
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 17, 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 17, 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, RTS and low-level programming languages like 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]

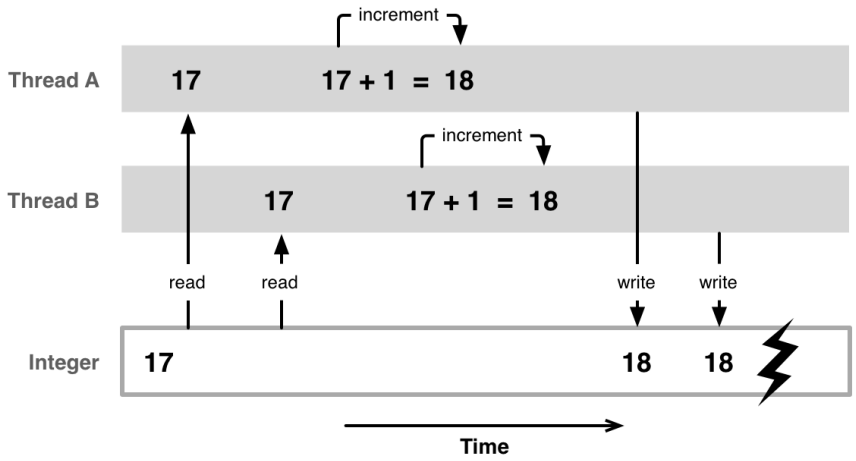
The problem with this is that the system somehow has to manage how the processes access the shared memory segment. This is mostly done by using different kind of synchronization techniques. Without any synchronization mechanism race conditions or inconsistent data can occur. [14], [15]

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.

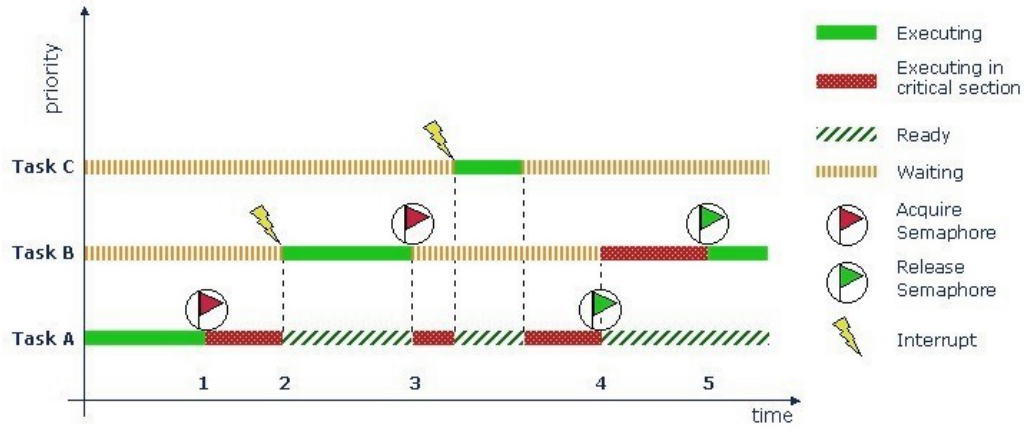


[19]

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.

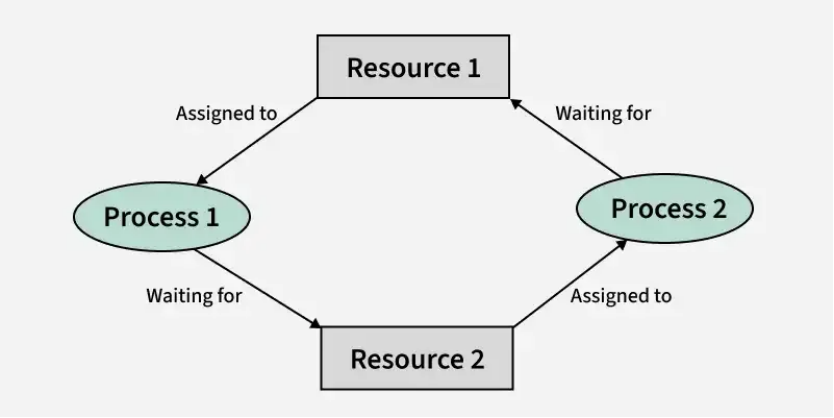
Figure 2.2.: Mutual exclusion between three tasks(processes), which access the same critical section. Multiple processes need to stop working and just wait for other processes to finish their work. See the waiting phase of the processes.



[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. The sequence which process might aquire the lock first to enter the critical section, when multiple processes wait for the access is mostly set by a scheduler. Since wait-free methods explained later in section 2.5 are lock-free, a scheduler is not needed and as a result of that scheduling will not be explained more in detail in this work. [2], [21]

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



[22]

Process Starvation

We mentioned several problems of synchronisation. What if when multiple processes try to enter the shared resource one after the other and one process keeps getting outperformed in aquiring a lock to enter the critical section. This process would wait for a indefinite time and will never enter the shared resource. A process that will never access a shared resource that it trys to enter starves out. This usually happens when a synchronization method allows one or more processes to make progress while ignoring a particular process or processes. This mostly happens in environments where some sort of process prioritization exists and processes are classified into low and high priority processes. When we always have high priority processes available and some low priority processes, it might happen that these 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 too. [23]

Deadlock

Even worse, what if two or more processes already accessed a resource and now each of them wait that the other process releases the lock for the resource each of them aquired? This results in a situation seen in fig. 2.3 where these processes now indefinetlywait for each other and never terminate. So the resources that these waiting processes hold are also never released and are also never available to other processes maybe needed by other systems. As one can see, this a system into a state which would make no progress any further and would also not respond to any command anymore. [22], [24]

For instance a driverless car with a brake-by-wire system, where processes responsible for braking are in a deadlock, could eventually not brake if needed and a fatal car crash would happen.

Priority Inversion

Now lets say we do not have process starvations or deadlocks. What could happen too is that a lower priority process already accessed a shared resource and after that a higher prioritiy process needs to access that specific resource too. If the lower priority process now gets delayed, the higher priority process gets delayed too. This would be called priority inversion now; a low priority process delaying an high prioritiy process. [25]

Increased Response Times

So as we see traditional synchronization is based on mutual exclusion via blocking processes. The result is processes that are waiting, which leads to increased response times of a system. This results in not meeting the timing constraints of a HRTS environment. So synchronization mechanisms are needed that are not based on locks.

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. One way to accomplish lock-freedom for example is the lock-free technique introduced by Michael and Scott, which is also the basis for some other wait-free algorithms. We will look into their algorithm to understand how synchronization without any locks can be achieved .[4], [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 calles tail, which references the end node of the list. The core concept of the algorithm is the enqueue and dequeue functions beginning at 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, see line 8 to 10 of the enque function. Beginning from line 11 to line 23 following happens: The process first checks if the pointer referencing to the next node after the tail node is NULL, see line 15. If it is null it tries to link the new node to the end of the list by using a Compare and Swap (CAS) seen in line 16. 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. The tail itself would be updated in line 24. [26]

So lets say 2 processes p1 and p2 untill line 16 executes one after the other. What could happen now is that if p1 executes line 16 before p2, p2 will fail the CAS from line 16. Now if p1 would not execute further thus not finalizing the enqueue with line 24 and p2 retrys the loop until line 15, the condition in line 15 would not be TRUE anymore for p1 and p1 would execute line 19 and 20 to help p2 to finalize its enqueue so other processes can work further with this algorithm. [26]

The dequeue function works analogously, but instead of adding a node to the end of the list it removes a node from the front of the list. And since another process which could not finish its enqueue would cause confusion for other processes in the dequeue function,

the process which could not finish its enqueue will also be helped in the dequeue function. [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]

We can observe that this approach does not need any locks explained in section 2.3. But to get to the point is that one major problem may occur. If for instance process p1 is trying to enqueue, it can happen that the CAS loop might fail indefinetly if for an indefinite time other processes are always executing line 16 immediately befor p1 could execute line 16. This means that in very high contention scenarios, a process may be delayed indefinetly and starve out. In a broader sense some kind of priority inversion could be also the conclusion if this happens, 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]

2.5. Wait-Free Synchronization

We saw that lock-freedom solves the problem off a system getting into a deadlock. But this is not enough, in a fully automated car for example we do not want any process to not complete its task, since that could mean that some processes that are responsible for braking would not finish their work in a worst case scenario. And such an occasion where the car would need to brake a fatal car crash would be the outcome. So we need a solution where every process finishes their task in a finite number of steps instead of just one process. So something is needed which builds upon such mechanisms and expand them. This is exactly what wait-free synchronization is. It 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. So by definition starvation cannot happen anymore. Also priority inversion is eliminated too, because processes do not have to wait for other processes anymore. This ensures system responsiveness and predictability thus the ability to define strict timing constraints to meet these needed timing constraints, which are essential for HRTS applications. But even wait-free algorithms introduce one problem. Wait-free algorithms are in most cases slower than their lock-free counterpart in execution. A solution to this problem was the fast-path / slow path method by Kogan and Petrank, where operations on a data structure is first done with a lock-free operation, and when failed a slower wait-free operation will substitute the failed lock-free operation [4]. This will be analysed more in detail in chapter 4. In that chapter we will also discuss which data structure is best suited for an wait-free implementation IPC in RTS.

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

```
1: function INITIALIZE(Q : pointer to queue_t)
2:     node = new node()                                  ▷ Allocate a dummy node
3:     node.next.ptr = NULL                               ▷ Make it the only node in the list
4:     Q.Head = node                                      ▷ Both Head and Tail point
5:     Q.Tail = node                                      ▷ to this dummy node
6: end function
7: function ENQUEUE(Q : pointer to queue_t, value : data_type)
8:     node = new node()                                  ▷ Allocate a new node from the free list
9:     node.value = value                                 ▷ Copy enqueue value into node
10:    node.next.ptr = NULL                               ▷ Set next pointer of node to NULL
11:    loop                                               ▷ Keep trying until Enqueue is done
12:        tail = Q.Tail                                  ▷ Read Tail (pointer + count) together
13:        next = tail.ptr.next                           ▷ Read next ptr + count together
14:        if tail == Q.Tail then                         ▷ Are tail & next consistent?
15:            if next.ptr == NULL then                   ▷ Tail is the last node?
16:                if CAS(&tail.ptr.next, next, ⟨node, next.count + 1⟩) then
17:                    break                              ▷ Link the new node; Enqueue is done
18:                end if
19:            else                                       ▷ Tail not pointing to the last node
20:                CAS(&Q.Tail, tail, ⟨next.ptr, tail.count+1⟩)   ▷ Move Tail forward (helping another enqueuer)
21:            end if
22:        end if
23:    end loop
24:    CAS(&Q.Tail, tail, ⟨node, tail.count + 1⟩)         ▷ Final attempt to swing Tail to the inserted node
25: end function
26: function DEQUEUE(Q : pointer to queue_t, pvalue : pointer to data_type)
27:    loop                                               ▷ Keep trying until Dequeue is done
28:        head = Q.Head
29:        tail = Q.Tail
30:        next = head.ptr.next                           ▷ Read head->next
31:        if head == Q.Head then                         ▷ Still consistent?
32:            if head.ptr == tail.ptr then               ▷ Empty or Tail behind?
33:                if next.ptr == NULL then               ▷ Queue is empty
34:                    return FALSE
35:                else                                   ▷ Tail is behind, help move it
36:                    CAS(&Q.Tail, tail, ⟨next.ptr, tail.count + 1⟩)
37:                end if
38:            else                                       ▷ No need to adjust Tail
39:                *pvalue = next.ptr.value               ▷ Read value before CAS
40:                if CAS(&Q.Head, head, ⟨next.ptr, head.count + 1⟩) then
41:                    break                              ▷ Dequeue is done
42:                end if
43:            end if
44:        end if
45:    end loop
46:    free(head.ptr)                                     ▷ Safe to free old dummy node
47:    return TRUE
48: end function
```

2.6. Rust Programming Language

The question now is which programming language suits best for this kind of research. Since we need fast communication between the processes to meet all HRTS timing constraints, one could think off the C programming language. C provides low-hardware

control and therefore also allows the implementation of fast low-latency communication. What is also important and necessary for a RTS is, that C does not have an automatic garbage collecter, which gets active and stops all processes from working to clean up allocated but no longer used memory space. Because of that all Real-Time Operating System (RTOS) are written in C. The main problem with C is that it does not provide any kind of memory safety, since C implements memory operations that are prone to buffer overflows or control-flow attacks. In the industry around 70% of vulnarabilities happen because of memory safety issues. If the real-time application would run on an isolated system with no internet connection, this would not be a problem. But in modern automation, where systems need to be connected to the internet for data exchange, such systems would be prone to security attacks. RTS nowadays is an integral part in various connected devices, including critical fields such as health or transportation. Conclusively it is extremely important that the security of such devices is guaranteed. With the rust programming language the problems of memory safety features are gone. The difference to C is that it can be as fast as C with the possibility to support low-level control and high-level programming features, while providing memory safety features in a real-time setting. The memory safety aspect is achieved by an ownership concept that controls how memory is handled in programs. This is strictly checked and therefore the executable program has guaranteed memory safety. In the model every value has a single owner represented by a variable. The owner is in charge of the lifetime and deallocation of that value. Rust will automatically free the memory associated with that value, when the owner goes out of scope. This behaviour is automatically done by using the memory reference feature provided by rust. Creating such references is also called borrowing. This allows the usage of these values without transferring the ownership. These references have its own lifetime, which can be explicitly defined by the programmer or implicitly inferred by rusts compiler. This ensures that the references are valid and do not exist longer as needed. Hence this also can play a role in lock-freedom, which is needed for wait-free synchronization, since shared resources can be shared with this ownership concept. Additionally rust is a type-safe language, which can be helpfull during implementation to avoid bugs and errors, since this also needs to be avoided in a RTS. As seen rust is a good choice for implementing wait-free synchronization mechanisms for IPC in RTS. [9], [27].

Further mechanisms on how with rust different kind of common memory safety issues are solved will not be discussed in detail, as that would go beyond the scope of this thesis. It is only important to know the basics on how rust is a type-safe and memory-safe programming language, to understand why rust is used for this work.

3. Related Work

The early foundations regarding wait-freedom was done indirectly by Leslie Lamport in 1977 and 1983 [28], [29]. Indirectly, because this laid the foundation for lock-freedom, which was later then extended to wait-freedom by Maurice Herlihy [1]. In 1977 he showed how one writer and multiple readers can share data without the need of locks, eliminating writer delays due to reader interference. It works by using atomic read and write operations byte wise. The writer will atomically write a value to a memory location from right to left, while the reader uses the same memory location to read the value from left to right. So even if the writer is still in the process of writing, the reader will not block the writer while reading (and vice versa), yet always sees a correct snapshot of the data. To prevent a reader process from saving inconsistent data (for example while the writer process writes), the writer process will tag each update by incrementing a start counter before an end counter after writing. Readers take the start counter, read the data, and compare if the start counter matches the end counter. If they do not match it means the data is inconsistent and the reader will retry reading the data until it gets a consistent value. To understand this better imagine a date with the format DD/MM/YYYY, where every digit is saved as one byte from right to left and read from left to write. This solves the problem of contention between readers and a single writer. If multiple writers are involved, the writers still have to be mutually exclusive, which means that they have to use locks to block each other to prevent inconsistency. [28]

In 1983 Leslie Lamport then gave a way to write and prove the correctness of any concurrent module in a simple and modular way independent of the used data structure, which he refers to as modules. He defined safety and liveness conditions, where safety means that for instance a queue drops or reorders items and liveness means that that queue when its nonempty an item eventually is dequeued. A module will get state functions, which are the abstract variables needed to define that module, initial conditions, which are simple predicates on those state functions and properties which are a mix of safety and liveness clauses. He also defines the usage of action sets and environment constraints, which seperate the module action from the environments (the program where the data structure runs in). An example would be an First In First Out (FIFO) queue, where the queue, input variable, return variable would be the state functions and the enqueue and dequeue operations would be the actions. The environment constraints would be the preconditions for the enqueue and dequeue operations, which are that the queue is not full or empty respectively. As safety condition one could define that the enqueue operation only adds when control is inside and as a liveness condition that the dequeue operation will eventually return a value. This concept is important, because it

allows to just re-speciffy the data structures interface instead of the whole environment it runs in. Also he proves these concepts, so there is a uniform recipe for oing from spec to code to proof for any concurrent data structure. [29]

These two insights are also the basis for the wait-free data structure implementation by Maurice Herlihy and his proof that the atomic operations of Lamport are linearizable, which is a correctness condition for concurrent data structures that guarantees every operation performed appears to take effect instantaneously at some point between the call and return of the operation [1], [30]. So while Lamport gave a way how to prove the safety and liveness of a data structure [29], Herlihy and Wing specified which safety and liveness conditions are necessary to prove linearizability [30].

Herlihy than extended Lamports work to wait-free data structures by using the atomic read and write operations of Lamport and the linearizability proof. He showed that every sequentual data structure can be made wait-free, which is the basis for wait-free algorithms available. His work (or work that builds upon his work) shows up conceptually in all of the wait-free algorithms [4], [31]–[35]. [1]

4. Choosing the Optimal Wait-Free Data Structure and algorithm

4.1. Optimal Wait-Free Data Structure

An important question is what data structure to use for the implementation of a wait-free synchronisation technique for IPC. M. Herlihy showed that every data structure can be made wait-free [1]. So it is important to choose the optimal data structure for our use. Considering that the reason of this work is to optimize modern manufacturing and automation, some form of correct data flow order is as well necessary for correct work flow for instance in an modern manufacturing line or more critical in a driverless car. From that point on we can think of an already natural fit like FIFO queues. Natural because in such queues a producer process can enqueue messages and the consumer process can dequeue messages sequentially. This models real-world data flows (sensor readings, commands, network packets), which are inherently sequential. Consequently with such queues the order of the data flow is preversed without the need of implementing additional functionalities. In contrast, data structures like stacks, sets, or maps do not maintain this kind of arrival order and moreover add semantics like Last In First Out (LIFO) order or key-value pairs, which are in most cases not desired or even unnecessary. This would bring in the need of additional functions to just get rid of undesired side effects. Furthermore in a queue we only have two operations, an enqueue and an dequeue operation. All the other data structures introduce more operations and therefore more complexity. The less operations we have, the less complex the implementation will be. Because of these advantages and also because of the fact that in most publications in the wait-free domain queues are beeing used, limiting this thesis to queues only is plausible. [36]

4.2. Wait-Free Algorithms

As we know now on which data structure we should focus an important question is which algorithms to use. There are different cases where different kind of algorithms make more sense. We will decide this contention dependant. There exist algorithms for Single Producer Single Consumer (SPSC), Multi Producer Single Consumer (MPSC), Single Producer Multi Consumer (SPMC) and Multi Producer Multi Consumer (MPMC) queues. Since all of them have different complexity in runtime and space, it is important to choose the right one for the right use case to save resources and have faster execution

times to meet the timing constraints of HRTS. In modern manufacturing and automation devices are used which can run multiple applications on the single device. This could mean that every application running on that device could be a producer and a consumer to each other and also maybe some single application of all applications running on that device produces data for just a single other consuming application. And maybe some single application is a producer for multiple consuming applications and multiple applications are producers of a single consuming application. So it can be that all cases can occur in just one design. This means that we have to consider all the different cases of contention. In the following sections we will discuss the different cases and their algorithms. We will also implement them and test them performance wise before we will implement a final solution which uses all the algorithms automatically for the right use of contention between applications. And even if only one pattern occurs for different devices it makes sense to implement all of them, so that in an automated system with multiple device only one implementated solution is needed to be installed.

4.2.1. Single Producer and Single Consumer

This is the most simple form of IPC. In SPSC there is no contention from other processes, because we only have one producer and one consumer. The producer can enqueue processes without any other process interfering. The same goes for the consumer. Here only lockfreedom between writer and reader process is sufficient to achieve wait-freedom.

4.2.2. Multiple Producer and Single Consumer

4.2.3. Single Producer and Multiple Consumer

4.2.4. Multiple Producer and Multiple Consumer

5. Implementation

6. Results

7. 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. 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.

[24] 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.

[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.

[27] I. Culic, A. Vochescu, and A. Radovici, "A low-latency optimization of a rust-based secure operating system for embedded devices", *Sensors*, vol. 22, no. 22, p. 8700, 2022.

[28] L. Lamport, "Concurrent reading and writing", *Commun. ACM*, vol. 20, no. 11, pp. 806–811, Nov. 1977. DOI: 10.1145/359863.359878. [Online]. Available: https://doi.org/10.1145/359863.359878.

[29] L. Lamport, "Specifying concurrent program modules", *ACM Trans. Program. Lang. Syst.*, vol. 5, no. 2, pp. 190–222, Apr. 1983. DOI: 10.1145/69624.357207. [Online]. Available: https://doi.org/10.1145/69624.357207.

[30] M. P. Herlihy and J. M. Wing, "Linearizability: A correctness condition for concurrent objects", *ACM Trans. Program. Lang. Syst.*, vol. 12, no. 3, pp. 463–492, Jul. 1990. DOI: 10.1145/78969.78972. [Online]. Available: https://doi.org/10.1145/78969.78972.

[31] H. Naderibeni and E. Ruppert, *A wait-free queue with polylogarithmic step complexity*, 2023. arXiv: 2305.07229 [cs.DC]. [Online]. Available: https://arxiv.org/abs/2305.07229.

[32] R. Nikolaev and B. Ravindran, "Wcq: A fast wait-free queue with bounded memory usage", in *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '22, Seoul, Republic of Korea: Association for Computing Machinery, 2022, pp. 461–462. DOI: 10.1145/3503221.3508440. [Online]. Available: https://doi.org/10.1145/3503221.3508440.

[33] P. Ramalhete and A. Correia, "Poster: A wait-free queue with wait-free memory reclamation", in *Proceedings of the 22nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '17, Austin, Texas, USA: Association for Computing Machinery, 2017, pp. 453–454. DOI: 10.1145/3018743.3019022. [Online]. Available: https://doi.org/10.1145/3018743.3019022.

[34] A. Kogan and E. Petrank, "Wait-free queues with multiple enqueuers and dequeuers", *SIGPLAN Not.*, vol. 46, no. 8, pp. 223–234, Feb. 2011. DOI: 10.1145/2038037.1941585. [Online]. Available: https://doi.org/10.1145/2038037.1941585.

[35] C. Yang and J. Mellor-Crummey, "A wait-free queue as fast as fetch-and-add", *SIGPLAN Not.*, vol. 51, no. 8, Feb. 2016. DOI: 10.1145/3016078.2851168. [Online]. Available: https://doi.org/10.1145/3016078.2851168.

[36] D. Adas and R. Friedman, "Jiffy: A fast, memory efficient, wait-free multi-producers single-consumer queue", *CoRR*, vol. abs/2010.14189, 2020. arXiv: 2010.14189. [Online]. Available: https://arxiv.org/abs/2010.14189.

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

FIFO First In First Out

LIFO Last In First Out

MPMC Multi Producer Multi Consumer

MPSC Multi Producer Single Consumer

SPMC Single Producer Multi Consumer

SPSC Single Producer Single Consumer

List of Figures

List of Tables

List of Algorithms

A. Appendix