



POLITECNICO
MILANO 1863

SCUOLA DI INGEGNERIA INDUSTRIALE
E DELL'INFORMAZIONE

EVA: A Rust-based Embedded Real-time Operating System

TESI DI LAUREA MAGISTRALE IN
COMPUTER SCIENCE AND ENGINEERING
INGEGNERIA INFORMATICA

Author: **Davide Mor**

Student ID: 10740237

Advisor: Prof. Federico Terraneo

Academic Year: 2025-26

Dedicated to the people around me, to whom I owe myself.

Abstract

As our world evolves, our reliance on embedded systems increases. This includes safety-critical applications, where the highest standards of coding are required. Despite this, the way we write embedded software never fundamentally changed. We believe Rust to be a promising and appealing language to address memory safety concerns. Yet, adoption of this language in the embedded world is still low and far behind other fields. We believe the culprit to be a lacking and inconsistent ecosystem. In this thesis we present EVA, our vision for a Rust-based embedded real-time operating system. Existing alternatives lack in one or more critical aspects, either by restricting language support to Rust only or by requiring specific hardware platforms to run. Our approach focuses on three main goals: portability across a wide spectrum of embedded devices; multi-language support in kernel space; support for higher level APIs, such as the Rust std library and POSIX. These last two points are critical in order to provide a viable platform for incrementally transitioning existing software to Rust. Our operating system has been benchmarked by measuring cycle counts of critical operations on an STM32 microcontroller. Different testing configurations highlight how Rust runtime checks have little impact on performance. Finally, these results are compared to a mature operating system, Miosix, showing the feasibility of Rust as an alternative to existing languages for embedded programming.

Keywords: Operating Systems, Rust, Embedded Systems, Real-time

Sommario

Con l'evolversi del nostro mondo, la nostra dipendenza dai sistemi embedded sta aumentando. Questo include applicazioni safety-critical, dove i più elevati standard di programmazione sono richiesti. Nonostante ciò, il modo in cui scriviamo software embedded non è mai fundamentalmente cambiato. Crediamo che Rust sia un linguaggio di programmazione promettente per affrontare il problema della memory safety. Tuttavia, l'adozione di questo linguaggio nel mondo embedded è ancora bassa e significativamente indietro rispetto ad altri campi. Riteniamo che la causa sia un ecosistema carente e inconsistente. In questa tesi presentiamo EVA, la nostra visione per un sistema operativo embedded real-time. Le alternative esistenti presentano carenze in uno o più aspetti critici, limitando il supporto a solo Rust o richiedendo piattaforme hardware specifiche. Il nostro approccio si basa su tre obiettivi fondamentali: portabilità su un'ampia gamma di dispositivi embedded; supporto multi-linguaggio nel kernel space; supporto per API di livello superiore, come la libreria std di Rust e POSIX. Questi ultimi due punti sono fondamentali per fornire una piattaforma valida per la transizione incrementale del software esistente a Rust. Il nostro sistema operativo è stato sottoposto a benchmark misurando il numero di cicli di operazioni critiche su un microcontrollore STM32. Diverse configurazioni di test evidenziano come i controlli a runtime di Rust abbiano un impatto minimo sulle prestazioni. Infine, questi risultati vengono confrontati con un sistema operativo maturo, Miosix, dimostrando la fattibilità dell'utilizzo di Rust come alternativa ai linguaggi esistenti per la programmazione embedded.

Parole chiave: Sistemi Operativi, Rust, Sistemi Embedded, Real-time

Contents

Abstract	i
Sommario	iii
Contents	v
1 Introduction	2
1.1. Rust and Embedded Systems	2
1.2. Contributions	2
1.3. Thesis Structure	3
2 Background	4
2.1. Embedded Systems	4
2.2. Real-time Systems	5
2.3. Operating System and Kernel	5
2.4. The Kernel Architecture Spectrum	6
2.5. Rust	7
2.6. C and C++ as System Level Languages	8
2.7. State of the Art	11
3 Designing in Rust	14
3.1. Safe and Unsafe	14
3.2. Invariant Domains	16
3.3. API Design and the Type System	17
4 Design	22
4.1. Design Goals and Objectives	22
4.2. High Level Design	22
4.2.1. Language Library Layer	23
4.2.2. Kernel Layer	24
4.2.3. BSP Layer	25
4.3. Kernel Building and Linking	25

4.3.1. How Cargo Works	25
4.3.2. Kernel Interface and Linking	26
4.3.3. Kernel Building	27
4.4. Threads	28
4.5. Scheduler	30
4.5.1. Layer 0: Architecture Level Atomic Operations	31
4.5.2. Layer 1: Scheduler Pause	32
4.5.3. Layer 2: Thread Suspend and Resume	33
4.5.4. Layer 3: Mutex and Condvar	35
4.5.5. Avoiding IRQ Disable in the Kernel	35
4.5.6. Idle Thread	37
4.6. Other APIs	37
4.7. Bootstrap Procedure	38
5 Implementation	40
5.1. Data Structures	40
5.1.1. Linked List	40
5.1.2. Sorted Linked List	44
5.1.3. Lock-free Stack	45
5.1.4. Slot Map	46
5.1.5. Bit Set	47
5.2. Scheduler Pause Implementation	48
5.3. Kernel Allocator	50
5.4. Scheduling	51
5.5. Priority and Timed Wake Lists	52
5.6. Timing Subsystem	56
5.7. Mutex and Condition Variables	57
5.8. Thread Local Storage	59
5.9. Board Support Package	60
5.9.1. Skyward Compute Unit BSP	60
5.9.2. Linux BSP	62
5.10. Porting the Rust Std Library	63
5.11. C as an Application Program Language	64
6 Experimental Results	66
6.1. Test Configurations	66
6.2. Benchmark Methodology	67

 Contents	vii
6.3. Results	68
7 Future Works and Conclusions	70
7.1. Future works	70
7.2. Conclusions	71
Bibliography	74
List of Acronyms	78
List of Listings	80
List of Figures	82
List of Tables	84
Ringraziamenti	86

1 | Introduction

1.1. Rust and Embedded Systems

As our world evolves, our reliance on embedded systems increases. This includes safety-critical applications, where the highest standards of coding are required. Despite this, the way we write embedded software has not changed fundamentally. As of 2023, 70% of embedded devices still rely on C and C++ as their main language of choice [1]. These languages are by design memory unsafe, moving the burden of validating memory accesses to the programmer. This has proven to be a great source of bugs and issues. Rust aims at solving this by providing a memory safe programming language that relies almost entirely on static compile-time checks.

Despite wide adoption in several industries, such as cloud infrastructure and high performance computing, Rust still has not gained much traction in the embedded world. We believe the culprit to be an inconsistent ecosystem, still dominated by C and C++. Many widely adopted and mature embedded operating systems have poor Rust support. While Rust-based embedded operating systems do exist [2], they suffer from the opposite problem of being too Rust centric and lacking support for existing infrastructure. This last issue needlessly heightens Rust commitment cost, reducing Rust adoption.

1.2. Contributions

This thesis presents EVA, our vision for a Rust-based embedded operating system. As explained in the previous section we believe the current Rust embedded ecosystem to be lacking. To solve this issue we designed an operating system specifically to fill the current ecosystem gap. As such EVA is designed around four primary design goals. The first is to be written and designed in Rust from the outset, as we believe this language to be an excellent candidate for low-level and safety critical embedded programming. The second is to provide multi-language support in kernel space. This is a feature that is practically unseen in Rust operating systems, as they mostly focus on Rust as the sole programming language. We believe that by removing this limitation EVA could become a valid platform to transition to Rust. The third is to provide higher level and more commonly used Application Program Interfaces (APIs), such as the Rust std library

and POSIX. This allows developers to work at a higher level, opening access to a wider already existing ecosystem of libraries. Finally, the last goal, is to provide portability across a wide spectrum of embedded systems, arguably a critical requirement for any embedded-centric operating system.

Alongside these fundamental goals we also experimented with some alternative designs in our system. EVA distinctly lacks any kernel-level primitive for disabling interrupts, leaving the task to lower-level drivers and hardware specific code. We also implemented a way to run the entire operating system inside a single Linux process, opening access to a multitude of existing tooling.

Finally, we benchmark EVA to provide some experimental results showing the usability of our system. We first highlight how different build configurations affect performance. These configurations also include unchecked versions of the code, with most runtime checks stripped away, to show how Rust runtime checks have a negligible impact on performance. Then we benchmark against Miosix, to provide some baseline to our results and how we compare to a mature alternative.

1.3. Thesis Structure

In Chapter 2 we provide the necessary background elements for this thesis. Giving the reader definitions for concepts and terms specifically for this work. We cover embedded systems, real-time systems, operating systems, and kernel designs. Finally, we give an overview of the current state of the art, confronting existing solutions and highlighting the ecosystem gap. In Chapter 3 we tackle the issue of working with Rust, explaining some good practices and design patterns useful for our purpose. In Chapter 4 we start to dive into EVA by first looking at its design. This chapter gives a high-level overview of how EVA is structured, what our goals are, and how the design achieves them. In Chapter 5, instead, we analyze the actual realization of those concepts. We examine data structures and give in-depth explanations of the inner workings of EVA. In Chapter 6 we explain how we conducted our final validation benchmarks, give experimental results, and analyze them. Finally, Chapter 7, we outline potential future works and draw our final conclusions.

2 | Background

2.1. Embedded Systems

An embedded system is a highly specialized computer system with a dedicated function within a larger system. Their usage in the real world ranges from home appliances to industrial robotics and aircrafts. Due to their nature, these devices need a very different approach to design and programming. They often require specialized peripherals and hardware not seen in traditional computation devices. To optimize manufacturing costs, the processing power of these devices is frequently severely limited, with processing speeds in the order of tens of megahertz and only a few megabytes of available RAM and flash. In some environments, such as portable or remote installations, energy efficiency is an important consideration, which requires sophisticated hardware management to reduce consumption to a minimum. For these reasons, embedded systems demand specialized software. The field of embedded software development is almost a completely separate branch of programming. The limited resources make considering even the most minute details critical.

Although the world of embedded devices is broad, for the sake of this work, we will focus on one particular class of embedded devices. When referring to embedded devices from now on, we will refer to low- to mid-end embedded devices. Table 1 shows rough attributes for what we consider devices pertaining to this category. The most relevant one is the lack of an Memory Management Unit (MMU), which severely impacts the design of the operating system in particular. Traditional techniques for memory protection and user space to kernel space separation are no longer possible and need different approaches. Since the kernel is written in Rust, additional restrictions are imposed on the target, which must be supported by LLVM.

Attribute	Value
RAM	100kB to 100MB
Flash	100kB to 100MB
Clock speed	1MHz to 1GHz
MMU	No

Table 1: Attributes for low- to mid-end embedded devices.

2.2. Real-time Systems

The term real-time system refers to hardware and software systems with strict time requirements. These are often expressed in terms of deadlines or latencies in the system. Depending on the failure mode of the system when such constraints are not satisfied, we have two major categories of real-time systems: soft real-time and hard real-time. Soft real-time systems are less strict than hard real-time systems, as failure to comply with constraints leads only to degraded performance. Hard real-time systems, on the other hand, are usually also considered safety-critical or mission-critical, and failure of these components leads to catastrophic results, either partial or total mission failure or danger to humans and their environment.

Developing for real-time applications requires predictability and consistency over raw performance. Since these time requirements need to be guaranteed always, the most relevant metric is the Worst-Case Execution Time (WCET), which tells us the maximum length of time that a piece of code could take to execute under all possible conditions. Due to this, hardware platforms specifically designed for real-time (such as Cortex-M and Cortex-R) lack several functionalities that would improve the average execution speed but do not affect worst-case guarantees. These functionalities, such as branch prediction and out-of-order scheduling, only negatively impact the system as they increase the execution time variance, leading to a less predictable execution time.

Real-time Operating Systems (RTOSs) are operating systems specifically designed to meet these real-time constraints. The scheduler becomes the primary element of interest, which needs to provide strong guarantees about the scheduling order and respect of priorities. Another important detail is the system responsiveness, which is how quickly a system can react to events such as a higher priority thread being woken up. There is significant overlap between real-time and embedded systems, consequently, real-time operating systems often need to operate under both real-time and embedded constraints.

2.3. Operating System and Kernel

An Operating System (OS) is defined as the software interfacing with hardware components and resources, abstracting low-level details for higher-level applications and code. An operating system is not a singular entity and instead is a collection of different modules. This definition is purposefully very broad, as there exist several kinds, and use cases differ greatly in structure and architecture. For this work we will focus on operating systems designed for use in embedded and real-time applications (Section 2.1. and Section 2.2.).

The kernel is the foundation of the operating system, which is responsible for providing the core set of abstractions for the rest of the system. The responsibilities of the kernel depend greatly on the particular kernel architecture chosen, but broadly speaking these are:

- **System bootstrapping**, performing initial hardware setup and getting the system ready for nominal operation.
- **Scheduling**, temporally multiplexing different execution flows (explained in more detail in Section 4.5.).
- **Time management**, allowing access to time and managing timed events.
- **Memory management**, providing support for interfacing with the system MMU (if it has one) and managing allocation and deallocation of memory regions.
- **Filesystem support**, interfacing applications with filesystem drivers and storage device drivers.
- **Device management**, abstracting how resources such as hardware devices are seen, acquired, and otherwise interacted with.
- **Input/output device management**, providing a convenient way of interfacing with the outside world, both for applications and the kernel itself.
- **Inter-Process Communication (IPC)**, allowing for safe communication between isolated applications or processes.

2.4. The Kernel Architecture Spectrum

As described in Section 2.3., kernel architecture is a spectrum, and several approaches have been studied and proposed over the years. In this section we analyze and detail the most common or relevant ones.

Monolithic: Monolithic kernels are among the most common for desktop and server applications. The term monolithic comes from having the majority of OS services run in kernel space. This simplifies hardware access, as drivers have direct access to it. But we make the trusted computing base larger than necessary, possibly introducing failure points. A buggy device driver can easily render the entire system vulnerable or unstable.

Microkernel: The microkernel design tries to address the main drawbacks of a monolithic kernel by minimizing the amount of code running in kernel space. All OS services are pushed inside the user space and are required to communicate via IPC. This provides a smaller trusted computing base but makes interaction between OS components slower.

Hybrid kernel/Fluid kernel: Hybrid kernels try to combine the advantages of both monolithic and microkernel designs. This results in a microkernel with some critical OS services running at a higher privilege level, allowing for faster execution of critical components. Fluid kernels [3] take this a step further, by providing an identical API to

both kernel space and user space, it is possible to easily switch components in and out of kernel space on a per-project basis. This is particularly important in the embedded world due to the high variance of possible hardware configurations. Running OS services in the kernel space is faster and more memory efficient but less robust, as such, it lends itself perfectly to low-end devices, while mid-end devices can afford to pay the computational price in favor of higher reliability.

Frame kernel: All the previous architectures provide safety through hardware mechanisms. By running the application in user space we remove the issue of memory corruption in the kernel space. Frame kernels [4], [5] take a different approach and instead use memory safe languages to provide safety. Depending on the language, this guarantees higher safety in the kernel at no extra runtime cost. Frame kernels in particular focus on splitting the kernel into two parts: the OS framework, which provides only essential hardware abstractions and is written in a memory unsafe language, and the rest of the kernel, written only in a memory safe language. This abstraction allows frame kernels to achieve a tiny trusted computing base even compared with microkernel designs, as through clever abstraction and API design, the framework exposes only the bare minimum.

Unikernel: Finally, we highlight the unikernel [6], which simply removes the separation between user space and kernel space completely. Applications are directly linked to the kernel and run in kernel space alongside it. This design is extremely common in environments where the hardware itself does not provide much to achieve clear separation between the two spaces, such as MMU-less embedded systems.

2.5. Rust

Rust is a relatively young programming language, with a focus on compile-time memory safety. Memory safety is the condition of being protected against several memory related bugs and errors, such as buffer overflows and dangling pointers. Memory safe languages have existed for many years, but most of them solve the issue by relying heavily on runtime checks. This makes these languages unsuitable for scenarios where the highest level of performance and hardware control is required. Rust solves this issue in a novel way by introducing a powerful type system and compile-time checker. Both of which have been formally proven to guarantee memory safety, under the assumption of correct unsafe soundness and compiler implementation [7], [8]. These checks are very conservative, as they restrict the set of valid programs (as in accepted by the compiler as error free) to a subset of all possible memory safe programs. This is in stark contrast with other languages, where the set of valid programs is a superset of memory safe programs. Most other memory safe languages rely on runtime checks, meaning that memory unsafe

programs are still accepted by the compiler and instead fail in a controlled way during runtime.

Since not all possible memory safe programs are representable, Rust augments its semantics with the notion of `unsafe`. This keyword allows us to bypass many of the compiler checks and allows more programmer freedom, at the cost of nullifying any memory safety guarantee. Programmers need to be careful and must uphold all the invariants assumed by the Rust machine model manually. By embedding a memory unsafe language inside a memory safe one, we retain most of the expressivity of any other fully memory unsafe language, but we restrict the scope of memory safety related bugs to very narrow sections of code.

2.6. C and C++ as System Level Languages

C and C++ are still the current de facto standard for system-level programming. According to [embedded.com](#) [1], as of 2023, they are used in 70% of projects targeting the embedded world. Despite this, we believe that these languages are unsuitable for the task, especially in safety-critical and real-time critical environments. This is due to programmer “cognitive load”, an important and valuable resource in these environments. C and C++ especially tend to increase cognitive load artificially by simultaneously introducing syntactic noise, through an often overly verbose syntax, and hiding critical details, making immediately understanding code behavior harder than it should be.

For example, Listing 1 shows a simple C program, yet this already contains a possible mistake. Due to C automatic (and silent) casting feature, `foo` could get truncated, but this depends on `bar` definition (Listing 2). This adds to the cognitive load of the programmer in the sense that we cannot rely on the compiler. Instead, we must check this manually, and to do so, we also need to keep in our mental model segments of the code that are not immediately visible. As an example, in Listing 1, `bar` could be defined in another part of the codebase. Yet another effect of this is an increase in code coupling. Changes to `bar` affect the truncation of `foo`, but the correlation of the two is not immediately obvious. We must say that modern compilers do add options to emit warnings and even errors for these specific issues, but this is not part of the standard and not mandated in any way.

```
1 #include <stdint.h>
2
3 uint32_t foo = UINT32_MAX;
4 bar(foo);
```

C

Listing 1: Example of subtle integer truncation in C.

```
1 #include <stdint.h>
2
3 void bar(uint16_t foo) {
4     // foo gets mistakenly truncated to only 16 bits
5 }
6
7 uint32_t foo = UINT32_MAX;
8 bar(foo);
```

Listing 2: Example of subtle integer truncation in C. (cont.d)

Issues like the one shown previously are even more pervasive in C++, where more “modern” features such as function overloading, operator override, and copy/move semantics add to the overall code complexity in subtle yet critical ways. For example, Listing 3 shows another simple C++ program. Even in this case we cannot immediately determine what the exact code behavior is due to several hidden factors:

- There could be more than one `bar` defined via overloading, which could be scattered in different parts of the codebase.
- Depending on how `Foo` is defined and used, its contents could be copied or moved inside `bar`, which might lead to subtle and hard-to-spot allocations in critical code portions.
- `foo` might be cast to another type before being passed to `bar` as an argument, leading to more confusion about how `bar` is called.
- `bar` could fail by throwing an exception, and it is not clear from this code alone if we need to protect this code section. This is especially problematic since the exception throwing code could be buried deep within the call stack. Furthermore, since C++ lacks what Java calls “checked exceptions” there is no way of telling from the function definition either.

All of these problems arise from language constructs and are not inherent to the program itself.

```
1 #include <stdint.h>
2
3 Foo foo;
4 bar(foo);
```

Listing 3: Example of ambiguity in C++.

One of the primary goals of Rust is to provide a platform that tries to solve these issues. Looking back at Listing 3, here is how Rust solves the issues highlighted before:

- In Rust there is no concept of overloading. Different functions in the same namespace must have different names, which in turn leads to a clearer identification of the relationship between definition and usage.
- Copy and move operations are semantically very different in Rust, as can be seen in Listing 4 and Listing 5. Move operations are guaranteed to never allocate, as the language does not provide any way to hook into these operations.
- Implicit casts do not exist in Rust. All casts are explicit, even when the cast should not affect the value (for example, by widening the integer size).
- Rust makes a clear distinction between recoverable and unrecoverable errors. Recoverable errors are handled through the return value of a function, and the compiler properly warns the developer if we ignore this value. Listing 6 shows what converting a recoverable error into an unrecoverable one looks like. As we can see, it is easily identifiable from the call site, and it doesn't require any knowledge about the definition of `bar`.

```
1 let foo = Foo::default();  
2 // Object is moved, no allocation is performed  
3 bar(foo);
```

Rust

Listing 4: Example of move operation in Rust.

```
1 let foo = Foo::default();  
2 // Object is cloned, this might allocate  
3 bar(foo.clone());
```

Rust

Listing 5: Example of copy operation in Rust.

```
1 let foo = Foo::default();  
2 // Compiler warns us if we ignore the return value of this function  
3 bar(foo).unwrap();
```

Rust

Listing 6: Example of error management in Rust.

Rust syntax is designed to always keep a small “memory footprint” in the programmer’s mind. Every line is as self-contained as possible, and every important aspect is explicit. This concept is known as “fearless programming”. By lightening the cognitive load due to language semantics, we can focus more on the problem at hand.

The effect of this is not only theoretical but very much practical. Google’s latest report indicates that Rust has a 1000x reduction in memory safety vulnerability density compared to C and C++ [9]. Not only that, thanks to the overall ease of usability, Rust has 4x lower rollback rate and 25% less time in code review. And Google is not the only one advocating for Rust, as other big entities are also pushing for wide Rust

adoption, like Microsoft [10] and the White House Office of the National Cyber Director (ONCD) [11].

2.7. State of the Art

The field of operating system development is not new, and as such there is already a significant amount of work. We collected and analyzed only relevant operating systems, focusing on language used for development, architecture, and overall language support.

Miosix: Miosix [12], [13] is an operating system developed at Politecnico of Milano since 2008. It stands out as one of the few embedded operating systems written entirely in C++. Historically it was based on a unikernel design with optional support for user space programs, making it more similar to a monolithic kernel. The recent unreleased 3.0 version instead switched to a newer fluid kernel design, explained in Section 2.4.. Rust support for Miosix is practically nonexistent, and due to the prevalence of C++-based interfaces in the kernel, it is practically impossible to include Rust at that level.

Zephyr: Zephyr [14] is a relatively young embedded operating system that has recently received much attention for its high flexibility and vast hardware support. Despite focusing on embedded targets, it advertises itself as a monolithic kernel. Although it provides support for both C++ and Rust, it is forbidden inside the kernel, driver, and system initialization code. Support for the Rust std library is also lacking.

RTEMS: RTEMS (Real-Time Executive For Multiprocessor Systems) [15] is an embedded unikernel. It supports several architectures, and it is of particular interest due to its use in space applications, such as at ESA (European Space Agency). This operating system features great support for C, C++, and Rust. All standard libraries are also supported, including the Rust std library.

QNX Neutrino: QNX Neutrino is a commercial, closed-source operating system developed by BlackBerry. It is included here to show how proprietary solutions do exist. It has good support for our considered languages and even their standard libraries.

FreeRTOS: FreeRTOS [16], [17] is designed to be a very minimal embedded operating system, as its primary component is a scheduler. It follows a unikernel design, but it has optional support for an Memory Protection Unit (MPU). It has been adopted by some hardware vendors as their primary operating system, such as Espressif in their ESP line of devices. Officially, just C is supported. Both C++ and Rust can work with FreeRTOS, but support is mostly community-driven and unofficial. Support for the C++ std library is only partial, while for the Rust std library it is non-existent.

Hermit: Hermit [18], [19] is a unikernel designed for the cloud and high-performance applications. It is noteworthy despite not targeting the embedded world, as it is very close to our vision of an embedded operating system. It supports both the Rust std library and multiple languages in kernel space. Both the libc and C++ std library are supported. Interestingly, this project was originally written in C and was named HermitCore, and it only recently switched to Rust.

Xous: Xous [20] is an embedded microkernel-based operating system designed with security as its primary design goal. It is written in Rust and designed around it, supporting the Rust std library in applications. The downside is that C and C++ are both completely unsupported, and even hardware support is severely limited, only being available for RISC-V 32.

Drone OS: Drone OS [21] is an unmaintained Rust-based unikernel. It supports just bare-metal Rust, with no std library support. It is notable as it was one of the first embedded operating systems written in Rust. Furthermore, it features some interesting design choices. For example, it completely decouples hardware implementation from the kernel itself by manually exporting and importing symbols. This technique is also used in EVA and explained in more detail in Section 4.3.2..

Tock: Tock [22], [23] is an embedded microkernel-based operating system. Its primary goal is application isolation and security. The kernel is written entirely in Rust with no support for the std library. It works on a system called capsules, which contain kernel drivers. These drivers are prohibited from using unsafe, in a fashion similar to a frame kernel. C and C++ are supported only for application development.

Table 2 shows a summary of the information just reported. This table also includes operating system not previously mentioned: ThreadX, NuttX, RIOT. These were not analyzed more in-depth due to their similarities to already covered operating systems.

From this data we can see how full Rust support is relatively uncommon, as only three of the analyzed operating systems provide full Rust std library compatibility. We can also see how Rust is used commonly across several operating systems, which often lack proper support for languages besides Rust itself. These systems either provide no support at all or limit themselves to supporting C and C++ only in user space (Tock).

This leads us to our conclusion: there is a clear gap in the current ecosystem. It is missing an embedded centric operating system, written in a memory-safe language, driven by the recent industry push. But one that is also capable of supporting existing languages and infrastructure, making it easily integrated into existing codebases.

Name	Written in	C	libc	C++	C++ std	Rust	Rust std	Embedded	Open source	Maintained
Miosix [12], [13]	C++	Yes	Yes	Yes	Yes	No	No	Yes	Yes	Yes
Zephyr [14]	C	Yes	No	Yes	Yes	Yes	No	Yes	Yes	Yes
RIOT [24]	C	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
RTEMS [15]	C	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
NuttX [25]	C	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
QNX Neutrino	C	Yes	Yes	Yes	Yes	Yes	Yes	No	No	Yes
FreeRTOS [16], [17]	C	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
ThreadX [26]	C	Yes	No	Yes	No	Yes	No	Yes	Yes	Yes
Hermit [19]	Rust	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
Xous [20]	Rust	No	No	No	No	Yes	Yes	Partial	Yes	Yes
Drone OS [21]	Rust	No	No	No	No	Yes	No	Yes	Yes	No
Tock [22], [23]	Rust	Partial	Partial	Partial	Partial	Yes	No	Yes	Yes	Yes

Table 2: Comparison of existing OSs.

3 | Designing in Rust

Rust approach to programming is very novel and as such requires a different mindset. Traditional patterns cannot be translated easily into Rust, and instead new solutions need to be developed. This chapter briefly describes some patterns and consideration relevant to the rest of the thesis.

3.1. Safe and Unsafe

The distinction between safe and unsafe code is the key abstraction that allows Rust to ensure that the program is free of any undefined behavior. Ideally, Rust code should be comprised of mostly safe code, limiting unsafe code to a few restricted key areas. Functions can be marked as unsafe, requiring unsafe code to be invoked. These functions work by contracts communicated via plain human language. The documentation should clearly state what preconditions are necessary for the unsafe function call to be valid. For example, the function `ptr::read`, which is used to dereference a pointer, states that the pointer must be valid for reads, properly aligned, and must point to a properly initialized value [27]. It is good practice to state the reason and how these contracts are being guaranteed at the call site via a safety note.

Unsafe code must also not “leak” into safe code, as the effects of improper unsafe code should be immediate and not rely on the correctness of following safe code. For example, Listing 7 shows a piece of code from EVA that does not follow this principle. This utility is used to erase the lifetime of a reference, bypassing compile-time lifetime checks. The need for this utility arises from the fact that Rust compile-time checks can not always properly model lifetimes, in which case we can either resort to runtime checks (for example, by using reference counting) or manually asserting the correctness of the code (the purpose of this utility). Listing 8 shows how one might use this utility. The unsafe code used to create the `UncheckedRef` makes assumptions about later safe code, the assumption being that “all instances of `UncheckedRef` are destroyed at the proper time”, which is clearly a violation of the aforementioned principle. A better API would be Listing 9, where destruction of the node is itself an unsafe operation, as this is the operation that could cause a memory error. We can clearly see the difference if we compare the two safety note wordings. In the first example, we are making an assertion

about something that will happen in the future, inside safe code. While in the second example we are using a present form, we are asserting that the condition is valid now, inside an unsafe operation. The only reason why the current implementation of this utility inside of EVA uses the first API design is that the second one is much harder to model inside the Rust type system and would require a mix of compile-time and runtime checks to work.

```

1 use core::ops::Deref;
2 use core::ptr::NonNull;
3
4 /// Erase the lifetime of a reference, allowing it to be moved around
  more easily.
5 #[derive(Clone, Copy)]
6 pub struct UncheckedRef<T> {
7     ptr: NonNull<T>,
8 }
9
10 impl<T> UncheckedRef<T> {
11     /// Construct a new `UncheckedRef<T>` by erasing the lifetime of an
    existing reference.
12     /// # Safety
13     /// You can view this object as an immutable reference, so you have to
    upheld all of the same requirements manually.
14     /// - The object pointed by `value` must not be destroyed while this
    object is alive.
15     /// - You cannot construct any mutable reference to the original object
    pointed by `value` (aka no mutable aliasing).
16     pub unsafe fn erase_lifetime<'a>(value: &'a T) -> Self {
17         Self { ptr: NonNull::from(value) }
18     }
19 }
20
21 impl<T> Deref for UncheckedRef<T> {
22     type Target = T;
23
24     fn deref(&self) -> &Self::Target {
25         unsafe { self.ptr.as_ref() }
26     }
27 }

```

Listing 7: Extract from UncheckedRef implementation.

```
1 let node = PriorityWakeup {  
2     // ...  
3 };  
4  
5 let erased = unsafe {  
6     // SAFETY: We ensure that all instances of erased are destroyed by the  
7     // end of the scope  
8     UncheckedRef::erase_lifetime(&node)  
9 };  
10 // erased can now be shared between threads easily  
11 shared_structure.insert(erased);  
12 // We must ensure that all instances of erased are destroyed before the  
13 // end of the scope!  
14 shared_structure.remove(erased);
```

Rust

Listing 8: Example of UncheckedRef usage.

```
1 let node = PriorityWakeup {  
2     // ...  
3 };  
4  
5 let erased = UncheckedRef::erase_lifetime(&node);  
6 // erased can now be shared between threads easily  
7 shared_structure.insert(erased);  
8 shared_structure.remove(erased);  
9  
10 // The user is enforced to manually signal node destruction  
11 // This unsafe asserts that all instances of erased are destroyed  
12 unsafe {  
13     // SAFETY: No instances of UncheckedRef exist anymore  
14     node.destroy()  
15 }
```

Rust

Listing 9: Example of UncheckedRef usage with improved API.

3.2. Invariant Domains

In this section we explain the concept of what we call “**invariant domains**”, and to do so we will show a real-world scenario involving the evolution of the EVA scheduler code.

One of the many principles of programming is code decoupling. When a codebase grows, it is important to keep concerns separated into modules to allow for easier maintainability. This becomes critical in the case of unsafe code. The original EVA scheduler suffered from high code coupling, but not in the traditional sense. Instead, large amounts of unsafe code were relying on assumptions made about code behavior located in different modules. In particular, the scheduling code was split between the thread module and the core scheduler module. Threads were inserted inside scheduling queues from both modules, and scheduling code makes assumptions about the presence of a thread in a particular scheduling queue. This makes reasoning about unsafe code, reserved for the most critical parts of the kernel, a daunting task.

The informal explanation of this issue is that of invariant domains. An invariant domain is a region of code that relies on a particular invariant to be true. For example, “a running thread is always inside a scheduling queue” is an invariant, and an invariant domain is the region of code where this invariant must be upheld. This region is then limited by clever API design and encapsulation. In the previous example, code outside the scheduler does not interact directly with the scheduling queues and instead uses APIs to wake and suspend threads, which guarantees internally that the invariant is always upheld. The programmer must be aware of these invariant domains and must be careful about their size.

Coming back to the case of the EVA scheduler, both the thread module and scheduler module were responsible for managing the scheduling queue. Invariants about the scheduling queues were now spanning both modules, making reasoning about code harder. The solution was to move part of the code from the thread module back inside the scheduler module, keeping the invariant domain contained inside the module boundary.

3.3. API Design and the Type System

Compared to C and C++, Rust offers a powerful and strict type system, which allows for much more expressivity when it comes to ensuring safety at compile time. For this reason, the area of API design in Rust is a fascinating and very complicated topic in itself. For simplicity, we highlight and dissect two simple examples of how one might use the Rust type system to enforce a particular invariant at compile time.

We start with a simpler pattern, the “**newtype pattern**”. This pattern is not strictly connected to Rust and can be implemented in other languages, but Rust actively encourages and facilitates the usage of this pattern. Its implementation is simple: we wrap a type inside a new one (hence the name of the pattern), which restricts in some way the allowed possible values. `NonNull<T>` is one such example, which wraps a simple pointer and asserts that the contained pointer can never be null. This pattern allows

for better communication of intent, as both code and the compiler can make direct assumptions about the allowed values in a particular variable. This is also leveraged in the compiler to make optimizations. One such optimization is the “null pointer optimization”, which allows the compiler to retain a smaller representation in the case of non-nullable pointers inside particular containers.

The second example is more complicated but better highlights the nuances and complexity of the Rust type system. The problem at hand is simple: we want to enforce access to a particular resource only in a controlled section of code. Inside of EVA, this pattern is used to guarantee that access to data synchronized via a scheduler pause is performed only while the scheduler is paused, so between a scheduler pause and unpause. We call this pattern the “**token pattern**”. Listing 10 shows a simplified implementation. We can note several interesting features:

- `PauseToken` contains a single lifetime. This is used for branding, a technique also shown in [28]. To properly implement type branding, we introduce a `PhantomData<&'a>`, which makes our type covariant over the branding lifetime. Subtyping and typing relationships are a very complicated topic in Rust, and for the sake of simplicity, we briefly explain them. These subtyping relationships dictate if a lifetime can shrink or dilate depending on its usage. We use covariance over the lifetime instead of invariance because our branding lifetime represents the lifetime in which the scheduler is paused. Covariance allows a longer lifetime to be converted to a subsection of itself, while invariance does not. We can see how this is correct for our use case since if the scheduler is paused for a specific lifetime, then it is paused for all possible subsections of this lifetime.
- `PauseToken` also contains marker types to denote that this type is not send nor sync. This prevents this type from being sent or shared between threads.
- `PauseToken` is a Zero Sized Type (ZST), which makes this type only exist during compilation, so we do not incur any runtime performance penalty, making this pattern zero-cost.
- To create a new branding and restrict its usage, we use a lambda callback. This is shown by `with_pause` in Listing 10 and in Listing 12. Due to how Rust infers the lifetime for the `with_pause` invocation, our branding lifetime is restricted to only live within the same lifetime of the lambda itself.

Finally, Listing 11 shows a “token cell”, a structure that allows access to the data contained within only if we own a `PauseToken`, and carefully associates lifetime so that the leased out borrow does not outlive our `PauseToken`. Overall, this shows how we can carefully use the type system to control the duration of a particular token object, which can then be used to signify a particular concept. In the example proposed, the existence of `PauseToken` signifies that the scheduler is paused, and this is enforced by the compiler itself.

```
1 use core::marker::PhantomData;
2
3 /// Token representing the paused state of the scheduler, existence of
4 /// this token proves that the scheduler is paused.
5 #[derive(Clone, Copy)]
6 pub struct PauseToken<'a> {
7     _marker: PhantomData<&'a ()>,
8     _not_send_sync: PhantomData<*mut ()>,
9 }
10
11 impl PauseToken<'_> {
12     /// Create a new `PauseToken`.
13     /// # Safety
14     /// Caller must guarantee that the scheduler will remain paused as long
15     /// as this token exists.
16     pub unsafe fn new() -> Self {
17         Self {
18             _marker: PhantomData,
19             _not_send_sync: PhantomData,
20         }
21     }
22 }
23
24 /// Enter a critical section using a scheduler pause.
25 pub fn with_pause<F, T>(f: F) -> T
26 where
27     F: FnOnce(PauseToken) -> T,
28 {
29     // ... make sure the scheduler is paused ...
30
31     let ret = f(unsafe {
32         // SAFETY: The scheduler is paused at this point
33         PauseToken::new()
34     });
35
36     // ... optionally unpause the scheduler ...
37     ret
38 }
```

Listing 10: Implementation of the token pattern.

```
1 use core::cell::UnsafeCell;
2
3 #[repr(transparent)]
4 pub struct PauseCell<T> {
5     inner: UnsafeCell<T>,
6 }
7
8 impl<T> PauseCell<T> {
9     /// Create a new `PauseCell`.
10    pub const fn new(value: T) -> Self {
11        Self { inner: UnsafeCell::new(value) }
12    }
13
14    pub const fn borrow<'a>(&'a self, _token: PauseToken<'a>) -> &'a T {
15        unsafe { &*self.inner.get() }
16    }
17 }
```

Rust

Listing 11: Implementation of a token cell.

```
1 fn callable_only_with_scheduler_paused(_token: PauseToken) {
2     // ...
3 }
4
5 with_pause(|token| {
6     callable_only_with_scheduler_paused(token);
7 })
```

Rust

Listing 12: Example usage of the token pattern.

4 | Design

This chapter delves into the design process of EVA, starting from our goals, analyzing every decision we made along the way, how they differ from or embrace the state of the art, and finally what paths were considered but ultimately discarded.

4.1. Design Goals and Objectives

As highlighted in the previous sections, EVA's goal is to fill the current gap in the ecosystem. As such, we identified four major points that act as the foundation for our design:

- Being written and designed in Rust from the outset.
- Multi-language support, none of the features of this operating system should be restricted to Rust alone.
- Optional support for higher level APIs such as the Rust std library and POSIX.
- Portability across a wide spectrum of embedded devices.

4.2. High Level Design

Figure 1 shows a high-level overview of how EVA is structured to achieve said design goals.

The design follows a layered approach, with four tiers: the application layer, the language library layer, the kernel layer, and the Board Support Package (BSP) layer. The application layer identifies the portion of code containing the final application. This is outside the scope of this work, but it is shown to highlight how applications can interact with EVA. The language library layer contains all the possible interaction points between the application and the kernel. We can see that applications can either interface directly with the kernel through the EVA Application Binary Interface (ABI) or by using the respective language library. Do note that greyed-out libraries reported in the figure are planned but not yet present in the current EVA release. The kernel layer contains the kernel itself, which is the primary component of EVA. Finally, the BSP layer contains code to specifically adapt EVA to a particular hardware platform,

as the kernel is completely hardware agnostic. All of these layers will be analyzed more in-depth in the following sections.

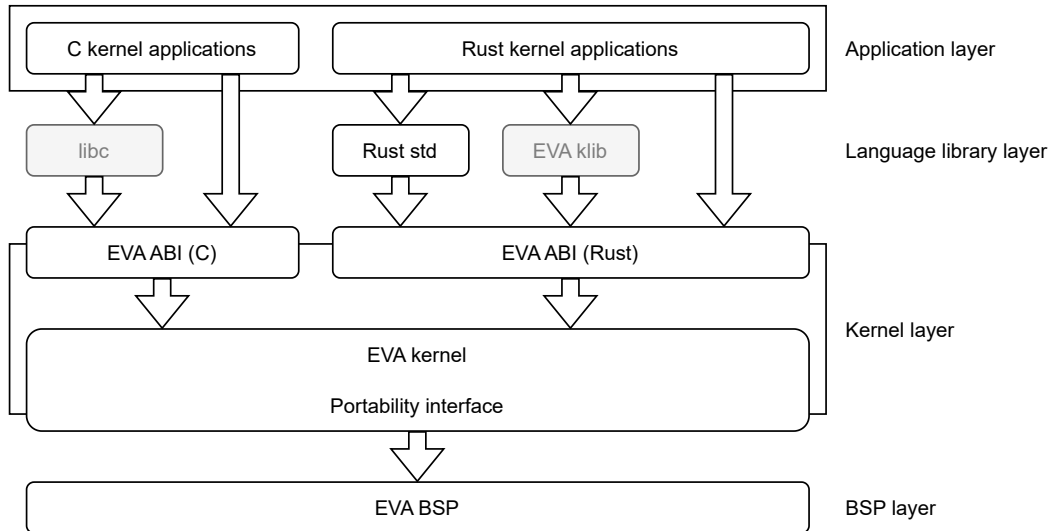


Figure 1: High level structure of EVA.

4.2.1. Language Library Layer

Supporting a rich set of applications and languages is one of the core tenets of EVA. For this reason, the application library layer and how these libraries interact with the kernel is critical. As we explained previously, the kernel exposes a formally defined API, which is known as the “EVA ABI”. This API allows applications to choose between two possible ABIs, the C ABI or the Rust ABI. Both of which provide full API coverage, functionalities are never exclusive to just one of these ABIs. The C ABI is provided for compatibility with other languages, due to its status as a de-facto lingua franca when it comes to interfacing different languages. Consequently, the Rust ABI is used when the application code is also using Rust. The Rust ABI is not stable, and it is allowed to change even in minor compiler releases, but it is specifically designed around Rust, and as such, it allows native types to be used. This makes the Rust ABI version of the API very natural and exposes many internal functionalities directly, while the C ABI requires a thin compatibility layer inside the kernel to translate between the two ABIs.

On top of the kernel API, there are language specific libraries. In general, these libraries are not strictly required, and in the embedded world, they are often left behind to allow for thinner executables and more direct control of resources. In EVA, we provide optional support for them, allowing programmers to choose on a per-project basis.

These libraries, which are either already supported or planned for future support, are the following:

- `libc` (the C standard library): More broadly, it is considered the “operating system standard library”, as it is typically used by other libraries and languages as their base.
- C++ std library: This library should rely mostly on the `libc`, but some ad-hoc support for the specific operating system is required.
- Rust std library: Completely optional and often disabled since it considerably bloats executable sizes. It is seldom supported on embedded targets, but some RTOSs do support it (see Section 2.7.).
- EVA `klib`: It is a library provided by EVA for Rust specifically. Its goal is to provide some operating system abstractions when the full Rust std library would be too resource consuming. Currently, this library is part of the kernel itself (as the kernel does not make use of the Rust std library), but in the future we intend to separate its code into a dedicated library for more usability.

Currently the only libraries with official support are the Rust std library and the EVA `klib` (as part of the kernel itself).

4.2.2. Kernel Layer

As we are targeting the embedded world, the unikernel architecture would be the most ideal choice. It is simple to implement at first but can easily be extended into a fluid kernel in the future, as shown by Miosix evolution [3]. We still want to maintain a clear separation between applications and the kernel, despite traditional unikernels not conforming to the standard definition of user space and kernel space, as they do not provide an abstraction over privilege levels. We achieve this by formally defining what is part of the kernel space applications API (the API exposed by the kernel to kernel space applications) as we would for a traditional user space to kernel space interface. This is in contrast with what other unikernels do (such as FreeRTOS and Miosix), in which the kernel space application API is simply part of the internal kernel API made publicly accessible. There are three main benefits to our approach:

- Transitioning to a fluid kernel in the future should be much easier, as we have already defined a formal interface that can be partially reused by the user space API.
- The compilation of the kernel is detached from its interface. The advantage of this is further made clear in Section 4.3.1. and Section 4.3.2..
- Accessibility of the kernel through a C ABI is better enforced. Since the API is explicit, it is much easier to make sure that the C API is complete.

For these reasons, we split the main EVA kernel library into two distinct entities, what we call the “EVA ABI” library, which defines just the kernel API, and the EVA kernel itself.

4.2.3. BSP Layer

From the need to support a vast amount of hardware platforms and configurations, embedded operating systems demand a portability layer named BSP. Many embed this component inside the operating system itself, often interacting with a complicated build and configuration system. In EVA we decided to decouple this component from the operating system as much as possible, which allows applications to easily define their own BSP, tailored to the specific platform and build system. Furthermore, in EVA, this module is responsible for providing the application with low-level hardware access, for example, by exposing an Hardware Abstraction Layer (HAL). In general, everything that cannot be provided in a mostly cross-platform way is exposed by the BSP and not the operating system directly. This reduces the coupling between the operating system and the hardware platform (which ideally should be none) and gives greater flexibility when it comes to driver and hardware support. The interaction between the kernel and the BSP is performed by a dedicated interface, called the portability interface, which allows the BSP code to be anywhere in the dependency tree, which is later explained in Section 4.3.. Since the portability interface is considered an internal part of the kernel, it is only accessible via Rust directly (it is not exported via the C ABI) as we decided that it was not necessary nor desirable to support other languages.

4.3. Kernel Building and Linking

Building a kernel is usually quite a complicated procedure, requiring a significant amount of configuration and several non-standard steps. For this reason, a custom build system is often a must. Sadly, such a component is still missing from the current EVA iteration. Therefore, this section will refer to how EVA interacts with the already in-place Rust build system, Cargo. As of now, relying on Cargo is not a problem, as the future EVA build system will be complementary to it, not a replacement. One of the particular features of Cargo is that it is not considered separable from its compiler, Rustc. Both of them are required to build a Rust project, as they are designed to work together, and replacing just the build system breaks various Rust features, for example, IDE support and procedural macros.

4.3.1. How Cargo Works

First we explore how Cargo works and how we can exploit its features. Rust code is structured into “crates”, each crate is what is usually called a library. Each crate is a self-contained unit of code, which can be self-referential, even supporting circular dependencies across modules. Each crate can optionally define a set of other crates as its dependencies. These dependencies form an acyclic graph structure, as multiple

crates can depend on the same crate, but no circular paths are allowed. A dependency link between two crates can be expressed in three primary ways: through a crate index (such as the official crates.io) by specifying the crate name and version, through git by specifying a URL locating a git repository, and finally via a relative path between the two. The build process starts by constructing the dependency graph, then crates are built from the leaf to the root, and finally everything is linked together. This is key, as it allows optimizations to be performed across crates, since when a particular crate is built, Rustc can see the dependencies source code. Furthermore, the notion that linking is performed at the end is noteworthy, as it allows crates to link circularly despite not even depending on one another, which is further explored in the next section.

4.3.2. Kernel Interface and Linking

We decided to keep most of the EVA kernel code inside a single crate, making kernel development easier at the expense of a bit of modularity. The two components we had to split off are the EVA ABI definitions and the EVA BSP as described earlier.

As we described in Section 4.3.1., Cargo does not allow for circular dependencies across different crates, which is a critical limitation for us. For example, the BSP and the kernel must be able to call each other circularly, as the BSP invokes the kernel for bootstrapping and the kernel invokes the BSP for the portability implementation. For this reason, we use a technique widely used to separate implementation from usage between crates, as seen with the critical-section crate [29]. We will refer to this pattern as “crate decoupling”. This pattern works by having a crate manually export some of its functions by controlling the exported symbol name via the `no_mangle` or `export_name = "..."` attributes. Then a crate wanting to use this symbol simply defines the symbol in an `extern` block, as shown in Listing 13 and Listing 14. This completely decouples the two crates, which now can have any dependency relationship (including none) between each other. EVA uses this pattern in two ways:

- The EVA ABI provides kernel symbols without needing to depend on the kernel itself. This offers two main benefits. First, it allows the kernel to depend on crates depending on the ABI, allowing for a generalized kernel library (EVA klib) that works seamlessly in the kernel and kernel applications. Second, we can build the kernel as a separate entity, with different flags and possibly even a different build system from the kernel application.
- The EVA portability interface needs to find the portability implementation with zero knowledge about how or where it is defined. For this reason, the portability implementation is exported with known names, which are then referenced and used by the kernel itself.

```
1 #[unsafe(no_mangle)]  
2 fn foo() {  
3     // ...  
4 }
```

Rust

Listing 13: Example definition used for crate decoupling.

```
1 extern "Rust" {  
2     safe fn foo();  
3 }  
4  
5 fn bar() {  
6     foo();  
7 }
```

Rust

Listing 14: Example import used for crate decoupling.

However, this does come at a cost. Functions defined and accessed through this interface become opaque to the compiler, which can no longer perform many optimizations such as inlining. This is critical for EVA, as we could lose significant performance going through this interface. As an example, the implementation of certain portability functions is basic, as many of them are just a few instructions long and thus can be inlined. But since Rust uses LLVM as a codegen backend, we can easily regain the lost optimization potential by enabling fat Link Time Optimization (LTO). This compiler feature performs an extra round of optimizations during the final linking phase, utilizing extra information embedded in the intermediate artifacts produced during compilation. This does, in fact, allow for inlining and other optimizations to be performed across our linking boundary, solving our issue. There is probably still some optimization potential lost since LTO is not performed at the Rust level but at the LLVM bitcode level, which, by the nature of being lower level, does not encode some higher level concepts.

4.3.3. Kernel Building

Figure 2 shows different ways in which a project can interact with or include EVA in the build process. In all three cases, solid black arrows signify a “depends on” relationship according to Cargo, while dotted lines indicate a “links to” relationship, referring to the pattern described above.

1. This is the simplest way to include EVA. Applications include both the EVA kernel and the BSP as direct dependencies. Currently, this is also the only way to interact with the internal EVA klib, but this will change in the future. Only Rust can depend on and interact with the kernel directly.

2. This is used for applications wanting to use the Rust std library. In this case the application never actually interacts with the kernel directly and always goes through the kernel ABI. The kernel is brought into the dependency graph through the BSP crate. In the future we might stabilize the portability API, allowing the kernel to be built separately even in this scenario.
3. This is used for applications written in another language or applications wanting to build the kernel separately. We make use of an empty crate called a “kernel build shim” to build and link together both the kernel and the chosen BSP, thus creating a self-contained artifact. Then the application can include this artifact in its build system, giving developers all the flexibility necessary for using whatever language and build system they feel comfortable with.

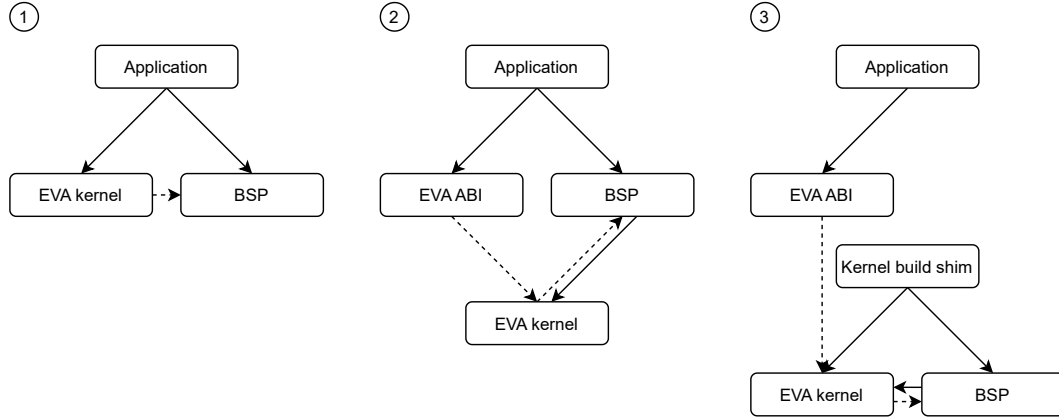


Figure 2: Different kinds of usage of EVA as a dependencies.

4.4. Threads

A thread is a single execution flow, which can be time-multiplexed by the scheduler.

Thread behavior is managed by a state machine (shown in Figure 3). Only threads in the ready state can be executed. Suspend and resume operations transition the thread in and out of the suspended state. Upon termination, a thread internally invokes the exit handler, which, among other tasks, transitions the thread to the terminated state. Since the transition to the final sink state can only be performed by the thread itself, it must be running, ensuring that this transition can only happen while in the ready state. The terminated state is required to represent threads that have concluded execution but can still be interacted with.

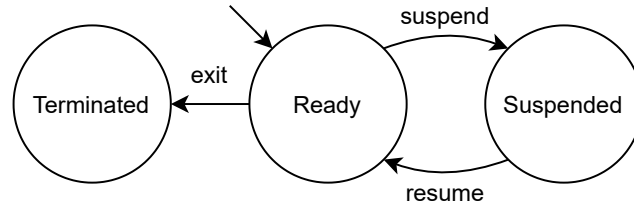


Figure 3: Thread state machine.

Figure 4 shows the internal layout of a thread. This block is subdivided into three areas, all of which is partitioned from a single allocated continuous chunk. These areas are:

- The **Thread Control Block (TCB)**, this area contains everything needed to schedule this thread, for example, its state and pointers used by the scheduler. It also contains pointers to all the other areas for easier access. Thread handles, used to refer to a specific thread, are pointers to the start of the TCB of the referenced thread.
- The **switch context** contains the last saved execution state. This is used by the scheduler to resume thread execution. Although considered part of the TCB this is stored separately since its layout is architecture dependent and dictated by the specific portability layer implementation.
- The **stack** contains the stack used by the thread. Technically this is architecture dependent, but we assume that the stack always grows downward, from higher addresses to lower ones. The stack sits below the switch context to partially protect against stack smashing. When one does occur, it should not corrupt the current thread TCB or switch context. This allows the scheduler to still read valid data, giving it the time to potentially detect the smash and issue an immediate abort.

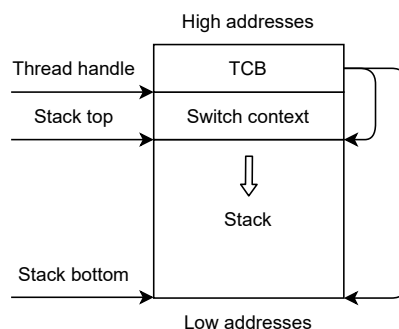


Figure 4: Thread layout.

Thread creation and destruction is controlled via the spawn, join, and detach APIs. The thread spawn is used to create a new thread. This operation involves allocating and initializing both the thread stack and TCB, registering the thread to the scheduler, and finally setting the thread state to “ready”, allowing it to run as soon as possible. Spawn takes a function as a parameter, which will be the thread entry point, the first function executed in the newly created thread context. When that function returns, the thread internally invokes the exit function. This function is responsible for cleaning up and correctly terminating a thread execution. It starts by running various thread cleanup functions, for example, Thread Local Storage (TLS) destructors, then switches the thread execution state to terminated, and finally runs the termination action. Three termination actions are available:

- **No action** (the default): The terminating thread will not perform any action and becomes a “zombie”. No resources are freed in this case, and the system relies on a call to either join or detach to free thread resources.
- **Resume**: The terminating thread will resume execution of another thread, needed to implement the join operation. No resources are freed as we expect the joining thread to do it.
- **Detach**: The terminating thread is responsible for clearing its resources. Since it cannot do this directly, we use a garbage collector that gets run periodically to free these resources. This garbage collector is run inside the idle thread, allowing it to run only if the system has no tasks to schedule. This implies that if an application uses 100% of the CPU time, the system can never reclaim resources used by detached threads.

Join is used to put the invoking thread in suspend and wait until a target thread is terminated. If the target thread is already terminated, the flow continues. Otherwise, the calling thread registers itself for a resume by using the target thread termination action. When the target thread terminates, the thread invoking the join is responsible for clearing up its resources. A thread joining itself results in a deadlock. Finally, detach is used to let a thread clean up its resources. First, the invoking thread checks if the target thread is already terminated, in which case it is the calling thread responsibility to clear up the target thread resources. Otherwise, it sets the target thread termination action to detach and continues execution. A thread is allowed to detach itself.

4.5. Scheduler

The primary feature offered by EVA is a real-time scheduler. This scheduler offers 33 levels of priority, 32 of which are usable for applications, and one is reserved for the idle thread. The algorithm used is highest priority first scheduling, with round-robin across the same priority level. Currently, the scheduler is only designed for single-core systems. This component also includes a timing subsystem, used for task preemption

and timed operations such as task sleep and suspend timeout. The resolution used for timekeeping varies depending on the hardware and specific portability implementation and can potentially go up to nanosecond precision. Despite this, it is inadvisable to go above millisecond precision since the current time subsystem is tick-based, and the performance cost scales linearly with the scheduling precision.

As discussed previously, the scheduler operates on distinct execution flows called threads. A “context switch” is the operation involving the switch between the currently executing thread and another. It is performed by invoking the scheduler code. This operation saves the current execution state in the current thread switch context, then runs the scheduling algorithm to find the next thread to run, and finally installs the new thread switch context as the currently running context. The scheduler code can be invoked in two ways: programmatically through a “yield” operation or periodically via the scheduler tick, the latter being used for preemption and time quantum expiration.

The scheduling primitives are subdivided into layers (shown in Figure 5), and each layer relies on the layers below to provide its services. The main objective of this design is to be “complete”, which means providing applications with all the synchronization primitives required. Layers 0, 1, and 2 provide such completeness, while layer 3 is purely implemented on top of the layers below without any direct interaction with the internals of the scheduler.

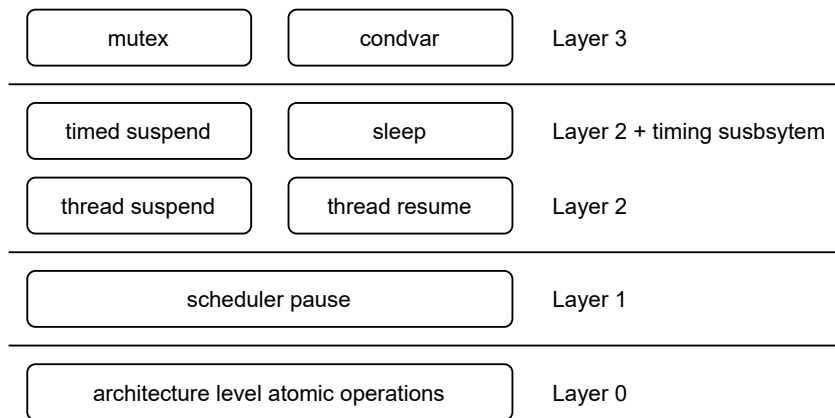


Figure 5: Layers of scheduling primitives.

4.5.1. Layer 0: Architecture Level Atomic Operations

This layer is provided by the hardware itself. The current iteration of EVA requires three atomic operations:

- Atomic loads and stores, operations with guaranteed visibility and ordering semantics in concurrent systems. Since currently EVA only supports single-core targets, these operations are mapped to their non-atomic counterpart.
- Atomic Compare-And-Swap (CAS)
- Atomic swap

An issue arises where these operations are not present at the hardware level, like in the Cortex-M0. Those targets are single-core, and as such, there isn't any form of actual parallelism, and the only source of non-atomicity is the disruption of regular instruction flow by an Interrupt request (IRQ). In these scenarios, loads and stores are always considered atomic, and atomic CAS and atomic swap are emulated by disabling and enabling interrupts globally, avoiding the issue of IRQs. To keep performance high, EVA uses atomic CAS and atomic swap operations sparingly, since, as just mentioned, they are not always directly available. The current iteration of EVA uses those operations just in two cases: to provide a fast path to mutex locking and to implement the IRQ defer operation queue (mentioned in Section 4.5.5.).

4.5.2. Layer 1: Scheduler Pause

The scheduler pause is a way to protect critical sections across different threads and the scheduler itself. A thread makes use of this feature by acquiring a global scheduler pause and releasing it when the critical section ends. The current implementation works by having the scheduler try to acquire a scheduler pause as its first operation, which has three effects:

- If a scheduler pause is already active, the scheduler fails to acquire the pause and cannot run. From the point of view of the currently running thread, this effectively disables any form of preemption.
- The scheduler runs with an active pause, making sure that the scheduling code cannot be interrupted by itself, and only one instance of the scheduling code is allowed to run in a given span of time.
- This mechanism could be used to synchronize normal code with IRQ handlers. The IRQ handler must be aware that acquiring a scheduler pause could fail and must provide a fallback in that case.

When the scheduler code tries to acquire a scheduler pause and fails, it sets a pending flag and immediately returns execution to the current thread. When the current thread finally releases the scheduler pause, it will check the value of this flag, and if set, it will immediately issue a yield. This ensures that if the scheduler gets called during an active scheduler pause, the scheduler will run at the first available instant, which is immediately after a scheduler pause release.

Listing 15 shows the API design for acquiring a scheduler pause in Rust. The function `with_pause` runs the argument `lambda` within an active scheduler pause. This function is also re-entrant, which is important, as it allows this function to be used in destructors and other places where we do not know whether the outside code is already inside a scheduler pause or not. Despite being re-entrant, we use this functionality sparingly, and the kernel provides paused and not paused versions for almost all APIs, avoiding unnecessary pauses when it is known that the kernel is already paused.

```
1 fn callable_only_with_scheduler_paused(_token: PauseToken) {  
2     // ...  
3 }  
4  
5 with_pause(|token| {  
6     callable_only_with_scheduler_paused(token);  
7 })
```

Rust

Listing 15: Example usage of a scheduler pause.

4.5.3. Layer 2: Thread Suspend and Resume

Thread suspend and resume are both low-level primitives to implement waiting. A thread can only suspend itself, and the purpose of this operation is to put the calling thread on an indefinite wait (the thread will continue to run but won't be scheduled anymore after the first preemption or yield operation). To continue scheduling a waiting thread, we must call resume on that thread. By suspending a thread and then resuming it from another one, we can implement wait and signal operations, the basis used to implement more complicated synchronization primitives. Both suspend and resume have paused versions that can be called with the scheduler paused, which is of particular interest for suspends, as this makes sure that the calling thread will still run as long as the pause is active.

Listing 16 shows how a suspend might be used. First, we enclose the entire code inside a scheduler pause critical section. Then we register the current thread for a future resume, which might be a single variable or a complicated data structure holding multiple paused threads. We issue a `suspend_and_yield_paused`, which is a utility function provided by the kernel that releases the scheduler pause, suspends the current thread, and yields immediately. Upon returning from the yield, it reacquires the scheduler pause and continues execution of the parent function. It is important to note how this operation is performed in a loop, and we rely on external information (not necessarily the end of the suspended state) to check that our wait is actually over. This relates to the notion of spurious wakeups, the idea that a suspend might end prematurely, without a matching

resume ever being called. This relaxes requirements on how the scheduler can schedule a suspended thread and makes code using suspend operations more reliable. Even though EVA does not suffer from spurious wakeups intrinsically (threads are guaranteed to never leave the suspended state without a matching resume), we still want resilience over uncontrolled invocations of resume. For example, the kernel implementation of the mutex makes use of suspend and resume operations to function, but user code can still call resume on a thread suspended waiting on a mutex acquire operation. We should be resilient to such scenarios, as the “spurious” call to resume should not cause our mutex code to misbehave. This is critical in kernel code, but user code should follow this convention too. Furthermore, it is important that a resume can be called on a thread not currently in a suspended state, which should have no effect. This allows for multiple sources to potentially resume the thread. Only the first resume has the desired outcome, and all the others are null operations.

```
1 with_pause(|token| {  
2     // Register the current thread for resume  
3     loop {  
4         suspend_and_yield_paused(token);  
5  
6         if /* Resume condition met */ {  
7             break;  
8         }  
9     }  
10 }
```

Rust

Listing 16: Example usage of a suspend.

This layer also interacts with the timing subsystem, allowing wait operations to have temporal semantics. The first of these operations is the timed suspend, which stops the current thread execution for no longer than a set amount of time. Internally, this is implemented by registering the thread inside the timing subsystem, which will issue a resume itself when the deadline expires. Then there is the sleep operation, which is similar to the timed suspend, but all resumes are ignored except for the one coming from the timing subsystem, with the result of suspending thread execution until the deadline expires. All of these operations take a time instant in the future, which can be expressed both relative to the current instant of time, for example, sleep for a set amount of time. Or in absolute terms, referencing some instant during kernel bootstrap as the instant zero. Internally, timekeeping is only absolute, and relative references are converted to absolute ones. This makes absolute timings more precise, as converting from absolute to relative introduces delays, skewing the intended instant of time. Applications should use absolute timings when timing consistency and precision are needed, while relative timings are reserved for simpler cases where precision is not critical.

4.5.4. Layer 3: Mutex and Condvar

EVA also provides higher-level abstractions over the previous synchronization primitives.

The mutex, short for mutual exclusion, provides a way to regulate access to a particular resource. To access a shared resource, a thread needs to first acquire a mutex lock. This lock is not global like the scheduler pause, instead each mutex is instanced and can be acquired individually. If acquiring a mutex fails, implying that another thread has already acquired that particular mutex, the thread in question is put in a suspended state until it can successfully acquire the mutex. When exclusive access is no longer needed, the thread holding the lock releases it, allowing other threads to acquire it. This enables the creation of critical sections that work between threads, which are tied to a particular mutex instance. Multiple threads can therefore access different critical sections simultaneously only if they are tied to different mutex instances.

Condition variables, or condvars are also supported. They are used with mutexes to provide a higher abstraction over resume operations. A thread holding a mutex lock can wait on a condvar, atomically releasing the mutex and suspending the current thread, reacquiring the lock immediately after execution restarts. Other threads can signal one or more threads waiting on a specific condvar, resuming their execution. The atomicity of the wait operation guarantees that the thread is eligible for a wakeup the instant it releases the mutex lock, as wakeups cannot happen after the mutex release but before the thread suspends itself. These primitives are not intrinsically susceptible to spurious wakeups, but as explained in Section 4.5.3., application code should still behave as if they exist.

These primitives do not provide any protection against priority inversion as they do not yet support priority inheritance. They do, however, guarantee strict signaling ordering, which also respects thread priorities. If multiple threads are waiting for a mutex lock or a condvar signal, the order of resumes is guaranteed. Threads with higher priority are always resumed before threads with lower priority, and threads within the same priority group are resumed in order of arrival (First In First Out (FIFO)).

Finally, these primitives support timed operations. Both mutex acquires and condvar waits accept an optional timeout, guaranteeing a maximum wait time after which the operation fails.

4.5.5. Avoiding IRQ Disable in the Kernel

A striking difference between EVA and other embedded unikernels is a lack of what is often referred to as an “IRQ disable lock”. This lock provides a critical section that also

works across IRQ handlers. It can be considered an upgrade to the scheduler pause, as the latter does not provide any guarantee regarding IRQ handler execution. A scheduler pause cannot be interrupted by the scheduler or other thread but can be interrupted by an IRQ handler. EVA avoids this primitive in the kernel intentionally, as we believe that doing so has major benefits:

- A scheduler pause and IRQ lock have a very similar role, and frequently they need to be acquired together. This has two major issues: first, inconsistent ordering for acquiring the two locks has the potential for a deadlock to occur, Second there could be discussions and disagreement about what requires an IRQ lock and what simply requires a scheduler pause. By removing IRQ locks completely in the kernel, we avoid both of these issues.
- Modern IRQ controllers are very nuanced and allow for very fine control over singular interrupts. But an IRQ lock ignores this complexity and instead condenses it into a single enabled or disabled state. We could explore a more advanced API exposing IRQ control to the application, but this unnecessarily increases coupling between the kernel and the hardware, something we want to avoid as stated in Section 4.2.3.. By avoiding IRQ locks completely, we retain much more freedom with how applications handle and manage interrupts.
- By not disabling IRQs, we retain much more responsiveness to IRQs, arguably a critical property of the system. For example, clock skew can be accumulated when an IRQ lock critical section length exceeds the tick IRQ period, and multiple tick IRQs are missed.
- Since the kernel does not use any form of IRQ lock, the kernel API never requires nor enforces a particular IRQ state. This in turn pushes IRQ management inside of drivers, the only place where IRQ state actually matters.

All of the above can be summarized in the following concept: the notion of IRQ is a key concept when interfacing with hardware directly, but both kernel and application code work at a higher level of abstraction and should not worry about lower-level details. Moreover, there is no point in abstracting and generalizing IRQ management at the kernel level. Device drivers are the primary users of IRQs and, by definition, are hardware dependent, and they do not require any further generalization.

This creates one major issue: kernel pauses cannot be reliably acquired in every execution context, such as IRQ handlers, in turn making most, if not all, of the kernel API unavailable in those scenarios. The solution is to provide a very limited subset of kernel APIs, which are guaranteed to be lock-free, thus making them available in all execution contexts. The only operation currently supported is thread resume, which should be enough for the majority of use cases. These APIs work by using an intermediate lock-free queue between the kernel and code calling the API, called “IRQ defer operation queues”, since they are used to defer certain kernel operations from inside an IRQ context or any other context where scheduler pauses are not available. The implementation details

of these queues are explained in Section 5.1.3.. These queues contain operations to be performed by the next scheduler code execution, and the code calling these APIs is responsible for ensuring that the scheduler is run as soon as possible. For example, by invoking an immediate yield or interacting with the interrupt controller to issue a scheduling interrupt directly.

4.5.6. Idle Thread

The scheduler must always have at least one thread ready. To ensure this, we add a special thread, referred to as the idle thread, which is guaranteed to never get suspended or terminated. This thread has a special priority that cannot be used for normal threads and is considered below every other allowed priority, so the idle thread will always be executed only if no other thread is ready. Despite being the lowest priority, this special thread has some important tasks to perform. During bootstrap it is the first thread to run, and as such, it is responsible for finalizing the scheduler starting procedure (more on this in Section 4.7.). Then, during normal execution, it performs some periodic clean-up tasks, such as clearing terminated detached threads. Special care needs to be taken so that the idle thread never holds a scheduler pause for longer than necessary. For example, when cleaning up multiple detached threads, the scheduler pause is acquired and released after each cleanup operation to allow other threads to run in between.

4.6. Other APIs

Since EVA is more than just a scheduler, it provides some auxiliary functions too. Most of these are designed around the Rust std library, which requires a minimum set of features to support a new operating system. These miscellaneous functionalities are:

- A function to retrieve the current time, which is expressed in nanoseconds from the bootstrap of the operating system. This function goes directly to the portability layer, so actual resolution and inner workings depend greatly on the BSP used.
- A memory allocator. During bootstrap, the BSP registers to the allocator all available memory. The algorithm currently used for allocation is a linked list allocator with first fit, but in the future different algorithms will be explored. The basic interface is identical to the one used by the Rust alloc and std library. To support C and other languages, a small layer that emulates the behavior of malloc and free is provided.
- Textual input and output. These also go straight through to the portability layer and are an implementation detail of the BSP.
- TLS support. This is provided strictly to comply with requirements for the Rust std library. Applications create a set of global keys, which can then be used by each thread to retrieve their local copy of the value associated with the key. Keys can also have associated destructors, which are run as part of the thread cleanup process. We

provide the same level of guarantees as pthread keys. Destructors can set other TLS values while running, as the API is re-entrant. Destructors are guaranteed to run until all values are destroyed, even in the scenario of values added by another running destructor. This way of implementing TLS is known as “dynamic TLS”, which differs from “native TLS” in the sense that TLS entries are allocated dynamically and not statically at compile time.

4.7. Bootstrap Procedure

Bootstrapping EVA is performed by the BSP, and even though it is currently an implementation detail of the BSP, we provide some guidelines to follow. Below is a description of the individual steps:

1. Execution starts. These first few initialization steps are written in assembly since they are used to prepare the minimum set of features required by Rust to run soundly. These include:
 - Zero out the `.bss` section.
 - Copy initial values into the `.data` section.
 - Enable any required processor feature, such as the Floating Point Unit (FPU).
 - Set up a working stack. This can be either a temporary stack or the same stack used by interrupt handlers.
2. **Stage 0 initialization:** this step is optional and is used to perform additional initialization steps that are not related to EVA but are performed directly in Rust. An example of such a step is initializing the system and CPU clocks.
3. **Stage 1 initialization:** this is the first step requiring interaction with EVA. From this point onward, the steps are no longer generic but specific to how EVA works. It is subdivided into the following sub-tasks:
 - 3.1. Set up a working input-output interface used for logging. Any progress or error past this point can now be reported to the outside world.
 - 3.2. Register at least one memory region usable by the EVA dynamic heap allocator. This is required as all further steps require dynamic allocation to be available.
 - 3.3. Enable everything required to perform a yield operation. Usually this means enabling and configuring a software interrupt dedicated for this purpose.
 - 3.4. Prepare the global timer and the periodic tick used for task preemption.
 - 3.5. Spawn the initial thread, which will be used to run the stage 2 initialization code and the main application code.
 - 3.6. Invoke the EVA scheduler init function, which will create and register the idle thread as well as configure it as the first thread to be executed.
4. Upon returning from stage 1, the code can execute to the first thread. To do so, it manually loads the currently installed thread context and jumps to it.

5. Execution of the idle thread starts, which sets the scheduler state to running and yields to the first available thread immediately after. This concludes the scheduler initialization, and the system can finally properly schedule threads.
6. **Stage 2 initialization:** at this point any initialization requiring a working scheduler can be performed. When the system is ready and set up, the stage 2 initialization code invokes the application main, and bootstrapping is concluded.

5 | Implementation

The subjects of this chapter are the implementation details of the various components of EVA. We already explained the reasoning behind various technical choices, here we simply dive into how those objectives are concretely achieved.

5.1. Data Structures

Data structures are foundational to the entire operating system. As such, they require the utmost attention, since a wrong design decision could easily cause a butterfly effect on the entire system. For this reason, as of now, the design of these data structures is where most of the engineering efforts were spent.

5.1.1. Linked List

Linked lists might seem trivial, as they are usually learned during an introductory course on programming and computer science, yet designing them soundly and correctly around EVA was probably one of the hardest challenges of this entire work.

The definition of linked lists is simple: an ordered sequence of elements similar to arrays, but where each element is allocated separately from the rest. Structure to the data is provided by pointers embedded in each node, which are used to retrieve the next and optionally the previous node in the sequence. Finally, a root element exists to provide an entry point into the list. This creates a tradeoff: random access is prohibitively slow, since it requires iterating over the entire sequence. Sequential access is also generally slower compared to traditional arrays, as this structure is not cache-friendly and pointer dereferences are relatively slow. However, we gain two major benefits: removal is extremely fast, as we can simply adjust the pointers of nodes adjacent to the one we are removing, and nodes can be allocated individually. This last property is important to provide “zero allocation linked lists”. Since the lifetime of each node allocation unit is tied to that of the node itself, we can use the stack to allocate the node if we know that said node will not outlive the current scope. Another significant property of linked lists is intrusivity. An intrusive linked list stores the pointers required for its operations inside the node itself, instead of outside in a wrapper structure. This allows nodes to

store multiple pointers, allowing them to be inserted inside multiple linked lists or even multiple data structures that share the same intrusive property.

There exist several ways to implement a linked list. Here we highlight three, which are also represented visually in Figure 6:

1. We use a sentinel value (usually null) to signal the end of the linked list. The linked list then contains pointers to either or both ends of the list. This is the simplest and most common way of implementing a linked list and typically what gets taught in introductory courses.
2. The list is fully circular, as the last node points back to the first one. The root of the list now stores a single pointer to the head of the list. The main advantage of this design is that it lacks a sentinel value, and the root is lighter, as it uses only one pointer for a doubly linked list. Unfortunately, implementation is generally a bit more complicated compared to the traditional design.
3. The root of the list is itself part of the list. This requires a particular design where nodes are not necessarily homogeneous, since the root is a node. The implementation of this linked list is easier since updating the root doesn't need any special logic, and removing a node can be done without direct access to the root. In turn, this simplifies logic around these lists, as we don't have to track externally which list contains this node. The tradeoff is that the root cannot be moved or copied around in memory, as doing so invalidates the entire list.

We decided that for EVA, design number 1 was the best. Having a movable root is essential for certain data structures (for example, mutexes), and modeling an unmovable (or pinned) root in the Rust type system is quite challenging and error-prone. Design 2 doesn't really offer many advantages over design 1. Finally, we decided to use just a single implementation for the entire operating system, as design number 1 is flexible enough to cover all possible applications.

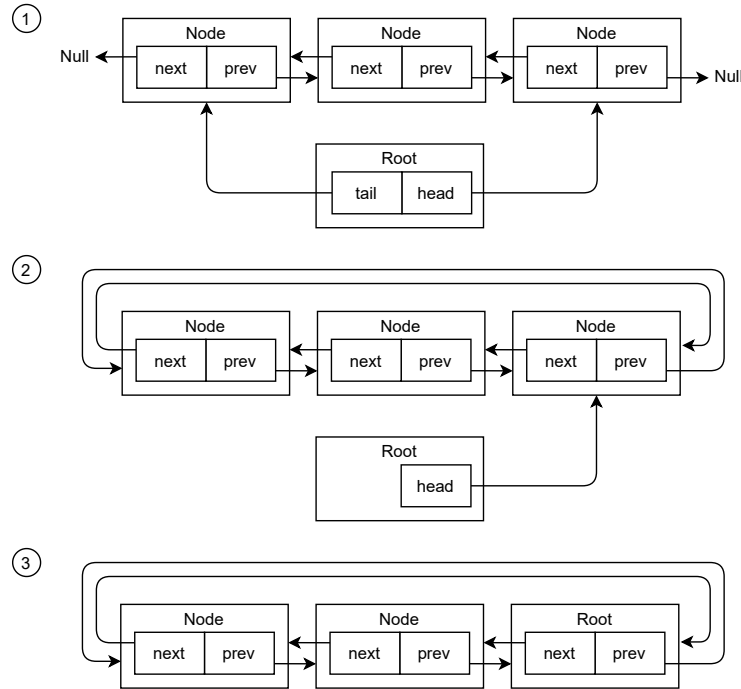


Figure 6: Visualization of possible linked list implementations.

Another major issue when designing linked lists in Rust is node aliasing. As discussed earlier, intrusivity of the linked list is a desirable property. This, among other advantages, allows us to keep a reference to an inserted node outside the linked list data structure, for example, by using reference counting. This creates a major problem, though: aliasing. As we can see from Figure 7, when we are allowed to keep references to nodes outside the linked list, we open multiple paths to reach a given node. In the example, node 2 is accessible both via a direct reference and through the root of the linked list, shown via the dotted line. Aliasing is a considerable topic in Rust, as this language strictly forbids certain types of mutable aliases to occur. As such, modeling the linked list becomes quite challenging. Solving this problem requires several implementation details:

- The pointers stored inside the node are wrapped so that they are marked as internally mutable. The specific structure depends on the requirements of the overall linked list. We use `Cell` for when sharing the root and notes between threads is not necessary, for example, the TLS, and `Atomic<...>` for every other scenario. Since we can mutate these pointers only by going through a mutable reference to the root, it might be possible to use `Cell` even when sharing data between threads is required. The mutable reference to the root ensures that only one thread can mutate these pointers at every given instant. This is not possible in our case since we provide a convenience method

to check whether a node is inside a list or not, and this method reads the pointers directly. This creates the possibility of concurrent writes and reads. As such, we must use atomic access in this case.

- The entire node needs to be marked as `!Unpin`. This has two purposes. First, it is required since when a node is inserted inside a linked list, it cannot be moved around in memory without corrupting the list. Second, this marks the entire structure as potentially aliasable according to the code generation backend.
- We cannot generate any mutable reference to a node from the root itself, as doing so would potentially create a mutable alias on the node. The current implementation can optionally support this, but only if we assert that all nodes contained in a list are owned by that list. In that case, we can obtain references to a particular node only by going through the root itself.
- Some implementations of linked lists never create any direct reference to the node or links when mutating the list. This is a very conservative approach. When we create either a mutable or immutable reference to an object, we are implicitly asserting a series of statements about the object itself. By never creating a reference in the first place, we avoid accidentally asserting false statements, which can be used by the optimizer to optimize away critical code. In the present implementation, we don't use this conservative approach since it adds several complications and has a greater potential to introduce undefined behavior. Despite being more reliable and correct, it requires a more in-depth understanding of the machine model used by Rust.

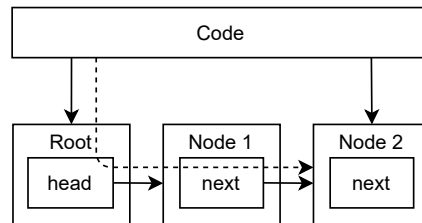


Figure 7: Visualization of node aliasing.

As noted previously, one of the primary reasons why linked lists are used so pervasively throughout EVA is the ability to work without dynamic memory allocation. We call these linked lists “zero allocation linked lists”. Listing 17 shows exactly this pattern. We can see how the node is fully allocated on the thread stack, avoiding dynamic allocation, which would be significantly slower compared to stack allocation. We make use of the `UncheckedRef` discussed in Section 3.1. to erase the lifetime and manually assert the correctness of this code.

```
1 fn foo() {
2     let node = Node {
3         // ...
4     };
5
6     list.push_back(unsafe {
7         // SAFETY: The node is always removed from the list before going out
8         // of scope
9         UncheckedRef::erase_lifetime(&node)
10    });
11    // ... other code ...
12
13    if node.is_linked() {
14        // Node is still linked, we must remove it
15        let mut cursor = unsafe {
16            // SAFETY: This node is inside the list and the pointer is valid
17            list.cursor_mut_from_raw(NonNull::from(&node))
18        };
19
20        cursor.remove_current();
21    }
22 }
```

Rust

Listing 17: Example of “zero allocation linked lists”.

As a final note, the majority of the complexity comes from the API design. It was challenging to hide all of the above concepts while maintaining the necessary flexibility. This required quite a significant amount of iteration and effort, but the final result is correct while mostly using information known at compile time. Some amount of runtime checks are still needed, but this has been reduced to a minimum.

5.1.2. Sorted Linked List

Sorted linked lists are used when a specific element ordering is required. They work by supplementing a normal linked list with a predicate for evaluating ordering between elements. This kind of list, among other use cases, is used by the timing subsystem to keep events ordered by time. We must note one crucial concern about this data structure: it is hardly the best structure in any given scenario. Sorted insertion inside a linked list requires potentially scanning the entire list, making this operation $O(N)$ in terms of computational scaling. This is easily beaten by smarter sorted data structures

such as the red-black tree or AVL tree without loss of features. This is a conscious decision, as big-O notation gives a good sense of scaling but fails to capture effective performance, especially in situations with low N counts. As such, performance needs to be studied more carefully to achieve good results for both low-count N and high-count N cases. With very few N, a sorted linked list performs surprisingly well and is easy to implement, which is why it has been chosen. In the future, a more in-depth analysis will be performed, and a more suitable data structure will replace the current implementation.

5.1.3. Lock-free Stack

Both Section 5.1.1. and Section 5.1.2. describe data structures that require full mutable access to the entire data structure to mutate, implying that they cannot be used across multiple threads without some form of locking around them. A faster alternative is a lock-free stack, also known as a Treiber stack [30], which supports only a minimal set of operations but can be used safely in concurrent contexts without locks. In our system, the supported operations are push, which inserts an element at the head of the stack, and take, which atomically removes all elements from the stack and returns them as a new non-atomic list.

Both of these operations are implemented in a lock-free way, completely negating the requirement of a lock. As such, they can be used in various contexts where such locks are not available, for example, inside IRQ handlers. Linked lists have historically been the primary candidates for lock-free data structures, but this usually comes with a big caveat: they require a complicated allocation scheme to prevent stale pointers from becoming dangling prematurely. We avoid this completely by supporting only the minimum set of operations required by our system. “push” does not access any node besides the node currently being inserted, which by definition is not inside the list yet and cannot be deallocated. “take” atomically takes full ownership of all the nodes before letting code access any of the nodes.

A pseudocode representation of these operations is shown in Listing 18. These algorithms make use of atomic CAS and atomic swap operations, both of which could be implemented using a retry loop, depending on the specific hardware platform. This can create problems in hard real-time systems with very high levels of compliance, as a retry loop does not provide any upper bound on the WCET. It does, however, provide the guarantee of progress. When a retry operation fails, it is guaranteed that at least one other execution thread has succeeded and completed its operation. In practice the absence of an upper bound on execution time is not a concern, as these retry loops are very short and starvation is never realistically an issue. However, going forward, we might need stricter proof to comply with higher standards of safety.

```
1 fn take(list) {  
2     return SWAP(list.head, null);  
3 }  
4  
5 fn push(list, node) {  
6     let mut head = LOAD(list.head);  
7     loop {  
8         node.next = head;  
9         if CAS(list.head, head, node) {  
10             break;  
11         } else {  
12             head = LOAD(list.head);  
13         }  
14     }  
15 }
```

Pseudo

Listing 18: Pseudocode for “take” and “push” operations.

5.1.4. Slot Map

Slot maps are used to provide fast random access by key to a set of elements. The interface is very similar to any other map, but the keys are generated by the structure itself and obtained upon insertion. This data structure is useful when we want to have a lightweight handle that can be easily moved around and has guarantees of safety, but we do not have any requirements on the handle itself. Accessing a stale handle should return an error instead of causing undefined behavior like a dangling reference. Compared to traditional hash maps, slot maps retain the same $O(1)$ retrieval time, but they have stronger guarantees, as access is always constant, not amortized, and with a lower constant factor. The only downside is that they cannot release memory reliably. Reducing the number of elements in a slot does not reduce its memory footprint, as the memory utilization will always be a multiple of the maximum number of concurrent values that have ever been stored in the data structure.

The implementation of this data structure utilizes a normal resizable array as the backing storage. Each element of this array contains a value, which might be populated or not, an integer identifying the current generation of this slot, and a boolean indicating whether this slot is used or not. The last two elements are often combined into a single integer to simplify implementation. A handle is composed of an index into the array and a generation value. Each time a value is removed, we mark it as free. To insert a new value, we first check if a free slot is available. If one is available, we increment its generation and return the index of the slot along with the new incremented generation.

Otherwise, we append to the array a new slot with generation zero and return the corresponding handle. To speed up querying for free slots, we store an additional index indicating the first available free slot or none if the array is full. Then, inside each free slot, we use the memory reserved for the value, which is unused in this case, to store the index of the next free slot. This forms a chain that can be used to efficiently retrieve all the free slots. This chain must be kept in sync by insert and remove operations. Finally, to retrieve an element, we use the index portion of the handle to efficiently retrieve the slot, then we use the generation to check if this slot has been reused during this handle lifetime. This last check provides some safety since indices are reused very frequently. A handle composed of just the index can easily outlive a remove and insert operation without correctly detecting that the value has changed. We do still note that this guarantee is weak and best effort, since the generation has a finite amount of bits it can overflow, causing old generations to be reused just like indices. Despite this being a rather rare occurrence, we still inform programmers of this behavior and consider relying on this property to strongly detect stale handles a mistake.

These data structures are currently only used by the TLS subsystem to allocate new keys. In the future we might also use them to generate unique thread IDs or phase them out completely in favor of some other solution.

5.1.5. Bit Set

In situations where the range of an integer set is small, bit sets are used to represent the set in an efficient and compact manner. This data structure provides both $O(1)$ insertion, deletion, and query. Querying for the largest element is dependent on the size of the set, but it has an extremely low constant factor, so even for large bit sets it provides compelling performance. Elements inside a bit set are stored as bits. The number of bits provided determines the range of values that can be stored. For example, using a simple 4-byte integer as a backing storage will yield 32 bits available, and it can store numbers between 0 and 31. This data structure trades the flexibility of a traditional hash set or binary tree, both being able to store arbitrary elements, for an extremely compact representation and fast access times. Another advantage of this data structure is that many operations can be optimized via hardware. For example, retrieving the highest and lowest value elements can be done by calculating the number of trailing and leading zeroes respectively, both of which are often hardware-accelerated operations for small integers.

This is currently used to record which priority levels have a non-empty ready queue, thus making retrieval of the highest priority non-empty queue almost a single hardware operation. For this exact reason, EVA currently supports 32 actual priority levels, since they fit perfectly in a bit set backed by a single processor word.

5.2. Scheduler Pause Implementation

The scheduler pause primitive has been carefully designed to only use atomic loads and stores and completely avoid any other atomic operation. The reason for this choice is twofold. First, not all targets support atomic operations, with all the consequences and considerations already done in Section 4.5.1.. Second, even if the target does support atomic operations, depending on the particular implementation, they might not have any guarantee about the execution time.

“try pause”, shown in Listing 19, tries to pause the scheduler, an operation that can fail only if the current thread has already acquired a scheduler pause. As such, it can be used to provide reentrant scheduler pausing. “unpause”, shown in Listing 20, unpauses the scheduler, assuming that the scheduler is paused and thus can never fail. This last operation is also responsible for yielding immediately if a scheduler invocation is pending, signified by the pending status flag. Finally, “run scheduler”, shown in Listing 21, is used to run the scheduler with an active pause, the reason for which was given in Section 4.5.2..

```
1 fn try_pause() {  
2     if LOAD(PAUSED) {  
3         return false;  
4     } else {  
5         STORE(PAUSED, true);  
6         return true;  
7     }  
8 }
```

Pseudo

Listing 19: Pseudocode for scheduler try pause implementation.

```
1 fn unpause() {  
2     STORE(PAUSED, false);  
3     if LOAD(PEND) {  
4         yield_now();  
5     }  
6 }
```

Pseudo

Listing 20: Pseudocode for scheduler unpause implementation.

```
1 fn run_scheduler(f) {  
2   if LOAD(PAUSED) {  
3     STORE(PEND, true);  
4   } else {  
5     STORE(PAUSED, true);  
6     STORE(PEND, false);  
7  
8     f();  
9  
10    STORE(PAUSED, false);  
11  }  
12 }
```

Pseudo

Listing 21: Pseudocode for run scheduler implementation.

Despite using only loads and stores, this code is absent from any race condition stemming from the global paused state. The key intuition is that despite being part of global state, paused behavior is only visible locally, as it is impossible for a thread to witness a store coming from another thread. This is shown in Figure 8. Dotted lines show possible preemption points. The only preemption point that could cause thread A to experience the global variable becoming true would be the preemption point marked with 1. Ultimately this is impossible, as at that instant the scheduler pause is active and a preemption cannot happen until paused returns to the value false, the value originally read by thread A. From thread A’s perspective, nothing happened. “run scheduler” also does not have any issues, since this function can only interrupt a thread and not the other way around, implying that it is impossible that a thread gets to run in the middle of the scheduler routine. The pending state has similar guarantees, as it is always set inside an active scheduler pause and always read outside a scheduler pause. Thus, it is impossible to miss a pending yield, which is critical, since missing a pending yield can lead to a higher priority thread waiting for longer than necessary, reducing the overall reactivity of the system. It is possible to perform an extra yield, though. If inside the “unpause” routine, a preemption happens exactly between the load of pend and the yield performed by the unpause, once execution returns to the unpause routine, the yield is still performed, even though the scheduler has already been run. For EVA, this is an acceptable compromise.

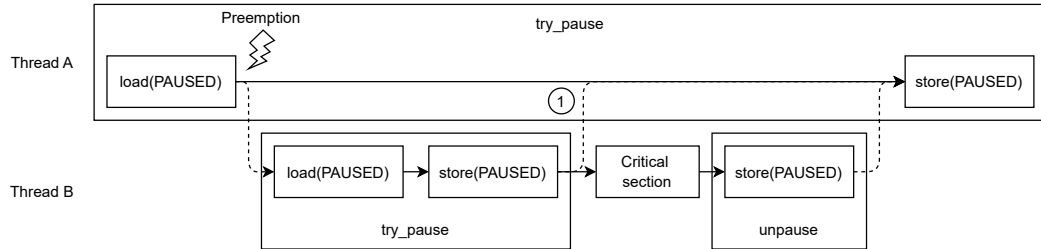


Figure 8: Preemption points during scheduler pause.

5.3. Kernel Allocator

The main allocation algorithm is provided by the `linked_list_allocator` crate [31], an already popular and widely used library. This library implements a simple linked list based allocation algorithm [32]. Internally unused memory regions are stored inside a linked list containing the start and end addresses of the region. A first-fit algorithm is used to locate a suitable region. This algorithm finds the first area greater than or equal to the requested size. If the requested size is smaller than the size of the area, a split is performed, and the remaining unused memory region is put back into the list. When freeing memory, the allocator checks for the possibility of coalescing neighboring regions, which get immediately merged back into a single memory region. This algorithm is rather simple and has a very low memory footprint, as the links of the linked list are stored in the unused memory regions, requiring no external data structure to track memory state. This makes it ideal and commonly used in embedded environments.

In the Rust allocator, the layout of the allocated region is passed both during allocation and deallocation, meaning that the allocator does not need to track this information separately. This is in contrast with what the C standard library does, where the only argument for deallocation is the address of the region to be deallocated. To solve this issue, C allocators track the size of the allocated region inside the allocated region itself, in a small header just before the start of the portion of the allocation usable by the program. Since we want to support languages besides Rust, we decided to provide optional emulation for this behavior in our allocator. By using a different set of APIs, application code can obtain the same semantics and functionalities of `malloc` and `free`.

5.4. Scheduling

Scheduling is performed via a highest priority first, round-robin algorithm. Threads are stored in up to two linked lists at a time.

The first list is used to track all threads that are currently allocated and possibly running, therefore considered “alive”. This is used to check if a thread handle corresponds to a valid thread or not. To query for this information, we perform a linear search on the list. This functionality is used in most application-facing APIs to prevent application code from inadvertently invoking operations on a non-existent thread. Applications can opt out of this check by using unsafe, unchecked versions of said functions. Using a linear search makes this operation unnecessarily slow. An exploration of alternative solutions will be done in future developments.

The second list is used to hold all threads with a ready state. Any thread can be present in one and only one ready thread queue. Figure 9 shows a simplified representation of these queues. Threads transitioning to the ready state are appended at the back of the queue corresponding to their priority level. Each of these queues provides the round-robin functionality, as ready threads get inserted in the back, while threads to be scheduled get pulled from the front. A bit set accelerates the search for the highest priority non-empty scheduling queue. Finally, a special bucket is reserved for the idle thread, which does not require an entire queue and gets selected only when the accelerating bit set is empty. The currently running thread is treated specially, and despite being in the ready state, it does not belong to any queue and instead is in its bucket. When the scheduler code runs, if the previously running thread is still in the ready state, it gets appended to the correct ready queue. This is done to respect the round-robin property. Threads transitioning to the ready state during the execution of the current thread should be scheduled before the current thread. Therefore, by inserting the just preempted thread at the back in the corresponding ready queue at the end of its run period, we guarantee this property.

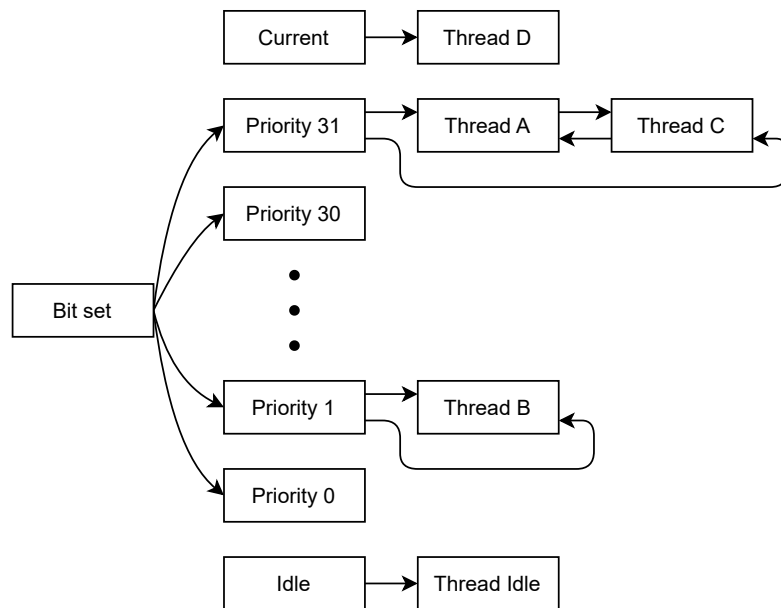


Figure 9: Scheduler queues.

5.5. Priority and Timed Wake Lists

The basis for the timing subsystem and synchronization primitives are the priority and timed wake lists. These lists are used to store waiting threads and provide a way of resuming their execution according to a specific order. Both of them are based on a sorted linked list, and the only difference is the ordering criterion. Priority wake lists sort the waiting threads by their priority, while timed wake lists sort based on time of resume. Listing 22 shows pseudocode for how the wait and signal operations are implemented internally. The wait node contains the current thread handle and other information, for example, the deadline or priority, and is allocated on the stack to avoid dynamic allocation. This node then gets inserted into the list, and the thread is repeatedly suspended until the node is no longer in the list. This is used to protect against spurious wakeups since only a signal operation will actually remove this node from the list. The signal operation tries to remove one node from the list and resumes its execution. We do note that this example is quite bare-bones for the sake of brevity. The actual implementation is more complicated and has to handle considerably more use cases, such as interrupting the wait early.

```
1 fn wait(wait_list) {
2     let node = WaitNode {
3         thread: current(),
4         // ...
5     };
6
7     wait_list.push(node);
8     loop {
9         suspend_and_yield();
10        if !node.is_linked() {
11            break;
12        }
13    }
14 }
15
16 fn signal(wait_list) {
17     let node = wait_list.pop();
18     if node {
19         resume(node.thread);
20     }
21 }
```

Pseudo

Listing 22: Pseudocode for wait and signal operations.

These data structures are only used internally and never made public, since the API is designed around a particular issue and is very specific to Rust. The issue in question is that of waiting on multiple events at the same time. For example, a thread can decide to implement a timeout by waiting on both a priority wake list and a timed wake list. The easiest and most common way of solving this issue is by nesting the two wait operations. This is shown in Listing 23. We can see that the inner wait is not protected against spurious wakeups and instead returns immediately, only unlinking its waiting node if necessary. This is done to allow for the outer wait to escape the inner in case of a resume. This is essentially how the timed suspend is implemented in the application facing APIs, as it behaves very similarly to the inner wait. Despite being the most intuitive, this design is quite limiting. First, it is not flexible enough to easily provide multiple combinations of inner and outer waits. Due to this limitation, implementations end up with just one combination: an inner wait providing the timeout operation and the outer for the actual event wait. Second, it quickly degrades in performance if multiple spurious wakeups occur. A spurious wakeup in this scenario causes the inner wait to exit and the outer wait to reissue an inner wait, which causes the inner wait to re-insert the node inside the queue, possibly an expensive operation. Indeed, since our current

implementation uses a sorted linked list, the act of inserting a node is considered a slow operation, and we want to avoid this if possible.

```
1 fn wait_inner() {
2     let node = WaitNodeInner {
3         // ...
4     };
5
6     wait_list_inner.push(node);
7     suspend_and_yield();
8
9     if node.is_linked() {
10         wait_list_inner.remove(node);
11     }
12 }
13
14 fn wait_outer() {
15     let node = WaitNodeOuter {
16         // ...
17     };
18
19     wait_list_outer.push(node);
20     loop {
21         wait_inner();
22
23         if !node.is_linked() {
24             break;
25         }
26     }
27 }
```

Pseudo

Listing 23: Pseudocode for inner and outer waits.

To solve both of these issues, we redesign the API completely. The wait operation now accepts a lambda, which is responsible for looping on the waiting condition and suspending. This is shown in Listing 24 and Listing 25.

```
1 fn wait(f) {  
2   let node = WaitNode1 {  
3     // ...  
4   };  
5  
6   wait_list.push(node);  
7   f(node);  
8   if node.is_linked() {  
9     wait_list.remove(node)  
10  }  
11 }
```

Pseudo

Listing 24: Pseudocode for improved wait.

```
1 wait_outer(|node| {  
2   loop {  
3     suspend_and_yield();  
4     if !node.is_linked() {  
5       break;  
6     }  
7   }  
8 })
```

Pseudo

Listing 25: Example usage of improved wait.

We can then combine multiple waits easily by just nesting waits at the outer level. This works because we have effectively flipped where the wait loop is located. Instead of being at the top level of the nested waits, it now sits at the bottom. Listing 26 and Listing 27 show two different approaches to combining waits. The first uses an “or” as its exit condition, meaning that either of the two wait conditions needs to be fulfilled before exiting. Conversely, the second shows an “and” condition, something that was not as easily achievable with the previous approach. We can see how combining and nesting multiple waits is expressed much more naturally. Finally, spurious wakeups are also no longer an issue, since the loop is performed internally, implying that we never remove nodes from lists that might still be necessary.

```
1 wait_outer(|node_outer| {
2     wait_inner(|node_inner| {
3         loop {
4             suspend_and_yield();
5             if !node_outer.is_linked() || !node_inner.is_linked() {
6                 break;
7             }
8         }
9     })
10 }
```

Pseudo

Listing 26: Example of combining multiple waits (OR).

```
1 wait_outer(|node_outer| {
2     wait_inner(|node_inner| {
3         loop {
4             suspend_and_yield();
5             if !node_outer.is_linked() && !node_inner.is_linked() {
6                 break;
7             }
8         }
9     })
10 }
```

Pseudo

Listing 27: Example of combining multiple waits (AND).

5.6. Timing Subsystem

The timing subsystem is responsible for providing the infrastructure to resume thread execution after a given amount of time. Internally this subsystem uses the timed wake list described in Section 5.5.. Threads register themselves for timed resume via the internal `with_timed_wakeup` function. The same functionality is exposed to applications through `suspend_and_yield_paused_until`. Each time the scheduler code is invoked, with the modalities explained in Section 4.5., it runs the timing subsystem to resume all threads with an expired deadline. The timing subsystem does not implement any timekeeping mechanism internally and instead relies solely on the “get time” primitive provided by the portability layer. This is done mainly to not accumulate clock skew, as the execution of the timing subsystem is not guaranteed to have a strict periodicity. As a side effect, using a separate clock for resume allows the system to be tied to the resolution of the clock, not the periodicity of the scheduler tick. This is crucial when the application CPU time utilization is below 100%, which in real-life applications is

almost always the case, as in that scenario the idle thread is used to fill the unused time. The idle thread is designed to invoke the scheduler code constantly, which in turn executes the timing subsystem resume routine. As a consequence, the timed wake list is checked much more often, achieving a wakeup resolution much closer to that of the external clock.

5.7. Mutex and Condition Variables

Both mutexes and condition variables might seem relatively trivial but present interesting challenges when trying to provide a fair and robust implementation.

Our mutex implementation is based on a priority wake list and a single atomic boolean variable indicating whether the mutex is currently locked or not. Mutex locking, shown in Listing 29, first tries to acquire the lock directly with a single CAS operation, providing a fast path in case the mutex is not locked, which should be the majority of cases under normal circumstances. Since not all hardware architectures support CAS efficiently and instead resort to emulation, this fast path could be disabled on such architectures and solely rely on the slow path. If this first acquisition fails, we fall back to the slow path. We first enter a scheduler pause, then try to lock the mutex again using a CAS, since a preemption could have occurred between the first attempt and us obtaining a scheduler pause. If we fail once again, we resort to the priority wait list and wait until an unlock operation signals us. An unlock operation, shown in Listing 28, begins by trying to signal one of the threads in the wake list. If this succeeds, the new thread will wake up and own the lock, so we do not need to unlock the mutex yet. This provides fairness of the lock, as no other thread can “sneak in” and steal the lock during this lock owner transitioning phase. If no thread is currently waiting, we simply release the lock.

```
1 fn mutex_unlock(mutex) {  
2   with_pause(|token| {  
3     let waked = mutex.wait_list.wakeup_one(token);  
4     if !waked {  
5       STORE(mutex.locked, false);  
6     }  
7   })  
8 }
```

Pseudo

Listing 28: Pseudocode for mutex unlock implementation.

```
1 fn mutex_lock(mutex) {
2     if CAS(mutex.locked, false, true) {
3         return;
4     }
5
6     with_pause(|token| {
7         if CAS(mutex.locked, false, true) {
8             return;
9         }
10
11         mutex.wait_list.with_wakeup(token, |wakeup| {
12             loop {
13                 suspend_and_yield_paused(token);
14                 if wakeup.is_signaled() {
15                     break;
16                 }
17             }
18         })
19     })
20 }
```

Pseudo

Listing 29: Pseudocode for mutex lock implementation.

While implementing condition variables, it is easy to commit critical mistakes in the wait operation. We must maintain atomicity of the mutex unlock and notify wait, as the first notification event happening after the mutex lock must be captured by the waiting thread. To do so, we introduce paused variants of the mutex lock and unlock, which can be used within an active scheduler pause. With these new operations, we can unlock the mutex without the possibility of a context switch before we actually register ourselves in the wait list. The implementation of the wait operation is shown in Listing 32, and we can see how the listing reflects this. Finally, the notify operation, both variants shown in Listing 30 and Listing 31, simply issues either a single or multiple wakeups, as that is all that is necessary to signal the waiting threads.

```
1 fn condvar_notify_one(condvar) {
2     with_pause(|token| {
3         condvar.wait_list.wakeup_one(token);
4     })
5 }
```

Pseudo

Listing 30: Pseudocode for condvar notify one implementation.

```
1 fn condvar_notify_all(condvar) {
2     with_pause(|token| {
3         condvar.wait_list.wakeup_all(token);
4     })
5 }
```

Pseudo

Listing 31: Pseudocode for condvar notify all implementation.

```
1 fn condvar_wait(condvar, mutex) {
2     with_pause(|token| {
3         mutex_unlock_paused(mutex, token);
4
5         condvar.wait_list.with_wakeup(token, |wakeup| {
6             loop {
7                 suspend_and_yield_paused(token);
8                 if wakeup.is_signaled() {
9                     break;
10                }
11            }
12        });
13
14         mutex_lock_paused(mutex, token);
15     })
16 }
```

Pseudo

Listing 32: Pseudocode for condvar wait implementation.

The actual implementation provided by EVA supports timed variants of waiting and locking, which fail if the wait time exceeds a given amount of time. These are implemented using the mechanisms and techniques described in Section 5.5. for combining different wakeup sources.

5.8. Thread Local Storage

The TLS requires two components to work: a global key allocation system and a per-thread local key-value store. The global key allocation system is implemented using a slot map. Keys are the slot map handles, and each entry pairs the key with its optional destructor function. The local key-value store is located in a special area of the TCB, which contains mutable data only accessible by the thread associated with said TCB. This key-value store is implemented using a simple linked list that holds both the keys and associated values. Existing elements are searched linearly by key in the list, while

new ones are appended at the end. A terminating thread is responsible for running TLS destructors, a procedure that needs to be aware of several edge cases. Other APIs related to the TLS must be callable from a running destructor. New key-value pairs can be added by running destructors. Finally, keys related to a particular value might not exist anymore during the destruction phase, which is considered an error but still well behaved. To guarantee all of the above requirements, we never lock the local key-value store or global key store more than necessary, and these locks are never active when invoking a destructor. The routine running the destructors always checks for new keys after running a destructor and does not assume that the number of key-value pairs will remain the same. Additionally, if a key is not found in the global key allocator, we simply skip that entry in the TLS, leaking the value, to maintain sane behavior.

5.9. Board Support Package

Some BSPs are provided as part of EVA itself for convenience. The current release of EVA only supports two targets, both in the form of an additional crate that provides the implementation of the portability interface for the given target. The first target is the Skyward Compute Unit revision 1, a standalone computational module used by Skyward Experimental Rocketry in their rockets. This board features an ST Microelectronics STM32F767ZI, 32 MB of RAM, a built-in ST-Link-based programmer, four LEDs, and a custom interface exposing all the pins of the microcontroller through an M.2 interface. This platform lends itself perfectly to the initial phase of testing and development of an operating system. The second target is the Linux user space. While not an actual hardware platform, it is still useful for debugging locally the entire operating system. This platform allows us to run EVA entirely and without any modifications inside a single user space Linux process.

5.9.1. Skyward Compute Unit BSP

As said in the previous section, this is a custom board designed by Skyward Experimental Rocketry, shown in Figure 10. Its main purpose is to be a reusable computational module and a compact development platform. The main microcontroller is an STM32F767ZI, which houses several peripherals, most of which are still unsupported as of the current EVA release, alongside a core based on the Cortex-M7 architecture and an FPU coprocessor. Most of the startup code for the Cortex-M7 is already provided by the Cortex-M crate. Despite this, we purposefully decided to avoid it due to its current development status, as the last version officially released is now more than three years old. For simplicity, we also do not enable the external RAM provided on the development board.

We decided to use the PendSV interrupt mechanism provided by the Cortex-M7 core to invoke the scheduler and perform a context switch. This is a specialized software interrupt provided exactly for this purpose. To guarantee that this interrupt can only be executed when the processor is in thread mode, in other words, when it is executing normal application code, we lower its priority to the minimum allowed. This prevents it from preempting any other IRQ handler execution. We can invoke this interrupt any time by setting a special bit in the Interrupt Control and State Register (ICSR), which can also be done inside an IRQ handler. During the dispatch of an IRQ, the hardware first saves a subset of the hardware register on the thread stack, then switches to the handler stack and continues execution. Therefore, our switch context will need to contain only the subset of registers not already saved by the hardware, which includes the thread mode stack pointer. Finally, to perform the context switch, we need to save the current thread context in the currently installed switch context, run the scheduler code, and finally load the newly installed switch context to continue execution from the new thread.

As a global timer, we use the SysTick peripheral provided by the Cortex-M7 architecture. This is a low-resolution clock capable of providing a periodic interrupt. This interrupt is used for two purposes. First, it increments a global time variable used to keep track of time. Second, it issues a PendSV interrupt, which performs an immediate context switch. The period of this timer is set to 10 ms, which is the maximum time that a thread is allowed to run before being preempted. Do note that this timer is always running and does not get reset during a context switch. As a consequence, there is no guarantee on the minimum time a thread is guaranteed to run before a context switch, since a yield could happen right before the SysTick interrupt fires.

The initial startup code is written entirely in assembly, as detailed in Section 4.7., its job is to set up the `.data` and `.bss` regions and to enable the FPU. We do not have to initialize the stack register, as that is already done by the hardware reset circuitry. For this initial stage, we reuse the stack reserved for IRQ handlers as the temporary stack. During stage 0 initialization, we enable hardware fault handlers and initialize the HSI (internal oscillator). During stage 1, we first set up Real-Time Transfer (RTT), which is an extremely fast protocol for communicating with an attached debugger or probe. This protocol will be used as a textual input/output interface. To complete the stage 1 initialization, we register the heap memory region to the EVA allocator, we enable the PendSV interrupt, we set up the SysTick peripheral, and we initialize the scheduler. Finally, during stage 2, we simply invoke the application main, as at this point there are no more initialization steps required.

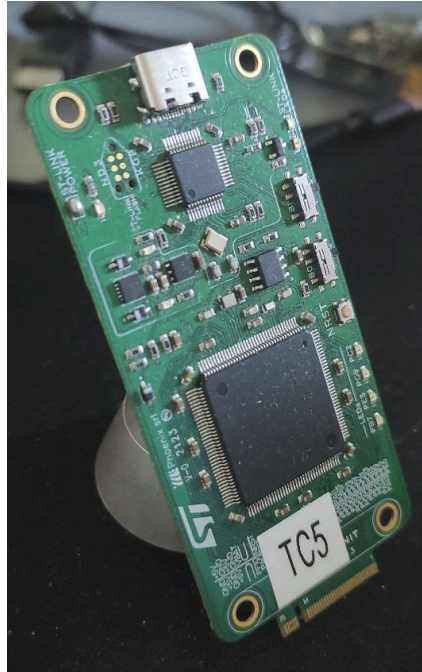


Figure 10: Skyward Experimental Rocketry Compute Unit.

5.9.2. Linux BSP

A common pain point of embedded application development is testability. Hardware is not always available, and emulators are often lacking in features. To solve this issue, we provide a special BSP used to run directly in a user space Linux process. Not only does this make applications easier to run and test, but it also opens up the possibility of using a multitude of existing tools, for example, debuggers and fuzzers. The key concept exploited to achieve this is using Linux signals to emulate hardware interrupts. Currently we use just one signal, taken from the real-time Linux signal set, which is available for application use. Context switching and running the scheduler code are triggered through this signal, both by code directly, through the kill syscall, and periodically by a timer. This last timer is also used as a global timer for timekeeping. To read and modify thread context from within a signal handler, we make use of the `SA_SIGINFO` flag when registering said handler. When this flag is applied, our handler is supplied with extra information on invocation. This information includes the context of the interrupt execution flow, which allows us to easily edit it directly. One last key detail about the Linux BSP is that we effectively cannot rely on any system library, as doing so could conflict with our EVA libraries, for example, in the case of the libc or the Rust std. Therefore, it is necessary to implement our abstraction over raw Linux

syscalls, which is not ideal since some details are not public and purposefully hidden inside the libc.

5.10. Porting the Rust Std Library

The Rust std library requires a minimum set of features to be present to support a new target. These features are a dynamic allocator, an abort system, a way to implement condition variables and mutexes (either natively or through some other primitive such as futex), and some flavor of TLS. Complying with this basic set of requirements allows us to support and compile the Rust std library for our target, but several services are disabled, for example, file system support. We regain some of these lost functionalities by adding support for thread lifetime management (spawning, joining, and detaching) and textual input/output.

Since the Rust compiler and std library are so intertwined, we cannot easily compile the std library separately, as technically the compiler we are using could be incompatible with that specific version of the library. As such, we need to compile the entire compiler and toolchain, even if our patches only span the standard library. Once we get the custom toolchain working, patching the actual standard library is easy. Most, if not all, the platform-specific code resides on a special module called `sys`, which is used internally to implement the application-facing functionalities. This module is subdivided by functionality, each with a platform-specific implementation. Hooking into this system is just a matter of adding a new implementation for our platform and including it via conditional compilation. As said previously, not all of these modules require specific support from our target. Those that do not require it default to either an unsupported stub or a software fallback. Finding which specific functionality is strictly required is also easy. By compiling the Rust std library for our target with no patches applied, it is immediately clear from the compiler error messages which modules need patching. Since our target is not officially supported, even if we compile a custom toolchain, it won't have the Rust std library pre-built for our target. To solve this, we simply use the nightly `build-std` feature of cargo to build it on demand. One last key detail is panic unwind support. This requires a special library providing unwinding functionality to be linked against our executable. This is normally not an issue when using a standard C or C++ compiler toolchain, but since we are using Rust, it is much harder to provide one reliably. The easiest way is to use an unwinding library written in Rust so that it can be included just like any other dependency. One such library exists, but it is rather experimental and can only support Linux in our case. An alternative route would be to use a more common unwinding library, such as LLVM `libunwind`, but we need to manually compile it for our target and link it. None of these routes is particularly challenging, and experiments were done by using the Rust native unwinding library, but more work is required to support this feature properly. For the present release of

EVA, we simply compile the Rust std library with the unwind behavior configured to abort, which immediately issues an abort on panic.

5.11. C as an Application Program Language

Since one of the primary features of EVA is support for multiple languages in kernel space, we decided to include a small demo of how a developer might use C with EVA. This is a very early proof of concept, and in the future there will be multiple ways of including C in a project. In this example we leverage the Rust build system as much as possible to avoid including a different build system. Only a minimum amount of Rust is needed to get the crate to compile. In practice, just an empty panic handler, since we are not using Rust, panics will not happen, and an annotation telling the Rust compiler there is no main, since it is defined on the C side. Then we use a Rust build script to compile and link our C source code. Currently we need to manually specify a compiler and explicitly disable linking with standard libraries, but in the future this process will be streamlined. Finally, we tell the C compiler to include the header defining the EVA APIs. This header is provided by the `eva-abi` crate and is made available to our build script via build script metadata key-values. Despite being an early proof of concept, this is already surprisingly ergonomic and scalable to mid-sized projects, but in the future, integration with existing build systems, such as CMake, will be provided to further improve flexibility.

6 | Experimental Results

We provide some experimental evidence to show how EVA compares to another well-established and mature embedded operating system, Miosix. We choose Miosix specifically as we believe it more closely resembles EVA in philosophy and scope, with the major difference being the choice of language.

6.1. Test Configurations

In this chapter we will be testing two different versions of EVA, the latest unstable codebase and an altered version called “unchecked”. To obtain this modified version of the codebase, we started from the latest unstable commit and removed any code pertaining to runtime checks. This includes removing all asserts, turning `RefCell` into `UnsafeCell`, turning `Option::unwrap` and `Option::expected` into `Option::unwrap_unchecked` and generally removing any instance of checked code. Refer to the codebase for all the changes applied [33]. We use this altered version to show the performance impact caused by runtime checks, which are used to significantly reduce the amount of unsafe code in the codebase.

Both versions of EVA will also be tested in two build configurations each. A standard release build and a custom configuration, which we call “releaseplus”. This last configuration is a more aggressive release build, with slower compile times and, more importantly, fat LTO enabled, which we expect to have a big performance impact, as discussed in Section 4.3.2.. The content of both configuration files is shown in Listing 33 and Listing 34.

Alongside EVA, Miosix is used as a reference. Specifically, version 2.81 with O2 optimization level enabled.

All test configurations are tested on the same STM32F767ZI microcontroller, using only the internal HSI oscillator and configured with a SysClk and CPU speed of 16 MHz.

```
1 [profile.release]
2 opt-level = 3
3 debug = false
4 debug-assertions = false
5 overflow-checks = false
6 lto = false
7 panic = "abort"
8 incremental = false
9 codegen-units = 16
10 # Other options omitted
```

Toml

Listing 33: Configuration file for release.

```
1 [profile.releaseplus]
2 inherits = "release"
3 opt-level = 3
4 lto = "fat"
5 codegen-units = 1
```

Toml

Listing 34: Configuration file for “releaseplus”.

6.2. Benchmark Methodology

All the benchmarks are microbenchmarks, showing the execution time and cycle count of elementary kernel operations. We believe that this best showcases how both operating systems compare and how different optimization levels and checks affect Rust execution times.

Execution times are obtained by toggling one of the GPIOs on and off, which is an extremely fast hardware operation, only requiring four instructions to achieve. We then use an oscilloscope to measure the time between the rising and falling edge of the signal, equal to our execution time. The entire experimental setup is shown in Figure 11.

We focus on benchmarking these four scenarios:

1. Time between a yield and execution of the next thread.
2. Time between a condition variable signal and execution of the thread waiting on that condition variable.
3. Time of an uncontended mutex lock followed by an uncontended mutex unlock.
4. Time between a contended mutex lock and the execution of the next thread.
5. Time between a contended mutex unlock and the execution of the thread waiting on that lock.

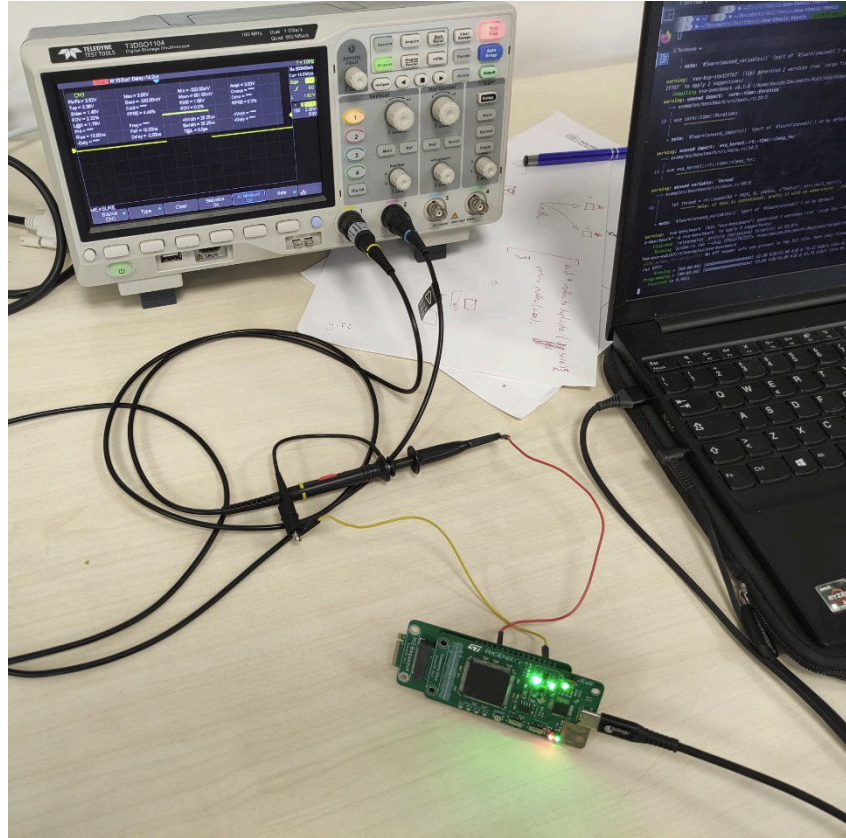


Figure 11: Experimental setup for benchmarks.

6.3. Results

Experimental results are shown in Table 3. Data is reported using cycles as a unit of time to normalize the data with respect to the processor speed. By analyzing the data in terms of speedups and slowdowns, we can gather:

- EVA is slower compared to Miosix 2.81. This was somewhat expected, as our goal was not performance. Yet, the average slow is just -40% , which many might still consider a worthy tradeoff considering the advantages of Rust. We hope that in the future we might be able to regain some lost performance and close the gap between EVA and similar operating systems.
- As speculated in Section 4.3.2., fat LTO has a considerable effect on performance. The speedup gained by just enabling this optimization is $+45\%$, which is practically free as it just requires longer compile times.
- Rust runtime checks definitely play a role in performance, but they are arguably a worthy tradeoff. For release builds, the speed-up obtained by removing checks is $+17\%$, while for “releaseplus” builds it’s $+5\%$. Once again, we see how fat LTO

is critical for obtaining good performance. In the data we see two outliers for the “unchecked” version of the code, as for both release and “releaseplus” builds, we see how bench 3 becomes slower, which seems counterintuitive. This bench tests an uncontended lock and unlock, a rather simple benchmark, which should behave identically in both scenarios as it should not incur any runtime checks. We suspect that by removing certain runtime checks, we inadvertently make some aspects of our program more opaque to the Rust compiler, thus reducing the optimization potential.

Experimental data (cycles)

OS	Bench 1	Bench 2	Bench 3	Bench 4	Bench 5
Miosix 2.81 (-O2)	428	572	84	480	486
EVA (release)	716	1600	259	1008	1472
EVA unchecked (release)	576	1216	281	832	1248
EVA (releaseplus)	492	1139	160	720	1056
EVA unchecked (releaseplus)	454	1032	166	720	944

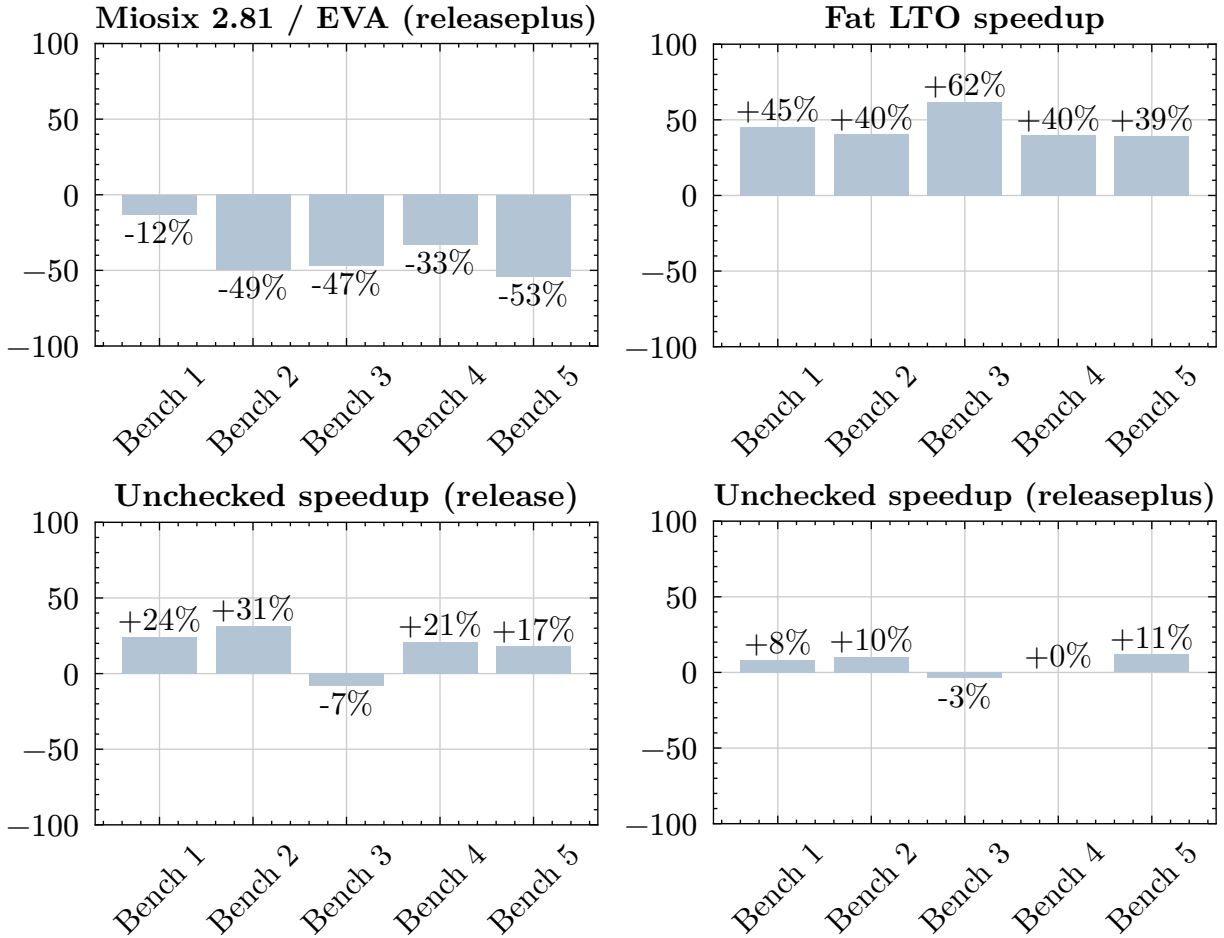


Table 3: Results of experimental benchmarks.

7 | Future Works and Conclusions

7.1. Future works

The current iteration of EVA is still young and far from being production-ready. Throughout this thesis, we already mentioned several of the shortcomings and how they can be resolved in the future. This section is dedicated to aggregating that information and adding additional thoughts.

Numerous data structures in the current EVA release are suboptimal for the task. This was done deliberately, both for simplicity and to avoid premature optimization. Now that EVA is capable of running small applications, we can properly benchmark alternatives and find more optimized solutions.

We lack a dedicated build and configuration system. As our operating systems grow, the limitations of relying solely on Cargo become apparent. To allow for better scaling, we must decouple at least partially from Cargo and provide our custom solution. A custom solution allows us to more easily configure our kernel and components while also providing better support for the different build configurations mentioned in Section 4.3.2..

There is currently no support for unwinding, as mentioned briefly in Section 5.10.. Support for this feature is critical to not only enable full Rust support but also other languages dependent on unwinding, such as C++. Ideally, this should be provided without relying on C or C++. This is currently not possible, as there doesn't exist any library written in Rust providing unwinding functionality implementing the ARM EHABI32 standard. This is the primary reason why Rust embedded targets never support unwinding, unless they rely on an existing operating system, not written in Rust, that bundles a libunwind implementation.

Testability is yet another weak point of EVA, as testing on embedded targets has always been quite challenging. Rust already has a powerful built-in test harness and platform, relying both on unwinding panics and the Rust std. Since we already support the latter, we just need a functioning unwinder, as explained in the previous paragraph. Once

support for the standard Rust test harness is in place, we hope that we will be able to reach good test coverage in our operating system.

Several language-specific libraries (libc, C++ std library, and EVA klib) are planned but not actually implemented. Going forward, to truly support multiple languages, we must also support their respective standard libraries.

The BSPs layer is very lacking. Currently, we just support the bare minimum for a proof of concept. To become “production ready” we must first augment our target support with drivers and support for more targets. This is currently gated behind the custom configuration system, as that will simplify a lot of what is needed to support many targets and target configurations.

Finally, requiring a custom compiler is not ideal. To solve this issue, we would need to upstream all of our custom patches to the Rust compiler and repository. This will have to wait, as upstreaming patches, although not impossible, is very time-consuming and restricts significantly our flexibility going forward.

7.2. Conclusions

We first explored the current state of the art and highlighted what is missing in the current ecosystem: an operating system both capable of supporting new and existing languages and focused on targeting low- to mid-end embedded targets.

We proposed our solution to the problem: EVA. An operating system designed from the outset in Rust, capable of running multiple languages even in kernel space and easily portable to different embedded devices. During the design phase, we also highlighted how support for higher levels APIs is desirable as it reduces friction during application development. Our architecture focuses on a layered design to achieve said goals. Through separation of the different modules, we can easily decouple the portability layer and the kernel interface from the kernel itself. This allows extreme flexibility in how EVA is built and included in a project. Both the portability implementation and kernel can be built separately and then linked with the final application or be included as a normal dependency, and this choice is left to the application developer. To achieve multi-language support, our kernel interface is multi-ABI, providing both a Rust ABI and a C ABI version. This dual-ABI allows us to provide the same interface to Rust native applications, retaining maximum compatibility, but without sacrificing support for languages besides Rust. Our scheduler also follows a layered design, with higher-level primitives relying solely on the primitives below it. Finally, we highlighted how we do not provide any kernel-level primitive for disabling interrupts, which contrasts with similar operating systems. This offers several advantages, but primarily, it allows us to

retain much more responsiveness to IRQs and freedom of implementation when it comes to IRQ management, a very desirable property considering the amount of complexity of modern IRQ managers.

Lastly, we benchmarked EVA. This has shown how EVA is slower when compared to alternatives but still within an acceptable range. Then we have demonstrated how fat LTO is fundamental for performance, something we had speculated in Section 4.3.2.. Simply by enabling this option, we gain roughly a +45% speedup, with the only downsides being slightly longer compilation times. Finally, the data has shown how Rust runtime checks have a negligible impact on performance, causing just a −5% slowdown on fat LTO builds, and are definitely a worthy tradeoff.

Bibliography

- [1] Embedded.com, “Embedded Market Study 2023 (Programming Languages in Embedded Systems).” [Online]. Available: <https://www.embedded.com/wp-content/uploads/2023/05/Embedded-Market-Study-For-Webinar-Recording-April-2023.pdf>
- [2] A. Levy, B. Campbell, B. Ghena, P. Pannuto, P. Dutta, and P. Levis, “The Case for Writing a Kernel in Rust,” in *Proceedings of the 8th Asia-Pacific Workshop on Systems*, in APSys '17. Mumbai, India: Association for Computing Machinery, 2017. doi: 10.1145/3124680.3124717.
- [3] F. Terraneo and D. Cattaneo, “Fluid Kernels: Seamlessly Conquering the Embedded Computing Continuum,” *IEEE Transactions on Computers*, vol. 74, no. 12, pp. 4050–4064, 2025, doi: 10.1109/TC.2025.3605745.
- [4] Y. Peng, H. Tian, J. Xian, S. Zhou, S. Yan, and Y. Zhang, “Framekernel: A Safe and Efficient Kernel Architecture via Rust-based Intra-kernel Privilege Separation,” in *Proceedings of the 15th ACM SIGOPS Asia-Pacific Workshop on Systems*, in APSys '24. Kyoto, Japan: Association for Computing Machinery, 2024, pp. 31–37. doi: 10.1145/3678015.3680492.
- [5] Y. Peng *et al.*, “Asterinas: A Linux ABI-Compatible, Rust-Based Framekernel OS with a Small and Sound TCB.” [Online]. Available: <https://arxiv.org/abs/2506.03876>
- [6] A. Madhavapeddy *et al.*, “Unikernels: library operating systems for the cloud,” *SIGARCH Comput. Archit. News*, vol. 41, no. 1, pp. 461–472, Mar. 2013, doi: 10.1145/2490301.2451167.
- [7] S. Kan, Z. Chen, D. Sanan, S.-W. Lin, and Y. Liu, “An Executable Operational Semantics for Rust with the Formalization of Ownership and Borrowing.” [Online]. Available: <https://arxiv.org/abs/1804.07608>
- [8] R. Jung, J.-H. Jourdan, R. Krebbers, and D. Dreyer, “RustBelt: securing the foundations of the Rust programming language,” *Proc. ACM Program. Lang.*, vol. 2, no. POPL, Dec. 2017, doi: 10.1145/3158154.

- [9] J. V. Stoep, “Rust in Android: move fast and fix things.” [Online]. Available: <https://security.googleblog.com/2025/11/rust-in-android-move-fast-fix-things.html>
- [10] MSRC Team, “Why Rust for Safe Systems Programming.” [Online]. Available: <https://www.microsoft.com/en-us/msrc/blog/2019/07/why-rust-for-safe-systems-programming>
- [11] G. Gross, “White House urges developers to dump C and C++,” *InfoWorld*, Feb. 2024, [Online]. Available: <https://www.infoworld.com/article/3713203/white-house-urges-developers-to-dump-c-and-c.html>
- [12] T. Federico, “Miosix.” [Online]. Available: <https://github.com/fedetft/miosix-kernel>
- [13] A. Leva, M. Maggio, and F. Terraneo, “Performance analysis of operating systems schedulers realised as discrete-time controllers,” in *2012 IEEE International Conference on Control Applications*, 2012, pp. 745–750. doi: 10.1109/CCA.2012.6402429.
- [14] T. Z. P. (Linux Foundation), “Zephyr.” [Online]. Available: <https://github.com/zephyrproject-rtos/zephyr>
- [15] O. Corporation and R. C. Developers, “RTEMS.” [Online]. Available: <https://gitlab.rtems.org/rtems/rtos/rtems>
- [16] A. W. Services, “FreeRTOS.” [Online]. Available: <https://github.com/FreeRTOS/FreeRTOS-Kernel>
- [17] D. Déharbe, S. Galvão, and A. M. Moreira, “Formalizing FreeRTOS: First Steps,” in *Formal Methods: Foundations and Applications*, M. V. M. Oliveira and J. Woodcock, Eds., Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 101–117.
- [18] L. Stefan and H. P. Contributors, “Hermit.” [Online]. Available: <https://github.com/hermit-os/kernel>
- [19] S. Lankes, J. Breitbart, and S. Pickartz, “Exploring Rust for Unikernel Development,” in *Proceedings of the 10th Workshop on Programming Languages and Operating Systems*, in PLOS '19. Huntsville, ON, Canada: Association for Computing Machinery, 2019, pp. 8–15. doi: 10.1145/3365137.3365395.

- [20] C. Sean and X. Contributors, “Xous.” [Online]. Available: <https://github.com/betrusted-io/xous-core>
- [21] V. Valentyn, “Drone OS.” [Online]. Available: <https://github.com/drone-os/drone-core>
- [22] L. Amit and T. Community, “Tock.” [Online]. Available: <https://github.com/tock/tock>
- [23] L. Schuermann, B. Campbell, B. Ghena, P. Levis, A. Levy, and P. Pannuto, “Tock: From Research To Securing 10 Million Computers,” in *Proceedings of the ACM SIGOPS 31st Symposium on Operating Systems Principles*, in SOSP '25. Lotte Hotel World, Seoul, Republic of Korea: Association for Computing Machinery, 2025, pp. 36–49. doi: 10.1145/3731569.3764828.
- [24] R. Community, “RIOT OS.” [Online]. Available: <https://github.com/RIOT-OS/RIOT>
- [25] A. S. Foundation, “NuttX.” [Online]. Available: <https://github.com/apache/nuttx>
- [26] Eclipse, “ThreadX.” [Online]. Available: <https://github.com/eclipse-threadx/threadx/>
- [27] The Rust Project Developers, “Rust Standard Library Documentation.” [Online]. Available: <https://doc.rust-lang.org/std/>
- [28] J. Yanovski, H.-H. Dang, R. Jung, and D. Dreyer, “GhostCell: separating permissions from data in Rust,” *Proc. ACM Program. Lang.*, vol. 5, no. ICFP, Aug. 2021, doi: 10.1145/3473597.
- [29] N. Dario and R. Embedded, “critical-section.” [Online]. Available: <https://github.com/rust-embedded/critical-section>
- [30] R. K. Treiber, “Systems Programming: Coping with Parallelism,” Yorktown Heights, NY, USA, technical report RJ5118, 1986.
- [31] O. Philipp and R. OSDEV, “linked_list_allocator.” [Online]. Available: <https://github.com/rust-osdev/linked-list-allocator>
- [32] P. Wilson, M. Johnstone, M. Neely, and D. Boles, “Dynamic Storage Allocation: A Survey and Critical Review,” In: *International Workshop on Memory Management*, vol. 986, p. , 1999, doi: 10.1007/3-540-60368-9_19.
- [33] D. Mor, “EVA.” [Online]. Available: <https://github.com/Tazdevil971/eva>

List of Acronyms

- ABI** Application Binary Interface. 22, 23, 24, 25, 26, 28, 71
- API** Application Program Interface. 2, 6, 7, 14, 15, 17, 22, 23, 24, 28, 30, 33, 36, 37, 38, 44, 50, 51, 53, 54, 60, 64, 71
- BSP** Board Support Package. 22, 25, 26, 27, 28, 37, 38, 60, 62, 71
- CAS** Compare-And-Swap. 32, 45, 57
- FIFO** First In First Out. 35
- FPU** Floating Point Unit. 38, 60, 61
- HAL** Hardware Abstraction Layer. 25
- ICSR** Interrupt Control and State Register. 61
- IPC** Inter-Process Communication. 6
- IRQ** Interrupt request. 32, 35, 36, 45, 61, 72
- LTO** Link Time Optimization. 27, 66, 68, 72
- MMU** Memory Management Unit. 4, 6, 7
- MPU** Memory Protection Unit. 11
- OS** Operating System. 5, 6, 7
- RTOS** Real-time Operating System. 5, 24
- RTT** Real-Time Transfer. 61
- TCB** Thread Control Block. 29, 30, 59
- TLS** Thread Local Storage. 30, 37, 38, 42, 47, 59, 60, 63
- WCET** Worst-Case Execution Time. 5, 45

ZST Zero Sized Type. 18

List of Listings

Listing 1	Example of subtle integer truncation in C.	8
Listing 2	Example of subtle integer truncation in C. (cont.d)	9
Listing 3	Example of ambiguity in C++.	9
Listing 4	Example of move operation in Rust.	10
Listing 5	Example of copy operation in Rust.	10
Listing 6	Example of error management in Rust.	10
Listing 7	Extract from <code>UncheckedRef</code> implementation.	15
Listing 8	Example of <code>UncheckedRef</code> usage.	16
Listing 9	Example of <code>UncheckedRef</code> usage with improved API.	16
Listing 10	Implementation of the token pattern.	19
Listing 11	Implementation of a token cell.	20
Listing 12	Example usage of the token pattern.	20
Listing 13	Example definition used for crate decoupling.	27
Listing 14	Example import used for crate decoupling.	27
Listing 15	Example usage of a scheduler pause.	33
Listing 16	Example usage of a suspend.	34
Listing 17	Example of “zero allocation linked lists”.	44
Listing 18	Pseudocode for “take” and “push” operations.	46
Listing 19	Pseudocode for scheduler try pause implementation.	48
Listing 20	Pseudocode for scheduler unpause implementation.	48
Listing 21	Pseudocode for run scheduler implementation.	49
Listing 22	Pseudocode for wait and signal operations.	53
Listing 23	Pseudocode for inner and outer waits.	54
Listing 24	Pseudocode for improved wait.	55
Listing 25	Example usage of improved wait.	55
Listing 26	Example of combining multiple waits (OR).	56
Listing 27	Example of combining multiple waits (AND).	56
Listing 28	Pseudocode for mutex unlock implementation.	57
Listing 29	Pseudocode for mutex lock implementation.	58
Listing 30	Pseudocode for condvar notify one implementation.	58

Listing 31 Pseudocode for condvar notify all implementation.	59
Listing 32 Pseudocode for condvar wait implementation.	59
Listing 33 Configuration file for release.	67
Listing 34 Configuration file for “releaseplus”.	67

List of Figures

Figure 1	High level structure of EVA.	23
Figure 2	Different kinds of usage of EVA as a dependencies.	28
Figure 3	Thread state machine.	29
Figure 4	Thread layout.	29
Figure 5	Layers of scheduling primitives.	31
Figure 6	Visualization of possible linked list implementations.	42
Figure 7	Visualization of node aliasing.	43
Figure 8	Preemption points during scheduler pause.	50
Figure 9	Scheduler queues.	52
Figure 10	Skyward Experimental Rocketry Compute Unit.	62
Figure 11	Experimental setup for benchmarks.	67

List of Tables

Table 1 Attributes for low- to mid-end embedded devices. 4

Table 2 Comparison of existing OSs. 13

Table 3 Results of experimental benchmarks. 69

Ringraziamenti

E come se il tempo non fosse trascorso, sono arrivato alla fine anche di questo percorso. In questi ultimi giorni di chiusura ho avuto modo di riflettere su tante cose. Riflettere su tutto quello che ho passato e quello che mi aspetta.

Ma una cosa mi ha commosso, sono questi giorni che mi ricordano che sono circondato da persone incredibili. Persone che mi hanno reso chi sono, arricchendomi ogni giorno, sopportandomi e amandomi.

Persone come Skyward Experimental Rocketry, che è stata la mia casa per tutti questi anni. Persone che mi hanno insegnato tutto quello che potevano, e chi mi hanno permesso di raggiungere traguardi incredibili.

Persone come i miei compagni di università, con cui ho condiviso la noia mortale delle lezioni, i buchi lunghi ore e i progetti fatti all'ultimo e consegnati in ritardo.

Persone come l'HEAPLab, che sin da subito mi hanno accolto come uno di loro. Spero che questo luogo diventerà presto il proseguo del mio viaggio.

Persone come il mio relatore, Federico Terraneo, che nonostante mi sia presentato con un'idea folle, ha deciso da darmi una chance e tutta la sua fiducia.

Persone come il mio migliore amico, Nicolas, che mi ha sempre accettato per chi fossi, in tutte le mie sfaccettature.

E infine la mia famiglia, che mi ha nutrito, supportato e spronato in tutto, alimentando quella fiamma che mi ha sempre spinto oltre.

Grazie a tutti voi, nessuno nasce, vive e muore da solo, ma ognuno di noi è la somma di ciò che ci circonda, e io ho la grande fortuna di avere persone incredibili attorno.

Quest'ultimo capitolo lo dedico a voi, scritto con le lacrime agli occhi, provenienti da un'emozione incredibile.

Grazie ancora.