Interim Report:
Developing Device Drivers in Rust

Kyle Christie
B00415210

School of Computing, Engineering
and Physical Sciences

BSc (Honours) Computing Science
University of the West of Scotland

Supervisor: Paul Keir
Moderator: Stephen Devine

Table of Contents

Table of Figures

Abstract

Memory safety is a critical issue faced by device drivers - a technology which already suffers from a range of problems – and it is believed that a solution lies in the Rust programming language. As well as expanding on the issues (and related consequences) surrounding device drivers, this report serves to introduce the Rust programming language alongside the high criticality of memory safety. With insight into the work that preceded this project , this report also provides a summary of current progress and considers work which will be carried out in the future.

1. Overview

Device drivers are a vital component of Operating Systems and facilitate the use of common peripheral devices, interaction with hardware as well as providing a multitude of extensions to an Operating System in its file system(s), network protocol, anti-virus capability and more (Ball et al, 2006). Drivers can also be described as the "software layer that lies between applications and physical devices" (Corbet et al, 2005.). While drivers are a clear necessity within an Operating System, they suffer from a range of issues that have various consequences.

First of all, drivers continue to be programmed with the C programming language. C was first developed at Bell Labs between 1969 and 1973, alongside early development of Unix (Ritchie, M. D, 1993). It was designed as a "system implementation language for the nascent Unix operating system" (Ritchie, M. D, 1993). C, C++ and Assembly have the potential to be memory unsafe (Gaynor, 2019) which can then lead to critical vulnerabilities as observed by several organisations over the years (Thomas and Gaynor, 2019).

Memory safety is an attribute of select programming languages that prevents developers from introducing certain bugs that strongly relate to memory management (Prossimo, 2022) Issues with memory safety usually lead to security problems with typical vulnerabilities being Out-of-bounds reads, out-of-bounds writes and use-after-frees (Gaynor, 2019).

Next, Drivers have seen little to no change within the last two decades. Evidence pointing to this can be found in Linux Device Drivers 3, a book written for Linux Kernel 2.6 (Corbet et al, 2005), where its code examples can compile and successfully run on more recent kernel versions with little to no change. Further evidence supports this point as even online tutorials from 2014 (Karthik M, 2014) continue to compile and run on recent kernel versions.

Such examples have been built and executed on a small collection of Linux distributions that utilise more recent kernel versions, specifically 4.19.0-17-amd64, 5.15.0-52-generic and 5.15.67-v7l+.

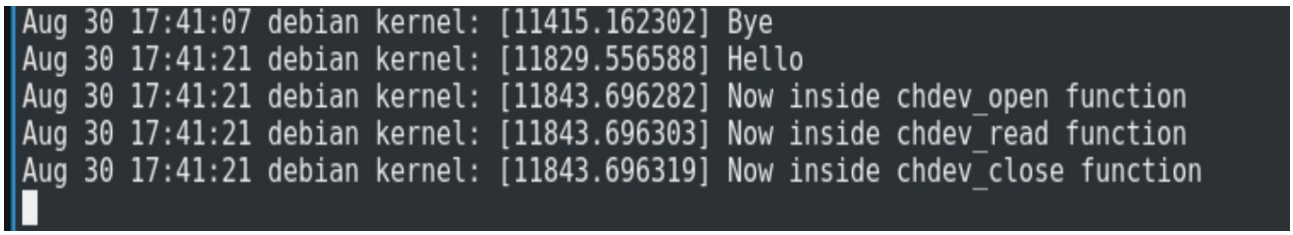

Figure 1: Debian Virtual Machine with Linux kernel 4.19.0-17-amd64 running character driver from Karthik M tutorials.

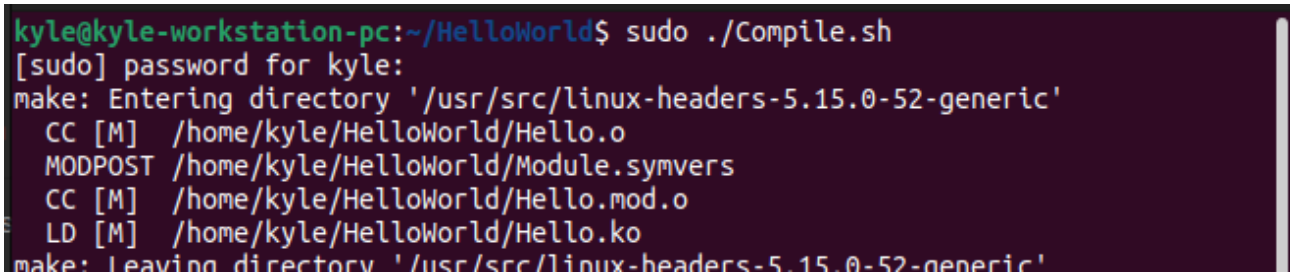

Figure 2: Ubuntu Workstation with Linux Kernel 5.15.0-52-generic compiling Hello World driver example from Linux Device Drivers 3

Figure 1 demonstrates execution of a character driver from the Linux Device Drivers Training series by YouTube Channel 'Karthik M' (Karthik M, 2014). Figure 2 demonstrates the compilation of a Hello World example available in Linux Device Drivers 3 (Corbet et al, 2005). It is clear that Drivers, especially Linux Kernel Modules, have not evolved in any major way as code which targets Linux Kernel versions from over a decade ago continues to run on more recent versions.

The goal of this project is to develop a Linux Driver in Rust. It is a relatively young language with several benefits and features that aim to improve memory safety. It continues to spread through industry as it was recently incorporated into the Linux Kernel version 6.1 (Vaughan-Nichols, 2022) and there have been public calls from developers for Rust to be utilised more. An example of this being Microsoft Azure CTO, Mark Russinovich, urging the industry (regarding to C and C++) 'For the sake of security and reliability, the industry should declare those languages as deprecated.' (Claburn, 2022).

2. Literature Review

2.1 Rust

Rust is a "compiled, concurrent, safe, systems programming language" (Klabnik, 2016) which released in 2015. It was originally invented by Graydon Hoare, an employee at Mozilla, who started the project in 2006 which was then adopted by Mozilla in 2010. Rust has several features which are highly attractive especially with regards to drivers and memory safety.

Figure 3: Social media logo for the Rust programming language (Mozilla, n.d.)

Cargo is the build tool and package manager for the Rust language and is responsible for managing dependencies within a project while also allowing users to create their own packages (Rust Community, n.d.). Rust projects typically include a .toml configuration file which cargo uses to read dependencies. This way cargo can automatically download and install dependencies. If necessary it will also manage dependencies of dependencies and is therefore a highly convenient tool for developers. Cargo is supplemented by 'Crates.io' which is an open-source repository (or registry) that holds all public crates or libraries.

Rust is accompanied by a powerful compiler that makes use of a strong type system and enforces good practices in code. It checks code at compile time so errors can be detected before code is deployed (Li et al, 2019). Therefore, the compiler is also used to highlight errors and prevent developers from making common mistakes (Klabnik, 2016) as it gives clear feedback on errors and how they may be solved (Oatman, 2022). This is critically important, especially within drivers, as it was previously established that writing device drivers is no easy task. Developers previously

struggled with the Windows XP driver API (Ball et al, 2006) and it has been highlighted that writing C code for the kernel is difficult (Renzelmann and Swift, 2009). The compiler also disallows unused variables and enforces correct concurrency (Oatman, 2022). If a variable is sent to be owned by a thread or channel, it can no longer be read, and a compiler error occurs if an attempt to read is made (Oatman, 2022). The compiler also forces the developer to handle errors (Oatman, 2022).

Rust code is immediately reliable as code will always be backwards compatible with old code always able to compile with new versions of the language (Oatman, 2022). This means that old code will benefit from optimisations made to the rust toolchain, code of all ages will improve and speed up alongside the language itself. The added benefit of this is a small revolution in code maintenance, some of the most popular crates can be considered 'complete'. In some cases, they have not been updated in a long time, as the code has no issues and is less likely to rot.

Rust has no defined memory model thus has simple memory structures compared to that of JVM and Go. As there is no garbage collection there is no generational memory or complex substructures. Memory is managed as part of execution, applying the Ownership model during runtime (Sasidharan, 2020).

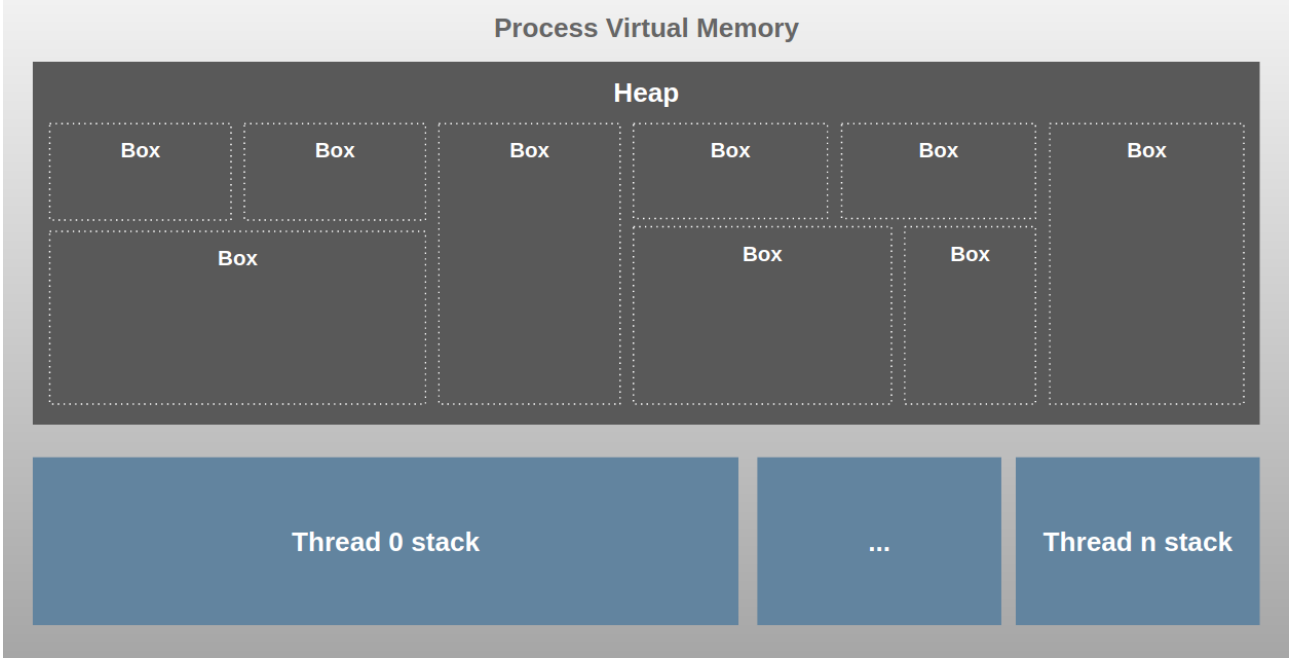

Figure 4: Rust process memory layout (Sasidharan, 2020)

Rust, of course, implements a Stack and Dynamic Heap within programs. Typically all variables are placed on the stack with the following exceptions; A manually created box and when the variable size is unknown or grows over time. In these cases, the variable is then allocated to the heap with a pointer to the data placed on the Stack. A box is an abstraction that represents a heap-allocated value. In order to manage memory, Rust uses a system of Ownership upheld by three rules which are applied both the stack and heap;

1. Each value must be owned by a variable
2. There must always be a single owner for a variable at any time
3. When the owner goes out of scope, the value is dropped

These rules are checked at compile-time, memory management is conducted at runtime with execution, this means there is no cost to performance or further overhead. Ownership can be changed with the `move` function. This is performed automatically when a variable is passed to a function or when the variable is re-assigned, `copy` is instead used for static primitives. Rust utilises RAIL - Resource Acquisition is Initialisation - which is enforced when a value is initialised. Under RAIL, the variable owns its related resources with its destructor called when the variable goes out of scope, which reduces the need for manual memory management. This concept is borrowed from C++. Rust also implements a system of borrowing where a variable which can be used rather than taking ownership of the variable, a borrow-checker enforces ownership rules (Sasidharan, 2020).

Variables have lifetimes which is important for the functionality of the ownership system . A variables lifetime begins at initialisation and ends when it is closed or destroyed. This should not be considered variable scope. The borrow-checker uses this concept at compile time to ensure that all references to an object are valid. It is clear that the implementation of memory management of Rust will help in ensuring memory safety, an important factor for the application of Rust within drivers.

2.2 Memory Safety and Vulnerabilities

Memory unsafe languages allow programmers to potentially access memory which is supposed to be outside the bounds of a given data structure (Gaynor, 2019). In the case of data structures, memory unsafe languages allow programmers to access memory which is supposed to outside the bounds of a given data structure. For instance, an array is able to access an element that doesn't exist. This then means that the program fetches whatever happens to be at that position in memory. When this is the case in a memory safe language, an error is thrown which forces the program to crash.

As an example, we can consider a program that manages to-do lists for several users. If implemented in a memory unsafe language, it is possible for the programs data structure to both access negative elements and positive elements that don't exist thus the data structure can access data which is outside of its bounds. This can lead to users having the ability to read each others lists which would then be a security vulnerability in the program, this is known as an 'out-of-bounds read'. If users were able to change elements in other users lists, this is known as an 'out-of-bounds write'. If a to-do list is deleted and later requested then a memory unsafe language has the ability to fetch the memory that it was previously finished with. Within the program, this space might now contain another users list, this is known as a 'user-after-free' vulnerability.

3. Preliminary Work

Work on this project began before the start of the academic year during July and initially continued the plan of the previously written research proposal; 'Investigation into improving performance and reliability of modern device drivers'. A GitHub repository was privately opened in order to store my work, make it available to others and to act as a backup in the case that my workstation broke down. Work began by conducting research into Linux Device Drivers and quantifying findings from the research proposal. Following this, research then continued by exploring the potential application of the Rust programming language within Device Drivers of a range of Operating Systems including Windows, Linux and Apple products.

```rust

use std::io;

▶ Run | Debug
fn main() {
    //print program title
    println!("~~~~~ BMI Calculator ~~~~~");

    //create a new string to handle input
    let mut input: String = String::new();

    //ask for input height, read height to input string, handle error(s) if necessary
    println!("Please enter your height in Metres");
    io::stdin() Stdin
        .read_line(buf: &mut input) Result<usize, Error>
        .expect(msg: "Failed to read line");

    //create a new 64bit float variable for height, cast the input string to a float and store.
    let height: f64 = input.trim().parse().unwrap();

    //REMEMBER TO CLEAR VARIABLES IF THEY ARE USED AGAIN!!!!!!
    input.clear();

    //ask for input weight, read to string, etc
    println!("Please enter your weight in Kg");
    io::stdin() Stdin
        .read_line(buf: &mut input) Result<usize, Error>
        .expect(msg: "Failed to read line");

    //create a new f64 for weight, cast and store string
    let weight: f64 = input.trim().parse().unwrap();

    //print message and call calculation function
    println!("Calculating BMI...");
    calculate_bmi(height, weight);

} fn main

//function calculates user BMI and prints messages based on inputs
fn calculate_bmi(height: f64, weight:f64 ){
   //bmi calculation stored in new float var
   let bmi: f64 = weight / (height * height);

   /*if height or weight is 0, print a notification and restart the program
   /
   /BUG: Using recursion leads to an issue where the `bmi: NaN` shows up after all other messages
   /     I think this is the recursion 'rounding off' or 'finishing up' by returning to the first
   /     loop
   */
   if height == 0.0  || weight == 0.0 {
     println!("Your height or weight was invalid, please try again");
     main();
   }

   //print user BMI
   println!("Your BMI is {:.64}", bmi);

   //check BMI against these standards and print a related message
   if bmi < 18.5{
        println!("You are underweight");
   }else if bmi >= 18.5 && bmi <= 24.9{
        println!("You are normal weight");
   }else if bmi >= 25.0 && bmi <= 29.9{
        println!("You are overweight");
   }else if bmi >= 30.0 && bmi <= 34.9{
        println!("You are obese");
   }else if bmi > 35.0{
        println!("You are extremely obese");
   }
} fn calculate_bmi

```

At this time, the Rust programming language was studied both in theory and practise, a small collection of programs were written to learn Rust on both Linux and Windows machines, one such

example being a BMI Calculator program. This allowed for the learning of basic and fundamental Rust concepts including variable assignment, standard library functions and so on. The study of Linux kernel modules continued which lead to a fundamental understanding in how drivers may be written for an Operating System, especially for Linux. It was possible to learn about how drivers are compiled, how they are written, what libraries are used and how exactly driver software differs from standard application software. Various resources were used from textbooks to online tutorials.

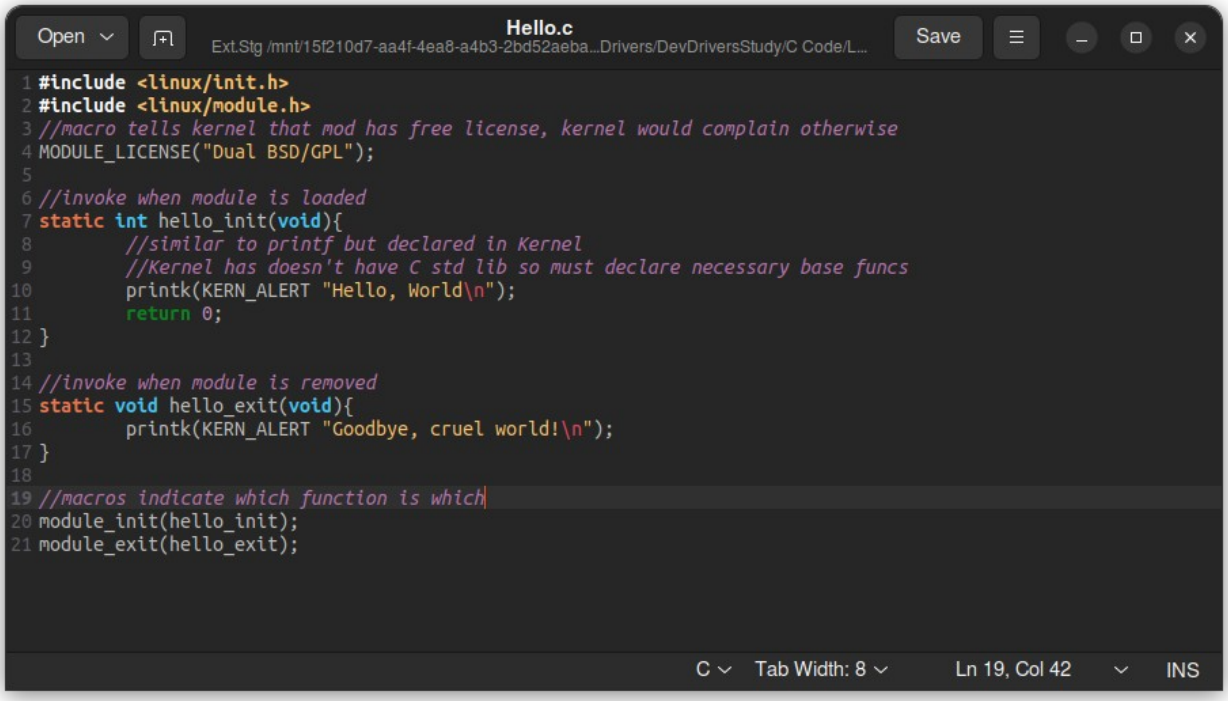

Figure 5: C Code for a 'Hello World' kernel module.

Between August and September 2022, research was continued on relevant literature alongside the previously mentioned studying. From August 19th 2022, regular diaries were kept explained the work carried out, thoughts/notes on the work or topic and anything else which was relevant. These diaries have been up-kept throughout the project, this will continue for the entirety of the project to provide an in-depth record of work carried out, thoughts, theses, notes, justifications and so on. During this time, I also reached out to my supervisor to discuss the project and request their supervision.

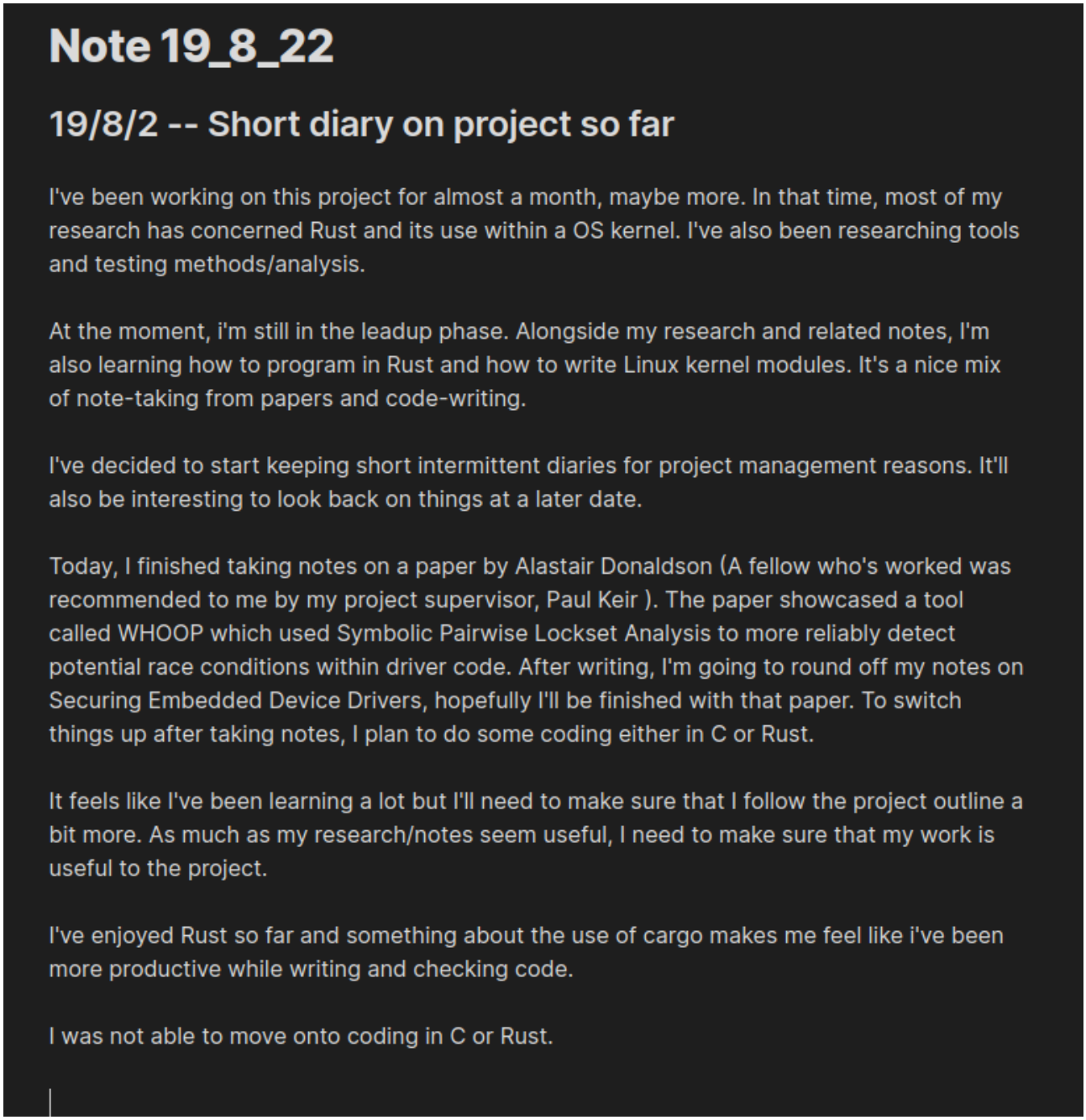

Figure 6: First written Diary produced for project.

Before the University term had commenced, A basic knowledge in the Rust programming language and Linux kernel modules written in Rust was gained. A supervisor had been informally agreed. Research had been conducted on a multitude of papers and topics including Rust driver frameworks, differences in drivers between various Operating Systems, exokernels, writing a device driver using Rust and static analysis tools. Prominent figures in Game development and Software Engineering were contacted who gave their thoughts, best wishes and potential resources.

4. Current progress and Future work

From the start of the academic year, the previously mentioned work was continued. A Raspberry Pi 400 was obtained intended to be used as a driver development workstation with an external hard drive also purchased to extend the Pi's storage capability. This Raspberry Pi machine was configured for Driver Development and previous C drivers were built and tested.

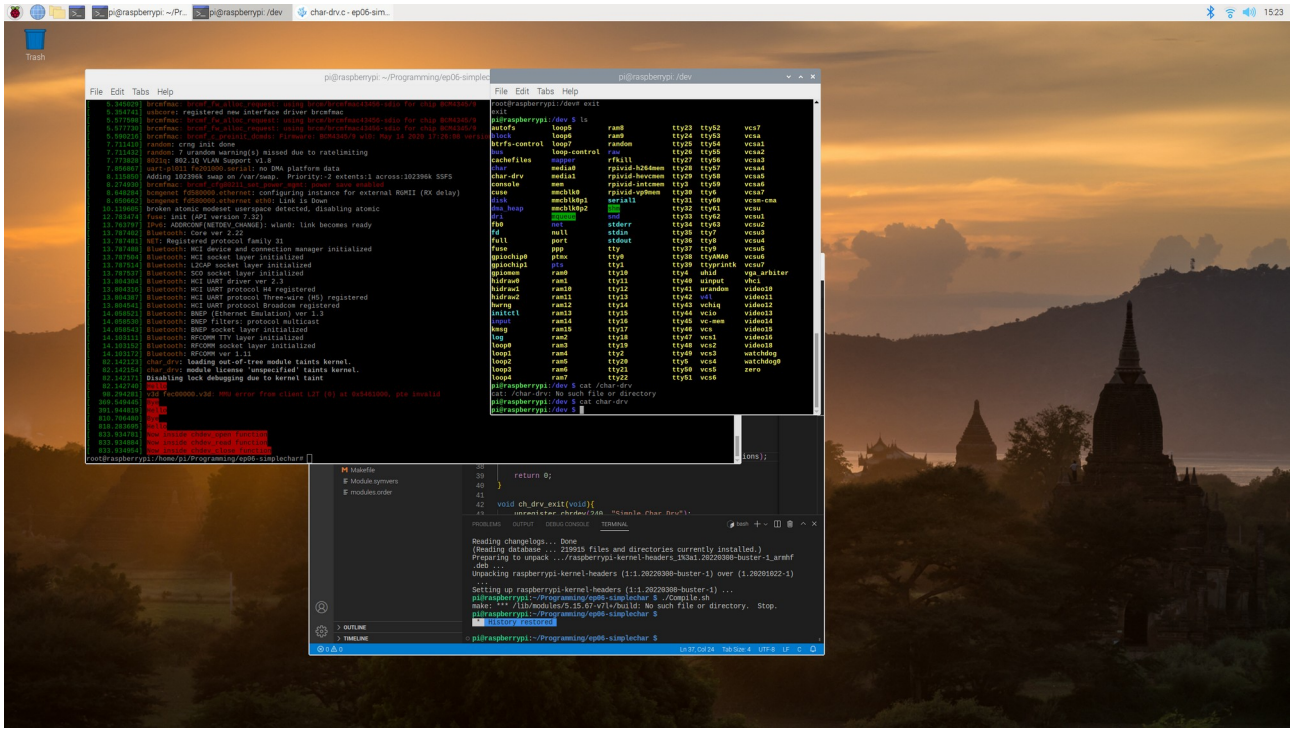

Figure 7: Raspberry Pi with Linux Kernel 5.15.76 v7l+ running character driver from Karthik M tutorials.

Most recently, research has been conducted into using Rust in Linux kernel modules. Several projects have attempted to combine Rust and Drivers but none have been as successful as the Rust for Linux project, which provided the foundation for Rust to be incorporated into the Linux kernel alongside C (Wikipedia, 2022) in Linux kernel 6.1 (Vaughan-Nichols, 2022). With this, research was continued with a focus on the Rust for Linux project. As a result of this, a Linux virtual machine was developed which runs a custom kernel developed by the Rust for Linux team alongside BusyBox. This virtual machine was used to successfully test both a Hello World driver and 'Char' driver which were completely written in Rust.

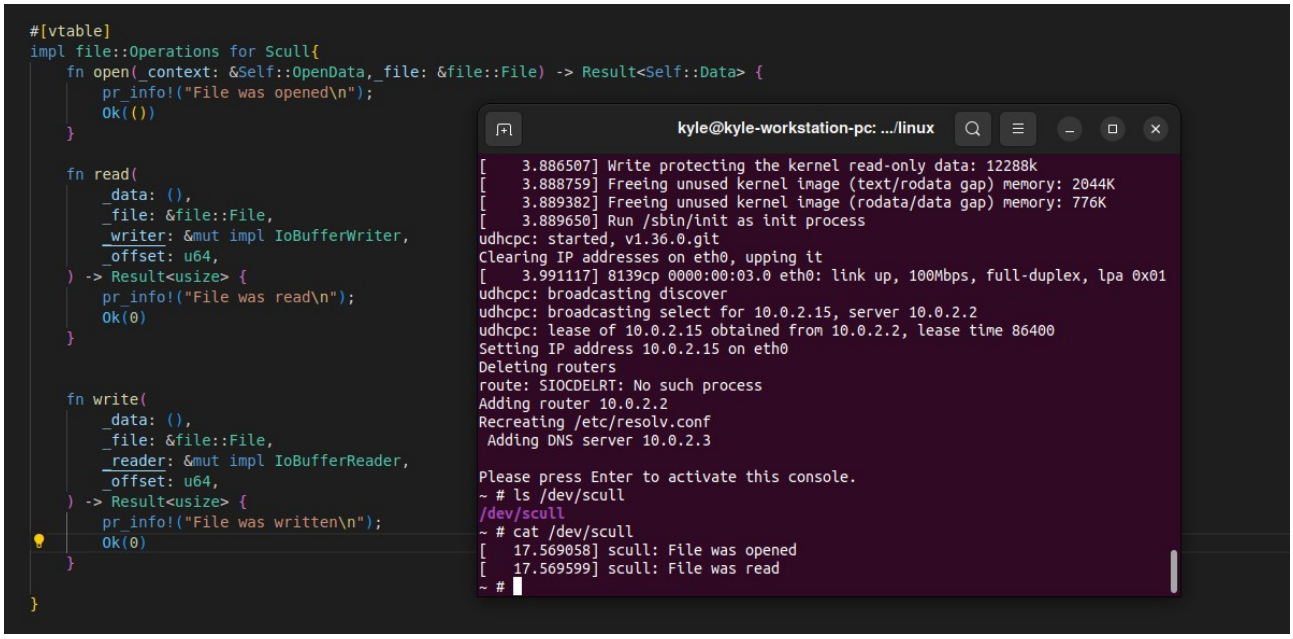

Figure 8: Rust code for file operations of character driver alongside the driver running in a Virtual Machine instance. MAY NEED REPLACED

Figure 9: Rust Hello World kernel module. NEEDS REPLACED

After the successes of this Virtual Machine, it was decided that it would be suitable to install this custom kernel to the Raspberry Pi. In order to safeguard the Pi, a test installation was carried out on a Debian Virtual Machine via VirtualBox where the Linux 6.1-rc3 Kernel was compiled, built and installed. It was believed that the official release candidate would be more suitable for the project as it is part of the official Linux kernel. This test installation was used as an opportunity to learn about compiling and installing a Linux kernel while also acting as an additional test in using Rust within Kernel modules.

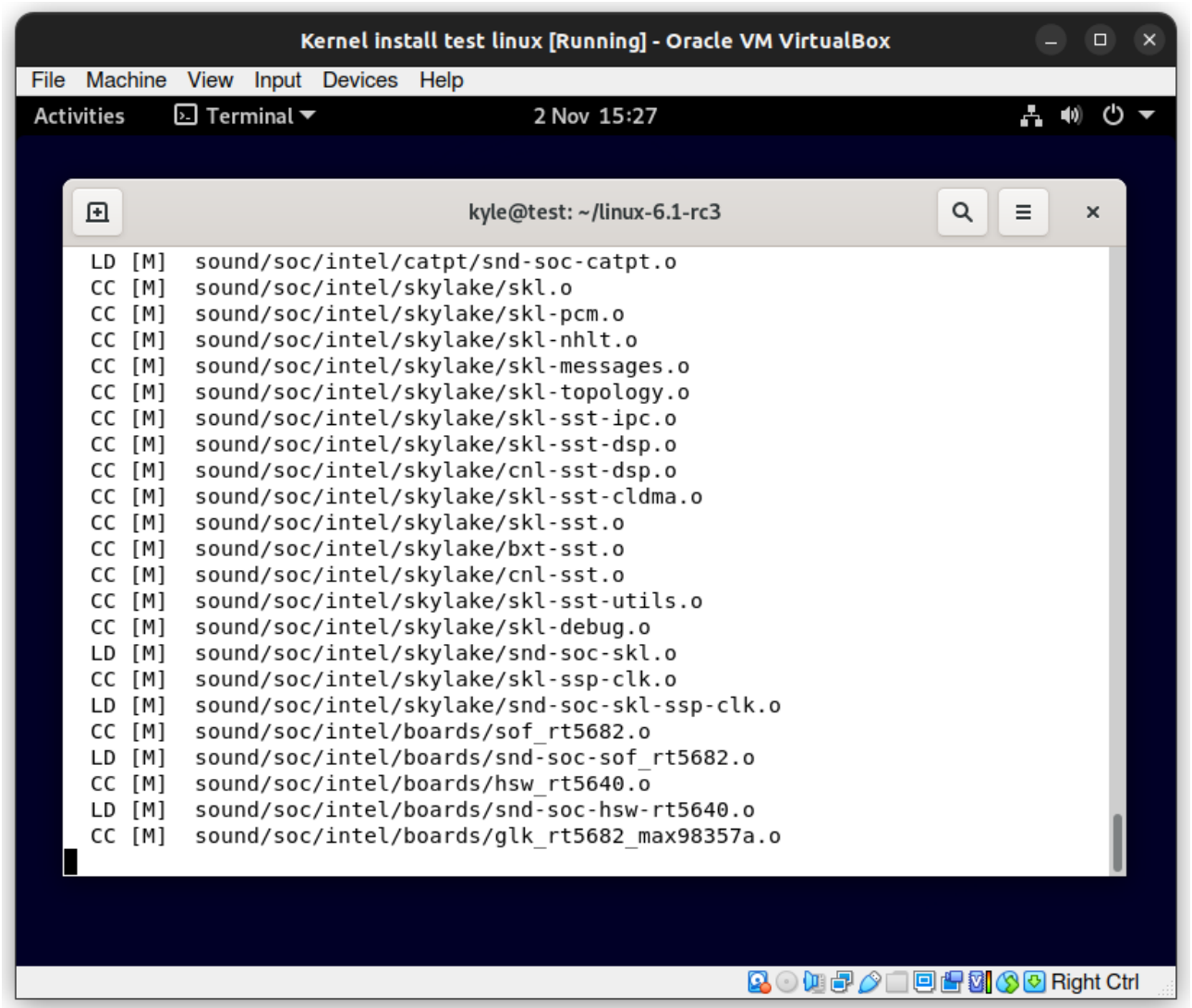

Figure 10: Linux Kernel 6.1-rc3 compiling on Debian Virtual Machine.

This test was less successful. While the 6.1 kernel was successfully installed, this machine was not able to successfully build and compile a Rust kernel module. The main reason behind this was an issue with the driver build system and Rust. It was verified that C kernel modules could build and run with no issue but Rust kernel modules could not build if a specific file is missing.

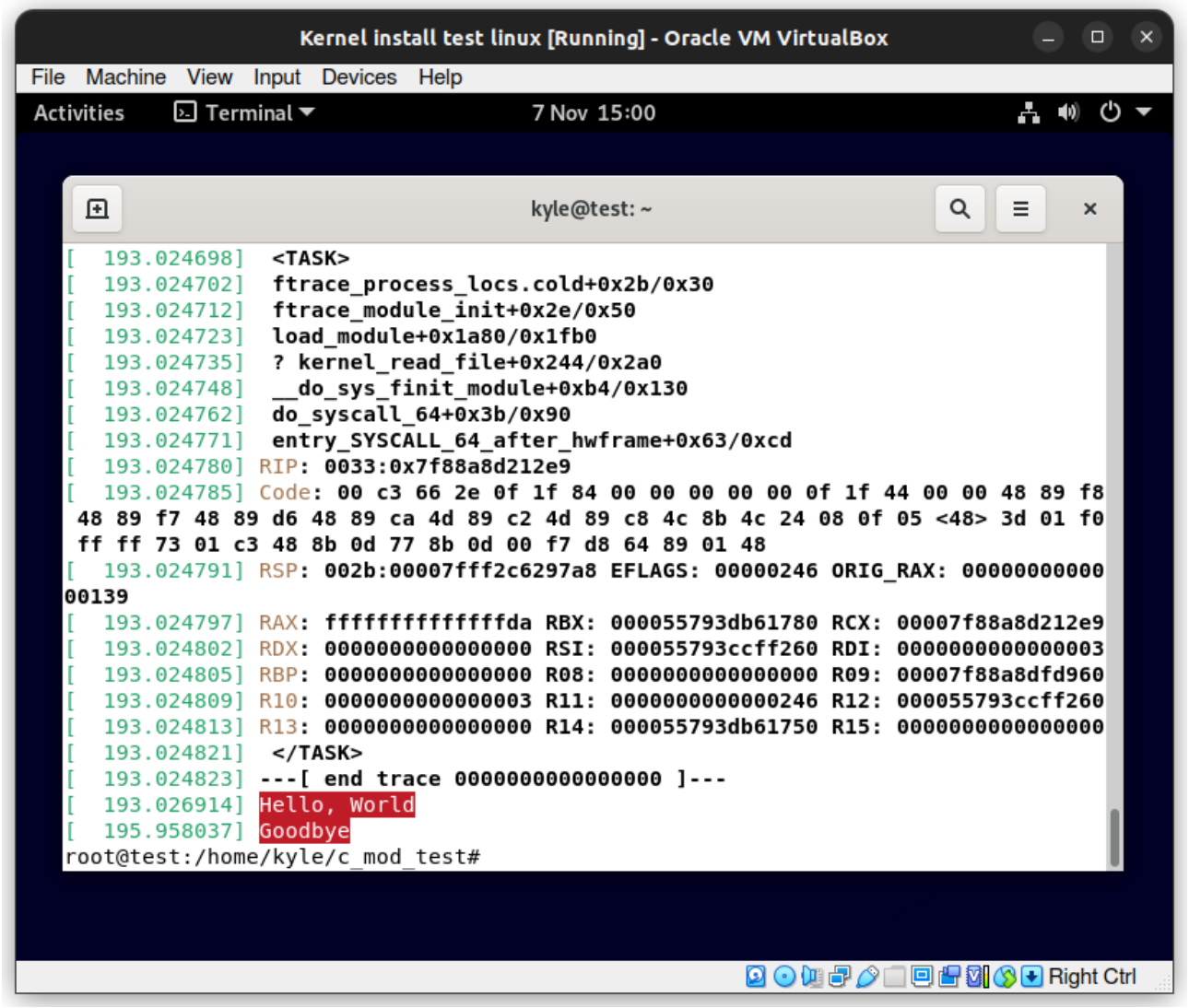

Figure 11: Dmesg utility showing Hello World C driver running on Debian test machine.

To elaborate, as part of the Rust toolchain for drivers, rust-analyzer is used to act as a language server. The main use for this server lies in IDEs. In the case of visual studio code (and its open source counterpart, 'vscodium') it is possible to install the rust-analyzer extension. When installed, this server allows the IDE to read documentation which is inlined to the rust kernel code as comments. This means that the developer can easily read relevant documentation from within the IDE and does not need to refer to an external document or attempt to read through code comments. In order to facilitate this, rust-analyzer relies on a JSON file which is typically produced via make but on the Debian machine this JSON file could not be produced. The rust-analyzer alongside 'rustdoc' and rust tests were not able to run or build via make and attempting to do so results in

errors. As a result of the rust-analyzer problem the driver build system would not allow rust drivers to build as the JSON file was not present.

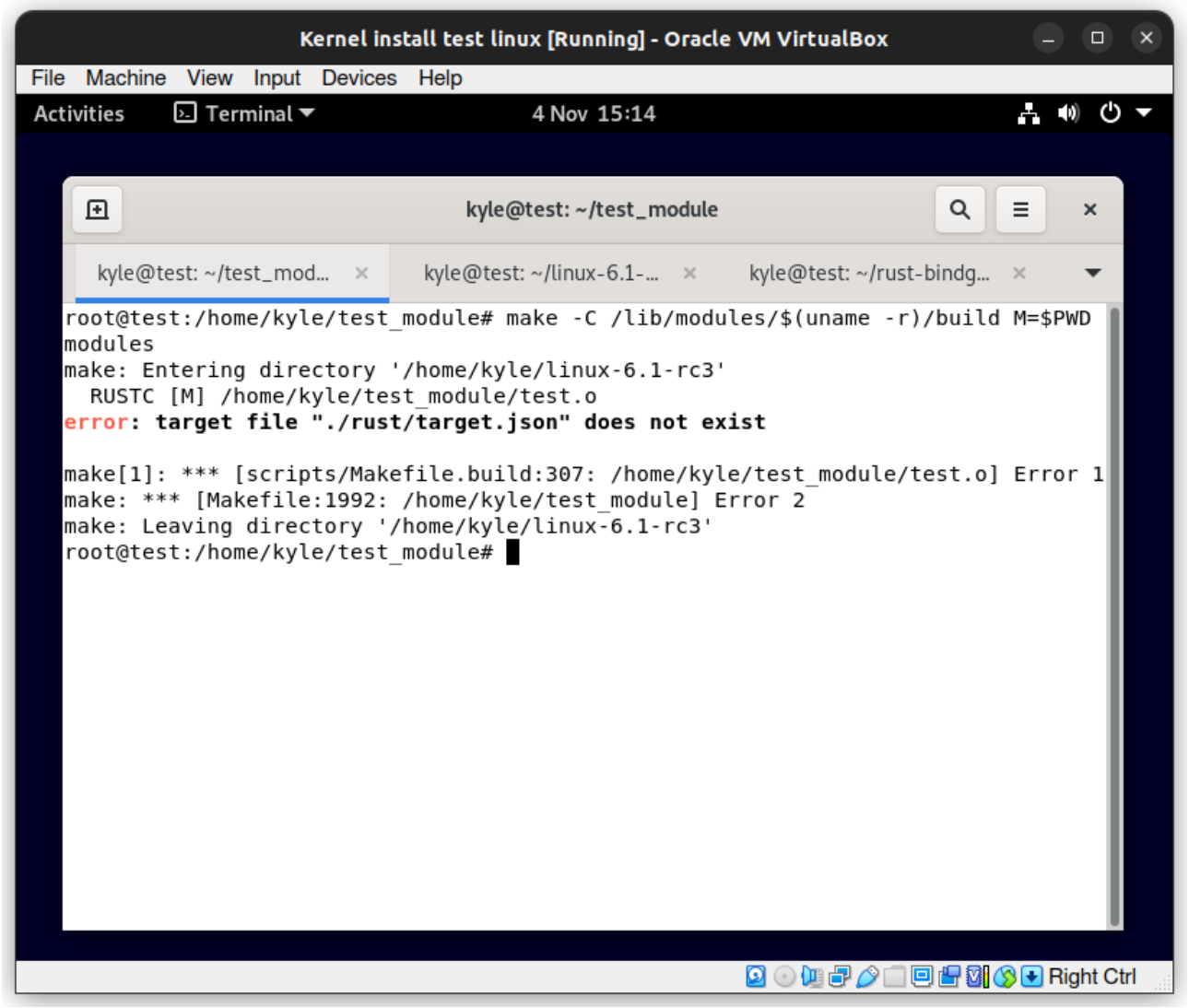

Figure 12: Make throwing error as rust-analyzer file doesn't exist.

Another virtual machine test was carried out with Ubuntu. The ‘torvalds/linux’ repository was used to easily fetch new updates via git. This test was a success with all previous problems alleviated and solved. The underlying reason for the previous issue was a missing command ‘`make defconfig`’ which was necessary to enable Rust support. A test build of a ‘Hello, World’ driver was carried out where it was discovered that C will likely need to be used alongside Rust. Sample Rust code was also tested and successfully executed.

In future it is planned to use the Raspberry Pi to develop a Rust driver. The specific driver has not yet been decided but there is hope that a driver for a physical device (such as a mouse, USB stick and so on) may be written. It should be noted that in the event that the Pi can't be used or there are problems with Rust in the 6.1 kernel then it is planned to simply use the working BusyBox Virtual Machine and develop drivers for 'virtual' devices similar to that of 'Scull' (an example contained within Linux Device Drivers 3). Regardless, as part of this work, it will be necessary to conduct research into the various libraries currently provided by the Linux kernel, and how they are utilised within a device driver.

References

Ball, T. Bounimova, E. Cook, B. Levin, V. Lichtenberg, J. McGarvey, C. Ondrusek, B. Rajamani K, S. Ustuner, A. (2006) Thorough static analysis of device drivers [Online] ACM. Available: https://dl.acm.org/doi/abs/10.1145/1218063.1217943 [Accessed 14 October 2022]

Claburn, T. (2022) "In Rust We Trust: Microsoft Azure CTO shuns C and C++". [Online] The Register. Available: https://www.theregister.com/2022/09/20/rust_microsoft_c/ [Accessed 21 September 2022]

Corbet, J. Rubini, A. Kroah-Hartman, G. (2005) "Linux Device Drivers". 3rd ed. O'Reilly Media, Inc. [Accessed 23 July 2022]

Gaynor, A. (2019) "Introduction to Memory Unsafety for VPs of Engineering". [Online] Available: https://alexgaynor.net/2019/aug/12/introduction-to-memory-unsafety-for-vps-of-engineering/

Li, Z. Wang, J. Sun, M. Lui, C.S. J. (2019) "Securing the Device Drivers of Your Embedded Systems: Framework and Prototype" [Online] ARES '19: Proceedings of the 14th International Conference on Availability, Reliability and Security. Article 71. pg 1 to 10. Available: doi.org/10.1145/3339252.3340506 [Accessed 8 August 2022]

Karthik M. (2014) Linux Device Drivers Training 06, Simple Character Driver 24.54 mins. [Online] Available: https://www.youtube.com/watch?v=E_xrzGlHbac [Accessed 30 August 2022]

Klabnik, S. (2016) "The History of Rust". ACM. [Online] Available: https://youtu.be/79PSagCD_AY [Accessed 23 October 2022]

Klabnik, S. Nichols C. and Rust Community. (2022) "The Rust Programming Language" [Online] Available: https://doc.rust-lang.org/book/ [Accessed 3 August 2022]

Oatman, T. (No Boilerplate) (2022) "Stop writing Rust (and go outside and play!) [RUST-5]" 9.40mins. [Online] Available: https://youtu.be/Z3xPIYHKSoI [Accessed 3 August 2022]

Palix, N. Thomas, G. Saha, S. Calves, C. Lawall, J. Muller, G. (2011) "Faults in Linux: Ten Years Later"[Online] ACM SIGPLAN Notices, Volume 46, Issue 3. pg 305 to 318. Available: https://dl.acm.org/doi/10.1145/1961296.1950401 [Accessed 5 October 2022]

Prossimo (2022) "What is memory safety and why does it matter?" [Online] Internet Security Research Group. Available: https://www.memorysafety.org/docs/memory-safety/ [Accessed 14 October 2022]

Renzelmann J, M. Swift M, M. (2009) Decaf: moving device drivers to a modern language [Online] University of Wisconsin-Madison. Available: https://static.usenix.org/events/usenix09/tech/full_papers/renzelmann/renzelmann.pdf [Accessed 23 February 2022]

Ritchie, M. D. (1993) "The Development of the C Language" [Online] Bell Labs, Lucent Technologies. Available: https://www.bell-labs.com/usr/dmr/www/chist.html [Accessed 14 October 2022]

Rust Community (n.d.) "The Cargo Book" [Online] Available: https://doc.rust-lang.org/cargo/index.html [Accessed 23 October 2022]

Sasidharan, K. D. (2020) "Visualizing memory management in Rust". [Online] Technorage. Available: https://deepu.tech/memory-management-in-rust/ [Accessed 28 October 2022]

Thomas, G. Gaynor, A. (2019) "Linux Kernel Modules in Rust". [Online] Linux Security Summit. Available: https://youtu.be/RyY01fRyGhM [Accessed 1 August 2022]

University of Washington, Department of Computer Science and Engineering (2004) CSE 341: Unsafe languages (c) [Online] Available: https://courses.cs.washington.edu/courses/cse341/04wi/lectures/26-unsafe-languages.html [Accessed 14 October 2022]

Wikipedia (2022) "Memory safety" [Online] Available: https://en.wikipedia.org/wiki/Memory_safety [Accessed 14 October 2022]

Wikipedia (2022) "Rust for Linux" [Online] Available: https://en.wikipedia.org/wiki/Rust_for_Linux [Accessed 8 November 2022]

Vaughan-Nichols, S. (2022) "Linus Torvalds: Rust will go into Linux 6.1". [Online]. zdnet. Available: https://www.zdnet.com/article/linus-torvalds-rust-will-go-into-linux-6-1/ [Accessed 2 October 2022]

Appendices

COMPUTING HONOURS PROJECT SPECIFICATION FORM
(Electronic copy available on the Aula Computing Hons Project Site)

Project Title: Developing Device Drivers in Rust

Student: Kyle Fraser Christie **Banner ID:** B00415210

Programme of Study: BSc (Hons) in Computing Science

Supervisor: Paul Keir

Moderator: Stephen Devine

Outline of Project:
Device Drivers within Operating Systems suffer from numerous issues, one of which being the use of unsafe programming languages. This project seeks to test if Rust would be a suitable, safe candidate to replace the C programming language in drivers, exploring exactly how this task might be carried out.

Rust is a young systems programming language with a focus on safety through various features. From its compiler to its model of memory management. It is my intent to attempt to write a Linux driver in Rust to test its suitability for applications within drivers and the wider world of Operating System and Kernel development. Doing so would allow me to highlight where Rust may make improvements or prevent errors when compared to that of C.

During the project, I also want to gain a fundamental understanding of drivers in their present state. I plan to investigate how they are developed, uses, tools and differences between major Operating System vendors. How they link to the Kernel and rest of the OS. I want to utilise this project to thoroughly investigate and highlight the issues surrounding device drivers, discussing previous works that have attempted to alleviate driver issues.

Rust continues to grow in popularity and more developers call for it to replace C and C++. It is therefore necessary to test its suitability for integration into existing systems and to test if Rust truly has potential to eventually replace such programming languages.

A Passable Project will:

Review literature on Device Drivers and related issues.
Discuss previous works, other methods/ideas
Highlight how Rust can be used to improve applications and safety
Develop a basic Rust driver.
Evaluation of Rust driver with explanation and results.
Discuss recent works that relate or will progress the field.

A First Class Project will:

1. Provide detailed literature review highlighting most prominent, key issues of topic at hand which form the basis of this project.
2. Discuss previous research efforts, results and findings – how they relate to and inform the project work.
3. In-depth technical discussion on Rust's security features, how these compare to that of C/C++ and areas where improvements may have been made.
4. Develop a working Rust driver that can control (or give basic functionality to) an external computer peripheral (Mouse, USB stick, Simple gamepad/controller).
5. Detailed evaluation of end product and comparison to an existing equivalent in C.
6. Detailed discussion on recent findings, work, events that strongly relate to the project in order to highlight future opportunities, research and potential developments.

Reading List:

Corbet, J. Rubini, A. Kroah-Hartman, G. (2005) *Linux Device Drivers 3rd Edition*

Li, Z. Sun, M. Wang, J. C.S. Lui, J. (2019) *Securing the Device Drivers of Your Embedded Systems: Framework and Prototype*

Gaynor, A. Thomas, G. (2019) *Linux Kernel Modules in Rust*

Choum A, Yang, J. Chelf, B. Hallem, S. Engler, D. (2001) *An Empirical Study of Operating Systems Errors*

Palix, N. Calves, C. Thomas, G. Lawall, J. Saha, S. Muller, S. (2011) *Faults in Linux: Ten Years Later*

Chen, H. Mao, Y. Wang, X. Zhou, D. Zeldovich, N. Kaashoek M. F. (2011) *Linux Kernel vulnerabilities: State-of-the-art defenses and open problems*

Resources Required: *(hardware/software/other)*

Personal workstation with Linux OS installed
Raspberry Pi 400 (used as development/test machine)
VirtualBox (as backup for driver development)

RUSTC Compiler
VSCodium Open Source Code Editor
Git
GitHub
Trello (project/task management)

Marking Scheme:

	Marks
Background	10
Literature Review	20
Development	40
Experiments	10
Conclusion	10
Reflection	10

AGREED:

Student	**Supervisor**	**Moderator**
Name: Kyle Fraser Christie	**Name:** *Paul Keir*	**Name:** *Stephen Devine*

IMPORTANT:

By agreeing to this form all parties are confirming that the proposed Hons Project will include the student undertaking practical work of some sort using computing technology / IT, most frequently achieved by the creation of an artefact as the focus for covering all or part of an implementation life-cycle.

By agreeing to this form all parties are confirming that any potential ethical issues have been considered and if human participants are involved in the proposed Hons Project then ethical approval will be sought through approved mechanisms of the School of CEPS Ethics Committee.