Interim Report

Project Title:	An Analysis Suite for Rust's Advantages in Linux
Name:	Fahad Khan
Student Id:	20454859
Email Address:	efyfk3@nottingham.ac.uk
Programme:	BSc Computer Science
Module:	COMP3003

Contents

1 Introduction

- very quickly get out of the way the point and what will be achieved. add a quick paragraph on exactly how optimisation is difficult. how something on many machines, optimisation needs to be perfect. how developing memory-safe easily can really help.

1.1 Context and Motivation

Linux is an operating system that runs on billions of computers worldwide, with its underlying kernel worked on by thousands of individuals. Unlike its prominent market competitors, the ongoing development effort is open-source, allowing any individual to view and contribute to the source code. Adopters continue to further advantages such as hardware versatility, security and all-encompassing software utility [1]. For example, the kernel modularises processor-specific code, allowing it to be compiled and ported to any processor. This allows Linux to be run across a great variety of network architectures and embedded systems, which take advantage of its powerful system capability [2]. Linux's growing capital and market share have increased interest and contribution from large corporations, such as Google and Intel, becoming dependants [3]. Given the scale of development and sheer prevalence of Linux distributions worldwide, questions are naturally raised about the safety and maintainability of the kernel.

As demonstrated by the July 2024 CrowdStrike incident, where a small update with a memory issue to kernel-level software caused global Windows outages, without safety guarantees a pivotal error may result in catastrophic propagation across an operating system. Therefore, it may be important to re-evaluate the foundation on which Linux is built. The Linux kernel is written almost entirely in C, a programming language pervasive in operating system programming due to providing developers with low-level memory control and minimal runtime overhead. Optimisation is difficult yet incredibly rewarding, with precise manipulation of memory addresses enabling efficient data structure operations [4]. However, C's lack of modern safety features risks developer errors in memory management such as buffer overflows or dangling pointers, compromising system security and stability. This creates incentive to explore safer alternatives for the kernel while maintaining efficiency.

Rust now becomes the main contender for Linux development, a modern systems programming language designed to provide memory safety guarantees without sacrificing the low-level performance essential for kernel development [5]. Due to the continuing frustration with memory and concurrency issues within Linux, the value of preventing the various harms endemic to C grew into the "Rust for Linux" movement, culminating in the official addition of Rust kernel modules to the codebase in 2022 [6]. This ongoing effort has raised significant contention over the second language's practical efficiency and the prospective maintenance burden, forming the central focus of the project.

To carefully address these concerns, this project will produce a software suite to assess Rust's practical advantages within the Linux kernel. An automated framework will benchmark performance by executing a comparative suite that tests Rust modules provided by the user against other languages and variables.

1.2 Related Work

Extensive related work has focused on quantitative performance benchmarks between Rust and C, with findings largely indicating that Rust introduces performance overhead and does not eliminate all bug classes [7], [8]. could even mention the United States' Biden Administration 2024 report on using memory safe languages, "Future Software should be Memory-Safe"

1.2.1 Research Gap

There may be a research gap in these assessments of Rust's advantages, as purely quantifying overhead is insufficient to capture the full trade-off, and the project will address this by including evaluation of high-level abstractions and built-in safety guarantees.

1.3 Aims and Objectives

- make individual aspects more clear

The aim of this project is to develop an analysis software that assesses the performance of Rust against alternative languages in Linux kernel modules. Crucially, results evaluate and identify key maintainability critera, such as memory safety and abstraction.

- The key objectives for this project are to:
  1. Research and conduct a thorough evaluation of existing methods for performance benchmarking and code analysis in systems-level programming, across various language types.
  2. Develop a user-friendly software suite, such as a command-line interface, that automates the compilation and execution of modules written in Rust and alternatives.
  3. Implement a robust analysis engine within the suite that uses empirical data on kernel maintances indicators, in order to facilitate an evaluation of key maintainability criteria.
  4. Automatically process and present the collected data in a clear, comparative report in order to convey the performance and safety trade-offs between the languages.

2 Technical Background

This section will establish the necessary technical background and definitions used for the software project, which will systemically measure Rust's advantages within Linux. To effectively quantify Rust's effectiveness in this context, it is essential to identify the specific characteristics of programming language that are advantageous for general operating system development. Performance benchmarks will assess how Rust handles these characteristics, with additional tests such as failure rates under adverse conditions. These evaluation metrics also must account for C being the incumbent language within the Kernel, which has significant practical consequences for Rust's adoption and continued integration within Linux. This requires assessment of interoperability, long-term maintainability and developing a deep understanding of its abilities with direct comparisons to C whenever possible.

2.1 Evaluating Languages for Kernel Programming

General Requirements

- introduce and explain in technical terms what a kernel needs from a programming language. tie this into non-deterministic latency and garbage collection and maybe even zero cost abstraction, ruling out most languages
- deterministic latency (real-time constraints)
- no garbage collection / manual control of allocation [9]
- zero-cost abstractions & predictable code generation
- low runtime overhead / minimal runtime footprint
- fine-grained memory layout control & aliasing model
- safe concurrency primitives or provable absence of data races
- interoperability with existing C ABI and kernel build system

current sources to fit in:

- making an enhanced kernel, highlights good/bad aspects [10]
- improving the linux kernel itself, involved languages [11]
- ruling out high level languages [12]

Benchmarking Methods

Evaluating a programming language in a kernel context is an inherently difficult problem in software measurability. Software is not inherently fit for empirical analysis as its form heavily relies on the expression of the programmer. This can introduce subjectivity to quality metrics, as was reported for future memory safety by the United States' Office of the National Cyber Director [13]. This challenge is magnified in an operating system kernel as intricate interactions with hardware and concurrent processes mean its behaviour is not entirely deterministic which, as the report also notes, "hinders the capacity to reliably and consistently measure" its true performance and security characteristics. The goal is to therefore use industry-accepted empirical metrics alongside carefully planned novel ones to satisfy the research gap.

- Introduce and explain technically, how languages are benchmarked performance-wise. explain why the technical limitations are there, insert from above paragraph

2.2 The C Language and Linux

- Make a good introduction
- Once proper language requirements are added, write in certain features how it suites/violates them. also specific benchmarking once that's there

C's Memory Management

C's memory model allows direct manipulation of memory through pointers. A pointer is a variable that stores a memory address, allowing a program to indirectly access or modify the data located there. Programmers can therefore traverse memory, dynamically reference complex data structures, and interface directly with hardware registers. Through pointer arithmetic and explicit allocation functions such as malloc() and free(), one can also determine exactly where and when memory is created or released. This model eliminates any form of automatic garbage collection and avoids the indeterminate pauses associated with managed runtimes, meeting the deterministic property in kernel and embedded development. The ability to manually control allocation and object lifetime allows the Linux kernel to achieve predictable latency and fine-grained optimisation at the cost of significantly increasing the programmer's responsibility for correctness [14].

```c
int value = 10;
int *p = &value;      // p stores the address of 'value'
printf("%d\n", *p);   // Dereference to access data at that address


p = malloc(sizeof(int));
*p = 42;              // Pointer Assignment
free(p);              // Memory Released
*p = 7;               // Dangling Pointer: undefined behaviour
p = NULL;             // Manually reset pointer to avoid dangling pointer error
```

However, this same flexibility makes memory management in C both error-prone and unsafe by modern standards. Because the compiler performs minimal safety checking, even subtle mistakes in pointer usage can corrupt critical memory regions or compromise process isolation. Common errors include dangling pointers, where memory is freed while still being referenced elsewhere; buffer overflows, which occur when writes exceed allocated bounds; and double frees, where deallocated memory is erroneously released a second time. Each of these can lead to undefined behaviour, system instability, or exploitable security vulnerabilities. These issues are notoriously difficult to detect through testing alone, since incorrect memory access may not immediately manifest as a visible fault but can propagate silently until triggering a crash or privilege escalation. [13] Also, the C language provides no intrinsic mechanism for enforcing ownership, tracking lifetimes, or validating aliasing between pointers. The design assumes the programmer has full knowledge of which objects are valid and accessible at any given time, leaving correctness dependent on discipline, conventions, and manual review. While static analysis tools and runtime sanitizers can mitigate some risks, they cannot remove the fundamental unsafety of C's model. This gives us the overall trade-off for C, precise control over kernel memory layout for optimisation, which itself is a source of vulnerabilities and maintainability challenges [15].

Concurrency, and such Issues

- some information like semaphores i might not need later, check back on this after design

Concurrency refers to the ability of a program to perform multiple operations at the same time, a necessity in operating systems where several processes or threads must share system resources efficiently. In C, concurrency is achieved primarily through the use of threads, independent sequences of instructions that can execute in parallel. Threads share the same address space, meaning they can access the same variables in memory. This allows data to be exchanged quickly between threads but also makes the program responsible for ensuring that shared memory access occurs safely. To coordinate this access, programmers use synchronisation primitives such as mutexes (mutual exclusions),

which allow only one thread to access a shared variable at a time through a lock-unlock mechanism. Semaphores act similarly but have counters to control the number of threads with access to a resource. These mechanisms are provided by the operating system rather than the C language itself, typically through libraries like POSIX pthread. While this gives the programmer precise control over timing and parallelism, this also introduces incredibly contrived potential errors. If synchronization is omitted or implemented incorrectly, multiple threads may interfere with each other's operations.

The most common and dangerous form of this interference is the data race, where two or more threads access the same memory location simultaneously and at least one modifies it. As C does not automatically enforce mutual exclusion to a resource, such races can cause inconsistent or unpredictable results depending on timing and processor scheduling. The C standard leaves the behaviour of data races undefined, meaning outcomes may differ between runs or hardware architectures. Synchronisation errors can also lead to deadlocks, where two threads wait indefinitely for each other to release resources, or livelocks, where they continue executing but make no progress. In kernel development, such as within Linux's scheduler or I/O subsystems, even small timing errors in concurrency control can propagate into system-wide instability or crashes. As a result, concurrency-related faults account for a significant portion of quantified Linux vulnerabilities.

2.3 Rust for Linux

Rust's Memory Management

- need to be clearer and link to quantifying this, such as kernel panics etc

Rust introduces a unique approach to memory management that combines the efficiency of manual control with the safety of automated checks. Unlike C, which relies on explicit allocation (malloc) and deallocation (free), Rust's system has three self-regulating concepts: ownership, borrowing, and lifetimes [16]. Together, these rules allow memory to be managed deterministically at compile time without the need for a garbage collector. The central principle is that every value in Rust has a single owner, such as a variable responsible for its lifetime. When that owner goes out of scope, the compiler automatically inserts code to drop and deallocate the associated memory. This behaviour ensures that memory is always released exactly once, preventing a programmer from causing unforeseen issues prevalent in more manually managed languages such as C.

Broadening the understanding of Rust's memory model, references to data can be borrowed temporarily by other parts of a program under strict compile-time conditions. Rust enforces that multiple immutable references can coexist safely, but if a mutable reference exists then no other references are allowed. These constraints are checked by the compiler's borrow checker which analyses variable lifetimes and reference validity before the program is run. This mechanism guarantees the absence of both dangling pointers and data races, by design rather than by convention or additional runtime detection. The compiler effectively encodes the logic of safe memory sharing, preventing one thread from modifying data while another is reading it. As a result, Rust can deliver the low-level control and predictable performance expected in kernel-level development, while removing a significant class of concurrency and safety errors at compile time.

Unsafe Blocks

Although Rust enforces strict memory safety through its ownership and borrowing rules, some operations in low-level systems programming cannot be expressed entirely within those constraints. To enable such functionality, Rust provides the unsafe block, a controlled mechanism that allows developers to perform actions which the compiler cannot guarantee as safe [17]. Within an unsafe block,

programmers may dereference raw pointers, call external functions through the C Application Binary Interface (ABI), access or modify mutable static variables, and perform certain low-level operations required for direct hardware interaction. Only these unsafe blocks, similarly to C, the programmer assumes responsibility for upholding Rust's safety guarantees manually. In the context of Linux kernel development, part of it relies on unsafe code, it is essential for interacting with hardware registers, device memory, and existing C subsystems that operate outside Rust's ownership model. The benefit over C is clearly within its containment of unsafe operations: they are explicitly marked and typically encapsulated within higher-level safe abstractions. For example, a device driver may use an unsafe function internally to perform register access but expose only safe interfaces to the rest of the kernel.

The presence of unsafe code also provides an additional metric for evaluating kernel reliability. Quantifying the frequency and scope of unsafe usage can act as a proxy for the degree of risk or potential for kernel panics resulting from memory misuse. The "Rust for Linux" initiative tracks these boundaries explicitly, **expand on this**.

Potential Extra Benchmarking Methods for Rust

3 Project Description

3.1 Design

3.1.1 Benchmarking Approach

3.1.2 Optimisation Tools

3.2 Methodology

4 Progress

5 Bibliography

[1] E. Siever, *Linux in a Nutshell*. O'Reilly, 2005.

[2] D. Abbott, *Linux for embedded and real-time applications*. Newnes, an imprint of Elsevier, 2018.

[3] J. West and J. Dedrick, "Open source standardization: The rise of linux in the network era," *Knowledge, Technology & Policy*, vol. 14, pp. 88–112, 2001, doi: 10.1007/pl00022278.

[4] L. O. Andersen, "Program analysis and specialization for the C programming language," 1994.

[5] N. D. Matsakis and F. S. Klock, "The rust language," Association for Computing Machinery, Portland, Oregon, USA, 2014. doi: 10.1145/2663171.2663188.

[6] H. Li, L. Guo, Y. Yang, S. Wang, and M. Xu, "An Empirical Study of Rust-for-Linux: The Success, Dissatisfaction, and Compromise," in *2024 USENIX Annual Technical Conference (USENIX ATC 24)*, Santa Clara, CA: USENIX Association, Jul. 2024, pp. 425–443. [Online]. Available: https://www.usenix.org/conference/atc24/presentation/li-hongyu

[7] S.-F. Chen and Y.-S. Wu, "Linux Kernel Module Development with Rust," 2022, doi: 10.1109/dsc54232.2022.9888822.

[8] F. Garber, "Rust in the Linux Kernel: Analyzing Rust Implementations of Device Drivers," *Tuwien.at*, 2025, doi: 10.347s.2025.127963.

[9] Y. Chang and A. Wellings, "Garbage Collection for Flexible Hard Real-Time Systems," *IEEE Transactions on Computers*, vol. 59, no. 8, pp. 1063–1075, 2010, doi: 10.1109/TC.2010.13.

[10] Z. Shao, "Advanced Development of Certified OS Kernels," *Yale FLINT Group*, 2015, [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.367.9646

[11] S. Saha, J. Lawall, and G. Muller, "An approach to improving the structure of error-handling code in the linux kernel," *acm*, pp. 41–50, 2011, doi: 10.1145/1967677.1967684.

[12] C. Cutler, M. F. Kaashoek, and R. T. Morris, "The benefits and costs of writing a POSIX kernel in a high-level language." [Online]. Available: https://www.usenix.org/conference/osdi18/presentation/cutler

[13] The United States Biden-Harris White House, Office of the National Cyber Director (ONCD), "Back to the Building Blocks: A Path Toward Secure and Measurable Software," 2024.

[14] D. M. Ritchie, S. C. Johnson, M. Lesk, B. Kernighan, and others, "The C programming language," *Bell Sys. Tech. J*, vol. 57, no. 6, pp. 1991–2019, 1978.

[15] D. Gay, R. Ennals, and E. Brewer, "Safe manual memory management," in *Proceedings of the 6th International Symposium on Memory Management*, in ISMM '07. Montreal, Quebec, Canada: Association for Computing Machinery, 2007. doi: 10.1145/1296907.1296911.

[16] S. Klabnik and C. Nichols, *The Rust programming language*. No Starch Press, 2023.

[17] B. Qin, Y. Chen, Z. Yu, L. Song, and Y. Zhang, "Understanding memory and thread safety practices and issues in real-world Rust programs," in PLDI 2020. London, UK: Association for Computing Machinery, 2020. doi: 10.1145/3385412.3386036.

The number of words in this is 2436