



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

Examined by

Supervised by

Submitted on

M. Sc. Informatik

Prof. Andreas Wortmann

Marc Fischer

July 12, 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	<i>Wait-free synchronisation for inter-process communication in real-time systems</i>
German title	<i>Wait-free Synchronisation für Interprozesskommunikation in Echtzeitsystemen</i>

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 12, 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 Synchronisation, 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

Kurzfassung	ii
Abstract	iii
1. Introduction	1
1.1. Motivation	1
1.2. Objective	2
1.3. Structure of the Thesis	2
2. Background	3
2.1. Real-Time Systems	3
2.2. Inter-Process Communication	3
2.2.1. Shared Memory	4
2.3. Synchronization	4
2.3.1. Mutual Exclusion	5
2.4. Lock-Free Synchronization	7
2.4.1. Michael and Scott's Lock-Free Queue	8
2.4.2. Atomic Primitives	10
2.5. Wait-Free Synchronization	11
2.6. Rust Programming Language	11
3. Related Work	13
4. Methodology	15
5. Analyzing existing Wait-Free Data Structures and Algorithms	17
5.1. Optimal Wait-Free Data Structure	17
5.2. Wait-Free Algorithms	17
5.2.1. Single Producer Single Consumer (SPSC)	18
5.2.2. Multi Producer Single Consumer (MPSC)	27
5.2.3. Single Producer Multi Consumer (SPMC)	28
5.2.4. Multi Producer Multi Consumer (MPMC)	28
6. Implementation	31
7. Benchmarking and Results	32
8. Conclusion and Future Work	33

Contents

Bibliography	34
List of Acronyms	40
A. Appendix	44

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 without 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 even though 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 communi-

1. Introduction

cation 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 through 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 disaster. The system 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

2. Background

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]

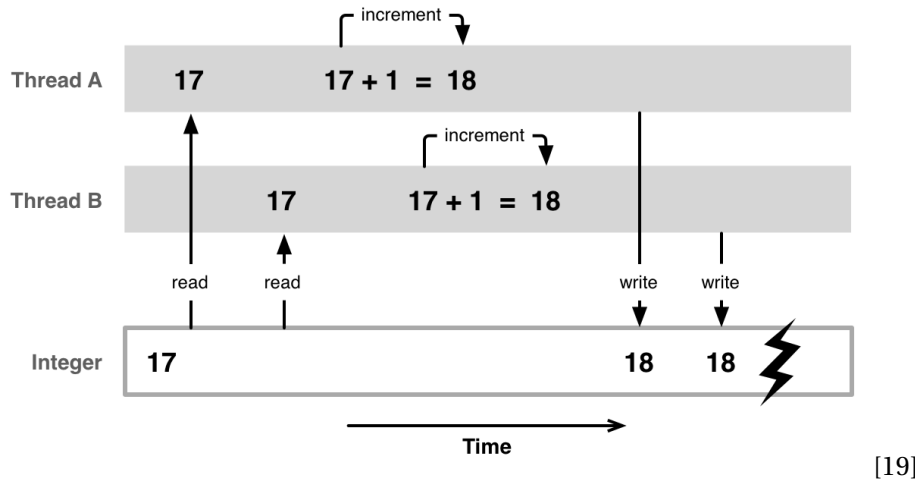
2.3. Synchronization

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.

2. Background

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.



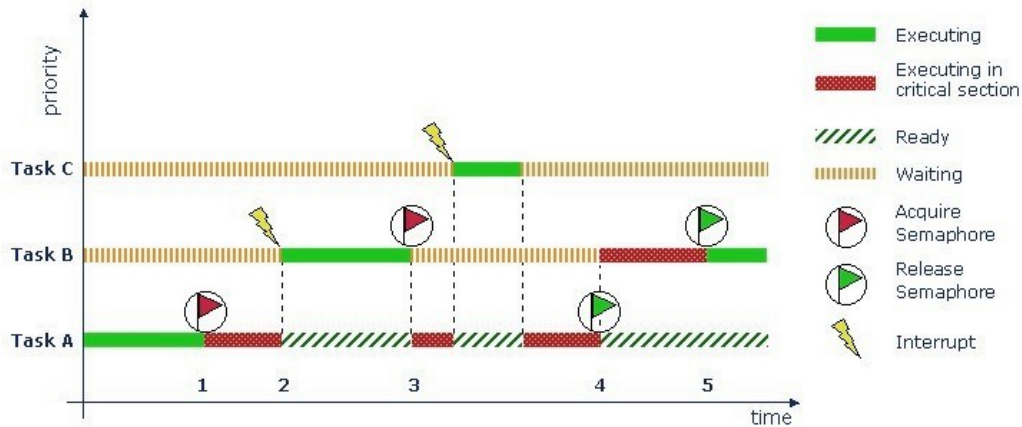
2.3.1. Mutual Exclusion

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 p_1 already accessed the shared resource x and is still working on it, a second process p_2 , which tries to access that shared resource x has to wait until the process p_1 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 acquire 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 until the lock is released for that resource.

It is clear that this approach inherently relays 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 acquire the lock first to enter the critical section, when multiple processes wait for the access is mostly set by a scheduler. Since wait-

2. Background

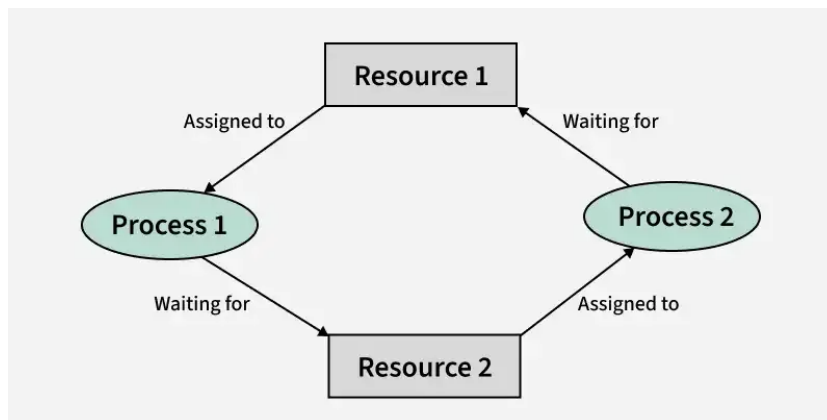
Figure 2.2.: Mutual exclusion between three tasks(processes), which access the same critical section. Multiple processes need to stop working and just wait for other processes to finish their work. See the waiting phase of the processes.



[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 acquiring a lock to enter the critical section. This process would wait for an indefinite time and will never enter the shared resource. A process that will never access a shared resource that it tries to enter starves

2. Background

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 acquired? This results in a situation seen in fig. 2.3 where these processes now indefinitely wait 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 priority 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 priority 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 synchronization 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

2. Background

synchronization does not guarantee that all processes will complete their operations in a finite number of steps. This means that starvation or even priority 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 called 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 tries 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 enqueue 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 let's say 2 processes p1 and p2 until 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 retries 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 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 indefinitely if for an indefinite time other processes are always executing line 16 immediately before p1 could execute line 16. This means that in very high contention scenarios, a process may be delayed indefinitely and starve out. In a HRTS this could lead to violating timing constraints, since the process would not finish

2. Background

his task in the defined timing window, which is unacceptable. This is why a slightly different approach which guarantees 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

```

1: function INITIALIZE(Q : pointer to queue_t)
2:   node = new node()
3:   node.next.ptr = NULL
4:   Q.Head = node
5:   Q.Tail = node
6: end function
7: function ENQUEUE(Q : pointer to queue_t, value : data_type)
8:   node = new node()
9:   node.value = value
10:  node.next.ptr = NULL
11:  loop
12:    tail = Q.Tail
13:    next = tail.ptr.next
14:    if tail == Q.Tail then
15:      if next.ptr == NULL then
16:        if CAS(&tail.ptr.next, next, (node, next.count + 1)) then
17:          break
18:        end if
19:      else
20:        CAS(&Q.Tail, tail, (next.ptr, tail.count + 1))
21:      end if
22:    end if
23:  end loop
24:  CAS(&Q.Tail, tail, (node, tail.count + 1))
25: end function
26: function DEQUEUE(Q : pointer to queue_t, pvalue : pointer to data_type)
27:  loop
28:    head = Q.Head
29:    tail = Q.Tail
30:    next = head.ptr.next
31:    if head == Q.Head then
32:      if head.ptr == tail.ptr then
33:        if next.ptr == NULL then
34:          return FALSE
35:        else
36:          CAS(&Q.Tail, tail, (next.ptr, tail.count + 1))
37:        end if
38:      else
39:        *pvalue = next.ptr.value
40:        if CAS(&Q.Head, head, (next.ptr, head.count + 1)) then
41:          break
42:        end if
43:      end if
44:    end if
45:  end loop
46:  free(head.ptr)
47:  return TRUE
48: end function

```

▷ Allocate a dummy node
 ▷ Make it the only node in the list
 ▷ Both Head and Tail point to this dummy node
 ▷ Allocate a new node from the free list
 ▷ Copy enqueue value into node
 ▷ Set next pointer of node to NULL
 ▷ Keep trying until Enqueue is done
 ▷ Read Tail (pointer + count) together
 ▷ Read next ptr + count together
 ▷ Are tail & next consistent?
 ▷ Tail is the last node?
 ▷ Link the new node; Enqueue is done
 ▷ Tail not pointing to the last node
 ▷ Move Tail forward (helping another enqueue)
 ▷ Final attempt to swing Tail to the inserted node
 ▷ Keep trying until Dequeue is done
 ▷ Read head->next
 ▷ Still consistent?
 ▷ Empty or Tail behind?
 ▷ Queue is empty
 ▷ Tail is behind, help move it
 ▷ No need to adjust Tail
 ▷ Read value before CAS
 ▷ Dequeue is done
 ▷ Safe to free old dummy node

[26]

2. Background

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 architectures 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 fundamental 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 than CAS, if conditions before updating do not need to be checked. [28]

2.5. Wait-Free Synchronization

Lock-freedom solves the problem of 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 expands 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 Petrunkin, where operations on a data structure are 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 collector, 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 Systems (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 vulnerabilities happen because of memory safety issues. If the real-time application would run on an isolated system with no internet connection, this would

2. Background

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 dequeuing). He also defines the usage of action sets and environment constraints, which separate 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:

3. Related Work

- 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 methodologies 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' execution (for later understanding: 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]

Herlihy's 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 Petrak later invented a method called fast-path slow-path, where 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 than 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 queries:

- “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

4. Methodology

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 constraints 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 language 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

5. Analyzing existing Wait-Free Data Structures and Algorithms

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. In this subsection the focus will lie on benchmarking all the different SPSC approaches found so that the optimal algorithm for the SPSC use case can be selected. The following list shows the algorithms that are used for the SPSC contention case. 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 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 by computing

5. Analyzing existing Wait-Free Data Structures and Algorithms

$\text{write} - \text{read} = N$, 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` and increments `write` to signal that the written data is available. The consumer mirrors this behavior by checking if the queue is empty with $\text{read} = \text{write}$ by reading the producer's `write` index. If the queue is not empty the consumer reads that data from the slot at position `read` and increments its `read` index to signal that the slot is available. 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, which reduces performance. According to Ulrich Drepper in 2007 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 the performance can degrade by 390%, 734%, and 1,147% for 2, 3, and 4 threads respectively and 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 Lazy Lamport Queue [49]

```

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

```

[49]

Lazy Lamport Queue

Reduces the described cache misses of Lamports queue by postponing index reads until necessary, as shown in algorithm 3. The producer maintains a local `read_shadow` copy and only updates it when running out of known free slots. 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, typically 8) permanently empty, preventing producer and consumer from touching the same cache line when the queue is full. These optimizations work well when one thread is faster than the other, reducing worst-case misses from 3 to about 2 per item. [49]

5. Analyzing existing Wait-Free Data Structures and Algorithms

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:      $q.read\_shadow \leftarrow q.read$                                        ▷ Update shadow
4:     if  $q.write - q.read\_shadow = N - K$  then
5:       return -1                                                         ▷ No space
6:     end if
7:   end if
8:    $q.slots[q.write \wedge q.mask] \leftarrow e$ 
9:    $store\_release\_barrier()$ 
10:   $q.write \leftarrow q.write + 1$ 
11:  return 0
12: end function
13:
14: function LLQ_DEQUEUE( $q$ )
15:   if  $q.read = q.write\_shadow$  then                                       ▷ Lazy load check
16:      $q.write\_shadow \leftarrow q.write$                                        ▷ 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:    $e \leftarrow q.slots[q.read \wedge q.mask]$ 
23:    $q.read \leftarrow q.read + 1$ 
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`, filling slots without updating the shared `write` index. Only `blq_enqueue_publish` makes the batch visible by advancing `write`. The consumer works symmetrically, using `read_priv` for local progress before updating `read`. With typical batch sizes like $B = 32$, synchronization overhead is amortized across many 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]

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` before writing. When the slot is empty (`NULL`), the producer writes the data directly and advances its private `head` index. The consumer follows a similar pattern by reading from the current slot position, and if data is present (non-`NULL`), it retrieves the value, writes `NULL` to mark the slot empty, and advances its private `tail` index. This design couples control

5. Analyzing existing Wait-Free Data Structures and Algorithms

Algorithm 4 Batched Lamport Queue Operations [49]

```

1: function BLQ_ENQUEUE_SPACE( $q$ ,  $needed$ )
2:    $space \leftarrow N - K - (q.write\_priv - q.read\_shadow)$ 
3:   if  $space < needed$  then
4:      $q.read\_shadow \leftarrow q.read$  ▷ Update shadow
5:      $space \leftarrow N - K - (q.write\_priv - q.read\_shadow)$ 
6:   end if
7:   return  $space$ 
8: end function
9:
10: function BLQ_ENQUEUE_LOCAL( $q$ ,  $e$ )
11:    $q.slots[q.write\_priv \wedge q.mask] \leftarrow e$ 
12:    $q.write\_priv \leftarrow q.write\_priv + 1$ 
13: end function
14:
15: function BLQ_ENQUEUE_PUBLISH( $q$ )
16:    $store\_release\_barrier()$ 
17:    $q.write \leftarrow q.write\_priv$ 
18: end function
19:
20: function BLQ_DEQUEUE_SPACE( $q$ )
21:    $available \leftarrow q.write\_shadow - q.read\_priv$ 
22:   if  $available = 0$  then
23:      $q.write\_shadow \leftarrow q.write$  ▷ Update shadow
24:      $available \leftarrow q.write\_shadow - q.read\_priv$ 
25:   end if
26:   return  $available$ 
27: end function
28:
29: function BLQ_DEQUEUE_LOCAL( $q$ )
30:    $load\_acquire\_barrier()$ 
31:    $e \leftarrow q.slots[q.read\_priv \wedge q.mask]$ 
32:    $q.read\_priv \leftarrow q.read\_priv + 1$ 
33:   return  $e$ 
34: end function
35:
36: function BLQ_DEQUEUE_PUBLISH( $q$ )
37:    $q.read \leftarrow q.read\_priv$ 
38: end function

```

and 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]

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 (typically 32) is empty before proceeding, ensuring it works far ahead of the consumer. This check happens only once

5. Analyzing existing Wait-Free Data Structures and Algorithms

Algorithm 5 FastForward Queue Operations [50]

```

1: function FFQ_ENQUEUE( $q, data$ )
2:   if  $q.buffer[q.head] \neq \text{NULL}$  then
3:     return EWOULDBLOCK
4:   end if
5:    $q.buffer[q.head] \leftarrow data$ 
6:    $q.head \leftarrow \text{NEXT}(q.head)$ 
7:   return 0
8: end function
9:
10: function FFQ_DEQUEUE( $q$ )
11:    $data \leftarrow q.buffer[q.tail]$ 
12:   if  $data = \text{NULL}$  then
13:     return EWOULDBLOCK
14:   end if
15:    $q.buffer[q.tail] \leftarrow \text{NULL}$ 
16:    $q.tail \leftarrow \text{NEXT}(q.tail)$ 
17:   return  $data$ 
18: end function

```

every H items when `write` reaches `limit`. The consumer delays clearing slots through `iffq_dequeue_publish`, 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]

Algorithm 6 Improved FastForward Queue Operations [49]

```

1: function IFFQ_ENQUEUE( $q, e$ )
2:   if  $q.write = q.limit$  then                                     ▷ Check limit
3:      $next\_limit \leftarrow q.limit + H$ 
4:     if  $q.slots[next\_limit \wedge q.mask] \neq \text{NULL\_ELEM}$  then
5:       return -1                                                  ▷ No space
6:     end if
7:      $q.limit \leftarrow next\_limit$                                 ▷ Free partition
8:   end if
9:    $q.slots[q.write \wedge q.mask] \leftarrow e$ 
10:   $q.write \leftarrow q.write + 1$ 
11:  return 0
12: end function
13:
14: function IFFQ_DEQUEUE_LOCAL( $q$ )
15:   $e \leftarrow q.slots[q.read \wedge q.mask]$ 
16:  if  $e = \text{NULL\_ELEM}$  then
17:    return  $\text{NULL\_ELEM}$ 
18:  end if
19:   $q.read \leftarrow q.read + 1$ 
20:  return  $e$ 
21: end function
22:
23: function IFFQ_DEQUEUE_PUBLISH( $q$ )
24:  while  $q.clear \neq \text{next\_clear}(q.read)$  do
25:     $q.slots[q.clear \wedge q.mask] \leftarrow \text{NULL\_ELEM}$ 
26:     $q.clear \leftarrow q.clear + 1$ 
27:  end while
28: end function

```

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, then `biffq_enqueue_publish` writes them to the queue in a rapid burst. 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 significant improvement when the queue operates near empty, making BIFFQ effective across all operating conditions. [49]

Algorithm 7 Batched Improved FastForward Queue Operations [49]

```

1: function BIFFQ_WSPACE( $q, needed$ )
2:    $space \leftarrow q.limit - q.write$ 
3:   if  $space < needed$  then
4:     return  $space$  ▷ Force limit update
5:   end if
6:   return  $space$ 
7: end function
8:
9: function BIFFQ_ENQUEUE_LOCAL( $q, e$ )
10:   $q.buf[q.buffered] \leftarrow e$  ▷ Store in buffer
11:   $q.buffered \leftarrow q.buffered + 1$ 
12: end function
13:
14: function BIFFQ_ENQUEUE_PUBLISH( $q$ )
15:  for  $i \leftarrow 0$  to  $q.buffered - 1$  do
16:     $q.slots[q.write \wedge q.mask] \leftarrow q.buf[i]$  ▷ Fast burst
17:     $q.write \leftarrow q.write + 1$ 
18:  end for
19:   $q.buffered \leftarrow 0$ 
20:   $q.limit \leftarrow q.write + H$  ▷ Update limit
21: 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. The consumer's adaptive backtracking algorithm (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 the recorded size is below `BATCH_MAX`, the algorithm optimistically increments 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. This eliminates the need for manual parameter ad-

5. Analyzing existing Wait-Free Data Structures and Algorithms

justment or manual calling of a publish function while maintaining cache line separation and preventing deadlocks. [48]

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  $q.buffer[(q.head + BATCH\_SIZE) \bmod q.size] \neq \text{NULL}$  then
4:       return -1                                                    ▷ Queue full
5:     end if
6:      $q.batch\_head \leftarrow q.head + BATCH\_SIZE$ 
7:   end if
8:    $q.buffer[q.head \bmod q.size] \leftarrow e$ 
9:    $q.head \leftarrow q.head + 1$ 
10:  return 0
11: end function
12:
13: function BQUEUE_DEQUEUE( $q$ )
14:   if  $q.tail = q.batch\_tail$  then                                     ▷ No filled slots
15:      $batch\_tail \leftarrow \text{ADAPTIVE\_BACKTRACK}(q)$ 
16:     if  $batch\_tail = -1$  then
17:       return NULL
18:     end if
19:      $q.batch\_tail \leftarrow batch\_tail$ 
20:   end if
21:    $e \leftarrow q.buffer[q.tail \bmod q.size]$ 
22:    $q.buffer[q.tail \bmod q.size] \leftarrow \text{NULL}$ 
23:    $q.tail \leftarrow q.tail + 1$ 
24:   return  $e$ 
25: end function
26:
27: function ADAPTIVE_BACKTRACK( $q$ )
28:    $batch\_size \leftarrow q.batch\_history$                                ▷ Start from historical value
29:   if  $batch\_size < BATCH\_MAX$  then
30:      $batch\_size \leftarrow batch\_size + INCREMENT$                      ▷ Try larger batch
31:   end if
32:   while  $batch\_size > 0$  do
33:      $batch\_tail \leftarrow q.tail + batch\_size$ 
34:     if  $q.buffer[(batch\_tail - 1) \bmod q.size] \neq \text{NULL}$  then
35:        $q.batch\_history \leftarrow batch\_size$                            ▷ Remember successful size
36:       return  $batch\_tail$ 
37:     end if
38:      $batch\_size \leftarrow batch\_size / 2$                                ▷ Binary search
39:   end while
40:   return -1
41: end function

```

Dyncamic Single Producer Single Consumer (dSPSC)

Uses a linked list of dynamically allocated nodes for unbounded capacity shown in algorithm 9. The producer first attempts to obtain a node from the cache by calling `cache.pop()`, which uses a bounded SPSC circular buffer (buffer from Lamport queue) to recycle previously used nodes. If the cache is empty, it falls back to malloc to allocate a new node. After obtaining a node, the producer writes the data value, sets the next pointer to NULL, then executes a write memory barrier (WMB) to ensure data visibility on weakly ordered memory models like

5. Analyzing existing Wait-Free Data Structures and Algorithms

PowerPC where write-to-write relaxation allows stores to execute out of program order. The producer then updates `tail->next` to link the new node and advances the tail pointer. The consumer checks if `head->next` is non-NULL to determine if data is available, saves the current head node reference, extracts data from `(head->next)->data`, then advances the head pointer to make the data node become the new dummy head. Before returning, the consumer attempts to push the old dummy node to the cache for reuse, only calling `free()` if the cache is full. The dummy head node technique ensures head and tail pointers never point to the same active data node, keeping them in separate cache lines. According to Torquati's measurements, this approach is approximately 6 times slower than bounded SPSC queues when producer and consumer run on different cores, degrading from 11-15ns to 60-120ns per operation due to poor cache locality from pointer chasing and dynamic memory management overhead, even with node caching reducing malloc/free calls [46].

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

```

1: struct Node { void* data; Node* next; }
2: Node* head;                                ▷ Points to dummy node
3: Node* tail;                                ▷ Points to last data node
4: SPSC_Buffer cache;                          ▷ Bounded cache for node recycling
5:
6: function ALLOCNODE
7:   Node*  $n \leftarrow$  NULL
8:   if cache.pop(& $n$ ) then                    ▷ Try cache first
9:     return  $n$ 
10:  end if
11:   $n \leftarrow$  (Node*)malloc(sizeof(Node))
12:  return  $n$ 
13: end function
14:
15: function PUSH(void* data)
16:   Node*  $n \leftarrow$  allocnode()              ▷ Get node from cache or malloc
17:    $n \rightarrow$ data  $\leftarrow$  data
18:    $n \rightarrow$ next  $\leftarrow$  NULL
19:   WMB()                                     ▷ Write Memory Barrier
20:   tail->next  $\leftarrow$   $n$                      ▷ Link new node
21:   tail  $\leftarrow$   $n$                            ▷ Update tail pointer
22:   return true
23: end function
24:
25: function POP(void** data)
26:   if head->next  $\neq$  NULL then              ▷ Check if data available
27:     Node*  $n \leftarrow$  head                    ▷ Save current dummy
28:     *data  $\leftarrow$  (head->next)->data          ▷ Extract data
29:     head  $\leftarrow$  head->next                  ▷ Advance to next node
30:     if !cache.push( $n$ ) then                 ▷ Try to recycle old dummy
31:       free( $n$ )                               ▷ Free if cache full
32:     end if
33:     return true
34:   end if
35:   return false                               ▷ Queue empty
36: end function

```

Unbounded Single Producer Single Consumer

Combines bounded SPSC buffers into a queue using a pool of circular buffers linked as a list. The producer writes to the current buffer until full, then requests a new buffer from the pool via the 'next_w()' method, updates the write pointer, and continues. The consumer reads from its current buffer, and when empty, checks if more buffers exist by comparing the read and write pointers. If buffers differ, it switches to the next buffer obtained via 'next_r()' and releases the empty one back to the pool for recycling. A write memory barrier ensures all data writes complete before buffer transitions, preventing data loss on weakly ordered memory models. [47]

Algorithm 10 Unbounded Single Producer Single Consumer Operations[47]

```

1: function USPSC_PUSH( $q, data$ )
2:   if  $q.buf\_w.full()$  then                                     ▷ Current buffer full
3:      $q.buf\_w \leftarrow q.pool.next\_w()$                            ▷ Get new buffer
4:   end if
5:    $q.buf\_w.push(data)$ 
6:   return true
7: end function
8:
9: function USPSC_POP( $q, data$ )
10:  if  $q.buf\_r.empty()$  then
11:    if  $q.buf\_r = q.buf\_w$  then                                     ▷ Same buffer?
12:      return false                                             ▷ Queue truly empty
13:    end if
14:    if  $q.buf\_r.empty()$  then                                     ▷ Recheck after comparison
15:       $tmp \leftarrow q.pool.next\_r()$ 
16:       $q.pool.release(q.buf\_r)$                                    ▷ Recycle buffer
17:       $q.buf\_r \leftarrow tmp$ 
18:    end if
19:  end if
20:  return  $q.buf\_r.pop(data)$ 
21: end function

```

MultiPush Single Producer Single Consumer

Reduces cache-line thrashing in Lamport's circular buffer by batching multiple elements before insertion. The producer accumulates items in a thread-local array, then writes them to the SPSC buffer in reverse order using the multipush method when the batch fills or flush is called. This backward insertion maintains distance between read and write pointers across cache lines, preventing false sharing. While adding an extra copy per element, the improved cache utilization from reduced coherence traffic more than compensates for this overhead, achieving better performance than standard push operations. [46]

Algorithm 11 MultiPush Single Producer Single Consumer Operations[46]

```

1: function MSPSC_PUSH( $q, data$ )
2:    $q.batch[q.count] \leftarrow data$ 
3:    $q.count \leftarrow q.count + 1$ 
4:   if  $q.count = BATCH\_SIZE$  then
5:     return MULTIPUSH( $q, q.batch, q.count$ )
6:   end if
7:   return true
8: end function
9:
10: function MULTIPUSH( $q, batch, len$ )
11:    $last \leftarrow q.write + len - 1$  ▷ Calculate end position
12:   if  $q.slots[last \bmod q.size] \neq \text{NULL}$  then
13:     return false ▷ Not enough space
14:   end if
15:   for  $i \leftarrow len - 1$  downto 0 do ▷ Reverse order
16:      $q.slots[(q.write + i) \bmod q.size] \leftarrow batch[i]$ 
17:   end for
18:   WMB() ▷ Ensure all writes visible
19:    $q.write \leftarrow (last + 1) \bmod q.size$ 
20:    $q.count \leftarrow 0$  ▷ Reset batch counter
21:   return true
22: end function
23:
24: function FLUSH( $q$ )
25:   if  $q.count > 0$  then
26:     return MULTIPUSH( $q, q.batch, q.count$ )
27:   end if
28:   return true
29: end function

```

Jayantis SPSC Queue

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. Also here multiple approaches are available from different papers. These are the algorithms that found for this case:

- Jayanti Queue: Distributes the global queue across n local queues per producer. Producers obtain timestamps via atomic counter reads, insert [element, timestamp] pairs into their local queues, then propagate front elements up a binary tree of minimum values (propagating means updating values level by level from leaves toward the root). The consumer reads the root to find the earliest timestamp, dequeues from that local queue, and propagates changes back up the tree. Internal nodes maintain minimum timestamps of their subtrees, enabling $O(\log n)$ operations using LL/SC (Since the hardware used for this work was accomplished on x86, LL/SC was substituted with CAS linked with version numbers to avoid ABA). [44]
- Drescher Queue: Uses a linked list with dummy node recycling to achieve wait-freedom. Producers atomically swap the tail pointer via FAS to append nodes, then link the previous tail to the new node. The consumer advances the head pointer and recycles

5. Analyzing existing Wait-Free Data Structures and Algorithms

the dummy node by re-enqueueing it when encountered, ensuring the queue never becomes empty. Instead helping atomic pointer swaps are used. [28]

- Jiffy Queue: Maintains elements in a linked list of fixed-size arrays to minimize memory usage, but still have good execution times. Producers use FAA to atomically claim indices, allocate new buffers via CAS if their index exceeds current capacity, then traverse to their target buffer and write elements. When the consumer finds an empty head cell, it scans forward for completed entries, marking them as handled and folding away fully consumed buffers. The second enqueueer in each buffer preemptively allocates the next segment to reduce contention. [43]
- DQueue: Buffers enqueue requests in thread-local arrays before batch writing to shared segments. Producers obtain indices via FAA, accumulate requests locally, then flush to shared memory when buffers fill, exploiting spatial locality. The consumer helps suspended producers by scanning all local buffers when finding empty cells. Key optimizations include eliminating CAS operations to avoid write buffer drains and helping all producers indiscriminately. [41]

5.2.3. Single Producer Multi Consumer (SPMC)

This case is trickier to implement, since multiple consumers have to be synchronized to not get any double read or losing an item. 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. No benchmark will be performed for this category, since there is no other algorithm to compete with in this category. The algorithm 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:

5. Analyzing existing Wait-Free Data Structures and Algorithms

- 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 `'enqueueers[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 bottom 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'`

5. Analyzing existing Wait-Free Data Structures and Algorithms

- + ‘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’ 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 enqueueers 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>.

Bibliography

- [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.

Bibliography

- [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 $\mu C++$* . 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>.

Bibliography

- [33] A. Kogan and E. Petrank, “Wait-free queues with multiple enqueueers 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/crtturnqueue-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.

Bibliography

- [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.

Bibliography

- [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

2.1. Race condition between two threads, which write to the same shared variable.	5
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.	6
2.3. Deadlock between two processes, which wait for each other to release the needed resources.	6

List of Tables

List of Algorithms

1. Michael and Scott's Lock-Free Queue	9
2. Lazy Lamport Queue [49]	19
3. Lazy Lamport Queue Operations [49]	20
4. Batched Lamport Queue Operations [49]	21
5. FastForward Queue Operations [50]	22
6. Improved FastForward Queue Operations [49]	22
7. Batched Improved FastForward Queue Operations [49]	23
8. B-Queue with Self-Adaptive Backtracking[48]	24
9. Dynamic List-based SPSC Queue (dSPSC) Operations [46]	25
10. Unbounded Single Producer Single Consumer Operations[47]	26
11. MultiPush Single Producer Single Consumer Operations[46]	27

A. Appendix