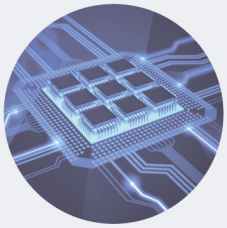
INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO
MESTRADO EM ENGENHARIA DE SISTEMAS COMPUTACIONAIS CRÍTICOS



Real-Time Parallel Programming in Rust

HUGO MARTINS COELHO DA SILVA
outubro de 2025

ISEP INSTITUTO SUPERIOR DE ENGENHARIA DO PORTO

Real-Time Parallel Programming in Rust

Hugo Martins Coelho da Silva

Dissertation submitted in partial fulfilment of the requirements for the Master's degree in Critical Computing Systems Engineering

Supervisor: Tiago Diogo Ribeiro de Carvalho

Co-supervisor: Luís Miguel Rosário da Silva Pinho

Evaluation Committee:

President:

António Barros, Instituto Superior de Engenharia do Porto

Members:

João Bispo, Faculdade de Engenharia da Universidade do Porto
Tiago Carvalho, Instituto Superior de Engenharia do Porto

Porto, September 22, 2025

Statement of Integrity

I hereby declare having conducted this academic work with integrity. I have not plagiarised or applied any form of undue use of information or falsification of results along the process leading to its elaboration.

I declare that the work presented in this document is original and my own, and has not previously been used for any other purpose.

I further declare that I have fully followed the Code of Good Practices and Conduct of the Polytechnic Institute of Porto.

ISEP, Porto, September 22, 2025

Resumo

O desenvolvimento de sistemas de tempo real é um dos temas com maior relevância dos últimos anos. A quantidade de esforço que é investido para o melhoramento destes sistemas tem sido elevada e com tendências de aumentar. Isto deve-se principalmente às exigências da atualidade no que diz respeito a tecnologia. O número de sistemas críticos tem aumentado cada vez mais, o que resultou num grande aumento de problemas que exigem soluções com maior rapidez e previsibilidade temporal. De forma a conseguir aumentar a performance destes sistemas, a utilização de computadores com mais do que um processador foi a solução que mais resultados obteve. No entanto, esta abordagem também alcançou o seu limite devido às restrições impostas pela programação sequencial. Como tal, as soluções mais recentes têm-se apoiado no uso de programação em paralelo.

Esta dissertação apresenta o design, a implementação e a avaliação de uma *framework* personalizada para suportar a execução paralela com requisitos de tempo real, utilizando a linguagem de programação Rust. Após uma análise inicial entre Rust e o modelo OpenMP em C, foi selecionado o Rust pelas suas garantias de segurança de memória, pelo seu ecossistema em expansão e pela sua adequação ao desenvolvimento de sistemas concorrentes e de baixo nível. O principal objetivo do projeto foi avaliar se seria possível alcançar um comportamento próximo de um sistema em tempo real através de um sistema baseado em Rust, recorrendo ao controlo detalhado da gestão de *threads* e tarefas.

Para isso, foi desenvolvida uma biblioteca que permite esse controlo, dando ao utilizador diferentes formas de criação de *threads* e distribuição de tarefas. Foram realizados diversos testes de desempenho para avaliar métricas como a duração da execução, a latência na distribuição de tarefas e o equilíbrio de carga entre as *threads*. Os resultados mostraram que, com uma configuração e design cuidadosos, é possível obter uma execução paralela estável e eficiente, com um comportamento temporal que se aproxima das exigências dos sistemas em tempo real, mesmo em ambientes que não são nativamente de tempo real, como é o caso do sistema operativo Linux.

Apesar dos resultados encorajadores, o trabalho também evidenciou algumas limitações, nomeadamente as restrições impostas pelo escalonador do Linux e a ausência de mecanismos mais precisos para a gestão de *deadlines*. Como possíveis melhorias futuras, propõe-se a extensão da interface de escalonamento com novos algoritmos, a integração com sistemas operativos de tempo real e o desenvolvimento em arquiteturas embarcadas, como por exemplo as plataformas baseadas em RISC-V.

Palavras-chave: Rust, Sistemas de Tempo Real, Programação em paralelo, Gestão de *Threads*.

Abstract

The development of real-time systems has gained increasing relevance in recent years, driven by the growing presence of critical applications in areas like robotics, automotive systems, and the Internet of Things. These systems require correctness and strict timing guarantees, which the increased complexity of modern systems has made more difficult. One common solution to increase performance has been the use of multi-core processors. However, the limits of sequential programming soon became clear, making parallelism a necessary step forward.

This thesis presents the design, implementation, and evaluation of a custom framework for real-time parallelism. After an initial comparison between Rust and the OpenMP model in C, Rust was selected for its safety features, low-level control, and growing ecosystem. The goal was to test whether real-time behaviour could be reliably achieved in a Rust-based environment by taking control over thread and job management.

A custom library was built and tested measuring metrics such as execution time, dispatch latency, and task distribution. Results show that, with careful design, efficient and predictable performance is possible even on a non-real-time OS like Linux. Future improvements include deeper scheduler integration, integration with real-time operating system, and deployment on embedded platforms such as RISC-V.

Keywords: Rust, Real-Time Systems, Parallel Programming, Thread Management.

Table of Contents

List of Figures

List of Tables

Acronyms and Symbols

List of Acronyms

API	Application Programming Interface
CPU	Central Processing Unit
FIFO	First In First Out
ISEP	*Instituto Superior de Engenharia do Porto*
MPI	Message-Passing Interface
OpenMP	Open Multi-Processing
OS	Operating System
PoC	Proof of Concept
RR	Round Robin
RTOS	Real Time Operating System
WCET	Worst-Case Execution Time

List of Symbols

T	Period

1 Introduction

Real-time systems research is a challenging, growing field of engineering with many uses from the automotive industry to robotics and internet of things. These systems differ from the rest because of the importance placed on the timed delivery of the product, limited by the deadline or time frame specified by the application. For a real-time system to be considered flawless, it must obey to the accuracy and time restrictions placed on its output. The predictability that needs to be assured for these systems to be successful is what characterizes real-time systems, together with its efficiency.

The development of real-time systems has been a topic of discussion and a subject of work for a long time, reaching what is thought to be a ceiling when considering the programmability to be limited for one core or processor. For this, the need of new technologies capable of supporting concurrent tasks is needed both in hardware and software. Due to its cheapness, multi-processor chips have been the most common solution for increasing the computational and processing power of the system. This brings an even bigger challenge for the software side of this solution as sequential programming does not scale well with multi-processor systems, thus not making full use of its potential. The expansion of real-time systems to fields with close human-machine interactions also made this problem harder, bringing another constrains as the time requirements are even more critical (Burns and Wellings, 2009; Pinho, Nélis, *et al.*, 2015).

While multiprocessor systems have become more accessible, the ability to exploit their full potential in real-time applications remains limited. Real-time software developers often face significant difficulties in ensuring that parallel tasks do not interfere with each other's timing guarantees. This becomes especially problematic in environments where precise task coordination and bounded response times are mandatory. Common parallel programming frameworks prioritize throughput over predictability, which can lead to missed deadlines, or priority inversion which are issues that are unacceptable in hard real-time systems. Therefore, traditional models fall short when predictability is more important than raw performance (Cucu-Grosjean and Goossens, 2009; Blue Goat Cyber, 2025).

In the last decades there has been a large commitment and investment in application performance, and it has proven to be very successful. These were focused mainly in partitioning the work between processors, thus meaning that a task was being solved in each node, allowing for bigger and more complex problems to be solved faster. On the other hand, each task was still being ran sequentially, giving it more room for improvements with the introduction of parallel programming. (Hwu, 2014)

Another critical problem is the lack of predictable timing behaviour in most general-purpose operating systems and programming frameworks. For instance, the default thread schedulers in modern operating systems are often designed with fairness or an average case performance in mind rather than strict timing constraints. In such contexts, developers must resort to complex and error prone workarounds, such as pinning threads to cores, adjusting real-time priorities using system calls, or relying on platform-specific APIs. This increases the complexity of the development cycle and reduces portability across systems (Cucu-Grosjean and Goossens, 2009; Cerqueira, Gujarati and Brandenburg, 2014; Lee *et al.*, 2022; Red Hat, 2025).

These approaches were not introduced earlier due to the same challenges to have parallel programming in real-time systems. That is, the time at which a processor can access memory shared data is much lower than the time that it takes for arithmetic operations to be performed in the processor. If we want real-time systems to have the lowest response time possible, then the parallelism can be objectively opposed to that. This challenge has been addressed since then by many researchers, with the goal to create more tools that enable scalable parallel algorithms (Cucu-Grosjean and Goossens, 2009; Cerqueira, Gujarati and Brandenburg, 2014; Lee *et al.*, 2022; Red Hat, 2025). This work shares the same goal, focusing more directly on real-time systems.

Bridging this gap requires not only better runtime behaviour but also tooling that gives developers explicit control over low-level execution parameters such as CPU affinity, scheduling policy, task periodicity, and priority levels. Despite the abundance of libraries for concurrent and parallel execution, few offer this degree of configurability, especially in a safe and ergonomic form suitable for real-time system developers. In other words, the current ecosystem lacks solutions that combine predictability, configurability, and safety in one package (Cucu-Grosjean and Goossens, 2009; Cerqueira, Gujarati and Brandenburg, 2014; Ngo, 2020; Lee *et al.*, 2022).

Consequently, two technologies were initially considered for the development of a proof of concept for real-time parallel programming: the Rust programming language and the OpenMP (Open Multi-Processing) tasking model.

The Rust programming language (Rust, 2023) excels from other languages because of its memory ownership model that provides significant safety feature normally inexistent in the programming languages. This is even more useful when taking in consideration critical systems, making Rust a very plausible replacement for languages like C or ADA. However, there is still work to be done for the development of real-time systems with Rust as it does not provide the capabilities needed for it such as control of scheduling and support for real-time attributes like periods and deadlines.

OpenMP (OpenMP, 2023) is the standardized application programming interface (API) for shared-memory parallel programming, being one of the most used models for high performance computing. Its value has been increasing such as its use is increasing in embedded systems as well as in applications used for critical computing systems like video processing in autonomous driving. OpenMP already enables the use of real-time models, but it still lacks the support for real-time scheduling properties such as deadlines or worst-case execution time (WCET). Contrary to the case in Rust, a proposal (Serrano, Royuela and Quiñones, 2018) has recently been made to define an OpenMP profile for critical real-time systems. Although it already covers a lot of groundwork,

further research is possible and encouraged particularly on the specification of real-time properties and how to support the integration of multiple parallel real-time activities in the same application.

Upon further investigation and analysis, both technologies were seen as a valuable option for real-time parallel programming, but a decision was made to advance the work utilizing the Rust approach. This was decided on a few factors such as the memory management and safety protocol of Rust, which implementation in real time systems is very interesting and innovative. Another point that attracted this choice was the fact that Rust provides high level abstractions without decreasing its performance which is many times compared to Cs as equal. This, together with some personal preference for the language in itself and the fact that it is more recent, lead to the choice of Rust as the language to be used in the dissertation project.

After having made the decision to advance the work using Rust, a library was developed to tackle the specific needs of parallel execution within a real-time context. This library aimed to provide a more predictable and controllable execution model, focused on minimizing complexity for the user. The implementation prioritized aspects such as thread creation, job distribution, and execution timing, which are essential for evaluating whether a system can meet real-time demands. The library served as both a proof of concept and a testing ground, making it possible to collect detailed runtime data and analyse performance and predictability.

To evaluate the effectiveness of the proposed approach, a series of experiments were conducted, each focusing on different aspects of the execution model, ranging from simple thread creation benchmarks to more complex job scheduling scenarios. The experiments were designed to simulate realistic real-time behaviour, and while the setup did not aim to meet hard real-time constraints, it helped identify both the strengths and limitations of the current approach. The system showed generally consistent timing behaviour, low overhead for thread and job creation, and a balanced task distribution, but also revealed areas where further development could improve precision and configurability.

1.1 Objectives

The goal of this work is to design and develop a proof of concept of an API for real-time parallel programming. This is done by analysing the requirements for programming frameworks for parallel real-time systems together with the study of previous approaches and research made in the subject. Based on this analysis, a study of the state-of-art was done and analysed on existent real-time parallel programming models. As the thesis objectives are to design a framework for real-time parallel support in either OpenMP or Rust, the analyses of existent Rust libraries for concurrent and parallel development, as well as existent proposals for supporting real-time in OpenMP was also done.

Having chosen Rust as framework used for this proof of concept, an initial idea of the specification and design of a model to support real-time programming in multicore platforms was to be achieved, focusing on the framework of choice. The operating system of choice is Linux (Linux, 2023) so a study of its scheduler and task manager was also expected.

Apart from this, deep research on low level parallel programming was to be conducted, going through existing approaches in C and Java languages as Rust uses them as its base of inspiration, and, in case of C, can even interact with it. Also, a similar kind of research is intended for Rust and existing implementations of parallel programming within it.

This dissertation project comes also as an answer to many questions regarding parallel programming. One of these being the possibility of achieving a solution in Rust for concurrent programming and parallelism in real time. If so, could it allow the configuration of the threads used, or even, if using a thread pool, could the thread pool have threads with different attributes but still be managed together? Would it also be possible to create tasks that just inherit the parent thread's attributes? Can we achieve hard real time with this approach?

This set of questions are the main reason and motivation for the development of this dissertation, having the plan be the ability to answer them positively or setting responses able to invite other further investigations in the field.

1.2 Structure of the Thesis

This thesis is divided in 6 main sections, these being the Introduction, State of the Art, Real-Time Parallel Programming API for Rust, The Design of *rt_lib*, Results and Conclusion. The introduction gives an overview of the problem that is being challenged and defines the goals, motivation and objectives of the work.

The State of the Art starts with a deeper dive on the major concepts present in this thesis, starting with real-time systems, parallel programming, and following with the Rust programming language and the OpenMP tasking model. It then goes on the study and evaluation of previous works done in real-time systems with parallel programming, focusing initially on the proposal for the OpenMP framework followed by existent approaches of parallel programming with Rust on real-time systems and by the existing proposals for real-time parallel programming with Ada. After this, it goes on a deeper analysis of the existent proposals and possible approaches, focusing on Rust and other lower level languages that inspired Rust, like C and Java. Finally, there is a critical analysis of the existent proposals and possible approaches for the work, together with the existing models of parallelism in C and Rust.

The Real-Time Parallel Programming API for Rust presents the design achieved during the development of the proof of concept, starting with presenting the basis from where we decided to create our library from. Then, with it, a template that would present the initial ideas gathered and wanted from it and an initial dive into how the library is organized internally.

The Design of *rt_lib* dives deeper into the library methods and implementations, explaining thoroughly what functionalities it offers and to use them. It then jumps in complexity by diving into how the library works and manages its threads and thread pools within the library.

Then, in the Results section, a group of examples are presented, showcasing the simplicity we were able to maintain while still creating real time programs with fine-tuned thread management. With these examples adapted to it, results were then gathered through performance testing. These results are then presented and analysed.

Finally, in the Conclusion we dive through all the thesis contents. A final balance of the work is made, highlighting the main aspects of what was done, formulating critical judgments, both positive and negative, about what was achieved. In this case, suggestions for future work were also identified and proposed.

2 State of the Art

This thesis focuses on the development of a proof of concept for real-time parallel programming. At the beginning, a decision was yet to be made on the framework, it being either a C framework using OpenMP's API or a Rust framework. As such, we focused our state-of-the-art analysis on the use of parallel programming for real-time applications with either OpenMP or Rust. Even so, real-time systems are not restricted to these two options for parallel programming. As such, a study of other models in languages like Ada, a commonly used framework for real-time systems, is favourable as it can be useful and give insights about potential approaches in the development of this thesis. This kind of analysis also gives us a better insight on what are the common desires and methods expected from other programmes that work with parallel solutions.

With the use of these analyses and personal preferences, the decision was made to follow-up the work using Rust as the framework of choice. Due to this decision, a deeper analysis is also made on the existing parallel programming and concurrency models for low-level languages, focusing on Rust.

In the end, a summary complemented with a critical analysis of the state of the art is carried out, evaluating the possible approaches of the thesis and highlighting the framework initially selected for the development of the thesis.

2.1 Concepts

This section introduces the topics in which this thesis is based, giving a perception about the importance and need for more studies of these subjects. This section also presents the technologies that were initially proposed to be used in the thesis.

2.1.1 Real-Time Systems

A system known as a "real-time system" is one where the time needed to compute and produce the results is as important as the accuracy and exactitude of the results computed. These systems are distinguished by their ability to *process* data and respond to external inputs in a predictable manner that meets the deadlines set. A deadline is the maximum amount of time that the system must complete a task which frequently corresponds to its period when talking about periodic tasks (Kopetz and Steiner, 2022).

Real-time systems are often divided in two real-time system types, this being soft and hard real-time systems. Soft real-time systems are those in which the importance of meeting a deadline is low, this meaning that there are still deadlines but it can tolerate delays and deadline missing as it will not affect the correctness of the system. Hard real-time systems, on the other hand, cannot

tolerate any deadline misses as it means catastrophic ends to the system, for example, failing to meet a deadline could lead to serious damages or even loss of lives. Some examples of hard real-time systems are nuclear power plants or flight control systems in which the meeting of a deadline is imperative. As for soft real-time systems, this could be data acquisition systems, online gaming or multimedia systems for example. There still is another type of real-time systems which is designated as firm real-time system. This type of system is somewhere in the middle of the two other types, where the missing of a deadline is not critical but deadline misses can frequently lead to failure of the system and the late delivery of such data is not beneficial. An example of this type of system is video conferencing where the failure of meeting occasional deadlines will not affect the system too much, only causing delays in the call, but if it happens frequently will degrade the performance and correctness of the system (Burns and Wellings, 2009).

These systems are designed to handle different types of events that differ in their time constraints and in the way they arise from the physical impact in the controlling system's environment. Periodic tasks are events that occur every T units, with T being its period, such as the arrival of a new frame in a video conferencing system. These tasks typically have their deadline being equal to its period, but it can also be less or greater than it. An aperiodic task is one that is activated at unpredictable times like in every system in which a task is launched by an external input by its operator. A sporadic task is similar to an aperiodic one in a sense that it does not have a period, but instead has a time constraint in which there is a time interval between task activations (Stankovic, 1996).

2.1.2 Parallel Programming

In the current times, the investment made for maximum out computational power has provided very good and fast evolutions. Due to this, the use of computers was spread along almost all fields of investigation and with it, a big number of problems that could be solved by its use appeared. Its potential intrigued a lot of investigators, which lead to a growing need for the evolution of the computers' performance to keep its pace. With this technology beginning to reach its full potential, a lot of proposals were made to enable the progress of such technology. One of them is the Amdahl's law which describes a formula that states that the potential increase in the performance of a programming system is limited by the fraction of time that the program must execute serially. This law is often used in parallel programming to predict the speedup when using multiple processors, thus, showing that a program must have as much parallel code as possible for it to have the maximum possible improvement in its performance (Amdahl, 2013). As such, parallel programming is an increasingly important computing technique as it takes advantage of multiple cores or processors by performing tasks concurrently (Brawer, 1989).

Parallel programming works by dividing the work among the available cores or processors, a process known as task partitioning, which is a vital stage in the development of parallel software. Another important aspect of parallel programming is synchronization, which ensures that the various sections of the program work together correctly. This needs to happen in at least one of two different ways, task parallelization and data parallelization, depending on the task at hand. Task parallelization works by sharing the workload for the various cores, each one doing its designed tasks. On the other hand, data parallelization works by partitioning the data used for the

problem among the different cores and every core working separately with part of the shared data. The latter option can be accomplished through the use of locks, semaphores, or other synchronization primitives in cases where critical zones can appear. A critical zone is a block of code that is using shared data that can be also accessed at the same time by other cores, causing a race condition that can potentially endanger the correctness of the system (Pacheco, 2011).

One common method of parallel programming is to use threads, which are lightweight and independent, and can be scheduled to run simultaneously on different cores or processors. Threads can be managed by the operating system or by a library, also being able to communicate with each other using shared memory or message passing. Message passing is often used in distributed systems and it allows different processes or threads to communicate with each other by sending and receiving messages. A distributed system is one where the processes or threads are running on different machines or cores and each of them has its own private memory thus having to communicate with each other by sending messages across a network. This is contrary to a shared-memory system where each core can have access to a shared memory location (Pacheco, 2011).

Parallel programming can also be accomplished through the use of various programming models such as OpenMP, MPI (Message-Passing Interface), and *pthreads*. OpenMP is a programming model for shared memory in which each core can potentially have access to all available memory. OpenMP is an advanced API that allows developers to specify parallel regions of code, via annotations and API calls, that can be executed on different cores or processors at the same time, in some cases without any specification of the behaviour of each thread, leaving this work for the compiler and the OpenMP run-time system. MPI is a library of functions that can be used in par with programming languages like C and C++. It is a programming model designed for distributed system that allows developers to send and receive messages between different processes or threads. Similar to OpenMP, *pthreads* is a shared memory programming model that allows for the use of shared memory between cores and processes. On the other hand, when compared to OpenMP, this model needs more care as all the specifications need to be done by the programmer when initiating each thread and its shared memory spaces (Pacheco, 1997, 2011).

2.1.3 Rust Programming Language

The world of low-level system programming has been dominated by C and C++ for a long time and it still is, having recently undergone modernizations. This is the case majorly because of the legacy code written in these languages, and it being used by many organizations that still use and maintain the code. This, accompanied by its high performance, portability and big community are the main reasons C and C++ have yet not been replaced as the go to for low-level programming. Despite the fact that they are a widely used programming language, these languages lack many safety guarantees, leaving the burden of ensuring memory safety onto the programmer. This adversity is what usually leads to security vulnerabilities and is one of the most tiresome tasks in programming in these languages. Rust is a programming language built with the purpose of replacing languages like C by covering and fixing most of the memory safety issues in these languages (Jung, 2020).

Rust was created by an employee at Mozilla and soon bought by the company. It was designed to support parallelism by taking full advantage of modern hardware. For this, one of Rust's main focuses is memory safety but this by itself is not sufficient to justify its growing relevance in the field. Rust also invested in its performance by adapting a C++ approach where there is no need for a garbage collector as all Rust types can be allocated in the stack. Other advantages of the Rust language come from other features such as its ecosystem. Cargo is a build system and package manager tool used to facilitate almost all possible needs for the building of a software project, making it one of the major tools of the Rust ecosystem. This tool is used for dependency managing in downloading and compiling, does all the build management including debugging and release, can have unit and integration testing, and much more. Additionally, Cargo has the ability to install other tools that work with it, and this include a big number of existent libraries, some of which try to use Rust's support for parallel programming at its full (Matsakis and Klock, 2014; Bugden and Alahmar, 2022).

2.1.4 OpenMP

OpenMP (OpenMP, 2023) is the standardized application programming interface (API) for shared-memory parallel programming, which is capable of efficiently utilizing highly parallel and heterogeneous embedded architectures and offers programmability and portability benefits. The OpenMP API aim is to facilitate parallel programming in shared-memory systems and to do this, OpenMP has a lot of features for parallel programming. This framework works in a fork/join method where fork is the means of starting a parallel construct where a thread or team of threads is created form the main thread, only terminating when, and if, a join is present in the main thread for each other thread created before. When a main thread does not wait for its thread "children" by using join, the threads will terminate when the main thread terminates, not guaranteeing that all the threads of the program will run their entire program. OpenMP also allows the programmer to choose between two options of parallelization, this being, thread-based or task-based, both working in similarly ways but managing their memory differently. The synchronization of tasks differs from the threads in a way that it occurs based on task priorities and, *taskwait* and *taskgroup* directives.

When it comes to the shared-memory characteristics within OpenMP, it offers three different memory spaces available for each thread. The first is the memory where a shared space resides, and all its data can be accessed by all threads. The second one is a temporary view of memory for each thread. Lastly, each thread has a *threadprivate* memory where each thread can access its own memory that is not available for the other threads. Also, OpenMP is supported by a wide range of compilers, including GCC, LLVM, and Intel C++ Compiler, which includes some of the most widely used programming languages for real-time systems in C and C++. These are some of the reasons that make OpenMP a widely available and portable option for parallel programming, which leads to high potential use for real-time systems (Ayguade *et al.*, 2009).

2.2 Existing Work for Real-Time Parallel Systems

Having established the fundamental concepts for this proof of concept, this section now transitions to an exploration of existing research in the field. Real-time systems have been the subject of extensive study, resulting in a diverse range of approaches to implement them with parallel programming. In this section, we focus more deeply on OpenMP since this was one of the two frameworks chosen as a possibility for the thesis. Based on this and having chosen Rust as the framework of choice for this project, we dive into other approaches in low-level languages, such as Rust itself and ADA, to gather what was already implemented and what are the key ideas that can be the used for the development of this proof of concept.

2.2.1 Critical Real-Time Systems with OpenMP

The demand for parallel execution in critical real-time systems is on the rise to meet the performance needs of advanced technologies such as autonomous driving and unmanned aerial vehicles and one solution to this is OpenMP.

In the work by Serrano et al. (Serrano, Royuela and Quiñones, 2018) an examination of the use of OpenMP to implement critical real-time systems is carried. This work focuses on the design implications and scheduling decisions necessary to efficiently exploit fine-grain parallelism within real-time tasks and concurrency among them, while also ensuring that the timing behaviour meets current real-time practices. The major challenges tackled in this paper are the efficient exploitation of the parallelism capacities of OpenMP within and among real-time tasks, and how to handle the recurrence of real-time tasks.

To do this, a study was carried to describe how to use OpenMP to support a real-time scheduler. The first problem found were the task priorities. It was concluded as having the need of the development of an OpenMP task scheduler in which the priority clause would determine the schedule behaviour for real-time systems as this is not guaranteed by the OpenMP specification, version 4.5. Another problem for the support of real-time systems was the preemption strategies required. As far as this goes, it is stated that OpenMP supports non-preemptive and limited preemptive strategies, meaning that it supports two of the three types of pre-emption strategies defined on the real-time scheduling theory, only missing the full-preemptive strategy, which is not desirable as it carries high preemption overheads that can contribute negatively to the predictability of the system. Finally, the allocation and migration of real-time tasks is the other main point that helps evaluating real-time schedulers. In regards with allocation strategies, OpenMP already has the resources needed for dynamic allocation and has a static allocation strategy proposed for it too. It is also stated in this study that, utilizing the tied and untied tasking model of OpenMP, it is possible to have partitioned and global scheduling algorithms, respectively.

For the model proposed in this study, OpenMP tasks are executed by only one group of threads which enabled the opportunity to have the real-time scheduler controlling all the OpenMP tasks over the threads. This is important as the majority of schedulers for real-time systems assume a

direct mapping between the tasks and the cores they are assigned to, and OpenMP provides the tools for implementing this behaviour.

It is then concluded that, although it has the tools to build a critical rea-time system, the OpenMP runtime is not yet ready to develop or execute one. This was tested using both GCC 8.1 and Nanos++ runtimes and it is stated that, even if not having the full capacity for it, “Nanos++ already implements some of the fundamental features needed by critical real-time systems” (Serrano, Royuela and Quiñones, 2018) which is not the case for GCC 8.1. This leaves open the opportunity for direct improvement of the design, but it also left the study of different approaches such as an event-driven execution model open as it is not directly supported by OpenMP. Other opportunities for the use of parallel programming with OpenMP could involve the use of the LLVM compiler instead of GCC 8.1.

2.2.2 Real-time parallelism in Rust

Real-time systems have been around for a very long time now and its relevance has even increased ever since. Nonetheless, up until recent years, its security has not been much of a problem as there have never existed problems related to network interfaces. This opened the surface of attacks by a big margin, letting the system unsafe from attacks that could interfere in communications thru the network. Even now, a big percentage of real-time systems still use C or C++ as their main language due to performance requirements, but at the same time bigger problem appeared. C and C++ are languages that do not have guarantee memory safety, leaving it all to the care of the programmer, which is a very hard and time-consuming task. The Rust language appeared as an answer to this. This is a language that defines data ownership through semantics that by itself guarantees memory safety at compile time. With this, Rust solves most of the major memory security problems present in C and C++ (Getreu, 2016).

André Pinho et al. (Pinho, Couto and Oliveira, 2019; Pinho, 2020) does a thorough investigation on the use of Rust language for the development of safety-critical systems by comparing the various features of Rust with the C language. This study concluded that, when trying to implement typical safety guidelines used for C within the Rust language, a lot of them could be eliminated as Rust does not allow some coding practices because of its safety measures. It is stated, that, even with this extra safety implementations present in Rust and its poor maturity as a fairly recent language, its performance is not far off that of C. However, when developing and comparing two implementations of a pre-emptive and a cooperative scheduler, it was identified some limitations of Rust such as the implementation of assembly code which is possible with C and the longer compiling times.

In their development of an embedded operating system for microcontrollers, Amit Levy et al. (Levy *et al.*, 2015), used Rust as their framework as it achieves memory and type safety without garbage collection. In their development, they concluded that Rust’s memory safety was not optimize for hardware and device drivers that are present in the system at all times, and that, for real-time systems, the closure requirements that exist in the language are not desirable. To help surpass these challenges and allow for a better implementation for event-driven real-time systems, they

proposed the notion of *execution contexts* that works using execution threads as the unit of isolation. This feature is a compile-time operation that identifies and allows safe memory sharing when hardware constraints or execution models prevent concurrency issues. Even with this addition, there is still work that needs to be done as the solution is not tested for different environments and there still are challenges to tackle when considering more complex types of real-time systems.

2.2.3 Real-time parallelism in Ada

Ada (Ada, 1983) is a high-level programming language designed for safety-critical and real-time systems. This language was originally created for the use of the United States Department of Defence (Defense, 1983), thus having a wide application domain. It was developed based on Pascal (Hansen, 1975) but also including some other features more common in specialized languages. These features include built-in support for concurrency and tasking, modular programming, and others. Therefore, having this support already implemented, the study of parallel programming with Ada has been a topic of development for a long time.

This study started several years ago with works like Thornley's (Thornley, 1994) which focused on extending Ada with keywords that allowed a block of code or a *for* loop to be executed in parallel and others to be execute sequentially. More recently, other proposals have appeared for the expansion of Ada, such as Moore's work (Moore, 2010) focusing on adding generics and pragmas, and Ali, et al. work (Ali and Pinho, 2011) having more emphasis in language constructs. These two papers were the base for a more recent proposal for parallelism with Ada programming language (Michell, Moore and Pinho, 2013; Pinho, Moore and Michell, 2014; Taft *et al.*, 2014; Pinho, Moore, Michell and Taft, 2015; Pinho, Moore, Michell and Tucker Taft, 2015).

This proposal came from the need of a better model for fine grain concurrency in Ada and it started in (Michell, Moore and Pinho, 2013) by presenting the term *tasklet* which is a single execution trace of parallel code that can be implicitly created by the compiler or expressed by the programmer with special syntax. This parallel code can be a construct, a parallel block, a for loop, and so on. In (Pinho, Moore and Michell, 2014) the model is refined in a way that each Ada task is taken as a controlled group of *tasklets* using a fork-join model. What this also means is that *tasklets* now can create other *tasklets* using the fork function and there is a subsequential need for synchronization within the main *tasklet* using the join function. In this model the *tasklets* inherit all the properties from its corresponding Ada task which means they have the same priority, deadline, and identification.

This model was then redefined in (Taft *et al.*, 2014) so that the parallelism became a task under the control of the controller and run-time instead of the programmer. Here, the programmer would just need to use the syntax defined to indicate blocks where parallelism could be present in the code and the compiler and run-time would be the responsible to evaluate and, if possible, execute the parallelism. (Pinho, Moore, Michell and Tucker Taft, 2015) extended this model by improving the reasoning of the execution model with progress guarantees without constraining the implementation done by the compiler and run-time. This was achieved by making a model for the execution of *tasklets* based on the abstract notions of executors. What this means is that each real-

time task maps to an Ada task that generates a set of *tasklets*, which can then generate other *tasklets*, and controls its dependencies. This proposal also proposes the use of this model in real-time systems which is then extended and deepened in (Pinho, Moore, Michell and Taft, 2015).

As discussed and proposed in a study by Pinho et al. (Pinho, Moore, Michell and Taft, 2015), this model gives the programmer a way to control the underlying parallel behaviour. Still, there are a few issues and improvements that need to be addressed before it has the facilities needed to be used in real-time systems. Some of these issues are the definition of priorities and deadlines of each *tasklet* which still are inherited by the Ada task that creates it, the inexistent migration of running *tasklets* between executors, and the lack of studies about the behaviour of the model to changes on the *tasklet* priorities and how to handle the possible parallelization of interrupt and timing events.

2.2.4 Fine-Grained Parallelism with Predictable Scheduling

The study of specific programming languages and toolchains is important, but another important area of research is around architectural models that support parallelism while meeting real-time demands. A notable example in this field is the research by Schmid et al. (Schmid, Fritz and Mottok, 2019, 2022), who investigate how runtime systems can be organized to allow fine-grained concurrency while also ensuring predictability which is crucial for hard real-time systems, where timing is key.

In their study Parallel Programming in Real-Time Systems (Schmid, Fritz and Mottok, 2019), the authors critique traditional runtime environments for their lack of timing determinism and insufficient support for concurrency control. They explain that common scheduling strategies in general-purpose systems often do not meet the strict requirements of real-time operations. Instead, they propose a design direction that prioritizes controlled task distribution and runtime awareness of temporal constraints.

Building on these concepts, the authors offer a clearer solution in their later work, Fine-Grained Parallelism Framework with Predictable Work-Stealing for Real-Time Multiprocessor Systems (Schmid, Fritz and Mottok, 2022). In this paper, they introduce a framework that uses a deterministic version of the work-stealing scheduling algorithm, designed specifically for real-time applications. The approach ensures predictable task migration and bounded response times even under high degrees of parallelism, which is achieved by enforcing strict control over scheduling decisions and reducing the unpredictability typically associated with dynamic task creation and load balancing.

Although the implementation language is not explicitly mentioned, the focus on low-level scheduling behaviour, real-time constraints, and system-level optimizations strongly suggests the use of languages such as C or C++, which offer direct control over hardware and execution timing. Overall, these works provide a perspective on parallelism that is not tied to any specific programming language but is still focused on timing, emphasizing the need for custom runtime mechanisms in environments where predictability is essential. The ideas presented underline the

importance of balancing parallel performance with real-time guarantees, a challenge shared across many real-time system implementations, regardless of which language or platform they use.

2.3 Analysis of Concurrency and Parallel Programming in Low-Level Languages

In this section, we analyse what models of concurrency and parallelism already exist for low-level languages. This work helps determine in a more technical way what exists and has already been implemented in these languages.

After analysing the gathered work about previous research on the topic, a decision was made to focus the proof of concept solely on Rust language, implemented with the Linux operating system. This decision came majorly because of Rust's big potential and the lower level of investigation done on it. Its use on real-time critical systems is imminent and that was a big motivation for this choice. OpenMP, on the other hand, should not be discarded and is highly recommended as a framework for the development of a research on real time systems as well.

As Rust is a recent language compared to the most used low-level existing ones, a thorough research on its methods and existing libraries is necessary, as much as an analysis of languages like C and Java, which were used as inspiration for much of Rust's implementations, especially, when it comes to parallel and concurrency programming.

To achieve a real-time parallel programming approach, one must understand what it means to have true parallelism. True parallel programming is not just dividing tasks, it requires creation and management of multiple threads, each running concurrently and potentially on separate processors or cores. This also mean that the programmer needs to be confident that what they want from each thread is what will happen and will occur during each program run, trying to maximize the use of resource.

Basically, to reach parallel programming, the use and control of threads is needed. This use of threads started being popular around the 80s when UNIX was already well established. The UNIX Operating System uses C as its main language, and this was what enabled developers to push the boundaries of what was possible using concurrency. It is in UNIX, thanks to the use of C, that POSIX standard (Portable Operating System Interface) was introduced. POSIX implements the first globally used threading model, *pthreads*, which provided a consistent API for creating and managing threads across different UNIX-based systems (Tanenbaum and Woodhull, 2006).

Thread management was a very complex and hard job to do, and that is where *pthreads* was able to shine. *Pthreads* presented an encapsulated way of doing this more easily and upfront. Because of this, *pthreads* was and still is used until now as a base for threading models in several other languages.

Rust was not an exception to this, and includes an implementation inspired on *pthreads*. Based on this, an understanding of how parallel programming works should start in its original implementation, in C language.

2.3.1 Parallel programming in C

The C language relevance, more often than not, revolves on its basis being on low-level programming, giving developers more freedom on memory access and resources management. This made C known as the language of system-level programming. With the CPU's starting to evolve having more than one core, the need for a language capable of handling such new opportunities grew, and thus made C the prime contender for this.

Parallel programming was one of the main focuses desired from this era of programming. With the possibility of now creating programs that ran in more than one core, the need of a language capable of managing such difficult tasks, brought C to high standards in the field. To do this, C made use of threads, primarily. Threads are smaller units of process execution that can run concurrently and thus, perfect for parallel programming. (Lewis and Berg, 1998)

To manage this in C language, it is typically used the POSIX threads library, *pthreads*. This library was introduced in the 1980s and was quickly introduced as the official library used for thread management within UNIX systems. It provides a set of APIs for creating, synchronizing, and managing threads, which allows developers to create code that can run concurrently (Lewis and Berg, 1998).

As a way to create threads, *pthreads* has *pthread_create*, which returns an integer, meaning if the thread was created successfully or not. If so, *pthread_created* will return a 0, otherwise it will return an error code, different from 0, reporting what the error was. This function takes as arguments the thread identifier, the attributes for it, the function to be ran by that thread, and an argument that can be passed to that same function. The thread identifier is defined as a custom data type defined in *pthreads* named *pthreads_t*, used to interact with the thread before creating it. As for the thread attributes, they can be passed as *null* if we do not want to define any attributes, meaning that the thread will run with the default ones defined in the *pthreads* library. If we want to run the thread with defined attributes, *pthread_create* expects a *pthread_attr_t* object which hold the desired thread attributes (Oracle, 2019).

The *pthread_attr_t* is part of the *pthreads* library, and it is used to specify the attributes of a thread when creating it. To manage this object, *pthreads* provides various functions such as *pthread_attr_init*, which initializes a *pthread_attr_t* object with default attributes. To override those attributes, *pthreads* has setters for: detach state, which defines if a thread is joinable or not, passed as an integer; stack size of the thread which is defined as a *size_t* variable; scheduling policy, such as FIFO (First In First Out) or RR (Round Robin), which are defined as integers in the *pthreads* library; and scheduling parameter, this being the priority, which have an unique struct defined in *pthreads* called *sched_param* that includes an integer variable for the priority. These functions take the initialized object as an argument, together with the desired attribute, and sets that within the *pthread_attr_t* object (Oracle, 2019).

As a way to facilitate working with the parameters of a thread created with *pthreads*, this library has getters for each setter described above. These getters take as arguments the *pthread_attr_t* object, and a pointer to the variable where the attribute will be stored. The getter functions, on contrary to the setter ones, have a return value which tells if the function was succefull in retrieving

the desired value of not. It returns a 0 value if the operation was successful and an error code if not. Together with the setter functions, these all follow the same standard naming, starting with *pthread_attr_*, followed by get or set, and the attribute desired, for example *pthread_attr_setstacksize* or *pthread_attr_getschedparam*. Since *pthread_attr_t* utilizes resources, such as any other object, *pthreads* also has a *pthread_attr_destroy* function which takes that same object as an argument and frees any resource used by it (Oracle, 2019).

To manage and synchronize threads, the *pthreads* library utilizes the *pthread_join*. This function has the purpose of allowing a thread to wait for another to complete before resuming. Just as *pthread_create*, *pthread_join* also returns an integer, either it being 0, meaning success in joining the pretended thread, or an error code. *Pthread_join* takes as arguments the thread identifier of the thread it is waiting for, and a pointer to a pointer of the memory space where the return value of the function ran in the thread can be stored. The second argument can be NULL if there is no value expected. To receive this return from the function ran on the thread, the *pthread_exit* function had to be used (Oracle, 2019).

The *pthread_exit* function is used within the function ran on a thread created with *pthread_create* with the goal of sending a return value to that thread. This function takes as an argument a pointer to any value that we wish to return to the thread that is calling *pthread_join*, and, although it could be of any type, it must be passed as void pointer, *void** (Oracle, 2019).

Working with threads was a hard task in itself, but if the developer wants to make the most use of parallel programming, just having various processes running different tasks unrelated to each other is not enough. With only these functionalities, a developer is only able to have threads where each has its own resources. The need to have threads running different tasks but with shared resources increased. This meant that a way to avoid memory access problems such as race conditions appeared. A race condition is when two or more thread try to access the same block of shared memory, having the outcome depend on which thread executes first. This kind of condition makes the code unpredictable and unreliable. To avoid these problems, *pthreads* integrates two functions that work as mutual exclusions, mutexes, and make this type of synchronization between threads possible (Oracle, 2019).

To be able to have mutexes, *pthreads* has a custom data type named *pthread_mutex_t*. This variable is used by the various threads as a key, able to lock or unlock access to certain parts of the code. To do this, *pthreads* has *pthread_mutex_lock*, the function used to lock the access to the *pthread_mutex_t* key for other threads. When a thread run this function, all other threads that try to run it as well, will get stuck and only run when the key gets released. The moment the key is released by the first thread, one of the waiting ones will get the key and block other threads that could be waiting to run *pthread_mutex_lock*. This function takes as an argument only the pointer to the *pthread_mutex_t* it wants to lock, and returns an integer, either 0 if it was successful, or an error code (Oracle, 2019).

To unlock a *pthread_mutex_t* there is only one way and that is to run the *pthread_mutex_unlock*. This is the function used to unlock a previously locked *pthread_mutex_t* and it can only unlock keys locket by that same thread. If a *pthread_mutex_t* gets locked, but the thread ends without running the unlock function, this will cause an error since the blocked threads trying to run

pthread_mutex_lock will never run again. The *pthread_mutex_unlock* takes as argument a pointer to the *pthread_mutex_t* and the return is an integer, working in a similar fashion as the *pthread_mutex_lock* function (Oracle, 2019).

Such as with *pthread_attr_t* used for the thread attributes, *pthread_mutex_t* also holds resources, and as such, has a *pthread_mutex_destroy*, designed to free those resouces. This function only takes a pointer to the mutex object that is wanted to be destroyed (Oracle, 2019).

Code 1 demonstrates how two threads can be created in C language using the *pthreads* library, setting some attributes to these before creating them. Outside of main, two variables are first defined, one as a shared variable that will be accessed by both threads, and the initialization of the mutex that will be used by the threads to avoid a race condition when trying to access that sheared variable. The mutex is being initialized using a macro present in the *pthreads* library. This macre is the *PTHREAD_MUTEX_INITIALIZER* that sets up the mutex so that it is initialized with the default attributes. This is similar to creating the variable mutex and then running the command *pthread_mutex_init* passing the mutex created and *NULL* as its attributes, meaning that it will use the default ones.

In the main function, the threads are first initialized by *pthread_t*, their identifiers, together with *pthread_attr_t*, which will hold their attributes and the struct *sched_param* that will be used to set the priority attributes for a thread. Then, both thread attributes are initialized with the default attributes set in the *pthread* library. After defining the priority to be 10 in *param*, we are able to set the thread priority by running *pthread_attr_setschedparam* for *attr1*. Similarly, we run *pthread_attr_setschedpolicy* with SCHED_FIFO to override the default scheduling policy for *attr2*. SCHED_FIFO is a real time scheduling policy available in pthreads that stand for First In First Out. This scheduling process is based on priority and, if more than one thread is run with the same priority, they will run in the same order as they were added to the queue, hence the name FIFO. After having set some attributes for both *attr1* and *attr2*, the threads are created using *pthread_create* which takes as arguments the thread identifier, the attributes previously defined for each thread, the function *thread_job* that is defined in lines 4 to 10, and a string that will be passed to that function. Finally, a *pthread_join* is run for both threads, what will mean that the main thread will wait for both threads to end before resuming. At the end, *pthread_attr_destroy* and *pthread_mutex_destroy* will free the resources used by both *attr* and the *mutex* variables.

```
1:  pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
2:  int shared_resource = 0;
3:
4:  void* thread_job(void* print_string) {
5:     pthread_mutex_lock(&mutex);
6:     shared_resource++;
7:     printf("%s - Shared Counter: %d\n", (char*)print_string, shared_resource);
8:     pthread_mutex_unlock(&mutex);
9:     return NULL;
10: }
11:
12: int main() {
13:    pthread_t thread1, thread2;
14:    pthread_attr_t attr1, attr2;
15:    struct sched_param param;
16:
17:    pthread_attr_init(&attr1);
18:    pthread_attr_init(&attr2);
19:
20:    param.sched_priority = 10;
21:    pthread_attr_setschedparam(&attr1, &param);
22:
23:    pthread_attr_setschedpolicy(&attr2, SCHED_FIFO);
24:
25:    pthread_create(&thread1, &attr1, thread_job, "Thread 1:");
26:    pthread_create(&thread2, &attr2, thread_job, "Thread 2:");
27:
28:    pthread_join(thread1, NULL);
29:    pthread_join(thread2, NULL);
30:
31:    pthread_attr_destroy(&attr1);
32:    pthread_attr_destroy(&attr2);
33:    pthread_mutex_destroy(&mutex);
34:    return 0;
35: }
```

Code 1 - Example of Parallel Programming in C using *pthreads*

The function ran by both threads during their executing is *thread_job*. This function both takes as argument and is defined as v*oid**. This happens so that there is a flexibility on the type of variable that is passed and return by the threads. In this case, the thread expects to receive a *char**, which correlates to a string in C language, that is used later in line 7 to print a message. The function accesses a shared resource, *shared_resource*, in which it firsts increments 1 to it before printing it. Without the use of a mutex, the two threads could be accessing this same resource at the same time, meaning that both could increment and the print, resulting in an output like this:

```
1: Thread1 - Shared Counter: 2
2: Thread2 - Shared Counter: 2
```

Code 2 - Possible threads output without the use of mutex (1)

But it could also happen that one access it first that the other, printing something like this:

```
1: Thread1 - Shared Counter: 1
2: Thread2 - Shared Counter: 2
```

Code 3 - Possible threads output without the use of mutex (2)

This would make the code unpredictable which is often not desired in programming. To solve this a mutex shared by both threads is firstly locked, meaning that only one of the threads will run the following code at a time. After printing the mutex is released with *pthread_mutex_unlock*, making the other thread free to run. This makes the code predictable, only having two outputs possible:

```
1: Thread1 - Shared Counter: 1
2: Thread2 - Shared Counter: 2
```

Code 4 - Possible threads output with the use of mutex (1)

Or

```
1: Thread2 - Shared Counter: 1
2: Thread1 - Shared Counter: 2
```

Code 5 - Possible threads output with the use of mutex (2)

This shows us how we can create threads and manage them in C, but as can be seen, for such a simple code, it still has 35 lines, excluding the imports of *pthreads* library for example, and it required a lot of caution to manage the resources correctly so to avoid any memory issues.

2.3.2 Parallel Programming in Java

Parallel programming is complex, and it becomes exponential hard when the user wants to run a big number of tasks. Also, by running various tasks in different threads it means to have bigger overhead when creating and deleting them. Because of this, another concept named thread pools was born, able to manage and reuse threads in a group, increasing concurrent programming performance.

Thread pools came as an efficient solution to managing multiple independent tasks that can be executed concurrently. For example, when a straightforward approach is taken, where each task runs in its own thread, the system performance can degrade due to resource starvation. Alternatively, creating and destroying threads as needed also introduces high overhead, which negatively impacts performance. A thread pool addresses these issues by maintaining a fixed number of threads that execute tasks and then remain idle until new tasks are queued. This approach eliminates the need of repeatedly creating and destroying threads, which reduces the overhead, and minimizes the risk of resource starvation. The thread pool usually works by, within the pool, submitting tasks to a queue, and worker threads fetching and executing them as they become available. If no tasks are waiting, the threads simply sleep, not impacting the system performance while still holding their resources (King, 2024).

However, implementing a thread pool is not so easy. Failing to ensure thread-safe access to shared data, causing thread leaks by not properly releasing references, or unnecessarily re-instantiating threads are some common mistakes made by developers when implementing a thread pool. For this reason, many libraries started to appear, from C to C++ and Java, offering customizable thread pool implementations. One example of thread pool implementations that was an inspiration for a lot of other implementations in different languages, including Rust, is the Java's *ExecutorService (King, 2024)*.

The concept of managing a group of threads came almost as early as the concept of threads. This came as a way to remove the overhead of creating and removing resources every time a new thread had to be created or destroyed. Threads was an idea meant to make programs run faster, but that was not happening when the number of threads created was high, making the overhead of always creating new threads not rentable. To solve this Java introduced thread pools as part of their concurrency model in early 2000's, and finally the *ExecutorService* officially in Java 5. This pattern is one of the most influential implementations of thread pools, being still used now in Java 21 and having been the inspiration for other languages implementation (Göetz *et al.*, 2006).

Java's *ExecutorService* interface offers a framework for managing a thread pool that addresses common challenges of concurrent programming, taking that stress out of the developers and helping the program be less error prone. One of the most common implementations of the Java's *ExecutorService* interface is called *ThreadPoolExecutor*. This is a Java class that handles a lot of the most common difficulties in a thread pool such as the creation, execution, and termination of worker threads. It also handles other points like of how tasks are queued and how threads are managed (Göetz *et al.*, 2006).

The *ThreadPoolExecutor* tackles thread reuse and thread management in an efficient way, abstracting the creation of the thread from the user. The complexity of thread creation and follow-up management are all removed from the programmer, having these made in the package. This way, securing an efficient use of system resources as well has lowering the chance of thread leaks. When a thread pool is created using the *ExecutorService* interface, the *ThreadPoolExecutor* sets the *corePoolSize*, defining how many threads the user wants. These threads can be created automatically on definition or created based on demand, this process is called lazy creation. What it does is tell the executor, how many threads the user wants by default, but, if these are not created immediately, the executor only creates the threads when a task is queued, and none of the threads are available. The executor does this until the number of threads hits the *corePoolSize* defined. If the executor is not configured for lazy creation, it automatically creates the number of threads defined in *corePoolSize*. After this, if the executor receives a task and the queue is full, it may create more threads beyond the desired number depending on its configurations. The threads created in this manner, on contrary to the main ones defined by *corePoolSize*, have a *keepAliveTime* which has a default value that can be overridden, and when it remains idle for more than this time, without running any tasks, it gets deleted. As a way to control how many threads we want beyond our *corePoolSize*, we can also define a *maximumPoolSize* using the *ThreadPoolExecutor*'s class. On the same basis, to define a number minimum of threads to be created on a lazy creation configuration, we can define a *minimumPoolSize*. With all of the above already assured by the executor, the programmer only has to create and configure the thread pool to have it spun up and manage the threads needed for his work automatically. The next thing that this framework tackles is the task submission (Göetz *et al.*, 2006; Oracle, 2024).

The *ExecutorService*'s interface has two ways of doing task submission. The first one is by running the command *execute*. This function takes as an argument a *Runnable* task and adds it to a queue to be executed in the future by one of the threads in the thread pool. In Java, a *Runnable* task is a block of code that is intended to be ran by a thread, without expecting any return from that task. The second way of submitting a task to a thread pool is by running a *submit*. The difference between

this method and a *execute* is that the *submit* can receive a return value form the task that is ran by the thread. The *submit* command can be called in two ways, one passing as an argument a *Runnable* task, the other passing a *Callable* task. The difference between these two is that the *Runnable* does not receive a return from the task running in the thread itself, but it receives a *Future<?>* which can be used to check the current status of that task and implement it as a way to wait for its completion. On the other hand, the *submit* with a *Callable* returns a *Future<T>* where *T* is the result of the task, allowing the user to retrieve its return value. These two functions allow the user to deploy tasks to a queue, whether they expect a return or not, without obliging the user to be aware of what thread it will be run in or how that is all managed. After having the tasks created and sent to a queue, it is the Java's *ExecutorService* job to manage how the task handling will occur (Göetz *et al.*, 2006; Oracle, 2024).

Task handling works with the use of a queue where all the tasks are added. The executor then checks all the threads active in the thread pool to see if any of them are available. If so, it allocates the task to that thread and removes it from the queue. If the there are no available threads, it keeps the task in the queue and waits until any of the threads become free. Another feature available in the *ExecutorService* is task scheduling, which allows for tasks to be scheduled to run after a delay or at fixed intervals in cases where periodic tasks are needed (Göetz *et al.*, 2006; Oracle, 2024).

To enable the end of the thread pool, Java has implemented in its *ExecutorService* some graceful shutdown methods. Two of the most used are the *shutdown* and the *shutdownNow* methods. The first one waits for all the tasks running at that moment it was called to end, and then terminates the thread pool, deleting the threads and resources used by them. The *shutdownNow* is able to terminate the thread pool without waiting for the tasks to end, interrupting them immediately (Göetz *et al.*, 2006; Oracle, 2024).

This approach by Java turned out to be very popular due to its efficiency and simplicity for the used to use. Because of this, various other languages used it as inspiration for their own implementation of thread pools. One of these languages was Rust and its *Threadpool* crate (Göetz *et al.*, 2006; Oracle, 2024).

```
1:  public class ThreadPool {
2:   public static void main(String[] args) {
3:     ThreadPoolExecutor executor = new ThreadPoolExecutor(
4:        2, // corePoolSize
5:        4, // maximumPoolSize
6:        10, // keepAliveTime
7:        TimeUnit.SECONDS,
8:        new LinkedBlockingQueue<Runnable>(2)
9:     );
10:
11:    executor.execute(() -> {
12:       try {
13:           Thread.sleep(2000);
14:       } catch (InterruptedException e) {
15:           Thread.currentThread().interrupt();
16:    }});
17:    executor.shutdown();
18:  }
19: }
```

Code 6 - Example of a Thread Pool in Java using *ThreadPoolExecutor*

Code 6 is a simple example of how a thread pool can be created using the *ThreadPoolExecutor* call. In the example above we use this class instead of the *ExecutorService*'s interface directly so that we can set variables like the *maximumPoolSize*. To initiate an instance of the *ThreadPoolExecutor* we have to pass a group of arguments that will configure how we want our executor to work. In this case we set the number of threads to be two and having a maximum number set to four. The threads created above our two default ones will have a *keepAliveTime* of ten seconds, as defined by the *TimeUnit* set. We also chose the queue maximum capacity to be two, meaning that if a third one was to be added, the executor will try to create another thread to be able to handle all tasks. The program then only executes a task where the thread will sleep for two seconds. This task will be added to a queue and executed by one of the threads created, Then the thread pool is terminated by running the *shutdown* command.

2.3.3 Parallel Programming in Rust

In Rust, parallel programming works the same way as in any other language. It involves the use of threads and concurrent tasking in order to improve performance. But Rust, on contrary to C, is a more advanced language, not so low level which enables more freedom and comfortability for the programmer. Rust excels from other languages because of its focus being on the safety of the user's program. Due to this, there are a lot of crates already created that help the developer by taking care of inconveniences like resource management and avoiding memory leakages. This is also the case for parallel programming. Parallel programming in Rust already has its own simplicities, achieved by the use of the *Rayon* library, one of the most popular in this subject.

2.3.3.1 Rayon Library

Rayon is a parallel programming library design to take advantage of multi core systems, distributing tasks through the multiple CPU's which allows them to run concurrently. Differently from what is allowed in C language, *Rayon* disposes an API which abstracts away the complexity of writing concurrent programs. To enable this, *Rayon* extends Rust's standard *iterator* (Blandy, Orendorff and F.S. Tindall, 2021).

The *iterator* present in Rust is a powerful tool used to work with elements from a collection like arrays, vectors, one by one. To create an *iterator* out of a collection, there are three available functions: *iter*, which creates a normal *iterator*; *iter_mut*, this generates a mutable *iterator*; and, *into_iter* which is the same as *iter* but consumes the collection and holds its ownership, making it impossible to use the same collection afterwards (Rayon, 2024a, 2024b).

It is based in this *iterator* that the *Rayon* library works. More specifically, what this library does is extend the *iterator* functions, making it work in a concurrent manner. Instead of iterating and running each element in a collection, it iterates the collection and splits the work into blocks that can be ran concurrently. This evaluation only happen at runtime and the thread and workload is managed automatically by *Rayon*, abstracting its complexity from the user. This mean that if the workload is considered not worth of parallelization, *Rayon* will run the code serially (Rayon, 2024a, 2024b).

To use the *Rayon* library main functions is very simple and easy to understand, since it is very similar to Rust's standard *iterator*. The main functions available are named *par_iter*, *par_iter_mut*, *into_par_iter* and they differ from each other the same exact way as the *iterator*'s *iter*, *iter_mut* and *into_iter* explained previously. All these functions, in the same way as in the standard *iterator* have integrated adaptors. As examples: the *map* command which applies a function to each iteration collecting their result; *for_*each that works similarly to the *map* but does not collect anything; and the *filter* command which filters iterations based on a predicate defined (Rayon, 2024a, 2024b).

```
1:  fn main() {
2:      let numbers: Vec<i32> = (1..1000).collect();
3:
4:      let sum_of_squares: i32 = numbers
5:          .par_iter()
6:          .map(|&x| x * x)
7:          .sum();
8:
9:      println!("Sum of squares: {}", sum_of_squares);
10: }
```

Code 7 - Example of parallel programming in Rust using Rayon

The example above shows a code snippet of parallel programming using *Rayon* library. In this code a vector with all numbers from one to one thousand is created. Then, using *par_iter*, the vector *numbers* is iterated, letting *Rayon* decide in how many pieces of code it will divide the work and pass it as tasks to threads it creates. Each iteration runs a *map* which collects the square of the number being iterated and sums it to then print the final result after all iterations have ended. When doing this with parallel work, instead of summing the result of each iteration, *Rayon* does the sum separately for each block running in a thread, and, in the end, sums the result of all the threads, getting the final result and optimizing the performance.

Rayon also has other functions that help create parallel programming, some of these being parallel ways of interacting with collections apart from iterating them. Some of these are, for example, *par_sort* which will sort the collection in a parallel fashion, and *par_extend* that extends a collection and can be used before a parallel iteration. Another type of functions present in *Rayon* are functions that can ensure that two different tasks are run and managed by *Rayon* and the program will wait for them to finish before resuming. These are the *join* command which can run two different tasks and waits for both to finish. The other one is *scope,* and it is very similar to *join* but the difference is that it allows the user to create a scope where multiple tasks can be spawned and it will wait until the scope ends before resuming. Both commands are managed by *Rayon* which means that the thread creation and management is all done by the library depending on the workload and conditions of the tasks (Blandy, Orendorff and F.S. Tindall, 2021; Rayon, 2024a, 2024b).

Working with *Rayon* and libraries of the like is reliable and often delivers the sufficient level of parallel programming needed, but this is not always the case. When the user wants to achieve optimal levels of optimization, they need to use lower level code such as in C language. Rust has its own integration of this with *libc*.

2.3.3.2 Libc

Creating parallel programs that are efficient and optimized is hard since it uses a lot of resources and memory accesses, meaning that it is very difficult to have a standardized library capable of doing it for every code. To enable this kind of management it is better to use low level coding since the user can control everything and achieve optimization when creating concurrent programs. This is where Rust's *libc* crate comes into play.

Libc is a crate in Rust that provides binding to the standard C library functions and types. These functions include the *pthread* library which providing the user a low level access to parallel programming and thread management. This crate includes all the functions present in *pthread* which lets the user code similarly to what they would in a C environment, granting him more freedom with threading (Rust, 2024a).

But there is an issue with *libc* use in Rust. Unlike C, Rust has a great emphasis on safety, meaning that all default functions are secured by Rust to be safe. This model implemented by Rust assures security by implementing strict rules about how memory and data are accessed and modified. Since *libc* grants the user direct access to low level functions present in C, these are not guaranteed by Rust's safety protocol and could lead to undefined behaviour. As such, the functions present in *libc* are not marked as safe and need to be covered by an *unsafe* bock for Rust to let them run (Blandy, Orendorff and F.S. Tindall, 2021; Rust, 2024a).

Code 8 presents an example of a code made in Rust using *libc*. From this code it can be seen how similar it is to the example written in C in Code 1. To use the *libc* functions the user can either call them with the *libc::* prefix or just import the crate functions. Apart from that, the only differences between the implementations are the use of an *unsafe* block which includes the call of *pthread_join* and *pthread_crate* from the *libc* library, which as explained before is because they are not covered by Rust's safety protocol. And the use of *extern "C"* in the definition of *thread_func*, this is necessary since it will be passed to the *thread_create* function which expects as an argument a type C function.

```
1:  extern "C" fn thread_func(_: *mut c_void) -> *mut c_void {
2:     println!("Hello from the thread!");
3:     std::ptr::null_mut()
4:  }
5:
6:  fn main() {
7:     let mut thread: libc::pthread_t = std::mem::zeroed();
8:     unsafe {
9:        libc::pthread_create(&mut thread,
10:          std::ptr::null(),
11:          thread_func,
12:          std::ptr::null_mut());
13:
14:       libc::pthread_join(thread, std::ptr::null_mut());
15:    }
16: }
```

Code 8 - Example of parallel programming in Rust using libc

2.3.3.3 Threadpool

Similarly to Java's approach, Rust's *threadpool* crate was developed to help users efficiently run their codes concurrently by creating tasks and having them be run on different threads in a parallel fashion. This crate work similarly to the Java *ExecutorService* where, by running the command *new* defined in it, the user can specify how many threads they want to have on his thread pool and the function will create these, leaving them ready to run any task any time the user wants. This way, similarly to Java, there is no overhead for running new tasks since it will reuse the same threads, maintaining their resources (Blandy, Orendorff and F.S. Tindall, 2021; Rust, 2024b).

In Rust's *threadpool*, the task handling and execution works the same way as in Java, by the use of a queue. In Rust's case, to submit a new task, the user has to run the command *execute* which adds the task to the queue, where all the threads are watching. Once a thread finishes its task it checks that queue to see if any tasks are there. If so, the thread will remove that task from the queue and run it, otherwise it will sleep, waiting for a task to appear in the queue (Blandy, Orendorff and F.S. Tindall, 2021; Rust, 2024b).

To delete a thread pool in Rust there is no command like in Java's *ExecutorService*. Rust handles that shutdown by itself when the thread pool runs out of scope and its threads have finished running their tasks. This happens because Rust's principles emphasize safety and so they handle all this in a way that avoids data leakage and other undefined behaviours (Blandy, Orendorff and F.S. Tindall, 2021; Rust, 2024b).

```
1: fn main() {
2:     let pool = ThreadPool::new(4);
3:
4:     for task in 0..8 {
5:         pool.execute(move || {
6:             println!("Task {}", task);
7:         });
8:     }
9: }
```

Code 9 - Example of a Thread Pool in Rust using the *ThreadPool* crate

The code above is a very simple example of how a thread pool can be created in Rust. As can be seen, it is very simple to create the threads, only having to run the command *new*. This command created four threads that are then managed by the thread pool. Then, using a *for* loop, eight tasks are added to the queue using the *execute* function. These tasks will be run by the four threads created for the thread pool and, when it is finished, will automatically clean all the resources associated to the thread pool since it goes out of scope.

All these libraries and implementations of parallel programming tools are remarkable and greatly help the developers code parallel programs, and that is evident by their popular use. But they are still not optimized for every system and use case. In Rust, thread pools are not made in a way that users can manipulate the threads within, choosing different parameters for each thread, and controlling which thread runs which task. Being able to do this would help the development of real time systems able to run with concurrent code.

2.4 Summary

Parallel programming in real-time systems is a demanding feature for the current state of modern technology, especially for critical systems. Currently, this field is dominated by C and C++ languages, but there have been developments in other frameworks to potentially improve the performance and safety of this systems. OpenMP with C/C++ and Rust are two frameworks that are good candidates for the development of models for real-time systems with parallel programming.

For this thesis, both these options were initially taken into account and the state of the art of both technologies proves that they are both suitable for the task. OpenMP has a proposed model (Serrano, Royuela and Quiñones, 2018) with room for improvements, and there are other ways of confronting the problem such as using the LLVM (LLVM, 2023) compiler instead of the chosen GCC 8.1. Other approaches like an event-driven execution model are also open as there is no existent support from OpenMP for this (Serrano, Royuela and Quiñones, 2018).

Rust, on the other hand, has little work when it comes to proposals for parallelism in real-time systems but with a lot of theoretical and closed-environment studies that prove that Rust is a real choice for this matter. It has similar performance levels as C only lacking more in compiling time but leveraging levels of memory safety way higher than those of C language. For the development of work in real-time parallel systems, a possible adaptation of the proposals made for Ada and OpenMP can be taken into the account, however, the bigger challenge seems to be the use and workaround of the ownership features of the language. This poses a possible problem has they are useful for memory safety but also limiting when it comes to variable and memory sharing between threads.

Both frameworks would be a great choice for the development of this proof of concept and they both have they needs for following studies, however, the one that this thesis will be based on is the Rust language. This choice was made majorly because of its innovation factor and the fast growing of the language and its community. Rust as a language has a big potential and its use on real-time critical systems is imminent and that was a big motivation for this choice. Even if it was not chosen for this thesis, the development of a proof of concept of a real-time system with parallel programming with the OpenMP framework is encouraged and seen as a very good theme of work development.

Based on this decision, a deeper analysis was made on the concurrency and parallel programming models present in low-level languages. In this analysis a greater emphasis was made around the Rust language and its precedents, in which the language implementation was based on. This analysis went through various libraries and crates that are very commonly used in their environments, which helped us create the concepts desired for the proof of concept.

3 Real-Time Parallel Programming API for Rust

A real-time parallel approach for programming concurrent software. This translates to having a code designed to have tasks running simultaneously, increasing efficiency and quickness. On the other hand, to be a real-time system is to act under time-constrains that, when missed can lead to system failures or undesirable outcomes. This combination increases the complexity of scheduling and synchronization greatly, making it a very hard task that diverges a lot from system to system. With the current tools available in Rust, it is possible to achieve it, but the difficulty level is high since it will leave most of the work to the programmer. This is increasingly true when the complexity of the real time system increases since the need to control every aspect of each thread is high. With the current set up the developer needs to program almost everything from the basics. More specifically, they would need to use a low-level API working directly with C, such as the *libc library,* since it would allow them to manage all the threads attributes more easily and concisely.

Therefore, there is a need for a more concrete and focused parallel programming library for real time systems. A library capable of managing threads and creating thread pools, but at the same time giving the user all the freedom to finely manage the threads within the thread pool.

To architect our real-time parallel programming library, we first conducted a study of which are the needs of a programmer to build software for real-time systems. We designed the library, named rt_lib, based on this study and considering other standard and good practices in the context of both real-time systems and in parallel programming.

3.1 The Needs of a Real-Time System Programmer

When working with threads in system-level programming, there are certain expectations that a programmer usually has. Especially in real-time or high-performance systems, having control over how threads behave becomes very important. In many cases, a programmer wants to be able to decide not only how many threads to use, but also how those threads run, what their scheduling policy is, what CPU core they run on, and what priority they should have. Most common libraries in Rust do not give this level of control. Libraries like Rayon are good for regular parallel tasks, but they’re not built for low-level thread management. Because of that, a more specialized solution is needed (Rayon, 2024a).

One of the first things that a programmer working with real-time or embedded systems usually wants is the ability to set a thread’s scheduling policy. In Linux, policies like FIFO and RR are used when timing is critical. So, a good thread library should allow the programmer to choose these policies easily. The *rt_lib* library solves that by letting the user set the scheduling class through a scheduling parameters struct. Another common need is to bind threads to specific CPU cores. In

real-time systems or multicore optimization, this is sometimes done to reduce latency or avoid conflicts, and a programmer who knows their system's structure might want a certain thread to always run on core zero, while another runs on core three. With this library, that's possible through the same scheduling parameters struct, where the user can choose the core for each thread to run on. If they do not set it, the system will pick one based on what CPU is free, but having that option can be very useful when performance needs to be fine-tuned (Cerqueira, Gujarati and Brandenburg, 2014; Red Hat, 2025).

Setting priority is also something that programmers look for when building low-latency systems. Threads that handle more important tasks should run before others, that's why priority settings are important, but many libraries in Rust do not offer this level of access. In this custom library, priorities can be set for each thread individually which makes it much easier to design systems where tasks are not all treated equally, which is more realistic in most real-world scenarios (Lee *et al.*, 2022).

Programmers also tend to look for flexibility in thread creation. Some situations require many threads with the same behaviour, while other situations need each thread to act differently. A good thread library should allow both. In the *rt_lib* library, if the programmer gives just one set of scheduling parameters, the library will apply them to all threads, but if the developer wants to set different parameters for each thread, they can do that too. This keeps the library easy to use, while still being flexible enough for more complex needs. There are also times when a programmer might already have a scheduling setup defined outside the program, for example, using some shell script before running the application, or when they want threads to create others equal to it. In these cases, it's helpful if threads can just inherit the scheduling behaviour that's already active. This issue is also tackled in a method within this library. It checks what scheduling parameters the current thread has, and then uses that same setup for the new threads which saves time and avoids mistakes from reconfiguring the same thing twice (Cerqueira, Gujarati and Brandenburg, 2014; Lee *et al.*, 2022; Red Hat, 2025).

When it comes to running jobs in threads, developers usually want to assign tasks in a clear and simple way. They may want to send a function to a specific thread, or just let the system pick any available one and they might also want the job to run every few milliseconds, which is common in control loops or real-time audio. This library supports both of these with its execute function, which lets the user choose the thread, and optionally add a periodic time for it. These are things that aren't usually included in basic thread libraries, but which make a big difference for specialized systems (Ngo, 2020; Red Hat, 2025).

Another thing that is often helpful is to inspect how the threads are behaving. A programmer might want to check the current parameters of a thread to make sure everything is working properly. This is especially true when debugging a real-time system. The get function in this library allows that. It returns the parameters of all threads, or only the ones the user asks for.

Finally, any good threading tool should allow clean shutdown and recovery. Sometimes we want the thread to finish what it's doing and exit calmly, other times we need to kill the thread right away. In the *rt_lib* library we give the developer a choice depending on how critical the moment is. We

also give the developer the chance of only exiting the threads that are not periodic, letting the rest still run until another command is sent.

Some tasks may not need a thread pool at all, they just need one thread running separately. In these situations, a programmer will appreciate having an option to do this, instead of having to create a thread pool with only one thread. In this library, we let the user create and manage a single thread outside any pool, which is helpful for isolated tasks. It also gives full control over stopping or killing that thread, just like with the pool.

To make things even easier, the library includes small helper functions to retrieve information form each thread like its current CPU, priority or even process identifier. These are good to have when the developer wants to quickly check system values without diving into unsafe C code or writing complex wrappers around *libc*.

In the end, what most programmers want in a thread library is a balance between control and safety which this Rust library gives. It allows deep control over scheduling, priority, and CPU usage while still respecting Rust's rules around memory and safety. It does not try to replace higher-level crates like Rayon, but instead fills a gap for working closer to the hardware and to all its capacities. For people building embedded systems, real-time software, or high-performance backends, these are exactly the kinds of tools they're looking for.

Based on what was gathered during this investigation, we decided to create a set of user stories to help us keep track of every goal set with the intent of later revisit and use them to help evaluate the final product of the *rt_lib*:

- US1 – As a developer, I need to customize the scheduling policy of a thread so that it meets timing requirements in real-time environments.
- US2 – As a developer, I need to bind threads to specific CPU cores to optimize performance and reduce latency in multicore systems.
- US3 – As a developer, I need to set thread priorities individually so that more critical tasks are executed before less important ones.
- US4 – As a developer, I need the option to run a single thread outside of any thread pool.
- US5 – As a developer, I need flexibility in thread pool creation so I can create many threads with either shared or individualized parameters.
- US6 – As a developer, I need the option for thread pools to inherit scheduling settings from their parent or current context so I can maintain consistency across thread hierarchies.
- US7 – As a developer, I need to assign jobs to specific threads or allow the system to assign them dynamically in case of thread pools, depending on the situation.
- US8 – As a developer, I need the option to schedule periodic jobs with a fixed time interval.
- US9 – As a developer, I need fine-grained control over how threads are shut down, including selective termination of non-periodic threads.
- US10 – As a developer, I need helper functions that let me query thread attributes easily without dealing with unsafe code.
- US11 – As a developer, I need the option to inspect parameters inside a thread to verify the system is operating correctly and assist with debugging while not dealing with unsafe code.

3.2 Translating Needs into Design: Methodologic Overview

Based on the expectations and constraints described in the previous section, the development of *rt_lib* was guided by a design approach centred on simplicity and control for the user. We began by identifying what a programmer would ideally want when creating a real-time system with threads and then shaped the API around those expectations.

The first design idea was to reduce the overhead of having to set thread parameters manually. Real-time developers often need to define scheduling policy, priority, and CPU affinity so, instead of requiring users to write those individually every time, we created a single struct for all the scheduling parameters. This struct could then be reused across methods to set up threads with clear and concise definitions. From there, the idea was to offer simple methods that could take in these parameters, together with a task, and then spawn a thread accordingly. For example, calling something like *runThread* would be enough to create a new thread with the desired attributes and execute the task. Similarly, *runPeriodicThread* would handle periodic execution using the same parameter structure but also including a time interval on creation. These two functions became core to our approach because they cover both one-time and periodic task execution, a very common need in real-time systems.

```
1:  params1 = {
2:     sched = FIFO,
3:     priority = 70,
4:     cpu = 2
5:  }
6:
7:  runThread(
8:     params = params1,
9:     task = {
10:       …
11:    }
12: )
13:
14: runPeriodicThread(
15:    params = params1,
16:    periodSec = 5,
17:    task = {
18:       …
19:    }
20: )
```

Code 10 - Pseudocode for Thread Pool usage with Real Rime Parallel Library (1)

Beyond individual threads, we recognized that many use cases would require several threads working together but with different or shared configurations. So, we expanded the design to support two types of thread pools. The first type allows the user to define each thread's parameters individually. The second lets threads inherit the attributes of their creator, which is useful in systems that already configure threads through external tools or setup routines.

```
1:  params1 = {
2:     sched = FIFO,
3:     priority = 70,
4:     cpu = 2
5:  }
6:  params2 = {
7:     sched = FIFO,
8:     priority = 90,
9:     cpu = 3
10: }
11: params = list(params1, params2)
12: numberOfThreads = 2
13:
14: threadPool1 = new threadPool(params1, numberOfThreads)
15: threadPool2.run(
16:    thread = 0,
17:    task = {
18:       …
19:    }
20: )
21:
22: threadPool2 = new inheritedThreadPool(numberOfThreads)
15: threadPool2.runPeriodic(
16:    thread = 1,
17:    periodSec = 5,
18:    task = {
19:       …
20:    }
21: )
```

Code 11 - Pseudocode for Thread Pool usage with Real Rime Parallel Library (2)

This balance between simplicity but at the same time complexity of the thread system created, was a key design goal for this work. With this, users are able to control every thread if needed, but they aren't forced to do so for simpler or more repetitive cases. This approach gives the user full control of the thread pools they create, but in case the user does not wish to always control the thread in which the task will run, an option to leave it to the system is open by not inputting that field, giving even more freedom to the user. Finally, these methods were developed to reflect the patterns that real-time developers are already used to, just in a safer and more Rust-native way. Instead of requiring direct usage of *libc* or unsafe wrappers, the *rt_lib* library offers clean abstractions that still allow deep system-level control.

Just like the execute methods, this approach opens the library to other possibilities that the user can have to iterate with inside its thread or thread pool. One example are the exit methods that, by giving the user a variable with its setup, they can use to then manipulate when they want them to finish. These methods can easily be executed by just running a *threadPool.exit*, for example.

By shaping the API around actual developer needs, rather than just system capabilities, the resulting library allows developers to focus on the structure of their system, while leaving the lower-level thread management to a predictable and flexible interface.

3.3 Internal Organization of the *rt_lib*

To support the high-level design goals of the *rt_lib*, the internal architecture needs to be organized around a set of custom Rust structs that encapsulate both the thread lifecycle and scheduling logic. While users interact mainly with public-facing abstractions, the library internally will coordinate thread creation, job execution, and scheduling through a modular structure. This separation maintains safety and simplicity in the API while still enabling detailed control under the hood.

To achieve this, two main struct types have to be exposed to the user, one for the thread pools, and one for isolated threads. This is needed so that we can maintain all the thread's data safely secure inside the system, while still giving the user the freedom to iterate with its functions. In this case, what needs to be created, are public structs with private arguments that are then managed by the library internally. To keep the complexity hidden from the user's perspective, these structs will be as simple as possible, letting the all the hard work be deeply stored inside the library and away from the user's access.

In the isolated thread's case, when the object is created, it will create another struct deeper within the library that will be the one managing the thread's data and store that struct inside itself. Since in this case the system only has to manage one thread, the only data needed are the process identifier of the thread created, and a flag that will be shared with the thread created. This flag will be used to signal the thread when it has to stop, which is set when the user runs the exit method. Finally, when creating the thread, another group of variables need to be present inside it. For isolated threads, it needs its own task, the period set by the user, and the same flag present in the outer struct, so that the thread can shut itself when it is set to true. Figure 1 helps to illustrate this.

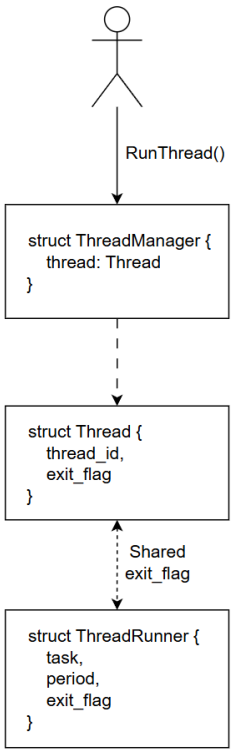


Figure 1 - How Isolated Threads are managed within the library

For the thread pool, the setup presented to the user keeps the same simplicity, by having the parameters set by the user saved within, together with another struct built to manage a thread pool from the system side. This *ThreadPool* struct is more complex and needs more information to be able to manage groups of threads, such as a group of queues for each thread. This is already a different approach from other thread pool implementations since, instead of only having a global queue that is shared between all threads, in this library each thread has its own queue, which is what enables the user to specify in which thread they want to run each task. Together with it, the struct also holds the values for the process identifiers of all threads created, and some flags that are used to manage the threads' exit routine.

Just like for isolated threads, when passing the data to the threads that are running inside the thread pool, a new struct had to be built for them. This struct, *ThreadPoolThreadRunner*, includes the same shared flags that are present in the *ThreadPool* struct, the thread correlation identifier which consists of the internal identifier of that thread used within the library, the queue for the respective thread, and the own *ThreadPool* struct in which it makes part of. This is necessary since, to be able to handle periodic tasks, we opted for the thread to create another thread responsible for handling that task's periodicity, removing complexity from itself.

This periodic thread handlers, when created, also must be able to manage workload inside the internal system of threads. As such, the *PeriodicTaskRunner* struct created holds the characteristics of the task it is managing, the queue of the thread that task is meant to be sent to, and also the shared flag used to handle safe termination of all threads.

Since we give the user the option to ignore the assignment of a task to a thread, this needs to be handled by the system. Considering this, another thread is always going to be created inside the thread pool system in order to manage the workload sent without a defined destination. This thread will also need to have access to shared data between the other threads and the system, and as such, a *LoadBalancerRunner* struct was also designed. This structure takes all the job queues, the shared termination flag, and the own *ThreadPool* struct to be able to, as in the other thread pool threads, create a periodic handler for a task if needed.

The complete structure of how thread pool data is managed within the library is illustrated on Figure 2.

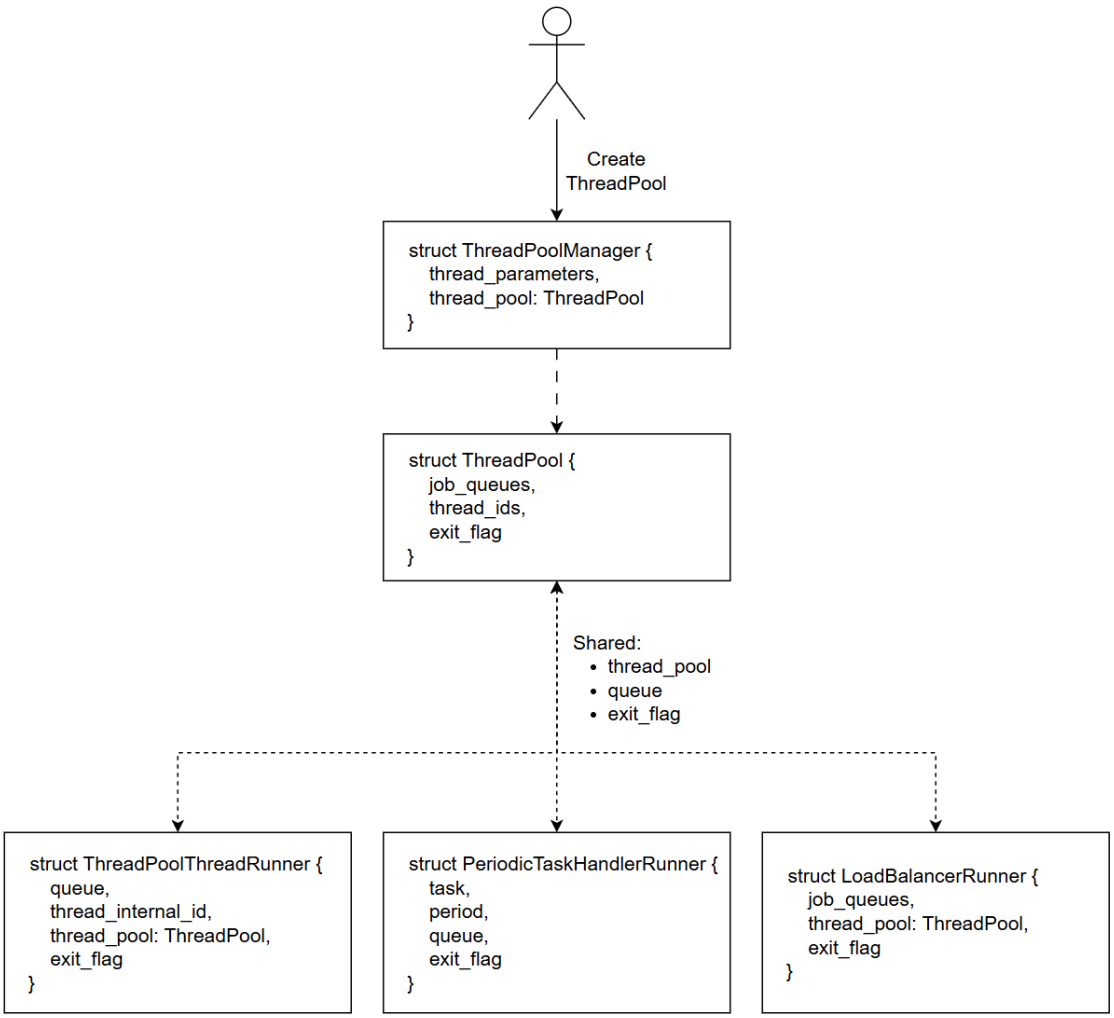


Figure 2 - How Thread Pools are managed within the library

3.4 Summary

In this chapter, we presented the conceptual motivations, design strategies, and internal architecture behind the *rt_lib*, a tool designed to facilitate real-time parallel programming in Rust. It introduces how it aims to combine low-level thread control with Rust's safety guarantees, offering a practical and flexible interface for developers working on real-time or embedded systems. The library addresses common challenges in thread management, such as scheduling policy configuration, CPU core affinity, and periodic task handling, by abstracting away unsafe code interactions and exposing a clean, intuitive API.

The chapter is structured into three main sections: the programmer's needs, which explores the rationale behind the library's features; a methodologic overview, which translates these needs into design strategies; and the internal design of the *rt_lib*, which breaks down how the system is architected and how its components work together.

The first section focuses on understanding what a programmer typically needs when building real-time or performance-critical systems. It explores the common gaps found in existing thread libraries, such as lack of control over scheduling parameters inside the thread pool's threads, and presents how these needs shaped the foundation of the *rt_lib* library. Whether it's assigning periodic tasks, managing individual threads outside a pool, inheriting scheduling attributes, or having fine-grained control over execution parameters, this section lays out the reasoning behind every major feature. By starting from the programmer's perspective, it becomes clear why an alternative to the default Rust thread model was necessary and how *rt_lib* aims to offer both low-level control and ergonomic usability for real-time systems. At the end of this sector, we also created some user stories, basing them on the programmer needs identified. These are later addressed and are a way of helping us determine if the *rt_lib* is indeed a library made accordingly with the programmer needs.

The second section of this chapter explains how the library's structure emerged from the needs previously identified, focusing on balancing simplicity with the flexibility required in real-time systems. Instead of forcing the programmer to deal with scattered, low-level configurations, we grouped the most important thread attributes, scheduling policy, priority, and core affinity, into a single struct, making thread creation more intuitive. From there, we designed methods the outlines for methods to allow straightforward thread execution, with the right parameters and behaviour already built in. The approach naturally extended into supporting thread pools, offering both fine-grained control per thread or automatic inheritance of parameters when desired. Throughout, the emphasis was on keeping the interface clean and familiar to real-time developers, while still providing access to the kind of low-level control typically only available through unsafe code or direct calls to *libc*. This methodology ensures the API remains close to what developers actually need, without overcomplicating more common use cases.

The third section dives into the internal organisation of the library, explaining how the *rt_lib* is structured and how it manages real-time threads under the hood. The core public-facing types are *ThreadPoolManager* and *ThreadManager*, which act as entry points for the user while encapsulating internal complexity. Behind them are internal types which handle system-level interactions such as thread lifecycle management and synchronization. Specialized structs are then used to manage different thread behaviours, such as periodic execution or task balancing. Each of these types works together to coordinate task execution, monitor shutdown conditions, and ensure safe concurrent access to shared data.

In summary, the library offers a complete and focused solution for managing real-time threads and thread pools in Rust. It simplifies low-level thread management by wrapping complex *libc* interactions in safe, high-level abstractions, while maintaining the level of control required for real-time behaviour. With support for core features such as priority assignment, CPU pinning, periodic execution, and task queuing, the library provides a versatile platform for building robust concurrent systems. Its architecture prioritizes modularity, safety, and performance, making it well-suited for developers working in real-time systems.

4 The Design of *rt_lib*

The *rt_lib* is a library that takes into account the ideas present in the *ThreadPool* crate (Rust, 2024b) and implements them into a spectrum of real time systems, with more freedom and control for the user, mixed with a new approach that simplifies the creation of threads and running of tasks concurrently. *Rt_lib*'s approach is also to add shortcuts and more simplicity for functionalities that are usually not present in thread pool libraries as well as to access to other that are normally only accessed by directly using low level code like in *libc*, without the need to explicitly use unsafe code and risk bad memory management.

The library is divided into four sections like it is presented in Figure 3. The *lib*, which is the one where the methods presented to the user are and works as an interface. Then, the *libc_abstractor*, the *thread_manager* and the *thread_runner* which implement all the functions and logic used by the *lib*. The *thread_manager* module contains all the data and methods needed to manage and create threads and thread pools, while the *thread_runner* manages the running cycle of the threads created, running the tasks sent by the user and guaranteeing periodicity. Finally, the *libc_abstractor* section contains all the unsafe code and different uses of the external library *libc*, keeping the code inside all the modules cleaner and concise. This separation helps to organize and module the library, keeping the complexity and hardcoded parts hidden from the user. This modulization also helps to organize the library and its methods, dividing them accordingly with their purpose.

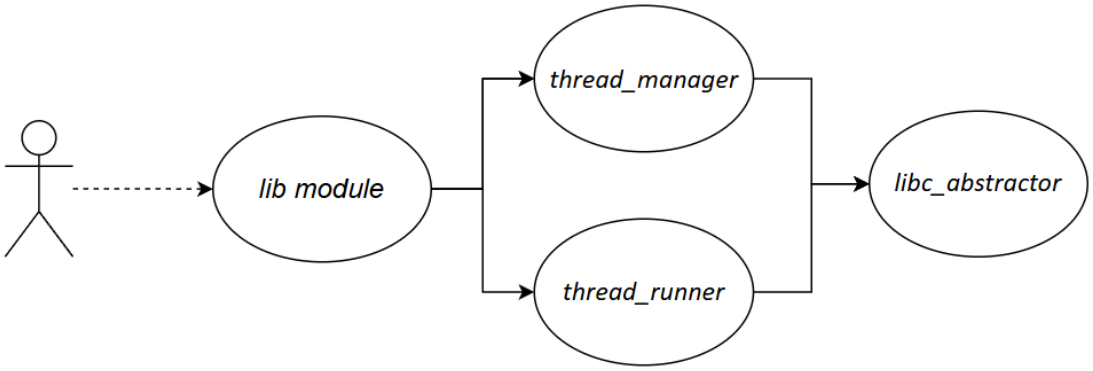


Figure 3 – Structure of the *rt_lib* library

To better explain how *rt_lib* works, its logic will be divided into two main flows, the creation and management of threads from the user perspective, both inside and outside of a thread pool, and the work done by the library to manage the threads based on the user's commands and needs.

4.1 Creating and Managing Threads using *rt_lib*

The management of threads inside the *rt_lib* is mainly done for the threads that reside inside a thread pool, this is the case because running threads isolated has less requirements and does not involve much management. For a normal thread, its creation and execution were implemented in a very simple manner so that using it could be done by abstracting all the error prone code from the user. The same was done for the thread pool, but since this involves more threads working together, the implementation was more complex and carefully done to give the user all the tools they would need without degrading the integrity on the program.

To give the user the possibility of running threads utilizing the parameters they want, and to apply what was imagined and designed in the previously methodology work, the struct *SchedulingParameters* was created. In this struct the user can define every aspect needed for a real time thread, these being the *class*, *core*, and *priority*. The *class* is where the scheduling class of the thread is defined, using a *SchedulingClass* which is an enumerator with the available real time scheduling classes that can be used, FIFO and RR, in this case. This value is an optional variable, this way the class will not be set and the system will use its default one. The *core* is also an optional variable where the user can set the core in which they want the thread to run. This can be left empty if the user wants the library to choose the core according to availability. The *priority* sets the priority value of the thread, which is needed for real time scheduling classes. An example of how to use this *struct* is shown below.

```
1:  params = rt_lib::SchedulingParameters {
2:    class: Some(SchedulingClass::FIFO),
3:    core: Some(0),
4:    priority: 9,
5:  }
```

Code 12 - Example of *SchedulingParameters* use

Having decided on a struct where the user can easily decide on the attributes wanted for the thread to have, we designed what functionalities would be expected of a real time library. One thing that was thought as a requirement for creating real time parallel systems is the ability to run tasks in threads using the parameters set in the *SchedulingParameters*, without worrying about dependencies and removing the difficulty of creating the threads, guarantying concurrent programming. This approach is implemented for both the creation of isolated threads and for the creation of threads within a thread pool.

4.1.1 Isolated Threads

When developing code to run concurrently, a very useful tool to have is a simple yet effective method which would receive the parameters the user wants the thread to have, but also the job they want the thread to run. In *rt_lib* this is possible by using the parameters struct presented above, by simply running the *create_running_thread* command as demonstrated in Code 13. This method is available through a public struct name *ThreadManager* included in the library.

To create an isolated thread, the method receives the struct containing the scheduling parameters, but also an optional value for the period in which the user might want the job to run in that thread. If the value, expected in milliseconds, is present, the job will be scheduled periodically in the same thread. Otherwise, the job is executed once, and the thread terminates. The job is sent by using the keyword *move* which transfers ownership of all variables used inside the closure into the closure itself. This way, the code inside the variable can be sent forward until it reaches the thread itself without losing ownership.

```
1:  params = rt_lib::SchedulingParameters {
2:    …
3:  }
4:
5:  job = move || {
6:    …
7:  }
8:
9:  period_ms = Some(1500)
10:
11: thread = match ThreadManager::create_running_thread(params, period_ms, job) {
12:   Ok(thread) => thread,
13:   Err(e) => …
14: }
```

Code 13 - Example of *create_running_thread* method

The *create_running_thread* method was also made in order to help handle errors that might occur during the creation of the thread. This method returns a *Result* type variable that, if unsuccessful, returns a string with the error that occurred or, if successful, a *Thread* struct.

This new struct is not entirely available to the user since it is used to manage the thread internally. But nonetheless, it is very useful for the developer since it has two other methods that can be used to manage the thread created. These methods are *exit_running_thread* and *kill_running_thread*. Both commands do not require arguments and are used to terminate the thread. The first one is only relevant when running periodic tasks since, after being used, it only tries to end the thread when no job is running. This meaning that if a job is midway through it will wait until before the next iteration to stop the thread and end the periodically running task. The second one can be used for both periodic and non-periodic tasks since it will terminate the thread immediately, regardless of where the thread is at that moment.

4.1.2 Thread Pool

Thread pools, on the other hand, are way more complex to manage since they can involve any number of threads the user wants. This complexity was the reason we mainly focused this thesis work on them. In an ideal user perspective, they would be able to create a thread pool with whatever number of threads they want while still being able to set the parameters of each thread in a simple manner. This was made possible by using the same *SchedulingParameters* struct which the user can create for each thread individually and will be set respectively. The user can, otherwise, only create one struct, and, in this case, these parameters would be used for all the number of threads wished by the user.

To create a thread pool using this method, the user can use a public struct named *ThreadPoolManager* present in *rt_lib* library. With this, the user can call the method *new_rt* which takes as arguments a group of *SchedulingParameters*'s and the number of threads the user wants to create. This way, the method can create the thread pool and send the created struct back with whom the user can them iterate to manage the threads inside it. To use this method, the user can either create a group of parameters equal to the number of threads they want to create, or a group with only one set of parameters, with which the method will then use to create all the threads. If the user sends more than one set of parameters and the number of threads also does not correspond to it, it will fail. Like with the *create_running_thread* command, this function also returns a *Result* type variable which, when unsuccessful send a string back with the error for the user to be able to handle if they want to. This method does not run anything on the threads, only creating them and having them ready to receive commands. The following example shows how the user can create a thread pool with three threads, all with different parameters.

```
1:  params = vec![
2:    rt_lib::SchedulingParameters {
3:      …
4:    },
5:    rt_lib::SchedulingParameters {
6:      …
7:    },
8:    rt_lib::SchedulingParameters {
9:      …
10:   }
11: ]
12:
13: number_of_threads = 3
14:
15: pool = match ThreadPoolManager::new_rt(number_of_threads, params) {
16:   Ok(thread_pool) => thread_pool,
17:   Err(e) => …
18: }
```

Code 14 - Example of *new_rt* method

Another way of creating threads that might be useful for the developer, is to create without having to set any parameters because these were already set before. As an example, the user might have run a shell script before compiling his program, where they created a thread with specific parameters to have his program run on. If the user wishes to maintain his structure, then the existence of a method that could create threads by inheriting its parent's parameters would be very useful.

For these cases, the *rt_lib* library has a method named *new_inherited*, which, just like *new_rt*, creates a new thread pool with any number of threads the user wishes. Since it inherits parameters, the only thing this method requires is the number of threads to create and, with it, it generates the corresponding *SchedulingParameters* to create the threads with. This method returns a *Result* type variable which the user can then use for handling in case of error, otherwise, it also sends a *ThreadPoolManager* struct with its newly created threads. The example below demonstrates how to create a thread pool with three new threads using this method.

```
1:  pool = match ThreadPool::new_inherited(3) {
2:    Ok(thread_pool) => thread_pool,
3:    Err(e) => …
4:  }
```

Code 15 - Example of *new_inherited* method

After having created the thread pool, the user would want to run tasks on its threads and that where the *execute* function comes in. This method is run inside the *ThreadPoolManager* struct and, in it, the user can define in which thread they want the task to run, the task itself, and if they want the thread to run periodically and with what period. Both the task and the period value are passed similarly to how it is in isolated threads, but, unlike it, this method also expects a value for the thread in which the task should run. The values assigned to each thread corresponds to the one created with the parameter based on the order sent to the *new_rt* previously run. This value is an optional value and, if sent, it guarantees that the task will always on that same thread, otherwise, the user can leave this to the system to choose. In this case, the task will not have an assigned thread assigned to it and will, each time, run on the thread that is available sooner. This is very useful for cases where the user wants to have different threads available but does not care about which runs what, just wants the thread to run as soon as possible. The following example shows how the user can run jobs on two different threads, using the same parameters, after creating them inside a thread pool.

```
1:  params = vec![
2:    rt_lib::SchedulingParameters {
3:      …
4:    }
5:  ]
6:
7:  number_of_threads = 2
8:
9:  pool = match ThreadPoolManager::new_rt(number_of_threads, params) {
10:   …
11: }
12:
13: job = …
14: thread = Some(0)
15:
16: match pool.execute(thread, None, job) {
17:   Ok(t) => (),
18:   Err(e) => …
19: }
20:
21: period_ms = Some(1000)
22:
23: match pool.execute(None, period_ms, job) {
24:   …
25: }
```

Code 16 - Example of *execute* method

In Code 16, we show a quick example of how to create a thread pool two threads using the same set of scheduling parameters and then using that same thread pool to run a task. In the example, we first chose to run a job on thread with the identifier zero and with no periodicity. The second execute function does not have a thread associated to it, and as such, its task will run on the first thread the system sees available at the time. Since we also set a value for the period, this means

that this task will run every second, every time on a thread the system sees free. Also, as we can see from the first execute call, this method, like the other ones, also returns a *Result* type variable to let the user be free of handling the error how they want.

After our thread pool is created and able to run tasks on it, functions to stop these threads are also useful tools to have. Just like for isolated threads, there exists an *exit_threads* and a *kill_threads* method. The kill command works just as expected, where it terminates all threads in the thread pool, no matter at what point they are. Unlike the prior implementation, the exit method for the thread pools has two ways of working. Since with thread pools we always have the threads created and ready to run tasks, the user might want to exit only the threads that not have periodic tasks running inside them. This way, the user can free the system of the non-used threads but keep the period tasks running. Otherwise, the user can also choose to exit all threads, including the ones running periodically. In this case, the thread is going to wait until before the next iteration to terminate the thread. To be able to choose which threads the user wants to terminate, the *exit_threads* command takes as argument a *bool* variable that, when set true will also terminate threads running periodic tasks, otherwise, will exclude these from exiting. Both these methods also return a *Result* variable for possible error handling.

Another useful tool to have while debugging and programming with thread pools, is a get function that sends out the information of all threads inside a thread pool. For this, the *rt_lib* library includes a *get* method inside the *ThreadPoolManager* struct which sends this information to the user. This information is sent as a group of *SchedulingParameters* which the user can then use freely.

```
1:  params = vec![
2:    rt_lib::SchedulingParameters {
3:      …
4:    }
5:  ]
6:
7:  number_of_threads = 5
8:
9:  pool = match ThreadPoolManager::new_rt(number_of_threads, params) {
10:   …
11: }
12:
13: job = …
15: thread = Some(0)
16: period_ms = Some(1000)
17:
18: match pool.execute(thread, period_ms, job) {
19:   …
20: }
21:
22: sched_params_0-2 = pool.get(Some(vec![0,2])))
23:
24: match pool.exit_threads(false) {
25:   Ok(t) => (),
26:   Err(e) => …
27: }
28:
29: match pool.exit_threads(true) {
30:   …
31: }
```

Code 17 - Example of *exit_threads* and *get* methods

In Code 17 we can see an example of the creation of a thread pool with five threads, where a periodic job is being run on thread zero. Then, the user uses the method *get* to retrieve information on the scheduling parameters of the threads inside his thread pool. This function lets the user send as an argument an optional value which is a *vector* with all the threads they want to have the parameters from. In this case, the user gets the *SchedulingParameters* for threads zero and two stored in his local variable *sched_params_0-2*. If the user chose to not send this, the return would be the parameters of all five threads. Then, by running the method *exit_threads* and by sending *false* as an argument, we chose to exit all threads that are not running any periodic tasks, which in our case related to every thread except the first one. After this, by running the same method but setting the variable as *true*, the remaining thread, zero, will also exit.

When working with threads, both being inside thread pools or outside, some useful tools to have would be methods that are able to retrieve information about the thread the code is running on, without having to use unsafe code by calling other libraries like *libc*. To enable this for the developer, the *rt_lib* has the *get_policy*, *get_cpu*, *get_priority*, and *get_tid*. By running these methods, the user gets the policy, cpu, priority, or the identifier, respectively, for the thread they want. The get methods for the cpu and thread identifier do not require any arguments to run, but, on the other hand, the methods for the priority and policy require the user to send as argument the identifier of the thread they want to get the information from.

4.2 Behind the Scenes of *rt_lib*

To deliver the level of simplicity we aimed for, a lot of the complexity involved in thread creation had to be hidden from the user. This includes everything from low-level calls to the *libc* library to the internal handling of thread attributes and scheduling policies. In Figure 4 we can get a better insight of the complexity that exists inside the library, organised inside each module of the *rt_lib*.

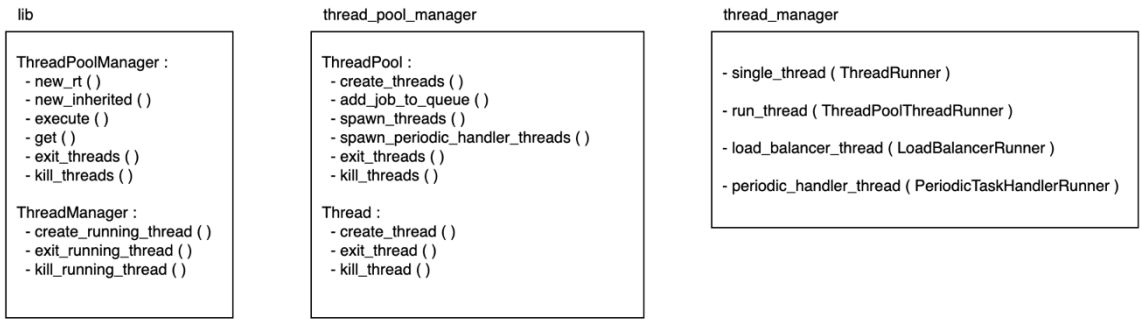


Figure 4 - Structure inside the *rt_lib* modules

The *lib* is the one where the *ThreadPoolManager* and the *ThreadManager* structures reside, together with the methods that are part of it. This module then has access to the *thread_pool_manager* which includes the *Thread* and *ThreadPool* structs, initially presented in Chapter 0, that are not accessible to the user and is where the data regarding the created threads and thread pools are managed. Here a collection of methods were created for each struct which are going to be addressed in this section. Finally, the *thread_manager* module, which is the one

where the methods that run inside each created thread reside, will also be dove into, explaining how these methods work.

In the case of an isolated thread, when the user tries to create one using *create_running_thread*, the *rt_lib* takes on a series of tasks to configure and launch the thread behind the scenes. The process starts with the initialization of the thread's attributes using POSIX's *pthread_attr_t* struct. This structure holds all the necessary configuration details, and its configuration makes sure it is explicitly set so that the new thread does not inherit scheduling settings from its parent. If the user has specified a target core in the scheduling parameters, the library sets the thread's CPU affinity accordingly, pinning it to that core. After that, we configure the scheduling policy. If the user specified a scheduling class, we convert it into the corresponding *libc* constant and set it in the thread attributes. Finally comes the priority setting, which is also passed to the thread through these attributes, making sure it aligns with the chosen scheduling policy.

Once all the attributes are set up, we prepare the job that the user passed in. This job, defined as a closure, is wrapped together with the period, if one was specified, and a shared flag used to control whether the thread should be exited. This flag is how we control when to exit the thread once in case the user tries to call that method. This whole package is then boxed and transformed into a raw pointer so it can be safely passed to the system call. The actual creation of the thread is done using a create method form *libc*, which receives the attributes and the job. If everything goes well, the thread is created and its ID is stored in the thread struct. If not, we make sure to clean up any initialized attributes and return a descriptive error.

To be able to run the job inside the created thread but also be sure that, if it is periodic, it continued to run using that same periodicity, a *single_thread* method was created. From the moment the thread is launched, it begins executing this function, which is the main loop for that isolated thread. Inside this loop, the thread checks if it should terminate by inspecting the shared exit thread flag. If that flag is set to true, the thread breaks the loop and exits gracefully. If not, it proceeds to execute the user's job. After executing the job, the thread checks whether a period was specified. If a period was set, it sleeps for that duration which leaves the CPU free to run any other tasks, and then loops back to check the termination flag again. If no period was given, it means the job is a non-periodic task, and the thread exits right after executing it once. This type of methods that correlate to the work that runs inside a thread are included in the *thread_manager* module inside our library.

All the work around the flags and jobs are conducted by the use of lock mechanisms, which guarantee that even though multiple threads can be running independently, access to the job and flag remains safe and synchronized.

In the end, what the user gets is a fully configured real-time thread, running the job they defined, either periodically or just once, without having to touch unsafe code or deal with low-level details. All this code is managed inside the *Thread* struct on the *thread_runner* section, except for all the calls to unsafe code using *libc*. These slices of code are stored separately inside the *libc_abstractor* module to keep the code cleaner and without the user of unsafe blocks inside the main modules.

The method *exit_running_thread* is also included inside the same struct and, when called, sets the exit flag sent to the created thread to true. This is what will trigger the thread to stop once no job is running and terminate safely. On the other hand, the *kill_running_thread* makes use of the thread identifier, which is stored inside the *Thread* struct, to kill its process and terminate the thread abruptly.

In contrast, to be able to manage a thread pool while giving the user the means to manage the work on each thread separately, its implementation needed a lot more complexity. While creating a new thread pool, the *rt_lib* utilizes the same logic for either *new_rt* and *new_inherited* methods. This is done by creating a group of *SchedulingParameters*'s which is then passed on to a flow of methods present in the *thread_runner* module. These functions handle the thread's creation and grouping in the struct *ThreadPoolManager* that is sent to the user to operate with, if successful.

When a thread pool is created, a queue for each thread is generated. It is in these queues where the jobs assigned to each thread are going to be stored to then be consumed and run. Apart from a queue to each thread, a global queue that will be used to store tasks that are executed without a specific thread assigned to them is also included. Along with the queues, the threads themselves are also created as soon as the thread pool is, by using the parameters set by the user. Along with these, an extra thread only known to the system is also generated. This extra thread is what is called within the *rt_lib* as the load balancer, and it is the thread responsible for managing the global queue. When it has tasks in its queue, this thread looks at all the other thread's queues to detect when one of these becomes empty to then move the first task in it to that respective queue. This means that this thread never runs any tasks sent by the user, only working as a literal load balancer for the ones that are not assigned a specific thread.

When creating the threads for the thread pool, a struct named *ThreadPool* is created for it. This struct holds all the queues, including the load balancer one, all the identifiers of the threads running on that thread pool and two flags used for exiting the threads, one for the non-periodic, and one for the periodic ones. The creation of the threads is very similar to the implementation for the isolated ones, but in this case, this is done by iterating all *SchedulingParameters*, including one set created for the load balancer thread. The steps for setting the parameters of each thread follow the same flow as in the previous implementation.

When creating the load balancer thread, the system generates a new struct named *LoadBalancerRunner* to send to it. This struct includes all the queues created for the thread pool, the flag for exiting periodic threads, since this is the one thread that will be always running as long as at least one other thread is also alive inside the thread pool, and the *ThreadPool* struct, from which the thread pool itself is part of. The latter is needed since this thread needs to be able to utilize methods only available inside that struct. This struct is then boxed to be sent to the thread where the load balancer will run.

Regarding the remaining threads, which are the ones the user will be working with, the system generates a different struct named *ThreadPoolThreadRunner* for each one, that will be also boxed and sent to the respective threads. This struct contains the task queue for the respective thread, its own identifier within the thread pool, its own *ThreadPool* struct to also be able to use methods implemented in it, and both flags created for exiting the thread for the case where it has a periodic job running in it, and in case it has not.

Just like for the isolated threads, a method was created for both thread types of a thread pool. These methods are the ones responsible for managing how the threads will handle the jobs sent by the user.

In the load balancer case, the *load_balancer_thread* method is the one that manages it. This method consists of a loop where every iteration begins by checking the exit periodic tasks flag to know whether it should stop running. This shared value allows the system to gracefully shut down the load balancer without abrupt interruption. If the stop signal is not present, the thread moves on to verify whether there are jobs in the global queue. If there are no jobs available, the thread briefly sleeps and re starts the loop to avoid unnecessary CPU usage. If a job is found, the thread scans the available queues to find one that is currently empty, aiming to distribute workload evenly and avoid bottlenecks. If an available thread is found, the job is moved from that queue to the respective one, otherwise it retries the loop again without removing the job from its queue. If the job is marked as periodic it checks whether a handler thread has been created for it. This handler thread for periodic tasks will be talked about later in this chapter. This process ensures that jobs are dynamically assigned to threads based on availability, allowing the system to scale and adapt to varying loads without requiring the user to handle every manual distribution.

For the threads that will be running the tasks, the method created is the *run_thread*. This function represents the core behaviour of what a running worker thread actually does during its lifetime. Inside the main loop, the thread first checks whether it should terminate. This is done differently depending on whether the thread is currently handling periodic or aperiodic tasks, determining to which flag the thread will listens to. This design allows different categories of threads to be gracefully shut down through distinct triggers. If no termination condition is met, the thread proceeds to check its associated queue for work. If the queue is empty, it briefly sleeps before looping again, helping reduce CPU usage during idle times. If there is a job available, the thread starts by checking if it is periodic and if a handler thread has already been spawned for it. If not, it marks the thread as running a periodic thread, making it unable to exit without the corresponding flag. This check ensures that each periodic job is only initialized once, avoiding redundant or conflicting behaviour. The thread then locks and executes the job closure. Once finished, the job is removed from the queue, and the thread loops again, ready to handle the next task.

In conclusion, when creating a thread pool the system creates a group of queues, one for the load balancer and one for each thread the user wants. Then, using those, the system creates all the threads associating them to the corresponding queue. These threads will be the ones responsible for handling and running all the tasks the user tries to create. Figure 5 helps illustrate this.

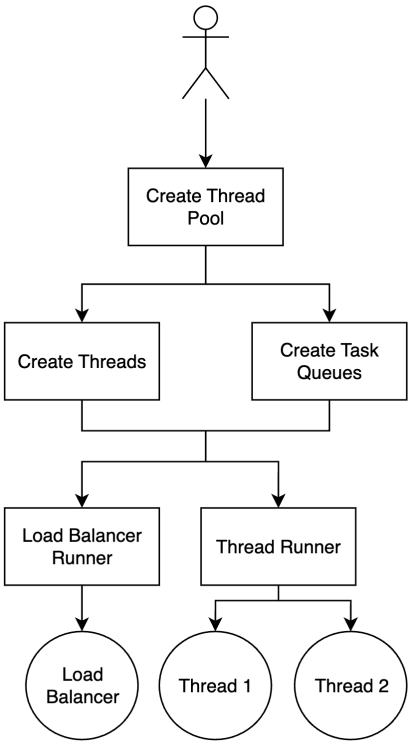


Figure 5 – How threads are created within a Thread Pool

After creating the thread pool, the *ThreadPoolManager* struct is now available for the user. This struct's arguments are hidden from the user, but they include the *ThreadPool* struct created for that thread pool where all the data for it, including task queues and exiting flags. With this now available, the user can run the other available methods for it, the most important being the *execute*.

This method calls another function inside the *ThreadPool* struct named *add_job_to_queue* where the real implementation resides. When this function is called, it first wraps the user's job in a *Mutex* which ensures that the job can be safely shared and executed across threads. The method then checks whether the user specified a target thread and if it is valid. If so, the job is added to the top of that specific thread's queue along with its associated periodicity, set as zero if not set by the user, and a flag that tracks whether a periodic handler has already been spawned, only used when a period is also present. If no thread identifier is provided, the job is instead pushed to the top of the load balancer's queue. This queue is reserved for unsorted jobs and is monitored by the load balancer thread, which will later redistribute the job to the first available worker. Once added to the queue, the method exits successfully, and the job becomes visible to the runtime threads waiting for work.

In both cases, the job is stored as a *tuple* variable, which holds three different variables inside a closure. This containing the task itself, the optional periodic value which is set to zero if not preset, and the flag used internally to tack if a handler thread was already created for the job if it has a period associated with it. Inside both *load_balancer_thread* and *run_thread*, this variable is used for running the task, but also for managing periodic tasks. In any iteration of the loops inside both functions, if the job comes populated with a value for the period different from zero and the flag for the periodic task handler set to false, then it starts a flow to create a new thread that will handle

that task's work and sets the flag to true, indicating the next iterations that there is no need to run the flow again.

This new handler thread is created using the method *spawn_periodic_handler_threads* inside the *ThreadPool* struct, and, just like the other spawn thread methods, it starts by setting the thread's parameters. This method, when called, takes as arguments the job *tuple* and the thread identifier for the thread in which the task is intended to run, it being from either the load balancer thread or any other thread. These are then stored inside the new struct named *PeriodicTaskHandlerRunner*, together with the shared flag for exiting threads running periodic tasks. The thread is then created, and another method was created for it to run on.

The *periodic_handler_thread* manages the work of a periodic task. Since the thread that created the periodic task has already run the job the first time, it starts its loop by sleeping for the time of the period set by the user. Then it looks into the shared flag for exiting threads running periodic tasks to exit gracefully if it is set to true. Finally, it accesses the shared queue for it corresponding thread and adds the job to the top of that queue before looping back. Similarly to the load balancer, this thread does not run any task sent by the user in it, only managing the periodic task and queue associated with it.

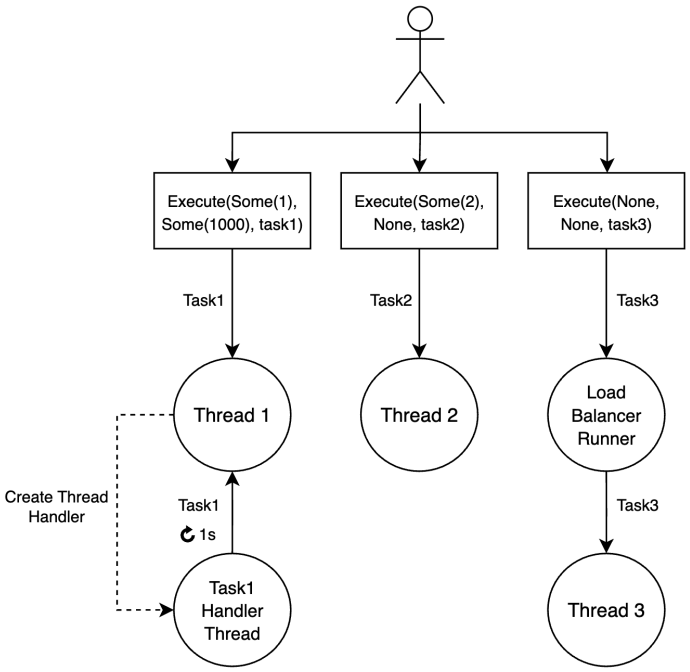


Figure 6 – How tasks are handled within a Thread Pool

In Figure 6 we can see how the *execute* function will work on the background when the user tries to run tasks on a thread pool they created. For this illustration we assume a thread pool with three threads. When the user tries to execute a task on thread one with a periodicity of one second, the job is firstly sent to the corresponding thread, and, when identified as periodic, the creation of a thread handler for that task is created. The thread one will run the job the first time and, from then on, the thread handler will re deploy that same job to it each one second so that thread will run that job again. On the second example, a job is sent to thread two without a defined period. In this

case, it will only run the job once and the thread will be on standby waiting for other tasks. For the third example, a task without an assigned period is sent without specifying any thread to run it on. In this case, the task will be sent to the load balancer thread. This thread will look into the queues of all the threads in the thread pool and see that thread one is running task1, thread two is running task2, but thread three is empty. In this case, it will then send task3 to thread three's queue and the flow will run as in the example for task2.

Finally, to shut down the running threads gracefully, the user can run the *exit_threads* method which itself runs a different function inside the *ThreadPool* struct. This method reaches into the struct itself and accesses the *running_threads_id* variable inside it. This variable is available and can be accessed by all threads and thus it is safely protected to avoid sharing issues. All threads, including the load balancer and periodic task handlers, when created, have their process identifier saved inside it. This is very important because, when the user tries to exit all the threads, the *exit_threads* method loops every thread identifier on the list and, after setting the flags for exiting all threads, joins each of them. Joining, in this case, means that the method will wait until the corresponding thread has been terminated. To avoid having threads missing from this method, the *running_threads_id* is always checked and then freed in every loop iteration. This allows currently running threads to not be stuck trying to access it if trying to create new threads. These newly created ones will also be added to the variable and thus, the loop will end up also iterating them.

If the user tries to only exit the threads not running any periodic task, the method only sets the corresponding flag to *true* and those threads will then exit, not waiting to join any of them. These threads will also not be removed from the running threads pool but that's not a problem since the method *join* does not fail when trying to wait for an already non-existent thread, instead it immediately skips the join and continues.

The method *kill_threads* is also implemented inside the *ThreadPool* struct and what it does works similarly to the exit method. This also iterates the values inside *running_threads_id* variable, following the same logic of safely accessing each value in its list. But instead, it just kills the running process. This will abruptly exit the threads, potentially leading to memory leaks and which going strongly against the philosophy of the *Rust* language.

Inside all these methods created inside the *thread_runner* section, several different functions of *libc* are used. These methods are not managed by *Rust* since *libc* is an external library that comes from *C* language which has no safety guarantees. Also, *Rust* can't enforce its usual protections around memory and concurrency and so, by using these methods, we are bypassing *Rust*'s type system and safety checks. Because of it, we need to use *unsafe* block which disable these checks. To have the library clean of this block of code, all the *libc* accesses are managed inside the *libc_abstractor* section. This also helps to keep the code cleaner to read and easier to manage without risking problems regarding memory access inside these methods.

4.3 Summary

This chapter explains the design and structure of the API created for real-time parallel systems with Rust, the *rt_lib* library. It highlights its architecture, and its management of real-time threads and thread pools. The main objective of *rt_lib* is to simplify real-time thread management by abstracting low-level complexities, typically requiring unsafe operations or direct interaction with the *libc* library, while maintaining efficient and safe memory management. The *rt_lib* design is inspired by Rust's *ThreadPool* crate, but extends its functionality to real-time systems, offering enhanced control and customization for the users.

This chapter is divided into two main sections, the presentation of the functionalities the library has for creating and managing threads and the behind the scenes, where it is presented how the *rt_lib* works within the library to manage the thread systems.

The first section starts by explaining how this library is organised. This is relevant to help understand our thought process when developing the library. In the second section the focus then shifts to how *rt_lib* internally manages the creation and control of real-time threads, both isolated and grouped, while keeping the interface clean and user-friendly.

Isolated threads are configured through an high-level method that abstracts low-level system details, such as setting CPU affinity, scheduling policies, and thread priorities. Jobs are defined as closures and wrapped with metadata to support optional periodic execution. Once launched, threads run inside a managed loop that checks for termination flags and, if periodic, sleeps between executions. Thread lifecycle methods allow safe or immediate termination, and all interactions with unsafe code are contained in a separate abstraction layer, ensuring the core logic remains idiomatic and safe.

For more advanced workloads, the library also supports the creation of thread pools. Each pool consists of multiple worker threads, each with its own task queue, along with a dedicated load balancer thread that redistributes tasks when no specific target is defined. This design avoids the contention of a global queue and allows users to assign tasks to specific threads. If a task is periodic, a handler thread is spawned to periodically re-queue it, ensuring consistent timing without overloading the main worker. All threads share synchronized access to their identifiers and control flags, enabling safe shutdown.

In summary, *rt_lib* offers a thorough solution for managing real-time threads and thread pools in Rust. It simplifies thread creation and management by providing a simple interface while utilizing *libc* for low-level operations. The library supports both specific and global task queues, periodic task handling, and load balancing, making it a robust tool for real-time concurrent systems. *Rt_lib*'s design prioritises flexibility and safety, ensuring that users can manage complex thread-based systems with ease, without the need to interact directly with error prone and unsafe code.

5 Results

In this chapter, we present practical examples that demonstrate the effectiveness and simplicity of the *rt_lib* library. Rather than relying solely on performance benchmarks or abstract metrics, the focus here is also on showcasing how real-time thread management can be expressed through clean and minimal code. These examples highlight how the abstractions provided by *rt_lib* allow developers to implement complex scheduling behaviours and thread interactions with ease, without having to manage low-level details themselves. Through these scenarios, we aim to validate the design goals of the library: safety, usability, and clarity.

Showcasing these examples will also help clarify the second part of the result chapter that will focus on the performance benchmarks of this library. This section will make use of the example shown on the first part so an overall overview of them is beneficent.

At this point we are also able to revisit the user stories presented in Chapter 3.1 which help us evaluate the successfulness of *rt_lib* in regards of the programmer needs for real time thread management. These user stories are again presented in Table 1 together with the solutions for each one.

Considering that each user story outlined is met with a direct and functional solution within *rt_lib*, the table reinforces the library's suitability for real-time thread management. For example, core real-time requirements such as setting scheduling policies (US1), defining CPU affinities (US2), and assigning thread priorities (US3) are addressed through the *SchedulingParameters* struct, giving developers fine-grained control over thread behaviour. More advanced needs, such as flexible thread pool configuration (US5), scheduling inheritance (US6), and periodic task execution (US8), are also supported, enabling a wide range of real-time scenarios. Additionally, *rt_lib* provides essential lifecycle and observability tools, such as selective shutdown options (US9) and safe querying of thread attributes (US10, US11), which are particularly helpful during debugging or system monitoring. Overall, this comprehensive coverage demonstrates that *rt_lib* was designed with practical programmer needs in mind and supports not only basic threading but also complex, timing-sensitive applications.

Table 1 – User Stories Evaluation

User Story	Programmer Need	Solution
US1	Customize the scheduling policy of a thread	The *SchedulingParameters* struct lets the programmer set scheduling policies
US2	Bind threads to specific CPU cores	The programmer can set CPU affinity via the *SchedulingParameters* struct
US3	Set thread priorities individually	Each thread accepts an individual priority level through the *SchedulingParameters* struct
US4	Run an isolated thread	The programmer can run an isolated thread using the *create_running_thread* method
US5	Flexibility in thread pool creation with shared or individualized parameters	Using *SchedulingParameters*, we can create thread pools with both shared and individualized sets of parameters using the *new_rt* method
US6	Inherit scheduling settings from parent/current context	The *rt_lib* lets the programmer create a thread pool by inheriting its parent's parameters by using *new_inherited* methods
US7	Assign jobs to specific threads or allow the system to assign them dynamically	The *execute* methods can both assign a job to a specific thread inside a thread pool, or if not set, let the decision to the system
US8	Schedule periodic jobs with a fixed time interval	The *execute* can take as argument a value for the period in milliseconds
US9	Fine-grained control over thread shutdown, including selective termination of non-periodic threads	The *exit_threads* and *kill_threads* methods let the programmer have full control on how the threads will terminate and give the option to not terminate periodic running tasks
US10	Helper functions to query thread attributes	The method *get* give the option to query for the attributes of any or all threads in a thread pool
US11	Inspect parameters of a thread	The methods *get_policy*, *get_cpu*, *get_priority* and *get_tid* let the programmer query for the attributes of the thread which calls them

5.1 Implementation Examples of *rt_lib*

We will first focus on showcasing the implementations provided by the library for creating and managing isolated real-time threads. These examples demonstrate how developers can use the *ThreadManager* abstraction to spawn individual threads with fine-grained control over their scheduling parameters, such as priority, core affinity, and scheduling class. By isolating each thread, it becomes easier to evaluate the behaviour and performance of specific configurations in a controlled manner. This approach also serves to highlight the simplicity and clarity offered by the

library's interface, which abstracts away much of the low-level setup typically required in real-time systems programming. The following examples progressively grow in complexity, from minimal thread instantiation to more complex strategies that will be used to monitor performance in a later stage.

The first example, Code 18, demonstrates how effortlessly a real-time thread can be created and managed using the *rt_lib* interface. With just a few lines, the user defines the scheduling parameters, sets an optional period, and passes a closure as the job to be executed. What sets this apart from many existing threading libraries is the direct control over real-time attributes, like policy (e.g., FIFO), priority, and CPU core affinity, without needing to interact with low-level code. Moreover, introspective helper methods such as *get_tid*, *get_policy*, *get_cpu*, and *get_priority* provide runtime insight into the thread's configuration. This gives the user not just real-time capabilities, but also transparency. Other libraries often abstract away this control or require other complex bindings to access it, whereas *rt_lib* exposes it in a safe and declarative way. This simplicity makes the creation of predictable and monitorable real-time threads accessible even to users without deep expertise in Linux scheduling or low-level thread management.

```
1:  let sched_param = SchedulingParameters {
2:      class: Some(SchedulingClass::FIFO),
3:      core: Some(0),
4:      priority: 10,
5:  };
6:
7:  let no_period = None;
8:
9:  let job = || {
10:     println!(Running periodic job with TID: {}, Policy: {}, CPU: {},
11:      Priority: {}", get_tid, get_policy, get_cpu, get_priority);
12: };
13:
14: let mut thread = match ThreadManager::create_running_thread(sched_param,
15:  no_period, job) {
16:     Ok(t) => t,
17:     Err(e) => println!("Failed to create thread: {}", e),
18: };
19:
20: sleep(std::time::Duration::from_secs(1));
21:
22: match thread.exit_running_thread() {
23:     Ok(_) => (),
24:     Err(e) => println!("Error exiting thread: {}", e),
25: };
```

Code 18 - Creating and Managing a Thread with *rt_lib*

The previous example showcased a simpler real-time thread scenario that will serve as a baseline for performance tests. The task here is a simple print operation, serving as a minimal and controlled case. This kind of setup is especially useful for baseline benchmarking, where overhead, activation latency, or scheduling behaviour can be isolated and measured with minimal interference. Building on that, the next example, Code 19, demonstrates how *rt_lib* enables a real-time parent thread to dynamically spawn multiple child threads, each configured with their own scheduling parameters. This highlights the library's flexibility, allowing nested real-time thread creation within running threads, managed through straightforward and safe abstractions rather than complex system calls.

From a usability standpoint, this approach significantly reduces the complexity typically involved in managing real-time threads, making sophisticated scheduling and concurrency patterns accessible without sacrificing control. It demonstrates how *rt_lib* handles real-time scheduling for nested, programmatic thread creation, enabling more dynamic workloads while still maintaining predictability and control. This pattern is valuable for stress testing and periodic workload evaluation, helping us assess how the system performs under more complex real-time execution flows, where threads may be launched at runtime. It also helps test hierarchical management, CPU affinity behaviour, and scheduling interference across cores. Furthermore, this example sets the stage for advanced performance benchmarking by creating a realistic multitasking environment, enabling evaluation of thread scheduling, priority management, and CPU affinity in a controlled, real-time context.

```
1:  parent_params = SchedulingParameters {
2:    class: Some(SchedulingClass::FIFO),
3:    core: Some(1),
4:    priority: 80,
5:  }
6:
7:  parent_period = Some(1000)
8:
9:  parent_job = move || {
10:   for i in 0..3 {
11:     child_params = SchedulingParameters {
12:       class: Some(SchedulingClass::FIFO),
13:       core: Some(1),
14:       priority: 70 + i,
15:     }
16:
17:     child_period = Some(500)
18:
19:     child_job = move || {
20:       println!("Child thread {} is running", i);
21:     }
22:
23:     child_thread = match ThreadManager::create_running_thread(child_params,
24:      child_period, child_job) {
25:       Ok(mut child_thread) => child_thread,
26:       Err(e) => println!("Failed to spawn child: {}", e),
27:     }
28:   }
29:   sleep(std::time::Duration::from_millis(500));
30: }
31:
32: parent_thread = match ThreadManager::create_running_thread(parent_params,
33:  parent_period, parent_job) {
34:   Ok(mut parent_thread) => parent_thread,
35:   Err(e) => println!("Failed to create parent thread: {}", e),
36: }
37:
38: let _ = parent_thread.exit_running_thread();
```

Code 19 - Creating Nested Real-Time Threads with *rt_lib*

After showcasing how *ThreadManager* allows for the creation and management of isolated real-time threads with fine control, we now move toward exploring *ThreadPoolManager*, the other key abstraction provided by *rt_lib*. In the following examples, we demonstrate how to configure and use the real-time thread pool, starting with a simple static setup and gradually increasing in

complexity. These examples continue to emphasize the clarity and simplicity of using the library, while also being designed to serve as the foundation for performance benchmarks in the next phase of this work.

This third example, Code 20, showcases the creation of a thread pool with three real-time threads, all configured identically with a FIFO policy, pinned to the same core, and assigned a fixed priority. A periodic task is defined and dispatched to each of the threads using *pool.execute*, followed by a clean exit of all threads. The design of the *rt_lib* interface ensures that this setup can be written in just a few lines, with minimal effort and strong safety guarantees. What typically requires intricate control with native *POSIX* threads is here reduced to straightforward method calls.

```
1:  params = vec![
2:     SchedulingParameters {
3:         class: Some(SchedulingClass::FIFO),
4:         core: Some(1),
5:         priority: 50,
6:     }
7:  ]
8:
9:  number_of_threads = 3
10:
11: pool = match ThreadPoolManager::new_rt(number_of_threads, params) {
12:     Ok(p) => p,
13:     Err(e) => println!("{}", e),
14: }
15:
16: job = || {
17:     // Some periodic task logic
18: }
19:
20: period_ms = Some(100)
21:
22: for i in 0..number_of_threads {
23:     match pool.execute(Some(i), period_ms, job) {
24:         Ok(_) => (),
25:         Err(e) => println!("{}", e),
26:     }
27: }
28:
29: match pool.exit_threads(true) {
30:     Ok(_) => (),
31:     Err(e) => println!("{}", e),
32: }
```

Code 20 - Creating a *ThreadPoolManager* with uniform threads with *rt_lib*

This setup will serve as a baseline in the benchmarking phase to measure how a system behaves when executing equal tasks on identical threads. It helps us evaluate thread dispatch, scheduling accuracy, and overhead under optimal symmetry.

In the following example, Code 21, each of the three threads in the pool is configured with its own real-time policy, CPU core affinity, and priority. One uses FIFO on core zero with high priority, another uses RR on core one with a lower priority, and the third returns to FIFO with a mid-level priority. This flexibility showcases a core strength of *rt_lib*. It allows the user to describe complex real-time scenarios briefly while hiding the unsafe, error prone low-level setup. This is especially important when working with real-time constraints across heterogeneous systems.

From a testing perspective, this example allows us to measure how the system handles and prioritizes tasks across different policies and cores. It's ideal for identifying priority inversion risks or imbalances in scheduling fairness under mixed conditions. In the execute method, we also do not specify any thread, this way we can also check how well does the system partition its load when the user does not set it himself.

```
params = vec![
    SchedulingParameters {
        class: Some(SchedulingClass::FIFO),
        core: Some(0),
        priority: 70,
    },
    SchedulingParameters {
        class: Some(SchedulingClass::RR),
        core: Some(1),
        priority: 40,
    },
    SchedulingParameters {
        class: Some(SchedulingClass::FIFO),
        core: Some(2),
        priority: 60,
    }
]

number_of_threads = 3

pool = match ThreadPoolManager::new_rt(number_of_threads, params) {
    Ok(p) => p,
    Err(e) => println!("{}", e),
}

job = || {
    // Some periodic task logic
}

for i in 0..number_of_threads {
    match pool.execute(None, Some(500), job) {
        Ok(_) => (),
        Err(e) => println!("{}", e),
    }
}

match pool.exit_threads(true) {
    Ok(_) => (),
    Err(e) => println!("{}", e),
}
```

Code 21 - Creating a *ThreadPoolManager* with different threads with *rt_lib*

The last example, Code 22, shows how simple it is to launch a real-time thread pool using inherited attributes. Instead of manually specifying policies, cores, and priorities, the user leverages the current process' configuration, a powerful feature for scenarios where system-wide setups are already enforced or required by design. The code remains minimal and readable while providing full access to real-time execution behaviour, making it a good choice for developers who prioritize convenience without sacrificing control. In this example's full implementation, we first create an isolated thread for the thread pool to inherit its parameters from.

For our performance tests, this example serves as a useful baseline. Since all threads share the same inherited settings, we can observe how uniform scheduling affects execution timing and consistency, helping us compare against more customized configurations from previous examples while validating the inheriting capabilities of the library.

```
1:  number_of_threads = 4
2:
3:  pool = match ThreadPoolManager::new_inherited(number_of_threads) {
4:     Ok(p) => p,
5:     Err(e) => println!("{}", e),
6: }
7:
8:  job = || {
9:     // Some periodic task logic
10: }
11:
12: for i in 0..number_of_threads {
13:     match pool.execute(Some(i), Some(1000), job) {
14:         Ok(_) => (),
15:         Err(e) => println!("{}", e),
16:     }
17: }
18:
19: match pool.exit_threads(false) {
20:     Ok(_) => (),
21:     Err(e) => println!("{}", e),
22: }
```

Code 22 - Creating a *ThreadPoolManager* with inherited properties with *rt_lib*

After demonstrating examples of how the library can be used to create and manage both isolated threads and thread pools, we now move on to validate its performance. The examples shown in this subchapter will serve as the basis for testing the timing accuracy, overhead, and real-time behaviour of the system.

5.2 Performance Evaluation

The *rt_lib* library is designed specifically for real-time programming, where timing guarantees, responsiveness, and determinism are critical. In real-time systems, performance is not just a matter of speed, it is a fundamental requirement for system correctness and reliability. As such, conducting a thorough performance evaluation of the library is essential to assess whether it meets the standards expected in real-time applications.

This chapter presents a series of performance tests aimed at analysing key aspects of the library's behaviour under different configurations and workloads. The focus is placed on metrics such as thread creation time, execution duration, scheduling latency, and task distribution across processing cores. These tests help determine the library's efficiency, predictability, and suitability for use in time-sensitive systems. By measuring these parameters, we aim to highlight both the strengths and potential limitations of *rt_lib* in real-world real-time environments.

To help us evaluate and to gather results, the examples above were slightly modified to collect timestamps and iterate on the data collected. Some of the tools used for this were the *std* and the *starts* libraries. The first one is the standard library included within *Rust*, this was already being used, but some other aspects had to be added to the example for us to be able to evaluate it as intended. The *Instant* property was used to register and collect different timestamps, and the *Mutex* was needed so that we could pass values from inside the jobs that were sent to the threads to our main code. Regarding *starts*, this is a statistics and probability library that provides tools for descriptive statistics that we wanted. Some of this are the average, minimum, maximum and mean values, which we used to collect the data from our runs.

For every example shown above, we isolated parts of the code, like the creation of a thread or thread pool, the deploy of a task or the execution time of a whole section of code. For each section we created an *Instant* variable to gather its running time and collected the values into different lists. We chose to run each example one hundred times to have a diverse number of results to perform evaluations on. This was especially needed because the tests were being run inside a virtual machine with limited resources which makes our tests unreliable at times due to the system being shared with the main one. With all this, at the end, we had an array of at least one hundred examples for each case we wanted to test. These values were then iterated using the methods from the *starts* library for us to evaluate.

5.2.1 Experimental Setup

The experiments were conducted on a desktop machine equipped with an Intel Core i7-9700K CPU with 8 cores @ 3.6 GHz, no hyperthreading and 16 GB of RAM, running a virtual machine. The virtual machine was running on Ubuntu with a 4 core CPU and 8GB of RAM.

In addition to the evaluation of *rt_lib*, we also implemented two simple baseline benchmarks to provide context for our results. The first benchmark measures the cost of creating a single thread using C language with *libc* primitives. The second benchmark measures the creation and usage of a small thread pool using the widely used Rust *threadpool* crate. Both experiments were also executed one hundred times, collecting creation times and total execution durations. These results, presented in Table 2 and Table 3 serve as reference points, allowing us to compare the overhead and performance of *rt_lib* against simpler threading approaches.

Table 2 - Benchmark result for Thread Creation in C language

Metric	Creation Time
Count	100
Minimum	0,104 ms
Maximum	0,919 ms
Average	0,318 ms
Median	0,284 ms
Deviation	0,182 ms

Table 3 - Benchmark result for Thread Pool Management using Rust's *Threadpool* crate

Metric	Thread Pool Creation Time	Individual Job Launch Time
Count	100	300
Minimum	0,066 ms	0,000 ms
Maximum	17,559 ms	8,205 ms
Average	2,299 ms	0,131 ms
Median	1,173 ms	0,001 ms
Deviation	3,071 ms	0,721 ms

It is important to reiterate that all tests were performed on a virtual machine which comes with both advantages and trade-offs. On the one hand, VMs offer isolation and reproducibility, but on the other hand, they introduce scheduling uncertainty and resource sharing that can distort timing sensitive measurements. For example, the variance observed in thread creation and execution times is likely a consequence of VM-level scheduling. While such behaviour does not invalidate the results, it suggests that even better performance and tighter timing could be expected on bare-metal systems. Therefore, future work should include tests on physical hardware to fully validate the library's real-time capabilities.

5.2.2 Experiments

Starting with the first experiment, Code 18, the goal was to measure the basic cost of creating threads and running tasks within them. The test involved spawning a thread tasked with executing a simple workload, designed to mimic real-time behaviour. Results, presented in Table 4, showed that thread creation times were consistently low, with most values under one millisecond. Execution durations averaged around a little over one second but looking at the median, it gets even closer to the target value, aligning well with the intended workload timing set with the *sleep* function. There was a noticeable, though moderate, standard deviation in execution times which can be explained by the non-deterministic scheduling behaviour of virtual machines. Occasionally, execution times peaked to over two seconds and a half, suggesting temporary contention for CPU resources. However, such outliers remained relatively rare. When compared to the C benchmark, Table 2, where average thread creation time was measured at 0,318 ms, our results are very much in the same range. This provides an external confirmation that thread creation costs in *rt_lib* are competitive with the baseline provided by C.

Table 4 - Code 18 Performance Statistics

Metric	Thread Creation Time	Execution Duration
Count	100	100
Minimum	0,100 ms	1,000410 s
Maximum	5,868 ms	2,664832 s
Average	0,370 ms	1,064884 s
Median	0,200 ms	1,003677 s
Deviation	0,769 ms	0,227675 s

The second example, Code 19, introduced a parent-child relationship between isolated threads. Parent threads were responsible for spawning and coordinating three child threads, each with its own periodic task, leading to three hundred samples for child creation time. This configuration is relevant for real-time systems where a central controller might dispatch and synchronize lower-level tasks. Performance data, Table 5, indicated that child thread creation was exceptionally fast, with minimal deviation. Parent thread, on the other hand, exhibited more variability but the majority of values still sat under two hundred milliseconds. Execution durations were longer than in the previous test, typically ranging between 4 to 10 seconds, this is likely due to the increased complexity of coordinating multiple child threads. Overall, the test confirmed that the library handles hierarchical thread structures well, although the added coordination introduces some cost. Compared with Table 2 results, child thread creation times are even lower than the C baseline, showing that the library efficiently manages lightweight threads under a parent–child model.

Table 5 - Code 19 Performance Statistics

Metric	Parent Thread Creation Time	Parent Execution Duration	Child Thread Creation Time
Count	100	100	300
Minimum	0,047 ms	4,303785 s	0,015 ms
Maximum	5,150 ms	10,218855 s	0,297 ms
Average	0,356 ms	5,689450 s	0,037 ms
Median	0,158 ms	5,491274 s	0,035 ms
Deviation	0,733 ms	1,072915 s	0,028 ms

In the third experiment, Code 20, the focus shifted to the performance of the thread pool system. A pool of threads was created with three threads, after which jobs were dispatched to them. The idea was to test whether the pooling mechanism, which avoids repeatedly creating and destroying threads, would result in lower overhead and more predictable behaviour. As can be seen on Table 6, thread pool creation times varied more significantly than expected. While most creations were fast, taking less than three milliseconds, outliers reached as high as thirty-eight milliseconds which may be justified by the VM's resource allocation during rapid spawning of multiple threads. Despite this, job execution durations were very consistent and stayed close to the one second target.

Individual job dispatch times were negligible, usually measured in microseconds which supports the notion that once the thread pool is initialized, the cost of launching jobs is extremely low. Equating these results to Rust's *threadpool* crate benchmark, Table 3, we see that while our average pool creation time is slightly higher than the baseline, the median values remain in the same order of magnitude. Job dispatch times in our tests were also negligible and well aligned with the 0,131 ms average reported for the external benchmark, supporting the conclusion that job launching overhead is consistently low across different implementations.

Table 6 - Code 20 Performance Statistics

Metric	Thread Pool Creation Time	Total Execution Duration	Individual Job Launch Time
Count	100	100	300
Minimum	0,199 ms	1,017868 s	0,000 ms
Maximum	38,190 ms	1,292063 s	0,400 ms
Average	5,129 ms	1,087325 s	0,003 ms
Median	3,031 ms	1,089092 s	0,001 ms
Deviation	5,714 ms	0,039506 s	0,023 ms

The fourth test case, Code 21, expanded the load slightly by increasing the complexity and number of iterations within the test loop. But also, this test tracked how tasks were dispatched across the available worker threads when no specific thread target was specified. The data, Table 7, showed a relatively even distribution of tasks across the three worker threads. Thread zero processed a bit over the one hundred expected tasks but this can be easily justified. In cases where the first task finishes early and the system takes more time to deploy the other two tasks, thread zero can become available in time to catch another one. This also makes sense to justify thread three being the one missing the two tasks expected since the load balancer goes through the threads to see the first available and thread three is the last on the list. Regarding the execution times, presented in Table 8, this averaged a bit over one seconds and a half which is a little over the expected. This is probably justified by the load balancer factor which was highly added in this scenario. Job dispatch latency remained extremely low, confirming the efficiency of the system's internal queue and scheduling mechanisms, even under increased load. Again, when cross-referenced with Table 3, the job dispatch values we obtained are comparable to those of the standard Rust *threadpool* implementation, suggesting that *rt_lib* achieves a similar level of efficiency in distributing tasks dynamically.

Table 7 - Code 21 Task Distribution by Thread

Thread	Parent Execution Duration
0	102
1	100
2	98

Table 8 - Code 21 Performance Statistics

Metric	Total Execution Duration	Individual Job Launch Time
Count	100	300
Minimum	1.529734 s	0,000 ms
Maximum	3.249423 s	0,011 ms
Average	1.708475 s	0,001 ms
Median	1.611986 s	0,000 ms
Deviation	0.267858 s	0,001 ms

The final performance scenario, Code 22, involved initializing the thread pool from within an already running thread. This is a slightly more complex use case and is relevant in applications where dynamic task management is required during runtime rather than setup. This also introduced a different way of creating a thread pool, this time using inherited values from the parent thread. As can be seen by the results on Table 9, pool creation times were very low on average and most of them remained under two hundred and fifty milliseconds. Parent thread execution times showed more significant variability, likely because in every iteration there was initialization overhead now included in their timing. In some cases, execution times reached over fifty milliseconds, which highlights the importance of pre-initialization if deterministic timing is required. When viewed alongside the external Rust benchmark Table 3, the average pool creation time of 0,657 ms stands out as especially low, reinforcing the idea that pool initialization inside an already running thread can be efficient under the right conditions.

Table 9 - Code 22 Performance Statistics

Metric	Thread Pool Creation Time	Parent Thread Execution Time
Count	100	100
Minimum	0.120 ms	0.279 ms
Maximum	17.682 ms	50.227 ms
Average	0.657 ms	2.394ms
Median	0.249 ms	0.847 ms
Deviation	1.924 ms	6.523 ms

Overall, the results demonstrate that *rt_lib* offers low-overhead thread management and efficient job dispatching, making it a viable candidate for real-time programming in Rust. Thread creation, both inside and outside a pool, are consistently fast, execution durations are predictable under most conditions, and task scheduling across worker threads is well-balanced. The use of virtual machines adds some noise to the data, but the general performance profile remains strong and in line with real-time system expectations which can be seen by the median values of every test case. When compared to the external baselines, the results further strengthen this conclusion. Thread creation times match closely with C implementations, while thread pool creation and job dispatch

times remain in the same order of magnitude as Rust's established *threadpool* crate. This suggests that the design of *rt_lib* achieves both competitive performance and reliability, validating its use in real-time contexts.

5.3 Summary

We started this segment by revisiting Chapter 3.1 list of user stories created, based on the programmer needs collected. These were addressed and a solution for each was presented, helping solidify the *rt_lib* as a library that attends to real-time application programmer needs.

Throughout this section, we demonstrated the simplicity and flexibility of using *rt_lib* to manage both isolated real-time threads and configurable thread pools. Starting with isolated threads, we explored how users can quickly spawn them with precise timing and scheduling requirements, all with minimal and readable code. We then shifted to the thread pool interface, which extends this usability to larger-scale, multi-threaded scenarios, whether through explicit configuration or inherited attributes, while preserving the same clarity and ease of use.

These examples do more than illustrate a clean API, they also lay the foundation for the performance benchmarks that follow. Because they are realistic yet minimal, they allow us to test the library in practical conditions without unnecessary complexity. From isolated execution timing to thread coordination and scheduling overhead, these setups served as test cases to evaluate how *rt_lib* performs under different usage scenarios. We analysed the timing precision, overhead, and consistency of the examples provided, offering a clearer picture of the strengths and limitations the library can have when used in real-world, time-sensitive applications.

Across all tests, the library demonstrated consistently low thread and job launch overhead, as well as stable execution times under most configurations. Thread pool creation times typically remained below a few milliseconds, with occasional outliers likely caused by the virtual environment. For instance, average pool creation times ranged from under half a millisecond to just over five, with even the most complex setups very rarely exceeding seventy. When contrasted with baseline implementations, such as native C thread creation or Rust's *threadpool* crate, the performance of *rt_lib* falls well within the same order of magnitude. While C still offers the lowest raw creation costs, the differences are modest, and *rt_lib* compensates by exposing real-time features like scheduling classes, affinity, and priorities that are absent in these baselines. These results suggest that *rt_lib* is able to initialize thread resources quickly and efficiently, a crucial property for real-time systems that may need to adapt to changing workloads. Job execution durations, particularly in scenarios where a workload was repeated multiple times, generally hovered around one to bellow seconds on average when the expected was one. These numbers reflect not just the time taken by the threads themselves, but also the synchronization and lifecycle overhead of managing execution within a pool. The standard deviation values were modest in most cases, indicating good consistency which is an important feature for applications where predictability is critical. The most notable feature is the extremely low job dispatch latency, which consistently measured in microseconds or even less across all tests. In many cases, the recorded average launch times per job were under three microseconds, with median values effectively at zero, meaning job dispatch

occurred nearly instantaneously from the perspective of the measuring thread. This highlights the internal efficiency of the scheduling and queuing mechanisms within *rt_lib*. This result is particularly relevant when compared with the *threadpool* crate, where average job dispatch latency was slightly higher. The fact that *rt_lib* consistently delivers dispatch times in the microsecond or sub-microsecond range highlights its efficiency, despite offering a richer feature set.

Thread utilization was also evaluated in one of the test cases, where jobs were scheduled without specifying the target thread. The distribution of executed tasks across three threads was nearly even, with only a small difference which is expected and easily justified by the way the load balancer works within *rt_lib*. This demonstrates that the library's load balancing logic functions effectively which is an important feature in real-time systems.

When comparing tests that use different pool initialization methods or execute within nested thread contexts, some performance trade-offs become evident. For instance, creating a thread pool inside a running thread slightly increased overall execution time, while more lightweight configurations performed more efficiently. These trade-offs are expected and do not indicate flaws, but they reinforce the importance of matching the threading model to the application's needs. From a broader perspective, execution times, even in the worst-case measurements, remained within acceptable bounds, especially considering the use of a virtual machine. Variability introduced by the environment was mostly contained, and deviations remained relatively low in comparison to the mean values. However, it is likely that testing on real-time operating systems would yield even more deterministic results.

In conclusion, the performance metrics gathered from all test scenarios paint a positive picture of *rt_lib*'s readiness for real-time contexts. The library shows strong performance in key areas such as thread pool setup time, dispatch latency, and load distribution, all while maintaining predictable execution durations. Taken together with the baseline benchmarks, these findings show that *rt_lib* not only matches established solutions in terms of raw performance but also extends them with real-time control primitives. This positions it as a competitive and practical option for developers who need predictable timing behaviour without sacrificing flexibility or safety. These properties are essential in any system that requires timely and reliable thread orchestration. Although further testing in more isolated environments would help refine these results, the evidence so far supports the idea that *rt_lib* offers a solid foundation for building real-time systems in *Rust*.

6 Conclusion

The work developed throughout this dissertation aimed to contribute to the ongoing research and development of real-time systems, a field that continues to grow in relevance, complexity, and application diversity. From robotics to automotive and aerospace industries, the demand for highly reliable systems with deterministic timing behaviour is expanding. In this context, the integration of real time constraints with modern parallel computing models represents both a necessary evolution and a considerable challenge.

The initial phase of this work addressed the limitations of traditional real-time programming approaches, particularly considering the increasing presence of multi-core processor architectures. As outlined, the focus on sequential programming began to show its limits when faced with the need to scale performance without compromising temporal predictability for real-time systems. This shift in hardware capabilities, toward low-cost, high-performance multiprocessor systems, demanded a new approach to how software is developed to fully take advantage of these new capabilities while still meeting strict real-time constraints.

To support the development, this project studied and compared two technologies: the Rust programming language versus the C programming language with the OpenMP API. Both approaches bring valuable features to the domain of real-time programming, OpenMP being a mature and widely adopted tool in high-performance computing, provides a well-established interface for parallelism, and recent efforts have extended it to better support real-time systems. On the other hand, Rust, although younger, introduces a compelling set of advantages, most notably, its strong guarantees around memory safety, concurrency, and performance. Ultimately, the dissertation proceeded with Rust as the chosen approach for the implementation and experimentation. This decision was made based on Rust's innovative memory ownership model, its ability to provide low-level control while maintaining high-level ergonomics, and its promising ecosystem for systems programming. Also, the choice to use Rust aligns with recent investments in safety critical software development, where reliability and correctness are a must.

The core contribution of the project revolved around the implementation and evaluation of a custom thread management model designed for real-time applications. This system was constructed to allow greater control over thread and thread pool creation and management, task dispatching and thread scheduling parameters, enabling fine-grained analysis of system behaviour under different configurations. Mid-way through we were able to identify a set of requirements for the library that were later addressed as complete. These were based on evaluations of the programmer needs when working with real-time applications. Through a series of structured performance tests, several metrics were gathered to evaluate the behaviour and characteristics of the system. These included thread and thread pool creation times, total execution duration, job dispatch latency, and task distribution across threads. These results, collected over hundreds of test iterations, offered a comprehensive view into the practical runtime behaviour of the system.

The statistics collected, such as average execution times, standard deviations, and median values, allowed for meaningful comparisons across different configurations and use cases. Although a formal comparative analysis was not the main objective, some patterns did emerge. For instance, thread pools created using different initialization methods showed variation in setup overhead, and test cases involving more explicit scheduling strategies tended to produce more consistent timing results. However, the differences were generally small enough to affirm the robustness of the system under varied conditions. In general, thread and pool creation times were fast and consistent, though some variance was observed, particularly in examples where more memory and CPU management was necessary, which are explained by the environment being ran on a virtual machine. Execution durations remained within acceptable bounds compared with the expected values, and the system demonstrated a good degree of balance in task distribution across threads. Additionally, job dispatch latency remained extremely low, reinforcing the efficiency of the design. The overall performance profile was stable and predictable, which although not perfect, is essential for real-time systems.

Additionally, when comparing the performance of *rt_lib* to baseline implementations in the same environment, such as thread creation in C and Rust's *threadpool* crate, our library demonstrates competitive performance while offering advanced real-time features. Thread creation and pool setup times, although slightly higher than raw C threads in some cases, remain consistently low and predictable. Job dispatch latency is extremely low matching or even outperforming the *threadpool* crate benchmarks registered. This comparison highlights that *rt_lib* provides a balance between raw efficiency and enhanced control over scheduling that are essential for real-time applications but not typically present in baseline implementations. Overall, these results reinforce the library's readiness for real-time programming contexts and validate the design choices made throughout this work.

In conclusion, after being able to develop this library and thoroughly test it, we were able to show that with proper configuration and care, real-time behaviour can be achieved even in environments like Linux with Rust based software. The Rust programming language, combined with deliberate system-level design choices, provides a viable path forward for building concurrent systems that meet the needs of modern real-time applications.

Despite the positive outcomes obtained from this implementation and testing, it is important to acknowledge the limitations encountered during development and evaluation. One of the most notable constraints comes from the environment in which the system was developed and tested, Linux. While Linux offers many powerful features and robust tooling for systems development, it is not, by default, a real-time operating system. Although it does support some real-time extensions, its scheduler is not designed for strict temporal determinism.

In this project, the system was constrained to two available real-time schedulers, *SCHED_FIFO* and *SCHED_RR*, limiting the flexibility and scope of experimentation. This is another area where future research could prove. The current implementation is limited by what the Linux kernel allows, but a scheduling model layered on top of the operating system could enable support for additional policies such as earliest deadline first, constant bandwidth server, or mixed-criticality systems.

These enhancements would require precise task monitoring and timing control but could significantly increase the applicability of the library in more constrained or safety-critical settings.

A promising path for future development would be to migrate the current approach to a proper real-time operating system (RTOS). Transitioning to an RTOS could remove several constraints experienced here, particularly regarding predictable interrupt handling and tighter control of scheduling policies. Furthermore, this would provide a cleaner environment for exploring and integrating more advanced real-time scheduling algorithms such as constant bandwidth server, which offers a more adaptive way to handle sporadic tasks while maintaining temporal isolation between tasks. The implementation of this and other real-time scheduling mechanisms could be facilitated in an RTOS with support for soft real-time guarantees.

Another possible way of improving this thesis's work would be targeting a hardware platform such as RISC-V for future iterations. The RISC-V architecture, due to its openness and modular design, offers a high degree of customization at both the hardware and software levels. This could allow a better design of task scheduling mechanisms in both the operating system and processor itself. Running the real-time Rust-based system on a RISC-V platform could also help achieve better timing guarantees, since many of the services and extra layers used in desktop systems can be left out (RISC-V, 2025).

Finally, the project would benefit from expanding the scale and variety of the test scenarios. While the performance tests executed provided valuable insight into the system's behaviour under controlled conditions, incorporating more dynamic and mixed workloads, such as interrupt-driven events, sporadic tasks with varying deadlines, or long-running dependent chains of jobs, could better simulate real-world challenges. This would also allow stress-testing of scheduling policies, job prioritization mechanisms, and the thread dispatch system under more extreme and varied operational conditions.

In summary, although the library built and tested in this work has shown strong potential, it remains a proof of concept with ample room for refinement and expansion. Its stability, performance, and internal organization provide a solid foundation for further experimentation, and the use of Rust has proven effective and practical for real-time systems development. However, unlocking its full potential will require to move from a Linux like system to real-time operating environments, and integrating richer scheduling and timing models that better match the diversity and complexity of modern real-time applications.

References

Ada (1983) ‘ANSI/MIL-STD-1815A-1983’.

Ali, H.I. and Pinho, L.M. (2011) ‘A parallel programming model for ada’, in *Proceedings of the 2011 ACM annual international conference on Special interest group on the ada programming language*. New York, NY, USA: ACM, pp. 19–26. Available at: https://doi.org/10.1145/2070337.2070350.

Amdahl, G.M. (2013) ‘Computer Architecture and Amdahl’s Law’, *Computer*, 46(12), pp. 38–46. Available at: https://doi.org/10.1109/MC.2013.418.

Ayguade, E. *et al.* (2009) ‘The Design of OpenMP Tasks’, *IEEE Transactions on Parallel and Distributed Systems*, 20(3), pp. 404–418. Available at: https://doi.org/10.1109/TPDS.2008.105.

Blandy, J., Orendorff, J. and F.S. Tindall, L. (2021) *Programming Rust*. O’Reilly Media, Inc.

Blue Goat Cyber (2025) *RTOS: A Comprehensive Guide*. Available at: https://bluegoatcyber.com/blog/rtos-a-comprehensive-guide/ (Accessed: 25 June 2025).

Brawer, S. (1989) *Introduction to Parallel Programming*.

Bugden, W. and Alahmar, A. (2022) ‘Rust: The Programming Language for Safety and Performance’.

Burns, A. and Wellings, A. (2009) *Real-Time Systems and Programming Languages: Ada, Real-Time Java and C/Real-Time POSIX*.

Cerqueira, F., Gujarati, A. and Brandenburg, B.B. (2014) *Linux’s Processor Affinity API, Refined: Shifting Real-Time Tasks towards Higher Schedulability*.

Cucu-Grosjean, L. and Goossens, J. (2009) ‘Predictability of Fixed-Job Priority Schedulers on Heterogeneous Multiprocessor Real-Time Systems’.

Defense, U.S.D. of (1983) *Reference Manual for the ADA Programming Language*. New York, NY: Springer New York. Available at: https://doi.org/10.1007/978-1-4612-5016-6.

Getreu, J. (2016) ‘Embedded System Security with Rust’, p. 26.

Göetz, B. *et al.* (2006) *Java Concurrency In Practice*. Addison-Wesley.

Hansen, P.B. (1975) ‘The programming language Concurrent Pascal’, *IEEE Transactions on Software Engineering*, SE-1(2), pp. 199–207. Available at: https://doi.org/10.1109/TSE.1975.6312840.

Hwu, W.M. (2014) ‘What is ahead for parallel computing’, *Journal of Parallel and Distributed Computing*, 74(7), pp. 2574–2581. Available at: https://doi.org/10.1016/J.JPDC.2014.02.005.

Jung, R. (2020) ‘Understanding and Evolving the Rust Programming Language’, (August).

King, I.E. (2024) *Understanding Thread Interactions*.

Kopetz, H. and Steiner, W. (2022) *Real-Time Systems*. Cham: Springer International Publishing. Available at: https://doi.org/10.1007/978-3-031-11992-7.

Lee, J. *et al.* (2022) ‘Optimal priority assignment for real-time systems: a coevolution-based approach’, *Empirical Software Engineering*, 27(6). Available at: https://doi.org/10.1007/s10664-022-10170-1.

Levy, A. *et al.* (2015) ‘Ownership is theft’, in *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. New York, NY, USA: ACM, pp. 21–26. Available at: https://doi.org/10.1145/2818302.2818306.

Lewis, B. and Berg, D.J. (1998) *Multithreaded programming with Pthreads*. Prentice-Hall, Inc.

Linux (2023) *Linux OS*. Available at: https://www.linux.org/ (Accessed: 13 February 2023).

LLVM (2023) *The LLVM Compiler Infrastructure*. Available at: http://llvm.cs.uiuc.edu/ (Accessed: 13 February 2023).

Matsakis, N.D. and Klock, F.S. (2014) ‘The rust language’, *ACM SIGAda Ada Letters*, 34(3), pp. 103–104. Available at: https://doi.org/10.1145/2692956.2663188.

Michell, S., Moore, B. and Pinho, L.M. (2013) ‘Tasklettes – A Fine Grained Parallelism for Ada on Multicores’, in, pp. 17–34. Available at: https://doi.org/10.1007/978-3-642-38601-5_2.

Moore, B.J. (2010) ‘Parallelism generics for Ada 2005 and beyond’, in *Proceedings of the ACM SIGAda annual international conference on SIGAda*. New York, NY, USA: ACM, pp. 41–52. Available at: https://doi.org/10.1145/1879063.1879078.

Ngo, V.C. (2020) *Implement Real-time Periodic Task for RTOS using Timers*. Available at: https://channgo2203.github.io/rt_periodic_task (Accessed: 25 June 2025).

OpenMP (2023) *OpenMP Application Programming Interface*. Available at: https://www.openmp.org/specifications/ (Accessed: 31 January 2023).

Oracle (2019) *Multithreaded Programming Guide*. Available at: https://docs.oracle.com/cd/E53394_01/pdf/E54803.pdf (Accessed: 16 September 2024).

Oracle (2024) *Java 8, Oracle Documentation*. Available at: https://docs.oracle.com/javase/8/docs/api/ (Accessed: 28 August 2024).

Pacheco, P.S. (1997) *Parallel Programming with MPI*. Morgan Kaufmann Publishers, Inc.

Pacheco, P.S. (2011) *An Introduction to Parallel Programming*, *An Introduction to Parallel Programming*. Available at: https://doi.org/10.1016/B978-0-12-804605-0.00002-6.

Pinho, A., Couto, L. and Oliveira, J. (2019) 'Towards Rust for Critical Systems', in *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. IEEE, pp. 19–24. Available at: https://doi.org/10.1109/ISSREW.2019.00036.

Pinho, A.B. de (2020) 'Exploring Rust for Embedded and Critical Systems', p. 95.

Pinho, L.M., Moore, B., Michell, S. and Tucker Taft, S. (2015) 'An Execution Model for Fine-Grained Parallelism in Ada', in *Ada-Europe International Conference on Reliable Software Technologies*, pp. 196–211. Available at: https://doi.org/10.1007/978-3-319-19584-1_13.

Pinho, L.M., Nélis, V., *et al.* (2015) 'P-SOCRATES: A parallel software framework for time-critical many-core systems', *Microprocessors and Microsystems*, 39(8), pp. 1190–1203. Available at: https://doi.org/10.1016/j.micpro.2015.06.004.

Pinho, L.M., Moore, B., Michell, S. and Taft, S.T. (2015) 'Real-Time Fine-Grained Parallelism in Ada', *ACM SIGAda Ada Letters*, 35(1), pp. 46–58. Available at: https://doi.org/10.1145/2870544.2870551.

Pinho, L.M., Moore, B. and Michell, S. (2014) 'Parallelism in Ada: Status and Prospects', in, pp. 91–106. Available at: https://doi.org/10.1007/978-3-319-08311-7_8.

Rayon (2024a) *Rayon Documentation*. Available at: https://docs.rayon.design/documentation/rayon-documentation (Accessed: 25 August 2024).

Rayon (2024b) *Rayon Github Repository*. Available at: https://github.com/rayon-rs/rayon (Accessed: 25 August 2024).

Red Hat (2025) *Red Hat Documentation*. Available at: https://docs.redhat.com/en/documentation/red_hat_enterprise_linux_for_real_time/8/html-single/optimizing_rhel_8_for_real_time_for_low_latency_operation/index#assembly_setting-cpu-affinity-on-rhel-for-real-time_optimizing-RHEL8-for-real-time-for-low-latency-operation (Accessed: 25 June 2025).

RISC-V (2025) *RISC-V Documentation*. Available at: https://riscv.org/about/ (Accessed: 21 June 2025).

Rust (2023) *Rust*. Available at: https://www.rust-lang.org/ (Accessed: 31 January 2023).

Rust (2024a) *Libc Documentation*. Available at: https://docs.rs/libc-nnsdk/latest/libc/ (Accessed: 26 August 2024).

Rust (2024b) *ThreadPool Documentation*. Available at: https://docs.rs/threadpool/latest/threadpool/ (Accessed: 29 August 2024).

Schmid, M., Fritz, F. and Mottok, J. (2019) 'Parallel Programming in Real-Time Systems', in *ARCS Workshop 2019; 32nd International Conference on Architecture of Computing Systems*. VDE.

Schmid, M., Fritz, F. and Mottok, J. (2022) 'Fine-grained parallelism framework with predictable work-stealing for real-time multiprocessor systems', *Journal of Systems Architecture*, 124, p. 102393. Available at: https://doi.org/10.1016/J.SYSARC.2022.102393.

Serrano, M.A., Royuela, S. and Quiñones, E. (2018) 'Towards an OpenMP specification for critical real-time systems', *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 11128 LNCS, pp. 143–159. Available at: https://doi.org/10.1007/978-3-319-98521-3_10.

Stankovic, J.A. (1996) 'Real-Time and Embedded Systems', *ACM Computing Surveys*, 28, p. 4.

Taft, S.T. *et al.* (2014) 'Safe parallel programming in ada with language extensions', in *Proceedings of the 2014 ACM SIGAda annual conference on High integrity language technology*. New York, NY, USA: ACM, pp. 87–96. Available at: https://doi.org/10.1145/2663171.2663181.

Tanenbaum, A.S. and Woodhull, A.S. (2006) *Operating Systems Design and Implementation Third Edition*. NJ: Prentice Hall, Inc.

Thornley, J. (1994) 'Integrating parallel dataflow programming with the Ada tasking model', in *Proceedings of the conference on TRI-Ada '94 - TRI-Ada '94*. New York, New York, USA: ACM Press, pp. 417–428. Available at: https://doi.org/10.1145/197694.197742.