

**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

[Abstract 4](#__RefHeading___Toc510_1174827738)

[1. Background 5](#__RefHeading___Toc432_2946013826)

[1.1 Device Drivers 5](#__RefHeading___Toc434_2946013826)

[1.2 Input from Industry 6](#__RefHeading___Toc436_2946013826)

[1.3 Project goal 7](#__RefHeading___Toc438_2946013826)

[1.4 General Concepts 7](#__RefHeading___Toc440_2946013826)

[1.4.1 Kernel 7](#__RefHeading___Toc442_2946013826)

[1.4.2 User space and Kernel space 7](#__RefHeading___Toc444_2946013826)

[1.4.3 Device node system 8](#__RefHeading___Toc446_2946013826)

[1.4.4 Build system 8](#__RefHeading___Toc448_2946013826)

[1.4.5 Makefile 8](#__RefHeading___Toc450_2946013826)

[1.4.6 Development 8](#__RefHeading___Toc452_2946013826)

[2. Literature Review 9](#__RefHeading___Toc454_2946013826)

[2.2 Operating System Drivers 9](#__RefHeading___Toc456_2946013826)

[2.2.1 Linux 9](#__RefHeading___Toc458_2946013826)

[2.2.2 Apple 9](#__RefHeading___Toc460_2946013826)

[2.2.3 Windows 10](#__RefHeading___Toc462_2946013826)

[2.2.4 FreeBSD 10](#__RefHeading___Toc464_2946013826)

[2.3 Rust 11](#__RefHeading___Toc466_2946013826)

[2.3.1 Rust for Linux 12](#__RefHeading___Toc468_2946013826)

[2.3.2 Criticisms 12](#__RefHeading___Toc470_2946013826)

[2.3.3 Google & Android 13 13](#__RefHeading___Toc472_2946013826)

[2.4 Memory Safety 15](#__RefHeading___Toc474_2946013826)

[2.4.1 Garbage Collection 16](#__RefHeading___Toc579_2946013826)

[2.5 The Exo-kernel 17](#__RefHeading___Toc476_2946013826)

[3. Development 18](#__RefHeading___Toc581_2946013826)

[4. Experiments 18](#__RefHeading___Toc583_2946013826)

[Conclusion 18](#__RefHeading___Toc585_2946013826)

[Reflection 18](#__RefHeading___Toc587_2946013826)

[References 18](#__RefHeading___Toc589_2946013826)

[Appendices 18](#__RefHeading___Toc591_2946013826)

Table of Figures

[Figure 1: Debian Virtual Machine (Linux kernel 4.19.0-17-amd64 running character driver from Karthik M tutorials.) 6](#Figure!0|sequence)

[Figure 2: System layer break down of Linux (Wikipedia, 2022) 7](#Figure!1|sequence)

[Figure 3: Memory safety error statistics within major codebases (Gaynor, 2020) 15](#Figure!2|sequence)

# Abstract

# 1. Background

## 1.1 Device Drivers

Device drivers are a vital component of Operating Systems which allow for the control of peripheral devices while interacting with underlying hardware. Drivers also provide facilities which can be used to extend an Operating System via file systems, network protocol, anti-virus capability and more (Ball et al, 2006). Described as the "software layer that lies between applications and physical devices" (Corbet et al, 2005), drivers are clearly a necessity within an Operating System however they suffer from a range of issues with dangerous consequences.

Drivers continue to be programmed with the C programming language which 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). Languages such as 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 found within various programming languages with the aim of preventing the developer from introducing certain bugs which strongly relate to memory management (Prossimo, 2022). Memory safety issues usually lead to security problems with common vulnerabilities being out-of-bounds reads, out-of-bounds writes and use-after-frees (Gaynor, 2019).

Furthermore, Linux 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 were 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+. It is therefore clear that device drivers have not evolved in any significant way. Code which targets Linux kernel version from over a decade ago continues to run on more recent versions.

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

## 1.2 Input from Industry

During the project, prominent or relevant figures within Game Development and Software Engineering were contacted. These include Jonathan Blow (developer of Braid, The Witness and the Jai programming language) Dave Plummer (former Microsoft engineer, now entrepreneur, best known for his work in creating Windows Task Manager among other projects), Alex Gaynor (Software Resilience Engineer working on the Rust-For-Linux project), Miguel Ojeda (Software Engineer working on the Rust-For-Linux project) and Asahi Lina (Software Engineer working on a Linux GPU driver in Rust for the Apple M1).

Blow fed back that drivers are a good problem to consider though felt that the project was not ambitious enough. He spoke of a concept called an 'Exo-kernel model' which he feels would fix a lot of device driver problems as drivers are simply considered 'normal' code. He also feels that the field is held back as the basic idea of what an operating system should be has already been figured out, Blow seems to think there is little innovation in the field. Alongside his feedback, he also provided me with a USENIX ATC Keynote; 'It's Time for Operating Systems to Rediscover Hardware'.

Plummer explained that he had mostly worked on cache software and filesystems but had never really worked on drivers unless it was necessary. He recommended a textbook, 'Windows 7 Device Driver' by Ronald D. Reeves Ph.D. and gave his best wishes.

Gaynor praised the project, saying "Building a USB mouse driver with Rust for Linux sounds like a great project", and outlined the approach used by the RustForLinux community when writing Drivers which is as follows;

1. Check if RustForLinux already has existing APIs for the relevant kernel subsystem.
2. If not, they design a safe Rust API that exposes the original kernel APIs.
3. It is then possible to use these new abstractiosn to write the original driver.

Gaynor mentioned a pull request for the Rust-For-Linux repository which aims to add support for USB device drivers. After a short correspondence, Ojeda provided some advice and useful resources. He recommended getting used to reading C code within kernel. With regards to the Rust driver, his recommendation was to write a C version which can then be referenced for the Rust driver, this will also help in learning kernel APIs and verify whether issues are a result of Rust support or otherwise.

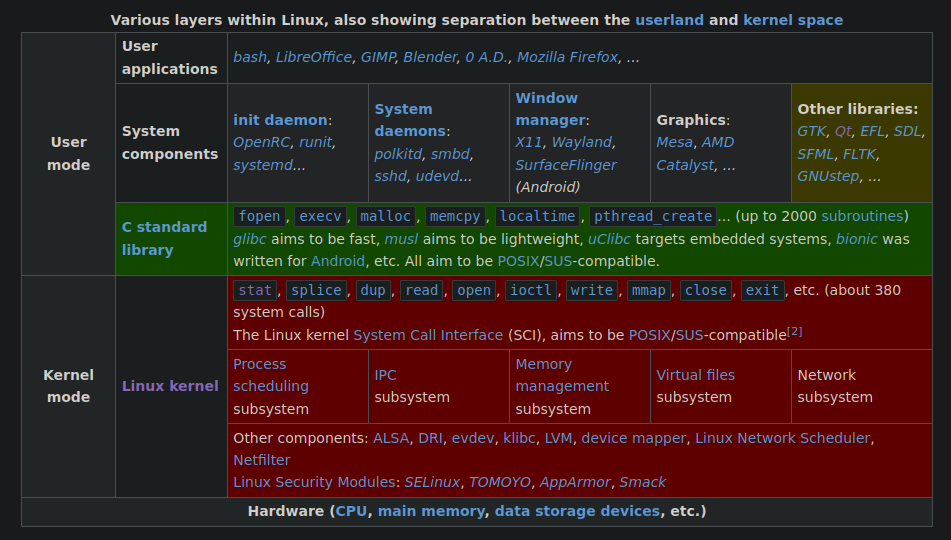
## 1.3 Project goal

The aim of this project is to try and overcome the previously highlighted issues by developing a Linux device driver in Rust. Not only will it replace C, Rust and its features should prevent issues with memory safety. Rust 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 from 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).

## 1.4 General Concepts

### 1.4.1 Kernel

A kernel is the primary interface between hardware and computer processes, ensuring resources are used as effectively as possible (Baeldung, 2022). The kernel runs within the operating system and controls the function of hardware alongside managing memory and computer peripherals.

Figure 2: System layer break down of Linux (Wikipedia, 2022)

### 1.4.2 User space and Kernel space

Kernel space is an area of memory used exclusively by the kernel and encapsulates device drivers (Wikipedia, 2022). User space is a separate area where user applications run and file systems can be managed (Baeldung, 2022). User applications communicate with the kernel via system calls. Kernel and user space is separated to protect memory and protect the hardware layer of the system as showcased in Figure x. The term 'space' is interchangeable with the term 'mode' as this concept is also related to processors.

### 1.4.3 Device node system

Device nodes are special file types (especially in Unix-based systems) which represent a resource allocated by the kernel. These resources are identified by a major number and minor number which are both stored within the structure of the node. Typically, the major number identifies the device driver while the minor number identifies a specific device (or collection of devices) that the driver can control with these numbers being passed to the driver.

### 1.4.4 Build system

Building refers to the process of converting (or translating) source code into executable binary files. Thus, a build system is "a collection of software tools used to facilitate the build process" (Zhang, 2020). Build systems have been used for over 30 years and have not seen major change. There are several build systems that can be used in the present including: Make, GNU Make, CMake, QMake and Ninja.

### 1.4.5 Makefile

A makefile controls the build system, 'make', and "describes the relationships among files in your program and provides commands for updating each file." (2022, Free Software Foundation, Inc.). The makefile contains a pre-written set of rules which control how make compiles and links a program. It is also possible to run various external commands via make, a clean-up operation as an example. When the makefile is run, it typically builds a piece of software, even updating specific files when additions are made.

### 1.4.6 Development

Driver development can occur across 2 different machines. As is the case in Windows and, previously, Apple systems. In such scenarios, the debugger and driver are run on separate computers (Microsoft, 2022). The computer running the debugger can be known as the 'host' machine while the computer running the driver can be known as the 'target' or 'test' machine.

# 2. Literature Review

## 2.2 Operating System Drivers

### 2.2.1 Linux

Linux utilises 'kernel modules' to implement most device drivers, these kernel modules are primarily written in C (Corbet et al, 2005). These kernel modules make use of kernel space, which is separated from user space though the kernel space itself is not compartmentalised. Kernel modules are typically compiled with makefiles. These convert the code into an object file by calling the kernel build system to link code and object files to '.ko' executables. Linux makes several commands available to add, remove and manage kernel modules which is one of the primary methods of testing custom built device drivers.

Kernel space is separated from user space by the CPU, the Linux kernel runs in the highest CPU level so there is little restriction on what a driver can do this also means that faults that occur within kernel space have a high likelihood of affecting the overall system. Kernel modules can be run in user space but the associated performance issues mean doing this is worthless.

### 2.2.2 Apple

Apple device drivers encapsulate iPad, iMac and iPhone. Apple recently re-structured their device driver technology, 'Kernel extensions' to implement additional driver types known as 'Driver extensions' and 'System extensions' (Auricchio et al, 2019). The reason behind the change being that Apple developers found several issues with kernel extensions; They are difficult to develop and debug and also pose a risk to data security, privacy and overall reliability. System and driver extensions improve on kernel extensions and thus are easier to develop while improving security and reliability.

A system extension is similar to a kernel extension but is instead a component of an application. They are intended to implement features that need kernel level co-operation, such as custom network behaviour (Apple Developer Documentation, 2022). These extensions run in user space and contain 3 types: Network; Endpoint Security (including Anti-virus and Data loss protection); Driver (USB, Serial, NIC and HID). System extensions can use any framework within the macOS SDK as well as any programming language. System extensions are said to be much easier to debug as they do not pause the kernel, there is no need to restart the machine and it is possible to build, test and debug on a single machine with full debugger support.

Developers also found that the Kernel is an unforgiving environment (Auricchio et al, 2019). That writing and debugging kernel code is difficult, kernel extensions need 2 machines in order to debug which introduces overhead and only has limited debugger support and that kernel extensions are a great risk to security as successful attackers can gain free reign in the kernel while any bug in a kernel extension could also be a critical reliability problem. Kernel extensions only support the C and C++ programming languages.

### 2.2.3 Windows

Windows also implements the concept of user mode and kernel mode. Microsoft documentation lists various driver types including: Function driver (which communicates directly with the device); Filter driver (which performs auxiliary processing) and Software driver (used when desktop software needs to access something in kernel space). Windows also provides different frameworks which can be used when writing device drivers through Visual Studio such as the User Mode Driver Framework (UMDF) and Kernel Mode Driver Framework (KMDF). Device drivers are still written in C but are usually categorised under C++.

Windows drivers seem to work similarly to those of Linux. Drivers are built into a '.dll' file with accompanying files including '.inf' (which stores driver information). Depending on the driver type, it may be built into a '.sys' file with a '.cat' file which is used by during installation to verify the drivers signature.

### 2.2.4 FreeBSD

FreeBSD drivers are similar to that of Linux and makes use of similar concepts. Drivers can be statically compiled into the Operating System or loaded via the dynamic kernel linker facility. FreeBSD makes use of the Unix device node system, known as '/dev' which is also found in Linux. FreeBSD makes use of KLD to dynamically manage and extend the kernel without rebooting, which is very similar to Linux kernel module commands. 'kldload', 'kldunload' and 'kldstat' are examples of commands used in managing drivers. FreeBSD makes use of 'pseudo-devices' where a driver emulates the behaviour of a device in software without hardware. Drivers on FreeBSD are split into two categories: Character and Network. Character devices are typically used to directly transfer data between a user process and other processes, they are the most common type of driver.

## 2.3 Rust

Rust is a "compiled, concurrent, safe, systems programming language" (Klabnik, 2016) which was 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.

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) thus highlighting errors with clear feedback and potential solutions which prevents developers from making common mistakes (Klabnik, 2016). This feature is critically important to drivers, it was previously established that writing device drivers is no easy task as developers previously struggled to work with the Windows XP driver API (Ball et al, 2006). It has also 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. The compiler also forces the developer to handle errors.

Rust is reliable as code is backwards compatible; ensuring old code is always able to compile with new versions of the compiler (Oatman, 2022). Another benefit of this is that old code will continue to benefit from optimisations made to the rust toolchain. Code of all ages will improve and speed up alongside the language itself. A further benefit 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.

As shown in Figure 4, Rust has no defined memory model, utilising simple memory structures comparable to that of JVM, Go and C++11. 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 a 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 to 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 which means there is no cost to performance or other miscellaneous 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. The copy function 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 own 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 resource can only ever have one owner at a time, variables can be passed by value or by reference and the Rust borrow-checker enforces ownership rules and ensures references make use of valid objects. (Sasidharan, 2020).

Variables have lifetimes, a concept which is important for the functionality of the ownership system. A variable's 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 Rusts implementation of memory management will no doubt help in ensuring memory safety, an important factor for the application of Rust within drivers.

### 2.3.1 Rust for Linux

Rust for Linux is a project, originally started in 2019 by Miguel Ojeda with the aim of introducing a new system programming language into Linux kernel. Rust would be chosen as it "... guarantees no undefined behaviour takes place (as long as unsafe code is sound), particularly in terms of memory management." (Ojeda, 2022) which would eliminate issues such as use-after-free, double free and data races. The project was created as there had long been desire to write Linux kernel code in Rust. Several attempts were made, the earliest being in 2013, though none of these projects provided Rust support from within the kernel.

There has since been various technical achievements within the project with several organisations from industry approaching Ojeda with interest including Google, Arm, Microsoft and Red Hat as well as private companies. Alongside these companies, academics have also reached out such as researchers at the University of Washington. Work carried out by the Rust for Linux project was recently integrated into the Linux kernel, starting from version 6.1, marking the first time a new programming language has successfully been introduced into the kernel.

### 2.3.2 Criticisms

Creator of C++, Bjarne Stroustrup, has previously criticised Rust and similar memory safe languages as "every safe language, including Rust, has loopholes allowing unsafe code" (Claburn, 2022) however in Googles security blog, the use of unsafe is described as an "... escape hatch which allows interacting with system resources and non-Rust code." (Vander Stoep, 2022). Klabnik also previously noted that " fundamentally the machine definitely is not immutable by default ..." therefore unsafe is important to the language (Klabnik, 2016).

The comparison could be likened to how I/O is implemented in Haskell, a purely functional language where no functions allow for the use of traditional I/O and there are no global variables. Haskell, similarly to Rust and 'unsafe', implements several backdoors for the sake of functionality within the language yet Haskell continues to be used doesn't seem to have met such criticism as Rust has.

It would seem that while the inclusion of 'unsafe' could potentially lead to issues within a safe programming language, it is ultimately deemed a necessity which allows such languages to be used within system environments and permit interactions with other languages while also considering how the hardware itself works. There are other languages, namely Haskell, that have similarly implemented necessary backdoors. However, it should be considered that the use of unsafe requires the developer to be responsible for safety. Stroustrups concerns may be valid although the implementation of unsafe seems to be a necessary trade-off for the language to function as intended.

### 2.3.3 Google & Android 13

Android 13 has recently seen a significant drop in memory safety vulnerabilities and an associated drop in vulnerability severity with the annual number of memory safety vulnerabilities dropping from 223 to 85 between 2019 and 2022 (Vander Stoep, J. 2022). Memory safety vulnerabilities now account for 35% of Androids total vulnerabilities (previously 76%) with 2022 being the first year where the majority of vulnerabilities are not related to memory safety. This drop coincides with a move away from memory unsafe programming languages with Android 13 being "the first Android release where a majority of new code added to the release is in a memory safe language".

Rust was announced in Android 12 as an alternative to C and C++ with the goal being to shift development of new code to memory safe languages over time. Now, in Android 13, 21% of all new native code is written Rust with approximately 1.5 million total lines of Rust found within Android Open Source Projects across a handful of new features. Google found that "To date, there have been 0 memory safety vulnerabilities discovered in Androids Rust code." it is not expected for this number to remain 0 but is a significant result which suggests that Rust is fulfilling is intended purpose in preventing Androids most common source of vulnerabilities. It's believed that 'it's likely that using Rust has already prevented hundreds of vulnerabilities from reaching production'.

Google also found that the use of Rust allows optimisation of both security and system health with fewer compromises as safety measures typically slow memory-unsafe languages. This usually means developers must make trade-offs between security and performance in adding sandboxing, sanitizers, runtime mitigations, hardware protections which negatively impact code size, memory and performance. It was also found that when compared to other vulnerabilities (which have a well defined scope of impact) Memory safety vulnerabilities are much more versatile. If code execution is obtained in a process, not only is access granted to the specific resource but to everything that the process can access which provides an attack surface to other processes. "Memory safety vulnerabilities are often flexible enough to allow chaining multiple vulnerabilities together", it was found that the majority of exploit chains abused in Google products use one or more safety vulnerability. Due to the decrease in severe vulnerabilities, there has been an increase in less severe types with around 15% of 2022 vulnerabilities being Denial of Service vulnerabilities which represents a drop in security risk.

## 2.4 Memory Safety

Memory unsafe languages allow programmers to potentially access memory which is supposed to be outside the bounds of a given data structure (Gaynor, 2019). This is even more detrimental as memory safety vulnerabilities consistently account for the highest percentage of vulnerabilities within large codebases as showcased in the figure below.

|  |  |
| --- | --- |
| Android | >65% of High & Critical security bugs |
| Android (bluetooth & media components) | 90% of vulnerabilities |
| IOS 12 | 66.3% of all vulnerabilities |
| MacOS Mojave | 71.5% of all vulnerabilities |
| Chrome | ~70% of serious security bugs |
| Microsoft | ~70% of CVE vulnerabilities |
| Firefox  (CSS subsystem) | 73.9% of bugs |
| Ubuntu kernel | 65% of CVEs (In security updates between November and May 2020) |

Figure 3: Memory safety error statistics within major codebases (Gaynor, 2020)

These statistics were observed and reproduced across several large code bases (containing millions of lines of code). Each code base was built by a different company, started development at various points in time and applies a different development methodology. The single common property that unites these codebases is that they are written in memory-unsafe programming language such as C or C++. Gaynor concludes that the magnitude of memory-unsafe vulnerabilities is higher than memory-safe vulnerabilities and that the research supports the notion that the use of memory-safe languages would critically reduce the total number of vulnerabilities.

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 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.

### 2.4.1 Garbage Collection

Garbage collection refers to automatic memory management which is carried out by what is known as a garbage collector. It can also be described as a "memory recovery feature" which is "built into programming languages" (Sheldon, 2022). A programming language which uses a garbage collector may utilise many collectors with the aim of freeing memory allocated to objects that are no longer in use or required by the program thus the free memory can be re-used for future object allocations. Garbage collection is utilised in several programming languages including: Java, C# and D.

D is a systems programming language that utilises garbage collection. The developer allocates memory as needed and from time to time the garbage collector will free unused memory, making such memory freely available once again. D garbage collection is carried out as follows;

1. All other threads are stopped and the current thread is hijacked for garbage collection.
2. Root memory ranges are scanned for pointers to allocated memory, this memory itself is recursively scanned for more pointers.
3. All memory that holds no active pointers is freed with unreachable memory requiring destructors queued.
4. All other threads are resumed and destructors run for all queued memory, any remaining unreachable memory is freed.
5. Current thread is returned to previous work.

Garbage collection has several benefits such as ensuring a program doesn't exceed allocated memory, ensuring continued functionality and taking responsibility from developers who would otherwise need to manually manage such memory thus reducing the likelihood of memory-related bugs. Specifically in the case of D, it was found that garbage collected programs are often faster, can't suffer from memory leaks (thus have more long term stability) and are faster to develop and debug (explicit de-allocation code is not developed, debugged, tested or maintained) (D Language Foundation, 2022).

## 2.5 The Exo-kernel

# 3. Development

# 4. Experiments

# Conclusion

# Reflection

# References

# Appendices