Exploring Microkernel Development: A Comparative Analysis of the Development Process of Rust and C for the Raspberry Pi 4

College of Information and Computer Sciences
syallapragad@umass.edu
January 2024

Introduction:

With the rapid development of IoT and embedded systems in today's world, there is a growing need for safe and power efficient systems-level programming. Rust has emerged as a strong contender in this domain, recently being included in the Linux Kernel. Its strengths lie in memory safety, performance, and concurrency guarantees, complemented by its ownership and borrowing system, which ensures that memory-related bugs, such as null pointer dereferences and data races, are caught by the compiler. These characteristics make Rust a compelling choice for systems programming.

Meanwhile, C continues to be a prominent language for kernel development, boasting a longstanding history in low-level systems and is the language of choice for a majority of modern-day operating systems. Using C allows developers to directly control memory management and optimize for performance. With decades of use and optimization, C has accumulated a vast knowledge base of existing libraries and documentation, making it a trustworthy language for system development.

In light of Rust's surging popularity, frequently acclaimed as the most beloved language on platforms like Stack Overflow [9,10], this honors portfolio aims to present a comparative analysis between Rust and C for bare-metal programming. By scrutinizing their respective attributes in the context of microkernel development on the Raspberry Pi 4 (RPi 4), this manuscript seeks to offer valuable insights into the evolving landscape of systems programming. It provides an analysis of trade-offs and advantages associated with each language and offers insight into the challenges faced by those intending to switch their systems projects from C to Rust.

The comparative analysis was conducted by implementing a microkernel in both C and Rust, designed to trigger single-core interrupts every 1000000 clock cycles and produce output to the miniUART. Notably, as Rust does not use header files and lacks a preprocessor, the structure of the two projects varies slightly. After implementing both these systems, a comparison of both the code bases was conducted to identify and analyze attributes of the two languages. Any discerned differences and encountered issues were comprehensively documented and explored within the contents of this manuscript.

An interesting perspective that inspired this project is the potential for transitioning applications to Rust, fostering a more energy-conscious approach among software developers, as exemplified by Amazon Web Services [7]. The author, alongside insights from another study [1], emphasizes the striking similarity in energy usage between Rust and C. By migrating code, such as server code, to Rust, considerable energy savings can be achieved in server operations while simultaneously increasing performance. This transition also mitigates the inherent risks associated with C programming, where manual allocation and deallocation of memory may introduce vulnerabilities and memory leaks. This research was inspired by the informative content found in an Amazon blog post [7], sparking a thought process that expands the idea of power efficiency from servers to low-power embedded systems, where preserving battery life is crucial.

This manuscript aims to enhance the understanding of transitioning from C to Rust by spotlighting tools, obstacles, challenges, and the overall feasibility. This holds significance in light of a substantial surge in transitioning established tools to Rust, exemplified by the ongoing rewrite of GNU Coreutils into Rust for the creation of uuutils as a cross-platform drop in replacement. Additionally, research indicates that Rust code exhibits fewer memory related vulnerabilities, attributed to the effective elimination of issues like dangling pointers and improper memory deallocation through its borrow checker system and compile time guarantees [6]. The proposition of enhanced security without compromising performance makes research into transitioning to Rust code relevant. Furthermore, the field of bare-metal Rust is notably under-documented when compared to C, lacking beginner-friendly resources. This manuscript functions as an introductory exploration, providing insights into the realm of bare-metal Rust development.

Throughout this project, it became evident that Rust offers distinct advantages with its user-friendly error messages, tools like cargo, and highlighting unsafe code explicitly through its compiler restrictions. This enhances the programming experience and makes it more accessible for developers, especially those transitioning from other languages. However, it is noteworthy that Rust's tutorial ecosystem appears less robust compared to the extensive resources available for C. Despite the initial intention to explore power consumption statistics, the testing setup encountered challenges, consequently hindering conclusive insights into the power efficiency of Rust versus C within the scope of this study. The incomplete exploration of power efficiency paves the way for future research using different methodologies to analyze and compare power consumption between Rust and C.

<u>Background:</u>

I initiated my research by exploring the paper titled "Ranking Programming Languages by Energy Efficiency". This paper offers a comprehensive study on the energy efficiency of various programming languages. Its objective is to shed light on the energy consumption of different languages and rank them based on their efficiency. The authors conducted computer language game benchmarks on identical hardware to obtain their results [1]. The paper addresses important questions such as whether a faster program is also energy efficient and whether a faster language is more energy efficient. However, it is important to note that the discussions in this paper took place in the context of a server system and may not directly apply to embedded systems. One crucial insight I gained from this paper is that energy measurements are influenced by temperature and external conditions. Additionally, the paper highlights the strong correlation between memory and energy consumption in certain languages but not all. For instance, there is a significant correlation between energy and memory in the case of the C language, whereas Rust shows only a moderate correlation. For C, careful memory management may be crucial to minimize energy consumption. In contrast, Rust's safety features may lead to a more consistent and energy-efficient behavior, even with varying memory usage.

After establishing that Rust and C exhibit relatively similar performance and energy usage in a benchmarking environment, I delved into the realm of unsafe Rust. It is widely acknowledged that programming in C often leads to unsafe programs if not thoroughly tested, as even minor errors can have catastrophic consequences. Therefore, I examined the paper titled "How Do Programmers Use Unsafe Rust?" to explore the perspective of Rust in terms of safety. Rust is considered a safe language because when using its default features, programmers are provided with safety guarantees related to types and memory. The paper conducted an analysis of Rust's crates and found that the majority of programmers utilize safe Rust, which offers the performance benefits of C while mitigating memory and type bugs [2].

Rust advocates for addressing unsafe Rust usage by employing well-developed “battle-tested” abstractions and libraries. It's important to notice that safe Rust does not imply secure Rust. While safe Rust ensures code that adheres to Rust's safety guarantees, it does not guarantee security against malicious exploitation, security vulnerabilities and programmer errors. Therefore, it is essential to verify that Rust crates also employ secure code.

With the understanding that a transition to Rust might be advantageous due to its enhanced safety compared to C without compromising performance, I shifted my focus to researching the applications and challenges of using Rust. This led me to the paper titled "Ownership is Theft: Experiences Building an Embedded OS in Rust." The paper argues that Rust poses difficulties in embedded systems programming due to its strict ownership model. Rust restricts resource sharing, which is considered safe and efficient in the embedded domain [3]. The authors of the paper encountered three main problems in their development efforts. Firstly, Rust's automatic memory management is not optimized for hardware resources and device drivers that are always present in the system as an alternative. Secondly, Rust's ownership model hinders resource sharing between closures and other kernel code due to unnecessary thread safety concerns in the embedded setting. Lastly, although closures are desirable for simplified event handling, their requirement for dynamic memory poses challenges for embedded systems where memory itself is often limited by hardware or cost requirements. However, there are still direct benefits to using Rust in embedded operating systems. Often, embedded systems do not have memory management units, which means language safety provides software safety where hardware safety is lacking. Another benefit of Rust is that it does not use a garbage collector, which allows embedded systems to efficiently use the limited memory.

Looking to gain a better understanding of Rust as a whole I discovered the paper titled "Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study." This paper offers a comprehensive examination of Rust as a secure programming language. It delves into core features and the ecosystem of Rust, providing in-depth explanations of concepts such as Ownership and Lifetimes models. The authors conducted a survey to gain insights into the motivations behind the significant shift towards Rust. Key findings include the importance placed on safety and performance, the challenges posed by the learning curve associated with the borrow checker and programming paradigms, and the perception of Rust's slow compilation speed compared to languages like Go [6].

The paper also highlights concerns about dependency bloat and the lack of certain libraries due to Rust's evolving nature. Furthermore, the authors assert that Rust enhances productivity in the software development cycle, however they also concede that areas such as web application, GUI and mobile development are areas Rust is not suited to. This paper deepened my understanding of the reasons software developers are increasingly adopting Rust and shed light on potential challenges I may encounter as I learn the language. It provided valuable insights into the motivations and roadblocks faced by developers, which I can anticipate and address during my own exploration of Rust.

The blog post titled "Sustainability with Rust" from Amazon (https://aws.amazon.com/blogs/opensource/sustainability-with-rust/) [7] provides valuable insights into the practical applications of Rust in the server space. It highlights the considerable improvements that can be achieved by adopting Rust, emphasizing its potential to enhance efficiency and optimize resource utilization. By leveraging Rust's unique features, including its strict compile-time safety checks, AWS services can benefit from reduced downtime, improved stability, and heightened security. Moreover, the blog post raises awareness about the environmental impact of server infrastructures and the urgent need for sustainable solutions. By embracing Rust's performance and energy efficiency characteristics, AWS services can effectively reduce their carbon footprint while maintaining high levels of reliability and scalability.

These insights hold significant relevance to my research, particularly as embedded systems also contribute to carbon emissions, making it crucial to minimize their environmental impact.

I delved into the available resources for building an operating system specifically for the Raspberry Pi as this is a widely accessible platform for beginners. I came across two exceptional blog-style tutorials that caught my attention. The first tutorial, found at https://www.rpi4os.com/ [4], provides a step-by-step guide to building an OS for the Raspberry Pi using the C programming language. It covers the topics of kernel building, miniUART, framebuffers, mailboxes, bluetooth, sound modules, interrupts, multicore as well as tcp/ip web servers. It also includes instructions to build breakout (a game). The second tutorial, available at https://github.com/rust-embedded/rust-raspberrypi-OS-tutorials [5], offers a similar guide but focuses on utilizing Rust as the programming language for operating systems development. It covers topics from a Rust perspective and delves into safe globals, kernel heap, timer callbacks, privilege level, virtual memory and kernel symbols. Both tutorials follow a comparable progression, leading developers through the OS-building process up to the network driver level. However, I intend to make certain modifications and adjustments to suit the specific requirements and goals of my project. I would also like to note that I found the C tutorial to be much more accessible as a beginner than the Rust tutorial as the Rust tutorial assumed a much higher level of understanding and technical competency.

To understand how to measure power efficiently and gain insights into power measurements in embedded systems, I referred to the paper titled "A Survey of Energy Consumption Measurement in Embedded Systems." This paper discusses different methods for energy measurements, including measurement-based, model-based, and simulator-based approaches. It covers instrument-based and platform-based measurement methods, model-based approaches using system calls, and simulator-based estimation techniques [8].

Simulators software rely often on instruction-level analysis. This enables them to estimate the energy consumption of embedded software rapidly, but this comes at the expense of development and verification of the simulator. The first step is to simulate various functions of the operating system to support applications. Then, the energy consumption of the application is calculated based on the datasheet or the existing power models.[8] The pitfalls of the simulator, which end up being the strengths of hardware testing, are that factors in the actual circuit design are ignored and simulator data is as accurate as the existing power model. Hardware testing provides a more accurate depiction of embedded os power consumption, but testing different programs and setting up different hardware configuration is a huge investment, while you can run multiple simulators at the same time, which are quicker and can work in parallel.

Based on this paper, I decided to attempt a physical measurement of the entire system at once to measure power efficiency due to the complexity of developing a simulator based energy measurement system. I will not be facing the drawback of testing different hardwares as I will only be using one RPI4 platform to run my benchmarks. I will be using a USB power meter which will enable me to obtain precise data on the entire system's power efficiency.

This method eliminates the complexity and potential inaccuracies associated with developing a simulator-based energy measurement system. The USB power meter will provide readings of milliwatt-hours (mWh), which will be crucial for tracking the power consumption. Additionally, the time taken for each benchmark to run will be recorded alongside the power measurements. This time data will be vital for analyzing the correlation between time and energy consumption in both languages. By correlating these metrics, I can gain insights into the power efficiency of the micro operating systems developed in Rust and C, and better understand how the languages' characteristics impact the overall system performance and energy usage.

In conclusion, the literature review provided valuable insights into the energy efficiency, safety, and challenges of using Rust and C in the context of operating systems development. Exploring the benefits and limitations of both Rust and C has helped shape my approach to this project as well as my methodology.

Methodology:

The primary aim of this research is to evaluate the inherent capabilities of the language. To achieve this, a conscious decision was made to refrain from incorporating externally developed and community-maintained tools commonly used by language practitioners. Instead, the emphasis is placed on utilizing core language attributes and tooling, specifically excluding external resources such as Rust crates and additional C libraries. The development environment necessitated the inclusion of a no_std flag, indicative of bare-metal development where a standard runtime environment is absent, requiring careful management of a compact kernel build footprint. While this methodology is designed to be fair to both languages, it's essential to acknowledge that my programming background is rooted in C, and I possess more experience in writing C code than Rust code.

Furthermore, the code inevitably includes Arm Assembly code as it is required to write the boot code which jumps to the kernel written in both the languages. The compilation process employed the same linker script for both projects and mostly the same Arm Assembly code. C and Rust were separately compiled in their respective branches of the project using the GNU arm cross compiler and Cargo respectively, resulting in the creation of the C microkernel labeled "CZPZ" and its Rust counterpart, named "RHD2," drawing inspiration from the iconic duo C3PO and R2-D2 from Star Wars. All the code can be found at https://github.com/Srinanda-Yallapragada/honors-rpi4os .

Hardware required:

Name	Specs	Notes
Development machine	ROG Zephyrus G14 Model: GA401QM AMD Ryzen™ 9 5900HS NVIDIA® GeForce 3060 RAM: 16GB DDR4 OS: Nobara 39 Linux	Main system that I developed the C and Rust implementation on. While I used Linux, any operating system should suffice.
Raspberry Pi Model 4B	Broadcom BCM2711, (ARM v8) 64-bit RAM: 8GB Standard 40-pin GPIO header Micro SD card slot for loading OS	I purchased the Vilros Raspberry Pi 4 8GB Basic Starter Kit from Amazon.
USB to TTL Serial Cable 	[Link to component]()	Purchased off Amazon Photo attached for reference.

Micro SD Card	SanDisk 64GB Extreme microSDXC UHS-I Memory Card	Purchased with usb to microSD adapter from amazon. Any Micro SD Card should suffice.
USB Power Meter Multi-Function Tester	Voltage: 4-30V Current: 0-3A Capacity range: 0-999999mAh Energy range: 0-999999mWh Timer: 0-99H Power: 0-150W Charging resistance: 0-999.9Ω Temperature: 0-80C Link to component	N/A

Development environment setup:

Name	Set up	Notes
Operating system Nobara 39 Linux	Installed from https://nobaraproject.org/ Ran the following commands sudo usermod -a -G uucp $USER on arch sudo usermod -a -G dialout $USER on ubuntu and fedora The TTY on linux is ttyUSB0 for me to connect the serial cable through Putty	This is a fedora based operating system. Ubuntu based and Arch based distributions have availability of the rest of the software on this list. Unfortunately, I cannot speak for the processes on MacOS or Windows based devices.
Putty: Terminal emulator and serial console application. Used to read and send data over the miniUART to communicate with the RPi4.	See file rp4 on the github for full Putty config settings. Notable settings set Speed(baud) -> 115200 Data bit -> 8 bits Stop bit -> 1 bit Parity -> None	See rpi4os.com for config settings for the pi for additional information.

Visual Studio Code: Popular code editor developed by Microsoft.	Extensions installed: Rust-analyzer C/C++ IntelliSense C/C++ Extension Pack	Rust-analyzer is officially supported by the rust foundation.
LunarVim: An IDE layer for Neovim with sane defaults.	N/A, LunarVim once installed auto configured based on file extension.	Language processing server automatically installed and configured itself when I opened C and Rust files.
C Programming Language	Used the Arm GNU cross compiler https://developer.arm.com/downloads/-/arm-gnu-toolchain-downloads Configured Makefile to use absolute paths to where I downloaded this cross compiler	Followed instructions from tutorial rpi4os.com [4]
Rust Programming Language	Installed Rust using rustup https://rustup.rs	Many online discussions suggested that a nightly version of Rust was

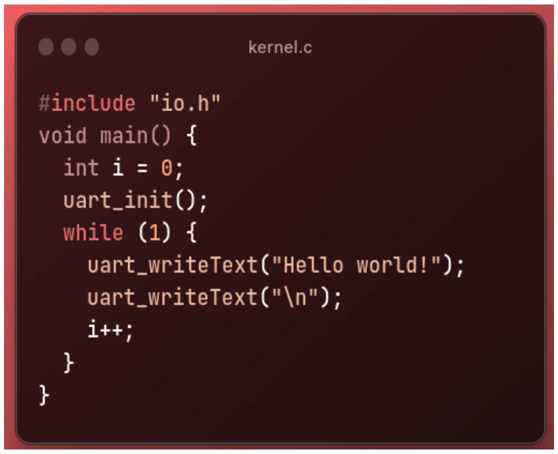
	Ran commands
	
	`rustup component add llvm-tools-preview`
	
	`rustup component add thumbv7-none-eabi`
	
	To get objdump command and cross compile to arm.
		necessary for bare-metal compilation; however, I successfully compiled and executed the code using the stable version.

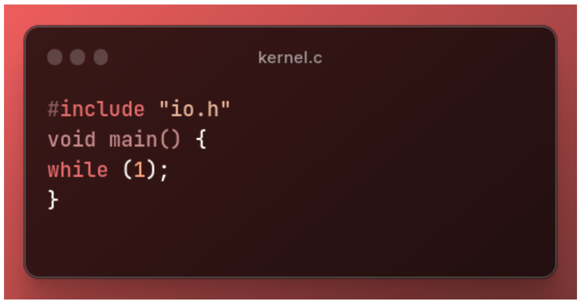
Arm Assembly	N/A	Assembly was used in the boot code and partly to enable interrupts. The code was linked and compiled using a linker script. It was compiled using C and Rust separately in their respective folders of the project.

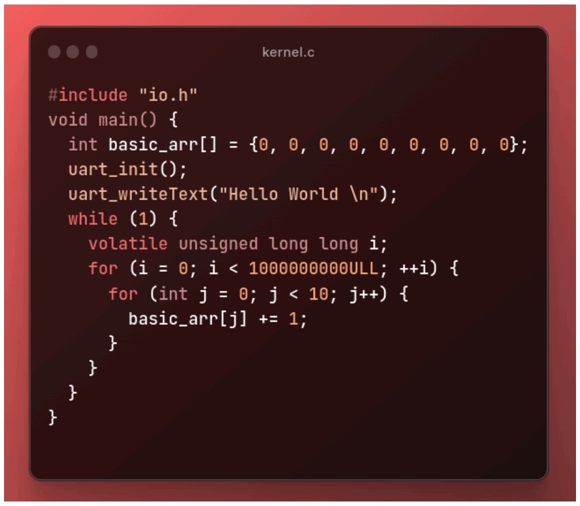
After obtaining the necessary hardware components and ensuring proper software setup, the first step was to run preliminary tests to explore the feasibility of measuring changes in power consumption on the RPi 4. The objective was to investigate potential variations in power usage based on the instructions and programming language running on the processor. Despite attempts to measure power usage using a USB power meter, it became apparent that the minimal system did not yield discernible differences in power consumption. Efforts were made to test power consumption during different scenarios, including what I initially felt were “active” states and “idle” states. However, the power consumption remained consistent, prompting further investigation.

Here is a preliminary table showing the initial readings from running a minimal kernel to see if I could measure differences in power consumption. This testing was done only in C to first see if I could measure different kernel.c power consumptions.

Code	Result after running for 3 min	Notes

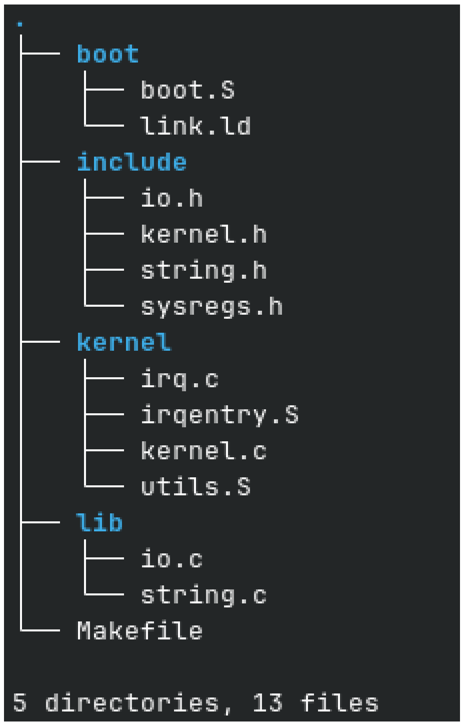
 kernel.c `#include "io.h"`
`void main() {`
`  int i = 0;`
`  uart_init();`
`  while (1) {`
`    uart_writeText("Hello world!");`
`    uart_writeText("\n");`
`    i++;`
`  }`
`}`
	94 mWh (Measure of power consumed)	I believed this to be an “active” state initially as I was writing out to the miniUART and had an i++ instruction in the code.

 kernel.c `#include "io.h"`
`void main() {`
`while (1);`
`}`
	94mWh	I believed this was an “idle” state initially as this was a while true loop running without doing any “active” computation.

 kernel.c
`#include "io.h"`
`void main() {`
`  int basic_arr[] = {0, 0, 0, 0, 0, 0, 0, 0, 0};`
`  uart_init();`
`  uart_writeText("Hello World \n");`
`  while (1) {`
`    volatile unsigned long long i;`
`    for (i = 0; i < 1000000000ULL; ++i) {`
`      for (int j = 0; j < 10; j++) {`
`        basic_arr[j] += 1;`
`      }`
`    }`
`  }`
`}`
	94mWh	The thinking behind this code is that maybe memory access would create a difference in the power consumption, however that was not the case.

After seeing this data, I acknowledged that my initial testing failed to induce any changes in power output. Upon further reflection, I recognized that even when the CPU executes a nop instruction, the RPi4 still consumes power. Implementing power-saving measures, such as putting the CPU cores in wait-for-event (wfe) or wait-for-interrupt (wfi) states, would be necessary to observe power variations. Additionally, I realized that I lacked the technical expertise and resources required to maintain controlled thermal environments for accurate power measurements. However, considering that this would stretch the scope of the project, I opted to discontinue the exploration of power-related metrics and pivot towards exploring the developer experience while continuing the microkernel development. As I began developing the two kernels, I noticed that they had taken different folder structures to better suit the way each language developed. I have described the folders and files for both kernels below.

The C version took the following folder structure



```
.
├── boot
│   ├── boot.S
│   └── link.ld
├── include
│   ├── io.h
│   ├── kernel.h
│   ├── string.h
│   └── sysregs.h
├── kernel
│   ├── irq.c
│   ├── irqentry.S
│   ├── kernel.c
│   └── utils.S
├── lib
│   ├── io.c
│   └── string.c
└── Makefile

5 directories, 13 files
```

The code was broken into 4 folders containing relevant files and a Makefile.

Boot folder files	Description
boot.S	ARM assembly instructions for timer configuration, EL1 setup, stack establishment, and main kernel entry point. Initially taken from rpi4os tutorial [4]
link.ld	Linker script used during compilation. Initially taken from rpi4os tutorial [4]

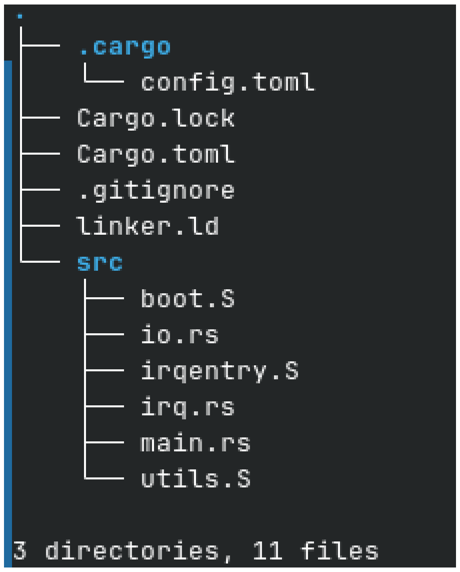
Include folder files	Description
io.h	Header file with UART function declarations. Initially taken from rpi4os tutorial [4]
kernel.h	Contains structs defining timer and interrupt registers, along with IRQ-related function headers. Initially taken from rpi4os tutorial [4]
string.h	Implements basic string functions (e.g., lenstr, cmpstr)
sysregs.h	Includes registers and bit flags for transitioning into EL1 during the boot process. Initially taken from rpi4os tutorial [4]

Kernel folder files	Description
irq.c	Manages enabling, disabling, and handling of interrupt requests. Initially taken from rpi4os tutorial [4]]

irqentry.S	Assembly code for loading the interrupt vector table and managing system state during interrupts. Initially taken from rpi4os tutorial [4]
kernel.c	Main kernel file where the system initializes the timer, sets up interrupts, and enters an endless loop.
utils.S	Assembly file with helper functions for vector table initialization, IRQ enabling, and IRQ disabling. Initially taken from rpi4os tutorial [4]

Lib folder files	Description
io.c	Implementation of UART functions. Initially taken from rpi4os tutorial [4]
string.c	Implementation of string functions.

On the Rust side this was the folder structure


```
.
├── .cargo
│   └── config.toml
├── Cargo.lock
├── Cargo.toml
├── .gitignore
├── linker.ld
└── src
    ├── boot.S
    ├── io.rs
    ├── irqentry.S
    ├── irq.rs
    ├── main.rs
    └── utils.S

3 directories, 11 files
```

Non-src files	Description
.cargo/config.toml	Configuration file instructing the compiler to use the correct linker script and triple target aarch64-unknown-none-softfloat.
Cargo.toml	Describes project metadata and dependencies.
Cargo.lock	Automatically generated file describing dependencies based on Cargo.toml.
linker.ld	Linker script used during compilation. Initially taken from rpi4os tutorial [4]

Src folder files	Description
boot.S	ARM assembly instructions for timer configuration, EL1 setup, stack establishment, and main kernel entry point. Initially taken from rpi4os tutorial [4]
io.rs	Rust file containing io.h declarations and their implementations, consolidating UART-related functionality.
irqentry.S	Assembly code for loading the interrupt vector table and managing system state during interrupts. Initially taken from rpi4os tutorial [4]
irq.rs	Contains structs defining timer and interrupt registers, along with IRQ-related function headers. Manages enabling, disabling, and handling of interrupt requests.
main.rs	Main kernel file where the system initializes the timer, sets up interrupts, and enters an endless loop.

utils.S	Assembly file with helper functions for vector table initialization, IRQ enabling, and IRQ disabling. Initially taken from rpi4os tutorial [4]

Here is each step of the methodology described as I explored writing the microkernel.

1) Software setup:

   I began by setting up the software environment, installing essential tools like the GNU arm cross compiler and Rust through rustup. I explored the extensions and language servers in VSCode and LunarVim. I prepared firmware files for the SD card, including bcm2711-rpi-4-b.dtb, fixup4.dat, start4.elf, config.txt, and uploaded our microkernel with the filename kernel8.img (all available on the portfolio github). Things to note during this step is that the kernel must be named kernel8.img and the sd card must be properly ejected, otherwise the RPi4 will not read it.

2) Hardware setup:

   I inserted the formatted SD card into the Raspberry Pi, connected the serial cable to designated GPIO pins [11], and supplied power through a type c power source which completed the hardware setup. No significant issues were encountered during this phase. Here is a picture of my setup.

3) Microkernel implementation:

   I started by implementing the miniUART interface for the C version of the operating system. Afterward, I translated this code into Rust, ensuring that both versions of the microkernel could send output through the miniUART. I then spent considerable time diving into the assembly code from the rpi4os tutorial, dissecting each line to understand flag and mode configurations during boot and how interrupts are activated [11]. I carefully followed the code's flow, adding comments for clarity. Lastly, I implemented single-core interrupts, initially in C and later replicated the process in Rust.

4) Codebase Analysis and Comparison:

   The final step involved a comprehensive analysis and comparison of the entire codebase. I scrutinized unsafe code sections to assess Rust's approach, drawing comparisons with C implementations to gain insights into language-specific nuances, particularly in pointer management. This code analysis aimed to shed light on the comparative strengths and weaknesses of Rust and C in the context of microkernel development for the RPi4.

No external training was undertaken for this project. However, relevant courses at UMass, such as How Computers Work Inside the Box (CS335) and Computer Architecture (CS535) provided foundational knowledge. The Operating Systems (CS337) course exposed me to topics like interrupts which was essentially to succeed at enabling interrupt. My proficiency in C was developed through the courses Introduction to C Programming (CS198C) and Computer Systems Principles (CS230), while I learned Rust independently for this project. I heavily relied on tutorial resources, including rpi4os.com and rust-raspberry-pi-tutorials for Raspberry Pi operating systems development, along with references to other resources that have been referenced throughout this document.

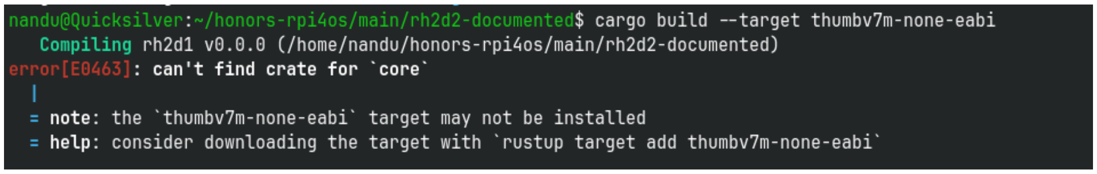
Comparison and Analysis:

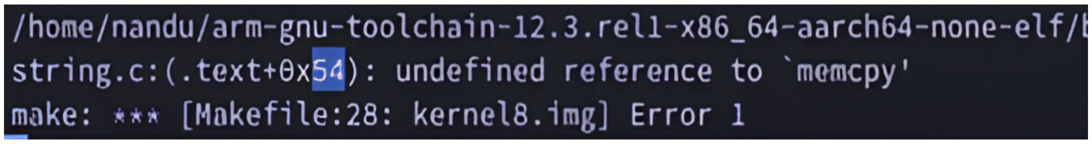
Before delving into the comparative analysis of the two systems, it's essential to acknowledge the underlying context of this project. As an undergraduate student, my journey is marked by a keen interest in learning Rust, primarily drawn to its promises of type safety and its perceived ease as a systems programming language in comparison to C. This project serves as my first venture into Rust and has actively shaped my learning experience. Throughout this analysis, it's important to note that I am still in the process of learning Rust, with my primary exposure to systems programming being in C up to this point. My learning of Rust has primarily been through online tutorials and the Rust book. The challenges and hurdles encountered may not point to shortcomings of the language but rather represent my learning journey and the obstacles I encountered. Considering that I was in a learning phase, I may not have discovered solutions to these issues during that time.

1. ***Tooling***
    a. ***Cross Compiler setup***:

       Setting up the C GNU Arm cross-compiler proved to be more cumbersome and tedious than its Rust counterpart. The C process involved multiple manual steps, such as downloading, unzipping, and manually configuring the compiler path in the Makefile. This complexity extended to searching for the correct version compatible with the target architecture. In contrast, Rust's approach with the "rustup install" command streamlined the installation of the cross-compiler, making it more straightforward and user-friendly. While both setups achieved the goal of enabling cross-compilation, the C process required more manual interventions and potential navigation through external resources, highlighting the smoother experience provided by Rust's tooling, particularly "rustup."

b. Build systems:

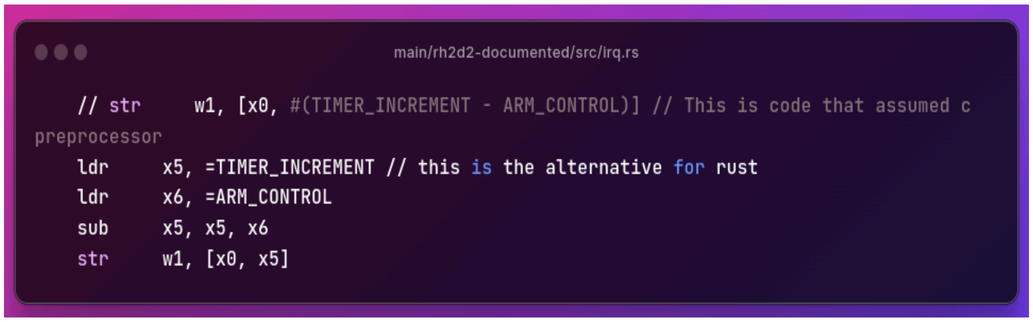

```
nandu@Quicksilver:~/honors-rpi4os/main/rh2d2-documented$ cargo build --target thumbv7m-none-eabi
   Compiling rh2d1 v0.0.0 (/home/nandu/honors-rpi4os/main/rh2d2-documented)
error[E0463]: can't find crate for `core`
  |
  = note: the `thumbv7m-none-eabi` target may not be installed
  = help: consider downloading the target with `rustup target add thumbv7m-none-eabi`
```


```
/home/nandu/arm-gnu-toolchain-12.3.rel1-x86_64-aarch64-none-elf/
string.c:(.text+0x54): undefined reference to `memcpy'
make: *** [Makefile:28: kernel8.img] Error 1
```

Cargo stands out as Rust's default and nearly ubiquitous build system, assuming a central role in the Rust ecosystem. In contrast, the C landscape offers a plethora of options, including Make, CMake, Ninja, and more, providing significant flexibility. It became evident that Cargo offers a more user-friendly experience, providing clear indications on how to solve encountered problems. A comparison between error messages in Make and Cargo illustrates this. While the Make build system might halt at an "undefined reference" error, the Cargo system not only identifies the error but also offers helpful notes and hints for resolution. Cargo's approach streamlines debugging and issue resolution, enhancing accessibility and increasing development speed. The consistent presence of such informative error messages characterizes the Rust development process, a feature notably absent in the context of C development. Cargo also includes quality of life commands like “cargo new”, which sets up a new application, complete with Cargo.toml files, a src folder, and a main.rs file with a simple hello world application. This further contributes to the overall ease of use of Cargo and, by extension, Rust.

c. ***Preprocessor:***

The presence of a preprocessor is a significant distinction between C and Rust. In C, the preprocessor provides macro definitions, conditional compilation, and the inclusion of header files, contributing to efficient code organization and simplifying certain tasks. Conversely, Rust lacks a traditional preprocessor but compensates with a robust macro system. While the Rust macro system offers flexibility, it differs in functionality and usage from the traditional C preprocessor. C developers transitioning to Rust may find the absence of the preprocessor a major point of adjustment, requiring the relearning of macros and conditional compilation in the Rustic way. The lack of a preprocessor in Rust can be challenging, leading to extensive code refactoring if coming from a C code base. This impact extends to assembly code, although my relative inexperience as a programmer might play a role in my inability to refactor code efficiently in the example below.



main/rh2d2-documented/src/irq.rs

```
    // str     w1, [x0, #(TIMER_INCREMENT - ARM_CONTROL)] // This is code that assumed c
preprocessor
    ldr     x5, =TIMER_INCREMENT // this is the alternative for rust
    ldr     x6, =ARM_CONTROL
    sub     x5, x5, x6
    str     w1, [x0, x5]
```

I could not maintain the one-liner syntax. Instead, I had to rewrite it into four separate lines to allow the code to compile. Adapting to the absence of a preprocessor is a learning curve, especially for developers transitioning from C.

d. ***Language processing tools***

Rust and C both excel in supporting language processing tools such as linting and auto-formatting. Rust goes a step further by offering guidance on coding style, providing suggestions for variable naming and issuing warnings when snake_case conventions are not followed. It's worth noting that these suggestions can be disabled in the code using the #![allow(non_snake_case)] attribute. Both languages swiftly highlight errors in real-time, and my testing in LunarVim and Visual Studio Code emphasized the seamless configuration and utilization of language processing servers for both Rust and C. The robust support for these languages through extensions and code-editing tools enhances the overall development experience.

e. ***Tool versions***

Regarding tool versions, it's essential to highlight the update process for both Rust and C. In C, updating the Arm GNU cross-compiler involves re-downloading and manually updating the system with the required version. On the other hand, Rust offers a more streamlined approach. A simple command, "rustup update," not only updates the Rust language itself but also covers any target components, such as thumbv7-none-eabi and LLVM tools. This unified updating mechanism significantly simplifies the management of programming languages and their supporting tools in the Rust ecosystem, contributing to a more efficient and user-friendly experience.

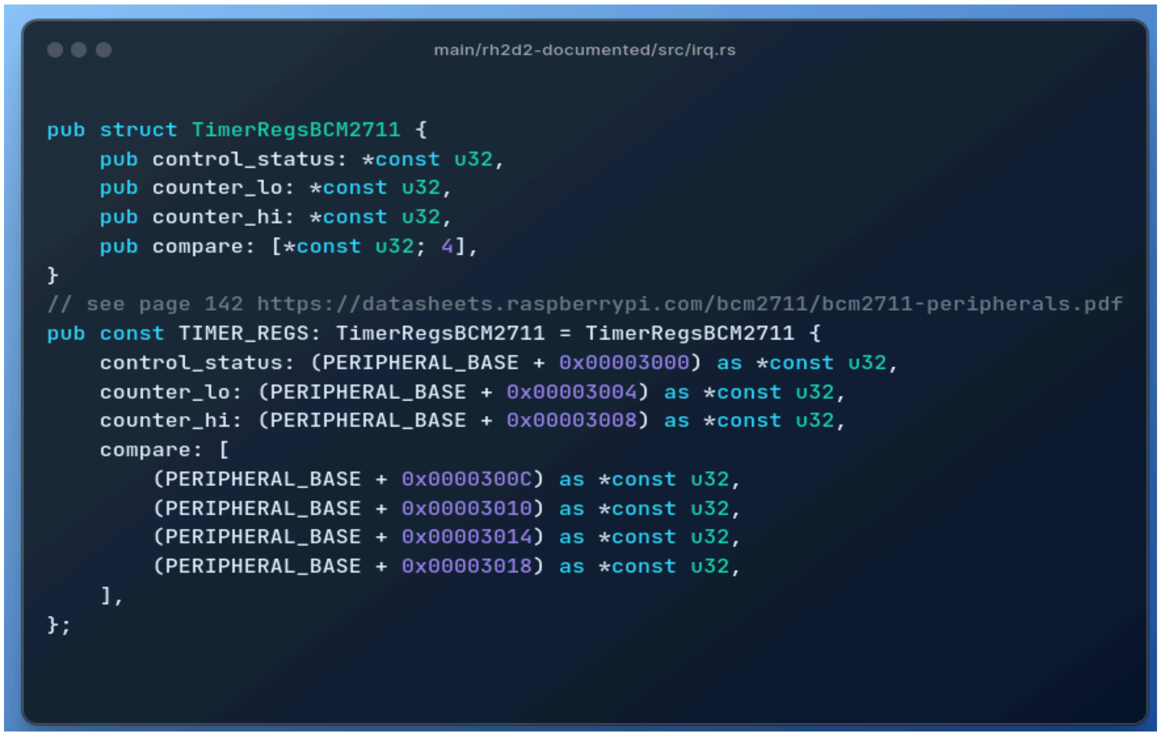
2. ***Tutorial resources and documentation***

   The abundance of resources available for C, coupled with its longer presence in the programming landscape made it easier to find comprehensive guides to starting bare metal and operating system development. In contrast, my exploration of resources in Rust encountered delays as I struggled to find materials suitable for my beginner-level understanding of the programming language. Although my greater experience with C may have influenced this observation, the prominent Rust tutorial [5] I followed primarily focused on demonstrating code differences without providing in-depth explanations of underlying concepts. Rust's comparatively shorter time in the systems spotlight, coupled with the fact that bare-metal programming and unsafe Rust are used irregularly by most Rust programmers, contributed to this roadblock. Additionally, many Rust resources assume a higher level of prior knowledge, making them less accessible for beginners compared to the more beginner-friendly resources available for C. Rust tutorials and documentation are continually expanding, featuring projects like the Rustonomicon, a tutorial-style book on using unsafe Rust. The Rustonomicon, however, comes with a disclaimer: "...we will be assuming considerable prior knowledge. In particular, you should be comfortable with basic systems programming and Rust..." [12]. Despite these challenges, both languages boast fantastic documentation, and their core language features are well-documented and supported with examples.

3. ***Rust code vs C code***

Code being referenced is presented below

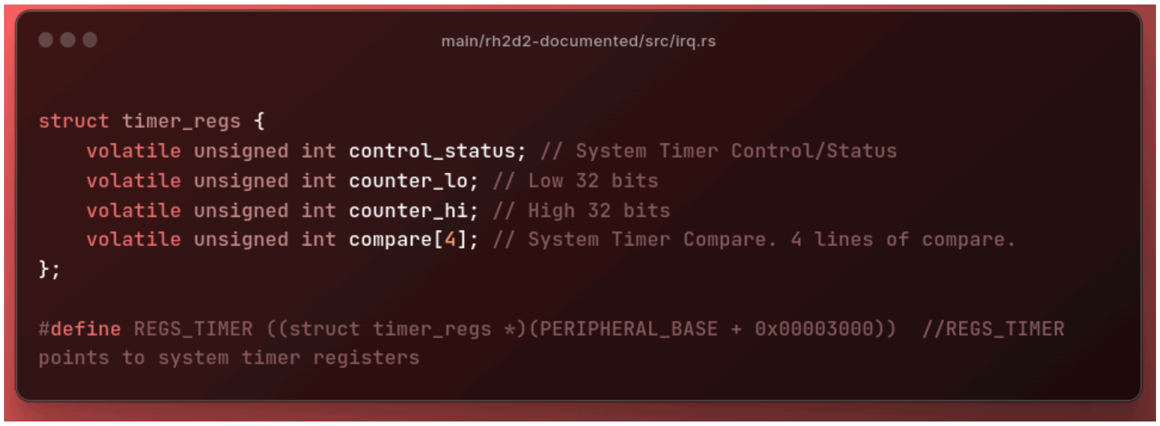
Below is the Rust code



main/rh2d2-documented/src/irq.rs

```rust
pub struct TimerRegsBCM2711 {
    pub control_status: *const u32,
    pub counter_lo: *const u32,
    pub counter_hi: *const u32,
    pub compare: [*const u32; 4],
}
// see page 142 https://datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf
pub const TIMER_REGS: TimerRegsBCM2711 = TimerRegsBCM2711 {
    control_status: (PERIPHERAL_BASE + 0x00003000) as *const u32,
    counter_lo: (PERIPHERAL_BASE + 0x00003004) as *const u32,
    counter_hi: (PERIPHERAL_BASE + 0x00003008) as *const u32,
    compare: [
        (PERIPHERAL_BASE + 0x0000300C) as *const u32,
        (PERIPHERAL_BASE + 0x00003010) as *const u32,
        (PERIPHERAL_BASE + 0x00003014) as *const u32,
        (PERIPHERAL_BASE + 0x00003018) as *const u32,
    ],
};
```

Below is the C code



main/rh2d2-documented/src/irq.rs

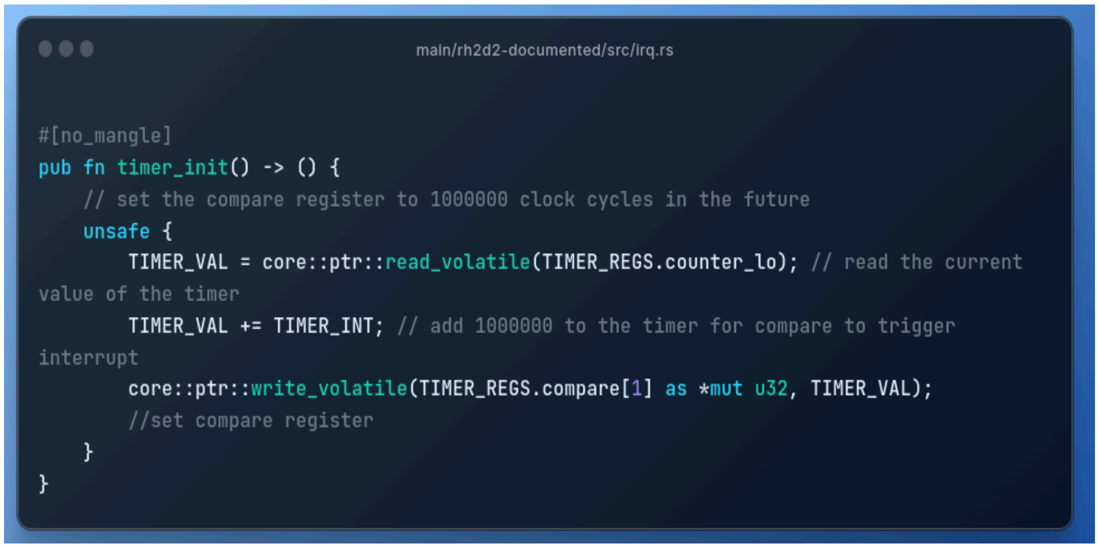
```c
struct timer_regs {
    volatile unsigned int control_status; // System Timer Control/Status
    volatile unsigned int counter_lo; // Low 32 bits
    volatile unsigned int counter_hi; // High 32 bits
    volatile unsigned int compare[4]; // System Timer Compare. 4 lines of compare.
};

#define REGS_TIMER ((struct timer_regs *)(PERIPHERAL_BASE + 0x00003000))  //REGS_TIMER
points to system timer registers
```

a. Scope: I had to declare the struct and its parameters as public using the pub keyword. This step was necessary to ensure proper access to the struct attributes within Rust code without triggering compiler errors. Unlike C, Rust requires explicit acknowledgment of which parts of the code are publicly visible and assumes that the properties of structs are private by default. This approach compels the programmer to explicitly identify which parts of the struct need to be publicly visible in the program which promotes better encapsulation of code.

b. Pointers: C allows for easy manipulation of data through pointers. In the code example, using pointers simplifies working with structs and enables the assignment of structs to timer memory registers on the RPi4. Only the start/base address of the registers is needed, and the struct can be constructed to match the required offsets. However, in Rust, this approach is hindered by the type safeties the language maintains. Manual specification of addresses for each component is required, introducing frustration, especially with more complex structs. C proves to be more straightforward in working with pointers at a bare-metal level, facilitating interactions with the RPi4. Meanwhile, Rust's emphasis on explicit typing adds friction when writing bare-metal code. However, using pointers in C is more error-prone, and Rust addresses the same tasks by enforcing more explicit code, theoretically resulting in safer code.

c. Safety: This is where Rust is expected to shine, as it markets itself as a memory-safe language. However, in bare-metal programming, where unsafe actions like direct memory reading and writing are essential, Rust's safety guarantees are considerably weaker. The unsafe keyword serves two purposes: to declare the existence of contracts that the compiler can't check and to declare that a programmer has verified these contracts. The burden of correctness is placed heavily on the programmer for unsafe code [12]. Many code blocks in Rust were wrapped in unsafe{} blocks, indicating that the programmer must take responsibility for ensuring correctness. In C, all code is written with the ability to manipulate memory freely, placing the safety of the code fully in the hands of the programmer. In Rust, the use of unsafe is encouraged to be minimized, and due to the memory safety guarantees of Rust, any potential memory bugs are confined to the unsafe code blocks. Rust forces developers to confront potentially risky code by flagging operations like write_volatile, unsafe code blocks, and static mutable as unsafe, ensuring explicit acknowledgment of their potentially risky nature.



main/rh2d2-documented/src/irq.rs

```
#[no_mangle]
pub fn timer_init() -> () {
    // set the compare register to 1000000 clock cycles in the future
    unsafe {
        TIMER_VAL = core::ptr::read_volatile(TIMER_REGS.counter_lo); // read the current
value of the timer
        TIMER_VAL += TIMER_INT; // add 1000000 to the timer for compare to trigger
interrupt
        core::ptr::write_volatile(TIMER_REGS.compare[1] as *mut u32, TIMER_VAL);
        //set compare register
    }
}
```

In the code above, TIMER_VAL is declared as a static mutable type which implies a potential data race for reading and writing to this variable. Despite the microkernel being single-threaded, I was directed by the Rust compiler to label any line involving static mut as unsafe. Rust's core functions, like write_volatile, are also named with the indication that they are unsafe functions, signaling the potential for issues beyond the Rust compiler's control.

The prevalence of such scenarios led me to the conclusion that Rust's type and safety guarantees are compromised in no_std Rust development. While Rust may still guide developers to write code in a Rustic way, relying on the compiler to maintain type safety becomes challenging for bare-metal applications. In this context, C is not significantly worse off in terms of safety than Rust, but I would argue that Rust remains more effective at writing safer code. This is because Rust forces programmers to acknowledge where unsafe code is, making them more aware of potential bugs, data races, and other issues.

d. ***Integrating assembly***

Working with assembly is relatively straightforward in both languages once you understand how each language handles it. While it might be easier to work with assembly in C, Rust also proves to be accommodating. Simple commands like #no_mangle, which allows functions to retain their string names for assembly code and linker identification, are helpful. Additionally, using constructs like global_asm!(include_str!("boot.S")) facilitates the inclusion of assembly code directly into the Rust codebase. These lines of code, along with the use of extern C functions, make it effective and straightforward to call existing assembly labels, simplifying the process of importing them into the Rust project. This ease of integration allows developers to leverage assembly code seamlessly in both languages. I was able to maintain the same linker script for both the versions because of the ease of including assembly in both languages. Both languages also have support for inline assembly, which is an added bonus.

Throughout my exploration, I didn't run into problems with the borrow checker or Rust's ownership model, which was unexpected given my initial concerns from prior readings like “Ownership is Theft” [3]. It's worth noting that the nature of the project, a relatively straightforward single-core interrupt-firing microkernel, might not have been complex enough to fully engage with Rust's advanced features like traits and the borrow checker. This suggests a potential avenue for future research to explore Rust's applicability and challenges in more intricate projects that make extensive use of its sophisticated features.

Conclusion:

This research explored bare-metal programming on the RPi4, aiming to highlight the developer experience of both C and Rust. The primary objective was to provide valuable insights for developers, particularly those venturing into the intricacies of low-level coding in Rust coming from a background in C. The study focused on comparing the differences in developing a microkernel between C and Rust in terms of safety, user-friendliness, and the unique challenges encountered in the realm of low-level development.

Through the comparison, I have found that C's extensive history and widespread adoption have established it as a go-to language for bare-metal programming, offering a plethora of tools and a well-established ecosystem. Its preprocessor capabilities facilitate efficient code organization, and the abundance of resources available simplifies the learning process, particularly for beginners. It shines in terms of pointer manipulation, allowing easy interaction with memory and structs, crucial for bare-metal development, however this easy interaction comes at the cost of making the language more error-prone while writing out code.

On the other hand, Rust, a relatively new entry in the programming landscape, stands out as a strong choice for bare-metal programming with its focus on type safety, memory safety and promoting good coding practices baked into the language design. The language's tooling, exemplified by Cargo and rustup, stands out as a user-friendly build system, streamlining project setup, compilation, updates, and dependency management.

While my undergraduate coursework equipped me with a robust theoretical understanding of OS and kernel development concepts, when it came to translating this theoretical knowledge into actual code, I faced a practical gap. Writing the code for interrupts and vector tables required a substantial adaptation period, where I spent a significant amount of time understanding and adding comments to the code from the rpi4os tutorial. This process involved piecing together the intricacies of the code and cross-referencing assembly instructions with the corresponding registers as outlined in the rpi4 documentation. This hands-on experience highlighted the need for a more practical-oriented approach to complement theoretical knowledge, suggesting potential improvements in educational strategies for OS and kernel development.

If presented with a fresh start, I would opt for writing this project targeting general x86 systems instead of confining myself to the Raspberry Pi. This broader approach could enhance the project's versatility and access to tutorial resources. Additionally, I would choose to execute the code through QEMU (open source machine emulator and virtualizer) instead of relying solely on hardware, significantly accelerating the development process. Embracing QEMU's debugging tools would vastly improve the overall developer experience, providing a more efficient and iterative debugging workflow. I've come to realize that debugging through hardware development can be exceptionally slow, necessitating frequent recompilation and hardware testing after minimal code changes. This suboptimal process has highlighted the need for a more streamlined and time-efficient development approach. Looking ahead, there is promising potential for future research to delve into the dynamic ecosystem of bare-metal programming. Exploring the community-maintained crates and standard files in Rust as compared to C could reveal valuable insights, shedding light on the status of each language’s most used community tools.

References:

1. Pereira, Rui, et al. “Ranking Programming Languages by Energy Efficiency.” *Science of Computer Programming*, vol. 205, 2021, p. 102609, https://doi.org/10.1016/j.scico.2021.102609.
2. Astrauskas, Vytautas, et al. “How Do Programmers Use Unsafe Rust?” *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, 2020, pp. 1–27, https://doi.org/10.1145/3428204.
3. Levy, Amit, et al. “Ownership Is Theft.” *Proceedings of the 8th Workshop on Programming Languages and Operating Systems*, 2015, https://doi.org/10.1145/2818302.2818306.
4. Greenwood-Byrne , Adam. “Writing a ‘Bare Metal’ Operating System for Raspberry Pi 4.” *Rpi4*, Sept. 2022, www.rpi4os.com/.
5. Richter , Andre. “Rust-Embedded/Rust-Raspberrypi-Os-Tutorials: Learn to Write an Embedded OS in Rust.” *GitHub*, Mar. 2023, github.com/rust-embedded/rust-raspberrypi-OS-tutorials.
6. Fulton, Kelsey R, et al. “Seventeenth Symposium on Usable Privacy and Security.” USENIX Association, *Benefits and Drawbacks of Adopting a Secure Programming Language: Rust as a Case Study*.
7. Miller, Shane, and Carl Lerche. “Sustainability with Rust.” *Amazon*, Feb. 2022, aws.amazon.com/blogs/opensource/sustainability-with-rust/.
8. Guo, Chen, et al. “A Survey of Energy Consumption Measurement in Embedded Systems.” *IEEE Access*, vol. 9, Apr. 2021, pp. 60516–60530, https://doi.org/10.1109/access.2021.3074070.

9. “Stack Overflow Developer Survey 2022.” *Stack Overflow*, 22 June 2022, survey.stackoverflow.co/2022#most-loved-dreaded-and-wanted-language-want.
10. “Stack Overflow Developer Survey 2023.” *Stack Overflow*, 13 June 2023, survey.stackoverflow.co/2023/.
11. BCM2711 Arm Peripherals - Raspberry Pi, datasheets.raspberrypi.com/bcm2711/bcm2711-peripherals.pdf. Accessed 22 Jan. 2024.
12. “The Rustonomicon.” Introduction - The Rustonomicon, doc.rust-lang.org/nomicon/intro.html.