Exposé: Wait-free Synchronization for Interprocess Communication in Real-Time Systems

Devrim Baran Demir

January 27, 2025

1 Introduction

1.1 Motivation

In modern manufacturing, real-time capable control systems are essential for the effective operation of machinery and equipment. Programming these controllers requires a specialized approach to meet strict temporal requirements. Both the operating system and the control program must satisfy real-time constraints to ensure processes run without delays or interruptions. Typically, real-time capable operating systems and system-level programming languages like C/C++ or Rust are employed to meet these stringent requirements.

Traditional synchronization mechanisms, such as mutexes or semaphores, can be problematic in real-time systems due to their unpredictable wait times and potential for deadlocks. These mechanisms often introduce latency and non-deterministic behavior, which are detrimental to the reliability and performance of real-time applications. [1]–[4]

Consequently, there is a growing interest in **wait-free synchronization algorithms**, which guarantee that every operation completes in a finite number of steps, regardless of the number of competing threads. This predictability is crucial in real-time applications where deterministic behavior is paramount. Wait-free algorithms eliminate the possibility of indefinite blocking, ensuring that high-priority tasks are not starved by lower-priority ones. [1], [5]

The Rust programming language offers a promising foundation for implementing such synchronization mechanisms, thanks to its stringent type system and memory safety guarantees. Rust's ownership model and compile-time checks prevent data races and ensure thread safety without incurring the runtime overhead associated with traditional

locking mechanisms. Furthermore, Rust's performance characteristics make it well-suited for real-time systems that demand low latency and high throughput. [6], [7]

Interprocess communication (IPC) is a critical component in real-time systems, enabling efficient data exchange between processes. Efficient and safe IPC mechanisms are necessary to maintain the integrity and performance of the system. Implementing wait-free data structures for IPC can significantly enhance the system's responsiveness and reliability, making it better suited for real-time applications. [8]–[11]

This project aims to identify, implement, compare, and validate suitable wait-free data structures for interprocess communication in real-time systems using Rust. By leveraging Rust's safety and performance features, the project seeks to develop robust IPC mechanisms that meet the stringent requirements of real-time applications.

The main motivations for this work include:

- **Enhancing Real-Time Performance**: Developing synchronization mechanisms that minimize latency and ensure predictable execution times.
- **Improving System Reliability**: Eliminating potential deadlocks and race conditions through wait-free algorithms.
- **Leveraging Modern Programming Languages**: Utilizing Rust's advanced features to create efficient and safe concurrent data structures.
- **Contributing to Real-Time Systems Research**: Providing empirical data and validated implementations that can be utilized in industrial applications.

Thus, this work addresses the critical need for efficient synchronization in real-time systems, leveraging wait-free algorithms and the Rust programming language to develop advanced IPC solutions.

1.2 Objectives

The primary objective of this work is to **identify and implement wait-free data structures** suitable for **interprocess communication** in **real-time systems**. Specifically, the goals are as follows:

- **Identify** appropriate wait-free data structures for IPC, focusing on **Single-Producer Single-Consumer** (**SPSC**), **Multiple-Producer Multiple-Consumer** (**MPMC**) **queues** and mixed queues.
- **Implement** these data structures in the Rust programming language, ensuring they meet real-time requirements. Based on this develop an IPC solution using **shared memory** that leverages the selected wait-free data structures.
- **Compare and analyze** the implemented data structures in terms of **performance**, **scalability**, and **real-time compatibility**.

- **Validate** the data structures through **empirical testing** and **benchmarking** to confirm their suitability for real-time systems.
- **Identify** the most effective algorithms for IPC in real-time systems based on the results obtained.

1.3 Methodology

The methodology to achieve the aforementioned objectives is structured into several phases:

1. **Literature Review and Analysis**: Investigate the current state of the art in wait-free synchronization and IPC in real-time systems. Identify relevant research papers and existing implementations.
2. **Selection of Data Structures**: Based on the literature review, select suitable wait-free data structures (SPSC and MPMC queues) appropriate for IPC in real-time systems.
3. **Implementation in Rust**: Develop the selected data structures in Rust, ensuring proper memory management and thread safety. Based on this we will create an IPC mechanism using shared memory that incorporates the validated wait-free data structures.
4. **Benchmarking and Comparison**: Conduct performance tests and compare the data structures based on throughput, latency, and scalability under various workloads.
5. **Validation**: Analyze the test results to validate the suitability of the data structures for real-time systems. Identify the most effective algorithms for IPC in real-time applications.
6. **Documentation and Final Report**: Compile the research findings and implementation results into a cohesive thesis document. Review, revise, and finalize the thesis for submission.

This approach is structured and scheduled as depicted in Figure 1.

2 State of the Art

Synchronization in parallel systems are a critical aspect of ensuring data consistency and system reliability. Traditional synchronization mechanisms, such as mutexes, spinlocks, and semaphores, have been widely used but often fall short in real-time environments due to their unpredictable latency and potential for causing deadlocks [1], [2].

Wait-free synchronization algorithms offer a compelling alternative by guaranteeing that every thread can complete its operations in a finite number of steps, thereby

eliminating the issues of indefinite blocking and ensuring high predictability. These algorithms are especially beneficial in real-time systems where timing guarantees are essential [1], [5].

2.1 Wait-Free Queues

Wait-free queues are fundamental for interprocess communication, enabling safe and efficient data exchange between processes without the need for locking mechanisms. Two primary types of wait-free queues are:

2.1.1 Single-Producer Single-Consumer (SPSC) Queues

SPSC queues are designed for scenarios where there is exactly one producer and one consumer. Their simplicity allows for highly efficient implementations with minimal overhead. These queues are typically implemented as ring buffers, where the producer and consumer maintain separate indices for writing and reading, respectively.

Advantages:

- **Simplicity and Efficiency**: Minimal synchronization overhead due to single producer and consumer.
- **True Wait-Freedom**: Each operation completes in a fixed number of steps without contention.
- **Predictable Latency**: Constant-time operations make them suitable for real-time applications.

Disadvantages:

- **Limited to Single Producer and Consumer**: Not suitable for multiple producers or consumers without modifications.
- **Fixed Capacity**: Typically implemented with a bounded buffer size, requiring handling of full or empty states.

[12], [13]

2.1.2 Multiple-Producer Multiple-Consumer (MPMC) Queues

MPMC queues on the other hand support multiple producers and consumers. They are more complex to implement due to the need to handle concurrent access by multiple threads, often relying on atomic operations and careful memory management to ensure consistency.

Advantages:

- **High Concurrency**: Supports multiple producers and consumers, making them suitable for scalable applications.

- **Non-Blocking**: Typically lock-free, ensuring system-wide progress even under high contention.

Disadvantages:

- **Increased Complexity**: More sophisticated algorithms are required to manage concurrent access.
- **Potential for Contention**: Higher contention can lead to increased latency and reduced throughput.
- **Memory Reclamation Challenges**: Safe memory management in lock-free structures can be complex, especially in languages without garbage collection.

[14], [8]

2.1.3 Relevant Research

3 Work Package and Time Plan

The work plan outlined in Figure 1 spans a period of approximately six months, structured into six main phases:

1. **Literature Review and Analysis (Weeks 1-4)**: Conduct an in-depth review of existing research on wait-free synchronization and IPC in real-time systems. Identify key algorithms, data structures, and implementation strategies.
2. **Exams (Weeks 3-6)**: Complete any ongoing exams that may overlap with master thesis timeline.
3. **Design and Selection (Weeks 5-8)**: Based on the literature review, select the most suitable wait-free data structures (SPSC and MPMC queues) for implementation. Design the overall architecture and implementation strategy.
4. **Implementation (Weeks 9-14)**: Develop the selected data structures in Rust. Implement the SPSC queue first, followed by the more complex MPMC queue, ensuring adherence to wait-free principles.
5. **Testing and Benchmarking (Weeks 15-20)**: Create comprehensive test cases to validate the correctness and performance of the implemented data structures. Conduct benchmarking to measure throughput, latency, and scalability under various workloads.
6. **IPC Solution Development (Weeks 21-24)**: Design and implement an IPC mechanism using shared memory that leverages the validated wait-free data structures. Ensure seamless integration and real-time compatibility.

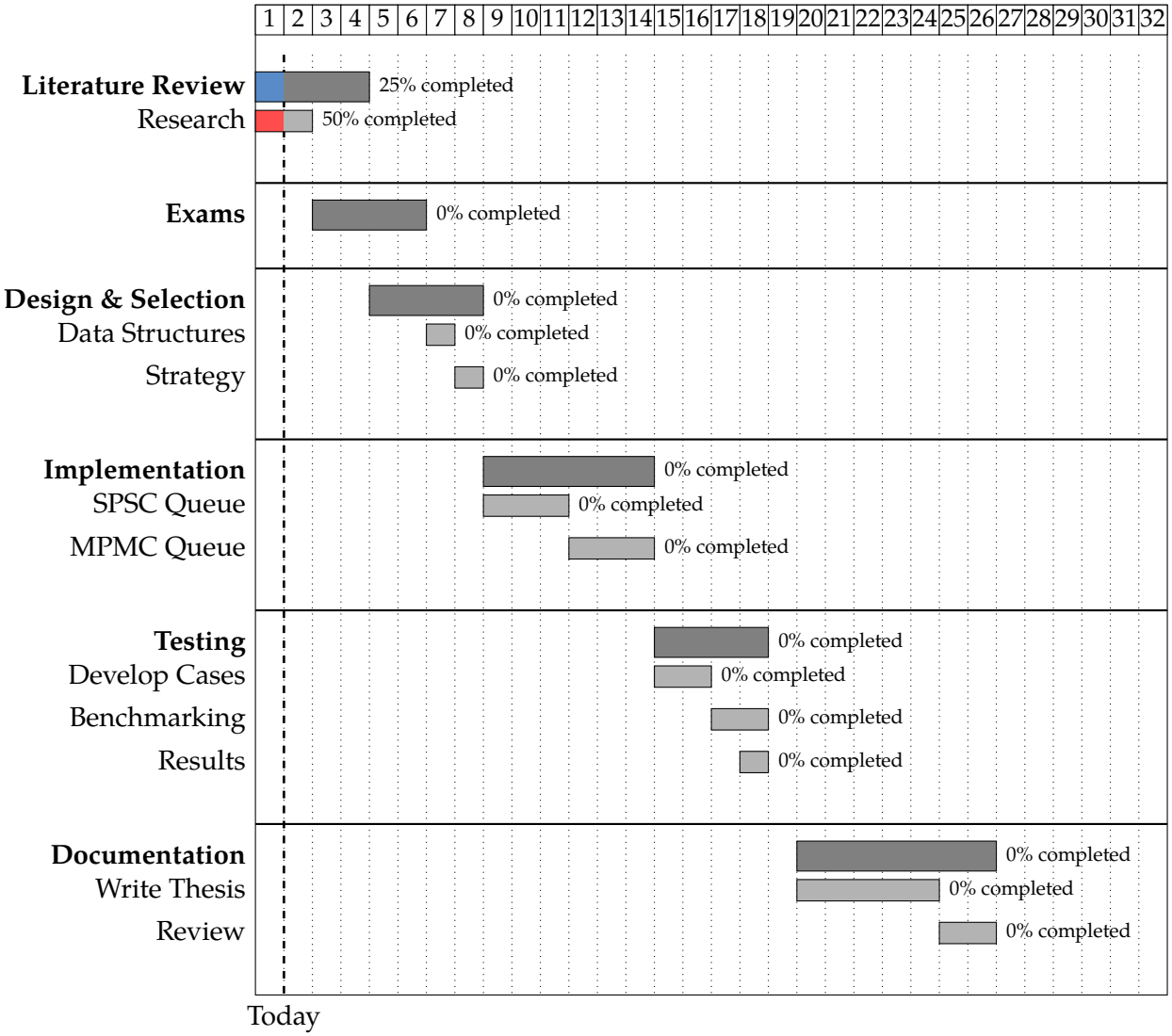

Figure 1: Gantt Chart of the Planned Workflow

7. **Documentation and Finalization (Weeks 25-26)**: Compile the research findings and implementation results into a cohesive thesis document. Review, revise, and finalize the thesis for submission.

This structured approach ensures a systematic progression from theoretical research to practical implementation and validation, culminating in a comprehensive thesis that addresses the research objectives.

4 References

References

[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] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming, Revised Reprint*, 1st. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2012.

[3] 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.

[4] O. Kode and T. Oyemade, *Analysis of synchronization mechanisms in operating systems*, Sep. 2024. DOI: 10.48550/arXiv.2409.11271.

[5] 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.

[6] 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*, L. Yuan, S. Yang, R. Li, E. Kanoulas, and X. Zhao, Eds., Singapore: Springer Nature Singapore, 2023, pp. 440–451.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] M. F. Dolz, D. del Rio Astorga, J. Fernández, J. D. García, F. García-Carballeira, M. Danelutto, and M. Torquati, "Embedding semantics of the single-producer/single-consumer lock-free queue into a race detection tool", in *Proceedings of the 7th International Workshop on Programming Models and Applications for Multicores and Manycores*, ser. PMAM'16, Barcelona, Spain: Association for Computing Machinery, 2016, pp. 20–29. DOI: 10.1145/2883404.2883406. [Online]. Available: https://doi.org/10.1145/2883404.2883406.

[13] 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.

[14] A. Gidenstam, H. Sundell, and P. Tsigas, "Cache-aware lock-free queues for multiple producers/consumers and weak memory consistency", in *Proceedings of the 14th International Conference on Principles of Distributed Systems*, ser. OPODIS'10, Tozeur, Tunisia: Springer-Verlag, 2010, pp. 302–317.