Systems Programming in Rust: A Survey

Agnishom Chattopadhyay Shanli Ding

December 7, 2021

Abstract

Rust has become prevalent in the recent years among software and systems engineers. Rust has seen wide-spread adoption in a variety of applications including OS Kernels, Browsers and even cloud services and compilers. In this survey, we first discuss the major pitfalls of programs written in C and discuss how Rust addresses these issues. Then, we look into Theseus, an operating system written in Rust with the goal of leveraging the safety features of the language to obtain safety guarantees for operating systems. Lastly, we discuss the challenges that software engineers face in adopting Rust.

1 The Pitfalls of Systems Programming in C

Traditionally, programming low level systems software including Operating System kernels, drivers, filesystems, etc have been carried out in a programming language like C (or C++). The programming ecosystems where C or C++ is the defacto standard also include embedded systems or browser engines. These two use cases demonstrate different facets of the utility of these languages: Embedded Systems are resource constrained environments where it is difficult to have runtime environments up. Rendering engines in browsers are not in a resource constrained setting, but they require non-trivial optimizations for good performance.

The main reason that C/C++ is considered so useful in such settings is that it is considered to be at an optimal level of abstraction. On the one hand, it has a nicer syntax than assembly providing features like variables, loops, function calls and pointers, instead of forcing the programmer to program in terms of concrete memory addresses, registers and jumps. At the same time, the abstractions provided by the language have a fairly simple translation to machine code which the average programmer has a good grasp of. These abstractions are also rather low-cost, in the sense that the instrumentation required to support their semantics consume very little resources. The simplicity of the C language makes it very portable as well.

The simplicity of the programming model of C is a core part of its philosophy, which is to believe that the programmer knows better than the compiler. This philosophy is most inconspicuous when it comes to memory management. The C programmer is responsible for manually managing all the memory that is allocated dynamically. In many cases, the memory management is somewhat orthogonal to the core logic of the application itself, and this is a major source of bugs.

In [16], we see a list of such bugs, many of which are deemed as *frequent* and *catastrophic*. We list some of them here. Most of these are runtime errors and depending on their role in the program, they can be very hard to find and debug. A more systematic listing of these bugs can be found in the Common Weakness Enumeration [14, 1] database under the classifications “Resource Management Errors”, “Memory Buffer Errors” and “Pointer Issues”.

1. **Memory Leaks:** This is the allocation of memory without subsequently freeing it up. More pedantically, a block of memory in the heap is said to have leaked if there is no chain of pointers connnecting it to the stack.

2. **Repeated or incorrect freeing of memory:** This may result in a run time error.

3. **Use after free:** Even if the memory may have been deallocated in the heap, a pointer to it may be still available in the stack. Trying to mutate this location may result in a runtime error.

4. **Null Dereferencing:** A null pointer may arise due to incorrectly initializing the pointer.

5. **Buffer Overflow:** Out-of-bounds reads or writes to adjacent memory objects. There has a large range of potential consequences. It could either have no visible consequences, silently corrupt some data or change the control flow of the program.

Many of these bugs, especially the last class, may be lead to severe security issues. Indeed, buffer overflows may allow attackers to read sensitive parts of data within a program or hijack its control altogether. This has been the focus of a huge amount of security research [9]. In 2019, Microsoft said that about 70% of their vulnerabilities are due to some kind of memory bug.[3]

To systematically track the allocation of memory, C programmers have developed several programming idioms. This includes concepts like region based memory allocation or idioms like *Resource Allocation is Initialization* (RAII). The C++ standard library has features like smart pointers, unique pointers which provide functionality similar to usual pointers but has additional functionality that enforces certain kinds of memory safety. As we will discuss in the next section, many of the features of the Rust language are inspired by these features.

Outside of the C/C++ ecosystem, the most common form of memory management is through the use of a Garbage Collector. The garbage collector is an instrumentation that runs alongside the main program freeing up resources which are no longer accessible from the stack. The garbage collector may run as a part of the virtual machine or interpreter in which the program runs (e.g, Java) or it may be bundled with the generated code, if the program is compiled. In a garbage collected language, memory management is automatic and the programming model does not involve the use of pointers. (However, in many cases null dereferencing is still a possibility. Although, 'physical leaks' are impossible, 'logical leaks' cannot be prevented.) Garbage collectors typically perform by pausing the program at frequent intervals and freeing up memory. They also produce an overhead that is caused due to the bookkeeping that the system must do. This is why garbage collection is not suitable for many programming tasks where C is typically used.

In a broad sense, types are a way of tagging objects so that incompatible objects cannot be used in interchangable contexts. While C does have a system of static typechecking, it is very rudimentary and common programming patterns often are set up in a way that allow for mistakes. For instance, `printf("%s", 12)` would cause a runtime error since 12 is not a string and the compiler cannot statically tell the difference between `printf("%s", 12)` and `printf("%d", 12)`. In addition, the C language offers functionality to cast data into arbitrary types as the programmer sees fit. In the context of programming systems, this feature is widely used to circumvent the type system and is known as *type punning*. Not being careful with types can also lead to memory errors, if one uses pointers of a type as a pointer for a different type, since different types of objects may take different amount of space in the memory.

The standard library for C is rather small and it contains functionality for mainly low level features like memory management or IO. It does not provide standard implementations of data structures like search trees or resizable vectors, which may be re-used in many applications. This is probably done in order to make the language as portable as possible.

2 Programming in Rust

2.1 Ownership and Borrowing

Ownership semantics is the central feature that sets Rust apart from other conventional programming languages. Instead of a garbage collected approach or a manually managed approach towards memory, the ownership semantics is used by the Rust compiler to manage memory statically. Every value in Rust has exactly one variable which owns it. The value is dropped when the owner goes out of scope. Since the scope can be determined statically, no garbage collector is needed. The compiler transparently inserts a destructor method (called `drop`) at the end of the lexical scope. See Figure 1 for example. (Many of these examples are inspired by [11] and [13])

The interesting cases arise when ownership is moved due to function calls or aliasing. After a value has been moved away from a variable, it is considered invalid. Consider the vector `v` in Figure 2. First, it is moved to `v1`, which renders the use of the variable `v` invalid. Next, `v1` is passed to the function `foo` by moving, which means that the vector is now owned by the (local) variable `u`. Note that for stack allocated values (or more precisely, for types with the trait `copy`) like 32-bit integers, aliasing is done by copying the value instead of moving it.

```rust
fn main() {
    {
        let mut v = vec![10, 11]; // note the use of `mut`
        v.push(12);
        println!("{:?}", v); //works fine
    }
    // v is dropped here
    //println!({}, v); //compiler error
}
```

Figure 1: Dropping Values that go out of scope

```rust
fn main() {
        let x = 5;
        let y = x; // the value in x is now copied to x
        println!("{}", y); //works fine
        println!("{}", x); //works fine

        let v = vec![10, 11];
        let v1 = v; // the value in v is now moved to v1
        println!("{:?}", v1); // works fine
        //println!("{:?}", v); // compiler error
        foo(v1); // the value in v1 is now moved to u
        //println!("{:?}", v1); // compiler error
}

fn foo(mut u : Vec<i32>){
    u.push(12);
    println!("{:?}", u); //works fine
}
// u is dropped here
```

Figure 2: Move Semantics

```
fn main() {
        let mut v = vec![10, 11];
        // the ownership of v is passed to foo
        // but it is then passed back
        v = foo(v);
        println!("{:?}", v); //works fine
}

fn foo(mut u : Vec<i32>) -> Vec<i32>{
    u.push(12);
    u // return the ownership of this vector back to the caller
}
```

Figure 3: Move by Return

```
fn main() {
        let mut v = vec![10, 11];
        println!("{:?}", v);
        {
            let v1 = &mut v; // v1 mutably borrows v
            //println!("{:?}", my_length(&v)); //compiler error
            foo(v1);
            println!("{:?}", v1);
        }
        println!("{:?}", my_length(&v));
}

fn foo(u : &mut Vec<i32>){
    u.push(12);
}

fn my_length(u : &Vec<i32>)-> usize{
    u.len()
}
```

Figure 4: Borrowing Semantics

Ownership is also transfered when a function call returns. This might be one way to gain back the ownership of a certain value. See 3.

Reclaiming ownership by returning may be an inconvenient strategy in general. Instead of transfering ownership, one may provide functions (or aliases) access to values via references or borrowing. As hinted by Figure 1 and 2 above, Rust encourages using values in an immutable manner whenever possible and requires marking variables that may be used mutably. Being mutable is especially consqeuential with borrowing semantics. The main rule of borrowing is as follows: A value can either be mutably borrowed by a single variable or be immutably borrowed by multiple variables, but never both at the same time.

Consider Figure 4. The function `foo` mutably borrows a vector, but the function `my_length` immutably borrows a vector. In the inner lexical scope of `main()`, the variable `v1` mutably borrows the vector. This is why, the expression `&v` cannot be used to simultaneously immutably borrow it. This is fine outside the lexical scope as demonstrated by the last line. Contrast the borrowing semantics of Figure 4 with the move semantics of Figure 2.

Note that implementing the move or borrow semantics does not necessarily require copying the entire heap allocated data structure, it can be implemented simply by passing around pointers. If a type supports cloning (i.e, implements the `clone` trait), it can be explicitly cloned by calling `.clone()` if necessary.

```rust
fn main() {
    let u = vec![1, 2];
    let v = vec![1, 2, 3];
    let (winner1, winner2) : (&Vec<i32>, &Vec<i32>);

    winner1 = longer(&u, &v);
    println!("The larger vector is: {:?}", winner1);

    {
        let w = vec![1, 2, 3];
        winner2 = longer(&u, &w);
    }

    // println!("We have a dangling pointer to : {:?}", winner2); // compiler error
}

fn longer<'a>(u: &'a Vec<i32>, v: &'a Vec<i32>) -> &'a Vec<i32> {
    if u.len() > v.len() {
        u
    } else {
        v
    }
}
```

Figure 5: Lifetimes

2.1.1 Lifetimes

When working with references, we must make sure that the reference does not outlive the value that they are referring to. Rust achieves this by carefully checking lifetimes of values. In some situations, it is not possible to do this automatically and some annotations are needed. For example, in Figure 5, we have annotated the function `longer`. This annotation can be read as "for any lifetime `'a`, given two references of lifetime `'a`, we return a reference of lifetime `'a`". We see that we are not allowed to call `longer(&u, &w)`. This is because the vector owned by `w` does not live as long as `u`. If we did manage to get the reference `&w` back from the `longer(&u, &w)` function, it would be a dangling pointer.

Now, we are in a position to compare how the memory management in Rust measures up to the programming practices in C. First off, the memory management is not manual, so there is no incorrect allocation and dellocation. Using lifetimes, the compiler can statically prove that there are no dangling references. In an attempt to prevent some of the very common errors that occur in C programs, Rust has some additional restrictions as well. For example, Rust does not allow using a variable without initializing it with a variable first. It also does not allow dereferencing arbitrary pointers.

In some cases, Rust is able to statically detect out-of-bound array accesses. However, if an out-of-bound access happens at runtime, the executable can detect it, and abort the execution. This is achieved by some low-overhead code that is injected in.

We consider a case study from [11] to understand the benefits further.

2.1.2 Case Study: Pointer Invalidation

A vector is a data structure representing a resizable array, i.e, an array whose size is not fixed at compile time. Let's say that the we had a vector `v` which contained two elements and we had a pointer to the second element of the vector. If more elements were pushed into the vector, the elements of the vector may be re-allocated, leaving the pointer obtained before invalidated. Thus, if the programmer tries to modify the second element of the original vector, this will not help since this now no longer refers to the original vector. As shown in Figure 6, this is automatically detected by the borrow checker. Since the `push_back` function mutably borrows the vector, this is in conflict with the `vptr` variable, which is also mutably borrowing the vector.

```
//C++
std::vector<int> v {10, 11};
int *vptr = &v[1]; //pointer inside v
v.push_back(12);
*vptr = 100; //bad!

//Rust
let mut v = vec![10, 11];
let vptr = &mut v[1]; //pointer inside v
v.push(12);
// *vptr = 100; // cannot mutably borrow v twice
```

Figure 6: Pointer Invalidation

2.2 How Rust deals with Concurrency

Instead of asking programmers to follow a rigid pattern of concurrent programming, Rust tries to support multiple approaches to concurrent programming via the standard library. This includes the common patterns like message passing and having shared state across threads. Having shared state across threads can lead to data race conditions, i.e, situations where the behavior of the system depends on the exact sequence of events in a concurrent setting. Interestingly, Rust's solution to these problems is rooted in its ownership and borrowing semantics, the same principles it uses to handle memory safety. Multi-threaded programs written in Rust have very little runtime overhead. (In fact, it does not provide any notion of *green threads* (language supported threads, instead of OS threads) because it wants to minimize runtime overheads.)

One technique of communicating between threads is to use message passing, where information is explicitly sent between threads. As an example, see Figure 7. We have created two channels with transmitters `tx1, tx2` and receivers `rx1, rx2`. We use one to pass integer values and other to pass a vector. Since the channels are typed, they may not be used interchangably. Now, we send the three elements of the vector `v` at first individually through the first channel and then, send the vector itself via the second channel. Notice that since the two spawned threads may outlive the main thread, one of these threads must own the vector `v`, otherwise it may be dellocated. Next, we notice that the vector `v` itself is sent via the channel via the move semantics. This means that `v` cannot be used after it has been sent to the receiver. This is to prevent the first thread from modifying the data or de-allocating it after it has been sent.

Concurrency based on shared state is more complex. As an example, let us consider the program in Figure 8 which hides a value behind a mutex. In the example, several threads are incrementing a counter. Often, these kind of code is guarded with critical sections so that no thread is exposed to an inconsistent state while a certain kind of update is happening. A mutex, or a lock, is implemented in a way that only a single thread can hold the lock at a time. The `counter.lock().unwrap()` produces a value of type `MutexGuard<i32>` which is a smart pointer behind which the counter value is present. When the `MutexGuard` goes out of scope, the lock is automatically released. This is managed by the destructor call that is inserted by the compiler at the end of the scope. This is yet another place where the ubiquitous RAII pattern is being cleverly used.

This example is also interesting from the point of view of the Rust ownership semantics. First, every thread mutably uses the counter behind the mutex. So, who owns the mutex? The answer is that the mutex is owned by a smart pointer, which is indicated by `Arc`, *Atomic Reference Counter*. With every call to `Arc::clone(&counter)`, the smart pointer produces an additional reference to the mutex. Additionally, it keeps track of the number of references that are pointing to it. When all the references go out of scope, the smart pointer can free the resource it is holding. A smart pointer like this one must be used here since it cannot be statically determined which thread would be the last one to terminate. The *A* in `Arc` stands for *Atomic*, meaning that the reference counter is updated in a thread safe manner when a new reference is requested. (Rust also has a `Rc` smart pointer, which cannot be used in multi-threaded contexts.) The other thing that we must notice is that multiple threads can mutate the value behind the mutex. While this is statically true, the implementation of the mutex makes sure that no thread can hold a mutable reference.

```rust
use std::sync::mpsc;
use std::thread;

fn main() {
    let (tx1, rx1) = mpsc::channel();
    let (tx2, rx2) = mpsc::channel();

    let v = vec![1,2,3];

    thread::spawn(move || { // note the move
        tx1.send(v[0]).unwrap(); // sending by copy
        println!("v[0] : {}", v[0]);
        tx1.send(v[1]).unwrap(); // sending by copy
        tx1.send(v[2]).unwrap(); // sending by copy
        tx2.send(v).unwrap(); // sending by move
        //println!("v : {:?}", v); // compiler error
    });

    thread::spawn(move || {
        println!("Got: {}", rx1.recv().unwrap());
        println!("Got: {}", rx1.recv().unwrap());
        println!("Got: {}", rx1.recv().unwrap());
        println!("Got: {:?}", rx2.recv().unwrap());
    });
}
```

Figure 7: Message Passing Example

```rust
use std::sync::{Arc, Mutex};
use std::thread;

fn main() {
    let counter = Arc::new(Mutex::new(0));
    let mut handles = vec![];

    for _ in 0..10 {
        let counter = Arc::clone(&counter);
        let handle = thread::spawn(move || {
            let mut num = counter.lock().unwrap();
            *num += 1;
        });
        handles.push(handle);
    }

    for handle in handles {
        handle.join().unwrap();
    }
    println!("Result: {}", *counter.lock().unwrap());
}
```

Figure 8: Programming with Mutex

2.2.1 Unsafe Rust

It turns out that one cannot implement primitives such as `Arc` or `Mutex` within the *safe* fragment of Rust. Instead, such features must be implemented in *unsafe* Rust, a fragment of the language which is relaxed about the aforementioned memory safety rules, marked unsafe by using the `unsafe` keyword. Code that uses unsafe Rust need not necessarily be unsafe itself. The operating philosophy of unsafe Rust is to allow the programming language to be flexible where it is necessary and trust the (responsible) programmer. The programmer is supposed to hide their implementation details behind an API which is safe and obeys the principles of Rust. As an example, the `Arc<Mutex<T>>` pattern above may have been implemented using code that cannot be checked by the Rust compiler to be safe, but it does indeed provide the same guarantees at runtime. The Rustonomicon [6] explains how unsafe Rust should be used and exposed with safe APIs. Using these tools, one can extend the concurrency primitives of Rust further if so necessary. Types can be marked with the `Sync` and the `Send` trait to indicate that they can be safely used in multi-threaded programs.

The concurrency and memory safety guarantees provided by Rust, even the safe fragment, do not cover everything. Complex code can still have deadlocks. Smart pointers using reference counting could leak memory if they are used in data structures with cycles.

2.3 Other features of Rust

While the borrowing and ownership semantics is the central element of Rust's design features, there are a number of other elements in Rust, that are inspired by other modern languages including functional languages like Haskell. Unlike C, the Rust ecosystem has a flagship build tool called Cargo. There is also a (relatively large) standard library which comes with a number of core datatypes, standard macros and support for multi-threading and I/O.

Rust is generally type-safe and has a much richer typing system compared to many other languages. This includes support for parametric polymorphism (in the sense of generics in Java, or templates in C++). There is no implicit casting between types, although some primitive data types can be cast into one another explicitly. There is no `null` value that can be used across types, or confused with legitimate values. Instead, Rust programmers use the `Option<T>` type for each type `T`. To consider the example of 32 bit integers `i32`, `Option<i32>` can either be `None` or `Some(i)` for some integer `i`. Interestingly, in some cases, the compiler is able to safely remove the additional indirection that would be introduced by the `Some()` [2]. Rust does not have control flow operations that correspond to exception handling; instead, it encourages handling (catchable) exceptions via the `Result<T, E>` type.

As is clear from the discussion above, Rust has support for algebraic data types. This could be used to implement C like enums, but in a typesafe manner. The `switch` statement from C has been replaced by a `match` statement which allows pattern matching for algebraic types. Match statements may also have guards or bindings, similar to languages in the ML family (Haskell, OCaml).

Rust allows for systematically dealing with ad-hoc polymorphism via the use of traits. Traits are similar to the notion of type-classes in Haskell or interfaces in Java. Types implementing a common trait all have common *methods* defined on them that are implemented differently, but satisfy an unified contract. For instance, the trait `std::cmp::Ord` is implemented by types which are linearly ordered. In the previous section, we have already talked about `Clone`, the trait of elements which can be cloned Some traits do not have any specific methods, but are markers to indicate certain kinds of behavior. For example, the `Send` trait is a marker trait indicating that a value can be transferred across boundaries. The compiler will implement the `Send` trait automatically if it deems it appropriate! Some other traits convey special semantic meaning. When a type implements the `Copy` trait, as we discussed before, values of that type are passed via copying instead of using the move semantics. Traits have a notion of inheritance: for instance, types implementing the `Copy` trait must implement the `Clone` as well; elements that are totally ordered via `std::cmp::Ord` must also be partially ordered via `std::cmp::PartialOrd`.

The trait system is extremely flexible. Not only can methods in traits can be polymorphic (have generic types), but the entire trait could be parametrized with a type variable. Additionally, one may use associated types as a part of the trait definition. As an example, the trait of `Iterator` have an associated `Item` type, which indicates the type of item the iterator is iterating over. With associated types, it is possible to encode higher kinded traits (such as Functor, Monad) [15].

There is some support for Object-Oriented style programming in Rust, but various concepts are viewed differently. Objects are implemented using `Struct`s, which are similar to C like structs. Structs in Rust owns its resources in the sense of the ownership semantics we described. One can define methods associated with a struct that may mutate the fields of the struct. However, instead of using inheritance between structs, Rust programmers must use trait objects to support polymorphism. This means instead of specifying a class which dynamically supports certain operations based on which subclass it actually is, the Rust programmer instead implements the same idea by specifying a type variable and the required traits it must implement. One may also dynamically return trait objects from a function. This means that the function would not have a concrete return type, but instead only a guarantee that the return type implements a certain trait. Rust generally wants to make sure that it knows the size of all the types it is using, but it cannot do so in this case since the specific type being returned will be dynamically determined. Thus, the trait object must be explicitly allocated on the heap using a box. In particular, if one wants to return a trait object of the trait `T`, they instead return a `Box<dyn T>` explicitly.

Idiomatic rust code takes advantage of traits and methods, and modern programming patterns like iterators and method chaining syntax. In many cases, these are zero cost abstractions, as the compiler can optimize away overheads present due to method calls and iterators. Some functional style list processing primitives like map and filter are also supported via iterators which accepts lambdas (function closures) as input.

3 Case Studies: Operating System

3.1 Theseus

Now that we understand the key principles of Rust, we can discuss an experimental Operating System *Theseus*, developed by Boos et al [7]. Theseus is built using Rust with the goal of utilizing the safety features of Rust directly to ensure the safety features of the operating system. The developers of the Theseus project have concluded that the ownership mechanism in Rust can be leveraged to develop a modular design for an operating system. They claim that their design facilitates live evolution and fault recovery, which would be desirable of systems such as network switches in data centers or space probes, where downtime (possibly due to applying system updates) is unacceptable.

3.1.1 Theseus Design Principles

Boos et al have identified *state spill* as a key weakness in the design of systems that make ensuring their safety difficult. In [8], the authors state that state spill occurs when “a software entity’s state undergoes lasting changes as a result of a transaction from another entity”. State spill, in the context of operating systems, could happen due to improperly designed indirection layers, improperly designed inter-entity collaboration or when using multiplexing or dispatcher mechanisms.

As the shining point of Theseus system architecture is that consisting many tiny components within a crate. To be specific, Thesesus states that each component is a cell, and cells are based on crates. The straightforward meaning is about the strategy of modular programming. Theseus’ first design principle is runtime-persistent cell bounds, which means to dynamically load cells at runtime, allow the system to track cell bounds, and also both for address space and privilege level will be single mapping. Secondly, maximizing the power of the language and compiler. According to this principle, it mainly focuses on advantages of Rust’s powerful abilities in dealing with the system safety issues. Hence, we will go further on this principle. Lastly, minimizing state spill between cells that is a root causes for various system problems.

By achieving the second and the third principles, Theseus applies Rust[12] as the compiling language for the system source code. Before we delve into how Theseus applies Rust to maximize the power for system and compiler and avoid state spill making system safer, There are important reasons for Theseus to choose Rust instead of C language. Firstly, as we discussed above, the ownership model, plays a crucial role inside building a memory safe system. Besides, according to all reports we surveyed about Rust’s efficiency and expressiveness among other coding languages show that Rust takes advantage to combine the power and semantic of a high-level coding language with a pretty desirable high efficiency, like C, without garbage collections. As the reason for designing the Rust language is to provide strong static types and memory safety guarantees, unlike C, at the compiling time. Therefore, based on these three main properties of Rust, it is a great choice for Theseus to use for constructing a evolvable, available and safe system.

3.1.2 Rust in Theseus

Theseus mainly takes advantage in *ownership*, and *borrowing*. The importance of ownership model in building a memory safe system stands for the fundamental of the whole architecture. Although we have discussed about the Rust's ownership detailly at previous section, at this section we will have a quick recap of how does the ownership work inside the code because there is one more dominant factor called lifetime that makes the system's memory lives in a comfortable zone. Inside the Figure 9, there is a real example from Theseus Operating System for understanding how the ownership and borrowing in Rust help it become memory safe. According to the whole code snippet, the function is dealing with a local variable named *mp_pgs* at *Line 6*. It gets assigned a variable by mapping pages. Start at *Line 7*, there is a handy usage for ownership's lifetime property. So at the *Line 10*, the local variable called *hpet* will be auto-deallocated since it reaches the end of the scope. As we come to *Line 12*, there is a transferring ownership operation on the variable *mp_pgs* because inside the main function, it calls another function called *send*() and it takes *mp_pgs* as the input. Therefore, even at *Line 14*, the memory for *mp_pgs* will not be de-allocated, instead its ownership has been already moved to another variable. According to the other function named *receiver_task*(), it demonstrates the way to use immutable borrowing and lifetime property as well. At the *Line 17* and *Line 18*, there are two variables with different properties. The variable *mp* means that it owns the ownership from the old owner, so it has now accessible for the current return property *receiver.receive*(). On the other hand, the variable *hpet* means that it is under immutable borrowing policy, thus, *hpet* is not able to gain the ownership from the original owner and modify any attributes inside the original owner's properties. At the end, when the program reaches at the *Line 21*, local variable *mp* will be auto-dropped by Rust and such ownership will disappear as well, because it gains the ownership from the original owner and reaches the end of its lifetime, function's scope. In short, this process will explain how a ownership model takes part in the Theseus Operating System.

Within the Theseus system, system engineers apply the Rust's ownership model on ensuring resource cleanup via unwinding. If we look back into C compiled kernel, we will find some difficulties when people wants to run some tasks that exchanges some pointers' value, and then the kernel may unexpected doubly-free[4] which waste available rooms. However, by utilizing the *ownership* property in Rust can handdle this issue easily. Back to Theseus' unwinder that is built upon Rust's ownership and borrowing properties. Simply, when some jobs need resources from Theseus, then they can own their objects on the stack without stack overflow. As for Rust compiler, it will track each object's ownership while jobs are running by processes, and then the compiler can determine when a section of resource is dropped by a job and where to execute the cleanup command for the resource. The unwinder function inside the Theseus starts only when an exception or a request to kill comes up. And it is not as same as garbage collectors due to no interference during the execution. All in all, Theseus apply Rust ownership to make up its own unwinder to handle the memory leakage issue in the system resource pools.

Moreover, delving into memory management of Theseus system. As normal memory issues as C kernel has including page mapping, bounds checking, and resource releasing. According to the first two main issues, Theseus engineers utilize functions to take ownership of the allocated pages and frames to return a new page object[8]. So it does secure the relationship between pages, frames and mapped pages due to ownership's property. And it also can assure that all of them will not be re-used for multiple repeated mappings. Besides, by using the lifetime property, Theseus can resolve the problem about the region bounds checking. According to solve the resource releasing issue, Theseus applies the same strategy on the ownership of values, so the compiler will know when and where to clean and release the resources.

3.1.3 Thoughts and Takeaways from Theseus

We have dig deeper into Theseus system architecture and its kernel source code compiler mechanism, namely the Rust language. The acquisitions we gained from learning the Theseus OS are from the usage of Rust's ownership property because within the Theseus OS there are lots of auxiliary and kernel functions done by ownership with resolving some kernel memory issues. And the values of ownership

```
struct HpetRegisters {...}

fn main() -> Result<()> {
        let frames = get_hpet_frames()?;
        let pages = allocate_pages(frames.count())?;
        let mp_pgs = map(pages, frames, flags, pg_tbl)?;
        {
                let hpet: &HpetRegisters = mp_pgs.as_type(0)?;
                print!("HPET device Vendor ID: {}", hpet.caps_id.read() >> 16);
        }
        ...
        sender.send(mp_pgs)?;
        Ok(()) // `mp_pgs` not dropped, the ownership is transferred
}

fn receiver_task(receiver: Receiver<MappedPages>) -> Result<()> {
        let mp: MappedPages = receiver.receive()?;
        let hpet: &HpetRegisters = mp.as_type(0)?;
        print!("Current HPET ticks: {}", hpet.main_counter.read());
        Ok(()) // `mp` auto-deallocated and unmapped
}
```

Figure 9: Example Ownership Model

model are ability for implementing isolation and zero-cost state transfer code within the operating system's components[8]. Moreover, by surveying many related works about using Rust as kernel's source code due to guaranteeing memory safety, but there is still a problem needed engineers to solve, memory leakage. At first, it is misunderstanding for both terminologies *memory safe* and *memory leak* because if people as a beginner like us will have an intuitive acknowledgement that memory leak is equal to memory unsafe. However, memory leakage will happen when the code is memory safe, because the meaning of memory safe is that the program is using the safe code. Therefore, inside the Theseus system, there is still problem about memory leak though many disadvantages from C kernel, such as Linux, are resolved as expect.

4 Human Factors

In the Stack Overflow Developer Survey, Rust has been voted as the most loved language every year since 2016, despite the fact that only 7% of the users claim to be using it.

Rust was used in a 2020 survey [5] to see how experienced programmers make use of resources to learn new programming languages. It appears that the programmers involved in the experiment spent a large amount of time examining example code from the *Rust by Example* book. The time spent by these programmers accessing this resource was twice the time spent by the programmers on Stack Overflow. They also reported that they found the inline-compiler errors very helpful in general, but can lead the novice programmer astray if they don't understand the related concept well.

In 2021, Fulton et al [10] surveyed 178 Rust programmers about their personal and professional experiences in using Rust. The participants generally agreed that the Rust ecosystem has good tooling and documentation. Even though most interviewees said that they learnt Rust out of curiosity, they describe the learning curve as rather steep. Even though the participants seem to value the safety features of Rust, unsafe blocks seem to be relatively common.

References

[1] CWE VIEW: software development. https://cwe.mitre.org/data/definitions/699.html. Accessed: 2021-10-30.

[2] Stack overflow. https://stackoverflow.com/a/16515488/1955231, 2013 Accesssed: 2021-11-07.

[3] A proactive approach to more secure code. https://msrc-blog.microsoft.com/2019/07/16/a-proactive-approach-to-more-secure-code/, Jul 2019.

[4] Doubly freeing memory. https://owasp.org/www-community/vulnerabilities/Doubly_freeing_memory, Accesssed: 2021-11-23.

[5] Abtahi, P., and Dietz, G. Learning rust: How experienced programmers leverage resources to learn a new programming language. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems* (2020), pp. 1–8.

[6] Beingessner, A., and Klabnik, S. The rustonomicon: The dark arts of advanced and unsafe rust programming, 2016.

[7] Boos, K., Liyanage, N., Ijaz, R., and Zhong, L. Theseus: an experiment in operating system structure and state management. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)* (Nov. 2020), USENIX Association, pp. 1–19.

[8] Boos, K., Vecchio, E. D., and Zhong, L. A characterization of state spill in modern operating systems. In *Proceedings of the Twelfth European Conference on Computer Systems* (New York, NY, USA, 2017), EuroSys '17, Association for Computing Machinery, p. 389–404.

[9] Cowan, C., Wagle, F., Pu, C., Beattie, S., and Walpole, J. Buffer overflows: Attacks and defenses for the vulnerability of the decade. In *Proceedings DARPA Information Survivability Conference and Exposition. DISCEX'00* (2000), vol. 2, IEEE, pp. 119–129.

[10] Fulton, K. R., Chan, A., Votipka, D., Hicks, M., and Mazurek, M. L. Benefits and drawbacks of adopting a secure programming language: Rust as a case study. In *Seventeenth Symposium on Usable Privacy and Security (SOUPS 2021)* (Aug. 2021), USENIX Association, pp. 597–616.

[11] Jung, R., Jourdan, J., Krebbers, R., and Dreyer, D. Safe systems programming in rust. *Commun. ACM 64*, 4 (2021), 144–152.

[12] Klabnik, S., and Nichols, C. The rust programming language. https://doc.rust-lang.org/book/, 2018 Accesssed: 2021-11-20.

[13] Klabnik, S., and Nichols, C. *The Rust Programming Language (Covers Rust 2018)*. No Starch Press, 2019.

[14] Martin, R. A., and Barnum, S. Common weakness enumeration (cwe) status update. *Ada Lett. XXVIII*, 1 (Apr. 2008), 88–91.

[15] Turon, A., and Klock, F. S. {RFC: associated items. https://github.com/rust-lang/rfcs/blob/master/text/0195-associated-items.md#encoding-higher-kinded-types, 2014 Accessed: 2021-11-08.

[16] Vipindeep, V., and Jalote, P. List of common bugs and programming practices to avoid them. *Electronic, March* (2005).