



Università
della
Svizzera
italiana

Faculty
of
Informatics

Bachelor Thesis

June 16, 2023

Rust for Kernel-level Distributed Services

Fabio Fabrizio Zampielo Lutz

Abstract

The Linux kernel is written in the C programming language, but recently a contender has risen in popularity: the Rust programming language. Recently, experimental support has been introduced in a stable release of the Linux kernel, making Rust a viable language for writing the kernel and its modules. The goal of this project is to analyze the current Rust kernel environment and attempt to build a proof-of-concept distributed service as a kernel module in Rust.

For this project, I began by dedicating time to study the Rust programming language. This served us to highlight the differences between the Rust programming language and C, which I was already familiar with thanks to previous courses taken during the bachelor.

Then we researched the capabilities of the new Rust support for the Linux kernel by compiling the official release of the kernel with the Rust support enabled and use it to create a working out of tree module written in Rust.

Advisor

Prof. Patrick Eugster

Assistants

Dr Pavel Chuprikov, Davide Rovelli

Advisor's approval (Prof. Patrick Eugster):

Date:

Contents

1	Introduction	2
2	Differences between Rust and C	3
2.1	Design	3
2.2	Safety	3
2.3	Flexibility	4
2.4	Ease of Use	4
2.5	Performance	4
2.6	Conclusions	4
3	Working with the Kernel	6
3.1	State of the Art	6
3.2	Comparison	6
3.3	Useful Information	8
3.4	Challenges and Obstacles Faced in the Project	9
3.5	Rust for Linux project	9
3.6	How to Contribute	9
4	Guide to implementation	10
4.1	Requirements	10
4.2	Compile the Kernel	11
4.3	Write your module	11
4.4	Install Kernel and insert the Module	12
5	Conclusion	14
5.1	Future Work	14
6	Acknowledgements	14

1 Introduction

Rust is a high-level programming language that appeared for the first time in 2015. It's focus is in performance, type safety, and concurrency. Born from a personal project of Graydon Hoare, Rust has seen a rise in popularity and in December 2022 it became the second high-level language to be supported in the development of the Linux kernel, the first being C, thanks to the experimental support in the stable release of the version 6.1 of the Linux kernel.

Our project aim is to study the differences between using C or Rust while working with the Linux Kernel, compile a functioning Linux Kernel with the new Rust support enabled and create a out of tree module using that Kernel as the base.

This report is divided in three main sections: Differences between C and Rust², Working with the Kernel³ and Guide to Implementation⁴.

In Differences between C and Rust² section we explore the biggest differences between the two programming languages, C and Rust, having comparisons between the different aspects which we found relevant, like safety, flexibility and much more. There are also other differences that we may have not express in this report but this are the ones that have more in relations with our project.

In Working with the Kernel³ section we instead write about how is to work with the Linux Kernel and the challenges we found. We discuss the state of the art for the Rust support of the Linux Kernel and what work is being done by the community and by a project called Rust for Linux^[2] which will be cited more time in this report since many of the progress in this particular field was done by this small group and we utilised the available resources we found for our project. We also compare writing a out of tree module using Rust or C, with images and examples to show in practise how different they are. The challenges and obstacles we faced while working on this project are also written in this section, which unfortunately are the reasons this project result is only partially what we had in mind to do for lack of time and inexperience. I added also a small paragraph to talk on how anyone interested could contribute to make Rust support more than just experimental.

In the last main section Guide to Implementation⁴ we have a practical guide on how to reproduce what we made and how to enable Rust support. It's based on official documentation for the Kernel^[1] but we made the guide so that anyone can have his virtual machine in Vagrant with the modified Linux Kernel installed and use it to make working module by doing simple steps.

2 Differences between Rust and C

To gain a better understanding of why one might choose to replace C with Rust for kernel development, it is essential to outline the general dissimilarities between the two programming languages. This section will highlight the key distinctions across various aspects, including Design, Safety, Flexibility, Ease of Use, and Performance. While these differences extend beyond the scope of kernel development, they provide valuable insights into the rationale behind choosing Rust over C. For specific discrepancies between the two within the kernel context, refer to the dedicated section in Working with the Kernel³.

2.1 Design

With design of a programming language I mean the main idea of the creators of the language when it was first created. Both C and Rust had a clear design and idea behind their creation but both wanted to offer an alternative from older programming languages.

C was designed to replace other low level languages, such as Assembly, which were hard to read and to maintain. With C they created a higher level language which is more readable than Assembly, able to do every low level operation, and easier to maintain, while remaining portable.

Additionally, C puts the responsibility of the code safety in the hands of the users, to allow a better flexibility while writing low level code and the management of data and memory. Rust was designed to be more performing than its predecessors, with highly secure concurrent systems and ensure memory security without using Garbage Collector, thanks to its system of ownership for data. Its aim is to compete with C and C++ in terms of performance, resource consumption and safety.

Therefore, C successfully aimed to replace Assembly, and now Rust is striving to replace or rival C and C++ in terms of performance, resource efficiency, and safety.

2.2 Safety

The safety of the code written in the C programming language largely depends on the skills and expertise of the users. C is a low level language that allows the users to manipulate data freely, this is by design. However, even experienced programmers can face safety issues due to the inherent vulnerabilities present in C libraries or within the projects they are working on. These projects may be old or open source, lacking thorough consideration of every possible corner case for ensuring safety. It is important to note that despite the users' skills, these factors can still contribute to potential safety problems.

Most of the vulnerabilities related to C programs are memory safety vulnerabilities related to its code. Some examples are buffer overflow, memory leaks and input validations. Working with parallelism and multithreading is also not safe while using C for the same reason as before, freedom for the user while writing code inherently means less secure code. So with no barrier or checks to verify the safety of the code means that the user will have difficulties with managing concurrency among threads.

Rust is safer in various aspects thanks to its own static analysis at the time of compilation, which impedes most of the problems that a user would have using C. Thanks to the restrictions imposed to the user during compilation, the user is forced to write safer code from the start, without the needs for debugging or the risk to have major safety risks.

For data management Rust uses a unique feature, a system of ownership. Ownership rules in Rust are enforced by the borrow checker, which ensures memory safety without the need for garbage collection. There are three main rules for ownership: Each value in Rust has an owner; There can only be one owner at a time; When the owner goes out of scope, the value will be dropped.

Since each value has a single owner at any given time, the owner is responsible for managing the lifetime of the value and deallocating its memory when it is no longer needed. When an owned value goes out of scope, Rust automatically cleans up the memory associated with it. Ownership can be transferred through moving or borrowing. Moving ownership involves transferring the ownership of a value from one variable to another. After the transfer, the original variable can no longer access the value. On the other hand, borrowing allows temporary access to a value without transferring ownership. Borrowing can be either mutable or immutable, depending on whether the borrowed value can be modified or not. The ownership system in Rust ensures that only one owner exists for a value at a time, preventing data races and memory leaks. This system also enables Rust to provide zero-cost abstractions and guarantees memory safety without sacrificing performance.

Rust has memory safety guarantees enforced at compile time as explained before, but there is a way to write unsafe code and it's considered as a second "hidden" language. Unsafe Rust allows to use unsafe functions or method or do other actions like dereferencing a raw pointer and more, without the compiler enforcing memory safety. Unsafe Rust

is really important for this project since most of the functionalities that allow Rust to work for the Linux Kernel are in fact unsafe.

2.3 Flexibility

If we consider the capabilities of the two languages, both have the potential to write any function but they have different approaches to flexibility.

C by design has put the responsibility of the code in the hands of the users, allowing them to manage as they see fit many low level functionalities, such as memory management. This allows the user a lot of freedom at the cost of the safety of the code. Rust in the other hand, to allow his design to work has traded flexibility for a more secure and stable code. It's easy to see this difference in flexibility just from how they both manage their memory. To allow Rust to work with his ownership system it had to put restraints that impeded the user to write certain code, which may be considered unsafe. Of course this is put aside if we consider Unsafe Rust which allows a more flexible code.

2.4 Ease of Use

When it comes to choosing between learning C or Rust, both languages have steep learning curves and are not particularly beginner-friendly. They need extra steps and more theoretical knowledge than some other easier to use languages, like for example Python. However, between the two, C offers more learning possibilities due to its longer history. There is a lot of documentation, libraries, tutorials, and books available for learning C. On the other hand, since Rust is a relatively young language, there are fewer resources available for learning it in depth.

After this small consideration about learning the two languages, we can say that both languages demand their particular way to write code and interact with memory. Both of which comes simpler only with experience. We could consider C more easier to use than Rust thanks to C having more flexibility in its code, allowing simpler and faster way to solve problems. On the other hand Rust is more strict with its code and it will almost certainly require more steps to do have the same functions from C. I experienced this first hand while translating code from C to Rust. One of the small projects that I worked on was a simple binary search tree implementation, which required a lot more knowledge, time and lines of code to write in Rust. This from the start of it with the structure of each node, having to declare certain type for the values of each leaf, since Rust doesn't allow a structure to have references itself without specific types to allow that. This alone had this project to introduce multiple extra libraries only for the structure and code to work with such data type. A small snippet of the two codes for the binary search tree can be seen in figure 1 written in C and in figure 2 written in Rust.

In conclusion, both can achieve the same results, but thanks to its flexibility, C is easier to use than Rust, while both remaining really hard to master and powerful languages.

2.5 Performance

The performances of the two languages are based on results of running the same algorithms with the two. Both time and memory efficiency are taken in consideration. The results are taken from the paper "Energy efficiency across programming languages: how do energy, time, and memory relate?" [5] and a table with the normalized global results is shown in 3. They did a great job to compare each programming language by using The Computer Language Benchmarks Game initiative [3] which include a framework for testing the languages on various famous algorithms. The results of this comparison, normalized with the global results of all the algorithms used, puts C and Rust at the top 2 positions for time efficiency, with only 0.04ms of difference between the two. For memory used, C beats Rust with a difference of 0.37Mb. Therefore, we can assume based on this data that C and Rust are very similar in terms of speed and memory usage with small differences between the two.

2.6 Conclusions

This comparison served us to know why Rust would be a great substitute for C in the kernel environment. As we have seen the performances of the two programming languages are mostly similar overall, this reassures us that with either of the two we can achieve very similar performance results. This leaves us with the focal point of Rust, safety, in both memory management and parallelism and many other situations, where C on the other hand by design leaves safety on the hand of the user. Rust safety has the cost of a worse ease of use and flexibility, by putting restraints over the user design for a sturdier code. In conclusion, I think both Rust and C are great programming languages to use with the kernel but the choice in the end is based on if the users value more the flexibility and ease of use of C with the freedom it gives to users, or the security and sturdiness of Rust.

```

use std::option::Option;

struct Node {
    value: i32,
    left: Option<Box<Node>>,
    right: Option<Box<Node>>,
}

impl Node {
    // Creates a new node with a value
    fn new(value: i32) -> Node {
        Node {
            value: value,
            left: None,
            right: None,
        }
    }

    // Create a new node in the tree with a value
    fn insert(tree: &mut Node, value: i32) {
        if value <= tree.value {
            if tree.left.is_some() {
                Self::insert(tree.left.as_mut().unwrap(), value);
            } else {
                tree.left = Some(Box::new(Self::new(value)));
            }
        } else {
            if tree.right.is_some() {
                Self::insert(tree.right.as_mut().unwrap(), value);
            } else {
                tree.right = Some(Box::new(Self::new(value)));
            }
        }
    }
}

```

Figure 1. Rust BST Code Example

```

#include <stdio.h>
#include <stdlib.h>

struct node {
    int key;
    struct node *left, *right;
};

struct node* newNode(int item)
{
    struct node* temp
        = (struct node*)malloc( sizeof(struct node));
    temp->key = item;
    temp->left = temp->right = NULL;
    return temp;
}

struct node* insert(struct node* node, int key)
{
    if (node == NULL)
        return newNode( item: key);

    if (key < node->key)
        node->left = insert( node: node->left, key);
    else if (key > node->key)
        node->right = insert( node: node->right, key);

    return node;
}

```

Figure 2. C BST Code Example

Total					
	Energy		Time		Mb
(c) C	1.00	(c) C	1.00	(c) Pascal	1.00
(c) Rust	1.03	(c) Rust	1.04	(c) Go	1.05
(c) C++	1.34	(c) C++	1.56	(c) C	1.17
(c) Ada	1.70	(c) Ada	1.85	(c) Fortran	1.24
(v) Java	1.98	(v) Java	1.89	(c) C++	1.34
(c) Pascal	2.14	(c) Chapel	2.14	(c) Ada	1.47
(c) Chapel	2.18	(c) Go	2.83	(c) Rust	1.54
(v) Lisp	2.27	(c) Pascal	3.02	(v) Lisp	1.92
(c) Ocaml	2.40	(c) Ocaml	3.09	(c) Haskell	2.45
(c) Fortran	2.52	(v) C#	3.14	(i) PHP	2.57
(c) Swift	2.79	(v) Lisp	3.40	(c) Swift	2.71
(c) Haskell	3.10	(c) Haskell	3.55	(i) Python	2.80
(v) C#	3.14	(c) Swift	4.20	(c) Ocaml	2.82
(c) Go	3.23	(c) Fortran	4.20	(v) C#	2.85
(i) Dart	3.83	(v) F#	6.30	(i) Hack	3.34
(v) F#	4.13	(i) JavaScript	6.52	(v) Racket	3.52
(i) JavaScript	4.45	(i) Dart	6.67	(i) Ruby	3.97
(v) Racket	7.91	(v) Racket	11.27	(c) Chapel	4.00
(i) TypeScript	21.50	(i) Hack	26.99	(v) F#	4.25
(i) Hack	24.02	(i) PHP	27.64	(i) JavaScript	4.59
(i) PHP	29.30	(v) Erlang	36.71	(i) TypeScript	4.69
(v) Erlang	42.23	(i) Jruby	43.44	(v) Java	6.01
(i) Lua	45.98	(i) TypeScript	46.20	(i) Perl	6.62
(i) Jruby	46.54	(i) Ruby	59.34	(i) Lua	6.72
(i) Ruby	69.91	(i) Perl	65.79	(v) Erlang	7.20
(i) Python	75.88	(i) Python	71.90	(i) Dart	8.64
(i) Perl	79.58	(i) Lua	82.91	(i) Jruby	19.84

Figure 3. Normalized global results for Energy, Time, and Memory

3 Working with the Kernel

In this section, we will explore various aspects of working with the Linux Kernel using Rust and examine how it differs from using C. The subsection titled "State of the Art" provides an overview of the current development status of Rust support for the Linux Kernel and discusses potential actions one can take. The "Comparison" section includes examples illustrating how to write modules in both Rust and C, highlighting relevant similarities and differences. Finally, we conclude with a brief section called "How to Contribute," which offers suggestions on expanding Rust's support within the Linux kernel.

3.1 State of the Art

Experimental support for Rust in developing the Linux Kernel was added in version 6.1 and remains so as of the time of writing this report. Even in this early stage, the ability to write modules and other functionalities for the kernel using Rust can be as powerful as using C. However, there is a lack of documentation and support. During our work on this project, we discovered that common Rust libraries cannot be used, as the kernel requires the use of a smaller, kernel-specific library. The Rust for Linux [2] project and the community are expanding this smaller library, but in the stable release of the Linux kernel, functionalities are limited compared to our initial expectations.

For example, we wanted to utilize parallelism to create a simple module for a "Ping Pong" scenario, where two threads coordinate to print "Ping" and "Pong." Unfortunately, the necessary libraries were missing in the stable release, although they were available in the Rust for Linux version of the kernel. As a result, we decided not to pursue the development of that module. Another challenge we encountered was the lack of comprehensive documentation on Rust support in the official Linux Kernel documentation. Only a small portion covers the requirements for enabling Rust support and provides specific recommendations. Despite having this documentation, it took us a significant amount of time to find a clear method to enable and compile the kernel with Rust support.

In conclusion, considering the current state of Rust support for Linux, we can acknowledge its potential as a powerful tool. However, several essential libraries may be missing, and the official documentation is still insufficient for a proper understanding of how to use Rust correctly in this context. Nonetheless, substantial efforts are being made to make Rust as capable as C for working with the kernel.

3.2 Comparison

This section presents a comparison between writing modules in Rust and C. One of the key differences I observed is in how they utilize their respective kernel libraries to provide functions and data for the modules.

In C, the module information is written, and the init and exit functions are defined using the kernel library functions, similar to regular C code. The module information, such as the description using `MODULE_DESCRIPTION("My ker-`

nel module"), is set, and the functions `module_init()` and `module_exit()` are used to specify the init and exit functions respectively.

On the other hand, Rust employs a struct-like approach where values are placed within a module structure. Additionally, a separate structure, referred to as the Type, implements the necessary functions, including the init and exit functions.

Here we can observe in Figure 4 how Rust employs a struct-like approach to store valuable information about the module. It also includes a Type value that corresponds to the name of the struct used to implement our functions and can even store data for us. In contrast, Figure 5 illustrates how C utilizes a function-based system to modify the information by directly accessing the variables.

```
module! {  
    type: RustOutOfTree,  
    name: "rust_out_of_tree",  
    author: "Rust for Linux Contributors",  
    description: "Rust out-of-tree sample",  
    license: "GPL",  
}  
  
struct RustOutOfTree {  
    numbers: Vec<i32>,  
}
```

Figure 4. Rust struct-like approach for module info

```
MODULE_DESCRIPTION("My kernel module");  
MODULE_AUTHOR("Me");  
MODULE_LICENSE("GPL");
```

Figure 5. C approach for module info

Here instead we observe the differences between the init and exit functions, as well as how they are specified for use within the module. As shown in Figure 6, Rust implements these functions directly within the struct, creating distinct sections for the init (loading) and exit (removal) phases of the module. The "Ok()" is used to declare the structure and its values to the module.

On the other hand, Figure 7 demonstrates that in C, the functions are simply created independently. They can then be passed to the `module_init()` and `module_exit()` functions, which specify which functions should be used as the init and exit functions for the module.

```
impl kernel::Module for RustOutOfTree {  
    fn init(_module: &'static ThisModule) -> Result<Self> {  
        pr_info!("Rust out-of-tree sample (init)\n");  
  
        let mut numbers = Vec::new();  
        numbers.try_push(72)?;  
        numbers.try_push(100)?;  
        numbers.try_push(200)?;  
  
        Ok(RustOutOfTree { numbers })  
    }  
}  
  
impl Drop for RustOutOfTree {  
    fn drop(&mut self) {  
        pr_info!("My numbers are {:?}\n", self.numbers);  
        pr_info!("Rust out-of-tree sample (exit)\n");  
    }  
}
```

Figure 6. Rust implementation directly into the struct


```

static int dummy_init(void)
{
    pr_debug("Hi\n");
    return 0;
}

static void dummy_exit(void)
{
    pr_debug("Bye\n");
}

module_init(dummy_init);
module_exit(dummy_exit);

```

Figure 7. C passing the function name

3.3 Useful Information

In this small subsection some small useful information that we found while working on this project that can help while working with the Rust support in the kernel.

The location of where the libraries you can use while writing your own out of tree module is located in the Rust folder inside the Linux folder. You can follow this path "linux-6.2.7/rust/kernel" for the version we are using of Linux 6.2.7 and you should have various Rust file as you can see in figure 8 that you can import into your module code, like you would do with normal libraries.

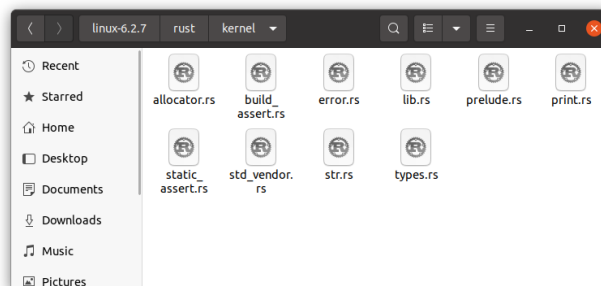


Figure 8. Directory with the Rust developed libraries

There are a couple of already built in sample of module made in Rust that you can enable like you would with other sample module. It's useful to read them and see how they structure they're code since they can be converted to an out of tree module without much effort. You can find them in the sample folder under the rust folder "linux-6.2.7/samples/rust". As you can see in figure 9 there are only two samples, but as for the libraries mentioned before, you can go to the Rust for Linux[2] version of the kernel which has more samples that you can look at. You can enable this samples in the menuconfig before compiling your kernel and they will be compiled and built in the kernel. You can follow this path to enable them and you should come to section line in the figure 10.

```

Kernel Hacking —>
  [*] Sample kernel code —>
    [*] Rust samples —>
      — Rust samples

```

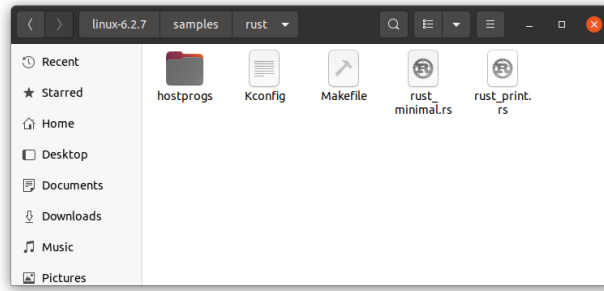


Figure 9. Directory with the Rust samples

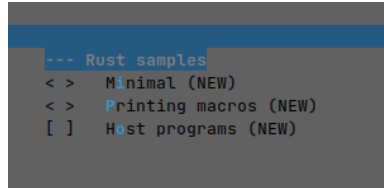


Figure 10. menuconfig section where to enable the samples

3.4 Challenges and Obstacles Faced in the Project

During the project, we encountered numerous obstacles that significantly slowed down progress, leaving us with limited time to develop a more complex module as the project's outcome. Initially, our plan was to create a sophisticated module that utilized both parallelism and network capabilities. This involved setting up two Vagrant nodes with our customized Kernel, enabling Rust, and establishing communication between them using our module. However, implementing a functional Kernel with Rust proved to be more complicated than anticipated. My lack of experience working with a Kernel, coupled with the relatively new Rust support, hindered our progress in finding a reliable method to compile a functional Kernel with Rust support that could be installed on a virtual machine like Vagrant. Nonetheless, we managed to overcome these obstacles despite my inexperience.

We encountered additional issues when attempting to write the module for the Kernel. We discovered that many standard Rust libraries had not been implemented in the official release of the Linux Kernel. Consequently, we were unable to proceed with the module we initially planned, as parallelism and network support were unavailable. We did find some of these functionalities developed in the Rust for Linux version [2], but due to time constraints, we were unable to delve further into this project at the time of this report.

3.5 Rust for Linux project

Rust for Linux[2] is an external project which is aiming to add support for the Rust language to the Linux kernel. They are the main contributors for the support for Rust in the Linux kernel, in fact you can read they're authorship in many of the samples and code written for Rust in the Linux kernel. The Rust for Linux project was announced in 2020 and in two years on October 2022 a pull request for accepting the implementation for Rust for Linux was approved by Torvalds for version 6.1. They decided to intentionally leave the Rust support minimal in order to allow developers to test the feature, which unfortunately didn't help us for our project. Miguel Ojeda and Gary Guo are the main contributors to this project, having developed most features in the project and they are still up to today developing for the Rust for Linux project, together with others members of the community. The project is receiving many attentions from tech corporations and received generous financial support from Google and Futurewei.

3.6 How to Contribute

If you are interested in contributing to the implementation of Rust support for the Linux Kernel, I recommend starting by following the Guide to implementation provided in this report. It serves as a valuable starting point and will help you gain initial experience in developing within the kernel environment using Rust. While the guide is comprehensive, it's important to note that some information may be missing. Therefore, I recommend referring to the official documentation of the Kernel for additional guidance. Once you feel confident and ready to contribute, I suggest exploring the Rust for Linux project [2]. This project is the main group actively working on developing Rust support

and has received sponsorship from companies like Google. They can provide further resources and support for your contributions.

4 Guide to implementation

Here is documented a guide on how to work on creating a module for the Linux Kernel using Rust. I used and written this steps to help whoever wants to contribute or try for himself to create and work with Rust on the Linux Kernel. Most of this steps are taken from the official documentation for the kernel and from the Rust for Linux project [2], which I will talk more about later in this report. This guide suppose that you are working on a Linux based system, so all of the commands here are written for the Linux terminal. We are going to use as a base a github repository made for this project made by Davide Rovelli and me.

4.1 Requirements

We are going to use Vagrant, an open-source software that we are going to use to create a virtual box used to install the modified version of the Kernel we are going to compile and use as the base for the compilation of the module. We also need to install Ansible. It's not necessary, but I suggest also to install Oracle VM Virtualbox 6.1, to easily manage the nodes we're going to make.

```
$ wget -O- https://apt.releases.hashicorp.com/gpg |
$ sudo gpg --dearmor -o /usr/share/keyrings/hashicorp-archive-keyring.gpg

$ echo "deb [signed-by=/usr/share/keyrings/hashicorp-archive-keyring.gpg]
$ https://apt.releases.hashicorp.com $(lsb_release -cs) main"
| sudo tee /etc/apt/sources.list.d/hashicorp.list

$ sudo apt update && sudo apt install vagrant
$ python3 -m pip install --user ansible
```

After having installed Vagrant, we will clone the repository made for this project. The version of the Linux Kernel that we used is the 6.2.7, but as already said before, Rust came available since the version 6.1 so any version after that one should work fine.

```
$ git clone git@github.com:swsystems/kernel-ds-rs.git
$ cd kernel-ds-rs
```

Now we need to start the node of vagrant and enter in the environment. The boot up will also download the Linux Kernel. There are three node available in this project for Vagrant, but we are going only to use one.

```
$ vagrant up
$ vagrant ssh node01
```

Suggestion while working with vagrant, this commands allow you to: reload the node; destroy it and return it to the starting state; suspend and save it's state

```
$ vagrant destroy node
$ vagrant reload
$ vagrant suspend
```

This will install the needed packages on the node and Rust, if still gives problem on install, try again `$ sudo apt -fix-broken install`

```
$ sudo apt update
$ sudo apt --fix-broken install

$ sudo apt install -y bc bison curl clang fish flex git

$ curl https://sh.rustup.rs -sSf | bash -s -- -y
$ source "$HOME/.cargo/env"

$ sudo apt-get install libssl-dev
```

This will install Bindgen, the version 0.56.0 is the one I used

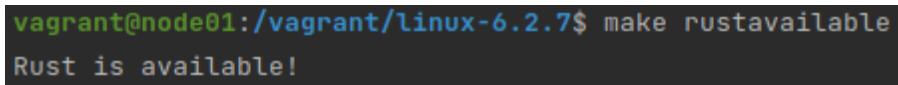
```
$ git clone https://github.com/rust-lang/rust-bindgen -b v0.56.0 --depth=1
$ cargo install --path rust-bindgen
```

Now we have to go into the Linux Kernel directory and set the version of Rust to the version required by that Kernel. Then we'll add rust-src.

```
$ cd linux-6.2.7
$ rustup override set $(scripts/min-tool-version.sh rustc)
$ rustup component add rust-src
```

The only thing that remains is to check if everything is in order. To do so use use this command

```
$ make rustisavailable
```



```
vagrant@node01:/vagrant/linux-6.2.7$ make rustavailable
Rust is available!
```

Figure 11

The output should be "Rust is available" as show in 11. If it's not, it should say what the problem is and you should resolve it before preceding.

4.2 Compile the Kernel

Before compiling the Kernel, we need to modify the configurations of the Kernel by first setting the configurations to the default ones. Then we need enable Rust support and the Vagrant virtual file system, so that when installed, Linux can mount the shared folder of Vagrant.

```
$ make defconfig
```

```
$ make menuconfig
```

After using menuconfig a GUI will appear, there we will enable Rust and Vagrant virtual file system. In image 12 we can see how it will look like. Here I also have written what we need to enable and where to find it.

```
General setup -->
[*] Rust support
```

```
Device Drivers -->
[*] Virtualization drivers -->
<*> Virtual Box Guest integration support
```

```
File systems -->
[*] Miscellaneous filesystems -->
<*> VirtualBox guest shared folder (vboxsf) support
```

Now you only need to save the new configuration and compile the Kernel. It will take some minutes to finish.

```
$ make -j2
```

4.3 Write your module

Now that we have successfully compiled the Kernel with Rust support available, we will be able to write and compile our first Kernel module. To do this we will make a out of tree module using a basic template made by Miguel Ojeda [4], a software engineer that contributes to the Rust for Linux [2] project. A modified version of that repository is already in the main directory.

```
$ cd ../rust-out-of-tree-module
```

```

[*] Support initial ramdisk/ramfs compressed using gzip
[*] Support initial ramdisk/ramfs compressed using bzip2
[*] Support initial ramdisk/ramfs compressed using LZMA
[*] Support initial ramdisk/ramfs compressed using XZ
[*] Support initial ramdisk/ramfs compressed using LZ0
[*] Support initial ramdisk/ramfs compressed using LZ4
[*] Support initial ramdisk/ramfs compressed using ZSTD
[ ] Boot config support
[*] Preserve cpio archive mtimes in initramfs
Compiler optimization level (Optimize for performance (-O2)) --->
[ ] Configure standard kernel features (expert users) --->
[ ] Embedded system
Kernel Performance Events And Counters --->
[*] Profiling support
[*] Rust support

```

Figure 12

```

AS      arch/x86/boot/header.o
LD      arch/x86/boot/setup.elf
OBJCOPY arch/x86/boot/setup.bin
BUILD   arch/x86/boot/bzImage
Kernel: arch/x86/boot/bzImage is ready (#1)

```

Figure 13

In this folder we can find the Makefile and Kbuild, which are basic files used to compile the module, and a file called `rust_out_of_tree.rs`, that is our Rust file with the code for our module. It is a base for how a module should be written using Rust and it is a small modified version of the original on the rust out of tree github repository. It's already a working module so we will use it as a first check to see if everything works as intended. To compile it we will use the make command and reference the directory where our compiled Kernel is.

```
$ make KDIR=../linux-6.2.7
```

And it's done, it should output something like in figure 14 and figure 15

```

vagrant@node01:/vagrant/rust-out-of-tree-module$ make KDIR=../linux-6.2.7
make -C ../linux-6.2.7 M=$PWD
make[1]: Entering directory '/vagrant/linux-6.2.7'
RUSTC [M] /vagrant/rust-out-of-tree-module/rust_out_of_tree.o
MODPOST /vagrant/rust-out-of-tree-module/Module.symvers
CC [M] /vagrant/rust-out-of-tree-module/rust_out_of_tree.mod.o
LD [M] /vagrant/rust-out-of-tree-module/rust_out_of_tree.ko
make[1]: Leaving directory '/vagrant/linux-6.2.7'

```

Figure 14

If everything worked fine, now it's time that we install that Linux kernel on our Vagrant node, so that we can install the module we just compiled.

For other way of writing code and examples, there is the section of samples in the Linux folder with Rust module, they are great example on what you can write on that version of the Kernel.

4.4 Install Kernel and insert the Module

The passages for installing the Kernel and inserting the module are not different from the ones you would normally do while working with the Linux Kernel.

First to install the Kernel in our node we first install the modules required and then the Kernel

```

$ cd ../linux-6.2.7
$ sudo make modules_install
$ sudo make install

```

```

vagrant@node01:/vagrant/rust-out-of-tree-module$ modinfo rust_out_of_tree.ko
filename:      /vagrant/rust-out-of-tree-module/rust_out_of_tree.ko
author:        Rust for Linux Contributors
description:    Rust out-of-tree sample
license:       GPL
depends:
retpoline:     Y
name:          rust_out_of_tree
vermagic:      6.2.7 SMP preempt mod_unload

```

Figure 15

```

vagrant@node01:/vagrant/linux-6.2.7$ sudo make modules_install
INSTALL /lib/modules/6.2.7/kernel/fs/efivarfs/efivarfs.ko
INSTALL /lib/modules/6.2.7/kernel/drivers/thermal/intel/x86_pkg_temp_thermal.ko
INSTALL /lib/modules/6.2.7/kernel/net/netfilter/nf_log_syslog.ko
INSTALL /lib/modules/6.2.7/kernel/net/netfilter/xt_mark.ko
INSTALL /lib/modules/6.2.7/kernel/net/netfilter/xt_nat.ko
INSTALL /lib/modules/6.2.7/kernel/net/netfilter/xt_LOG.ko
INSTALL /lib/modules/6.2.7/kernel/net/netfilter/xt_MASQUERADE.ko
INSTALL /lib/modules/6.2.7/kernel/net/netfilter/xt_addrtype.ko
INSTALL /lib/modules/6.2.7/kernel/net/ipv4/netfilter/iptable_nat.ko
DEPMOD /lib/modules/6.2.7

```

Figure 16

```

VirtualBox Guest Additions: Building the modules for kernel 6.2.7.

This system is currently not set up to build kernel modules.
Please install the Linux kernel "header" files matching the current kernel
for adding new hardware support to the system.
The distribution packages containing the headers are probably:
    linux-headers-generic linux-headers-5.15.0-67-generic
run-parts: executing /etc/kernel/postinst.d/xx-update-initrd-links 6.2.7 /boot/vmlinuz-6.2.7
I: /boot/initrd.img is now a symlink to initrd.img-6.2.7
run-parts: executing /etc/kernel/postinst.d/zz-update-grub 6.2.7 /boot/vmlinuz-6.2.7
Sourcing file '/etc/default/grub'
Sourcing file '/etc/default/grub.d/init-select.cfg'
Generating grub configuration file ...
Found linux image: /boot/vmlinuz-6.2.7
Found initrd image: /boot/initrd.img-6.2.7
Found linux image: /boot/vmlinuz-5.15.0-67-generic
Found initrd image: /boot/initrd.img-5.15.0-67-generic
Warning: os-prober will not be executed to detect other bootable partitions.
Systems on them will not be added to the GRUB boot configuration.
Check GRUB_DISABLE_OS_PROBER documentation entry.
done

```

Figure 17

Now we just need to exit the node and reload it. It should automatically have the new Kernel as the main one on boot. After it started, just enter again in the node and check if the version is correct, it should display the version of the Kernel we just installed and it should have the shared folder.

```
$ uname -r
```

Now we just need to try to insert the module and see if it is installed and if we can remove it.

```

$ cd /vagrant/rust-out-of-tree-module
$ sudo insmod rust_out_of_tree.ko
$ sudo rmmod rust_out_of_tree.ko
$ sudo dmesg

```

You should see our module working and printing in the log as in figure 18.

```
[ 74.266900] rust_out_of_tree: loading out-of-tree module taints kernel.  
[ 74.266997] rust_out_of_tree: Rust out-of-tree sample (init)  
[ 79.082574] rust_out_of_tree: My numbers are [72, 108, 200]  
[ 79.082588] rust_out_of_tree: Rust out-of-tree sample (exit)
```

Figure 18

5 Conclusion

Through this project, we have deepened our understanding of the recent integration of Rust support in the Linux Kernel. By creating a simple example of an out-of-tree module written in Rust, we have gained insights into the current possibilities and limitations of working with the existing Linux Kernel implementation. Our findings provide a strong starting point and a clearer understanding for future research and exploration of this new kernel feature. We now have a solid foundation to build upon and a better grasp of what can be accomplished and what areas require further development and refinement.

5.1 Future Work

Many things can be done from where we have left. Although time constraints limited our ability to delve deeper into the project, much needs to be done to write a more complex module for the Linux kernel. So for future work may focus on expanding the Rust support for the kernel. This can be achieved for example by writing more libraries into the kernel or add more features that will allow more complex work to be done with this new Rust support.

6 Acknowledgements

I would like to thank and express my gratitude towards Professors Patrick Eugster, Davide Rovelli and Dr. Pavel Chuprikov for allowing me to work on this project and for they're guidance and help trough the whole project.

References

- [1] The linux kernel documentation. <https://www.kernel.org/doc/html/latest/>.
- [2] Rust for linux. <https://rust-for-linux.com/>.
- [3] I. Gouy. The computer language benchmarks game. <https://benchmarksgame-team.pages.debian.net/benchmarksgame/index.html>.
- [4] M. Ojeda. Rust out of tree. <https://github.com/rnestler/rust-out-of-tree-module>.
- [5] R. Pereira, M. Couto, F. Ribeiro, R. Rua, J. Cunha, J. Fernandes, and J. Saraiva. Energy efficiency across programming languages: how do energy, time, and memory relate? pages 256–267, 10 2017.