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

Master's Thesis

Wait-free synchronisation 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	July 14, 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 synchronisation for inter-process communication in real-time systems*
German title	*Wait-free Synchronisation 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, July 14, 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 wait-free data structures 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, a fundamental overview of the key topics relevant to this research will be provided. This includes an introduction to RTS, Inter-Process Communication (IPC), and synchronization techniques, with a particular focus on wait-free synchronization. Additionally, the Rust programming language will be examined, as it serves as the primary development environment for this study. Furthermore, existing synchronization methods in RTS will be explored 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 like a shared memory region, which will be the method used and explained later in this thesis. 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]

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]

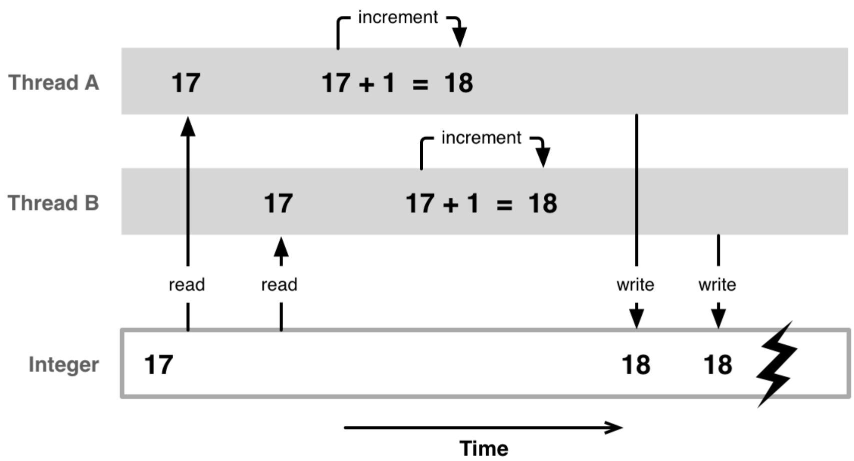
The problem with this is that the system somehow has to manage how the processes access the shared memory segment. This is mostly done by using different kind of synchronization techniques. Without any synchronization mechanism race conditions or inconsistent data can occur. [14], [15]

2.3. Synchronization

As observed, 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 one process p2 increments that number, the end result should be 19. But what could happen is that p1 could read the value 17 before p2 increments it and then before p1 increments that value p2 could also read 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 via threads simultaneously within that process. Another difference that will later be important in this thesis is that processes have their own private memory space, while threads share the memory space of the process they are part of. So naturally a process can not access the memory of another process. Regardless the following concept in fig. 2.1 can be used for processes too. [18]

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



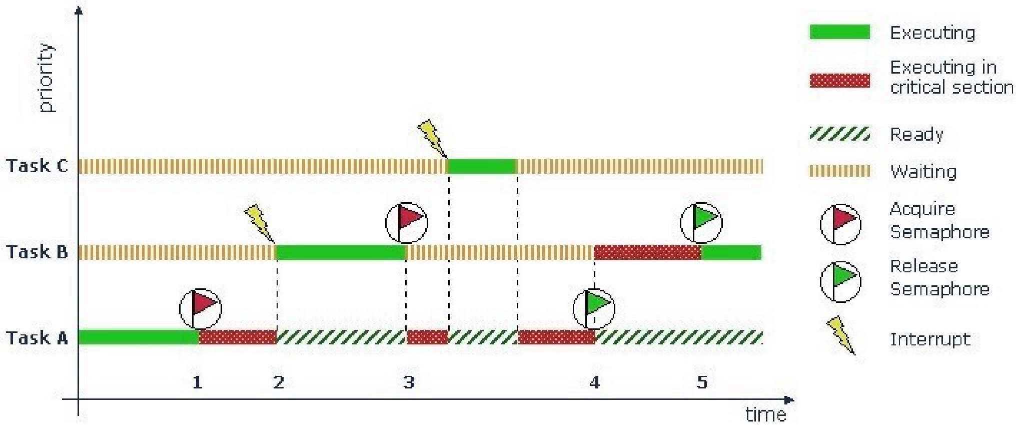
[19]

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.

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-

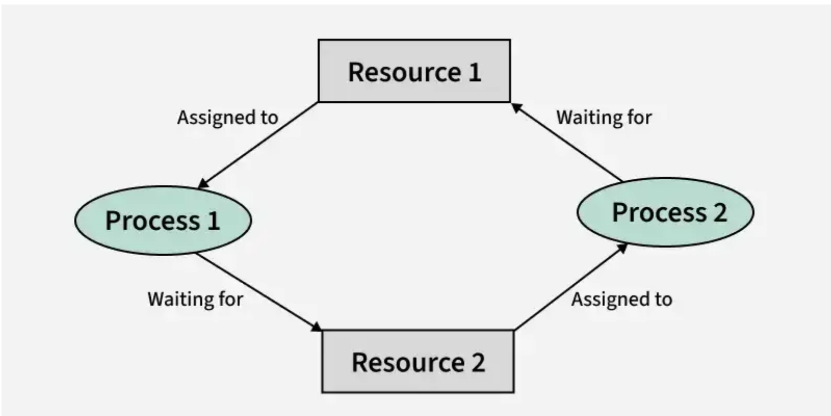
Figure 2.2.: Mutual exclusion between three tasks(processes), which access the same critical section. Multiple processes need to stop working and just wait for other processes to finish their work. See the waiting phase of the processes.



[20]

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

Another problem would be, 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 there are always 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 no process starvations or deadlocks happen. 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

As demonstrated 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.

2.4. Lock-Free Synchronization

Therefore synchronization techniques are required that do not block processes with any kind of locking mechanism. 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.

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]

Initialization starts at line 1 in algorithm 1, which 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. It can be observed that this approach does not need any locks explained in section 2.3. Still this approach has one major problem. 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 HRTS this could lead to violating timing constraints, since the process would not finish his task in the defined timing window, which is unacceptable. This is why a slightly different approach which guerantees that every process will complete its operation in a finite number of steps is in need. [26]

Algorithm 1 Michael and Scott's Lock-Free Queue

```
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, ⟨node, next.count + 1⟩) 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, ⟨node, tail.count + 1⟩)           ▷ 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, ⟨next.ptr, tail.count + 1⟩)
                end if
            else                                         ▷ No need to adjust Tail
                *pvalue = next.ptr.value                 ▷ Read value before CAS
                if CAS(&Q.Head, head, ⟨next.ptr, head.count + 1⟩) 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
```

[26]

While Michael and Scott's algorithm relies on CAS primitives, other atomic primitives provide alternative approaches that other algorithms shown later in this thesis use. A overview on the atomic primitives that are used in this thesis context is given in the following subsections.

2.4.2. Atomic Primitives

Load Linked/Store Conditional (LL/SC)

An operation available on ARM, MIPS and Alpha architecutures usually implied with a validate-link (VL) instruction.

- LL (R) returns value of register r
- "textSC (R, v) changes the value in register R to v and returns true, if and only if no other process performed a successful SC since the most recent call of LL of the current process. So SC fails if the value of the register has changed since it has been read" [27]
- "VL (R) returns true if no other process performed a successful SC on register R, which allows to test a register value without changing it" [27]

[27]

Compare and Swap (CAS)

It is already explained how CAS works in section 2.4.1. Furthermore to the explanation in section 2.4.1 CAS is an atomic primitive that is supported on "intel x386, x64 and most general purpose architectures with operands that are restricted to pointer size" [27].

- "CAS(R,e,n) returns true and sets the value of R to n if the value in R is e. Otherwise, it returns false." [27]

The problem with CAS furthermore than the issue explained earlier in section 2.4.1 is that it can lead to the ABA problem, which can also occur in wait-free algorithms:

- Process 1 reads value A from a shared variable.
- Process 2 changes the value to B and then back to A.
- Process 1's CAS operation succeeds, because it compares the value A it read earlier with the current value A, even though the value was changed in between.

This is a fundemental limitation of CAS. One solution would be to replace CAS with LL/SC, but that is not possible on x86 processors. So other solutions are needed that are discussed in chapter 6. [27]

Fetch and Add (FAA)

This primitive is used to increment "the value of a variable by a given offset and [return] the result. This instruction always succeeds." [27]

Fetch and Store (FAS)

This atomically stores a value into a variable and returns the previous value. This is similar to CAS, but it does not require a comparison and a retry loop. This is faster then CAS, if conditions before updating do not need to be checked. [28]

2.5. Wait-Free Synchronization

Lock-freedom solves the problem off a system getting into a deadlock. But this is not enough, in a fully automated car for example it is undesirable that any process does 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. Consequently a solution is necessary 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 vital 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 bounded 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 5.

2.6. Rust Programming Language

The question now is which programming language suits best for this kind of research. Since a fast communication between the processes is compulsory to meet all HRTS timing constraints, the C programming language would be a good choice. 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], [29].

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 [30], [31]. While these works did not directly address or formally define wait-freedom, they laid the groundwork for lock-freedom, which Maurice Herlihy later extended to wait-freedom [1].
In 1977, Lamport 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 byte-wise read and write operations. 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) and still see 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 written 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 (So with multiple writers this algorithm cannot be lock-free anymore). [30]

In 1983 Leslie Lamport then gave a formal method 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. These modules consist of three components:

- state functions, which are abstract variables describing the module's state.
- initial conditions, which are predicates on the state functions.
- properties, which are a mix of safety and liveness requirements.

Safety requirements define what must never happen (e.g., a queue must never drop an element), while liveness requirements define what must eventually happen (e.g., a non-empty queue must eventually allow dequeueing). He also defines the usage of action sets and environment constraints, which seperate the module action from the environments (e.g., the program where the data structure runs in). For example, a First In First Out (FIFO) queue specification would include:

- State Functions: queue contents, operation parameters, return values.
- Initial Conditions: queue starts empty.
- Safety Condition: queue maintains FIFO ordering.
- Liveness Condition: dequeue operation will eventually return a value.

This systematic methodology to prove the correctness of concurrent data structures laid the necessary groundwork for later developments too. [31]

Building on methodologys to prove concurrent data structure correctness Herlihy and Wing provided linearizability as a correctness condition for concurrent objects, which is a guarantee that every operation performed appears to take effect instantaneously at some point between the call and return of the operation [32]. This correctness condition was then used to formalize wait-freedom in 1991 [1]. In the latter work Herlihy proved that any sequential data structure can be transformed into a wait-free concurrent data structure [1]. A wait-free data structure must satisfy following three constraints:

- Linearizability: operations take effect instantaneously at some point between the call and return of the operation.
- Bounded steps: operations end in a finite number of steps.
- Independence: operations finish regardless of other processes' exection (for later understrandin: a process waiting for another process for a maximum number of time and then returning an error if that time is exceeded would still be considered wait-free, since it will finish regardless of the other process).

From this perspective the algorithm provided by Lamport in 1977 is already wait-free, even though the term was not yet defined at that time. [1], [32]

Herlihys universal construction and principles (or work that builds upon his work) appear conceptually throughout all of these wait-free algorithms [4], [28], [33]–[50]. [1]

Kogan and Petrank later invented a method called fast-path slow-path, were first usually a lock-free method (the fast-path) is used to try to complete an operation, since lock-free algorithms are usually faster than wait-free algorithms. These lock-free paths are bounded by a maximum number of steps and if the operation does not complete within that bound the algorithm tries to complete the operation using a wait-free method (the slow-path). So in cases where the fast-path succeeds often without switching to the slow path, the algorithm in general completes in a shorter time then a pure wait-free algorithm. This method is used by two algorithms with one of them having a great performance advantage, which will be demonstrated later in this thesis [38], [40]. [4]

4. Methodology

To achieve the goal that is defined in section 1.2, first all wait-free data structures that could be used for IPC through shared memory in HRTS need to be found. To do this a method was used that is more known from mapping studys or literature reviews. Multiple python scripts were implemented to do this [51]. A web scraper script was written to scrape over google scholar with the following querys:

- "wait-free queue"
- "wait-free" ("mpmc" OR "multi-producer multi-consumer" OR "multi-writer multi-reader" OR "many-to-many") "queue"
- "wait-free" ("mpsc" OR "multi-producer single-consumer" OR "single-writer multi-reader" OR "many-to-one") "queue"
- "wait-free" ("spsc" OR "single-producer single-consumer" OR "single-writer single-reader" OR "one-to-one") "queue"
- "wait-free" ("spmc" OR "single-producer multi-consumer" OR "multi-writer single-reader" OR "one-to-many") "queue"

In google scholar a whitespace is considered as an AND. The rest is interpreted as read. With this approach i got a list of 1324 papers. The papers were then written into a csv file split into query, rank (number of paper), title, year, authors, venue, citations, abstract snippet, full_abstract and url with ";" as an delimiter. To extract all of this the information in google scholar was extracted and then for the full abstract the scraper went onto the source url and extracted the abstract there. If the url was a direct pdf link, a pdf reader was used to find the abstract. If an abstract was not extractable (some source site which was not considered or other problems), "ABSTRACT_NOT_FOUND" was written instead or if the paper was accessible the full paper was written instead into that cell. Because a lot of scientific web pages will put ceptchas if continuously requests are made undetected_chromedriver was importet and used as the web driver. It is an enhanced version of chromedriver which bypasses anti bot detections. After that I also implemented a regex analyzer to analyze the abstracts of the found papers on the words "lockfree", "waitfree", and "obstructionfree" and also again without a hyphon inbetween these words. If these keywords were not found in the abstract or the abstract contained the flag "ABSTRACT_NOT_FOUND" the paper was removed from the csv. This left 475 papers that contained at least one of these words in their abstract. After that the duplicates were removed by the algorithm, which left 325 papers. The duplicates were removed by checking the url of the paper. Now what was left had to be manually analyzed to see if the paper was relevant for the topic. Since libre office and microsoft excel has a limit of 32.767 charachters per cell a abstract splitter had to be build to split the abstract into multiple cells. Analyzing was done by reading the paper and checking if the paper was developing an wait-free fifo queue. While dooing that also backward and forward search was done to even find more papers. Papers where only included, if and only if:

- All three of Herlihys contraints were met, that we listed in chapter 3.
- The algorithms could be implemented on x86 architecture using rust, since this is the architecture and programming languge that is used in the implementation of this thesis. The reason x86 is used, is because that is the architecture available and it is also the architecture that is used in most modern computers. For example Khanchandani and Wattenhofer [52] created a queue using theoretical atomic primitives called half-increment and half-max. These primitives do not exist on real hardware. The same goes for Bédin et al. [53] who created a queue using theoretical atomic primitives called memory-to-memory.
- The algorithm runs in an acceptable time complexity to even benchmark it. For example the algorithm by Johnen et al. [54] and Naderibeni and Ruppert [55] were running so slow that they were not been considered to be benchmarked, since they take too long to run and wait for results. Both use a really complex to implement tree structure that is creating an extreme overhead for just producing and consuming items.

In the end 17 papers were included from the 325 papers and 3 more paper were included by backward and forward search of the papers that were included. The papers were then split into 4 contention categories:

- MPMC queues [4], [33]–[40]
- MPSC queues [28], [41]–[44]
- SPMC queues [45]
- SPSC queues [31], [46]–[50]

Now the wait-free fifo queues had to be compared performance wise. Some papers were just improvements of other papers so only the improved version was used for the comparison. Some other papers showed multiple ways of implementing a wait-free fifo datastructure. In the end 6 MPMC queues, 4 MPSC queues, 1 SPMC queue and 11 SPSC queues were included into this work.

5. Analyzing existing Wait-Free Data Structures and Algorithms

5.1. 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 sequential 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. Hence 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 only two operations exist, an enqueue and an dequeue operation. All the other data structures introduce more operations and therefore more complexity and therefore more performance overhead. The less operations exist, 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 reasonable. [43]

5.2. Wait-Free Algorithms

With the appropriate data structure established, an important consideration is the selection of suitable algorithms. In chapter 4 4 different contention categories are defined. Which kind of algorithm going to be used will be decided contention based. Since all of them have different complexity in runtime, it is important to choose the right contention category 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 a single device. This could mean that every application running on one device could be a producer and a consumer to each other (MPMC) and also maybe some single application of all applications running on one device produces data for just a single other consuming application (SPSC). And maybe some single application is a producer for multiple consuming applications (SPMC) and multiple applications are producers of a single consuming application (MPSC). So it can be that all cases can occur in just one device. This means that all the different cases of contention have to be considered. In the following the different cases and their algorithms will be discussed. Moreover they will be implemented and their performance tested via a benchmarked (how fast an algorithm can produce and consume items concurrently). Subsequently from each category the best algorithm will be chosen and their performance will be compared with each other to Identify, if 4 different categories are necessary. The reason for that is, that for instance the best performed MPMC algorithm could outperform all other algorithms even for their contention category, since a MPMC approach can cover all contention cases. The goal with this approach is to have as little overhead as possible, since an algorithm explicitly implemented for a MPMC use case could have extreme overhead for a SPSC case. The implementation will be discussed in chapter 6 and the results of the performance readings will be discussed in chapter 7. The following subsections will give an overview of the different contention categories and their algorithms found. The subsections will also shortly describe how the enqueue and dequeue in these algorithms work. Given that all algorithms found are about inter-thread communication and not IPC, the following explanation of the specific algorithms will include the terminology of threads to be precise about the papers. In chapter 6 it will be elaborated how these thread based algorithms are adapted to IPC in Rust. Other minor Rust-specific deviations from the following algorithms (different types required by Rust's safety model, additional memory fences, etc.) can be seen in the GitHub repository accompanying this thesis [51]. These are not detailed here as such explanations would provide limited value to the topic of this work, and while a comprehensive analysis would be relevant, it would require more extensive exploration than is feasible within the scope of this work.

5.2.1. Single Producer Single Consumer (SPSC)

This is the most simple form of IPC. In SPSC there is nearly no contention from other processes, because only one producer and one consumer is working. The only contention is between the consumer and producer. This leads to the producer and consumer finish in a bounded number of steps withoput special synchronisation techniques like helping or atomic primitives. The only concern is that the data the consumer reads is consistent. Different approaches were testet in different paper, that will be seen here. Since Batched Lamport Queue (BLQ), Lazy Lamport Queue (LLQ), Batched Improved FastForward Queue (BIFFQ) and Improved FastForward Queue (IFFQ) are from the same paper, explanations and variables are shared between these algorithms to avoid redundancy:

Lamport's Circular Buffer Queue

Uses a circular array with two shared indices for synchronization, based on the algorithm originally proposed by Leslie Lamport in 1983 [31] and shown here in algorithm 2 following

Maffione et al.'s version [49]. The producer first checks if the queue is full (reached capacity N) in line 2, which requires reading the consumer's `read` index. If the queue is not full, it writes the input data to the slot at position `write & mask` in line 5, where the bitwise AND operation wraps the index around when it reaches the array end. The producer then increments `write` to signal that the written data is available in line 7. The consumer mirrors this behavior by checking if the queue is empty in line 12, which requires reading the producer's `write` index. If the queue is not empty, the consumer reads data from the slot at position `read & mask` in line 16, using the same modulo arithmetic through bitwise AND, and incrementing its `read` index to signal that the slot is available in line 17. This wraparound behavior creates the circular buffer structure, allowing the fixed-size array to be reused continuously. Unfortunately each operation requires accessing both shared indices plus the data slot, causing up to three cache misses per item when the queue moves between nearly empty and nearly full states. According to Drepper, cache misses occur when requested data is not in the local CPU core's cache and must be fetched from another core's cache, with the cache coherence protocol ensuring memory consistency across all cores [56]. Drepper showed that performance can degrade by 390%, 734%, and 1,147% for 2, 3, and 4 threads respectively. This happens because cache lines, the 64-byte blocks (on x86 architectures) that move between CPU caches, ping-pong between the producer's and consumer's cores as they take turns accessing the same memory locations [56].

Algorithm 2 Lamports Queue [49]

```
 1: function LQ_ENQUEUE(q, e)
 2:     if q.write − q.read = N then                      ▷ Check if full
 3:         return −1                                     ▷ No space
 4:     end if
 5:     q.slots[q.write ∧ q.mask] ← e
 6:     store_release_barrier()
 7:     q.write ← q.write + 1
 8:     return 0
 9: end function
10:
11: function LQ_DEQUEUE(q)
12:     if q.read = q.write then                          ▷ Check if empty
13:         return NULL_ELEM                              ▷ Queue empty
14:     end if
15:     load_acquire_barrier()
16:     e ← q.slots[q.read ∧ q.mask]
17:     q.read ← q.read + 1
18:     return e
19: end function
```

[49]

Lazy Lamport Queue

Reduces the described cache misses by postponing index reads until necessary, as shown in algorithm 3. Additionally to Lamports original enqueue function the producer maintains a local `read_shadow` copy and only updates it when running out of known free slots in lines

2 - 6. Similarly, the consumer uses `write_shadow` to avoid repeatedly checking for new items. Additionally, LLQ keeps K slots (where K is slots per cache line) permanently empty, preventing producer and consumer from touching the same cache line when the queue is full. This works well when one thread is faster than the other, reducing worst-case misses from 3 to about 2 per item. [49]

Algorithm 3 Lazy Lamport Queue Operations [49]

```
1:  function LLQ_ENQUEUE(q, e)
2:      if q.write − q.read_shadow = N − K then                ▷ Lazy load check
3:          q.read_shadow ← q.read                             ▷ Update shadow
4:          if q.write − q.read_shadow = N − K then
5:              return −1                                      ▷ No space
6:          end if
7:      end if
8:      q.slots[q.write ∧ q.mask] ← e
9:      store_release_barrier()
10:     q.write ← q.write + 1
11:     return 0
12: end function
13:
14: function LLQ_DEQUEUE(q)
15:     if q.read = q.write_shadow then                        ▷ Lazy load check
16:         q.write_shadow ← q.write                           ▷ Update shadow
17:         if q.read = q.write_shadow then
18:             return NULL_ELEM
19:         end if
20:     end if
21:     load_acquire_barrier()
22:     e ← q.slots[q.read ∧ q.mask]
23:     q.read ← q.read + 1
24:     return e
25: end function
```

Batched Lamport Queue

Extends LLQ with explicit batching to further reduce synchronization costs, as detailed in algorithm 4. The producer accumulates items using private `write_priv` in line 11, filling slots without updating the shared `write` index. Only the function `blq_enqueue_publish` in lines 15 - 18 makes the batch visible by advancing `write` in line 17. The consumer works symmetrically, using `read_priv` in line 32 for local progress before updating `read` in line 40. With typical batch sizes like $B = 32$, synchronization overhead is amortized across operations, reducing cache misses. However, the application using this queue design must explicitly call the publish functions even with partial batches to avoid unbounded latency, because items remain invisible to the consumer until published. This design particularly benefits applications that naturally process data in batches, such as network packet processing, where batch boundaries are well-defined. [49]

Algorithm 4 Batched Lamport Queue Operations [49]

```
function BLQ_ENQUEUE_SPACE(q, needed)
    space ← N − K − (q.write_priv − q.read_shadow)
    if space < needed then
        q.read_shadow ← q.read                          ▷ Update shadow
        space ← N − K − (q.write_priv − q.read_shadow)
    end if
    return space
end function

function BLQ_ENQUEUE_LOCAL(q, e)
    q.slots[q.write_priv ∧ q.mask] ← e
    q.write_priv ← q.write_priv + 1
end function

function BLQ_ENQUEUE_PUBLISH(q)
    store_release_barrier()
    q.write ← q.write_priv
end function

function BLQ_DEQUEUE_SPACE(q)
    available ← q.write_shadow − q.read_priv
    if available = 0 then
        q.write_shadow ← q.write                        ▷ Update shadow
        available ← q.write_shadow − q.read_priv
    end if
    return available
end function

function BLQ_DEQUEUE_LOCAL(q)
    load_acquire_barrier()
    e ← q.slots[q.read_priv ∧ q.mask]
    q.read_priv ← q.read_priv + 1
    return e
end function

function BLQ_DEQUEUE_PUBLISH(q)
    q.read ← q.read_priv
end function
```

FastForward Queue

Synchronization by embedding control information directly within the data slots, eliminating separate shared indices shown in algorithm 5. Unlike Lamport's queue which requires checking both `head` and `tail` indices, FFQs producer simply examines if the next slot contains `NULL` in line 2 before writing. When the slot is empty (`NULL`), the producer writes the data directly and advances its private `head` index in lines 5 and 6. The consumer follows a similar pattern by reading from the current slot position in line 11 and then checks in line 12 data is present (non-`NULL`). If not NULL it retrieves the value and writes `NULL` to mark the slot empty in line 15. Then it advances its private `tail` index in the line after that. This couples synchronization control and the actual data, reducing shared memory accesses from three (`head`, `tail`, `buffer`) to just one (`buffer` slot). Each thread maintains its own private index that never needs synchronization. The producer tracks where to write next through `head`, while the consumer tracks where to read next through `tail`. The `NULL` value serves dual purpose as both an empty indicator and the synchronization mechanism. While this approach reduces memory barriers and cache misses significantly, it still has the ping-pong effect when the queue has few elements, causing the producer and consumer to operate on the same cache line. [50]

Algorithm 5 FastForward Queue Operations [50]

```
function FFQ_ENQUEUE(q, data)
    if q.buffer[q.head] ≠ NULL then
        return EWOULDBLOCK
    end if
    q.buffer[q.head] ← data
    q.head ← NEXT(q.head)
    return 0
end function

function FFQ_DEQUEUE(q)
    data ← q.buffer[q.tail]
    if data = NULL then
        return EWOULDBLOCK
    end if
    q.buffer[q.tail] ← NULL
    q.tail ← NEXT(q.tail)
    return data
end function
```

Improved FastForward Queue

Prevents cache conflicts through spatial separation using a look-ahead mechanism shown in algorithm 6. The producer checks if a slot H positions ahead (4 cache lines ahead) is empty before proceeding in line 4, ensuring it works far ahead of the consumer. This check happens only once every H items when `write` reaches `limit` in line 2. The consumer delays clearing slots through the function `iffq_dequeue_publish` seen in lines 23 - 28, maintaining separation between producer and consumer regions. With $2H$ permanently unused slots as a buffer zone, producer and consumer operate on different cache lines, reducing cache misses even more. [49]

Batched Improved FastForward Queue

Addresses IFFQs weakness when the queue is nearly empty by adding producer-side buffering, as shown in algorithm 7. Items first accumulate in a thread-local buffer seen in line 10, then the function `biffq_enqueue_publish` beginning at line 15 writes them to the queue in a rapid burst in lines 15 - 18. Also like in BLQ the application using this queue must call this function explicitly to avoid deadlocks. This behavior creates an intended race condition, which is beneficial if all writes complete before the consumer notices. The cache line stays with the producer to avoid ping-pong effects. The consumer side remains unchanged from IFFQ. While theoretical worst-case behavior is similar to IFFQ, practical measurements show

Algorithm 6 Improved FastForward Queue Operations [49]

```
function IFFQ_ENQUEUE(q, e)
    if q.write = q.limit then                                   ▷ Check limit
        next_limit ← q.limit + H
        if q.slots[next_limit ∧ q.mask] ≠ NULL_ELEM then
            return −1                                           ▷ No space
        end if
        q.limit ← next_limit                                    ▷ Free partition
    end if
    q.slots[q.write ∧ q.mask] ← e
    q.write ← q.write + 1
    return 0
end function

function IFFQ_DEQUEUE_LOCAL(q)
    e ← q.slots[q.read ∧ q.mask]
    if e = NULL_ELEM then
        return NULL_ELEM
    end if
    q.read ← q.read + 1
    return e
end function

function IFFQ_DEQUEUE_PUBLISH(q)
    while q.clear ≠ next_clear(q.read) do
        q.slots[q.clear ∧ q.mask] ← NULL_ELEM
        q.clear ← q.clear + 1
    end while
end function
```

significant improvement when the queue operates near empty, making BIFFQ effective across all operating conditions. [49]

Algorithm 7 Batched Improved FastForward Queue Operations [49]

```
function BIFFQ_WSPACE(q, needed)
    space ← q.limit − q.write
    if space < needed then
        return space                                            ▷ Force limit update
    end if
    return space
end function

function BIFFQ_ENQUEUE_LOCAL(q, e)
    q.buf[q.buffered] ← e                                       ▷ Store in buffer
    q.buffered ← q.buffered + 1
end function

function BIFFQ_ENQUEUE_PUBLISH(q)
    for i ← 0 to q.buffered − 1 do
        q.slots[q.write ∧ q.mask] ← q.buf[i]                    ▷ Fast burst
        q.write ← q.write + 1
    end for
    q.buffered ← 0
    q.limit ← q.write + H                                       ▷ Update limit
end function
```

B-Queue

Addresses the deadlock issues inherent in batching approaches through a self-adaptive backtracking mechanism that dynamically adjusts to production rates shown in algorithm 8. The producer maintains local `head` and `batch_head` pointers, probing `BATCH_SIZE` positions ahead when needed in line 3. The consumer's adaptive backtracking algorithm from lines 27 - 41 maintains a `batch_history` variable that records successful batch sizes from previous operations. When searching for data, it starts from this historical value rather than always beginning at `BATCH_SIZE`, significantly reducing latency when the producer operates slowly. If in line 29 - 31 the recorded size is below `BATCH_MAX`, the algorithm optimistically increments `batch_size` by `INCREMENT` (typically one cache line) to probe for higher throughput when the producer accelerates. The binary search then proceeds from this adaptive starting point, halving the batch size until finding available data or reaching zero. In the dequeue function, the consumer uses this computed value to update `batch_tail` in line 15. This eliminates the need for manual parameter adjustment or manual calling of a publish function while maintaining cache line separation and preventing deadlocks. [48]

Dyncamic Single Producer Single Consumer

A dynamically space allocating queue using a linked list with node caching to reduce memory allocation overhead, as shown in algorithm 9. Unlike bounded circular buffers like the Lamport Queue, dSPSC dynamically allocates nodes as needed, making it suitable for scenarios where queue size cannot be predetermined. The implementation maintains a dummy head node `head` to ensure producer and consumer always operate on different nodes to prevent cache line conflicts. The `SPSC_Buffer` (line 4, which is just a Lamport Queue) serves as a node cache to recycle deallocated nodes to minimize malloc or free calls. When pushing, the producer first checks the cache for a recycled node in line 8, falling back to malloc only when the cache is empty in line 11. After setting the data and next pointer, a memory barrier ensures correct ordering before linking the new node into the list in lines 20-22. The consumer checks for available data by testing if the dummy head points to a data node in line 28. Upon a successful pop, the consumer advances the head pointer so the data node becomes the new dummy and then attempts to cache the old dummy for reuse in lines 30-33. While node caching improves performance, reading and referencing the pointer so often causes memory accesses spread over multiple cache lines. As shown earlier this leads to cache misses. [46]

Unbounded Single Producer Single Consumer

An unbounded queue that links multiple Lamport Queues to combine the cache efficiency of Lamports circular buffer queues with unlimited capacity, as shown in algorithm 10. Unlike dSPSC which uses scattered linked list nodes, uSPSC maintains spatial locality by keeping data in contiguous circular buffers while only linking the buffers themselves. The implementation uses two pointers `buf_w` pointing to the producer's current write buffer and `buf_r` pointing to the consumer's current read buffer. When pushing, the producer checks if the current buffer

Algorithm 8 B-Queue with Self-Adaptive Backtracking[48]

```
1: function BQUEUE_ENQUEUE(q, e)
2:     if q.head = q.batch_head then                                  ▷ No empty slots
3:         if q.buffer[(q.head + BATCH_SIZE) mod q.size] ≠ NULL then
4:             return −1                                              ▷ Queue full
5:         end if
6:         q.batch_head ← q.head + BATCH_SIZE
7:     end if
8:     q.buffer[q.head mod q.size] ← e
9:     q.head ← q.head + 1
10:    return 0
11: end function
12:
13: function BQUEUE_DEQUEUE(q)
14:    if q.tail = q.batch_tail then                                  ▷ No filled slots
15:        batch_tail ← ADAPTIVE_BACKTRACK(q)
16:        if batch_tail = −1 then
17:            return NULL
18:        end if
19:        q.batch_tail ← batch_tail
20:    end if
21:    e ← q.buffer[q.tail mod q.size]
22:    q.buffer[q.tail mod q.size] ← NULL
23:    q.tail ← q.tail + 1
24:    return e
25: end function
26:
27: function ADAPTIVE_BACKTRACK(q)
28:    batch_size ← q.batch_history                                   ▷ Start from historical value
29:    if batch_size < BATCH_MAX then
30:        batch_size ← batch_size + INCREMENT                        ▷ Try larger batch
31:    end if
32:    while batch_size > 0 do
33:        batch_tail ← q.tail + batch_size
34:        if q.buffer[(batch_tail − 1) mod q.size] ≠ NULL then
35:            q.batch_history ← batch_size                           ▷ Remember successful size
36:            return batch_tail
37:        end if
38:        batch_size ← batch_size/2                                  ▷ Binary search
39:    end while
40:    return −1
41: end function
```

is full in line 2, and if so, requests a new buffer from the pool via `next_w()` in line 3 before writing the data to `buf_w`. The consumer first checks if its current buffer is empty in line 10. If empty, it determines whether the queue is truly empty by comparing read and write buffer pointers in line 11. If they point to the same buffer, no more data exists. Otherwise, after rechecking emptiness to prevent race conditions in line 14, the consumer obtains the next buffer via `next_r()` and releases the empty buffer back to the pool for recycling in lines 15-17. This double-check prevents data loss when the producer writes to the current buffer between the initial emptiness check and the buffer comparison. By reusing entire buffers rather than individual nodes, uSPSC matches bounded SPSC queues cache behavior while providing unbounded capacity. [47]

Algorithm 9 Dynamic List-based SPSC Queue (dSPSC) Operations [46]

```
struct Node { void* data; Node* next; }
Node* head;                                   ▷ Points to dummy node
Node* tail;                                   ▷ Points to last data node
SPSC_Buffer cache;                            ▷ Bounded cache for node recycling

function ALLOCNODE
    Node* n ← NULL
    if cache.pop(&n) then                     ▷ Try cache first
        return n
    end if
    n ← (Node*)malloc(sizeof(Node))
    return n
end function

function PUSH(void* data)
    Node* n ← allocnode()                     ▷ Get node from cache or malloc
    n->data ← data
    n->next ← NULL
    WMB()                                     ▷ Write Memory Barrier
    tail->next ← n                            ▷ Link new node
    tail ← n                                  ▷ Update tail pointer
    return true
end function

function POP(void** data)
    if head->next ≠ NULL then                 ▷ Check if data available
        Node* n ← head                        ▷ Save current dummy
        *data ← (head->next)->data            ▷ Extract data
        head ← head->next                     ▷ Advance to next node
        if !cache.push(n) then                ▷ Try to recycle old dummy
            free(n)                           ▷ Free if cache full
        end if
        return true
    end if
    return false                              ▷ Queue empty
end function
```

MultiPush Single Producer Single Consumer

Reduces the ping-pong effect in Lamport's circular buffer by batching multiple elements before insertion, as shown in algorithm 11. Instead of writing elements one by one directly to the shared buffer, MultiPush Single Producer Single Consumer (mSPSC) accumulates items in a thread-local array `batch`. The producer stores incoming data in the batch array in lines 2 and 3, and when the batch reaches `BATCH_SIZE` in line 4, the producer calls `multipush` to insert all elements at once in line 5. The `multipush` function first calculates the final write position in line 11 and checks if sufficient space exists in line 12. As seen in lines 15 - 17 elements are written in reverse order, starting from the furthest position and working backwards. This backward insertion creates distance between the write pointer and where the consumer is reading, ensuring they operate on different cache lines. A write memory barrier in line 18 ensures all batch writes are visible before updating the write pointer in line 19. The batch counter resets in line 20, preparing for the next batch. The `flush` function in lines 24 -

Algorithm 10 Unbounded Single Producer Single Consumer Operations[47]

```
function USPSC_PUSH(q, data)
    if q.buf_w.full() then                              ▷ Current buffer full
        q.buf_w ← q.pool.next_w()                       ▷ Get new buffer
    end if
    q.buf_w.push(data)
    return true
end function

function USPSC_POP(q, data)
    if q.buf_r.empty() then
        if q.buf_r = q.buf_w then                       ▷ Same buffer?
            return false                                ▷ Queue truly empty
        end if
        if q.buf_r.empty() then                         ▷ Recheck after comparison
            tmp ← q.pool.next_r()
            q.pool.release(q.buf_r)                     ▷ Recycle buffer
            q.buf_r ← tmp
        end if
    end if
    return q.buf_r.pop(data)
end function
```

29 allows forcing partial batch writes when needed. While adding an extra copy per element from batch to buffer, the improved cache behavior from reduced traffic from the coherence protocol compensates for this overhead. [46]

Jayantis SPSC Queue

A queue specifically for composability in larger MPSC structures, as shown in algorithm 12. Unlike traditional SPSC queues, this implementation includes `readFront` operations that enables observation of the queue's head element, crucial for the MPSC construction. The queue maintains a linked list where the tail always points to a dummy node. When enqueueing, the producer converts the current dummy node into a data node by writing the value in line 4. Then the producer links a new dummy node in line 5 and updates the tail pointer in line 6. This ensures the consumer never sees a partially constructed node. The `Help` variable in line 15 stores the dequeued value, allowing concurrent `readFront_e` operations to obtain valid data even after the original node is removed. The announcement mechanism prevents use-after-free errors. When the producer calls `readFront_e`, it writes the front node pointer to `Announce` in line 32, signaling the consumer not to immediately free that node. If the consumer encounters an announced node in line 17, it defers the node's deallocation to `FreeLater` in lines 18-20, ensuring the producer can safely read the node's value. The separate `readFront_d` operation in lines 41-47 is simpler since the consumer knows no concurrent dequeue can occur. This coordination enables wait-free progress while supporting the propagation mechanism, the process of pushing each local queue's minimum timestamp up through a binary tree to maintain a global minimum, needed for the logarithmic-time MPSC operations as seen in section 5.2.2. [44]

Algorithm 11 MultiPush Single Producer Single Consumer Operations[46]

```
function MSPSC_PUSH(q, data)
    q.batch[q.count] ← data
    q.count ← q.count + 1
    if q.count = BATCH_SIZE then
        return MULTIPUSH(q, q.batch, q.count)
    end if
    return true
end function

function MULTIPUSH(q, batch, len)
    last ← q.write + len − 1                          ▷ Calculate end position
    if q.slots[last mod q.size] ≠ NULL then
        return false                                  ▷ Not enough space
    end if
    for i ← len − 1 downto 0 do                       ▷ Reverse order
        q.slots[(q.write + i) mod q.size] ← batch[i]
    end for
    WMB()                                             ▷ Ensure all writes visible
    q.write ← (last + 1) mod q.size
    q.count ← 0                                       ▷ Reset batch counter
    return true
end function

function FLUSH(q)
    if q.count > 0 then
        return MULTIPUSH(q, q.batch, q.count)
    end if
    return true
end function
```

5.2.2. Multi Producer Single Consumer (MPSC)

This is a bit more complex to implement than the SPSC case. Multiple producers can enqueue items at the same time while a single consumer dequeues items. This includes implementing other strategies then just adding some memory barriers, such as helping and atomic primitives to maintain wait-freedom and consistency between the producers. There are multiple approaches that are available from different papers to achieve this:

Jayantis MPSC Queue

Achieves logarithmic time complexity by distributing the global queue across n local SPSC queues implemented like in section 5.2.1 and organized under a binary tree, as shown in algorithm 13. Each producer owns a dedicated local queue, eliminating producer-producer contention. When enqueueing, a producer obtains a global timestamp via LL/SC on a shared counter in lines 2-3, creating a unique ordering even if the SC fails, since some other producer must have incremented it. If LL/SC not supported on system architecture it can be replaced with versioned CAS. The producer then inserts a timestamped pair into its local queue in line 4 and propagates this timestamp up the tree in line 5. The tree maintains the invariant that each internal node holds the minimum timestamp of its subtree. The `propagate` function

Algorithm 12 Jayanti's SPSC Queue Operations [44]

```
function ENQUEUE(q, v)
    newNode ← new Node()                  ▷ Create new dummy node
    tmp ← q.Last                          ▷ Get current dummy tail
    tmp.val ← v                           ▷ Convert dummy to data node
    tmp.next ← newNode                    ▷ Link new dummy
    q.Last ← newNode                      ▷ Update tail pointer
end function

function DEQUEUE(q)
    tmp ← q.First                         ▷ Get head node
    if tmp = q.Last then                  ▷ Only dummy remains?
        return ⊥                          ▷ Queue empty
    end if
    retval ← tmp.val                      ▷ Read value
    q.Help ← retval                       ▷ Help concurrent readFront
    q.First ← tmp.next                    ▷ Remove from queue
    if tmp = q.Announce then              ▷ Was announced by readFront?
        tmp′ ← q.FreeLater                ▷ Get old deferred node
        q.FreeLater ← tmp                 ▷ Defer current node
        free(tmp′)                        ▷ Free old deferred node
    else
        free(tmp)                         ▷ Free immediately
    end if
    return retval
end function

function READFRONT_E(q)                   ▷ Called by enqueuer
    tmp ← q.First                         ▷ Read head pointer
    if tmp = q.Last then                  ▷ Queue empty?
        return ⊥
    end if
    q.Announce ← tmp                      ▷ Announce to prevent free
    if tmp ≠ q.First then                 ▷ Head changed (was dequeued)?
        retval ← q.Help                   ▷ Use helped value
    else
        retval ← tmp.val                  ▷ Read directly
    end if
    return retval
end function

function READFRONT_D(q)                   ▷ Called by dequeuer
    tmp ← q.First
    if tmp = q.Last then
        return ⊥
    end if
    return tmp.val                        ▷ Safe - no concurrent dequeue
end function
```

in lines 18 - 26 walks from leaf to root, calling `refresh` at each node. The double `refresh` pattern in lines 22 - 24 ensures correctness. if the first refresh fails, another process updated the node. if the second refresh also fails, that process must have read the updated children values and installed the correct minimum. The `refresh` function uses LL/SC in lines 28 - 33 to atomically update a node with the minimum of its children's timestamps. The consumer reads the root to find the producer with the earliest element in line 9 and then dequeues from that local queue in line 13 and propagates any changes in line 14. This design transforms the $O(n)$ scan of all queues into $O(\log n)$ tree traversals, while the space complexity remains $O(n+m)$ where m is the number of queued items. [44]

Algorithm 13 Jayanti's MPSC Queue Operations [44]

```
1: function ENQUEUE(q, p, v)
2:     tok ← LL(q.counter)                                  ▷ Read timestamp
3:     SC(q.counter, tok + 1)                               ▷ Try increment
4:     enqueue2(q.Q[p], (v, (tok, p)))                      ▷ Add timestamp to local queue
5:     PROPAGATE(q, q.Q[p])
6: end function
7:
8: function DEQUEUE(q, p)
9:     [t, id] ← read(q.T.root)                             ▷ Get min producer
10:    if id = ⊥ then
11:        return ⊥
12:    end if
13:    ret ← dequeue2(q.Q[id])
14:    PROPAGATE(q, q.Q[id])
15:    return ret.val
16: end function
17:
18: function PROPAGATE(q, localQueue)
19:    currentNode ← localQueue
20:    repeat
21:        currentNode ← parent(currentNode)
22:        if ¬REFRESH(q, currentNode) then                 ▷ First try
23:            REFRESH(q, currentNode)                      ▷ Second ensures correctness
24:        end if
25:    until currentNode = q.T.root
26: end function
27:
28: function REFRESH(q, node)
29:    LL(node)                                             ▷ Load-link node
30:    stamps ← read timestamps from node's children
31:    minT ← minimum timestamp from stamps
32:    return SC(node, minT)                                ▷ Store-conditional
33: end function
```

Drescher Queue

Uses a linked list with a dummy head node to eliminate producer contention, as shown in algorithm 14. Unlike traditional MPSC queues that require retry loops, producers complete enqueue in exactly three steps. First the producers clear the item's `next` pointer in line 6, then it atomically swaps the tail pointer via `FAS` in line 7 and further links the previous tail to the new item in line 8. The `FAS` operation ensures multiple producers can enqueue concurrently without interference. The consumer reads the head and its `next` pointer in lines 12 and 13 and afterwards advances the head in line 17 if the queue is non-empty. Subsequently the consumer handles the special case of the dummy node in lines 18-24. When the dummy is dequeued, it's immediately re-enqueued in line 19 to maintain the invariant that the queue always contains at least one element to prevent complex empty queue conditions. The guard integration through VOUCH and CLEAR ensures all orders are enqueued before guard acquisition in line 2 of VOUCH. The guard is released before checking for pending orders in line 11 of CLEAR, preventing lost updates. This ordering guarantees that either the current sequencer or a new thread will process all pending orders, achieves the wait-free progress guarantees with constant time enqueue operations. [28]

Algorithm 14 Drescher's Wait-Free MPSC Queue Operations

```
dummy.next ← 0
head ← &dummy
tail ← &dummy

procedure ENQUEUE(guard, item)
    item.next ← 0                          ▷ Clear next pointer
    prev ← FAS(guard.tail, item)           ▷ Atomic swap tail
    prev.next ← item                       ▷ Link to new item
end procedure

function DEQUEUE(guard)
    item ← guard.head
    next ← guard.head.next
    if next = 0 then                       ▷ Empty queue?
        return ⊥
    end if
    guard.head ← next
    if item = &dummy then                  ▷ Dequeued dummy?
        ENQUEUE(guard, item)               ▷ Re-enqueue dummy
        if guard.head.next = 0 then        ▷ Still empty?
            return ⊥
        end if
        guard.head ← guard.head.next
        return next
    end if
    return item
end function
```

Jiffy Queue

A queue that uses a linked list of fixed-size arrays (buffers), as shown in function `ENQUEUE(data)` in algorithm 15 and in DEQUEUE function in algorithm 16. Unlike other linked-list queues that allocate nodes per element, Jiffy amortizes allocation overhead by storing multiple elements in each buffer. Producers use FAA on a global tail counter to reserve slots in line 2 of `enqueue`, eliminating producer-producer synchronization except during buffer allocation. Each buffer contains an array of nodes with data and a 2-bit `isSet` flag indicating the node's state: `empty` (uninitialized), `set` (data written), or `handled` (already dequeued). When the current buffer fills, producers allocate new buffers and link them via CAS in lines 7 and 8. To reduce allocation contention, the producer obtaining the second slot in each buffer proactively allocates the next buffer in lines 21 - 26, ensuring smooth transitions between buffers. The consumer maintains a local head pointer and scans for the first non-handled element in lines 3 - 9 of `dequeue`. To ensure linearizability when producers stall: if the head element is still `empty`, the consumer scans forward to find a `set` element in line 20, then rescans backward in line 24 to ensure no earlier element became `set` during the scan. This prevents violating FIFO ordering when a slow producer completes after a faster one. The consumer can “fold” the queue by deleting fully-handled buffers in the middle of the list during scans, to not use to much memory even with stalled producers. This achieves wait-free progress guarantees with minimal synchronization. Producers only need one FAA per enqueue, while the consumer performs no atomic operations at all. [43]

Algorithm 15 Jiffy MPSC Queue Enqueue Operation [43]

```
function ENQUEUE(data)
    location ← FAA(tail, 1)                                  ▷ Reserve global index
    tempTail ← tailOfQueue
    while location is in unallocated buffer do               ▷ Beyond last buffer?
        if tempTail.next = NULL then
            newArr ← new BufferList()
            if CAS(tempTail.next, NULL, newArr) then
                CAS(tailOfQueue, tempTail, newArr)
            else
                delete newArr                                ▷ Another thread succeeded
            end if
        end if
        tempTail ← tailOfQueue                               ▷ Move to new buffer
    end while
    while location not in tempTail's buffer do               ▷ Location in earlier buffer?
        tempTail ← tempTail.prev                             ▷ Walk backward
    end while
    index ← location − tempTail.startIndex                   ▷ Buffer-local index
    tempTail.buffer[index].data ← data
    tempTail.buffer[index].isSet ← SET                       ▷ Mark as ready
    if index = 1 AND tempTail is last buffer then            ▷ Second slot?
        newArr ← new BufferList()                            ▷ Proactive allocation
        if NOT CAS(tempTail.next, NULL, newArr) then
            delete newArr
        end if
    end if
end function
```

DQueue

Combines local buffering with a segmented shared queue to minimize synchronization overhead, as shown in function `ENQUEUE(data)` in algorithm 17 and in `DEQUEUE` function in algorithm 18. DQueue reduces contention by having producers accumulate enqueue requests in thread-local ring buffers before writing them to the shared queue in batches. Producers reserve slots using FAA on a global tail counter in line 7 of `enqueue`, storing both the data and the reserved cell index (`cid`) in their local buffer. Each producer maintains a buffer of `Request` structures with capacity L, using `local_head` and `local_tail` pointers to track buffer state. When the buffer fills, detected in line 2, the producer calls `dump_local_buffer` to flush all buffered requests. During flushing, producers write values directly to their reserved cells in line 15 without synchronization, as each cell is exclusively owned by the reserving

Algorithm 16 Jiffy MPSC Queue Dequeue Operation [43]

```
function DEQUEUE
    n ← headOfQueue.buffer[head]
    while n.isSet = HANDLED do                          ▷ Skip dequeued items
        head ← head + 1
        if end of buffer then
            move to next buffer and delete current
        end if
        n ← headOfQueue.buffer[head]
    end while
    if queue is empty then
        return ⊥
    end if
    if n.isSet = SET then                               ▷ Ready to dequeue?
        data ← n.data
        n.isSet ← HANDLED
        head ← head + 1
        return data
    end if
    if n.isSet = EMPTY then                             ▷ Incomplete enqueue?
        tempN ← Scan(find first SET element)
        if no SET element found then
            return ⊥
        end if
        Rescan(n, tempN)                                ▷ Check for newly set elements
        data ← tempN.data
        tempN.isSet ← HANDLED
        return data
    end if
end function

function SCAN                                           ▷ Find first SET element
    for each element from current position do
        if element.isSet = SET then
            return element
        end if
        if entire buffer is HANDLED then
            fold queue (delete buffer)
        end if
    end for
    return NULL
end function

function RESCAN(start, end)                             ▷ Check for ordering violations
    for each element from start to end do
        if element.isSet = SET then
            end ← element                               ▷ Found earlier SET element
            restart scan from start
        end if
    end for
end function
```

producer. Producers cache their current segment pointer (`pseg`) and update it when moving to newer segments in line 18. The `find_segment` function traverses the segment list and allocates new segments on-demand using CAS in line 29. To maintain wait-freedom when producers stall, the consumer encountering an empty cell that should contain data, checked in line 7 of `dequeue`, distinguishes between an empty queue in line 8 and a pending enqueue by checking if head equals tail. For pending enqueues, `help_enqueue` in line 11 iterates through all producers' local buffers from lines 19-31, writing any buffered values to their reserved cells in line 28. The helper skips producers that have already moved past the target segment in line 24, avoiding unnecessary work. This ensures minimal synchronization with only one FAA per enqueue and no atomics for dequeue and improves cache locality via batched writes that reduce false sharing and writes that directly write to known cell locations without searching. The consumer's dequeue operation has its linearization point at line 14 where it increments the head pointer. [41]

Algorithm 17 DQueue MPSC Queue Enqueue Operation [41]

```
function ENQUEUE(Producer p, data)
    if next(p.local_tail) = p.local_head then
        dump_local_buffer(p)                          ▷ Flush when full
    end if
    tail ← p.local_tail
    p.local_buffer[tail].val ← data
    p.local_buffer[tail].cid ← FAA(q.tail, 1)         ▷ Reserve slot
    p.local_tail ← next(p.local_tail)
end function

function DUMP_LOCAL_BUFFER(Producer p)
    while p.local_head ≠ p.local_tail do
        r ← p.local_buffer[p.local_head]
        seg ← find_segment(p.pseg, r.cid)
        seg.cell[r.cid mod N] ← r.val                 ▷ Write in batch
        p.local_head ← next(p.local_head)
        if p.pseg ≠ seg then
            p.pseg ← seg                              ▷ Update segment cache
        end if
    end while
end function

function FIND_SEGMENT(Segment *sp, int cid)
    curr ← sp
    for i ← curr → id; i < cid/N; i++ do
        next ← curr → next
        if next = NULL then
            new ← new_segment(i + 1)
            if CAS(curr → next, NULL, new) then
                next ← new
            else
                delete new                            ▷ Another thread succeeded
            end if
        end if
        curr ← next
    end for
    return curr
end function
```

Algorithm 18 DQueue MPSC Queue Dequeue Operation [41]

```
function DEQUEUE(Consumer c)
    seg ← find_segment(c.cseg, q.head)
    if c.cseg ≠ seg then
        c.cseg ← seg                                  ▷ Update segment cache
    end if
    cell ← seg.cell[q.head mod N]
    if cell = ⊥ then                                  ▷ Empty cell?
        if q.head = q.tail then
            return EMPTY
        else
            help_enqueue()                            ▷ Help stalled producers
        end if
    end if
    q.head ← q.head + 1                               ▷ Linearization point
    return cell
end function

function HELP_ENQUEUE
    for each Producer p in system do
        for each Request r in p.local_buffer do
            pos ← r.cid
            val ← r.val
            seg ← find_segment(p.pseg, pos)
            if seg.id > pos/N then
                break                                 ▷ Producer moved past
            end if
            if seg.cell[pos mod N] = ⊥ then
                seg.cell[pos mod N] ← val             ▷ Help write
            end if
        end for
    end for
end function
```

5.2.3. Single Producer Multi Consumer (SPMC)

This case is trickier to implement, since now multiple reading workers have to be synchronized to read consistently without any unwanted behavior. Multiple producers was simpler, since making producers write specific data each is not so hard. In this case every consumer has to be synchronized so that when one item is consumed by a consumer, it cannot be consumed again from another. That is most probably also the reason only one algorithm for this contention category was found. The approach to achieve this in a wait-free manner is the following:

David Queue

Uses a 2D array ITEMS where each cell is a Swap object, a 1D array HEAD where each element is an FAA object, and a shared ROW register. The producer writes to consecutive cells in the current row of ITEMS and when the producer detects it gets overtaken by a consumer (reads a value indicating the cell was read), it jumps to the next row. Consumers read the active row from ROW, use FAA on HEAD[row] to get a unique index, then swap the value, indicating the cell was read, into ITEMS[row, index] to retrieve the element. The algorithm achieves O(1) time complexity with 3-bounded wait-free operations. Row migration ensures linearizability by preventing consumers from accessing future write locations. [45]

5.2.4. Multi Producer Multi Consumer (MPMC)

Finally a look into the MPMC case can be made. Synchronizing MPMC is a bit simpler then synchronizing SPMC, because here each consumer consumes the items of one other producer instead of sharing the same pool of data that has to be consumed. These are the algorithms that were found:

Kogan and Petranks queue

Uses a priority-based helping mechanism built on the Michael-Scott lock-free queue. Threads obtain phase numbers by calculating the maximum phase across all threads in a shared state array and adding one. Each thread records its operation details (phase, pending flag, operation type, node reference) in the state array at its thread ID index. During execution, threads traverse the state array and help all operations with phase numbers less than or equal to their own. Producers append nodes to the tail using CAS then update the pending flag to false in the state array and then advance the tail pointer. Consumers write their ID into the deqTid field of the head node then update their state entry and then advance head. Empty queues are handled by checking if head equals tail with a null next pointer. The algorithm ensures each operation completes within $O(n^2)$ steps through systematic helping, where threads can only bypass each other a bounded number of times. Linearization occurs at the CAS operations that modify the queue structure. [33]

Turn Queue

Uses a circular turn-based consensus mechanism. Producers publish nodes in 'enqueuers[tid]' array then the next turn is determined by tail's 'enqTid', with all threads helping the first non-null request to the right (modulo the array size), guaranteeing completion within 'MAX_THREADS' iterations. Consumers use dual arrays ('deqself' and 'deqhelp') where 'deqself[tid]' = 'deqhelp[tid]' opens a request. Threads assign nodes via CAS on 'node.deqTid' and publish results in 'deqhelp[tid]' to close requests. Empty queues trigger the 'giveUp()' to rollback while preserving concurrent assignments. Linearization occurs at tail and head advances. [37]

YMC queue

Uses an unbounded array (emulated via linked list of segments) with FAA-based indexing and fast-path/slow-path design. Producers obtain slots via FAA on tail counter and attempt CAS insertion. On failure, the producers publish requests ('value', 'pending', 'cell_id') and enter slow path. Helpers use Dijkstra's protocol—after marking cells unusable with top or buttom markers and they check for pending enqueue requests. Helpers traverse a peer ring, advancing only after completing requests, guaranteeing progress within $(n-1)^2$ failures. Consumers

FAA the head counter and call 'help_enq' to secure values. Slow-path dequeues announce candidate cells with monotonically increasing indices via CAS on request state. Linearizability requires enqueued values have indices < T (tail) and dequeued values have indices < H (head), maintained by 'advance_end_for_linearizability' calls. Failed operations advance the helper ring until all threads become helpers ensuring all threads finish. [40]

Feldman-Dechev Queue

Uses a ring buffer with sequence number distribution for contention management. Producers obtain a unique position via FAA on the tail counter, receiving a sequence ID ('seqid') that determines their slot ('seqid' modulo the capacity). They attempt to replace an 'EmptyNode' (containing only a 'seqid') with a 'ValueNode' (containing both 'seqid' and data) via CAS, retrying while temporarily pausing and rechecking if the slot's 'seqid' indicates a delayed operation. Consumers similarly use FAA on the head counter and replace 'ValueNodes' with 'EmptyNodes' containing 'seqid' + 'capacity'. Both operations employ bitmarking to handle out-of-order completions meaning when encountering a delayed element, threads mark it with a delay bit to signal correction is needed. After exceeding 'MAX_FAILS' attempts, operations switch to a slow path using an announcement table where threads publish operation descriptors. All threads periodically check this table (every 'CHECK_DELAY' operations) and help announced operations, guaranteeing completion within 'MAX_FAILS' + 'NUM_THREADS^2' steps. The sequence numbers additionally prevent ABA problems and enable progress despite random delays. [34]–[36]

wCQ

Uses a circular array with fast and slow paths. Producers claim slots via FAA on a tail counter, then attempt CAS to insert if the slot's cycle number indicates availability. After exceeding a patience threshold, producers record their operation in per-thread descriptors and enter a slow path where helpers execute the same operation on behalf of a stuck thread. The slow path replaces uses a modified non-atomic FAA, which is a distributed protocol that coordinates helpers through 'INC' (increment pending) and 'FIN' (finalized) bits in thread-local counters, ensuring all helpers converge to the same counter value and exactly one successfully increments the global counter. Consumers similarly use FAA on the head counter and mark consumed entries with a special bottom value. Both operations use a two-step protocol meaning enqueuers initially insert with 'Enq=false', then set 'Enq=true' after finalizing their request, allowing concurrent dequeuers to help finalize pending operations. The queue maintains 2n slots for n threads with note fields storing cycle numbers to prevent helpers from incorrectly modifying entries from previous cycles. [38]

Verma Queue

Uses an helper that runs on a dedicated core and mediates all queue operations through a shared state array. Producers submit enqueue requests by writing [operation, element]tuples to their designated position in the state array indexed by a thread ID, then spin until the ‘isCompleted’ flag is set. The helper continuously traverses the state array in round-robin (a traversal algorithm not deeper explained in this work) order, processing pending requests by appending nodes to the tail of an underlying linked list for enqueues or removing from the head for dequeues. The traversal time is bounded (at most n iterations for n threads) so that every thread finishes in a bounded time, while contention is eliminated by serializing all queue changes through the single helper thread. The helper thread includes volatile variables to ensure cache coherence and padding to avoid false sharing between state array entries. [39]

6. Implementation

7. Benchmarking and 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] T. E. Anderson, B. N. Bershad, E. D. Lazowska, and H. M. Levy, "Scheduler activations: Effective kernel support for the user-level management of parallelism", *SIGOPS Oper. Syst. Rev.*, vol. 25, no. 5, pp. 95–109, Sep. 1991. DOI: 10.1145/121133.121151. [Online]. Available: https://doi.org/10.1145/121133.121151.

[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] T. Fuchs and H. Murakami, "Evaluation of task scheduling algorithms and wait-free data structures for embedded multi-core systems", 2014. [Online]. Available: `https://api.semanticscholar.org/CorpusID:218073330`.

[28] G. Drescher and W. Schröder-Preikschat, "An experiment in wait-free synchronisation of priority-controlled simultaneous processes: Guarded sections", Friedrich-Alexander-Universität Erlangen-Nürnberg, Dept. of Computer Science, Tech. Rep. CS-2015-01, Jan. 2015. [Online]. Available: `https://www4.cs.fau.de/Publications/2015/drescher_15_cstr.pdf`.

[29] 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.

[30] 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`.

[31] 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`.

[32] 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`.

[33] 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.

[34] S. Feldman and D. Dechev, "A wait-free multi-producer multi-consumer ring buffer", *SIGAPP Appl. Comput. Rev.*, vol. 15, no. 3, pp. 59–71, Oct. 2015. DOI: 10.1145/2835260.2835264. [Online]. Available: https://doi.org/10.1145/2835260.2835264.

[35] D. Dechev, S. Feldman, and A. Barrington, "A scalable multi-producer multi-consumer wait-free ring buffer.", Sandia National Lab. (SNL-NM), Albuquerque, NM (United States), Apr. 2015. [Online]. Available: https://www.osti.gov/biblio/1531271.

[36] A. Barrington, S. Feldman, and D. Dechev, "A scalable multi-producer multi-consumer wait-free ring buffer", in *Proceedings of the 30th Annual ACM Symposium on Applied Computing*, ser. SAC '15, Salamanca, Spain: Association for Computing Machinery, 2015, pp. 1321–1328. DOI: 10.1145/2695664.2695924. [Online]. Available: https://doi.org/10.1145/2695664.2695924.

[37] P. Ramalhete and A. Correia, "Poster: A wait-free queue with wait-free memory reclamation", *SIGPLAN Not.*, vol. 52, no. 8, pp. 453–454, Jan. 2017, Full version available at https://github.com/pramalhe/ConcurrencyFreaks/blob/master/papers/crturnqueue-2016.pdf. DOI: 10.1145/3155284.3019022. [Online]. Available: https://doi.org/10.1145/3155284.3019022.

[38] 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.

[39] M. Verma, "Scalable and performance-critical data structures for multicores", Jun. 2013, Thesis to obtain the Master of Science Degree in Information Systems and Computer Engineering. [Online]. Available: https://web.tecnico.ulisboa.pt/~ist14191/repository/Thesis_Mudit_Verma.pdf.

[40] C. Yang and J. Mellor-Crummey, "A wait-free queue as fast as fetch-and-add", in *Proceedings of the 21st ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, ser. PPoPP '16, Barcelona, Spain: Association for Computing Machinery, 2016. DOI: 10.1145/2851141.2851168. [Online]. Available: https://doi.org/10.1145/2851141.2851168.

[41] J. Wang, Q. Jin, X. Fu, Y. Li, and P. Shi, "Accelerating wait-free algorithms: Pragmatic solutions on cache-coherent multicore architectures", *IEEE Access*, vol. 7, pp. 74 653–74 669, 2019. DOI: 10.1109/ACCESS.2019.2920781.

[42] D. Adas and R. Friedman, “A fast wait-free multi-producers single-consumer queue”, in *Proceedings of the 23rd International Conference on Distributed Computing and Networking*, ser. ICDCN ’22, Delhi, AA, India: Association for Computing Machinery, 2022, pp. 77–86. DOI: `10.1145/3491003.3491004`. [Online]. Available: `https://doi.org/10.1145/3491003.3491004`.

[43] 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`.

[44] P. Jayanti and S. Petrovic, “Logarithmic-time single deleter, multiple inserter wait-free queues and stacks”, in *FSTTCS 2005: Foundations of Software Technology and Theoretical Computer Science*, S. Sarukkai and S. Sen, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 408–419.

[45] M. David, “A single-enqueuer wait-free queue implementation”, in *Distributed Computing*, R. Guerraoui, Ed., Berlin, Heidelberg: Springer Berlin Heidelberg, 2004, pp. 132–143.

[46] M. Torquati, *Single-producer/single-consumer queues on shared cache multi-core systems*, 2010. arXiv: `1012.1824 [cs.DS]`. [Online]. Available: `https://arxiv.org/abs/1012.1824`.

[47] M. Aldinucci, M. Danelutto, P. Kilpatrick, M. Meneghin, and M. Torquati, “An efficient synchronisation mechanism for multi-core systems”, in *Euro-Par 2012 Parallel Processing*, C. Kaklamanis, T. Papatheodorou, and P. G. Spirakis, Eds., ser. Lecture Notes in Computer Science, The provided PDF indicates this work is related to Euro-Par 2012. Exact page numbers for this specific title within the proceedings were not found in the immediate search; the closely related paper "An Efficient Unbounded Lock-Free Queue for Multi-core Systems" by the same authors appears on pages 662-673 of the same volume., vol. 7484, Berlin, Heidelberg: Springer, 2012.

[48] J. Wang, K. Zhang, X. Tang, and B. Hua, “B-queue: Efficient and practical queuing for fast core-to-core communication”, *International Journal of Parallel Programming*, vol. 41, no. 1, pp. 137–159, Feb. 2013. DOI: `10.1007/s10766-012-0213-x`. [Online]. Available: `https://doi.org/10.1007/s10766-012-0213-x`.

[49] V. Maffione, G. Lettieri, and L. Rizzo, “Cache-aware design of general-purpose single-producer–single-consumer queues”, *Software: Practice and Experience*, vol. 49, no. 5, pp. 748–779, 2019. DOI: `https://doi.org/10.1002/spe.2675`. eprint: `https://onlinelibrary.wiley.com/doi/pdf/10.1002/spe.2675`. [Online]. Available: `https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2675`.

[50] J. Giacomoni, T. Moseley, and M. Vachharajani, “Fastforward for efficient pipeline parallelism: A cache-optimized concurrent lock-free queue”, Jan. 2008. DOI: `10.1145/1345206.1345215`.

[51] D. B. Demir, `https://github.com/DevrimBaran/MA.git`.

[52] P. Khanchandani and R. Wattenhofer, "On the importance of synchronization primitives with low consensus numbers", in *Proceedings of the 19th International Conference on Distributed Computing and Networking*, ser. ICDCN '18, Varanasi, India: Association for Computing Machinery, 2018. DOI: `10.1145/3154273.3154306`. [Online]. Available: `https://doi.org/10.1145/3154273.3154306`.

[53] D. Bédin, F. Lépine, A. Mostéfaoui, D. Perez, and M. Perrin, "Wait-free algorithms: The burden of the past", Mar. 2024. DOI: `10.21203/rs.3.rs-4125819/v1`.

[54] C. Johnen, A. Khattabi, and A. Milani, "Efficient Wait-Free Queue Algorithms with Multiple Enqueuers and Multiple Dequeuers", in *26th International Conference on Principles of Distributed Systems (OPODIS 2022)*, E. Hillel, R. Palmieri, and E. Rivière, Eds., ser. Leibniz International Proceedings in Informatics (LIPIcs), vol. 253, Dagstuhl, Germany: Schloss Dagstuhl – Leibniz-Zentrum für Informatik, 2023, 4:1–4:19. DOI: `10.4230/LIPIcs.OPODIS.2022.4`. [Online]. Available: `https://drops.dagstuhl.de/entities/document/10.4230/LIPIcs.OPODIS.2022.4`.

[55] H. Naderibeni and E. Ruppert, "A wait-free queue with polylogarithmic step complexity", in *Proceedings of the 2023 ACM Symposium on Principles of Distributed Computing*, ser. PODC '23, Orlando, FL, USA: Association for Computing Machinery, 2023, pp. 124–134. DOI: `10.1145/3583668.3594565`. [Online]. Available: `https://doi.org/10.1145/3583668.3594565`.

[56] U. Drepper, "What every programmer should know about memory", *Red Hat, Inc*, vol. 11, no. 2007, p. 2007, 2007.

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

FAA Fetch and Add

FAS Fetch and Store

LL/SC Load Linked/Store Conditional

FFQ FastForward Queue

IFFQ Improved FastForward Queue

BIFFQ Batched Improved FastForward Queue

uSPSC Unbounded Single Producer Single Consumer

dSPSC Dyncamic Single Producer Single Consumer

mSPSC MultiPush Single Producer Single Consumer

BLQ Batched Lamport Queue

LLQ Lazy Lamport Queue

BLQ Batched Lamport Queue

List of Figures

List of Tables

List of Algorithms

A. Appendix