrency, what performance overhead is

caused by using some of the concurrency control mechanisms they offer, and whether there are significant differences in overhead between various RTOSes. Understanding these aspects is important for the system designer when choosing the suitable RTOS and implementing reliable software for critical applications. [3]

The methodology for this thesis is a literature review using primarily peer-reviewed scientific research papers as sources. The primary sources were searched through various databases, primarily Tampere University's Andor and ACM digital library. Many papers were found from IEEE Xplore. Additionally, relevant articles and textbooks were found through the references of papers found in the initial database searches. Textbooks were used as sources for foundational information. Non-academic sources were used when necessary. For example, when examining and comparing the implementations of different RTOSes, their technical documentation and source code were referenced.

The scope of this thesis is limited to embedded systems using single core CPUs without memory management units (MMU) as these types of CPUs are used in microcontrollers, commonly used in small embedded systems, which are often running RTOSes. Therefore, concurrency in the context of this thesis does not mean that tasks are truly executing at the same time, since there is only one CPU. The microcontroller architectures in question are comparable to the ARM Cortex-M family.[4] When examining real-world implementations and performance, mainly FreeRTOS, Zephyr and uC/OS-II or uC/OS-III are used as examples because they are open-source, and commonly used in literature and in the real world.[5] RTOSes were compared based on the metrics used in the literature. The comparability of the results found in the literature was considered based on factors like the hardware they were studied on and software versions used as well as the differences in the benchmark arrangements. Concurrency control is analyzed in a real-time operating system context even though it is possible to implement concurrency control without an RTOS.

2. OPERATING SYSTEMS AND MULTITASKING

An operating system (OS) is essential software that manages the hardware resources of a computer system and provides a consistent interface to application software. It is responsible for essential functionality such as memory management, input/output (I/O) handling, and most importantly for this thesis, multitasking and concurrency control. While modern general purpose operating systems (GPOSes) often provide complex abstractions and rich features tailored for desktop or server environments, the foundational concepts are equally relevant in the context of real-time operating systems (RTOSes) deployed in real-time embedded applications.

2.1 Kernel

At the core of any operating system lies the kernel, which manages fundamental services such as task scheduling, interrupt handling, and memory protection. The kernel is typically placed in a privileged region of memory known as kernel space where the CPU is allowed to execute instructions in privileged mode, also known as kernel mode. Application code, however, typically runs in user space, which has restricted access to system resources. This separation enhances system stability and security by preventing user programs from directly manipulating hardware, interfering with core operating system functions, or other user programs.

There are two primary loosely defined kernel architectures: *monolithic* and *microkernel*. A monolithic kernel integrates most system services—such as file systems, networking stacks, and device drivers—into a single binary, which runs in kernel mode [6, sect. 1.7.1]. This approach, seen in general-purpose operating systems like Linux, is often too heavy for resource-constrained microcontroller-based embedded systems. *Microkernels*, by contrast, include only essential functions such as inter-task communication and scheduling in kernel space. Additional services run in user space [6, sect. 1.7.3]. This separation improves modularity and allows developers to exclude unneeded functionality.

In many embedded systems—particularly those based on microcontrollers without MMUs, the separation of kernel and user space does not exist, at least not in the same way that it does on desktop systems. Instead, all code, including both kernel and application components, executes in the same address space, at the same privilege level. This design

simplifies the system and reduces overhead, but it also places greater responsibility on developers to maintain software correctness and prevent unintended interactions between tasks and system functions.

2.2 Tasks and Scheduling

Multitasking means the simultaneous execution of multiple tasks on the CPU. A task, sometimes called a thread or process, is a distinct "path" of execution or a job that the CPU needs to do. In GPOSes, a process usually has its own virtual memory space, while threads within that process share memory. Each thread maintains its own execution context, including a program counter and stack, but global variables are shared. [6, ch 2.] However, in embedded environments without virtual memory support, these distinctions are less significant, so the thread, process, and task can be used interchangeably.

Since we are focused on single-core systems, multitasking is achieved with *concurrent* execution (not parallel), which might give the illusion of multiple tasks running at the same time. In reality, the scheduler switches between tasks rapidly, preserving their execution states. The scheduler is the component of the operating system responsible for deciding which task runs at any given time. It is able to forcibly change the running task by utilizing interrupts, which are signals that can be sent to the CPU by various entities asynchronously. In the case of the scheduler, a physical timer on the MCU is typically interrupting the CPU periodically, for example once every millisecond, to make it jump to an interrupt service routine (ISR) where it evaluates which task should be running currently. This method of scheduling where tasks can be interrupted by other tasks is called preemptive scheduling. [7, 8, 9]

Figure 2.1 represents different states that the task can be in. There can be other states depending on the operating system implementation, but this example is picked from FreeRTOS due to its simplicity. The 'Ready' state means that the task is available to be scheduled and the scheduler will consider it when deciding which task should run. If the task is in the 'Blocked' state, it is waiting for something, such as a timer to expire, and therefore it will be ignored by the scheduler. A task can be disabled for an arbitrary period of time by setting it as 'Suspended', and the 'Running' task is naturally the task that is currently scheduled to be running on the CPU.

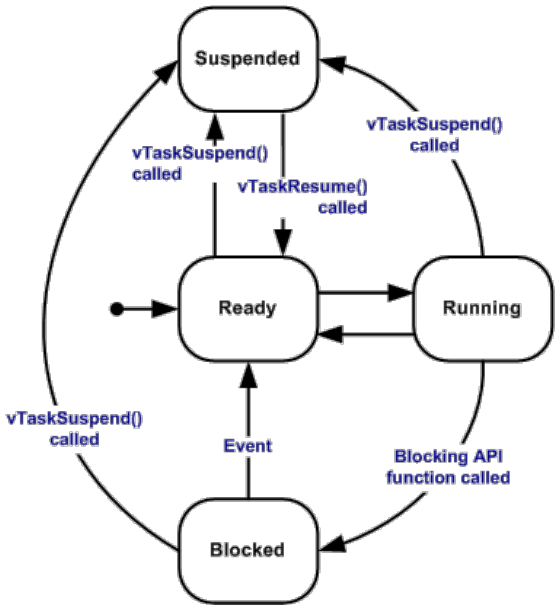


Figure 2.1. *Task states in FreeRTOS [10]*

In general-purpose operating systems, schedulers are often designed to optimize for responsiveness for the user and give a "fair" resource distribution among competing processes [10]. In real-time embedded systems, scheduling based on task priorities, which are assigned explicitly by the developer, is preferred to ensure that the most important tasks get done within their deadlines. A great number of different scheduling algorithms exist, but exploring them is out of scope of this thesis.

2.3 Real-Time Operating Systems

While general-purpose operating systems are designed to perform well under diverse workloads and conditions, they do not guarantee predictable behavior in terms of execution timing. In contrast, RTOSes are designed specifically for applications where meeting timing constraints is critical to the system's correct operation. In other words, GPOSes are optimized for best performance in the average case, while RTOSes are optimized to minimize the difference between best-case and worst-case execution times in order to get predictable performance. [2, sect 10.2]

RTOSes typically use deterministic scheduling algorithms, such as preemptive fixed-priority scheduling, and provide low-overhead mechanisms for task synchronization and inter-task communication [10, 11]. In fixed-priority preemptive scheduling, tasks are assigned priorities by the developer, and the scheduler always runs the highest-priority task that is ready [12]. RTOSes often also allow co-operative scheduling, which requires each task to manually yield and give up the CPU for other tasks to use, instead of the operating system doing so forcefully when using preemptive scheduling.

In many embedded environments, the need for fine-grained control and minimal overhead leads developers to choose RTOSes that are lightweight and customizable. These operating systems usually adopt microkernel-like designs and expose direct APIs for task management, inter-task signaling, and resource protection. For example, FreeRTOS by default provides only the core real-time scheduling functionality, inter-task communication, timing and synchronization primitives, and features like the command console interface and networking stacks can be added by the developer if needed [10]. Although RTOSes may appear limited compared to general-purpose operating systems, their design is guided by very different requirements. The predictability of execution, responsiveness to interrupts, and simplicity of the kernel are all features that make RTOSes well-suited for managing concurrent execution in resource-constrained embedded systems—a topic explored in the chapters that follow.

3. CONCURRENCY PROBLEMS AND SOLUTIONS

There are a handful of common problems which are usually encountered in concurrent programming. In this chapter, a few of them are explained briefly along with typical solutions. RTOS kernels offer application programming interfaces (API) for concurrency primitives, which allow programmers to implement software in a way that allows concurrent tasks to run efficiently and safely.[1] This chapter gives a brief introduction to basic topics in the field of concurrency.

3.1 Race conditions and mutex

A race condition occurs when multiple threads attempt to access and modify a shared resource simultaneously. This can lead to unpredictable behavior in the system and should be avoided.[6, sect. 2.3.1] In the C code snippet presented in Code 3.1, consider a global variable "shared_var" which in this case is the shared resource, two threads A and B, and an operating system environment with preemptive scheduling. Both threads will call the "increment_if_zero" function and thread A will do so first. If thread A is preempted right after the if clause, and the scheduler schedules thread B next, thread B will still see the value of shared_var as zero and evaluate the if clause to be true. Now both threads will increment "shared_var" by one. Thread A will do so when it is scheduled again and thread B will do it right away or later if it happens to also be preempted before the increment. Just by looking at the function's code, one might think that "shared_var" can only ever be incremented once and the maximum value it can reach is one, but a race condition can cause the increment to occur multiple times.

Code 3.1. *Example of a potential race condition in code*

```
1 int shared_var = 0;
2 void increment_if_zero() {
3   if (shared_var == 0) {
4     /* Preemption here */
5     shared_var += 1;
6   }
7 }
```

A region in the code where a race condition can happen is called a critical section. There, *mutual exclusion* is needed to ensure that only one thread at a time can access a shared resource. Since the problem of race condition is often initiated by an interrupt causing another task to be scheduled, one solution to this is to disable interrupts entirely for the duration of the critical section.[6, sect. 2.3.3] However, this is a suboptimal solution, especially for real-time systems, where disabling interrupts for too long can delay the execution of higher-priority tasks, potentially causing missed deadlines and system instability since the scheduler depends on timer interrupts to function. Another solution could be a simple boolean flag variable to indicate if a thread has entered the critical section. The problem with this approach is the fact that checking and writing to this variable is not *atomic*, meaning that the operation requires multiple CPU instructions to complete. Since an interrupt can occur after any instruction, this will not solve the issue completely.

A mutex (short for mutual exclusion) is a concurrency primitive which is a useful tool for solving race conditions. The mutex is a special lock-like object which guarantees atomicity. A thread can lock the mutex atomically with a library function call which interacts with the kernel. This call checks if the mutex is available, i.e. another thread has not locked it yet. If the mutex is available, the thread locks it and the thread is marked as the owner internally to the mutex. [2, sect. 5.10.2] After that, the thread is cleared to proceed to the critical section. If some other thread has the mutex when trying to lock it, the thread yields, meaning that it allows the OS scheduler to give CPU time to another thread. The thread is set to a 'blocked' state, waiting for the mutex to unlock. Once the mutex becomes available, the thread is set to a 'ready' state and it can be scheduled again. When that time comes, the thread enters the critical region and unlocks the mutex afterwards with another API call, so that the critical region can be re-entered.[6, sect. 2.3.6]

Code 3.1 can be improved by using a mutex to implement mutual exclusion to "shared_var" as seen in Code 3.2. The mutex type and functions used for mutex operations depend on the implementation and interface of the RTOS.

Code 3.2. *Example of using a mutex to prevent a race condition*

```
1  int shared_var = 0;
2  mutex_t shared_var_mutex;
3  void increment_if_zero() {
4    mutex_lock(shared_var_mutex);
5    if (shared_var == 0) {
6      /* Preemption here */
7      shared_var += 1;
8    }
9    mutex_unlock(shared_var_mutex);
10 }
```

3.2 Semaphores and synchronization

In embedded software where there are multiple tasks, some of which need to communicate with hardware and wait for the response, it is important that tasks are executed in the correct order. For example, there might be an analog-to-digital converter (ADC) that produces inputs to the system, and each input should be processed by a task. The ADC might generate a hardware interrupt when the conversion is ready. There needs to be a mechanism for the task to know when new data is available, and it would be desirable if the task would be blocked when there is nothing to do, so that wasting CPU resources can be avoided.

A semaphore is similar to a mutex in the sense that it can be taken or released, and if taking is not possible, it will cause the thread to block. However, semaphores have an internal counter which can be incremented and decremented to more than one [13]. The maximum value of the counter can usually be configured by the programmer. Semaphores with maximum count of one are called binary semaphores. The semaphore increment and decrement operations are usually called up and down [6]. The thread will also block if trying to increment a semaphore past its maximum value. Semaphores do not consider ownership as mutexes do, so any thread that has access to the semaphore can increment and decrement it [2, sect. 5.9]. Code 3.3 shows a pseudo code example of an application using semaphores.

Code 3.3. *Example of using a semaphore to synchronize two threads*

```
1  semaphore_t adc_semaphore;
2  fifo_t adc_results;
3  void adc_irq_handler() {
4    fifo_enqueue(adc_results, read_adc());
5    semaphore_up(adc_semaphore);
6  }
7  void adc_processing_task() {
8    while (1) {
9      semaphore_down(adc_semaphore);
10     process_adc_value(fifo_dequeue(adc_results));
11   }
12 }
```

3.3 Priority inversion

Priority inversion, illustrated in Figure 3.1, is a problem which can occur in a system where a high priority task depends on a resource that can be locked by a low priority task. Consider tasks H (high priority), M (medium priority), and L (low priority). Task L and H

depend on the same shared resource, but task M does not. Now, task L enters the critical section and locks the shared resource. Then, it is preempted by task H that also wants access to the shared resource. However, since the resource is already locked by task L, task H blocks and starts waiting for it to be unlocked. The problem is made worse when task M preempts task L while it is still in the critical section. Because task M has a higher priority than task L, it can preempt task L and consequently cause task H to wait even longer for the shared resource. [14]

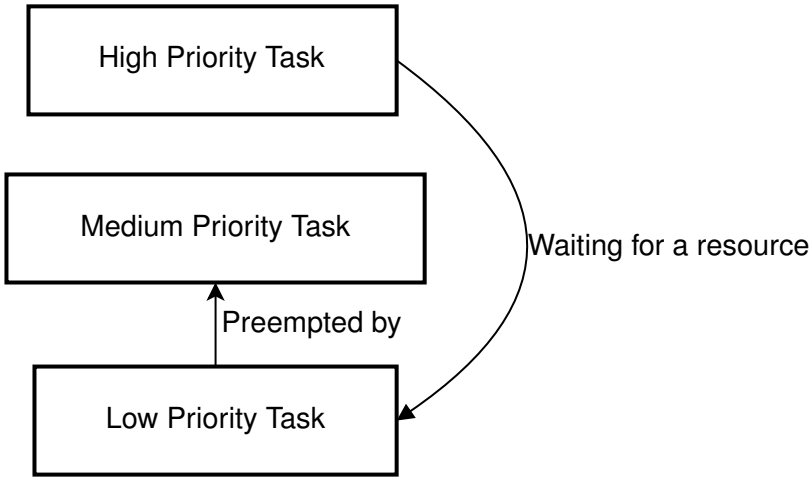


Figure 3.1. *Priority inversion*

3.4 Priority inheritance and ceiling

Priority inheritance is a method for alleviating the problems caused by priority inversion. Many RTOSes implement it for some concurrency primitives, usually for mutexes, because they are aware of the task which owns the lock. Considering the example described in section 3.3, the problem can be made less severe if the priority of task L is temporarily elevated to be equal to the priority of task H for the duration of it holding the lock. Then, task L will have "inherited" the priority of the higher priority task which it blocks, and this causes it to release the lock sooner. Also, task M can no longer preempt task L, since task L has temporarily a higher priority. [14]

The priority ceiling protocol is a static mechanism for preventing deadlocks and reducing blocking time. Deadlock is a situation where two or more tasks are indefinitely blocked, each waiting for a resource held by the other. Each resource is assigned a priority ceiling, which is defined as the highest priority of any task that may access it. A task is only allowed to enter a critical section if its priority is strictly higher than the priority ceilings of all resources currently locked by other tasks. If this condition is not met, the task is blocked, and the task holding the resource may inherit the blocked task's priority. This ensures that lower-priority tasks do not delay higher-priority ones by holding shared resources for extended periods. The protocol prevents circular wait conditions and limits blocking to at

most one lower-priority task. [14]

3.5 Jitter

Jitter refers to the variability in the timing of task execution or response times in real-time systems. For example, if a task is required to be activated exactly every 10 milliseconds, but due to jitter it sometimes starts after 9 milliseconds and other times after 12 milliseconds, the timing becomes unpredictable. Causes for this can be delays introduced by task preemptions, context switches, or concurrent resource contention. In real-time systems, where tasks must meet strict timing requirements, jitter can lead to missed deadlines, degraded system performance, or even failure of time-critical operations; therefore, minimizing jitter is desirable for maintaining the determinism and predictability required in real-time applications. Concurrency mechanisms like locks and semaphores can contribute to jitter when tasks are blocked or delayed while waiting for access to shared resources. This will be explored in the next chapter. [15]

4. COMPARING REAL-TIME OPERATING SYSTEMS

The techniques used to facilitate concurrency control in real-time operating systems do not come without issues, one of them being performance overhead.[15] The aim of this chapter is to explore the concurrency primitives used in different RTOS implementations and compare the performance between them. The criteria for what is considered "good" performance may vary between the requirements of the real-world system. For hard real-time systems, predictable performance of the RTOS, which could mean, for instance, a low amount of jitter, might be preferred over outright quickness. Softer or non-real-time applications might accept a higher amount of unpredictability in exchange for faster performance on average.

4.1 Implementation of synchronization primitives

FreeRTOS does not implement semaphores as primitives, but rather as message queues which hold empty elements, which might make them more complex than they really need to be [15]. However, FreeRTOS implements direct task notifications as an alternative to binary semaphores, which is supposed to reduce overhead, because there is no intermediary object in between. The performance benefit is claimed to be up to 45% with lower memory usage [10]. The limitation of using notifications is the fact that they can be sent only to a specific task.

The operating system needs to guarantee mutual exclusion on synchronization primitive operations. For RTOSes supporting multiple CPU architectures, a simple and portable solution to this is disabling interrupts for a small period of time, when for example, acquiring or releasing the mutex, to prevent multiple threads from doing so concurrently.[7, 8, 9] Though all RTOSes need to disable interrupts for a brief moment when doing operations on concurrency primitives, the length of the critical sections varies between implementations, and the performance consequences of that will be discussed in the next section.

In Zephyr, analysis of the semaphore operation functions shows that interrupts are disabled immediately upon entering these functions, similarly to the approach used in FreeRTOS. In uC/OS-III, the critical sections where interrupts are disabled in the equivalent semaphore functions are relatively short, therefore, their functions contain only very brief periods with interrupts disabled.[8, 9, 16]

FreeRTOS was criticized by multiple authors for its poor handling of priority inheritance. FreeRTOS does not implement priority ceiling to handle priority inversion, only priority inheritance. [17] FreeRTOS does not allow mutexes to be used from ISRs which limits its capability to handle priority inversion caused by threads that depend on ISRs.

4.2 Performance of synchronization primitives

A study by Belleza and Pignaton focuses on measuring the real-time performance of FreeRTOS, RIOT, uC-OS/II, and uC-OS/III, particularly in terms of task switching times, semaphore operations, and inter-task messaging [16]. They found that FreeRTOS, while relatively fast with task switching times in the order of microseconds, has significant jitter—meaning the timing between task switches isn't perfectly consistent. The same problems were found with Zephyr, while both uC-OS RTOSes show very low amounts of jitter. This can be seen in Figure 4.1.

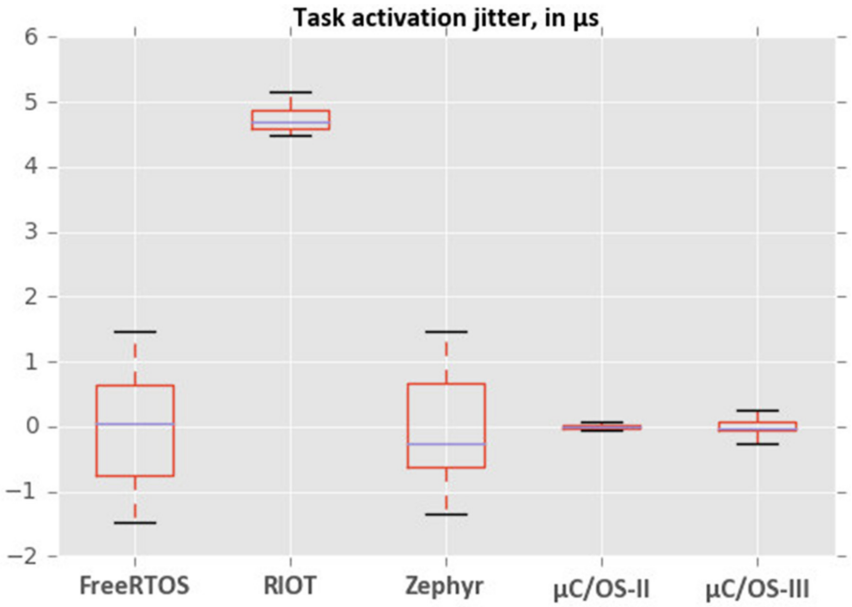


Figure 4.1. *Jitter variance in task activation times* [16]

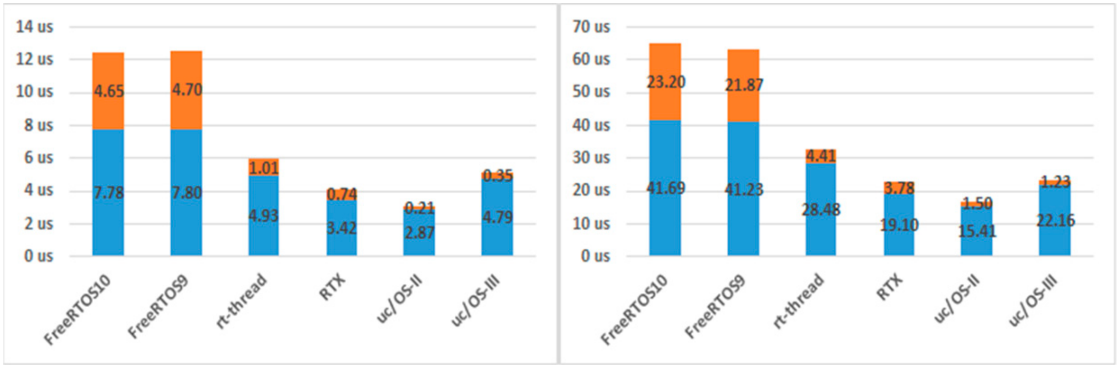
Belleza and Pignaton found that no single RTOS outperforms all others in every aspect; rather, each has strengths and weaknesses depending on the specific application and the features used. FreeRTOS, for example, is noted for its low memory usage and fast context-switching times, making it well-suited for resource-constrained IoT devices. However, it sacrifices advanced scheduling capabilities compared to other RTOS platforms. The study was comparing five different RTOSes with various benchmarks such as times for task switching, getting/releasing a semaphore, and inter-task messaging times.

Bertolotti and Kashani measured FreeRTOS's real-time performance being focused on measuring the time consumed by synchronization primitives in FreeRTOS to identify tim-

ing problems and analyze performance [15]. They found that FreeRTOS introduces a small but fixed overhead for these synchronization operations, which is typical for most RTOS. However, an interesting observation was that jitter occurred in the activation of the highest priority task, pointing to priority inversion issues. This phenomenon was suspected to be caused by the implementation of the operating system. According to the FreeRTOS documentation, semaphores do not provide priority inheritance, but mutexes do [18], which is also the case for uC-OS III[11].

The findings of Bertolotti and Kashani about significant jitter were consistent with the performance study conducted by Belleza and Pignaton, where FreeRTOS and Zephyr were found to have significantly more jitter compared to uC/OS-II and uC/OS-III even though neither of them implements priority inheritance for semaphores.

Ungurean compared RTOSes in a similar fashion as Belleza and Pignaton, though interestingly, the author included two versions of FreeRTOS, version 9 and 10 [19]. The version could be a significant factor when considering real-world performance. However, in this study, the performance differences between the two FreeRTOS versions were small, but measurable. The findings presented in this paper with regards to jitter are consistent with other studies, with uC/OS-II having significantly less jitter compared to FreeRTOS, especially when using semaphores. Figure 4.2 shows an example of benchmarking results from Ungurean's study for task switching latency from a semaphore event. The blue bar is the average time and the orange part is the jitter.

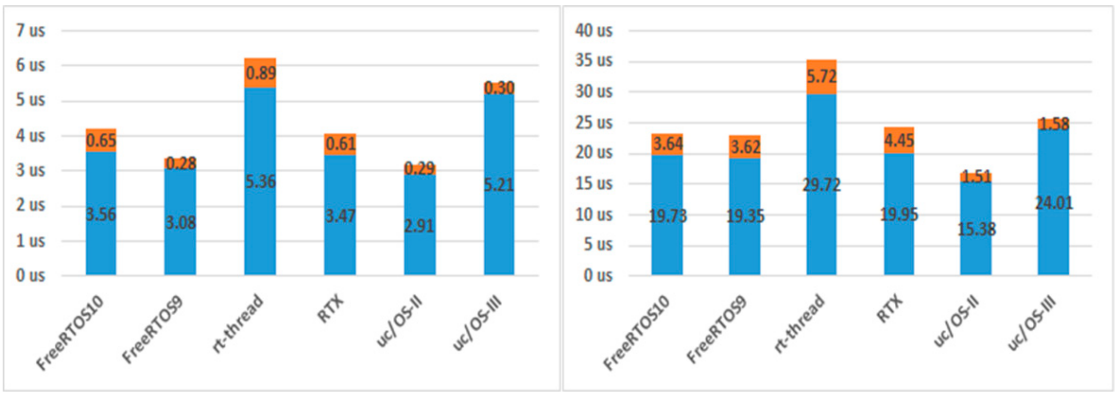


(a) STM32F407IG ARM Cortex™-M4 MCU (b) STM32L053R8 ARM Cortex™-M0+ MCU

Figure 4.2. *Time for task switching from the lowest priority task to a higher priority task triggered by a semaphore event* [19]

Figure 4.3 shows the results from Ungurean's study for the same benchmark as in Figure 4.2, but the task switching is initiated by a mailbox event. The author did not specify which FreeRTOS API was used, but presumably they were using queues. The results with mailbox events show considerably less jitter in FreeRTOS and it is comparable to other RTOSes studied. Interestingly, FreeRTOS version 9 outperforms the newer version 10. These results show that the choice of concurrency primitive and the RTOS family as

well as the version matter. [19]



(a) STM32F407IG ARM Cortex™-M4 MCU (b) STM32L053R8 ARM Cortex™-M0+ MCU

Figure 4.3. *The time for task switching from the lowest priority task to the higher priority task triggered by a 'mailbox' event* [19]

A paper by Fei Guang and colleagues analyzed the evolution of FreeRTOS performance by benchmarking its major releases from 3.0.0 to 8.0.0 [20]. Figure 4.4 shows an example result from the paper. It shows that in the early versions there were significant improvements to the semaphore acquire timing, but after version 5.0.0, there hasn't been any. This particular test was done in an environment where the amount of threads present in the system was varied, and the observation was that the semaphore acquisition time is linearly dependent on the amount of threads.

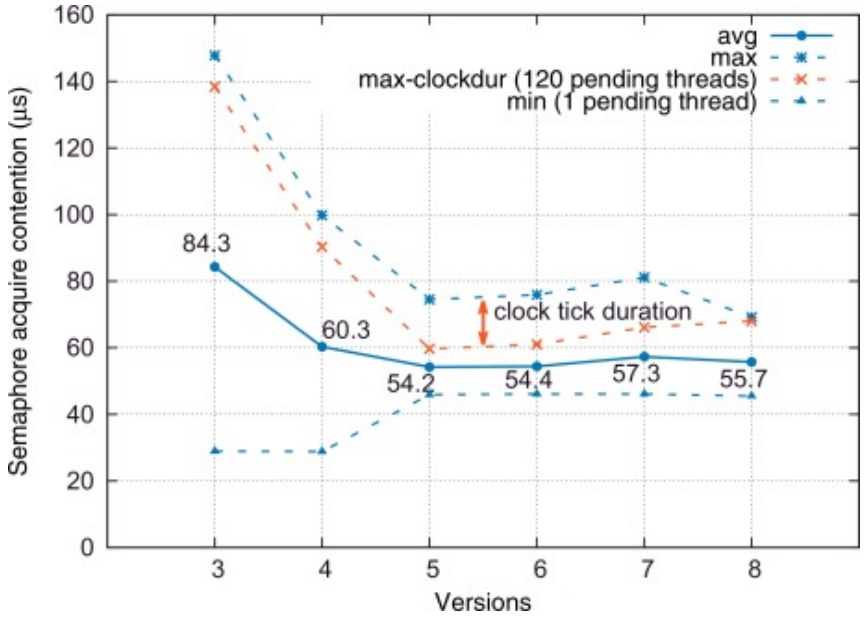


Figure 4.4. *Evolution of FreeRTOS semaphore acquire time over different versions* [20]

5. CONCLUSION

The goal of this thesis was to conduct a literature review, analyze concurrency handling techniques used in real-time operating systems, and compare the performance overhead caused by them across different RTOS implementations. The main findings were that different RTOSes make varying trade-offs between raw performance, predictability, and feature set when implementing concurrency primitives such as semaphores, mutexes, and task notifications.

FreeRTOS, was found to have low memory usage and relatively fast context-switching; however, it introduces more jitter in task switching compared to alternatives like uC/OS-II and uC/OS-III. While FreeRTOS implements optimizations such as direct task notifications to improve semaphore performance, its use of semaphores built on top of message queues, and its limited priority inversion handling, can introduce performance unpredictability. Studies by Belleza and Pignaton, as well as Bertolotti and Kashani, consistently observed that FreeRTOS and Zephyr exhibited higher jitter in synchronization primitive operations compared to uC/OS variants.

Zephyr's concurrency handling shows similar patterns to FreeRTOS, including relatively long critical sections where interrupts are disabled during synchronization operations. In contrast, uC/OS-III's design minimizes the time spent in critical sections, resulting in lower jitter and more predictable behavior, particularly valuable in hard real-time systems.

Multiple studies emphasized that no single RTOS universally outperforms the others across all performance dimensions. Instead, each system shows strengths depending on the task. The choice of concurrency primitive can also affect jitter and task switching times. For instance, when task switching was initiated by mailbox events, FreeRTOS demonstrated considerably less jitter compared to semaphore-based switching, and the results were comparable to other RTOSes studied. Version differences were also observed, with FreeRTOS version 9 slightly outperforming version 10 in some benchmarks.

In addition, Fei Guang and colleagues' analysis of FreeRTOS versions showed that significant improvements to semaphore acquisition times occurred up to version 5.0.0, after which no major improvements were observed. Semaphore acquisition times were also found to be linearly dependent on the number of threads in the system.

A challenge in this thesis was gathering and comparing results from different research

works, which were conducted using varied hardware platforms, software versions, and testing methodologies. The study by Fei Guang and colleagues on FreeRTOS evolution highlights a challenge in doing a literature review on performance, as studies from different years are compared and the results may vary between the RTOS versions used. These factors made direct comparisons difficult. To achieve more accurate and comparable results, a new empirical performance study using recent RTOS versions on standardized, widely used hardware would be beneficial. Additionally, a standardized benchmarking framework for RTOS performance evaluation could greatly assist developers in selecting the most suitable RTOS for their projects.

REFERENCES

[1] C. Walls. *Embedded RTOS Design*. Elsevier, 2021. ISBN: 9780128228517. URL: http://dx.doi.org/10.1016/C2019-0-02827-0.

[2] K. Wang. *Embedded and Real-Time Operating Systems*. eng. 1st ed. 2017. Cham: Springer International Publishing, 2017. ISBN: 3-319-51517-9.

[3] R. Xu, L. Zhang, and N. Ge. "Modeling and Timing Analysis for Microkernel-Based Real-Time Embedded System". In: *IEEE Access* 7 (2019), pp. 39547–39563. ISSN: 2169-3536. URL: http://dx.doi.org/10.1109/ACCESS.2019.2906011.

[4] J. Yiu. *ARM® Cortex®-M for Beginners*. [Accessed 25-03-2025]. 2016. URL: https://community.arm.com/cfs-file/__key/telligent-evolution-components-attachments/01-2142-00-00-00-00-52-96/White-Paper-_2D00_-Cortex_2D00_M-for-Beginners-_2D00_-2016-_2800_final-v3_2900_.pdf.

[5] Eetimes/embedded.com. *2017 Embedded Markets Study*. [Accessed 25-03-2025]. 2017. URL: https://www.eetimes.com/wp-content/uploads/2017-embedded-market-study1-1.pdf.

[6] A. S. Tanenbaum and H. Bos. *Modern operating systems*. eng. Fourth edition. Always learning. Boston: Pearson, 2015. ISBN: 978-1-292-06195-5.

[7] Amazon Web Services, Inc. *FreeRTOS: Real-Time Operating System Kernel*. Version Latest. Computer software. URL: https://github.com/FreeRTOS/FreeRTOS.

[8] The Zephyr Project. *Zephyr: Scalable Real-Time Operating System (RTOS)*. Version Latest. Computer software. URL: https://github.com/zephyrproject-rtos/zephyr.

[9] Weston Embedded Solutions. *μC/OS-III: Preemptive, Highly Portable, and Scalable Real-Time Kernel*. Version Latest. Computer software. URL: https://github.com/weston-embedded/uC-OS3.

[10] Amazon Web Services, Inc. *FreeRTOS Kernel Developer Docs - FreeRTOS™ — freertos.org*. [Accessed 17-02-2025]. URL: https://freertos.org/Documentation/02-Kernel/02-Kernel-features/00-Developer-docs.

[11] J. J. Labrosse. *µC/OS-III – The Real-Time Kernel*. [Accessed 10-03-2025]. 2010. URL: https://www.analog.com/media/en/dsp-documentation/software-manuals/micrium-ucos-iii-usersmanual.pdf.

[12] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings. "Fixed priority pre-emptive scheduling: An historical perspective". eng. In: *Real-time systems* 8.2-3 (1995), pp. 173–198. ISSN: 0922-6443.

[13] E. Dijkstra. "Co-operating sequential processes". English. In: *Programming languages : NATO Advanced Study Institute : lectures given at a three weeks Summer*

School held in Villard-le-Lans, 1966 / ed. by F. Genuys. United States: Academic Press Inc., 1968, pp. 43–112. ISBN: 0-12-279750-7.

[14] L. Sha, R. Rajkumar, and J. Lehoczky. “Priority inheritance protocols: an approach to real-time synchronization”. In: *IEEE Transactions on Computers* 39.9 (1990), pp. 1175–1185. ISSN: 0018-9340. URL: http://dx.doi.org/10.1109/12.57058.

[15] I. C. Bertolotti and G. G. Z. Kashani. “On the performance of open-source RTOS synchronization primitives”. In: *2015 IEEE 1st International Forum on Research and Technologies for Society and Industry Leveraging a better tomorrow (RTSI)*. 2015, pp. 398–402. DOI: 10.1109/RTSI.2015.7325131.

[16] R. Raymundo Belleza and E. Pignaton. “Performance study of real-time operating systems for internet of things devices”. In: *IET Software* 12.3 (June 2018), pp. 176–182. ISSN: 1751-8814. URL: http://dx.doi.org/10.1049/iet-sen.2017.0048.

[17] P. Parikh, R. Shah, U. Shah, and S. Deshmukh. “Performance parameters of RTOSs; comparison of open source RTOSs and benchmarking techniques”. eng. In: *2013 International Conference on Advances in Technology and Engineering (ICATE)*. IEEE, 2013, pp. 1–6. ISBN: 1467356182.

[18] Amazon Web Services, Inc. *FreeRTOS mutexes - FreeRTOS™ — freertos.org*. [Accessed 10-03-2025]. URL: https://www.freertos.org/Documentation/02-Kernel/02-Kernel-features/02-Queues-mutexes-and-semaphores/04-Mutexes.

[19] I. Ungurean. “Timing Comparison of the Real-Time Operating Systems for Small Microcontrollers”. In: *Symmetry* 12.4 (Apr. 2020), p. 592. ISSN: 2073-8994. URL: http://dx.doi.org/10.3390/sym12040592.

[20] F. Guan, L. Peng, L. Perneel, and M. Timmerman. “Open source FreeRTOS as a case study in real-time operating system evolution”. In: *Journal of Systems and Software* 118 (Aug. 2016), pp. 19–35. ISSN: 0164-1212. URL: http://dx.doi.org/10.1016/j.jss.2016.04.063.