University of Stuttgart
Institute for Control Engineering
of Machine Tools and Manufacturing Units
(ISW)

Master's Thesis

Wait-free synchronization for inter-process communication in real-time systems

submitted by

Devrim Baran Demir

from Hamburg

Degree program	M. Sc. Informatik
Examined by	Prof. Andreas Wortmann
Supervised by	Marc Fischer
Submitted on	April 1, 2025

Declaration of Originality

Master's Thesis of Devrim Baran Demir (M. Sc. Informatik)

Address	Kernerweg 22, 89520 Heidenheim an der Brenz
Student number	3310700
English title	*Wait-free synchronization for inter-process communication in real-time systems*
German title	*Wait-free Synchronization für Interprozesskommunikation in Echtzeitsystemen*

I now declare,

- that I wrote this work independently,
- that no sources other than those stated are used and that all statements taken from other works—directly or figuratively—are marked as such,
- that the work submitted was not the subject of any other examination procedure, either in its entirety or in substantial parts,
- that I have not published the work in whole or in part, and
- that my work does not violate any rights of third parties and that I exempt the University against any claims of third parties.

Stuttgart, April 1, 2025

Kurzfassung

Vorhersehbare und korrekte Interprozesskommunikation (IPC) ist für Echtzeitsysteme von entscheidender Bedeutung, da Verzögerungen, Unvorhersehbarkeit oder inkonsistente Datenstände zu Instabilität und Ausfällen führen können. Traditionelle Synchronisationsmechanismen verursachen Blockierungen, die zu Prioritätsinversionen und erhöhten Antwortzeiten führen. Um diese Herausforderungen zu bewältigen, bietet die wartefreie Synchronisation eine Alternative, die den Abschluss von, wie der Austausch von Daten zwischen mehreren Prozessen, in einer begrenzten Anzahl von Schritten garantiert und so die Systemreaktionsfähigkeit und -zuverlässigkeit sicherstellt.

Diese Arbeit untersucht die Nutzung von wait-free Datenstrukturen für IPC in Echtzeitsystemen mit Fokus auf deren Implementierung in Rust. Das Eigentumsmodell und die strengen Nebenläufigkeitsgarantien von Rust machen es besonders geeignet für die Entwicklung latenzarmer und hochzuverlässiger Synchronisationsmechanismen. Diese Arbeit analysiert bestehende wait-free Methoden für IPC in Echtzeitsystemen und bewertet ihre Leistung im Vergleich zu herkömmlichen Synchronisationsmethoden.

Stichwörter: Echtzeitsysteme, wait-free Sysnchronisation, lock-free Synchronisation, Interprozesskommunikation, rust

Abstract

Predictable and correct Inter-Process Communication (IPC) is essential for Real-Time System (RTS), where delays, unpredictability or inconsistent data can lead to instability and failures. Traditional synchronization mechanisms introduce blocking, leading to priority inversion and increased response times. To overcome these challenges, wait-free synchronization provides an alternative that guarantees operation completion, such as the completion of data exchange between multiple processes, within a bounded number of steps, ensuring system responsiveness and reliability.

This thesis explores the use of wait-free data structures for IPC in RTS, focusing on their implementation in Rust. Rust's ownership model and strict concurrency guarantees make it well-suited for developing low-latency and high-reliability synchronization mechanisms. This work examines existing wait-free techniques for real-time IPC, and evaluates their performance against conventional synchronization methods.

Keywords: real-time systems, wait-free synchronization, lock-free synchronization, inter-process communication, rust

Contents

1. Introduction

1.1. Motivation

In modern manufacturing and automation, control systems must operate under strict timing constraints to function reliably. If a system fails to meet these constraints, unexpected delays can disrupt processes, leading to instability or even hazardous failures in safety-critical environments. For this reason, Real-Time Operating System (RTOS) and low-level programming languages like C, C++, and Rust are widely used to ensure predictable execution times.

Many real-time applications involve multiple tasks that must run concurrently and share resources efficiently. Without proper synchronization, problems such as data corruption or race conditions can occur leading to unpredictable behavior. Traditional synchronization methods with locks are commonly used to manage access to shared resources by blocking processes so that only one process at time accesses the shared resource to exchange data in a proper way. However, these blocking mechanisms introduce significant challenges in real-time settings. Since Traditional synchronization methods require processes to wait for resource availability, they can lead to increased response times, potential deadlocks, potential process starvations, and potential priority inversions. These delays are unacceptable in systems that require strict timing guarantees. [1]–[3]

To overcome these limitations, there is an increasing interest in synchronization techniques without any blocking mechanisms. A lock-free algorithm for instance functions withouth any locking mechanism thus no blocking. This guarantees that at least one process completes in a finite number of steps, regardless of contention (multiple processes try to access the same shared resource). This property ensures that at least the system will still work eventhough one process might be lagging. The only problem is that this will not process starvation since there is no guarantee that every process will finish its task. [4]

A wait-free algorithm guarantees that every operation completes in a finite number of steps, regardless of contention (multiple processes try to access the same shared resource). This property ensures system responsiveness and predictability, which are essential for real-time applications. By eliminating blocking and contention-based delays, wait-free synchronization prevents priority inversion and ensures that high-priority tasks execute without interference. [1], [2], [4]

Efficient synchronization mechanisms are particularly important in the context of IPC, which plays a crucial role in RTS. IPC allows processes to exchange data efficiently, but its performance is heavily influenced by the synchronization techniques used. Traditional IPC mechanisms, which often rely on blocking some processes, can introduce significant latency and reduce throughput. Wait-free data structures offer a promising alternative by ensuring that communication operations complete within predictable time bounds. However, selecting appropriate wait-free data structures and evaluating their performance in real-time environments remains a challenge. [5]–[8]

The Rust programming language provides useful features for implementing real-time synchronization mechanisms. Its ownership model and strict type system prevent data races and enforce safe concurrency without requiring traditional locking mechanisms. Additionally, Rust offers fine-grained control over system resources, making it a strong candidate for real-time applications that demand both low latency and high reliability. [9], [10]

The concepts and methods introduced here, including RTS, IPC, synchronization techniques and problems, wait-free synchronization, and the rust programming language are explored in greater depth in chapter 2.

1.2. Objective

The primary goal of this research is to find a wait-free data structure that can be used to implement a wait-free synchronization for IPC though shared memory in RTS using Rust. To do so, this study aims to:

- Identify and analyze existing wait-free synchronization techniques for IPC through shared memory for RTS.
- Implement and compare the performance of existing wait-free synchronization mechanisms for IPC through a shared memory for real-time scenarios with each other.
- Choose and analyze which wait-free data structure for IPC through shared memory in a real-time setting using Rust that is best suited.

By addressing these objectives, this work contributes to the field of wait-free synchronization for IPC in RTS by providing a practical solution with rust. The insights gained from this research can help improve the reliability and performance of real-time applications across various domains.

1.3. Structure of the Thesis

2. Background

To establish a clear foundation for the concepts and definitions introduced throughout this thesis, we provide a fundamental overview of the key topics relevant to this research. This includes an introduction to RTS, Inter-Process Communication (IPC), and synchronization techniques, with a particular focus on wait-free synchronization. Additionally, we examine the Rust programming language, as it serves as the primary development environment for this study. Furthermore, we explore existing synchronization methods in RTS to contextualize the motivation and contributions of this work.

2.1. Real-Time Systems

In RTS the correctness of the system does not only depend on the logical results of computations, but also on timing constraints. These systems can be classified into Hard Real-Time System (HRTS) or Soft Real-Time System (SRTS). HRTS have strict timing constraints, and missing a constraint is considered a system failure and may lead to a catastrophic desaster. The sysetem must guarantee that every timing constraint has to be met. An use case would be industrial automation where all the machines and robotic modules have to communicate with each other as quick as possible to ensure no blockage of the manufacturing line. [11]

On the other hand, SRTS try to stick to the timing constraints as much as possible, but missing some timing constraints is not considered a system failure. Infrastructure wise SRTS are similar to HRTS, since it is still considered important to meet these timing constraints. An example would be a multimedia system where it would be considered fine if sometimes frames are dropped to guarantee the video stream. [11]

Sometimes these two systems appear in combination, where some functions have hard real-time constraints and some have soft real-time constraints. Krishna K. gives a good example in his paper where he describes that for the apollo 11 mission some components for the landing processes had soft real-time behavior and the rest still functioned with hard real-time constraints. [11]

Since the workfield of this thesis is within HRTS, the term RTS will be used synonymously with the terminology HRTS.

2.2. Inter-Process Communication

Now the processes used in a RTS also have to share information with each other so the system can function. So some kind of IPC is needed. IPC allows processes to share information with each other using different kind of methods. We will mainly focus on one method explained later. In general IPC is needed in all computing systems, because processes often need to work together (e.g. a producer process passes data to a consumer process). Lets take the brake-by-wire technology as example. Brake-by-wire is a technology for driverless cars where some mechanical and hydraulic components from the braking systems are replaced by wires to transmit braking signals, since there is no driver anymore to press on the braking pedal [12]. This of course requires different processes to share information together. In the context of this thesis this kind of communication requires strict timing constraints as stated as before, since any kind of delay or blockage would lead to fatal consequences. [13], [14]

To achieve IPC different kind of mechanisms are used. The focus for this work lies on shared memory, so that is also the mechanism we will look into.

2.2.1. Shared Memory

To achieve any kind of information sharing between processes, these processes will need to have access to the same data regularly. With a shared memory segment, multiple processes can have access to the same memory location. So all processes which are part of the IPC can read and write to this common memory space avoiding unnecessary data copys. With that processes exchange information by directly manipulating memory. This kind of IPC is particular useful for real-time applications, which handle large volumes of data or are required to quickly transfer data between sensors and control tasks. What is also important to know is that the section of the code, that programs these data accesses by different processes is called critical section. [14]–[17]

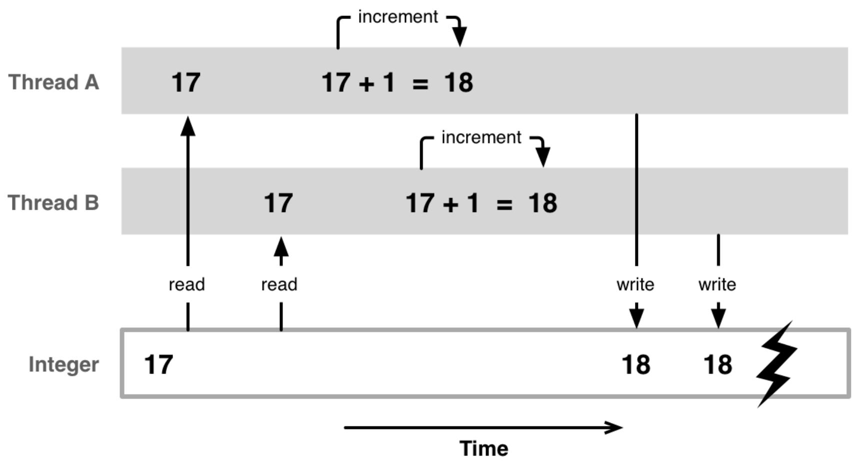
The problem with this is that the system somehow has to manage how the processes access the shared memory segment. This is mostly done by using different kind of synchronization techniques. Without any synchronization mechanism race conditions or inconsistent data can occur. [14], [15]

2.3. Synchronization

So as we see, synchronization is a crucial part of IPC in RTS, especially when processes communicate via shared memory. Communication through shared memory always has a risk of race conditions (when multiple processes acces the same data and cause unexpected behaviour) and data inconsistency if the processes are not properly synchronized. Tratitional synchronization techniques ensure mutual exclusion (only one process at a time uses shared resource) thus avoiding race conditions and ensuring data consistency. Race conditions happen when for example two processes write to the same resource.

Lets take a single counter instance with value 17 as a shared resource in a shared memory region. If one has process p1 and preocess p2 that increment the number we would think that the end result is 19. But what could happen is that p1 reads the value 17 and then p2 reads the value 17. Now internally both processes increment that number to 18 and both processes would write 18 to that shared resource. To understand this example more in detail fig. 2.1 visualizes a race condition with threads. The difference between processes and threads is just, that threads are part of a process which can perform multiple tasks simultaneously within that process [18]. So the concept shown in fig. 2.1 can be used for multiple processes too.

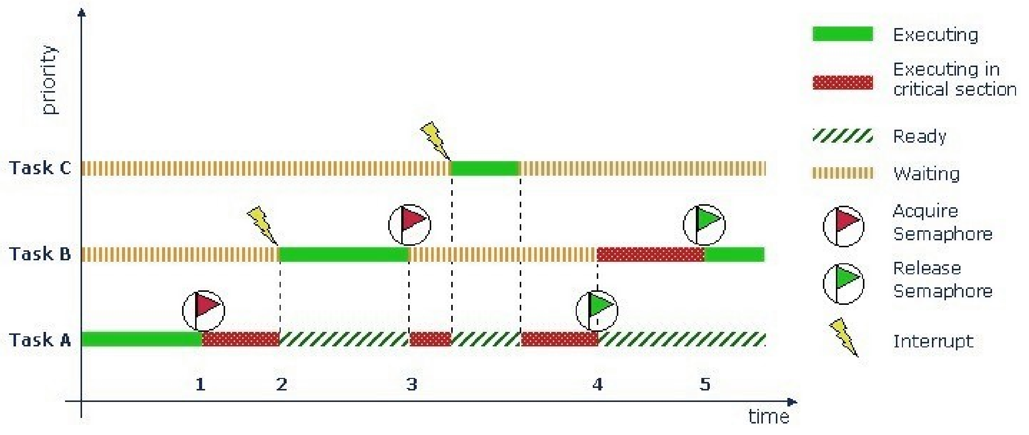
Figure 2.1.: Race condition between two threads, which write to the same shared variable [19].



2.3.1. Mutual Exclusion

As discussed mutual exclusion does only allow one process or thread to access the shared resource at a time. This includes that if a process p1 already accessed the shared resource x and is still working on it, a second process p2, which trys to access that shared resource x has to wait untill the process p1 finishes its task, where it needs that shared resource x. To achieve this mostly synchronisation techniques based on locks or semaphores are used to block the entry of an process to an already accessed and in use shared resource of an other process. See fig. 2.2 to gain an deeper understanding on how this works. This paper will not go into detail how traditional synchronisation techniques like locks or semaphores work, since for this work it is only important that these kind of methods manage the access of processes to shared resources in shared memorys via some kind of locks. A process will aquire a lock to access a shared resource and will release it, when its task is done. Another process trying to access the same resource while its in use has to wait untill the lock is released for that resource.

Figure 2.2.: Mutual exclusion between three tasks(processes), which access the same critical section. Multiple processes need to stop working and just wait for other processes to finish their work. See the waiting phase of the processes. [20]



It is clear that this approach inherently relys on blocking a set of processes. This may lead to several issues, including deadlocks, process starvation, priority inversion, and increased response times. [2], [21]

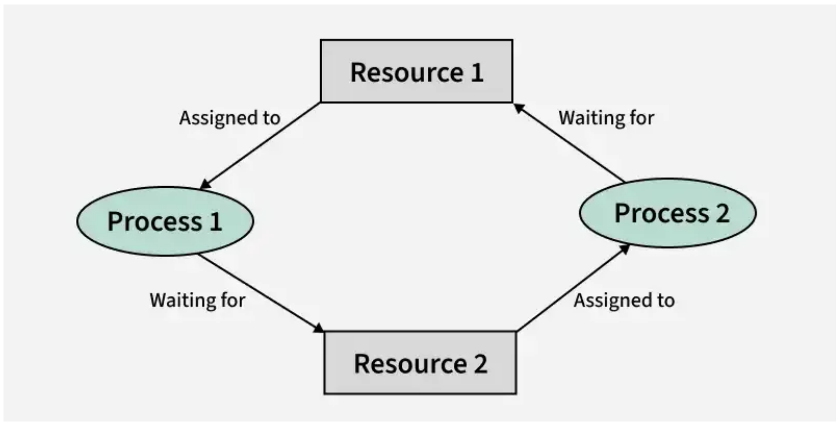
Deadlock

A deadlock happens when each process of a set of processes are waiting for a resource held by other processes in the same set. The processes which are in a deadlock wait indefinetly for the resources and never terminate. So the resources that these waiting processes hold are also not released and are also not available to other processes. As one can see, this leads to a situation where no process can make any process. To understand this better fig. 2.3 visualizes a deadlock with two processes. In this example Process 1 holds Resource 1 and waits for Resource 2, while Process 2 holds Resource 2 and waits for resource r1. So both processes are waiting for each other to release the needed resources. [22], [23]

Process Starvation

So will wait to enter the criticial section. And a process that trys to entry the critical section, must eventually enter. Starvation is when that particular process will never enter the critical section. This usually happens when a synchronization method allows one or more processes to make progress while ignoring a particular process or processes. So this requires some sort of low and high priority processes. When we always have high priority processes available and some low priority processes, it might happen that these

Figure 2.3.: Deadlock between two processes, which wait for each other to release the needed resources [22].



low priority processes will never be able to enter the critical section. This is a problem, since these low priority processes might be important for the system. [24]

Priority Inversion

What could happen too is that while a low priority process is accessing a shared resource, a high priority process is waiting for that resource. This is called priority inversion. [25]

Increased Response Times

So as we see traditional synchronization might lead to problems. But even if these problmes do not occure we can see that mutual exclusion will always lead to waiting processes which leads to increased response times. This could lead to not meeting the timing constraints of a HRTS environment. So synchronization mechanisms are needed that do not introduce blocking.

2.4. Lock-Free Synchronization

As we see, we need synchronization techniques that do not block processes with any kind of locking mechanism as we saw before. One way could be the implementation of lock-free syschronization techniques. This would allow multiple processes to access the shared resource concurrently. Lock-free synchronization ensures that at least one process will make progress in a finite number of steps. However some processes may be unable to proceed, because lock-free synchronization does not guarantee that all processes will complete their operations in a finite number of steps. This means that starvation or even prioritiy inversion is still possible, as some processes, even high priority processes may be indefinitely delayed by others. There are different kind of mechanisms to achieve this.

In this work the focus will lie on queue based techniques only to not exceed the limits of the thesis. We will look into one technique to understand in detail how lock free works. Many wait-free queues for example builds appon the lock-free technique introduced by Michael and Scott. [26]

2.4.1. Michael and Scott's Lock-Free Queue

Michael and Scott developed an algorithm seen in algorithm 1 using a linked-list as a shared data structure with a enqueue and a dequeue function to introduce lock-freedom. A linked-list is a list containing nodes containing data and a pointer called next which references the next node in the list, which can only be traversed in a single direction. There is also a pointer called head, which references the beginning node of the list and a pointer tails, which references the end node of the list. The core concept of the algorithm is the enqueue and dequeue functions in line 7 and 26 in algorithm 1 which are used to add and remove nodes to the shared data structure. When a process trys to add a node to the list, it first creates a new node and sets its next pointer to NULL. Then it tries to link the new node to the end of the list by using a Compare and Swap (CAS). This operation atomically compares the current value of the tail pointer with the expected value and, if they match, updates the tail pointer to point to the new node. If another process has already modified the tail pointer, the CAS operation will fail, and the process will retry until it succeeds. The dequeue function works similarly, but it removes nodes from the front of the list. It also uses CAS to update the head pointer and remove nodes from the list. We can observe that this approach does not need any locks explained in section 2.3. [26] The algorithm also uses a technique called helping, where processes assist each other in completing their operations by updating pointers when they notice that another process is stuck. This happens in both enqueueing and dequeueing. When a process for example sees that the tail pointer is not pointing to the last node, because tail.ptr.next in line 13 or 30 in algorithm 1 is not NULL, it means that another process started it process by adding its node to the list, but is somehow lagging behind. So another process that started this process too will try to update the tail pointer to point it to the last node. This makes certain that every process can make progress independently from any contention, since the tail truly stays the last node added. [26] The function Initialize we which starts at line 1 in algorithm 1 is just used to to create dummy nodes when there's no node in the list. This just simplifies the algorithm so that the head and tail pointers are not null. [26] But as we see one problem may occur. If a process is trying to enqueue or dequeue, it can happen that the CAS loop might fail indefinetly if other processes are constantly modifiying the tail or head pointer. This means that in very high contention scenarios, a process may be delayed indefinetly and starved out. In a broader sense some kind of priority inversion could be also the conclusion of process starvation, since the process starved out could be a higher priority process. This is why we need a slightly different approach which guerantees that every process will complete its operation in a finite number of steps. That approach will be explained in the next section. [26]

Algorithm 1 Michael and Scott's Lock-Free Queue [26].

```
function INITIALIZE(Q : pointer to queue_t)
    node = new node()                                          ▷ Allocate a dummy node
    node.next.ptr = NULL                                       ▷ Make it the only node in the list
    Q.Head = node                                              ▷ Both Head and Tail point
    Q.Tail = node                                              ▷ to this dummy node
end function
function ENQUEUE(Q : pointer to queue_t, value : data_type)
    node = new node()                                          ▷ Allocate a new node from the free list
    node.value = value                                         ▷ Copy enqueue value into node
    node.next.ptr = NULL                                       ▷ Set next pointer of node to NULL
    loop                                                       ▷ Keep trying until Enqueue is done
        tail = Q.Tail                                          ▷ Read Tail (pointer + count) together
        next = tail.ptr.next                                   ▷ Read next ptr + count together
        if tail == Q.Tail then                                 ▷ Are tail & next consistent?
            if next.ptr == NULL then                           ▷ Tail is the last node?
                if $CAS(\&tail.ptr.next, next, \langle node, next.count + 1\rangle)$ then
                    break                                      ▷ Link the new node; Enqueue is done
                end if
            else                                               ▷ Tail not pointing to the last node
                CAS(&Q.Tail, tail, ⟨ next.ptr, tail.count+1⟩)  ▷ Move Tail forward (helping another enqueuer)
            end if
        end if
    end loop
    $CAS(\&Q.Tail, tail, \langle node, tail.count + 1\rangle)$   ▷ Final attempt to swing Tail to the inserted node
end function
function DEQUEUE(Q : pointer to queue_t, pvalue : pointer to data_type)
    loop                                                       ▷ Keep trying until Dequeue is done
        head = Q.Head
        tail = Q.Tail
        next = head.ptr.next                                   ▷ Read head->next
        if head == Q.Head then                                 ▷ Still consistent?
            if head.ptr == tail.ptr then                       ▷ Empty or Tail behind?
                if next.ptr == NULL then                       ▷ Queue is empty
                    return FALSE
                else                                           ▷ Tail is behind, help move it
                    $CAS(\&Q.Tail, tail, \langle next.ptr, tail.count + 1\rangle)$
                end if
            else                                               ▷ No need to adjust Tail
                *pvalue = next.ptr.value                       ▷ Read value before CAS
                if $CAS(\&Q.Head, head, \langle next.ptr, head.count + 1\rangle)$ then
                    break                                      ▷ Dequeue is done
                end if
            end if
        end if
    end loop
    free(head.ptr)                                             ▷ Safe to free old dummy node
    return TRUE
end function
```

2.5. Wait-Free Synchronization

We saw that lock-freedom solves the problem, that a system can get into a deadlock. But this is not enough, in a fully automated modern manufacturing line for example we do not want any process to not complete its task, since that could mean that some modules in a automated manufacturing line would never finish their work in a worst case scenario. And since in such a line every part is necessary that would mean that the line would stop at some point. So we need a solution where every process finishes their task in a finite number of steps and not just one. Wait-free synchronization guarantees that every process will complete its operation in a finite number of steps, regardless of contention. This means that even process starvation is by definition not possible anymore. Since starvation cannot happen anymore priority inversion is eliminated too, because there is not anymore the possibility that a low priority process blocks the task of an high priority process. This ensures system responsiveness and predictability thus ability to meet timing constraints, which are essential for hard real-time applications. In section 2.7 we will look more into different techniques how wait-free synchronization can be achieved.

2.6. Rust Programming Language

2.7. State of the Art

3. Related Work

4. Methodology

5. Choosing optimal wait-free data structure

6. Implementation

7. Results

8. Conclusion and Future Work

Bibliography

[1] M. Herlihy, "Wait-free synchronization", *ACM Trans. Program. Lang. Syst.*, vol. 13, no. 1, pp. 124–149, Jan. 1991. DOI: 10.1145/114005.102808. [Online]. Available: https://doi.org/10.1145/114005.102808.

[2] B. B. Brandenburg, *Multiprocessor real-time locking protocols: A systematic review*, 2019. arXiv: 1909.09600 [cs.DC]. [Online]. Available: https://arxiv.org/abs/1909.09600.

[3] O. Kode and T. Oyemade, *Analysis of synchronization mechanisms in operating systems*, 2024. arXiv: 2409.11271 [cs.OS]. [Online]. Available: https://arxiv.org/abs/2409.11271.

[4] A. Kogan and E. Petrank, "A methodology for creating fast wait-free data structures", *SIGPLAN Not.*, vol. 47, no. 8, pp. 141–150, Feb. 2012. DOI: 10.1145/2370036.2145835. [Online]. Available: https://doi.org/10.1145/2370036.2145835.

[5] S. Timnat and E. Petrank, "A practical wait-free simulation for lock-free data structures", *SIGPLAN Not.*, vol. 49, no. 8, pp. 357–368, Feb. 2014. DOI: 10.1145/2692916.2555261. [Online]. Available: https://doi.org/10.1145/2692916.2555261.

[6] M. M. Michael and M. L. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms", in *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing*, ser. PODC '96, Philadelphia, Pennsylvania, USA: Association for Computing Machinery, 1996, pp. 267–275. DOI: 10.1145/248052.248106. [Online]. Available: https://doi.org/10.1145/248052.248106.

[7] H. Huang, P. Pillai, and K. G. Shin, "Improving Wait-Free algorithms for interprocess communication in embedded Real-Time systems", in *2002 USENIX Annual Technical Conference (USENIX ATC 02)*, Monterey, CA: USENIX Association, Jun. 2002. [Online]. Available: https://www.usenix.org/conference/2002-usenix-annual-technical-conference/improving-wait-free-algorithms-interprocess.

[8] A. Pellegrini and F. Quaglia, *On the relevance of wait-free coordination algorithms in shared-memory hpc:the global virtual time case*, 2020. arXiv: 2004.10033 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2004.10033.

[9] B. Xu, B. Chu, H. Fan, and Y. Feng, "An analysis of the rust programming practice for memory safety assurance", in *Web Information Systems and Applications: 20th International Conference, WISA 2023, Chengdu, China, September 15–17, 2023, Proceedings*, Chengdu, China: Springer-Verlag, 2023, pp. 440–451. DOI: 10.1007/978-981-99-6222-8_37. [Online]. Available: https://doi.org/10.1007/978-981-99-6222-8_37.

[10] A. Sharma, S. Sharma, S. Torres-Arias, and A. Machiry, *Rust for embedded systems: Current state, challenges and open problems (extended report)*, 2024. arXiv: 2311.05063 [cs.CR]. [Online]. Available: https://arxiv.org/abs/2311.05063.

[11] K. Kavi, R. Akl, and A. Hurson, "Real-time systems: An introduction and the state-of-the-art", in Mar. 2009. DOI: 10.1002/9780470050118.ecse344.

[12] X. Hua, J. Zeng, H. Li, J. Huang, M. Luo, X. Feng, H. Xiong, and W. Wu, "A review of automobile brake-by-wire control technology", *Processes*, vol. 11, p. 994, Mar. 2023. DOI: 10.3390/pr11040994.

[13] *Inter process communication (ipc)*, https://www.geeksforgeeks.org/inter-process-communication-ipc/, 2025.

[14] A. Venkataraman and K. K. Jagadeesha, "Evaluation of inter-process communication mechanisms", 2015. [Online]. Available: https://api.semanticscholar.org/CorpusID:6899525.

[15] K. C. Wang, "Process management in embedded systems", in *Embedded and Real-Time Operating Systems*. Cham: Springer International Publishing, 2023, pp. 115–168. DOI: 10.1007/978-3-031-28701-5_5. [Online]. Available: https://doi.org/10.1007/978-3-031-28701-5_5.

[16] S. Mogare, A. Mahamune, D. Sathe, H. Bhangare, M. Deshmukh, and A. Ingale, "Message passing vs shared memory-a survey of trade off in ipc", *International Research Journal of Modernization in Engineering Technology and Science (IRJMETS)*, vol. 06, Nov. 2024.

[17] M. A. Suleman, O. Mutlu, M. K. Qureshi, and Y. N. Patt, "Accelerating critical section execution with asymmetric multi-core architectures", in *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS XIV, Washington, DC, USA: Association for Computing Machinery, 2009, pp. 253–264. DOI: 10.1145/1508244.1508274. [Online]. Available: https://doi.org/10.1145/1508244.1508274.

[18] *Difference between process and thread*, https://opensourceforgeeks.blogspot.com/2014/01/race-condition-synchronization-atomic.html, 2025.

[19] A. Thakur, *Race condition, synchronization, atomic operations and volatile keyword.* https://opensourceforgeeks.blogspot.com/2014/01/race-condition-synchronization-atomic.html, 2014.

[20] *Managing mutual exclusion mechanism for real-time applications*, https://realtimepartner.com/articles/mutual-exclusion.html, 2016.

[21] S. Raghunathan, "Extending inter-process synchronization with robust mutex and variants in condition wait", in *2008 14th IEEE International Conference on Parallel and Distributed Systems*, 2008, pp. 121–128. DOI: 10.1109/ICPADS.2008.98.

[22] *Introduction of deadlock in operating system*, https://www.geeksforgeeks.org/introduction-of-deadlock-in-operating-system/, 2025.

[23] P. Chahar and S. Dalal, "Deadlock resolution techniques: An overview", *International Journal of Scientific and Research Publications*, vol. 3, no. 7, pp. 1–5, 2013.

[24] P. A. Buhr, "Concurrency errors", in *Understanding Control Flow: Concurrent Programming Using μC++*. Cham: Springer International Publishing, 2016, pp. 395–423. DOI: 10.1007/978-3-319-25703-7_8. [Online]. Available: https://doi.org/10.1007/978-3-319-25703-7_8.

[25] Y. Wang, E. Anceaume, F. Brasileiro, F. Greve, and M. Hurfin, "Solving the group priority inversion problem in a timed asynchronous system", *IEEE Transactions on Computers*, vol. 51, no. 8, pp. 900–915, 2002. DOI: 10.1109/TC.2002.1024738.

[26] M. Michael and M. Scott, "Simple, fast, and practical non-blocking and blocking concurrent queue algorithms", *Proceedings of the Annual ACM Symposium on Principles of Distributed Computing*, Mar. 1996. DOI: 10.1145/248052.248106.

List of Acronyms

IPC	Inter-Process Communication
HRTS	Hard Real-Time System
SRTS	Soft Real-Time System
RTOS	Real-Time Operating System
RTS	Real-Time System
CAS	Compare and Swap

List of Figures

List of Tables

List of Algorithms

A. Appendix