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

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 langages 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 succesfully 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+.

Figures 1 and 2 demonstrate execution of a character driver from the Linux Device Drivers Training series by YouTube Channel 'Karthik M' (Karthik M, 2014). Figure 3 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.

Cargo is the build tool and package manager for the Rust language and is reponsible 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 immediatley 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).

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.

2.3 Previous works

2.3.1 Apple restructures Kernel extensions

2.3.2 Driver Frameworks

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 programmming language within Device Drivers of a range of Operating Systems including Windows, Linux and Apple products.

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.

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 upkept 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.

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

References