Investigating The Future of Operating Systems: n nalysis of Systems Programming in Rust

Eugene Mak

Manning College of Information and Computer Sciences, University of Massachusetts mherst

bstract

Rust is a modern and quickly growing language, attracting the attention of many researchers and companies as a possible alternative to the current long-established languages, C and C++, for developing systems. To explore why Rust has grown in popularity and usage, I investigate core aspects of Rust, detailing how they work and exploring how they are advantageous for systems. I also investigate existing implementations of operating systems in Rust to explore how these aspects have presented benefits as well as challenges during development of these systems. Finally, I detail the implementation of a rudimentary file system in Rust to explore how Rust behaves when developing a component of an operating system, as well as detail the experience of learning Rust with no prior experience.

Introduction

For the past few decades, C has been the dominant language for systems programming. It is a powerful and advantageous language for operating systems, particularly in regards to performance: it allows for fine-grained control over resource management and features little overhead. However, these advantages come at a significant cost. Having little overhead comes at the expense of robust safety features that would be helpful for developing secure and stable systems. Over time, the cost of using an unsafe language for systems has grown, and developing bug-free systems with the language has proven challenging for even experienced developers. Thus, systems programmers investigate potential alternatives to C for developing their applications, searching for a language that has the best of both worlds: a language with safety features at its core while also having performance comparable to C. Rust could prove to be that very language, with a myriad of other capabilities that are particularly beneficial for systems.

Rust is a modern programming language, sponsored by Mozilla and first released in 2015. It boasts advantages in performance, memory management, type safety, and concurrency — all important aspects for systems programming. Over the past years, Rust has gained significant traction. Large tech companies such as Microsoft, Google, and mazon have already adopted the language for the development of new products, and certain industry experts have gone so far as to call C and C++ deprecated, and that they should be replaced with Rust [1]. Even within just this past year, Rust has been added to the Linux kernel, and Google has announced a new entirely Rust-based operating system, KataOS, for embedded systems [2]. With Rust becoming increasingly popular among developers, many have begun to wonder if Rust will truly dethrone C for systems programming — and as one can see due to C's lengthy dominance in the space since 1972, it would be no small feat.

Today, Rust is one of the most promising languages to replace C for systems programming. Thus, understanding how this came to be and whether its success will grow is intrinsically tied with the future of the very systems we use each day. Rust could help developers create more efficient and stable systems, leading to significant tangible benefits to everyday users. Take, for example, memory management. In 2019, Microsoft revealed that around 70 percent of all the vulnerabilities in Microsoft products addressed through a security update each year are memory safety issues [1]. Similarly, in 2020, Google software engineers stated that roughly 70 percent of all serious security bugs in the Chrome codebase are memory management and safety bugs [1]. By design, built-in aspects of Rust protect against many kinds of common memory management bugs such as use-after-free. While current mature systems like Chrome are likely too complex to meaningfully gain benefit in a reasonable span of time from switching to Rust, the adoption of Rust for future applications could prevent a significant amount of bugs and vulnerabilities, leading to more secure systems and freeing development time for other concerns.

The key literature will help explain, in detail, how Rust protects against many pitfalls that developers face when developing systems. The literature will address many of the benefits of Rust by analyzing certain aspects such as its compiler, the ownership type system, or concurrency behavior. The literature includes a straightforward analysis of how these aspects of Rust improve systems development, and development of a robust system to practically demonstrate how Rust can improve as well as possibly hinder the development of the artifact and the end result. By doing so, we can draw conclusions for how the authors feel about Rust's viability for systems programming.

Literature Review

Rust Overview

One of the central advantages of Rust that the literature focuses on is its safety. Jung et al. [1] goes into great detail on Rust's safety features without sacrificing control, or in other words, access to byte-level representation of data and the ability to use low-level programming techniques. They discuss how, in programming, safety and control are two important desirable aspects that are often at odds with each other. For instance, programmers want strong type systems, automatic memory management, and data encapsulation, but to account for performance and resource constraints they also need to optimize time and space usage using low-level programming techniques. Languages like Java exist with strong safety, but at the expense of control, leaving C or C++ as the only option for systems which allow programmers to precisely control resource management [3]. However, with Rust, many C++ derived low-level programming techniques and PIs are supported, while also featuring a strong static type system which enforces safe usage of low-level functions. It is claimed that Rust is, in fact, the first industry-supported language to overcome the trade-off between safety and control.

One of the main ways Rust achieves this is the ownership type system (also known as affine or structural type system). Ownership enforces safe patterns, restricting which references to an object may be used to mutate it. Put simply, memory in Rust always has a unique owner; therefore, when ownership of a piece of memory changes, the original owner no longer has access to this piece of memory. dditionally, when a variable holding owned memory goes out of scope, the compiler automatically deallocates this piece of memory by transparently inserting destructor calls, as we can be sure this memory is no longer needed. Consequently, Rust programmers rarely need worry about memory management, as it is largely automatic and is static (determined at compile time). This approach also is effective beyond memory management: for instance, it is applicable to resources like file descriptors, sockets, and locks, so the programmer would not need to manually close a file or release a lock. Thus, the system is helpful for the user by managing resources automatically and ensuring these resources do not get used beyond destruction, while also being optimized for performance by detecting these errors at compile-time.

Figure 1: *n example of an ownership move, from Klabnik and Nichols [4]. Since `Strings` are a mutable type, they do not implement the `opy` trait, so the ownership of the strings are moved to the `greet` function—thus, `m1` and `m2` have become undefined after calling `greet` and the code will not successfully compile.*

```
fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world");
    greet(m1, m2);
    let s = format!("{}␣{}", m1, m2); // Error: m1 and m2 are moved
}

fn greet(g1: String, g2: String) {
    println!("{}␣{}!", g1, g2);
}
```

nother core feature of Rust's design are the borrowing and lifetime mechanisms. s Jung et al. [1] describes, this system is closely tied with ownership. To illustrate: if one wants to provide data to a function temporarily, and get data back when the function returns, ownership prevents accessing the data since ownership has been transferred and consumed by the function. Thus, in order to achieve what we want in our example, we use borrowing. In our example, we would want our function to borrow ownership from the caller – to do this, we pass a mutable reference, rather than a value, to the function. This borrowed ownership is returned according to its lifetime, which dictates how long a reference can be borrowed. The Rust compiler infers lifetimes and ensures that a reference is used by a caller only according to when its lifetime is active. In summary: when something is passed by value, Rust interprets this as an ownership transfer; if something is passed by reference, Rust intercepts this as borrowing for a certain lifetime [1]. Thus, the mechanisms of borrowing and lifetimes allow for explicitly avoiding ownership transfers when needed, while also preserving all the advantages of the ownership system.

Figure 2: *resolved version of the code from Figure 1, from Klabnik and Nichols [4]. We could have previously resolved this issue by simply returning the values in `greet`, but we can instead use borrowing to limit verbosity. Here, the strings of `m1` and `m2` temporarily transfer ownership to `greet`, which gets automatically returned once the borrow lifetime, in this case being the scope of the function, is over.*

```
fn main() {
    let m1 = String::from("Hello");
    let m2 = String::from("world");
    greet(&m1, &m2); // note the ampersands
    let s = format!("{}␣{}", m1, m2);
}

fn greet(g1: &String, g2: &String) { // note the ampersands
    println!("{}␣{}!", g1, g2);
}
```

It is important to note that the ownership system is not without its challenges. s Balasubramanian et al. [3] point out, ownership prevents pointer aliasing, thus making certain data structures like doubly-linked lists impossible. Rust has two features that provide a solution: first, Rust features an unsafe subset that is not restricted to a single owner, and is also used to implement Rust's standard library. Secondly, using `unsafe`, Rust supports read-only aliasing, known as shared references [1], by wrapping the object in a reference-counted type such as `Rc` or `rc` [1]. On the other hand, to achieve write aliasing, objects are wrapped with a Mutex type – allowing single ownership to be enforced dynamically. Mutex utilizes a lock, which ensures mutual exclusion [3]. This enforces that the uniqueness property of mutability is maintained and no callers will have a mutable reference at the same time [1].

Levy et al. [5], mentions that ownership can be a challenge when using Rust to implement a kernel. Specifically, with ownership there cannot be two mutable references to the same memory. While this is important to prevent programs from circumventing the type system [2], kernels need to mutate a shared data structure. Fortunately, doing this does not require changes to Rust, but in this situation it requires trusting two types of unsafe code. One type is Rust's mechanism and libraries, which use unsafe code but provide a safe interface. Specifically, one of the important abstractions used for a kernel is `Cell`, which allows multiple references to a mutable object but uses a limited PI that copies data out to access it, rather than accessing in place. The second type of unsafe code is kernel code, written by kernel developers. Here, an important piece of unsafe kernel code is `TakeCell`, which allows the kernel to share complex structures with no runtime overhead, as it does not copy values out, but pass code in using closures. These key literature show that ownership provides great benefit, and while it presents challenges in certain applications, Rust provides tools to circumvent the ownership system, which provide unsafe functionality only as needed. s briefly mentioned, Rust does in fact give the option to use unsafe features. The previously mentioned `Rc`, `rc`, and `Mutex` types cannot be implemented in Rust by default, as the compiler would reject them for potentially violating Rust's aliasing rules or accessing uninitialized memory. However, these types verifiably do not do so, and thus their implementation uses explicitly defined unsafe code. PIs like these which are implemented with unsafe code are expected to be encapsulated with safe PIs, so that their usage is still safe. Notably, strauskas et al. [6] finds that, after conducting a large-scale study on current practices of Rust programmers, unsafe code is used more extensively than expected, but many of these unsafe functions are used to interoperate with C code. Otherwise, unsafe code is often small, self-contained, and not publicly-accessible. Thus, these

[1] There is an important distinction between `Rc` and `rc`: namely, `rc` is an atomically reference-counted pointer, which works with unstructured parallelism (e.g. where threads are spawned and run independently from the parent thread). The `rc<T>` pointer to T keeps track of how many such pointers exist, and deallocates T when the last pointer is destroyed. `Rc` uses non-atomic operations, thus is potentially faster but is not thread safe [1].

[2] One way circumvention could happen is given as an example with Rust's enums vs C's unions (functionally equivalent). If there was an enum and union which represented either a number or pointer, with the union, having multiple mutable references could lead to a situation where one reference is to a pointer, but the other reference can change it to a number, it can create an arbitrary pointer, allowing the code to potentially access any memory. By enforcing unique ownership, Rust prevents this scenario [1].

unsafe options in Rust are still important, and valuable to programmers today.

Thanks to its safety features, one of the main advantages that Rust claims is "fearless concurrency." This is particularly significant for operating systems, as concurrency is essential for multitasking, but developing concurrent programs can be challenging. When multiple programs mutate shared data, they risk reading or writing memory at the same time, causing unpredictable erroneous behavior. These are called data races, and it can be particularly challenging to prevent and debug concurrency-related errors. While locks can be used to avoid data races, developers may improperly implement locks, and locks themselves can cause scalability and performance issues [7].

Many of the previously mentioned safety behaviors such as ownership, borrowing, and lifetimes are useful for developing a concurrent application in Rust. nother feature useful for concurrency that was not previously mentioned is `Send` and `Sync`. These are traits of types that signify how the type can be treated by other threads. Specifically, types with the Send trait can be safely sent to another thread, while types with Sync can be safely accessed from another thread [7][1]. These traits are not only useful for safe multithreading, but they also enable options for more efficient non-thread safe types in the right scenario. For example, `rc` is an example of a type with `Send`, while `Rc` is not `Send`. However, `Rc` is more efficient, and thus with `Send`, the programmer has the option to use both thread safe, atomic data structures like `rcc` and non-thread safe structures like `Rc` in the same language in the right scenarios, providing the programmer with more flexibility [1].

In order to demonstrate Rust's safe concurrency capabilities, Saligrama, Shen, and Gjengset [7] implement hashmaps with increasingly complex concurrency support in order to analyze how Rust protects developers from errors, and whether it is necessary to temporarily escape from Rust's safety using unsafe. In their analysis, they lay out various ways Rust safety features benefited development:

- Ownership and mutability ensure that memory is never used after freeing, that values are not modified when read or modified elsewhere, and values defined as immutable are not modified. They also note that these compiler rules can be difficult to work with at first but resulted in better written code.
- significant design change compared to C and C++, locks wrap the type they are protecting. Thus, threads are forced to hold a lock when accessing the type at any point.
- When unsafe code such as pointers are needed, Rust requires the use of the unsafe keyword, which narrows down where concurrency bugs may appear.

Their lock-free concurrent hashmap exhibited a significant speedup for an increasing number of threads, particularly when compared to a standard implementation with locks. This resulting performance improvement is notable, as their implementation of a lock-free hashmap in Rust shows that Rust can enable new strategies for developing applications that can result in performance improvements.

This experience is shared by Cantrill [8], who explains that certain strategies in C such as usage of a double linked list should not be simply ported in Rust but rather re-implemented in a way that conveys ownership clearly. Even still, Cantrill explains that when implementing a doubly-linked list in Rust with stopgap solutions, his Rust implementation outperformed his C implementation. This is because Rust implements B trees, which boast superior performance to VL trees in C. He notes that this is a particular advantage of Rust, since VL trees are used in C as a compromise: B trees cannot be cleanly implemented in C, but due to the composability of Rust, it can be built into the standard library of Rust.

Today, many have already implemented operating systems or pieces of systems in Rust [5][8]. Through these examples, we can observe how the benefits of Rust have already influenced systems. For example, though Cantrill [8] expresses doubt on the usefulness of implementing a kernel in Rust, Levy et al. [5], find that with their own implementation, Rust's linear type system allows for safety mechanisms in operating system kernels while requiring only minimal unsafe abstractions. dditionally, certain operating systems such as Redox have been purposely developed from the ground up in Rust to take advantage of its security and safety advantages, and even new teaching operating systems in Rust such as IntermezzOS and BlogOS have been created to demonstrate Rust's capabilities [8]. Cantrill [8] has even advocated for a hybrid approach by interoperating Rust and C, potentially by retaining a C or assembly-based kernel while developing Rust in-kernel components, such as drivers or file systems. nother hybrid approach could be done with Rust OS components, such as with device managers or fault management. Certainly, a hybrid approach may be a significant way Rust can improve current systems: a notable recent example would be the development of ndroid 13 by Google, which describes how memory safety vulnerabilities have recently dropped significantly in ndroid. In fact, Google states that 2022 has been the first year memory safety vulnerabilities are not the majority of ndroid's vulnerabilities. This has been strongly correlated with the increased usage of memory safe languages such as Rust, and in particular they have discovered no memory safety vulnerabilities in ndroid's Rust code, even with the increased volume of new Rust code. Even with necessary usage of unsafe Rust, they have been able to use it sparingly and encapsulated, as intended [9].

Overall, examining Rust's safety features, performance, and developmental tools can explain much of the increased attention Rust has garnered for systems programming. dditionally, by exploring real world usages of Rust, we will be able to show how these features have directly correlated in creation of more efficient, safe, and secure systems.

Rust Implementations

In order to better understand how Rust can be used to improve systems programming, we now will take a look at a few real world operating systems implemented in Rust to go over the benefits as well as the challenges when using Rust for operating systems in real life.

Reenix To start, we will take a look at Reenix [10]. Published in 2015, Reenix is a unix-like operating system developed in Rust for an undergraduate thesis, with the hopes of determining where Rust may help or hinder development as well as compare the performance of the Rust kernel implementation with the C implementation. The operating system is based off of the Weenix operating system, which is used for Brown University's operating system course.

The beginning of Reenix's development quickly ran into issues, starting with the bootloader: the Weenix operating system that Reenix would be based off of used a custom 16-bit assembly bootloader, which did not support loading any kernel images larger than 4 megabytes. This limit is perfectly manageable with C and gcc, but Rust and rustc easily surpassed this limit and required rewriting the boot system. This issue is an example of how strategies in C cannot be easily translated in Rust, and specifically it shows how the differences with Rust compiler optimizations compared to C is important to consider.

dditionally, for the scope of the Reenix project, a memory allocator was not planned to be implemented in Rust, instead opting to utilize the Weenix allocator. However, the Weenix allocators allocate fixed size data structures from contiguous slabs of memory, while at the time Rust was only suitable for malloc style memory allocators. s a workaround, Reenix searches through all known allocators and selects the best one available, which requires initializing a full list of allocators during the startup process. It also meant that Reenix would not be able to always allocate perfectly sized memory chunks to common types. This issue shows that Rust was built with certain assumptions in mind which may not be easily circumvented, although now Rust seems to have added support for custom allocators in 1.9.0.

core part of Reenix is the implementation of the process control system, directly adapted from Weenix's PROCS. Reenix structures its processes by holding information about its child processes and memory map. Reenix aims to control its processes much like Unix does, but has an added challenge of following the ownership system, ensuring that processes own the process structures of its children. Thus, each process needs to keep track of its parent, but for usability a process must be able to be accessed with an arbitrary process ID rather than passing the current thread variable around. Thus, process structures are handled using reference counted (Rc) pointers and weak references, to maintain that the owner of a process structure is its parent while allowing access to the process. Rust's scope-based destructors are advantageous here, simplifying code and safely cleaning up errors. For example, if a new process creation fails, an error can simply be thrown and all temporary values are destroyed without needing to take cleanup actions.

Reenix is a single core system, so synchronizing processes is simply based around wait queues, where a process can wait on a wait queue and sleep until another thread signals the queue. Similar to processes, implementing synchronization requires answering questions related to ownership regarding who should own the queues. Here, queues do not have a clearly defined owner, and Rust's lifetime system creates problems since it is difficult to define a lifetime consistent for all threads. Lifetimes in Rust assume that threads can only see things that will be around forever or only exist in the current call stack, and since threads have separate lifetimes from others it is impossible to have references for wait queues that are safe. Reenix thus simply ignores some of Rust's safety checks, holding queued threads as pointers and casting them as threads when removed from the queue. Light notes that weak references could also be a viable solution.

For scheduling, Reenix used first-in-first-out (FIFO), which ran into similar issues as wait queues regarding ownership but can be solved in a similar manner. However, the Rust compiler would optimize away checks the scheduler would perform when there is no thread to run immediately, since it believes it has the only mutable reference to the list when in fact the list can be modified by interrupt contexts and must be checked. This behavior can occur in C as well with optimizations, but can simply be avoided by marking as volatile to stop optimization. However in Rust, all reads of the variable must be done by a special marker function in order to read the variable.

Device drivers were also implemented in Reenix, again following Weenix's implementation. Reenix takes advantage of the trait system to encapsulate drivers basic functions and easily implement Device traits multiple times with different type arguments. While most driver implementations were straightforward and sometimes even easier to implement with Rust, there was a notable difficulty when implementing the T disk driver: this driver requires a data structure called the Physical Region Descriptor Table (PRD) which must be strictly aligned on a 32 byte boundary, which is simple to do in C. However, Rust does not have a good way to force alignment of data, and in the end Reenix over allocates and manually aligns data when using the PRD table, which is a problematic solution.

Reenix also implements a portion of a virtual file system, but ran into difficulties when implementing the testing in-memory file system, RamFS. The C implementation implements files and directories as an array of bytes and casts them when needed, but Rust does not allow simple interfacing with raw untyped storage. This issue can be handled with a binary-serialization library, but instead Reenix circumvents this issue by changing the design; for instance, directories are simply maps from file-name to nodes. Overall, Rust proved to have many advantages as well as challenges for developing Reenix. Its benefits are significant: Rust is a high level language, providing significant abstractions and conveniences. Rust automatically destroys objects based on scope and also supports custom destructors. One of the most significant features of Rust, its type, lifetime, and borrow checking features, also provides benefits for operating systems. Its type safety features use wrapper types, significantly the Result type, which makes it impossible to access the result of a function without being mindful of possible errors which must be handled in some way. The lifetime system requires that all pointer references are provably safe and cannot be kept once the object pointed to is freed, avoiding most if not all use after free and uninitialized data bugs. The borrow checker also prevents concurrent modification of data structures. mong other things, Rust also supports built-in stack overflow detection, and has a powerful macro and plugin system.

Yet, Rust's challenges can often be encountered in unexpected ways when attempting to use strategies commonly found in other languages, particularly C. For instance, compiling in Rust can often be done using cargo, but is not suitable for the scope of Reenix, thus requiring a makefile. However, creation of the makefile proved to be more complicated due to the complexity of dependencies between crates compared to the simpler C dependencies. t the time of implementation, Rust also does not have inheritance among structures, resulting in types that require access to different functions but hold the same data. Rust also has difficulty in handling statically allocated data, disallowing structures to be allocated statically since there is no way to ensure destructors are run when the program exists. It is also impossible to tell the compiler to ignore this issue. The most significant problem encountered during Reenix development was heap-memory allocation, where Rust aborts if allocation falls and hides the fact that allocations may fail. While this problem may be a rare occurrence overall, this issue is most significant for kernel code, as allocation failure can be a common occurrence and is supposed to be easily handled. No elegant solution exists for this issue and may theoretically require patching the Rust compiler.

In the end, Light [10] acknowledges that Rust provides an overall benefit for developing an operating system, and believes that Rust could replace C in this field as long as its issues are resolved, specifically ones regarding memory allocation.

Hephaestus Hephaestus is a Rust based runtime meant to support Dios, a distributed operating system for data centers [11]. By taking advantage of Rust, Hephaestus aims to improve performance, simplify abstractions, and improve memory safety. Before Hephaestus, the Dios system call PI only supported low level languages. During the development of Hephaestus, Rust was still in early development and thus portions of development proved challenging due to instability and undiscovered bugs.

For setting up compatibility with Dios and Rust, Rust's standard library of crates had to be adapted, as these libraries are important for the functionality of the OS. Certain libraries required boilerplate code, but was not particularly challenging beyond that. Rust also allows conditional compilation, which allows the Rust compiler to target different architectures and operating systems.

Rust concurrency model changed during development of Hephaestus, ultimately changing for the worse with regards to compatibility with Dios as the new model assumes shared memory is available and pointers can be passed between threads, but Dios' execution units called tasks (similar to threads and processes) do not meet this requirement. Thus, a Task PI for Dios in Rust's standard library had to be developed independently.

Rust's ownership model serves as a good model for certain parts of Hephaestus, but for resource management, the ownership model is not applicable to managing references since objects are shared rather than having a single strictly defined owner. Counting references with locking is costly in the kernel, so instead atomic counters are used.

Overall, Hephaestus was a successful project, able to multitask and communicate Rust programs with good performance: CPU bound programs have equivalent performance to Rust running on Linux. Correctness was verified with various tests, and strict compile analysis eliminated common errors found in C. The code was reported to be shorter and cleaner, and thus easier to write and maintain.

Tock Tock is a Rust-based embedded operating system for microcontrollers [12]. Tock was intentionally developed in Rust for several reasons: safety is particularly important for kernel development. Language safety makes bugs such as buffer and integer overflows, which represent a significant portion of kernel bugs, impossible. However, safety becomes especially important for embedded systems. Embedded systems use processors and microcontrollers that lack hardware protections, embedded systems are less tolerant of crashes, and debugging embedded kernels is difficult as they lack logging features and require a physical debugger. ll these factors make language safety particularly beneficial, and Rust appears to be a good candidate for providing language safety as it promises safety as well as comparable performance to C and C++. s a result, the goal is to make the embedded system more reliable, with less need to debug at runtime.

However, during Tock's development, hurdles arose due to Rust's type systems that had to be worked around. Specifically, the ownership system proved to be problematic in several ways: first, ownership is unnecessary in contexts where resources are always present, such as hardware resources and device drivers. These resources are never freed, so ownership does not need to manage them.

Second, and more importantly, ownership prevents resource sharing between closures and other kernel code, due to unnecessary thread safety concerns. Rust memory safety presumes threads, which works well for systems such as threaded network servers, but it does not work well with non-thread based concurrency. Tock operates with single-threaded event-driven concurrency, and is unable to implement multithreading due to the cost of performance incurred. Thus, Tock can have scenarios where a reference to a piece of memory should be maintained at the same time, but ownership prevents this scenario from happening. However, due to Tock's main scheduler loop being single threaded, race conditions prevent two functions from ever accessing a piece of memory concurrently. The developers of Tock propose a feature called execution contexts, which would allow programs to mutably borrow multiple times as long as these borrows aren't shared between threads. This check would be performed at compile time, thus ensuring performance while allowing capabilities that are necessary for developing an embedded system.

Implementation

To gain first-hand experience of Rust and demonstrate how Rust can provide similar capabilities as C and C++ for systems, I implemented the File System project from CS 377 in Rust. In addition, I will be detailing my experience with using Rust, including its learning curve, from the perspective of an undergraduate student with no prior experience in Rust. To enable easy comparison, the file system implementation closely adheres to the original project specifications. Thus, the file system supports 128 KB, with a single root directory, and a max of 16 files. The max size of a single file is 8 KB, corresponding to 8 data blocks with each block being 1 KB. Each file must also have a unique name, no more than 8 characters. The first KB of the disk is the superblock, with the first 128 bytes of the superblock being a free block list signifying which blocks are available with a 0, or used by a file with a 1. The rest of the superblock consists of 16 index nodes, one for each possible file, containing metadata of the file such as its name, size, and which data blocks the file is using. Finally, the other 127 KB of the disk are data blocks that can be read and written to. The file system should have capability for creating, listing, and deleting files. The file system should also be able to read and write to specific blocks of a file, and should be able to close its own disk as well.

Results and Discussion

The implementation matches the capability of the original project with Rust, with the added benefit of using no unsafe code. While the overall structure of the file system is closely aligned with the original, some implementations of various components of the project ended up being significantly different.

The first noticeable difference is setting up the file system class. Rust does not have traditional classes as known in other languages, by design; instead, Rust takes advantage of the existence of structs and utilizes a feature called `impl` to define functionality for certain types. Thus, with `struct` and `impl` together, we can simulate the functionality of a class.

One hurdle encountered was setting up the strings for the file names. With the file system, we require the file name to be a strictly 8 character length array, in order to have a consistent index node struct size. The consistent size allows us to accurately read and write from the data buffer, knowing exactly how many bytes to read and write to the disk in order to retrieve and set a new index node. The size of the index nodes also dictates the overall structure of the disk; without a consistent size, the locations of certain sections of the disk may be indeterminate. While a fixed string max size is simple enough to do in C with a simple array of chars, in Rust, string types are dynamically allocated: specifically, the String type is a dynamic heap string type, similar to Vec, which can be used particularly if we want to modify the String data. Otherwise, `str` is commonly used, which is an immutable sequence of UTF-8 bytes, with a dynamic length. To get around this issue, we instead define an array of u8 unsigned integers, with a fixed length of 8. Using this array allows us to have a consistent index node size while also allowing variable length strings by converting this array to an `str`. This implementation essentially simulates the array of characters, as in C.

Due to the nature of Rust automatically freeing the file, the close disk function is likely not needed for this implementation. However, I implement the same functionality in order to replicate all functions, and Rust allows for manual freeing if desired using `drop()`.

The most significant challenge of the file system with Rust is serialization. With the original implementation, serialization is relatively simple: since the disk is simply a file, we can read bytes from the disk with `.read()`, and simply cast the bytes to our desired type, such as the index node `struct`. Writing to the disk is similarly straightforward, as we can utilize .write() to write a char* casted type to the type. However, Rust prohibits this capability, due to safety concerns; for instance, Rust cannot verify whether the bytes from read can actually be properly casted to the desired type, and in the event where we read bytes from an incorrect location or read an incorrect number of bytes, we will likely end up with erroneous undefined data. Thus, in order to do serialization and deserialization, we have several options.

One would simply be utilizing Rust's "escape hatch," the `unsafe` keyword. With `unsafe`, we can simply call `transmute()` and effectively use the same strategy as the original implementation. However, using `unsafe` is discouraged as it side steps much of what makes Rust a valuable language, so an alternate method that does not utilize `unsafe` would be a better implementation.

Instead, for deserialization, we can take advantage of the default methods for reading bytes from a file to a specified type. Specifically, we can take advantage of `.read_exact()`, which allows us to read a specific number of bytes into a predefined buffer. Using this method, we can read an entire `struct` by reading each individual field of the `struct`, retrieving the values we need, and defining a new `struct` that uses these new values. Since the use of these read methods do not require unsafe, we can define an `impl` for an index node, implementing this method to allow index nodes to have a read function.

We could likely use a similar approach for serializing, but for our last approach, we can simply use crates for serialization, `bincode` and `serde`. These creates allow us to easily serialize our index node, as long as we define the index node to derive serialization and deserialization.

Learning Curve

Before developing this project, I spent time learning the language by working on smaller scope projects in order to get a basic understanding of how Rust works. Working on smaller projects along with the file system project has allowed me to gain a greater understanding of how to use Rust. In this section, I will be describing the experience of learning Rust, and the challenges that were encountered.

Figure 3: *Implementation of an index node with the corresponding deserialize method,* `from_reader`*. This function allows us to read each field from a stream of bytes individually using the default read methods, making the use of* `unsafe` *unnecessary. We return a new struct that uses the new values read from the file. Unlike C and C++, the variables declared to be later read into must still be initialized, avoiding any possible undefined memory access.*

```rust
#[derive(Serialize, Deserialize)]
pub struct IdxNode {
    name: [u8; 8],
    size: i32,
    block_pointers: [i32; 8],
    used: i32,
}

impl IdxNode {
    // reads from file field by field to avoid using unsafe transmute.
    fn from_reader(mut rdr: impl Read) -> io::Result<Self> {
        let mut name = [0u8; 8];
        rdr.read_exact(&mut name)?;
        let size = rdr.read_i32::<LittleEndian>()?;
        let mut block_pointers: [i32;8] = [-1;8];
        for i in 0..8 {
            block_pointers[i] = rdr.read_i32::<LittleEndian>()?;
        }
        let used = rdr.read_i32::<LittleEndian>()?;
        Ok(IdxNode {
            name,
            size,
            block_pointers,
            used,
        })
    }
}
```

By default, many aspects of Rust are intentionally immutable. For example, defining variables or retrieving data from data structures are all immutable by default. In stark contrast to most other languages programmers are likely used to, a programmer in Rust must intentionally define every variable they wish to mutate with the mut keyword. By enforcing intentional mutability, Rust ensures that variables are no more mutabile than necessary–to the extent where the Rust compiler will warn you if you define a variable as mutable and never mutate it–thus limiting the potential for unexpected behaviors. However, immutability can at times be tricky to work around, especially as one must often define mutability in places one might not expect; for example, references must be defined as mutable if one wants to mutate the data the reference points to, and retrieving from certain data structures requires a specific mutable version of a get method if one wants to mutate the data in the data structure.

The ownership system can also be tricky to use properly when beginning to learn Rust. In practice, ownership can prevent many strategies that are easily done in other languages, such as passing data to a function and using that same data for something else. Understanding ownership is important to understand when to borrow, and knowing when to borrow ownership is integral to utilizing Rust properly and replicating functionality from other languages. Ownership can be especially tricky when taking into account when a variable is copied or moved; immutable variables always implement copy, so one can pass integers to functions and not encounter ownership related issues as the value of the integer was copied. However, in other cases with mutable types such as String, ownership will be moved and result in unexpected behavior. Thus, a intimate understanding of ownership and borrowing is necessary to properly program in Rust.

Rust also has prevalent usage of `Result<T>`, which forces error handling; one cannot access the desired data unless they intentionally account for the fact that the desired operation could have failed, and handle the error in some way. While some find the prevalence of `Result<T>` to be inconvenient [7], I find the usage of `Result<T>` akin to taking one's medicine: using `Result<T>` forces the programmer to intentionally account for any possible error vectors which ultimately results in healthier code. While one could simply use `.unwrap()` to get around using `Result<T>`, `.unwrap()` avoids the intention of accounting for a possible error and simply increases verbosity of the code. Instead, `.unwrap()` should be used when the programmer knows an error cannot occur, and in lieu of `.unwrap()` one should use `.expect()`, manual matching, or the ? keyword. t worst, usage of handling `Result<T>` can increase the length of the code and make code harder to read, but in practice, proper handling of `Result<T>` facilities debugging by easily identifying runtime errors, which may otherwise be missed and result in valuable development time dedicated to finding the error.

dditionally, Rust utilizes numerous ways of representing a number, which can present a small challenge at times. Due to Rust's strict typing rules, certain methods require a specific number representation, such as the File::seek method, which requires a u64 number. This restriction can cause confusion and unnecessary complexity if one wants to use an i32 representation of a number for this purpose, as one would be forced to either convert the number, which may potentially fail if it is unrepresentable in the new type, or change the original specification of the number to be the defined type by default. In practice, this behavior means that the programmer cannot simply define numbers to be i32 as in other languages which can simply be defined as int. Instead, they may need to understand ahead of time what functionality they wish to use this number for and ensure the type of this number matches its usage, to minimize unnecessary conversions.

s Rust aims to add abstractions for the convenience of the programmer, the syntax of Rust ends up being somewhat close to C and C++ while having significant differences for certain approaches. For example, defining for loops is intentionally different from C and C++, instead being something closer to Python, as Rust believes that this approach is clearer and easier to utilize compared to the traditional approach. In other cases, Rust's difference in functionality can allow for new conventions, such as how returning values at the end of a function does not require usage of the return keyword. This feature comes as a consequence of allowing functional programming behaviors, limiting the verbosity of a program.

Many of these new features of Rust can result in successful compilation to be surprisingly difficult to achieve. However, successful compilation means successful adherence to Rust's strict set of rules, and once compilation is achieved, one can be far more confident in correctness of a program and minimal bugs compared to a C or C++ counterpart.

Conclusion

It is, of course, impossible to say definitively whether Rust will replace C and C++. However, there is good reason to hope for this scenario to occur: the advantages of Rust in regards to safety and performance sets it apart from other languages, and these advantages are demonstrably useful for operating systems. Rust has shown that its benefits in safety make it an enticing option for developing systems over C and C++, and to date many large companies such as Google and mazon have embraced Rust as a primary systems language, hoping to eventually shift new development to memory safe languages.

However, Rust is certainly not a perfect language, and could fail to ultimately replace C and C++ for a variety of reasons. One such reason is its restrictive rules, which is also Rust's biggest strength. While its rules are ultimately for the benefit of the system, providing compile-time safety, examples of developing systems in Rust have shown to encounter issues with Rust's systems such as ownership. For example, while using Rust for developing an embedded system provides significant benefits, the rules of Rust can prevent the developers from doing crucial functionality such as resource sharing.

Similarly, learning how these rules work and how to work around them can be a challenging task. In order to properly utilize Rust, developers must understand how these systems work and work with them rather than avoid them with unsafe code, forcing C and C++ approaches in Rust. It is possible that the learning curve of Rust hinders adoption of Rust, leading to the traditional well known and well documented C and C++ languages to continue to be prevalent. Slow adoption could also lead to the rise of other competitors which could hypothetically fill the same purpose of Rust for systems. For example, Google's Carbon is one possible competitor, which boasts a gentler learning curve and interoperability with C++, while providing convenience and safety.

nother possible outcome could involve coexistence of Rust and C/C++, as Cantrill [8] posits, where the kernel is developed in C/C++ while all the other components of the operating system, including device drivers, are developed in Rust. In other words, Rust does not necessarily have to replace C/C++ to provide tangible benefits. Due to the fact that many systems today are developed in C/C++, many have looked into using Rust in addition to these languages. While it is not always applicable or clear where to separate the languages, this usage has certainly seen success. Most notably, ndroid Open Source Project (OSP) is predominantly in C and C++, yet the newest version of ndroid to date, ndroid 13, is the first version of ndroid where the most new code is in memory-safe languages. s a result, "memory safety vulnerabilities have dropped from 76% down to 35% of ndroid's total vulnerabilities," and "the ndroid team plans to step up usage of Rust, although there are no plans to get rid of C and C++ for its systems programming." [9] Thus, it is important to consider the possibility that Rust can coexist with C and C++ and provide significant and tangible benefits.

With the prevalent usage of Rust today, one could argue that Rust and languages are already replacing C and C++. Regardless, it is evident that the more Rust gets adopted, the safer, more secure, and easily maintained operating systems will become.

References

[1] Ralf Jung et al. "Safe systems programming in Rust". In: *Commun. CM* 64 (pr. 2021), p. 4. UR : `https://doi.org/10.1145/3418295`.

[2] J. V. Stoep. *nnouncing KataOS and Sparrow*. vailable at, 2022. UR : `https://opensource.googleblog.com/2022/10/announcing-kataos-and-sparrow.html`.

[3] bhiram Balasubramanian et al. "System Programming in Rust: Beyond Safety." In: *SIGOPS Oper Syst. Rev.* 51.1 (ug. 2017), p. 2017. UR : `https://doi.org/10.1145/3139645.3139660`.

[4] Steve Klabnik and Carol Nichols. *The Rust Programming Language*. vailable at, 2023. UR : `https://rust-book.cs.brown.edu/title-page.html`.

[5] mit Levy et al. “The Case for Writing a Kernel in Rust”. In: *Proceedings of the 8th sia-Pacific Workshop on Systems (PSys ’17). ssociation for Computing Machinery, , NY, US , rticle 1, 1–7.* 2017. UR : https://doi.org/10.1145/3124680.3124717.

[6] Vytautas strauskas et al. “How do programmers use unsafe rust”. In: *Proc. CM Program.* OOPSL , rticle 136 (November 2020), 27 pages: Lang. 4, 2020.

[7] ditya Saligrama, ndrew Shen, and Jon Gjengset. *practical analysis of Rust’s concurrency story. arXiv*. preprint. (pril 2019), 1-15, 2019. arXiv: 1904.12210. UR : https://arxiv.org/abs/1904.12210.

[8] Bryan Cantrill. “Is it time to rewrite the operating system in rust? (January 2019)”. In: *Retrieved December* 9 (2019), p. 2022. UR : https://www.infoq.com/presentations/os-rust/.

[9] J. V. Stoep. *Memory Safe Languages in ndroid 13, Google Online Security Blog.* vailable at, 2022. UR : https://security.googleblog.com/2022/12/memory-safe-languages-in-android-13.html.

[10] lex Light. *Reenix: Implementing a unix-like operating system in rust*. Brown University: Undergraduate Honors Theses, 2015.

[11] ndrew M. Scull. “Hephaestus: a Rust runtime for a distributed operating system”. In: *Computer Science Tripos Part II Dissertation. University of Cambridge Computer Laboratory* 23 (2015).

[12] mit Levy et al. “Ownership is theft: Experiences building an embedded OS in Rust”. In: *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*. 2015.