Tampereen yliopisto
Tampere University

Elias Meyer

Rust for Linux

Linux device driver development in Rust

Master's thesis
Tampere University
Faculty of Information Technology and Communication Sciences
Examiner: Master of Science Henri Lunnikivi
Examiner: Professor Timo Hämäläinen
May 2024

Abstract

Elias Meyer
Rust for Linux: Linux device driver development in Rust
Tampere University
Faculty of Information Technology and Communication Sciences
Master's thesis
May 2024

Keywords: Linux, Rust, operating system, device driver, kernel, module

The development of Linux began in the 90s. At the time, C was the most powerful and modern programming language. However, C's origins go back to the 1970s, which is reflected particularly into the cumbersome and error-prone software development, which in turn leads to the constant discovery of new security vulnerabilities.

Rust is a modern systems programming language which aims to improve on the weaknesses of C and C++, especially in terms of memory management, while maintaining the performance advantage. In recent years, large IT companies have begun to take an interest in Rust after realizing its advantages. In addition to efficiency, these include a high level of abstraction, which speeds up software development. The security in memory management prevents security vulnerabilities by preventing buffer overflows. A good memory management also allows for easy and efficient concurrency. Rust's modern development environment with package management, a built-in linter and a documentation creation tool facilitates software development.

Officially, Rust support was added into Linux in December 2022 in version 6.1. There is no intention to rewrite Linux, but only to make it possible for new device drivers to be written in Rust. Linux developers hope that the modern programming language will make it easier to maintain new drivers, reduce security problems and encourage new developers to contribute to Linux.

The purpose of this thesis was first to evaluate and implement a device driver using Rust for an embedded system of Wapice Ltd. Unfortunately, due to the lack of a needed API on Linux, it was not yet economically feasible. Linux still lacks many interfaces which allow device drivers to be easily written in Rust - currently, the device driver author has to create the interfaces very likely themself. For this reason, the purpose of this thesis shifted to exploring the scope of Rust support in the form of a literature review: what has already been written in Rust so far, and what benefits and problems have been identified in *Rust for Linux*.

The work presents six existing device drivers written in Rust and collects both the advantages and disadvantages reported by their authors. The device drivers are very different from each other: there are a few block device drivers such as the NVMe driver, a network device driver for a physical ethernet controller, an inter process communication bus for Android called the Binder, as well as GPU drivers for Apple and Nvidia graphics cards. The work will compare the differences between these drivers and the corresponding C drivers, if available. For example, for the NVMe driver the efficiency between the Rust and C versions of the driver are compared while for the Binder driver both the efficiency and the object file size are compared. This paper also presents two identified major challenges raised by *Rust for Linux*, and discusses their possible and most likely solutions.

The literature review shows that when implemented in Rust, the drivers are slightly slower than their C counterparts, and the size comparison shows that the device drivers implemented in Rust are slightly larger. However, the differences are not significant compared to the advantages of Rust, making it an excellent choice for programming device drivers on Linux.

Tiivistelmä

Elias Meyer
Laiteajurin kehitys Linux-käyttöjärjestelmään Rust-ohjelmointikielellä
Tampereen Yliopisto
Informaatioteknologian ja viestinnän tiedekunta
Diplomityö
Toukokuu 2024

Avainsanat: Linux, Rust, käyttöjärjestelmä, laiteajuri, kernelmoduuli

Linuxin kehitys aloitettiin 90-luvulla, jolloin C oli tehokkain ja modernein ohjelmointikieli. C sai kuitenkin alkunsa jo 70-luvulla, mikä näkyy ohjelmistokehityksen kankeutena sekä virhealttiutena käytettäessä C:tä. Virhealttius näkyy esimerkiksi jatkuvasti löytyvinä tietoturvaongelmina.

Rust on moderni järjestelmäohjelmointikieli, joka pyrkii parantamaan C:n ja C++:n heikkouksia etenkin muistinhallinnan suhteen, säilyttäen kuitenkin näiden kielien edun suorituskyvyssä. Monien etujensa ansiosta Rust on viime vuosina alkanut herättää yhä useamman suuren tietotekniikka-alan yrityksen mielenkiinnon. Etuihin lukeutuu tehokkuuden lisäksi korkea abstraktiotaso, joka nopeuttaa ohjelmistokehitystä. Muistinhallinnan varmuus taas ennaltaehkäisee puskurien ylivuotoja, jotka ovat usein tietoturva-aukkojen perimmäisiä syitä. Hyvä muistinhallinta myös mahdollistaa käyttäjille helpompaa ja tehokkaampaa rinnakkaisuutta, koska suurin osa rinnakkaisuusongelmista on pohjimmiltaan muistinhallintaongelmia. Moderni kehitysympäristö pakettienhallinnan sekä sisäänrakennetun koodin jäsentely- ja dokumentaatiotyökalun kanssa helpottaa omalta osaltaan ohjelmistokehitystä Rustilla.

Virallisesti Rust-tuki tuli Linuxiin joulukuussa 2022 versiossa 6.1. Linuxia ei ole tarkoitus kirjoittaa uudelleen, vaan tuen on tarkoitus ainoastaan mahdollistaa uusien laiteajurien kehittäminen Rustilla. Linuxin kehittäjät toivovat modernin ohjelmointikielen helpottavan uusien laiteajurien ylläpitoa, vähentävän niissä esiintyviä tietoturvaongelmia ja herättää uusien kehittäjien kiinostusta Linuxiin.

Työn tarkoitus oli aluksi arvioida laiteajurin kehitystä Wapice Oy:n sulautettuun järjestelmään Rustilla ja toteuttaa se. Valitettavasti Linuxin Rust-tuen puutteellisuuden vuoksi sen tekeminen ei ollut vielä taloudellisesti hyödyllistä. Linuxista puuttuu vielä monia rajapintoja, joiden avulla laiteajureita voidaan helposti kehittää Rustilla - nykyisellään laiteajurin tekijä joutuu hyvin todennäköisesti tekemään rajapinnat itse. Tästä syystä työn tarkoitus siirtyi Rust-tuen laajuuden tutkimiseen kirjallisuuskatsauksen muodossa: mitä kaikkea tähän mennessä ollaan jo tehty Rustilla Linuxiin ja mitä hyötyjä tai ongelmia Rustissa ollaan havaittu Linuxin suhteen.

Työssä esitellään kuusi olemassa olevaa Rustilla kirjoitettua laiteajuria ja kootaan niiden tekijöiden raportoimia kokemuksia niin eduista kuin haitoistakin. Laiteajurit ovat keskenään hyvin erilaisia: mukaan mahtuu muun muassa muutamia lohkolaiteajureita kuten NVMe-ajuri, verkkolaiteajuri Ethernet-ohjaimelle, Androidin prosessikommunikaatioväylänä toimiva Binder sekä lisäksi Applen ja Nvidian näytönohjainajureita. Työssä vertaillaan näiden ajurien eroja vastaaviin C:llä kirjoitettuihin ajureihin, mikäli sellaiset ovat saatavilla. Esimerkiksi NVMe-ajurissa vertaillaan ajurien välistä tehokkuutta, kun taas Binder-ajurissa vertaillaan sekä tehokkuutta että objektitiedoston kokoa. Tässä työssä esitellään myös kaksi suurta Rust-tuen esille tuomaa ongelmaa Linuxissa ja keskustellaan niiden mahdollisista ja todennäköisimmistä ratkaisuista.

Kirjallisuuskatsauksessa käy ilmi, että Rustilla toteutettuna ajurit ovat hieman vastaavia C-toteutuksiaan hitaampia. Kokovertailussa selviää, että Rustilla tehdyt laiteajurit ovat hieman suurempia. Erot eivät kuitenkaan ole merkittäviä Rustin muihin etuihin verrattuna, mikä tekee Rustista erinomaisen vaihtoehdon laiteajurien ohjelmointiin Linuxissa.

Alkusanat

Olen kiitollinen työnantajalleni Wapice Oy:lle, kun sain haastaa itseäni tämän diplomityön muodossa mielenkiintoisesta aiheesta. Kiitos TkT Sakari Junnilalle, joka ammattimaisesti ohjasi työtä haasteista huolimatta sekä DI Henri Lunnikivelle ja prof Timo Hämäläiselle työn tarkastamisesta yliopiston puolelta. Kiitos Topi uusien ideoiden antamisesta linkkaamalla ajankohtaisia artikkeleita. Kiitos myös DI Santtu Söderholmille, jonka Typst-diplomityöpohjan päälle tämä työ on ladottu.

Aloittaessani opintojani aikanani Tampereen Teknillisessä Yliopistossa en voinut kuvitellakaan, miten tärkeäksi yliopiston yhteisöt muodostuisivat minulle. Täten haluaisin kiittää Tampereen Yliopiston organisaatioista kiltaani TiTeä, Rakkauden Wappuradiota, Teekkarikuoroa, TTYkitystä, TTSS:aa sekä Castoria tästä seikkailusta. Haluan myös muistaa kaikkia ystäviäni, joita matkani varrella tapasin.

Suurin kiitos kuitenkin perheelleni tämän pitkän taipaleen horjumattomassa tukemisessa.

Tampereella 15. toukokuuta 2024,
Elias Meyer

Glossary

AOSP	Android Open Source Project [1]
FFI	Foreign Function Interface
FUSE	File system in user space
GSP	Graphics System Processor
initramfs	Initial RAM-based file system, a root file system for Linux
IPC	Inter Process Communications, collective activities for facilitating communications and data sharing between user space applications [2]
LDM	Linux Device Model
LKM	Linux Kernel Module framework [3]
MEMS	Micro-Electro-Mechanical Systems [4]
MMU	Memory Management Unit
monomorpization	When a compiler encounters a function with generic arguments, it will replace the generic function with monomorphic functions for all the different argument types it encounters
mutex	Mutual Exclusion, a binary-semaphore like locking primitive for locking a resource for one thread at a time [5]
NVMe	Technology for non-volatile memory over PCI Express [6]
OCIv1	Open Container Initiative, version 1
OOM reaper	Out-of-memory killer [3]
OS	Operating System, e.g., Linux or Windows
qemu	Quick Emulator: a generic and open source machine emulator and virtualizer
RAII	Resource Acquisition Is Initialization [7]
RCU	Read-copy-update, see section 2.4
RTOS	Real-Time Operating System
SoC	System-on-a-chip

Contents

List of figures

List of tables

List of code blocks

List of text blocks

1 Introduction

1.1 Motivation

Linux is ubiquitious in the world of embedded systems. One of the most important reasons for this is the transparency of the system - everyone is welcome to implement new device drivers for their device. Linux already has countless device drivers and the number is only growing as more devices are developed which need to be integrated into Linux.

Linux is written in C as it is the most common language for performance critical tasks and low level programming. But the systems programming languages have a new kid in town: Rust promises to solve security vulnerabilities caused by memory errors. The Linux kernel development has its own peculiarities with its API and traditionally a lot of time had to be invested in debugging memory errors and synchronization problems. Now that the Linux kernel is embracing Rust in the project *Rust for Linux* [28], it would be feasible to let the compiler check for more errors than ever before and thus improve the development speed, reduce the amount of new bugs as well as make the learning curve more gradual for newcomers.

This work was done for Wapice as part of the Finnish SoC Hub collaboration project [29]. Our intention is to develop a Linux distribution on the RISC-V core developed by SoC Hub. Because the project Rust for Linux is integrating support for writing device drivers in Rust into the Linux kernel, it seemed only desirable to study the new opportunity - the knowledge would directly be useful for both SoC Hub and us.

1.2 Conducted work and methodologies

This thesis evaluates the current status of the project *Rust for Linux* [28], which aims to bring the Rust programming language into the Linux kernel. In other words, the Linux kernel is not rewritten in Rust but rather it should be possible to write device drivers in both C and Rust. This thesis answers to questions, *what projects have already been implemented using Rust in the Linux kernel* and *what strengths and weaknesses have the developers found out when using Rust in the Linux kernel*. The analysis is being done by studying relevant mails of the Linux kernel mailing list as well as some articles by the authors of these new Rust related projects.

Linux has a huge codebase and concurrent development is done on different parts of it, the project *Rust for Linux* being no exception. For instance, the Linux device drivers for a SoC are widely using a so called `miscdev`-framework discussed in section 3.5 but also GPU drivers for both Nvidia and Apple hardware are discussed. For these reasons this thesis takes a broad perspective and the reviewed kernel projects are quite versatile, ranging from NVMe stor-

age device driver to the Android security framework driver Android Binder all the way to the Apple AGX GPU driver.

This thesis also discusses two major obstacles hindering *Rust for Linux* currently and explains what are the proposed solutions at the moment. The first problem is the safe pinned initialization problem (*pinned-init*, section 6.1) and the second one is the compile-time detection of atomic context violations (*klint*, section 6.2). As with the existing kernel projects, these are also inspected by studying relevant articles from the kernel developers.

Both the existing implementations and the identified challenges require extensive knowledge about the topics to be able to contribute to them, which is out of scope for this thesis. Most of the existing implementations already have comprehensive performance benchmarks conducted which is enough to be referenced, but no study seem to have been done regarding the size of the compiled kernel modules. This seems to be genuinely new study in this thesis.

1.2.1 Hypotheses for the study

There were three hypotheses for the study:
1. Drivers written in Rust are about 5 - 15% slower than drivers written in C
2. Writing a driver in Rust is faster than writing a driver in C
3. A driver written in Rust is somewhat bigger than a driver written in C

Hypothesis #1 is that even though Rust is on the same grade with C regarding performance, it still makes some runtime checks which makes it a little slower. Because the kernel is already written in C, most likely the current performance can’t simply be achieved. This will be tried to confirm by analyzing benchmarks from existing projects.

The speed alone isn’t the reason to choose Rust into the kernel: development ergonomics also play a major role, which brings us to the hypothesis #2. Rust is a modern systems programming language with good abstractions. Once properly integrated into the kernel, writing kernel modules is most likely faster than writing them in C. Although Rust is said to have a steeper learning curve than C, Rust programs are also said to lack debugging which will most likely reduce the complete time used for development. There is not much data yet about writing the same module in C and Rust so the hypothesis #2 will be tried to be solved by using the opinions of the authors of the existing Rust drivers.

Hypothesis #3 comes from the fact that in user space Rust binaries are not exactly small. This thesis examines two kernel drivers and scrutinizes the different sections of the object files and compares the C and Rust drivers by them.

Because all of these hypotheses concern the project *Rust for Linux* it is only feasible to try to find the answers by studying the above-mentioned project.

1.2.2 Information retrieval and sources

Most sources used in this thesis are documentations on the internet for Rust, Linux and *Rust for Linux*. Some academic work was also found but either they are obsolete or did not answer to the hypotheses for this thesis, which is why they were excluded. The information retrieval happened between 2023-05-24 and 2024-05-13.

This thesis is mostly a literature review which means a lot of literature was read, understood and referenced. A development environment was also set up, where kernel device drivers could be developed with both C and Rust, but because of immature infrastructure, nothing practical was eventually done.

Because the author had no experience in writing kernel modules, the book *Linux Kernel Programming* from Kaiwan N. Billimoria [3] was one of the best sources to explain the internals of the kernel. It was published in 2021 and it discusses the kernel version 5.4, which was the newest at the time of writing it. The kernel versions discussed in this thesis are from 6.3.0 up. For perspective, other bibliographies like *Linux Device Drivers, 3rd edition* [30] covers Linux 2.6 and *Understanding The Linux Kernel, 3rd edition* [31] discusses Linux 2.6.10.

The project *Rust for Linux* has its own collection of documentation at [28] which points to the original sources of the subprojects. It is good to know, that this documentation has evolved a lot while writing this thesis, and in the beginning some of the projects were not even listed in it - the sources were first found by other means.

Numerous other sources are used as well, but the most common recurring sources are https://kernel.org, https://phoronix.com and https://lwn.net for information about the Linux kernel, https://doc.rust-lang.org for information about the Rust programming language and https://github.com because most of the open source projects mirror their repository in GitHub.

1.3 Thesis structure

First, the Linux kernel is introduced in chapter 2 for the relevant parts and device drivers and their purpose is described in chapter 3. Chapter 4 explains how the kernel is configured, what the project *Rust for Linux* is and what are the benefits of it. Some existing and compelling Rust device driver implementations are introduced in chapter 5, where the possible benefits and drawbacks of Rust are also noted. Some of the projects have done performance comparisons with the corresponding device drivers written in C, which are also referenced. Because these results are not the work of this thesis but merely borrowed from other authors, the results are listed directly there and not in the chapter 8 Results.

In chapter 6, two major identified challenges regarding Rust with the Linux kernel API are explained thoroughly - what the problems are, how are the problems impacting the usage of Rust in Linux and what solutions are being developed to tackle the issues. Chapter 7 points out to the development environment explained in appendix A and explains the usage of it, as well as the execution of a kernel module code size comparison. The chapter also explains a project idea which was soon abandoned after some background research proved it to be way out-of-scope for this thesis. Chapter 8 lists the results for the kernel module size comparison as well as some background of the branches used in the *Rust for Linux* project. The thesis ends with the conclusions in chapter 10.

2 Linux kernel

Linux is an operating system kernel which means it is the component between the application and the hardware as seen in Figure 1. It is responsible for sharing the limited hardware resources between multiple processes. The kernel is completely invisible to the user. [32]

The kernel has four main responsibilities: memory management, process management, device drivers and security. The hardware resources are sparse, which makes the kernel responsible for dividing them as fairly as possible between all the running applications on the host. This shared use applies to the CPU, memory and IO-devices such as hard drives and external peripherals. [32]

The operating system also has a big role for the security of the system. It is responsible for segregation of the applications so they may not disturb each other. In Linux memory is paged, which means each process gets its own memory pages and won't be able to read or write to other processes memory pages. The kernel is responsible for providing these memory pages and to clean them before usage, so that no sensitive information from one process will be given to another process [33]. Failing the memory segregation will lead to security vulnerabilities, which may be used, e.g., for privilege escalation, like the *Dirty Cow* bug regarding private read-only memory pages [34].

On one hand, modern CPUs have multiple execution units (cores or hyperthreads), which are able to execute threads. On the other hand, processes in Linux may each contain one or multiple threads. Process management includes scheduling, which is used to give each process some execution time on the CPU core without letting any of the processes starve.

2.1 User space and kernel space

Microprocessors support different privilege levels by hardware, which is not a new invention. For instance, the Intel 80386 architecture has 4 ring levels [22], and the ARM microprocessor family supports up to seven privilege modes, called the *execution modes* [3]. This old design is still not outdated, because the modern amd64 CPUs are based on these older CPU architectures. Unix makes use of only two of these privilege levels (in the 80386 architecture) for platform security and stability: ring 0, which is called the *supervisor mode* or the *kernel space*, and the lowest ring, which is called the *user mode* or the *user space*.

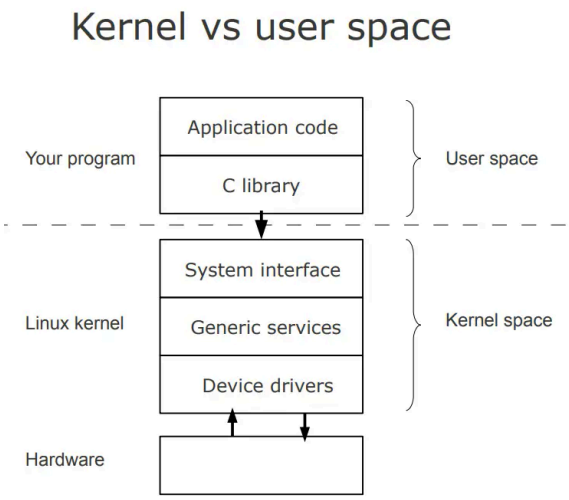


Figure 1: User space and kernel space [8]

The user space and kernel space are visualized in Figure 1. User space is used to run all normal applications and the kernel space is only used for kernel code which includes the device drivers. One of the ways a user space program may interact with the kernel is to use system calls, which is the API of the kernel. Other possible interaction methods for the Linux kernel are ioctl:s or pseudo-file systems like `/dev`, `/sys` and `/proc`. [3]

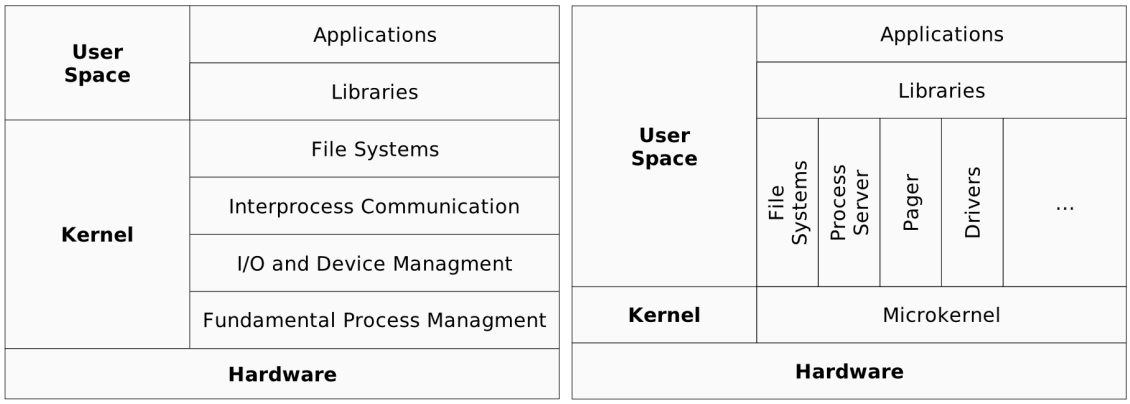
2.2 Kernel design

There are two main kernel architectures, monolithic kernel and microkernel. Most modern kernels - including Linux and NT (Windows) - are monolithic by design [9], whereas GNU Hurd is an example of a microkernel [35].

2.2.1 Monolithic and microkernel

In Linux, basic system services like memory management, process scheduling, file system and I/O communication are all run in kernel space. The kernel is built as layers, as seen in Figure 2a. The inclusion of all basic services directly into the kernel has three major drawbacks: huge kernel size, poor extensibility and bad maintainability [9]. Each time something is added into the kernel or bugs are fixed, the kernel must be compiled and linked again. The different parts of the kernel don't have a stabile API, but rather anyone who wants to, may change it, but then they must also fix the rest of the kernel using that particular interface.

The alternative to this would be to use a microkernel where major parts of the kernel are offloaded into the user space daemon processes as seen in Figure 2b. The kernel would still have basic I/O control and the daemon processes would now need an **IPC** (*Inter Process Communication*) service.



(a) : Monolithic kernel (b) : Microkernel

Figure 2: Two different kernel architectures [9]

2.2.2 Modules

Although Linux is not a microkernel but rather a monolithic one, it still provides a mechanism to load code dynamically using the **LKM**, *Linux Kernel Module* framework, if the support is enabled in the configuration of the kernel as shown in Text listing 6. The linking of external modules is most commonly used for device drivers [3]. In practice this means that the modules are not linked statically but rather dynamically. It is also possible to develop the kernel modules independently without compiling them against the kernel source tree. These two practices are differentiated in this thesis by calling them *in-tree* and *out-of-tree* modules. For instance, Kaiwan N. Billimoria seems to be using this naming convention in [3], but it is hard to find any source stating the truth about this naming convention directly. This information about *in-tree* and *out-of-tree* modules is needed in section 4.3.6.

A kernel module usually consists of one or more C source files, header files and a Makefile since it is built using `make`. The module is compiled into an object file. Until Linux 2.4 the modules had an `.o` suffix, but since Linux 2.6 they have the suffix `.ko` which stands for *kernel object*. [3]

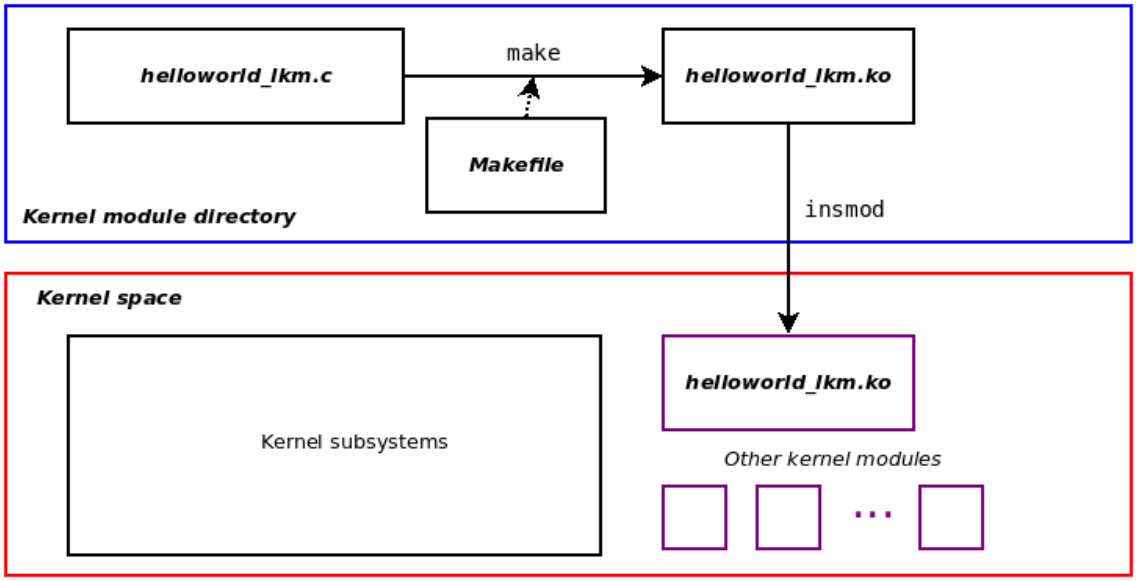


Figure 3: Building and inserting a kernel module into the kernel memory [3]

Not all kernel functionality is provided via the LKM framework, e.g., CPU scheduling or memory management would not be possible to offload into a module. Instead, the kernel modules are most useful for device drivers: not everything has to be compiled statically into the kernel, since each hardware device needs only a limited subset of drivers. These needs can then be loaded dynamically from a larger disk at runtime.

The compiled module can be inserted into the running kernel using the command `insmod` as seen in Figure 3, and it can be removed using the command `rmmod`. The command `modprobe` is a higher level utility, which will find the modules under `/usr/lib/modules/$(uname -r)`. All of these commands are symbolic links to the program `/usr/bin/kmod`, at least on Fedora 38, which is the daily driver OS of the author. The program `kmod` uses internally the `[f]init_module(2)` system calls to load the kernel module into the kernel memory and the `delete_module(2)` system call to unload the module from the kernel memory. It will also call the functions `module_init` and `module_exit` on insertion and removal of the module. [3]

2.3 Kernel contexts

Code can be run in the kernel in different contexts, the interrupt context or process context. Understanding the kernel contexts and their difference is inevitable for section 6.2.

Since Linux is a monolithic kernel, as discussed in 2.2, a process which encounters a system call is switched to privileged kernel mode to execute the code of the system call. There is no separate kernel thread which would take over, thus it is said that the kernel executes within the context of a user space process or thread. This is called the *process context* and a large portion of the kernel code is executed this way, for instance much of the code of device drivers. The concept can be seen in Figure 4 on the left, where a user mode thread issues a system call which is elevated to run the system call kernel

code in the kernel space, in process context [3]. The kernel itself also has some threads called pure *kernel threads*. The kernel threads are very similar to their user space counterparts, they also run in the process context with the distinction that they will not leave their context. They are not able to even see the user *virtual address space*, **VAS**. [3]

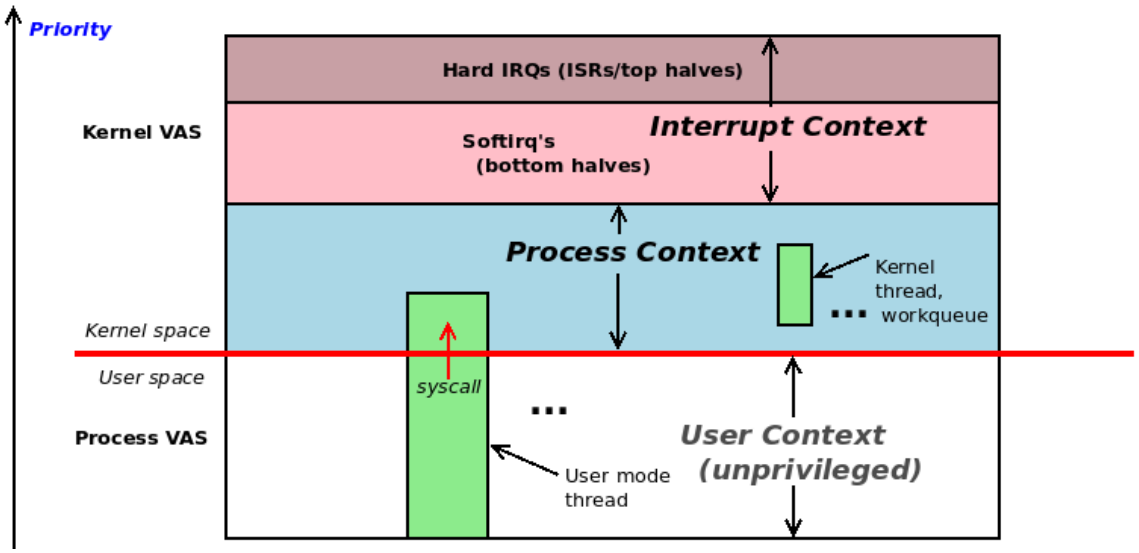


Figure 4: User space and kernel space [3]

There is also another context in which the kernel code may be executed, called the *interrupt context*. When the CPU gets a hardware interrupt from a peripheral device, the CPU will save its running context and switch to an *interrupt service routine*, **ISR**. This causes the kernel code to be run asynchronously, while the process context will always be run synchronously [3]. Code running in interrupt context is just one example of an atomic context [20]. The kernel will enter an atomic context also if a spinlock is acquired or when the code is inside an **RCU**, *read-copy-update* as discussed in section 2.4.

2.4 RCU

Read-copy-update is a synchronization mechanism, which is optimized for read-mostly situations. The basic idea behind the RCU is to be able to update a data structure while others are still accessing and reading it. Understanding what the RCU is and how it works, is essential for section 6.2.

2.4.1 Principle of operation

The RCU splits updates into *removal* and *reclamation* phases [24]. In the removal phase the data structure is updated but nothing is yet removed. Because of modern CPUs semantics, it's safe to modify the data structures while readers still use them - the readers will either see the old or the new version, but never a partially updated one.

Because the update phase is split into removal and reclamation phases, it is possible to perform the removal phase immediately. The reclamation phase

may be entered after all readers, which still hold references to the old data structure, have finished. Only readers, which were active at the removal phase, need to be considered. Any reader who started after the removal phase won't get a reference to the old data structure. The wait may be done as a blocking call or by registering a callback. [24]

Because of this ability to wait until all readers have finished, RCU readers can use a much more light-weight synchronization than traditional lock-based schemes. RCU-based updaters take advantage of modern CPUs atomic insertion, removal and replacement of data items in a linked structure without disrupting readers. [24]

Text listing 1: Example of RCU usage [24]

```
        CPU 0                   CPU 1                 CPU 2
    ---------------- ------------------------- ---------------
1.  rcu_read_lock()
2.                      enters synchronize_rcu()
3.                                                rcu_read_lock()
4.  rcu_read_unlock()
5.                       exits synchronize_rcu()
6.                                               rcu_read_unlock()
```

Text listing 1 shows an example of an RCU in use, where threads on CPUs 0, 1 and 2 all use the same resource at the same time. First, CPU 0 locks the resource with `rcu_read_lock`. Right after that CPU 1 enters the function `synchronize_rcu`, which ends the update-phase and starts the reclamation-phase. Since CPU 0 already has a read-lock on the resource, the call to `synchronize_rcu` blocks until CPU 0 releases its lock with a call to the function `rcu_read_unlock`. In the example CPU 2 also calls `rcu_read_lock` right after CPU 1 entered `synchronize_rcu`, but it won't block the synchronization call, because it entered afterwards and CPU 2 already got the newest references to the data.

2.4.2 Implementation and memory safety

In practice the synchronization is implemented in quite a clever way. The function `rcu_read_lock` compiles down to a single compiler barrier - `asm volatile("":::"memory")` if all the debugging facilities are off [20].

Linux kernel implements the `synchronize_rcu` so that it returns after each of the CPU cores have experienced a context switch at least once. In other words, the kernel implements the RCU in a way which treats RCU critical sections as atomic contexts, and calling sleep will cause a context switch which results in `synchronize_rcu` to return early, which in turn causes the memory to be freed too early leading to a use-after-free memory error.

This means, the RCU implementation in Linux elevates the problem of sleep in an atomic context from "it's bad because it might cause a deadlock" to "it's bad because it can cause a use-after-free" [20].

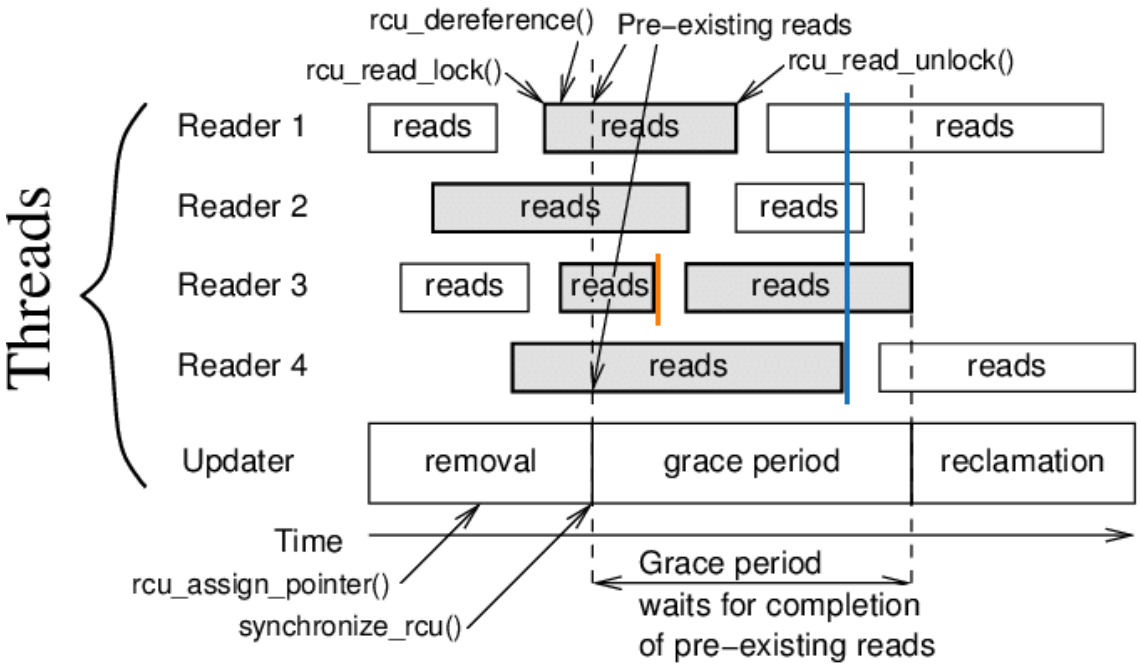


Figure 5: RCU grace period and read time critical section [10]

Figure 5 demonstrates the `synchronize_rcu` and the wait for its completion, the so called grace period. The read side critical sections of the different threads are shown as boxes - each read side critical section starts with the call to the `rcu_read_lock` and ends in a call to the `rcu_read_unlock`. Let's assume one of the threads, e.g., the *Reader 3*, was to sleep in the atomic context, and thus was to experience a context switch. This context switch is visualized as the orange vertical line over the read side critical section of the *Reader 3*.

The function `synchronize_rcu` will return as soon as all readers, which were present at the start of the call to the above-mentioned function, experience the context switch at least once. Because the *Reader 3* already experienced the context switch once, the RCU grace period is ended as soon as all the other readers have experienced their context switches - in this example this condition is met as soon as the *Reader 4* has experienced its context switch.

This hypothetical end of the grace period is demonstrated as the vertical blue line. Because the *Reader 3* is still reading the data structure after the reclamation period has started, a use-after-free memory error may happen.

2.5 NVMe

One of the existing kernel projects is a Rust implementation of the NVMe (technology to use *non-volatile memory* over *PCI Express* [6]) driver, which is introduced in section 5.2. Because of this, a short introduction to the NVMe is given.

NVMe is a storage interface, which is used to communicate with a storage media [36]. The NVMe controller works internally using ring buffers, which are located in the host memory. First a command is submitted to a submission queue, after which the controller has to be informed about the submission. After the controller has executed and completed the command, it will publish the result into the completion queue. The user may poll the queue for changes or the NVMe controller may be configured to emit an interrupt. The controller may support multiple IO-queues.

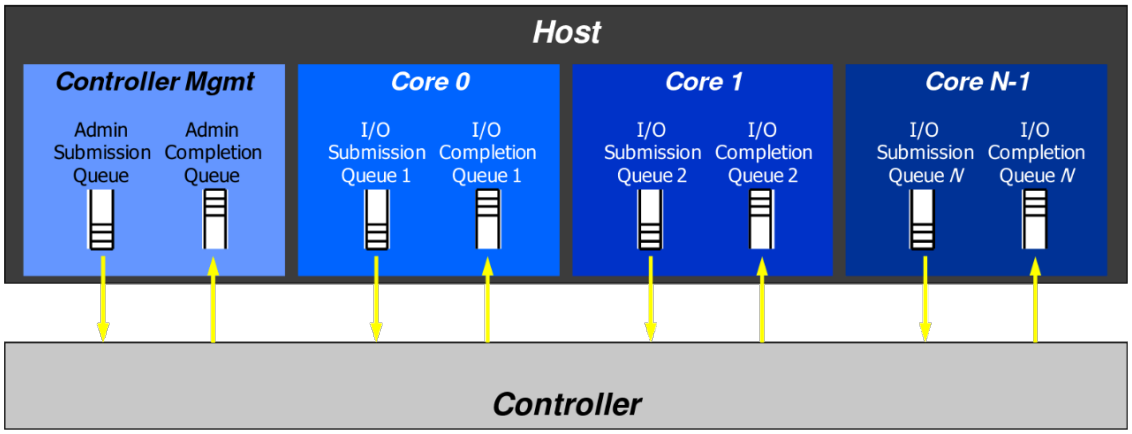


Figure 6: NVMe block diagram [11]

The NVMe driver is widely tested, optimized and deployed. There is no need to replace it, but because of the relative simplicity of NVMe, it is a good opportunity to test Rust in the kernel and introduce the storage community to Rust. It's an interesting target because of the high performance requirements [11], [36].

3 Device drivers

According to Microsoft [37] a driver is the software component which let's the operating system communicate with the hardware and vice versa. This is merely a slender perspective to it, and the book *Linux Device Drivers* [30] takes a deeper approach to the definition. It defines the device driver to be a "black box" which abstracts a particular hardware device to respond to a well-defined internal programming interface. It completely hides the details, how the actual hardware device works. Figure 7 shows the logical location of the device drivers in the kernel space of an operating system.

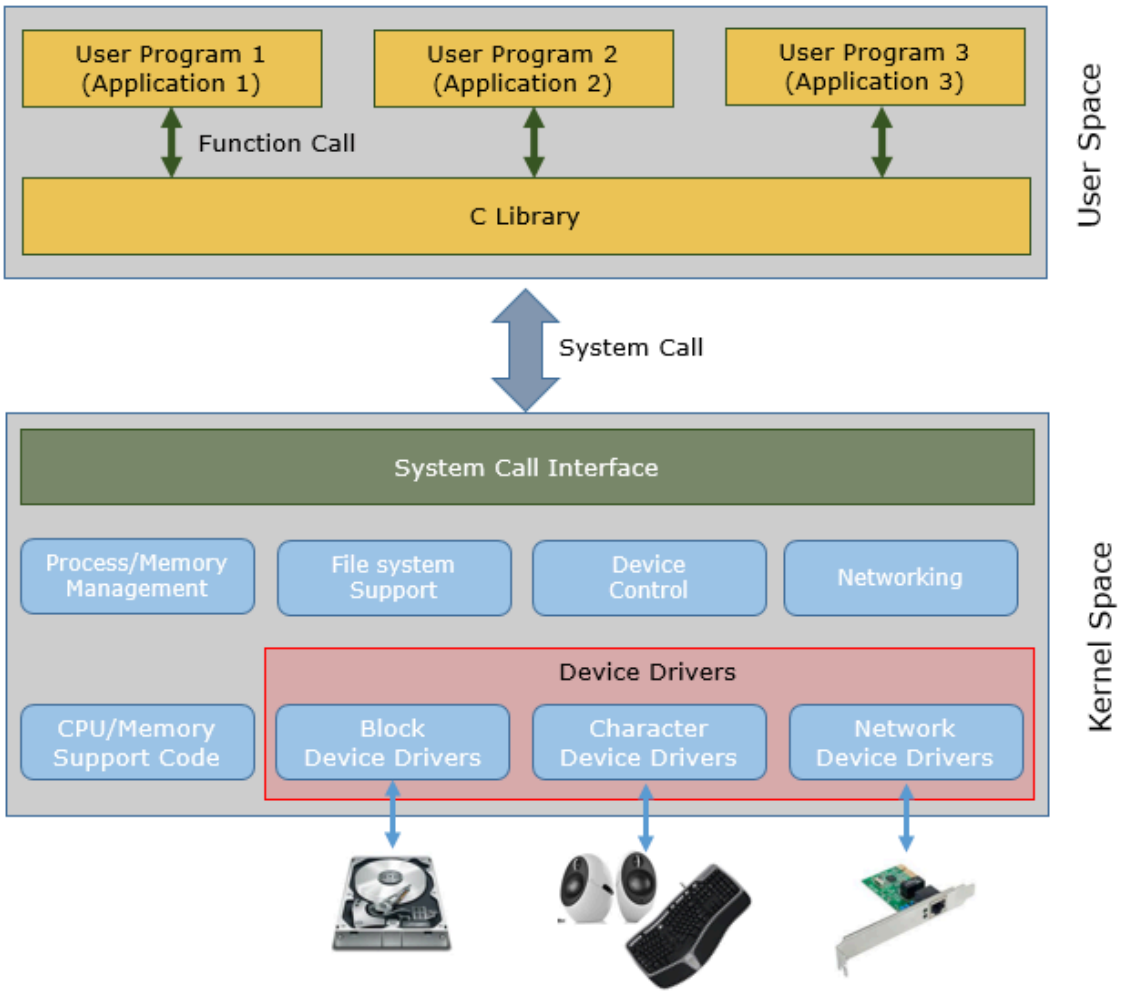


Figure 7: Role of device drivers in an operating system [12]

Many hardware devices are designed to implement a standardized interface, device profile or similar. For example, keyboards and mice implement the **HID** (*Human Interface Device*) USB-protocol. In this case, the hardware device vendor does not need to provide a driver since the driver is already implemented in the OS.

Not all drivers communicate directly with a hardware device but multiple drivers may be stacked, for example, for filtering or data modification purposes. The downmost driver, called a *function driver* in Windows or a *device driver*

in Linux, will communicate with the device itself. In Linux, drivers which don't interact directly with the hardware, are often called *logical drivers* [13] (*filter driver* or *software driver* in Windows). It seems that the naming convention varies a little from source to another. [37]

The Windows driver model is quite different from the model in Linux. In Windows, the drivers communicate with an ABI while in Linux, the drivers do not have a stable API or ABI, but are instead compiled into the kernel. [38]

3.1 Device drivers in Linux

The device drivers will most certainly run with OS privileges, i.e., in kernel context. User space drivers do exist in Linux (**FUSE**, *File system in user space*) [39], but they may suffer from performance issues. Device drivers can be built *in-tree* or *out-of-tree* as described in section 2.2.2.

There are different kinds of device drivers in Linux in which *character device* and *block device* drivers are the most common ones. Character devices handle data byte-wise, which is useful for, e.g., serial ports. Block devices handle data in defined block sizes, as storage devices do.

Text listing 2 is an example of a block device, an NVMe SSD. The text listing is the output of `ls -l /dev/nvme0*`. The leading `b` in the file permissions column shows it is a block device. Respectively, `c` would stand for character device.

Text listing 2: Example of block devices

```
brw-rw----. 1 root disk 259, 0 Sep  2 12:17 /dev/nvme0n1
brw-rw----. 1 root disk 259, 1 Sep  2 12:17 /dev/nvme0n1p1
brw-rw----. 1 root disk 259, 2 Sep  2 12:17 /dev/nvme0n1p2
brw-rw----. 1 root disk 259, 3 Sep  2 12:17 /dev/nvme0n1p3
```

The column after the ownership information is separated by a comma, e.g., `259, 0`. It tells the *major* and *minor* numbers of the driver [22]. The major number tells, which driver a device file uses for accessing the hardware. Similar hardware devices will use the same driver. In case of Text listing 2 all partitions which reside on the NVMe SSD, will use the same NVMe driver. The minor number is meaningful only for the driver to distinguish the different hardware. In this example, this NVMe driver sees the four different partitions as different logical hardware, although they reside on the same physical SSD.

3.2 Device tree

The kernel driver core code organizes all devices in a tree called the *device namespace* or *device hierarchy*: first a distinction is made by block and character devices. Each of these devices are represented using unique *major* numbers. The devices are then further classified using *minor* numbers. In Linux, the major and minor numbers are both stored in a same 32-bit integer, where the 12 MSB (Most Significant Bits, = 4096_{10}) are reserved for the major number

and 20 LSB (Least Significant Bits, = 1048575_{10}) are reserved for the minor number. This tree is visualized in Figure 8. [13]

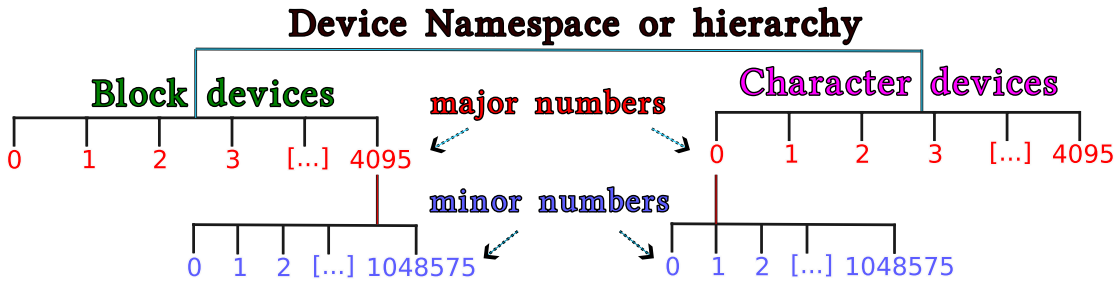


Figure 8: Device namespace or hierarchy drawn accordingly to [13]

The core difference between the character and block devices is that the block devices have direct support from the kernel for mounting file systems, which means that a block device may be attached to an existing, user-accessible file system to extend it. Character devices do not have support for mounting, which is why storage devices like SSDs or USB flash drives are usually block devices. [13]

In this thesis, both character and block devices are discussed. For instance, the NVMe driver in section 5.2 is a block device and the misc device in section 3.5 is a character device.

3.3 Linux Device Model

The basic tenet of **LDM** (*Linux Device Model*) is that every device on Linux must reside on a bus: a USB device will be on the USB-bus, an I2C device will be on the I2C bus. The different buses and devices on them can be found in Linux under the `/sys/bus` directory. [13]

The advantage of this bus system is that it organizes and splits the implementation of device drivers. For instance, a RTC chip which uses the I2C bus does not need to implement the I2C protocol itself, but it can rather resort to an existing implementation of the I2C driver, on which the RTC driver then registers itself. The I2C driver is then responsible for the corresponding bit-banging and the developer of the RTC driver has a neat API to use. [13]

For example, registering an I2C device to the I2C framework would be done using with the function `i2c_register_driver` and registering an RTC would be done with the function `rtc_register_device`.

3.4 Character devices

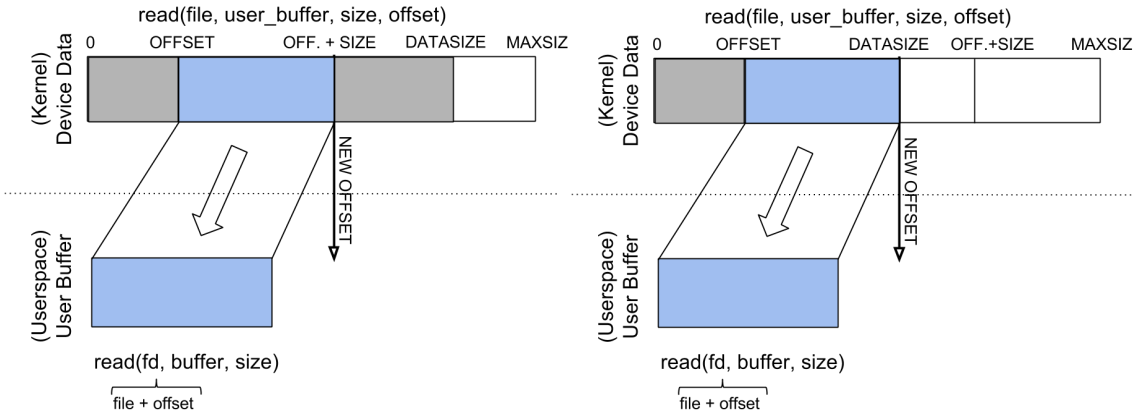
A character device driver is a simple driver, which can be, e.g., read or written one character at a time. Examples of such drivers could be used by serial ports (including USB) or more complex devices like audio interface device drivers or GPU drivers.

Struct `file_operations` is defined at `include/linux/fs.h`, which can contain numerous pointers to functions used as callbacks from the kernel. Some of them are listed in Code listing 1 where `device_read` etc. are the implementations of the device driver. Not all functions will be needed for all drivers, e.g., a display driver doesn't need to read from a directory structure so the corresponding fields can be left as `NULL`. [22]

Code listing 1: Instantiation of the `file_operations` struct [22]

```
struct file_operations fops = {
    read: device_read,
    write: device_write,
    open: device_open,
    release: device_release
};
```

Once the `read`-function has been called, the virtual file system kernel framework calls the corresponding `read`-function given in the struct `file_operations`. Figure 9 illustrates how the `read`-function behaves whether there is enough data to be read or not. By default, it tries to read as many bytes as possible and will return the number of bytes read. If there is enough bytes to be read, the return value will be the same as requested. If there is not enough data to be read, the return value will be positive but smaller than requested. If the data source is exhausted, `read` will return `0`, which is a sign to the user space application to stop reading. A negative return value will indicate a failure. [14]



(a) : Enough data to be read (b) : Data source exhausted

Figure 9: Transferring data with the `read`-syscall from kernel space to user space [14]

Because the buffer will be transferred from the kernel space to the user space, the function `copy_to_user` must be used. Writing works much like reading, but naturally in the other direction and the function `copy_from_user` needs to be used. A return value of `0` means neither bytes were written nor errors occurred, which leads the user space application to retry writing. [14]

3.5 Miscellaneous devices

Because there are only 4096 available major numbers, and there exists quite a few devices which need a kernel driver - think about all the different keyboards, mice, usb-accessories etc - exhaustion of the major numbers is a notable problem. To battle this, a decision way back was made: the major number 10 was assigned to a so called *misc device*, where the minor numbers now act as a second-level major number to distinguish between the different devices [13]. It contains now all sorts of miscellaneous drivers, some of which are listed at [40].

As discussed in section 3.3, every device on Linux should reside on a bus. But not all device drivers will dwell on a physical bus [13], so they will be assigned to a pseudo-bus instead. This is quite common on modern **SoC** (*System-on-a-chip*). The driver itself will be logical, it will not be used directly on any physical hardware but rather as a software driver.

The `miscdevice` API is shown in Code listing 2. It consists of an eponymous struct and two functions, `misc_register` and `misc_deregister`. [23] This API is needed in section 4.5.

Code listing 2: A part of the `miscdevice` C-API [23]

```
1  struct device;
2  struct attribute_group;
3
4  struct miscdevice  {
5      int minor;
6      const char *name;
7      const struct file_operations *fops;
8      struct list_head list;
9      struct device *parent;
10     struct device *this_device;
11     const struct attribute_group **groups;
12     const char *nodename;
13     umode_t mode;
14 };
15
16 extern int misc_register(struct miscdevice *misc);
17 extern void misc_deregister(struct miscdevice *misc);
```

Modern Linux distributions will automatically fill in the device nodes under `/dev`, but they could also be created manually using `mknod`. For instance, to create a misc device (major = 10, minor = 10) one needs to issue command `mknod /dev/mymisc c 10 10`.

4 Linux device driver programming

An objective of this thesis was to compare kernel driver implementation in Rust and C. The kernel drivers are included into the kernel by configuring the kernel, which is demonstrated in the beginning of this chapter. After that, the project *Rust for Linux* is discussed - why Rust is wanted in the kernel, what limitations does it bring and how the integration is implemented.

4.1 Kernel configuration

The Linux kernel can be configured before building. Device drivers may be compiled statically into it or as dynamically loadable modules. Drivers, which are unnecessary for the system, may also be left out. The configuration is done in the `.config` file at the root of the kernel repository, which consists of multiple thousands lines or configuration values. For instance, the build system used in this thesis (appendix A) has 1792 `CONFIG_*` entries.

The configuration file is not intended to be handcrafted but rather advanced tooling exists for it. `make menuconfig` is one of them, the usage of which is shown in Code listing 16.

The program `make menuconfig` works by searching through the directory structure and parsing the kernel configuration files named `Kconfig`. Those files contain code, which is used to create configuration options in the `make menuconfig` program. Those configuration blocks may be boolean or tristate questions, whether a module should be included or not or whether the module should be built as an in-tree or an out-of-tree module. Example of such a configuration block can be seen, for instance, in Text listing 3.

Text listing 3: Sample kernel configuration block for module written in Rust

```
config SAMPLE_RUST_MYMODULE
        tristate "Rust MyModule"
        help
          This option builds my own kernel module as in-tree by choosing 'y',
          out-of-tree module by choosing 'm' and does not link if choosing 'n'.

          If unsure, say N.
```

The configuration blocks will create symbols, which may be used in the `Makefile` to link the compiled object file into the kernel. An example is shown in Text listing 4. Please note that the `Makefile` uses tabs instead of spaces for indentation.

Text listing 4: Sample Makefile block for module written in Rust

```
obj-$(CONFIG_SAMPLE_RUST_MYMODULE)              += rust_mymodule.o
```

The `Kconfig` and `Makefile` will be split recursively into the kernel source code directories, the parent always including the children. This also leads to the source code usually being in the same directory as the corresponding `Kconfig` and `Makefile`.

After configuring the kernel a rebuild is naturally needed. If the build produces also out-of-tree modules (ko-files), these must be included into the initramfs which means building the initramfs again as shown in Code listing 21.

4.2 Necessary kernel configuration

To be able to use Rust in the kernel, some kernel configurations must first be made which are demonstrated in this section. In this section where kernel configuration settings are shown such as in Text listing 5, the text is copied from the terminal-user-interface of the configurator program `make menuconfig`. It contains a feature in which a symbol can be searched for and it will then show the tree traversal where the symbol can be found and which desicions must be made to get it enabled.

First the Rust support must be enabled by setting the symbol `CONFIG_RUST` shown in Text listing 5.

Text listing 5: Kernel configuration to enable Rust support

```
Symbol: RUST [=y]
Type  : bool
Defined at init/Kconfig:1943
  Prompt: Rust support
   Depends on: HAVE_RUST [=y] && RUST_IS_AVAILABLE [=y] && !MODVERSIONS [=n]
&& !GCC_PLUGINS [=n] && !RANDSTRUCT [=n] && (!DEBUG_INFO_BTF [=n] || PA-
HOLE_HAS_LANG_EXCLUDE [=y])
  Location:
    -> General setup
      -> Rust support (RUST [=y])
Selects: CONSTRUCTORS [=y]
```

If the modules are wanted as modules instead of compiling them statically into the kernel, loadable module support must be enabled by setting the symbol `CONFIG_MODULES` shown in Text listing 6.

Text listing 6: Kernel configuration to enable loadable module support

```
Symbol: MODULES [=y]
Type  : bool
Defined at kernel/module/Kconfig:2
  Prompt: Enable loadable module support
  Location:
    -> Enable loadable module support (MODULES [=y])
```

The kernel has also at this point of time two samples regarding Rust kernel modules which are shown in appendix D (`rust_print.rs` in Code listing 26 and `rust_print.rs` in Code listing 27). These can be included into the kernel by enabling `CONFIG_SAMPLES_RUST` shown in Text listing 7 and selecting also the child-selections of this menu item. Appendix C shows also a hello-world kernel module example in C for reference in Code listing 25.

Text listing 7: Kernel configuration to enable Rust samples

```
Symbol: SAMPLES_RUST [=n]
Type  : bool
Defined at samples/rust/Kconfig:3
  Prompt: Rust samples
  Depends on: SAMPLES [=n] && RUST [=y]
  Location:
    -> Kernel hacking
      -> Sample kernel code (SAMPLES [=n])
        -> Rust samples (SAMPLES_RUST [=n])
```

Unfortunately, these two samples are truly trivial and no more complex examples about Rust kernel modules exist on the kernel on the *Rust for Linux* repositorys branch `rust-next` [16]. The branch `rust` has some more examples, but they are not working anymore in newer kernel versions such as `rust-next`. There is more discussion about the branches in section 4.3.4.

4.3 Rust for Linux

The project *Rust for Linux* began officially in 2021 [41]. Adding support for a second language in the kernel is a huge step with high cost and risks. In case the new language adoption turns out to be a failure, it's much harder to drop the support since it means rewriting all modules which have been written with the new language.

Rust is a systems programming language which will bring many key advantages over C in the Linux kernel. Miguel Ojeda, who has been a driving force for the project *Rust for Linux*, discusses the pros et contras of the new programming language in the kernel in his letter to the Linux kernel mailing list [41], from which the next three subsections are cited from.

The goal of support for Rust in the Linux kernel is to write new leaf modules - e.g., device drivers - in Rust rather than to rewrite the whole kernel in it. Not even the smaller subsystems like `mm/`, `sched/` etc. are supposed to be rewritten. Instead, the Rust-support is built on top of those subsystems.

Rust has been officially supported by Linux since version 6.1, which emerged in November 2022, but the support has not been flawless. At the moment of writing this, the Rust support in the kernel version 6.9 is still a work-in-progress. The documentation for the project can be found on [28].

Rust for Linux is supported by the Internet Security Research Group (ISRG) as it supports Miguel Ojeda and previously also Gary Guo through its project *Prossimo* to work on the project *Rust for Linux*. It is made possible by generous financial support from Google and Futurewei [42].

4.3.1 Goals

By introducing Rust to the Linux kernel, the project hopes that the new code added will have less memory safety bugs and data races thanks to the programming language properties themselves. Since the compiler can check

much more than previously, maintainers would be more confident in refactoring and accepting patches for modules. New drivers would also be easier to write, because of abstractions provided by a modern language. Maybe also the usage of a modern programming language makes kernel development accessible for more people, which helps Linux in getting new developers. [41]

4.3.2 Why Rust?

Rust promises no undefined behaviour in its safe subset, which means safe memory management and no data races. Its strict type system will reduce logical errors and the distinction between safe and `unsafe` code makes it easy to give attention to program sections, which need it.

The programming language itself also contains types which don't change the programming paradigm too much. For instance, instead of returning a pointer which may be null, and which always has to be checked for, the Rust program may return an `Option<T>` for which the compiler enforces checking the value for `None`. For error handling a similar way is used: a `Result<T, E>` will be returned, which will be of type `Ok<T>` containing a value of type `T` or `Err<E>` containing the error of type `E` depending on success. This type of error handling helps in finding bugs early on, preventing the kernel from panicking in production just because of a forgotten return value check [43]. Rust is a featureful language with, for instance, pattern matching, generics, **RAII** (*Resource Acquisition Is Initialization*), lifetimes, shared and exclusive references, modules and visibility, powerful macros and so on.

Rust also has a great ecosystem with built-in formatter, linter and document generator. It also has a package manager but installing external packages is not used in the kernel as discussed further in section 4.4. Overall, Rust is a modern systems programming language which has leveraged decades of experience from both other systems programming languages as well as functional languages with new aspects, like lifetimes and borrowing, added.

4.3.3 Why not?

Of course, Rust does still have a few drawbacks compared to C in the context of the Linux kernel. Since the Linux kernel has been written in C, the environment has been polished and multiple projects already exist to help with the kernel development in C.

Rust only has a single implementation, based on the **LLVM** [44] compiler infrastructure. There is an ongoing project to create a frontend for the **GCC** (*GNU Compiler Collection*) [45] to compile Rust but it is still in a very early stage and is not yet able to compile real Rust projects [46].

Since Rust is such a complex programming language, its compilation is also slower than compiling C. It will get better with time as the Rust compilers get better.

Currently only a single version of Rust is supported [47]. The project can not guarantee future versions of Rust working, because of unstable features in use. Unstable in this context does not mean broken features, but rather experimental features, which may or may not be merged into future stable versions [48]. Removing the need for unstable features is a priority to be able to assign a minimum Rust version for the kernel.

In the Email Miguel also raised the point of Rust not being standardized as a negative aspect. However, this issue should now be settled since Rust was standardized in the project Ferrocene [49] at November 2023.

All together, although Rust has some disadvantages, it is notable that it is a much younger and less used language than C. As Miguel states in [41], they believe that Rust will take an important place in systems programming like C has been in an important role for the few last decades. As Rust gets older, many of the issues now will most likely be addressed and improved.

4.3.4 Branches

Rust for Linux was first developed on branch `rust` before the support was initially merged into the kernel, after which this branch was frozen. The purpose of this branch was to work as a prototype on how things could be implemented - some things will be taken into the upstream while others are reworked based on comments [50]. The new development happens principally in the other branches listed in Table 1, where `rust-next` is marked as default, or in the mainline Linux repository [51] as discussed in the kernel lore [52]. The last commit into `rust` was applied on 2023-07-30 which can be verified from Text listing 11 or the kernel repository itself [16].

Table 1: Development branches in *Rust for Linux* and their versions on 2024-05-08 [16]

Branch	Kernel version
`rust-next`	6.9.0-rc1
`rust-fixes`	6.9.0-rc6
`rust-dev`	6.9.0-rc1
`staging/rust-pci`	6.9.0-rc3
`staging/rust-net`	6.8.0-rc1
`staging/rust-device`	6.9.0-rc1
`staging/dev`	6.9.0-rc1
`rust`	6.3.0

Among the new branches `rust-dev` is the experimental branch for integrating new features and finding out bugs as soon as possible before they are merged into `rust-next`, which in turn is the branch containing the code to be submitted within the next merge window of Linux. The branch `rust-fixes` contains patches for fixing bugs in Rust on the current cycle of the Linux kernel. Both branches `rust-next` and `rust-fixes` are part of the `linux-next` branch.

The `staging/` branches are all newer and emerged while writing this thesis. They are meant for particular topics to enable code collaboration, when the code has not yet reached the mainline. The branch `staging/rust-net` is dedicated to networking-related abstractions and the branch `staging/rust-pci` for PCI related projects, such as the NVMe driver introduced in section 5.2. The branch `staging/rust-device` is meant for device driver related abstractions.

The branches `staging/rust-net` and `staging/rust-device` are regularly rebased on top of `rust-next`. Just like `rust-dev` is the integration branch for the main branhes, the branch `staging/dev` is intended for integration for the staging branches. [50]

The awareness of the different branches in *Rust for Linux* is prerequisite for the section 8.1.

4.3.5 Backporting to older versions

Because *stable long term support* (LTS) kernel releases only receive fixes and do not accept any new feature, it is not possible to backport Rust support into them. The cost of maintenance would also be significant. Nonetheless, there still is some interest to get Rust support merged into some stable/LTS kernel versions, at least 5.15 and 6.1. [53]

There are still many questions regarding the possibility of supporting an older stable / LTS release of Linux on the *Rust for Linux*'s side. For example, whether the support level of the backporting should be only best-effort or should it be production ready? Should all abstractions be backported, which appear in the mainline? How about the ones, which also need backporting other infrastructure on the C side of the kernel for them to work? There is also the question about Rust version policy: should there be a same, fixed version for the mainline version and the LTS version or could they use different Rust versions? Lastly, the workflow raises a question: should the patches be provided on a rebase-branch, or should a completely separate stable branch be used? [53]

4.3.6 Out-of-tree modules

As discussed in section 2.2.2, Linux supports both statically linked and dynamically loaded kernel modules. The dynamically loaded modules can further be divided into in-tree and out-of-tree built modules [3]. Both C and Rust modules can be built as either ones, and the project *Rust for Linux* provides basic templates for both approaches [54].

The *Rust for Linux* project is a part of the kernel and has always focused on bringing code into the mainline kernel. The internal API of the Rust support is not stable, just like the internal API in the Linux kernel C side may also change when necessary.

All patches accepted must contribute and be used by in-tree users. Abstraction for out-of-tree users will not be merged according to [54], not even if they would be useful in the future for in-tree users as well.

4.3.7 Fallible memory allocation

The computer may run eventually out of memory. Because the kernel is responsible for allocating and sharing the memory between the user space processes, it is also responsible for situations where memory runs out. User space programs may simply be killed with the OOM-reaper - after all, the health of the kernel is much more important. But once a kernel process needs to allocate memory - which has run out - the process must accept an allocation failure as an answer.

Since the kernel itself is responsible for letting the memory go out, it can also fix the problem itself. The kernel could, e.g., swap rarely used memory pages onto disk or kill the user space processes which are using too much of the memory. The kernel alone decides the memory allocation, which is why the memory allocation inside the kernel must be graceful. [19]

4.4 External Rust libraries

The package manager `cargo` [55] is not only used for downloading and installing Rust library packages, resolving their dependencies but also used as the build tool for Rust projects. It can, however, also be used to run other tools as `clippy` or `rustfmt`.

In kernel however, there are no external libraries [3]. Therefore, in the project *Rust for Linux* there is no need for Cargo as a package manager. Instead, the development workflow is united with the traditional way of kernel development, and everything is built using `make`.

The documentation for the project *Rust for Linux* [56] lists the actual requirements if an external library could be used. First, the library must only depend on `core` and `alloc`. Secondly, they must also have a compatible license and thirdly, they must support fallible initialization. A few popular crates have already been provided for interested users. For example, `proc-macro2`, `quote`, `syn`, `serde` and `serde_derive` were added in pull request [57].

4.5 `bindgen`

The support for *Rust for Linux* has been possible thanks to **FFI**, the *Foreign Function Interface*, which makes it possible to call functions written originally in C from Rust code and vice versa. The external function annotations must be written in `extern` code blocks. [58]

`rust-bindgen` is a project, which is used to generate Rust FFI bindings from C and some C++ header files [59]. It takes a C-header as input and provides the corresponding abstractions as Rust structs and extern function declarations. This allows the user to create structures in Rust, which are defined in C or call functions from Rust code, which are declared and defined in C. [60]

In praxis, the corresponding C-header of the desired Linux API must be declared in `linux/rust/bindings/bindings_helper.h` which is inside the Linux source directory. For example, if the miscellaneous character device shown in Code listing 2 is wanted, the line `#include <linux/miscdevice.h>` must be added into the aforementioned file.

Code listing 3: A part of the `miscdevice` API generated from Code listing 2 using `bindgen`

```
#[repr(C)]
#[derive(Copy, Clone)]
pub struct miscdevice {
    pub minor: core::ffi::c_int,
    pub name: *const core::ffi::c_char,
    pub fops: *const file_operations,
    pub list: list_head,
    pub parent: *mut device,
    pub this_device: *mut device,
    pub groups: *mut *const attribute_group,
    pub nodename: *const core::ffi::c_char,
    pub mode: umode_t,
}
impl Default for miscdevice {
    fn default() -> Self {
        let mut s = ::core::mem::MaybeUninit::<Self>::uninit();
        unsafe {
            ::core::ptr::write_bytes(s.as_mut_ptr(), 0, 1);
            s.assume_init()
        }
    }
}
extern "C" {
    pub fn misc_register(misc: *mut miscdevice) -> core::ffi::c_int;
}
extern "C" {
    pub fn misc_deregister(misc: *mut miscdevice);
}
```

Bindgen will then generate and write the Rust bindings into a file located at `rust/bindings/bindings_generated.rs`. The file contains all bindings generated for all APIs so it is quite long - about 87 KLOC (thousands of lines of code) on the author's development environment. A short section, the relevant part for the `miscdevice`, is shown in Code listing 3.

`bindgen` will use `#[repr(C)]` to ensure the binding struct generated in Rust has exactly the same layout and order of fields than the struct written in C. This ensures, the code can be linked and FFI can be used as intended [61]. Because the FFI interface can only be used from `unsafe` code, it is custom to create a safe-wrapper so the interface can be used from safe Rust. [62]

`bindgen` was split into two parts: the core library `bindgen` and a command line tool `bindgen-cli` in pull request [63]. This is notable because in the kernel version 6.3 the crate `bindgen` must be installed, but in the kernel version 6.8 the crate `bindgen-cli` must be installed. The reason for the split was that the bindgen should also be easy to be used as a library.

5 Existing implementations

There already exists some projects, which use Rust in the Linux device driver code and the number of projects is only increasing. The most interesting projects are discussed in this chapter. For now, only the Ethernet driver in section 5.1 is merged into the Linux mainline.

5.1 Fast Ethernet driver for ASIX AX88796B

“Linux is well known for its precise, to-the-letter-of-the-RFC, high-quality implementation of the well-known (and not-so-well-known) network protocols at all layers of the model, with TCP/IP being perhaps the most famous.” states K.N. Billimoria in [3] praising Linux’s renowned networking. At the time of writing this on 2024-05-08, the AX88796B driver is the first and only driver written in Rust in the Linux mainline repository [51].

Linux 6.8 finally involves a patch set, where it is also possible to write networking device drivers in Rust. The first implementation is a device driver for ASIX PHY driver for the AX88796B, which is a 100M Fast Ethernet controller. Although it is only Fast Ethernet, it is still in common use, for instance in industrial applications. It’s not the most exciting networking hardware, and it already has been supported by a C driver, but this project is a great start and involves also some infrastructure on which other networking drivers can be based on. [64]

5.2 NVMe driver

The Linux kernel already has an implementation of the NVMe driver in C, but a project has also been active in rewriting the driver in Rust. It was originally authored by Wedson Almeida Filho, but is now maintained by Andreas Hindborg at Samsung.

The NVMe driver's purpose is to provide a vehicle for developing safe Rust abstractions for implementing high performance device drivers in Rust as stated in [65]. It's not production ready, and only partially implemented. However, it shows great performance when compared with the highly optimized driver in production written in C. At the time of writing this thesis, the NVMe driver written in Rust is not yet in the Linux mainline.

5.2.1 Linux 6.7 Rebase Performance

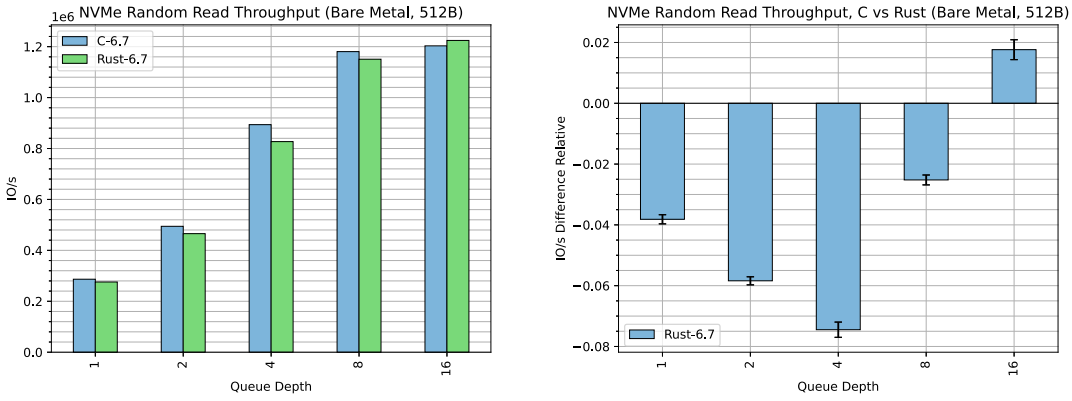
Performance metrics are referenced from [65].

Setup

- 12th Gen Intel(R) Core(TM) i5-12600
- 32 GB DRAM
- 1x INTEL MEMPEK1W016GA (PCIe 3.0 x2)
- Debian Bullseye user space
- LTO results are enabled by a build system patch (a hack) that was not yet published.

Results

- 30 samples
- Difference of means modeled with t-distribution
- P99 confidence intervals



(a) : Random read throughput (b) : Relative random read throughput

Figure 10: NVMe Rust vs C driver performance measurements for Linux 6.7 rebase

The relative difference in Figure 10b is calculated by using the formula `(mean_iops_r - mean_iops_c) / mean_iops_c`.

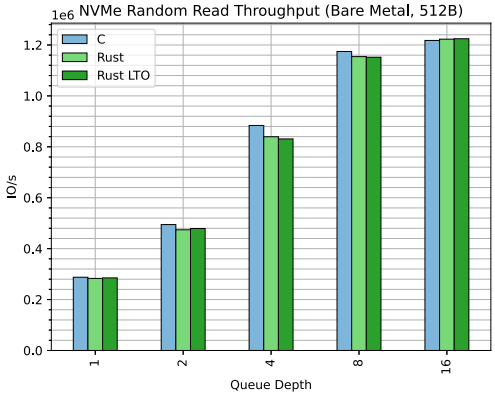
5.2.2 Performance November 2023

The tests were done using the version `nvme-6.6`. Performance metrics are referenced from [65].

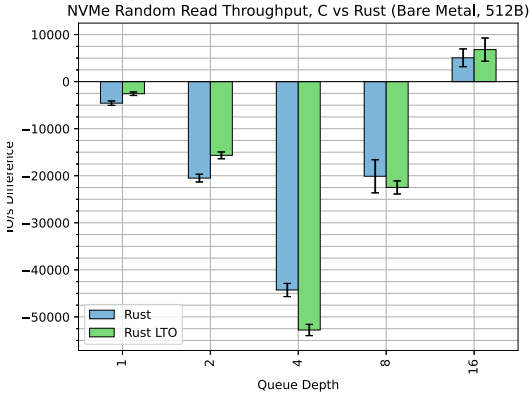
Setup

- 12th Gen Intel(R) Core(TM) i5-12600
- 32 GB DRAM
- 1x INTEL MEMPEK1W016GA (PCIe 3.0 x2)
- Debian Bullseye user space
- LTO results are enabled by a build system patch (a hack) that was not yet published

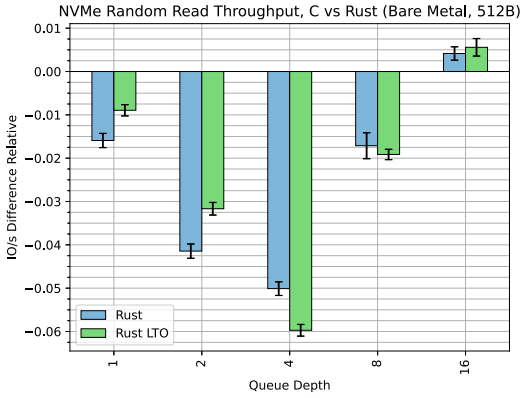
Results



(a) : Random read throughput



(b) : Difference random read throughput



(c) : Relative difference

Figure 11: NVMe driver performance measurements for November 2023

The relative difference in Figure 11c is calculated by using the formula `(mean_iops_r - mean_iops_c) / mean_iops_c`.

5.2.3 Performance September 2023

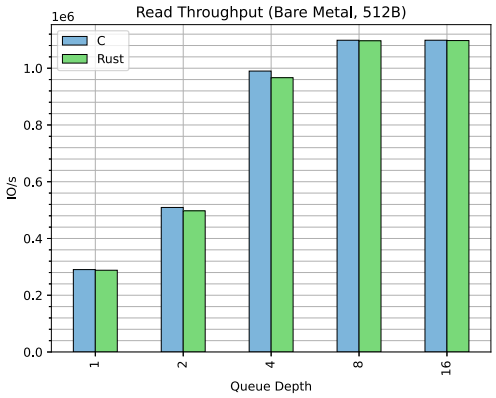
The driver was rebased on top of `rust-next` pull request for 6.6 in September 2023. Performance metrics are referenced from [65].

Setup

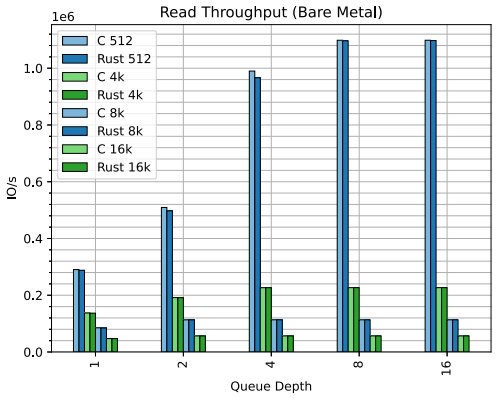
- 12th Gen Intel(R) Core(TM) i5-12600
- 32 GB DRAM
- 1x INTEL MEMPEK1W016GA (PCIe 3.0 x2)
- Debian Bullseye user space

Results

In the figures Figure 12a, Figure 12b, Figure 13a and Figure 13b the Y-axis illustrates the input / output operations per second with the function of queue depth, which is how many hardware dispatch queues the NVMe-driver uses.



(a) : Read throughput 512B [65]



(b) : Read throughput [65]

Figure 12: NVMe driver performance measurements for September 2023

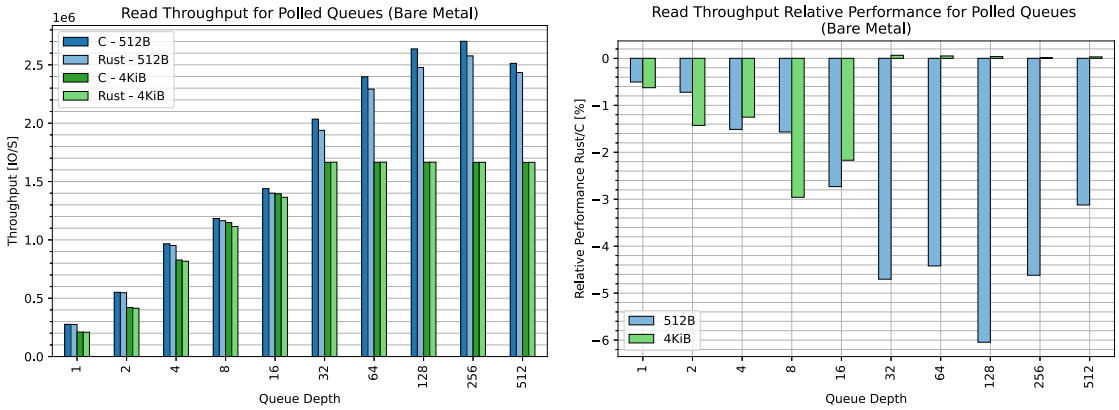
5.2.4 Performance January 2023

Performance metrics are referenced from [65].

Setup

- Dell PowerEdge R6525
- 1 CPU socket populated - EPYC 7313, 16 cores
- 128 GB DRAM
- 3x P5800x 16GT/s x4 7.88 GB/s (PCIe 4)
- Debian bullseye (linux 5.19.0+)

Results



(a) : IOPS [65] (b) : Relative performance [65]

Figure 13: NVMe driver performance measurements for January 2023

5.2.5 Analysis

Figure 12b, Figure 13a and Figure 13b show, that for 4KiB block size the results are quite unamous - the Rust NVMe driver performs almost as well as the original driver written in C. [65]

But for smaller block size of 512B, the C driver outperforms the Rust driver by up to 6%. Unlike in the 4KiB block size, the driver is not bandwidth limited but the Rust-driver rather has a higher overhead, which is why it performs worse.

5.2.6 Work to be done

Future work is also listed at [65], which includes, for instance, removing all `unsafe` code and supporting NVMe device removal. The DevOps side could also be improved by integrating `blktests` and `xfstests` in the CI. The driver is also not yet configurable by `nvme-cli`, a sys-fs node should be included for this purpose. More kernel configurations could be supported by deferring the initialization to a task queue, and lastly the overall performance could still be improved.

5.3 Null block driver

The null block device (`/dev/nullb*`) is used for benchmarking the various block-layer implementations in kernel. It does not execute any read/write operations, just mark them as complete in the request queue [66].

The `null_blk` is a fairly simple driver written in C, which makes it ideal for evaluating the Rust bindings for the block layer [17]. It is also usually not deployed in production environments, which makes it ideal for testing since potential issues are not going to disrupt important environments.

The replacement driver `rnull` is written entirely in Rust, with all `unsafe` code contained in the abstraction layer, which wraps the C APIs [17]. At the point of writing this, the `rnull` is not yet in the mainline kernel but rather an RFC has been made by Andreas Hindborg [67]. The source code resides in the repository at [68].

According to Andreas Hindborg, out of the 256 commits for the existing `null_blk` driver 27% were bugfixes. Out of those bugfix commits, 41% were related to memory safety issues [69]. Those bugs could have been avoided, had the software been written in Rust.

5.3.1 Feature implementation status

Table 2: The `rnull` driver feature comparison against the original `null_blk` driver [17]

Features implemented by `rnull`	Features in C `null_blk` not implemented by `rnull`
• blk-mq support • Direct completion • SoftIRQ completion • Timer completion • Read and write requests • Optional memory backing	• Bio-based submission • NUMA support • Block size configuration • Multiple devices • Dynamic device creation/destruction • Queue depth configuration • Queue count configuration • Discard operation support • Cache emulation • Bandwidth throttling • Per node hctx • IO scheduler configuration • Blocking submission mode • Shared tags configuration (for >1 device) • Zoned storage support • Bad block simulation • Poll queues

5.3.2 Changelog

Short changelog for comparing the three versions of `rnull` used in the tests in 5.3.4 is referenced from [17].

Changes from 6.6 to 6.7

- Move to Folio for memory backing instead of Page
- Move to XArray for memory backing instead of RaddixTree

Changes from 6.7 to 6.8

- Slight refactoring of patch order
- Change lock alignment mechanics
- Apply reference counting to Request
- Drop some inline directives

5.3.3 Test setup

The setup for all three performance tests Figure 14, Figure 15 and Figure 16 is the same:

- 12th Gen Intel(R) Core(TM) i5-12600
- 32 GB DRAM
- Debian Bullseye user space

The relative differences in all three tests Figure 14, Figure 15 and Figure 16 are drawn using the formula `(mean_iops_r - mean_iops_c) / mean_iops_c`. For the Linux 6.8 rebase in Figure 14 five samples were used, for the other two, Linux 6.7 (Figure 15) and Linux 6.6 (Figure 16) 40 samples were used. Differences were modeled using t-distribution and a P95 confidence interval was used for each of the tests.

The upper row for all three tests Figure 14, Figure 15 and Figure 16 is random read and the bottom row random write. The four columns are queue depths (`qd`) which is basically how many IO operations the OS will make before waiting for a response.

5.3.4 Test results

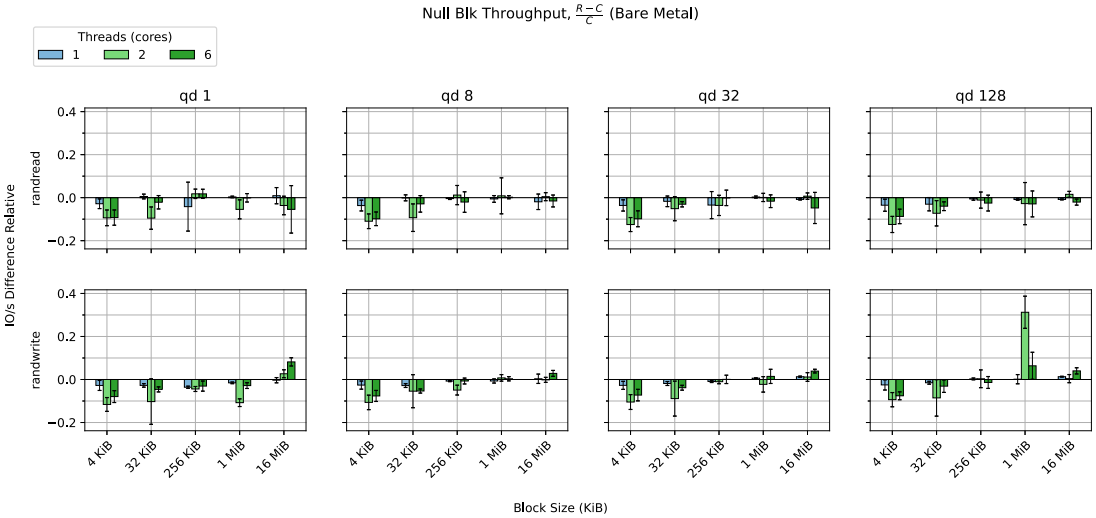


Figure 14: Linux 6.8 rebase performance metrics

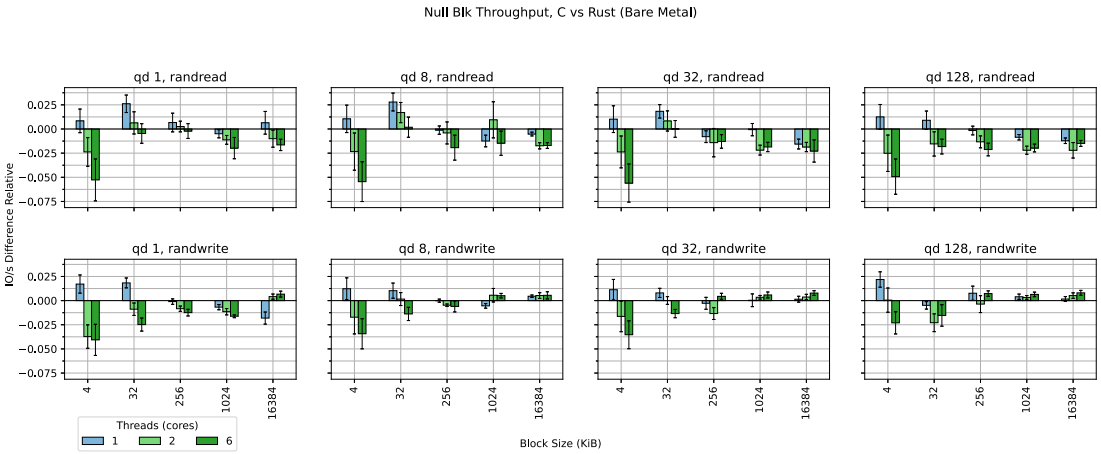


Figure 15: Linux 6.7 rebase performance metrics

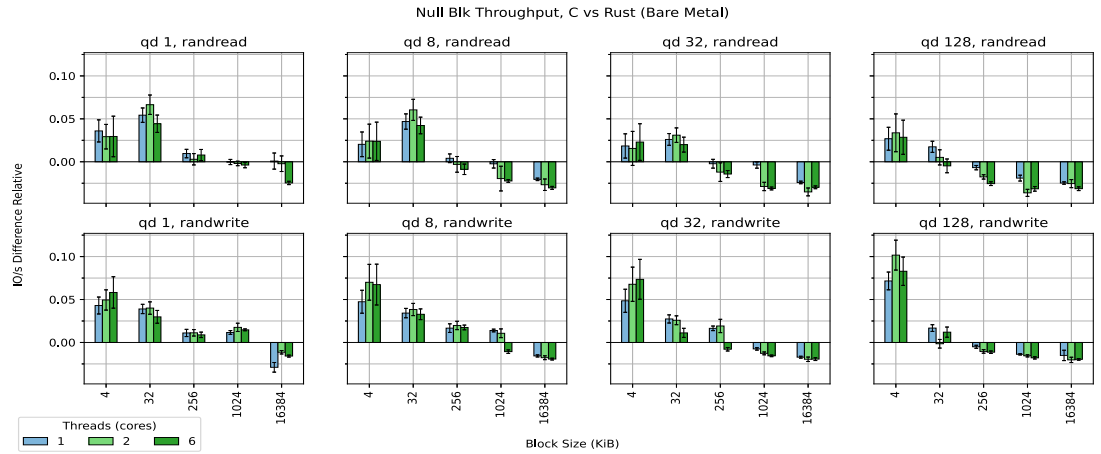


Figure 16: Linux 6.6 rebase performance metrics

5.4 PuzzleFS file system driver

PuzzleFS is a container file system designed to address the limitations of the existing **OCIv1** (*Open Container Initiative version 1*) format. The main goals for the project are to reduce duplication in images, reproducible image builds, direct mounting support and memory safety guarantees [27]. The *PuzzleFS* has taken inspiration from OCIv2 design document [70]. The *PuzzleFS* user space software can be installed and used according to the instructions in appendix B.2.

5.4.1 Problems with the traditional OCIv1 format

Most of the problems originate from the fact that the traditional OCIv1 format uses internally `tar`, which is quite simple. The `tar` format does not support seeking before extraction, which makes partial extraction unnecessary expensive [71]. The `tar` format is neither reproducible, since it does not define the file order which is left to the program traversing the file system - e.g., in inode order. *PuzzleFS* fixes this by introducing a canonical representation for the image format.

Reduced duplication has been obtained by using the **FastCDC**, a *Content Defined Chunking* algorithm. This chunking allows unmodified chunks to be shared between layers more efficiently, since offsetting the content by inserting data at the beginning does not modify the content for the rest of the chunks [18].

Table 3: *PuzzleFS* size comparison with traditional OCIv1 container format [18]

Type	Total size (MB)	Average layer size (MB)	Unified size (MB)	Saved (MB) / 766 MB
OCIv1 (uncompressed)	766	77	766	0 (0%)
PuzzleFS (uncompressed)	748	74	130	635 (83%)
OCIv1 (compressed)	282	28	282	484 (63%)
PuzzleFS (compressed)	298	30	53	713 (93%)

Although OCIv1 images are almost never used uncompressed, *PuzzleFS* still outperforms `tar`-based OCIv1-images while being uncompressed, by saving 83% of storage space. After utilizing compression in *PuzzleFS*, the saving of storage space is 93%. This storage saving has nothing to do with it being written in Rust, but rather in design decision.

5.4.2 Status

The project is a work-in-progress, but building, extracting and mounting a *PuzzleFS* file system already works. *PuzzleFS* supports fs-verity, but it requires file system support from the underlying data store. Optional zstd compression for data blobs is also possible. [27]

Because of memory safety guarantees, *PuzzleFS* has been written in Rust. Currently the file system is implemented as **FUSE**, *file system in user space*, but a read-only kernel file system driver is under development - it's at proof-of-concept level and not yet upstream [27].

5.4.3 Obstacles

The project tries to evade from writing the same code twice but some of the code must still be duplicated, because the kernel has its own build system. No third party crates can be added to the kernel from *crates.io* directly, like in user space applications. The *PuzzleFS* needs a few dependencies, namely `capnproto-rust` for metadata serialization and `hex` for working with hex-strings [18]. Integrating third-party crates is not trivial, `no-std` support is required to begin with as discussed in section 4.4.

The kernel Rust API is still missing many abstractions, because the infrastructure is immature and still under development. It makes writing a file system driver difficult. Also the file operations API is quite different in kernel space than in user space. [18]

The kernel only allows a fallible allocation API. For instance, with the function `try_new` the programmer needs to deal with the out-of-memory error. With the function `new`, the **OOM** (*Out Of Memory*) reaper kills processes to free memory on the system, as discussed in 4.3.7. In the kernel space, running out of memory is unacceptable and must be handled gracefully. [18]

5.5 Android Binder IPC

The Binder is, for example, responsible for **IPC**, *Inter Process Communication* through which applications (processes) may communicate with each other. Android also isolates each application from the operating system by assigning each of the apps a unique user ID (UID). The Binder is also responsible for the so-called *application sandboxing* and is therefore a core component in the Android Platform Security Model. The Rust driver was originally written by Wedson Almeida Filho, and is now maintained by Alice Ryhl for the AOSP, Android Open Source Project [72]. The code is located in [73].

In practice, the Binder has quite a broad spectrum of responsibilities in the Android OS. It must deliver the right transactions to the right threads, parse and translate complex contents containing different types (pointers, file descriptors) for transactions, which may interact with each other. The Binder is also responsible for controlling the user space thread pool sizes and assures that the transactions are assigned to threads without causing deadlocks, would the threadpool run out of threads. The Bindgen is also responsible for tracking the reference counts for objects, which are shared between multiple processes. These all include also proper error handling for numerous scenarious with 13 different locks, 7 reference counters and atomic variables. Of course, all of this should happen as efficiently as possible. [15]

The Binder has been advancing for a long time to meet the expectations of Android and because the requirements, expectations and complexity has grown significantly, the codebase has gained technical debt making the maintenance of the Binder difficult. In addition, becuse of the key role of the Binder in the Android platform security, it is crucial that it is robust against security vulnerabilities. [15]

For these reasons, the Binder was rewritten in Rust. Usually nobody wants to rewrite existing and working implementations but in this case the whole codebase needed an overhaul. Rust was a good choice of a programming language, since the language itself contained multiple features which directly address problems solved in the Binder. Rust prevents mistakes in reference counting, locking, bounds checking and contains a neat error handling mechanism. Additionally the more expressive type system makes it easier to handle, e.g., ownership semantics of structs and pointers. It takes the burden away from the programmer of managing object lifetimes, reducing the risk of use-after-free errors. [15]

Although Rust usually requires more lines of code than C on things left implicit in C, for example invariants, the Binder written in Rust is only 5.5 KLOC whereas the Binder written in C is 5.8 KLOC. This rewrite completely restructures the code, but the actual functionality of the driver stays the same - a lot of thought has gone into that beforehand and there are not any fundamental changes. The `binderfs` file system driver was not rewritten in Rust, for it has not had the same problems as Binder. Rewriting it in Rust would require many bindings to the Rust file system interface. [15]

The new Binder written in Rust passes all tests which validate the correctness of the Binder in the AOSP. Currently the Rust Binder implements all features also supported by C Binder. Some debugging facilities are still missing, but they are added before the implementation is submitted upstream. [15]

5.5.1 Performance

The drivers have been compared using *binderThroughputTest* shown in Figure 17 and *binderRpcBenchmark* shown in Figure 18. The benchmarks indicate, that the Rust implementation has very promising results although much more testing is still needed to see how the driver performs out on the field.

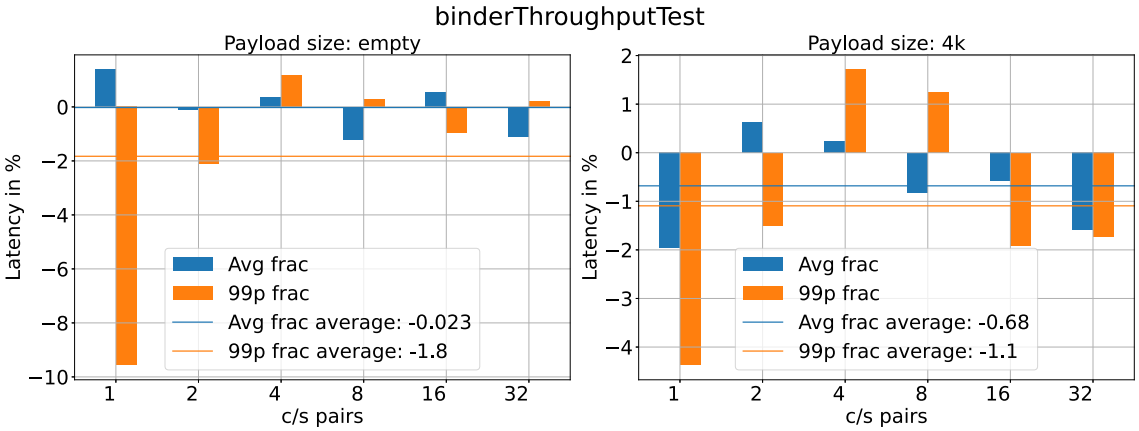


Figure 17: Comparison of Binder C- and Rust implementations using *binderThroughputTest*, graph drawn based on tables at [15]

Figure 17 represents the relative roundtrip latencies of transactions as measured by *binderThroughputTest* [15] as a function of client-server pairs. It is calculated by formula `(perf_rust - perf_c) / perf_c` where `perf_rust` and `perf_c` are the times taken for the Binder's Rust- and C implementations roundtrip latencies for the given tasks. The measurements are done for empty payloads and 4k sized payloads.

In the graphs, negative results are better for the Rust implementation. The Rust implementation seems to have similar performance results to the C-implementation, the average latencies fluctuating between −1.96% and +1.38%. [15]

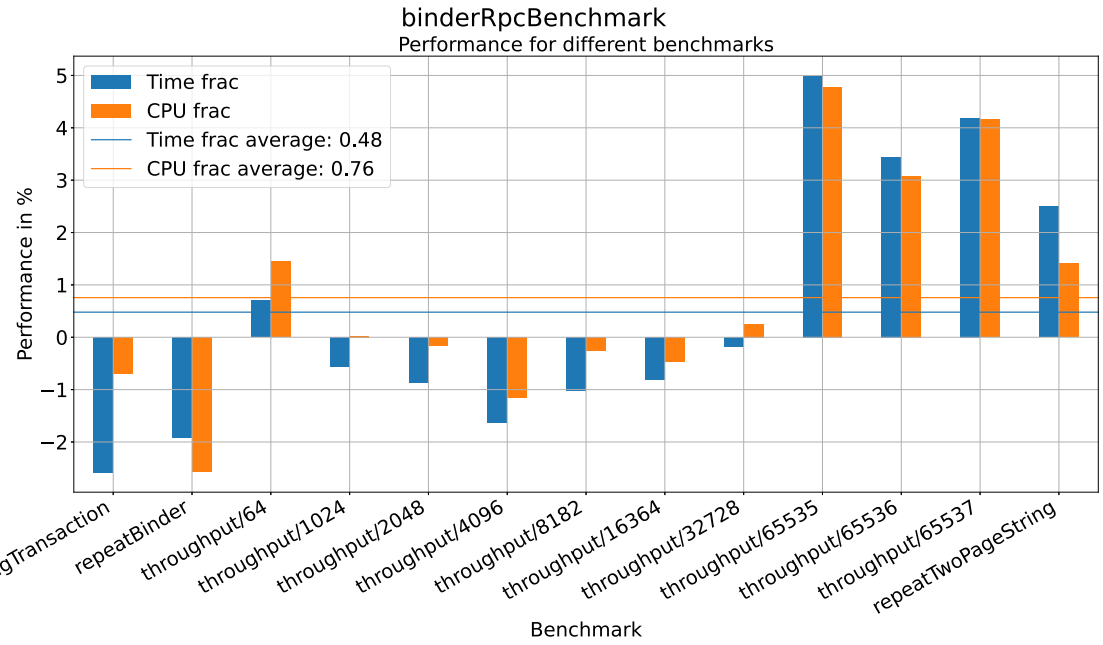


Figure 18: Comparison of Binder C- and Rust implementations using *binderRpcBenchmark*, graph drawn based on table at [15]

Figure 18 depicts comparison of different RPC benchmarks in the Rust and C-implementations of Binder. Again, negative results are better for Rust. The Rust implementation seems to get along great for all benchmarks except for the throughputs with the biggest payload sizes. However, the authors are convinced they can optimize it further with some work. Also, the use cases for the biggest payload sizes are really rare in practice. [15]

5.6 Apple AGX GPU driver

Asahi Linux is a project which aims to port a full Linux desktop onto Apple Silicon Macs, starting with the 2020 M1 Mac Mini, MacBook Air and MacBook Pro [74]. The goal is not only to get Linux running on these devices, but to refine the user experience to the point where Linux can be used as an everyday OS. Since Apple does not support Linux and the hardware is completely new, kernel drivers for devices like the GPU [75], sound card [76], trackpad, USB, DisplayPort, and Thunderbolt must be written. Because the hardware is proprietary and the documentation for it is nonexistent, writing the drivers requires reverse engineering the truly complex hardware, which is really time consuming. A custom hypervisor was implemented, on which the macOS was run to log all hardware accesses for helping the reverse engineering [77]. Asahi Linux is still out-of-tree [51], [78] and lives in its own repository at [79].

5.6.1 Firmware interface

The GPU for the M1 is quite different from a traditional GPU. Instead of the kernel driver communicating directly with the GPU, the kernel driver communicates with a secondary ARM64 CPU running a real-time operating system called RTKit. The firmware is responsible for, e.g., power management, command scheduling, preemption, fault recovery, performance counters, statistics and temperature measurements [75]. This makes the communication easier since many timing critical tasks - like preemption - are offloaded to the firmware. But since the GPU hardware is still a certainly complex system, communication with the firmware still needs hundreds of structs with numerous fields, which are passed through a shared memory segment.

5.6.2 Magic of Rust

Asahi Lina, the author of the GPU kernel driver chose to use Rust, because writing complex data structures is notoriously difficult in C. A single wrongly composed data structure will most likely crash the firmware after which a complete system reboot is needed - making the development unnecessary difficult. Also, manual lifetime tracking of the data structures would be required in C, which Rust is capable of doing automatically at compile-time. [75]

Besides, the interface between the firmware and the kernel driver may change at any time, so the kernel driver must support multiple versions. In C, supporting different interfaces would require the usage of `#define` preprocessor macros in the data structures requiring multiple compilations. In Rust, these different versions can be modeled using the powerful macro system [75].

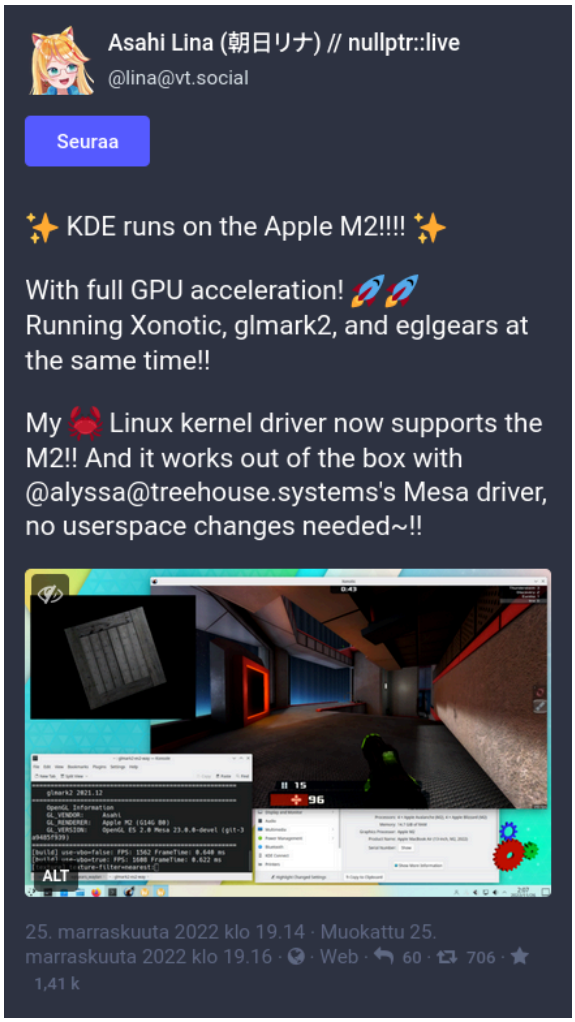
Because Asahi Linux is the first GPU driver written in Rust, the driver also needed a safe API for the Linux DRM graphics subsystem, which required 1500 lines of code. This naturally required also a lot of thinking and rewriting. [75]

Remarkable is also the amount of debugging while developing a this complicated task. Normally there would be all kinds of problems like race conditions,

memory leaks, use-after-free memory errors and other problems, but according to [75] Asahi didn't have those kinds of problems almost at all - there were only a few logic bugs to fix. Rust's safety measures guarantee the driver is safe from thread- and memory issues - as long as the unsafe-wrappers don't have any issues. The compiler guides the programmer to a good and safe design.

In Figure 19b Asahi Lina demoes Xonotic, glmark2 and eglgears running on a KDE Plasma session on an Apple M2.

(a) : Logo [74]



(b) : Asahi Lina on Mastodon

Figure 19: Asahi Linux

5.7 Nvidia GPU drivers Nouveau and Nova

Nvidia is developing GPUs in the same direction as Apple developes its new M series GPUs as described in section 5.6. Nvidia uses a new RISC-V processor called **GSP** (*GPU System Processor*), which is used for commanding the GPU itself. The GSP abstracts complex timing critical tasks away and provides a nice API for the OS GPU driver. Nvidia allows Linux to use that firmware, which makes it possible to reclock the GPU, running it in full speed with the Nouveau open source drivers, which was not possible before. [80]

There also exists disadvantages to this design, which are not that far away from the Apple AGX design. The firmware provides no stable ABI and a lot of the calls are not documented. Mainly because of the changing ABI and complex data structures the Nouveau graphics drivers may be rewritten in Rust in the future to make the development easier [80]. Nouveau has some technical debt, where problems were solved using a workaround [81] which also speaks for a refactoring. Nouveau has the burden of supporting all old Nvidia graphics cards, which makes this design change on newer GPUs even more complex to implement in the existing codebase.

For these reasons RedHat has started a new project, Nova [81], which aims to be a replacement for the Nouveau for the Nvidia Turing GPUs and newer (RTX 2000 series) [78]. Nova will be only for the GPUs with a GSP, and Nouveau will continue to be used with the older GPUs without a GSP. Nouveau nevertheless included an optional support for the GSP. Nova is being written in Rust in hope for better memory safety, better maintainability and a lower barrier for new developers to join the project [78].

The biggest obstacle with developing Nova - as well as Asahi Linux - is the missing infrastructure in the kernel for DRM and KMS. New infrastructure can’t be added before there is a user for it, but a user can’t be developed without the infrastructure as mentioned in 4.3.6. For KMS initial bindings were created by the author of `rvkms`, which is a rust-rewrite of the virtual KMS driver [82].

6 Identified challenges and solutions

Using Rust in Linux still has some major osbtacles. To solve them, some external projects have been created. The problems are explained and the proposed solutions are discussed in this chapter.

6.1 The safe pinned initialization problem (pinned-init)

Using mutexes in the kernel is inconvenient at the moment because it requires `unsafe` code. This section will go through the pinned initialization problem and discuss the solution most likely to be accepted for better ergonomics. This section is a summary mostly from article [19].

6.1.1 `Mutex<T>` and `Pin<P>`

Code listing 4 shows how the `Mutex` is implemented in Linux. It has three fields: owner, a wait spinlock and a wait list. If some process wants to acquire the `Mutex`, it first checks if the owner has been set (checking is atomic so no data race is possible). If the owner is `null`, the `Mutex` can simply be acquired by setting the owner. If the owner is already set, busy wait for a short time to see if the owner releases the lock shortly. If not, acquire the spinlock and add a new entry to the *wait_list* and go to sleep. Of course, the spinlock must be released beforehand, because it is not allowed to sleep in an atomic context, as explained in section 2.4.2. [19]

Code listing 4: `Mutex` implementation in Linux [19]

```
1 struct mutex {
2     atomic_long_t       owner;
3     raw_spinlock_t      wait_lock;
4     struct list_head    wait_list;
5 };
```

The problem lies within the implementation of the *wait_list* in Code listing 4, which is of type `struct list_head`. A crude example of this doubly linked list is shown in appendix E. This kind of doubly linked list is quite neat for low level code like for a kernel. The list itself does not care about memory management, the elements may live on the heap or even on the stack, the latter of which is faster. The list also has the benefit that inserting or removing individual elements can be done without branches, which can't lead to any pipeline stalls - thus the execution time is predictable. For these advantages these kind of lists are quite frequently used in the kernel. [19]

But this list is a problem in Rust because the list is self-referential, which is poorly compatible with Rust's ownership model. And since this kind of list is

quite clever for kernel code, there are numerous data structures like this one. It would require too much effort to rewrite everything in Rust. [19]

Turns out, the list described in appendix E has a memory error - trying to run it will trigger a segmentation fault. The troublesome function in this case is the function `add_element`, which creates a new element of type `list_head`, inserts it into the list after the given parameter `head` and returns the element. The new element is stored on the stack and the existing list is modified to point to it, but the stack is repurposed by functions `printf` and `debug_iter_list` after the function `add_element` has completed. The memory area for the new `list_head` element is overwritten, and trying to trace through the list in function `debug_iter_list` will result in a use-after-free memory error leading to a segmentation fault. [19]

The point of this code example is to show the problem in the ideology: it would be sound to return the element from the function if the memory address wouldn't change. But both C and Rust use return-by-value semantics, where the object is created on the stack and returning it copies the object to another place in memory.

Because this problem originates from a memory error, Rust has to ensure this invariant at compile time - this is why Rust is wanted in the kernel in the first place. The solution is to `Pin<P>` wrap the pointer `P`. For types of `!Unpin` it prevents from moving the pointee in the memory, which is achieved by not providing access to `&mut T`. Of course `unsafe` can still be used to do anything, but the invariant must be enforced by the user. [19]

Code listing 5: Accessing the mutex from outside of the struct [19]

```
struct DriverData {
    command_queue: Mutex<Queue<Command>>,
    // ...
}
impl DriverData {
    /// # Safety
    /// caller has to call `init`
    unsafe fn new_driver_data() -> Self {
        Self {
            // SAFETY: caller will call init()
            command_queue: unsafe { Mutex::new(Queue::new()) },
        }
    }

    fn init(self: Pin<&mut Self>) {
        Mutex::init(&mut self.mutex);
        //          ^^^^^^^^^^^^^^^^ error: cannot borrow as mutable
    }
}
```

Code listing 5 shows an artificial instance about this situation. For the Rust API to work with this, we would somehow need to be able to access the mutex from outside of our struct. Because `Pin` prohibits access to `&mut Self`, we can't borrow the mutex as mutable.

Accessing `Pin<&mut Mutex>` from a `Pin<&mut Self>` is a so-called pin-projection. Pin-projection is a tricky operation, since it needs to ensure that only a `Pin<&mut Mutex>` is ever produced - if not, the pin invariant would be violated. This operation is currently `unsafe`.

It is possible to make pin-projections in safe Rust, the crate `pin-project` contains a proc-macro to do so, but the crate requires the crate `syn` as a dependency. This is not an alternative for the kernel at present, because the crate `syn` contains over 55 KLOC whereas the Rust-bindings in the kernel are only about 22 KLOC. There is an alternative project `pin-project-lite` without proc macros, but it contains a very convoluted macro. From these options the kernel has been using the `unsafe` option up until this point.

If we wanted to initialize the `Mutex` using a typical approach in Rust, we would write `Box::new(Mutex::new())` and we would define the `Mutex` like in Code listing 6.

Code listing 6: Typical struct initialization in Rust [19]

```
impl<T> Mutex<T> {
    fn new(data: T) -> Self {
        Self {
            data,
            // ...
        }
    }
}
```

But for self-referencial structs like `list_head` this approach can't be used, because we don't know the memory address of the struct before we would need to populate the fields `prev` and `next`. Only after the initialization, the caller would move the struct to another location in memory - because of the return-by-value semantics - and the fields would now point to freed memory locations. This is why the kernel has been using a two-stage initialization shown in Code listing 7: first the struct is created with the `new`-function after which the struct is initialized using the `init`-method. This two-stage usage is bad design since it is really easy to forget to call the second function. What's even more terrible is that this API propagates to all structs containing a `struct list_head`. [83]

Code listing 7: The `unsafe` approach currently in use for initialization [19]

```
impl<T> Mutex<T> {
    /// # Safety
    /// the caller is required to call `init`
    unsafe fn new() -> Self;

    fn init(self: Pin<&mut Self>);
}
```

y86-dev - the author of [19] - wants to get rid of the `unsafe` from initialization. As a consequence, the initialization can also be done in a single function instead of the two-stage initialization with `new` and `init` functions. They are relatively confident that the projection issue can be solved soon. They have

been working on a solution which can currently be found in a pull request at [84].

A total of four possible solutions are rated by six attributes. The attributes by which the possible solutions are ranked, are

- Pinning guarantees
- Safety
- Fallible initialization
- Performance
- Ergonomics
- Ecosystem migration ergonomics

The pinning guarantee is paramount for any self-referencial struct. Safety is the primary reason, why Rust is wanted in the kernel development, as discussed in section 4.3.2. Fallible initialization is the only possibility in the kernel: the initialization may fail, but the kernel may not crash because of it, as explained in section 4.3.7. Performance is also important in an operating system - it's the primary reason Linux has been written in C. Ergonomics is the reason why this change is being made - to make the usage of the `Mutex` in Rust easier. Ergonomics in migration to a new ecosystem should also be good, so that rewriting existing parts using the `Mutex` is not that tedious. The possible solutions, which *y86-dev* discusses are

- the current `unsafe` solution
- `&uninit` T
- *placement-by-return*
- *in-place constructor*

where the *`unsafe` solution* is the currently used one and is shown as reference.

Table 4: Possible Pin-projection solutions ranked by 6 attributes [19]

Attribute \ solution	`unsafe`	`&uninit T`	*placement-by-return*	*in-place constructor*
Pinning guarantees	✓			✓
Safety		✓	✓	✓
Fallible initialization	✓			✓
Performance	✓	✓	✓	✓
Ergonomics		✓		✓
Ecosystem migration ergonomics				

Some of these solutions, like the `&uninit T` require new features - a new type - in the Rust language which the Rust-lang team is hesitant to make. The *placement-by-return* is an existing RFC, which would need to be modified drastically to support this use case. Without the modifications, it would only support the fields shown in Table 4, but with a lot of compiler magic and controversial or incomplete changes it could fill all the requirements. But with all the changes, it would become so much more complex than it already is, that *y86-dev* - the author of [19] - is not convinced it would be the way to go.

After these reasonings, as seen in Table 4, for now the *in-place constructor* is the strongest option for solution to the pinned initialization problem. This solution only requires 400 lines of code. [19]

The design of this paradigm is demonstrated in Code listing 8. The most notable part of this is that there is no `unsafe` needed. Implementing the `Mutex` with this is more complicated, since it requires using the FFI, as shown in Code listing 9.

Code listing 8: Example of an *in-place constructor* [19]

```
struct DriverData {
    command_queue: Mutex<Queue<Command>>,
    // we also have a big buffer that might fail to allocate
    buffer: Box<[u8; 1000_000]>,
}
impl DriverData {
    fn new() -> impl PinInitializer<Self, AllocError> {
        pin_init!(Self {
            command_queue: Mutex::new(Queue::new()),
            buffer: Box::try_new([0; 1000_000])?,
        })
    }
}
```

Code listing 9: Example of an *in-place constructor* with `Mutex` [19]

```
impl<T> Mutex<T>
    pub const fn new(
        t: T,
        name: &'static CStr,
        key: &'static LockClassKey,
    ) -> impl PinInitializer<Self, !> {
        fn init_mutex(
            name: &'static CStr,
            key: &'static LockClassKey,
        ) -> impl PinInitializer<Opaque<bindings::mutex>, !> {
            let init = move |place: *mut Opaque<bindings::mutex>| {
                unsafe {
                    bindings::__mutex_init(
                        Opaque::raw_get(place),
                        name.as_char_ptr(),
                        key.get()
                    )
                };
                Ok(())
            };
            // SAFETY: mutex has been initialized
            unsafe { PinInit::from_closure(init) }
        }
        pin_init!(Self {
            mutex: init_mutex(name, key),
            data: UnsafeCell::new(t),
            _pin: PhantomPinned,
        })
    }
}
```

There are currently still two major drawbacks. Firstly, `Box::new()` can't be directly used, but instead a new function must be implemented for it. Secondly,

it is not clear which fields are initialized in-place and which are initialized via another initializer.

Another thing to be considered about the *in-place constructor* is its usage in other things. It can be useful for initializing large data structures, because copying them would need additional space, which may be sparse, especially on embedded systems. On the other hand, most initializers in Rust are currently written in return-then-copy fashion. Changing the default style of initialization to in-place and rewriting existing implementations would be a huge migration. *y86-dev* is not sure, if they want to start that big of a transition. [19]

6.2 Compile-time detection of atomic context violations (klint)

The program `klint` is a static analysis tool for *Rust for Linux*, which is able to detect programming errors related to atomic contexts in the Rust kernel code, in compile time. While `klint` is not yet production ready, it is already able to find bugs. Most of this section is referenced from [20]. `klint` can be installed and used after instructions in appendix B.1.

As discussed in section 2.3, the kernel code may be run in either process context or atomic / interrupt context. Instead of process context, where the application is allowed to sleep, kernel code running in atomic or interrupt context must never sleep, as discussed in section 2.4.2 or as Billimoria states in [3]. Program code execution moves from process context into atomic context mostly in interrupts or while acquiring a spinlock. Another place where execution is in atomic context, is when the kernel is inside an RCU critical section. Please see section 2.4. [20]

Sleeping inside an atomic context is always bad - if a thread acquires a spinlock and then goes to sleep, after which another thread acquires a spinlock on the same resource, the system is very likely to be locked up. These mistakes are easy to make and notoriously hard to debug, especially if the sleeping calls are deeply nested [20]. But the real problem with sleeping in an atomic context comes from the implementation of the RCU critical section in Linux, as described in 2.4.2 - it may cause a use-after-free memory error.

The kernel has a mechanism for debugging this in the C code, functions may be annotated with a call to the function `might_sleep`. By enabling DEBUG_ATOMIC_SLEEP configuration, the kernel will count the preempt count: it will increment on entering an atomic section and decrement when exiting one. The kernel is still inside an atomic context, if the counter is non-zero. Calling the function `might_sleep` in this case will produce a warning to help debugging. [20]

6.2.1 RCU abstractions in Rust

In Rust it is typical to use RAII and scopes for synchronization primitives, unlike in C, where it is customary to use functions like `lock` and `unlock`. As an example, the RCU read-side critical section might be implemented in Rust like in Code listing 10.

Code listing 10: Model of an RCU read guard in Rust using RAII [20]

```
struct RcuReadGuard {
    _not_send: PhantomData<*mut ()>,
}

pub fn rcu_read_lock() -> RcuReadGuard {
    rcu_read_lock();
    RcuReadGuard { _not_send: PhantomData }
}

impl Drop for RcuReadGuard {
    fn drop(&mut self) {
        rcu_read_unlock();
    }
}

// Usage
{
    let guard = rcu_read_lock();

    /* Code inside RCU read-side critical section here */

    // `guard` is dropped automatically when it goes out of scope,
    // or can be dropped manually by `drop(guard)`.
}
```

The function `rcu_read_lock` creates a struct of type `RcuReadGuard` which is bound to a binding called *guard*, which will start the RCU read-side critical section. The lock will be opened as soon as the `RcuReadGuard` struct is released or if `drop(guard)` is called. The lifetimes of Rust are able to model RCU rather good, as demonstrated in Code listing 11, if neglecting the memory safety issue explained in section 2.4.2.

Code listing 11 shows that the lifetime of `'a` in the function `read` is tied both to `self` and the `RcuReadGuard`. The `RcuReadGuard` must outlive the returned reference, because by dropping it, the references returned by `read`-function references are not anymore readable. But this abstraction is not sound for the discussed sleep-in-atomic-context issue. [20]

Example Code 12 shows a situation, where two threads on different CPU cores use the same resource locked using an RCU. By sleeping in the atomic section of function `foo`, the function `synchronize_rcu` called inside function `write` - line 4 of function `bar` - prematurely returns while executing the function `bar`. By now dropping the resource behind pointer `old` in function `bar`, the pointer `p` in function `foo` gets invalidated. But by Rust's lifetimes, the binding `p` is still fully valid - using it would cause a use-after-free memory error.

Code listing 11: Example usage of an RCU read guard in Rust using RAII [20]

```rust
struct RcuProtectedBox<T> {
  write_mutex: Mutex<()>,
  ptr: UnsafeCell<*const T>,
}

impl<T> RcuProtectedBox<T> {
  fn read<'a>(&'a self, guard: &'a RcuReadGuard) -> &'a T {
  // SAFETY: We can deref because `guard` ensures
  // we are protected by RCU read lock
    let ptr = unsafe { rcu_dereference!(*self.ptr.get()) };
    // SAFETY: The lifetime is the shorter of `self` and `guard`,
    // so it can only be used until RCU read unlock.
    unsafe { &*ptr }
  }

  fn write(&self, p: Box<T>) -> Box<T> {
    let g = self.write_mutex.lock();
    let old_ptr;
    // SAFETY: We can deref and assign because we are the only writer.
    unsafe {
      old_ptr = rcu_dereference!(*self.ptr.get());
      rcu_assign_pointer!(*self.ptr.get(), Box::into_raw(p));
    }
    drop(g);
    synchronize_rcu();
    // SAFETY: We now have exclusive ownership of this pointer as
    // `synchronize_rcu` ensures that all reader that can read this pointer
    // has ended.
    unsafe { Box::from_raw(old_ptr) }
  }
}
```

Code 12: Example of safe Rust code, which allows a use-after-free [20]

```rust
fn foo(b: &RcuProtectedBox<Foo>) {
    let guard = rcu_read_lock();
    let p = b.read(&guard);



    sleep();

// Rust allows us to use `p` here
// but it is already freed!
}
```

```rust
fn bar(b: &RcuProtectedBox<Foo>) {


    let old = b.write(
        Box::new(Foo { ... })
    );
// `synchronize_rcu()` returns
    drop(old);


}
```

Multiple solutions were discussed. Firstly, all code which makes use of RCUs could be made `unsafe`, but this would be poor from the point of usability. It also wouldn't solve the issue when the C-code executes a Rust callback.

Another solution would be to force preemtion count and atomic context checking to be enabled, but it would be performance-wise a weak solution. All kernel code with spinlocks and RCUs would be affected by this.

None of the discussed approaches would provide the desired outcome so none of them was taken. It is not possible with the kernels Rust API design to prevent sleep-in-atomic-context from happening. It's possible to write a Rust-driver with only safe Rust and get a use-after-free condition, if the kernel is compiled without preemption count tracking [20].

6.2.2 Custom Compile-time checking with `klint`

The program `klint` checks for atomic context violations by tracking the preemption count at compile time. Each function is given two properties: the adjustment to the preemption count after this function and the expected range of preemption counts allowed when calling this function [20]. The properties for different functions are listed in Table 5.

Table 5: Adjustment and expectation for locking related functions [20]

Function name	Adjustment	Expectation
`spin_lock`	1	0 .. (any value)
`spin_unlock`	−1	1 .. (>= 1)
`mutex_lock`	0	0
`mutex_unlock`	0	0
`rcu_read_lock`	1	0 ..
`rcu_read_unlock`	−1	1 ..
`synchronize_rcu`	0	0

Sleepable functions like `synchronize_rcu` and `mutex_lock` will not change the preemption count because the *adjustment* value is 0, but they expect the preemption count to be precisely 0 because the *expectation* value is 0, as seen in Table 5. `spin_lock`, on the other hand, can be called from any context (expectation value is marked to be any natural number including zero) but it will adjust the preemption value after returning [20].

`klint` provides two attributes, the first one being `#[klint::preempt_count]` and the second one being `#[klint::drop_preempt_count]`. These can be used to annotate structs and enums for desired behaviour, when they are created or dropped, respectively. Some examples in Code listing 13 show how a `RcuReadGuard`-struct, `rcu_read_lock`-function or a sleep function could be annotated.

Code listing 13: Example annotations for `klint` [20]

```
#[klint::drop_preempt_count(adjust = -1, expect = 1.., unchecked)]
struct RcuReadGuard { /* ... */ }

#[klint::preempt_count(adjust = 1, expect = 0.., unchecked)]
pub fn rcu_read_lock() -> RcuReadGuard { /* ... */ }

#[klint::preempt_count(adjust = 0, expect = 0, unchecked)]
pub fn coarse_sleep(duration: Duration) { /* ... */ }
```

`klint` can be run just like rustc [25], but will also check the preemption counts. If a violation is found, for example if the function `coarse_sleep` is called with a spinlock or an RCU read lock held, an error shown in Text listing 8 is produced.

Text listing 8: Example error message from `klint` [20]

```
error: this call expects the preemption count to be 0
  --> samples/rust/rust_sync.rs:76:17
   |
76 |  kernel::delay::coarse_sleep(core::time::Duration::from_secs(1));
   |  ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
   |
   = note: but the possible preemption count at this point is 1
```

Some more difficulties which `klint` has to face are generic functions, recursion and indirect function calls from trait objects or function pointers.

For recursive functions `klint` will assume a default property and ask the user for an explicit annotation by giving an error. Generic functions are difficult since each implementation after monomorphization may be different. In worst case `klint` has to check all versions separately, but it tries to optimize and infer the properties on the generic function first. [20]

Code listing 14: Annotating trait methods for `ArcWake` for module `kasync` with `klint` annotations [20]

```
1  /// A waker that is wrapped in [`Arc`] for its reference counting.
2  ///
3  /// Types that implement this trait can get a [`Waker`] by calling
   [`ref_waker`].
4  pub trait ArcWake: Send + Sync {
5      /// Wakes a task up.
6      #[klint::preempt_count(expect = 0..)]
7      fn wake_by_ref(self: ArcBorrow<'_, Self>);
8
9      /// Wakes a task up and consumes a reference.
10     #[klint::preempt_count(expect = 0..)]
11     // Functions callable from `wake_up` must not sleep
12     fn wake(self: Arc<Self>) {
13         self.as_arc_borrow().wake_by_ref();
14     }
15 }
```

Function pointers can't be annotated like trait objects, like in Code listing 14. `klint` assumes the both are sleepable and makes no adjustments to preemption counts. The trait methods annotations will be used on virtual function calls, and they will be checked against their implementations.

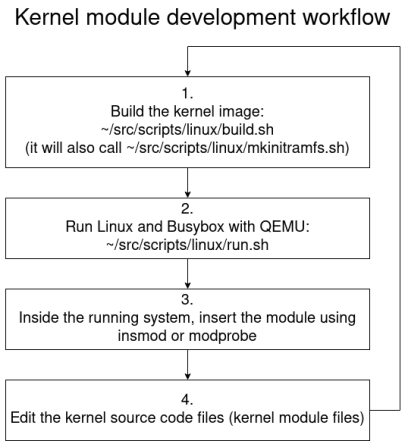
7 Case studies on Rust kernel driver implementation

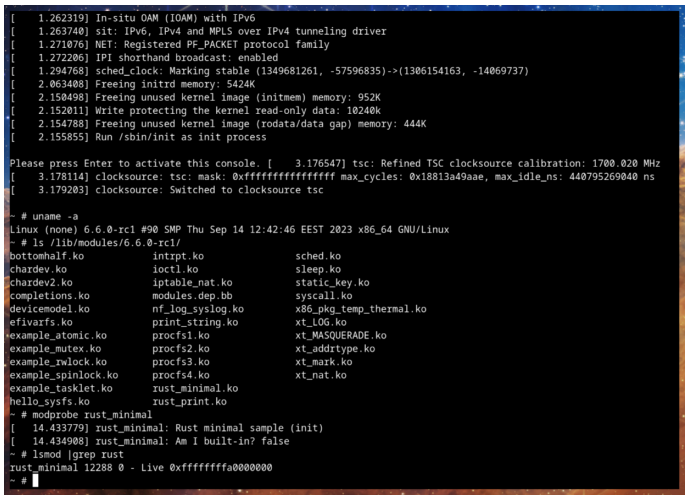
Some practical projects were planned for this thesis, which are discussed in this chapter: what was thougt about, why it didn't work out and what was done instead. Also the development environment for both C and Rust kernel module programming is discussed.

7.1 Development environment

A development environment, where the Linux kernel with Rust support could be compiled, was set up. The development environment is documented in appendix A.

The kernel module development workflow has been described in Figure 20. The contents of of the first script in Figure 20a, `build.sh`, is shown in Code listing 17 and the contents of the second script, `run.sh`, is shown in Code listing 18. Running the kernel with Busybox can be seen in Figure 20b.





(a) : Development workflow (b) : Linux kernel running with Busybox in QEMU

Figure 20: Kernel module development workflow

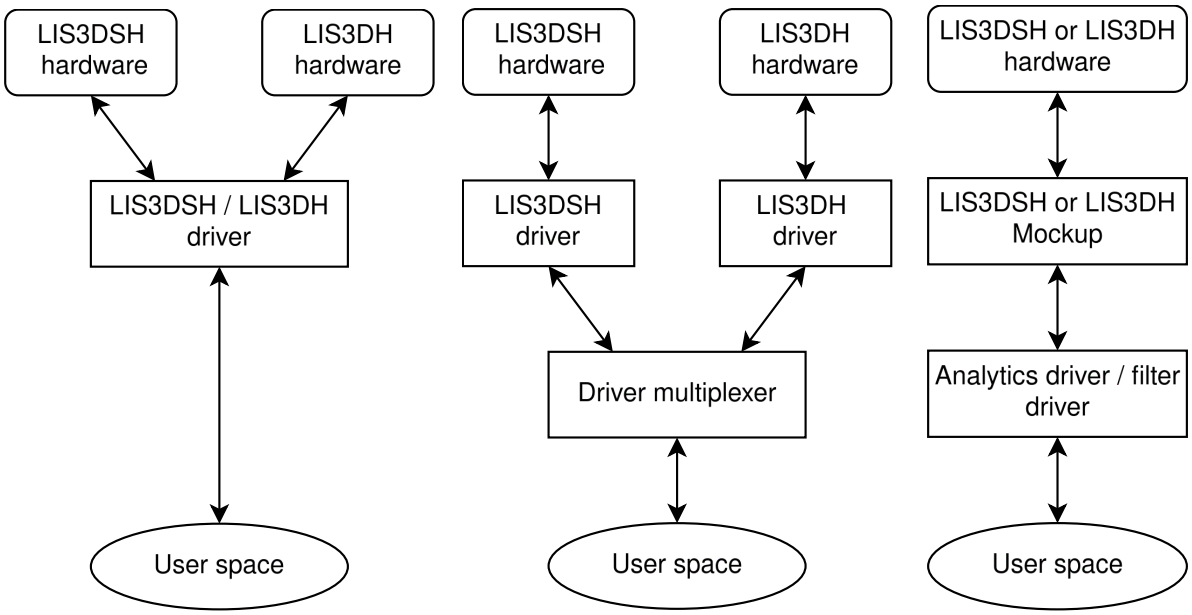
This development environment was indispensable for trying to write a kernel module in Rust as well as for building some of the existing projects to compare the sizes of the built kernel modules as described in section 7.3.

7.2 Thesis project idea: device driver for MEMS accelerometer

There were long considerations about the practical work for this thesis. Wapice Ltd has it's own IoT platform, which includes an embedded device running Linux. The hardware for the device is custom made, including a MEMS digital output motion sensor. Depending on the version of the device, the sensor is either LIS3DSH or LIS3DH.

The embedded device has two versions, 1 and 2. Version 1 contains the older sensor LIS3DSH which is now end of life. The device version 2 contains either LIS3DSH or the LIS3DH.

The two sensors are similar, with small differences in configuration. ST Microelectronics provides sample kernel drivers for both of these sensors. However, the embedded device now runs a custom-made kernel module which checks the sensor on board and configures the sensor accordingly. This makes the kernel driver unnecessary complex as shown in Figure 21a - the driver could be split into smaller drivers.



(a) : Current driver (b) : Future driver (c) : Mockup driver

Figure 21: Plans for driver development

A more elegant solution would be to have different hardware drivers for the different accelerometers as seen in Figure 21b. A third driver, *Driver multiplexer*, would then be used to communicate with these hardware device drivers. The driver multiplexer would provide an API for the user space applications. Implementing this driver in Rust would be a good challenge to learn kernel development with relatively simple hardware, requiring mostly usage of the SPI-bus on which the accelerometers resides on and the `miscdev` interface.

A simpler project idea was to mockup the LIS3DSH and / or LIS3DH drivers, against which a driver multiplexer or an analytics / filter driver could be implemented as seen in Figure 21c. This would at least be a step in the direction of the larger driver. The mockup could be written both in C and Rust, such that these two implementations could be compared.

The whole idea turned out not to be feasible for this scope as discussed further in chapter 8, because of the missing `miscdev` interface essential for all drivers in question.

7.3 Rust and C kernel module size study

In this thesis, a size comparison between the Rust and C kernel modules was made for the ASIX AX88796B and the Android Binder drivers. These two drivers were the most interesting drivers to compare for size. Apart from these two, only the null block driver and the NVMe driver have a C implementation as well - the PuzzleFS and Asahi Linux don't have an implementation in C, which makes them unsuitable for a comparison. The null block driver is quite small so it probably isn't the most interesting target for comparison, and the NVMe-driver does not implement the whole NVMe specification which is implemented by the C version, i.e., the comparison would be unfair.

7.3.1 Adding branches

To be able to compare the sizes of the compiled object files, the source code must be downloaded and compiled from the third party repositories - only the ASIX AX88796B driver is in the mainline at the moment. Branches from forked projects can be added to the already cloned Linux-repository as shown in Text listing 9.

Text listing 9: Adding a branch from the forked Android Binder repository

```
git remote add darksonn https://github.com/Darksonn/linux.git
git fetch darksonn b4/rust-binder
git checkout -b binder-b4/rust-binder darksonn/b4/rust-binder
```

This will fetch the branch `b4/rust-binder` from the Android Binder repository and add it to the local repository as a branch with the name `binder-b4/rust-binder`.

7.3.2 Kernel configuration

The ASIX AX88796B Rust driver can be compiled by enabling the CONFIG_AX88796B_RUST_PHY symbol shown in Text listing 10.

Text listing 10: Kernel configuration for compiling the ASIX AX88796B Rust-driver

```
Symbol: AX88796B_RUST_PHY [=n]
Type  : bool
Defined at drivers/net/phy/Kconfig:115
  Prompt: Rust reference driver for Asix PHYs
  Depends on: NETDEVICES [=y] && PHYLIB [=n] && RUST_PHYLIB_ABSTRACTIONS [=n]
&& AX88796B_PHY [=n]
  Location:
    -> Device Drivers
      -> Network device support (NETDEVICES [=y])
(2)     -> PHY Device support and infrastructure (PHYLIB [=n])
          -> Asix PHYs (AX88796B_PHY [=n])
            -> Rust reference driver for Asix PHYs (AX88796B_RUST_PHY [=n])
```

7.3.3 Size comparison

The size comparison is made using `objdump`, which tells the sizes and addresses of the different sections of an object file. Different sections are, e.g., `.text` which contains the executable program code, `.rodata` for read-only data and `.data` for variables. Fields containing the keyword `debug` are used for debugging information such as debug symbol names. Sometimes multiple fields are used for one category, like `.text`, `.init.text` and `.exit.text` as used by the `ax88796b.ko`, for instance.

For these multiple fields spanning a single category, a Python-script was written listed in appendix F. The idea is to run `objdump -h` and pipe the output to the Python script, which then summarizes the different categories and prints the output as a JSON-formatted dictionary. Based on those printouts, Table 9 and Table 10 were populated. The script must be manually adapted for other object files by including possible custom field names.

8 Results

This thesis has mainly results from literature review but some kernel module size comparison has also been made, results of which are discussed in this chapter.

8.1 State of Rust kernel driver support

As discussed in section 4.3.4, the development of *Rust for Linux* started in the branch `rust`, but it has been frozen since 2023-07-30, as seen in Text listing 11. The development now happens in either some other branch in the repository [16] or in the mainline Linux repository [51]. In the *Rust for Linux* repository, the branch `rust-next` is marked as default.

As for now, the necessary abstractions for `chardev` and `miscdev` are not yet implemented in the `rust-next` branch, although there already exists an implementation in the `rust` branch [16], [85]. The example shown in [86] works on the `rust` branch, but does not work on the `rust-next` branch. Copying the required `rust/kernel/miscdev.rs` file is also not trivial since it seems to depend on numerous synchronization primitives, including but not limited to mutexes and spinlocks.

Text listing 11: The amount of commits in branches `rust` and `rust-next` (updated on 2024-05-08)

```
# Most recent commit in branch `rust`
$ git switch rust
$ git log -n1
commit 18b7491480025420896e0c8b73c98475c3806c6f (HEAD -> rust, upstream/rust,
upstream/HEAD, origin/rust)
Merge: c8d1ae2cbe24 492857088c19
Author: Miguel Ojeda <ojeda@users.noreply.github.com>
Date:   Sun Jul 30 00:26:49 2023 +0200

# Most recent commit in branch `rust-next`
$ git log -n1 rust-next --
commit 97ab3e8eec0ce79d9e265e6c9e4c480492180409 (HEAD -> rust-next, upstream/
rust-next)
Author: Danilo Krummrich <dakr@redhat.com>
Date:   Wed May 1 15:47:43 2024 +0200

# Common ancestor
$ git merge-base 18b7491  97ab3e8
457391b0380335d5e9a5babdec90ac53928b23b4

# How many commits are there in branch `rust-next` from the common ancestor
$ git rev-list 457391b..97ab3e8 |wc -l
93290

# How many commits are there in branch `rust` from the common ancestor
$ git rev-list 457391b..18b7491 |wc -l
1551
```

Inspecting the git branches is described in Text listing 11. In the first two commands, the most recent commits from branches `rust` and `rust-next` are shown. The third command is for searching the most recent common ancestor, i.e., the commit from which these two branches diverged. The fourth and fifth command are used to calculate, how many commits the branches have gained respectively: `rust-next` is 93290 commits ahead, but `rust` is also 1551 commits ahead from the common ancestor. The commits, in which the branch `rust` is ahead, are not merged into `rust-next` and thus contain possible new unmerged features, such as the `miscdev` infrastructure. These numbers can also be seen at the public Linux repository [51] - but they will of course change after new commits arrive.

Text listing 12: Modifications of the `./rust/kernel/miscdev.rs` file in Linux repository

```
# Find out which commits changed the miscdev.rs
git log --follow --pretty=oneline -- ./rust/kernel/miscdev.rs
<redacted>
586e37da15c695ed4d156edacf51eec9d18f0a42

# Find out the kernel version on the earliest modification of the miscdev.rs
git show 586e37da15c695ed4d156edacf51eec9d18f0a42:Makefile
# SPDX-License-Identifier: GPL-2.0
VERSION = 5
PATCHLEVEL = 9
SUBLEVEL = 0
EXTRAVERSION =
NAME = Kleptomaniac Octopus
<redacted>
```

Text listing 12 shows also how to check in which commits the `miscdev.rs` has been changed. The list is long and unappealing wherefore all but the last commit hash have been redacted. The purpose is to show in which commit the miscdev support was firstly introduced, and the Linux version seems to be 5.9.0 at that moment.

8.2 Obstacles with the original project idea

8.2.1 Insufficient kernel version

Implementation of the accelerometer drivers in Rust would have been a perfect challenge for this thesis, but Rust for Linux requires at least kernel version 6.1, but 6.3 came with many improvements so it is preferred.

Unfortunately, the embedded device, for which the drivers were to be made, is still running Linux 5.4. Updating it is also not quite trivial, since it is based on Yocto version *Dunfell* (please see Table 6). The SoC manufacturer doesn't yet support a newer release, so the updating would have to be done manually by creating a custom Yocto recipe. The next version the embedded devices firmware is updated to will most likely be either *Kirkstone* or *Scarthgap*. From these options, *Kirkstone* does not provide a recent enough kernel version and

Scarthgap will be released at the earliest in April 2024 [21] which will be too late for this thesis.

Table 6: Yocto releases and the supported kernel versions [21]

Name	Yocto version	↑ kernel version	Source
Dunfell	3.1	5.4	[87]
Kirkstone	4.0	5.15	[88]
Mickledore	4.2	6.1	[89]
Nanbield	4.3	6.5	[90]
Scarthgap	5.0	6.6	[91]

With this reasoning the next idea was to implement a driver, which mocks the hardware driver for the LIS3DH. After this mocking driver is implemented, some other higher level driver could be written which uses this mocking driver as its source for the accelerometer sensor data. The good part about this would be, everything could be done in a virtualized environment and run in a virtual machine (QEMU).

No real hardware would have to be dealt with, which surely would be interesting but time consuming. Also, no time would have to be spent upgrading the Linux for the embedded device. But as one might guess from section 8.2.2, this plan was not executed.

8.2.2 Missing kernel infrastructure

As noted in section 4.3.4 and Table 1, multiple development branches exist in the *Rust for Linux* repository. The first branch, `rust`, was for testing and demonstration purposes only and has since been frozen. New, production grade, development happens in branch `rust-next`. The branch `rust` has still infrastructure which has not been moved into `rust-next`.

One of these missing feature is the `miscdev` framework, which would have been useful for this thesis. As described in section 3.5, the miscellaneous framework is useful, e.g., for SoCs and the accelerometer kernel driver would have been a good example about it.

Table 7: Newest commits of `rust` and `rust-next` and their common ancestor compiled from Text listing 11, updated on 2024-05-08

Description	Commit	Commit date	Ahead
`rust`	`18b7491`	2023-07-30	1551
`rust-next`	`97ab3e8`	2024-05-01	93290
Common ancestor	`457391b`	2023-04-23	0

Because the infrastructure is not yet merged to the branch `rust-next` and is only accessible in branch `rust`, the missing functionality would first needed to be implemented into `rust-next`. It could be possible to cherry-pick the neccessary commits from the `rust` branch but as shown in Table 7, 1552 commits are a lot to start searching from. Simply copying the `miscdev.rs` was also

not possible since it revealed having numerous dependencies, going into synchronization primitives such as mutexes and spinlocks. As study had already been made about synchronization problems in the kernel, regarding, e.g., the Safe Pinned Initialization Problem (section 6.1), it no longer seemed feasible to try manually copying content from the `rust` branch into `rust-next`. The best outcome would probably have been, that the `rust-next` branch would then contain only code from Linux 6.3.0.

Of course, the `miscdev` infrastructure seemed to be fine in the `rust` branch so the project could just be directly done on top of it, but since it was on a frozen and desolated branch, nobody would know whether the code would be ever useful and compatible with a future version of Linux.

8.3 Topicality of existing projects

The existing projects introduced in chapter 5 are gathered in Table 8. Information about the kernel version which they currently run on and a source citation to the source tree repository are also listed.

Table 8: Summary of the different projects described in chapter 5 and the kernel versions they currently are based on (updated on 2024-05-08)

Link to section	Driver	Kernel version	Branch	Repository
5.1	AX88796B	6.9.0-rc7	master	[51]
5.2	NVMe	6.9.0-rc3	rnvme	[68]
5.3	Null Block	6.9.0-rc3	rnull	[68]
5.4	PuzzleFS	6.7.0-rc5	master	[92]
5.5	Binder	6.6.0	rust-binder-rfc	[73]
5.6	Asahi	6.6	asahi	[79]
5.7	Nova	6.9-rc1	nova	[93]

Table 8 shows that most of the drivers work on updated Linux-versions. At the time of writing this, the newest version of Linux is 6.9.0-rc7.

8.4 Object file size comparison

As mentioned in section 7.3, a size comparison between the compiled Rust and C modules of ASIX AX88796B and Android Binder drivers was made. The results are listed in this section.

8.4.1 ASIX AX88796B

Table 9 presents a comparison between the two kernel modules object files. To gather the information of the different sections sizes, `objdump -h` was used. The C version contains multiple text and data sections named `.text`, `.init.text` and `.exit.text`, all of which are summed together. The same applies to `.data`, which is obtained by summing `.init.data` and `.exit.data` into it. The C version doesn't contain the `.bss` section and all sections containing the word *debug* are accumulated in the `.debug` row.

Table 9: Size comparison of AX88796B C and Rust drivers

	ax88796b.ko	**ax88796b_rust.ko**	**Rust / C**
Filesize	185.4K	426.5K	2.30
.text	265	668	2.52
.data	1656	1608	0.97
.rodata	78	120	1.54
.debug	90032	263322	2.92
.bss		16	NaN

8.4.2 Android Binder

Table 10 presents a comparison of the Binder C and Rust implementations object files. The Binder Rust driver has also less fields than the corresponding C driver, just like the ASIX AX88796B driver. In addition to the usual `.text`, `.rodata`, `.data` and `.bss`, there also exist fields called `.rodata.cst8`, `.rodata.cst16`, `.rodata.cst4` and `.rodata.cst32`, which are summed together into the `.rodata` row. The sum of the fields `.rodata.cst*` is 160 bytes. The C driver has also 226 fields with the name `__patchable_function_entries`, each of which has a size of 8 bytes. They are omitted in this measurement.

Table 10: Size comparison of Binder C and Rust drivers

	binder	**rust_binder**	**Rust / C**
Filesize	1.3M	4.1M	3.15
.text	70918	108021	1.52
.data	10800	8	0.00
.rodata	13852	9080	0.66
.debug	576494	2750208	4.77
.bss	20264	170	0.01

9 Discussion

Most of this thesis is a literature survey and much of the time was spent on both understanding the playground and studying the existing projects in chapter 5. Also the most notable big obstacles listed in chapter 6 took a lot of time to understand profoundly enough to be able to document them.

A size comparison of the kernel modules was conducted specifically for this thesis, which is discussed in section 9.1, and multiple performance comparisons were referenced and discussed in section 9.2.

9.1 Size comparison

In the drivers ASIX AX88796B and Android Binder the Rust and C versions were compared by size, and the results are shown in Table 9 and Table 10. These two drivers were chosen, because they had both C and Rust implementations. The ASIX AX88796B is unique because it is the first driver written in Rust in the mainline kernel, and the Android Binder driver is quite promising by usefulness - it is most likely to be in the mainline in the near future. Both of these drivers are quite comparable, because the features implemented by the Rust and C versions are more or less equal. For instance, the NVMe driver and the null block driver are not that good for this kind of comparison, because although they both have implementations in Rust and C, their Rust counterparts are trailing behind in features compared to their original C versions. For instance, the implemented features of the null block driver are listed in Table 2.

For some reason, no existing study was found on the internet regarding kernel module sizes, especially the sizes of different blocks within the kernel object file. This indicates that the comparison in this thesis is genuinely new information. Rust has been blamed for big binary sizes, which makes this comparison even more interesting. The kernel should run on as many devices as possible - including embedded systems with little memory.

Both Table 9 and Table 10 show, that the Rust version of the object file itself is a good two to three times larger, when only comparing the file size. This of course does not tell the whole truth, because there are multiple different segments within the object file. In addition, the file size is quantized to the used block size of the file system - for instance 4kB.

A deeper inspection reveals, that the biggest difference for both drivers comes from the debug symbols: both the AX88796B and the Binder Rust versions have 2.9 and 4.8 times bigger `.debug` segments, respectively. The program code in `.text` is somewhat larger in both Rust-versions, which directly indicates for a bigger executable code size. For the AX88796B driver, the segment is 2.5 times larger and for the Binder it is 1.5 times larger. The variables in `.data` are almost the same for the AX88796B driver in both Rust and C versions. But what's interesting is that the Rust version of the Binder has almost nothing in the `.data` segment: the segment size of the C version is 10KB

whereas the Rust version is only 8B - the variables must be stored somewhere else. The size of constants in `.rodata` is about 1.5 times for the Rust version in the AX88796B but only 0.66 for the Binder. No conclusions can be made from `.rodata`.

The ASIX AX88796B comparison isn't that informative. Both versions of the drivers, C and Rust, seem to be mostly wrappers to the underlying physical library, both files are only about 130 LOC [51]. To better compare between C and Rust kernel modules, bigger drivers would be needed. The Binder comparison is good - the driver has a reasonable size and it implements all features which the original C driver also supports, as disclosed in section 5.5. Some debugging facilities are still missing, but the Rust driver already contains more debugging code when compared to the AX88796B in Table 9.

Of course, this was only tested for two kernel modules and a much bigger test suite would be useful - that is many more kernel modules in this context. This test may also be misleading, because it does not take the infrastructure size into account - how much executable code is needed for the Rust ecosystem to work in the kernel? At the end of the day, the important question to ask is, does the whole kernel with all the necessary drivers fit into the memory of the device or not.

9.2 Performance comparison

Performance comparison was not carried out for this thesis specifically, but rather referenced and compared from NVMe driver introduced at 5.2, null block driver displayed at 5.3 and Android Binder driver unveiled at 5.5.

9.2.1 NVMe driver

In section 5.2 four different performance comparisons are listed, newest of which is performed with Linux 6.7. Table 8 shows that Linux 6.8 is the newest kernel version the NVMe driver currently runs on. The results are complicated to analyze thoroughly, because the source [65] hardly describes the performance graphs, which leaves the results open for interpretation.

Altogether, the performance results are good. Figure 13a and Figure 13b show, that for 4KiB block size both of the drivers perform similarly. The target is bandwidth limited for this configuration. For smaller 512B block size the target is not bandwidth limited but rather compute limited. [65] states that the Rust driver has a higher overhead and thus performs up to 6% worse compared to the C driver.

In the newer performance measurements, like in sections 5.2.1 and 5.2.2 a performance boost can be observed in the Rust driver compared to the C driver when using larger queue size. This was not the case back in measurement 5.2.3.

Figure 11a shows that there is hardly any difference between the Rust version with link time optimization turned on versus not using it. In some cases it even seems to be slower, but the difference is negligible.

The three newer measurements 5.2.1, 5.2.2 and 5.2.3 are comparable with each other, because the same hardware was used. 5.2.4 uses different hardware and thus is not directly comparable. As a simple longitudinal study, comparing the results in the abovementioned measurements for Linux 6.6 in Figure 11 and Linux 6.7 in Figure 10 reveals, that the performance of the Rust driver increased from about 1.09 to about 1.24 million IO/s when using queue dept of 16. However, also the performance from the C driver increased, which indicates that there may be some kind of randomness. The difference may also be explained by the fact that the performance charts in Figure 10 and Figure 11 state *random read throughput*, whereas Figure 12 states only *read throughput*. On the other hand, in Figure 12 the performance metrics are more uniform between queue depths 4, 8 and 16 than in the chronologically later tests.

9.2.2 Null block driver

The null block driver, `rnull`, introduced in section 5.3, is a small simple driver which has had a significant number of memory safety issues (27% of all commits are bugfix commits, out of which 41% are memory safety bug fixes) and thus it is a good candidate for being rewritten in Rust. The current version does not cover everything implemented in the older `null_blk` driver written in C, as seen in Table 2.

The performance has been measured with Linux versions 6.8, 6.7 and 6.6 and is shown in figures Figure 14, Figure 15 and Figure 16 respectively. As with the NVMe driver, the results are complex to analyze since not much is explained by the source [17] and one needs to understand the topic thoroughly to be able to draw conclusions.

All in all, the performance results seem good and united. For lower block sizes 4KiB and 32KiB the `rnull` driver is somewhat less performing compared to the older `null_blk`, which is expected. The difference is not too big, around 10%, but maximally up to 12% for the Linux 6.8 in Figure 14. For higher block sizes (256KiB, 1MiB and 16MiB) the driver is probably not anymore compute limited but rather bandwidth limited, which can be seen in almost equal performance for the drivers. The difference seems also to diminish with higher queue size. The usage of different amount of threads (cores) seems also not to have a great impact on the relative difference.

The `rnull` seemed to perform somewhat better in the past with Linux 6.6 as seen in Figure 16, at least on smaller block sizes. The changelog at 5.3.2 shows that some changes in the memory backing had been done, which may have caused these results. In conclusion, the null block driver has also shown that Rust will perform good enough for being used in the kernel alongside C.

9.2.3 Android Binder

The Android Binder driver introduced in section 5.5 is almost completely rewritten in Rust, mostly because its originally intended application has expanded and the original implementation in C has gained technical debt. The Rust implementation is sound and passes all tests, which validate the correctness of the Binder in the AOSP. It does not yet contain all debugging facilities, but they are added before the driver is submitted upstream to the mainline kernel.

Two performance tests were referenced in section 5.5 to compare the two implementations. The first, *binderThroughputTest* shown in Figure 17, tests for throughput on both implementations using different number of client-server pairs. The left graph is with empty payload sizes whereas the right one is for payload sizes of 4k. Negative numbers are better for Rust, and Rust seems to excel when only one client-server pair is present. The performance boost is significantly smaller with multiple client-server pairs - it is around 0% but the average is still negative, which is in favour of Rust.

binderRpcBenchmark, shown in Figure 18, is used to test the throughput using different block sizes and a few other metrics. The key takeaway is, that Rust seems to perform better for *pingTransaction* and *repeatBinder* tests as well as for the throughput-tests, except for the biggest block sizes. It is also mentioned in 5.5, that the use case for the biggest block sizes is sparse on the field and the developer team is confident in being able to optimize the results further.

Both drivers are almost 6 KLOC, the C version being a little bigger with 5.8 KLOC compared to the Rust version with 5.5 KLOC. Rust is a more expressive language, but on the other hand, Rust is less implicit and requires e.g. explicit type conversions from the programmer.

Rust may have a small performance boost in the Android Binder, but that’s not the biggest interest for using it in the kernel. Much more important advantages are, e.g., automatical lifetime tracking of objects. In the Binder, Rust also assisted with the powerful type system by helping in encoding ownership semantics for structs and pointers.

9.3 GPU drivers

As discussed in the section 5.6 about Asahi Linux and Apple’s AGX GPU firmware interface at 5.6.1, it was mentioned that the new GPUs use a separate processor, running an RTOS communicating with the actual GPU. This way all timing critical tasks will be offloaded to the RTOS, which is able to provide a sane, higher level interface to the operating system GPU driver. The same design principle can also be found on modern Nvidia GPUs, as mentioned in section 5.7 about the unofficial open source Nvidia GPU drivers Nouveau and Nova.

Section 5.6 about Asahi Linux highlights the benefits of Rust in this realm. As the firmware of the GSP may be updated over time, and both sides of the interface are governed by Apple - the same applies to the Nvidia GSPs - the interface may change at any time. Moreover, multiple different versions must be supported at the same time. In C, supporting different versions where, e.g., structs contain different amount of fields, requires the usage of `#define` preprocessor macros and multiple compilations - one for each version. Rust however, includes a powerful macro system, which makes it possible to support multiple different versions at the same time.

Also the distinctive lack of debugging after implementing a program with Rust after a prototype or a protocol is astonishing. The experience has shown, that developing something in C/C++ would require quite some time of debugging memory errors like memory leaks or use-after-frees, race conditions or others. But according to Asahi Lina they had almost no debugging apart from a few logic errors, as mentioned in section 5.6.

9.4 Missing kernel infrastructure

The consensus seems to be that the kernel is still missing many important interfaces, but the number of implemented interfaces is only increasing. As discussed in sections 4.3.4 and 8.1, the demo-branch `rust` still contains some code which hasn't been merged upstream into `rust-next`, the development branch. This includes the support for the `miscdevice` framework, which could have been used in this thesis in creating a character device driver for an accelerometer as discussed in 7.2.

The reason behind this particular interface for `miscdev` still missing is unknown. On one hand, the feature may be incomplete or poorly implemented. The `rust` branch was meant only for prototyping purposes, from which features may be cherry-picked into the production-grade `rust-next` branch, or they may be rewritten completely, as stated in 4.3.4. On the other hand, no infrastructure is accepted into the kernel without anyone using it, which means that the infrastructure for the `miscdev` driver is added as soon as someone uses it. Although the quality of this source can be questioned, these same toughts are speculated at [85]. It would have been an excellent opportunity to contribute to the kernel by implementing the missing interfaces, had there been enough time, experience on kernel development and knowledge about Rust.

In section 9.3 the drivers for Apple AGX GPU and newer Nvidia GPUs were discussed. The biggest problem for the new project Nova is, that Asahi Linux is still out-of-tree and no stable interface has been established. In practice this refers to the missing Rust bindings for the **KMS** (Kernel Mode Setting) and **DRM** (Direct Rendering Manager) interfaces.

Section 9.3 also mentions that there is a project, `rvkms`, which also requires the Rust interface for the KMS. These projects are expected to cooperate for reaching the common destination - to get video drivers written in Rust into the mainline kernel.

9.5 Benefits of Rust

In section 3.4 two typical kernel function call interfaces are described, functions `read` and `write`. The functions take a pointer to a buffer as a parameter, which is used to read the data from or write the data to, and the return value of the function denotes either the number of bytes read or written. A negative return value will indicate an error.

Rust makes the interpretations of the return values more straight forward. Firstly, the success may be encoded as a `Result<T, E>` where `T` would be the return value on success and `E` would indicate the error type once encountering a failure, as described in section 4.3.2. Secondly, the “wait, there is more to be read”, which is encoded in C as a positive integer, and the “please stop, there is nothing more to be read” can be encoded as an `Option<T>`. If there still was something to retrieve, the option would be of type `Some<T>` and if there wasn’t anything to be read anymore, `None`. Overall, this return value could be encoded as a type of `Result<Option<T>, E>`, where `T` would be the bytes read or written. This would have the benefit, that the compiler will enforce checking for error conditions or `None` values, omitting simple humane errors.

In C, the data is returned by giving a pointer to the function but this is not very idiomatic Rust; in user space a `Box<T>` could be used but there is no dynamic memory allocation in the kernel. Further, the memory segment for kernel space and user space are different, so the function is not even able to allocate the required memory on the kernel side for user space, it must be done in the user space instead.

9.6 Challenges with Rust

Including a new programming language into an existing project is never an easy task and with a project as large as the Linux kernel new problems will arise. Especially, because the kernel has been designed to be as efficient as possible and some shortcuts have been made in the very design. Chapter 6 explains two problems thoroughly, which are shortly recapped in the next sections, 9.6.1 and 9.6.2.

9.6.1 The safe pinned initialization problem (pinned-init)

The safe pinned initialization problem is explained deeper in section 6.1 but it can be described as the inconvenience, that using the `Mutex` currently requires `unsafe` code. The reason for this is, that the `Mutex` internally contains a data structure, which must be specially initialized: as it is initialized, it must be pinned into the memory and it must not move after initialization. But since both Rust and C use return-by-value semantics, the default behaviour is that the constructed object is moved in the memory right after the constructor returns the value.

Four possible solutions were compared for this problem, some of which are better than others, when ranked over six attributes as shown in Table 4. The solution `&uninit` T needs a new feature from the Rust language, which the Rust-lang team is hesitant to provide since it requires introducing a new type in the language. *Placement-by-return* seems all good at first glance but after deeper scrutiny it doesn't seem to really solve all the problems without some really complex modifications to an existing RFC, making it unattractive.

The most realistic solution to be taken into use is the *in-place constructor*, because it is the winning solution by most of the attributes shown in Table 4. But the ecosystem migration ergonomisc isn't the best even in the *in-place-constructor*, mostly because the implementation is not easy and requires the usage of FFI. The author of [19], *y86-dev*, is convinced that the pin-projection problem may be solved soon enough.

9.6.2 Compile-time detection of atomic context violations (klint)

The atomic context violation problem is explained in greater detail in section 6.2. The Linux kernel has some design choices, which prohibit sleeping in atomic contexts (explained in section 2.3). For example the RCU, *read-copy-update* (introduced in section 2.4), has a deliberate design choice, which improves its performance. The disadvantage is that sleeping inside an RCU read-side critical section may introduce a use-after-free memory error, as discussed in section 2.4.2.

In C, this problem has been solved by introducing a function `might_sleep` which will, if the symbol DEBUG_ATOMIC_SLEEP is enabled, warn about sleeping functions in atomic contexts. This kind of solution, run-time tracking of the atomic-context violations, was also discussed for the Rust API, but it was turned down by Torvalds [20].

In Rust the locking mechanisms are usually implemented using RAII and the Rust lifetimes can model the RCU pretty well, as discussed in section 6.2.1. But when the design decision of the Linux kernel is taken into account and the wickedness of sleeping within an atomic context, there are no safeguards in the kernel's Rust API abstractions that prevent the sleep-in-atomic-context from happening, as mentioned in [20]. That means, it is possible to write safe Rust code in the kernel which may introduce a use-after-free memory error.

This is why a custom linting tool, `klint`, was created. It checks for atomic context violations by tracking the preemption count at compile-time, as discussed in 6.2.2. It is not yet production ready, but it is already able to find bugs. It still has some challenges like generic functions, recursion and indirect function calls from trait objects or function pointers.

9.7 The future of Rust in the kernel

As discussed in 9.4, many interfaces are still missing, which makes Rust feasible in the kernel only for very specific purposes. As for now, good programming skills in both C and Rust and deep understanding about the Linux kernel are required to be able to create Rust-bindings.

The kernel has a huge codebase which will most likely never be rewritten entirely in Rust, as stated in section 4.3. Although C is a relatively simple language, kernel development in C has a high learning curve with its API and it is not uncommon to have to debug synchronization or memory errors. As Rust has a pedantic compiler, which is good in memory management, it can help tackle these issues making the development easier. This hopefully encourages new developers to join the kernel project. This is why the Rust support has the highest priority in the device drivers, because that's the most common starting point in the kernel programming for newcomers, as discussed in section 4.3.1.

The kernel already has multiple device drivers written in Rust, some of which were introduced in chapter 5. While writing this thesis, new projects emerged. For instance, the ASIX AX88796B PHY driver was merged into the mainline kernel. Most of the projects listed in chapter 5 were found before they were catalogued in the official *Rust for Linux* documentation at [28]. Not all Rust-related kernel projects could be included into this thesis, for instance, the projects `vkms` [82] and in-kernel codecs [94] were labeled as out-of-scope for this thesis, because they were found too late. The field is in rapid development.

9.8 Other observations: the FPU

At the time of writing this thesis, the author was writing a math program in the weekends about calculating the position of the Sun on the sky, when only the time and position on the Earth are known. The resulting program is not that complex but rather contains some math using many trigonometric functions such as `sin`, `cos`, `atan2` and also the square root function `sqrt`. A question emerged, if this user space program could be implement as kernel code, e.g., as a character misc device driver.

The first problem was the need to convert datetimes from the usual Gregorian calendar into Julian calendar, for which an external library `julian-rs` [95] was first used. Once the algorithm was found in [96], it turned out that converting between these two calendars is not that hard and implementing the algorithm in Rust was done in a few hours. The code *should* compile just fine as a kernel module, because it does not use any dependencies.

The next obstacles were the trigonometric functions. Although `core::f64`, which implements the 64-bit floating point primitive datatype `f64`, is available in the kernel Rust code, trigonometric functions are implemented in `std::f64` [97] which is not available in the kernel. This was not a problem per se, the musl's `libm` implementation [98] could be used instead - naturally by copying

the source code and linking it statically, since nothing could be included into the kernel with `cargo`.

After trying this, the next problem was that although `rustc` allowed the usage of `f64` in the kernel, the **FPU** (*Floating Point Unit*) was apparently not available. The reason is that *floating-point arithmetic is not allowed in kernel space!* [30] Surely it must be possible to use the FPU, since the kernel is directly in charge of the hardware, but it is a design decision of the Linux kernel developers that the FPU is not used inside the kernel [3]. Using the FPU involves saving and restoring its state - the registers - every time a context switch is done, i.e., every time the scheduler changes the running process. Saving the state takes some time and since the kernel is mostly responsible for running device drivers which usually don't need floating point values, the usage of the FPU inside the kernel has not been seen as worthwhile.

It *should* be possible to use the FPU inside the kernel between function calls `kernel_fpu_begin` and `kernel_fpu_end` but [3] demonstrates that it is not feasible even then - the kernel may run as intended, but it will emit a warning with the `WARN_ONCE` macro. It is worth realizing that most production environments have set the value of `/proc/sys/kernel/panic_on_warn` to `1`, which will cause the kernel to panic even on the first encountered warning. [3]

Not being able to use floating point calculations in the kernel is a complete barrier for this Sun-position-calculation program to be implemented as kernel level code, as a character misc device driver. For this program floating point values are needed for everything: converting the date from Gregorian calendar to Julian as well as the numerous coordinate transformations between the multiple coordinate systems used for the different celestial bodies. One step to investigate further would be the usage of *soft floats* but this is getting way too out of scope. There is no real reason to implement this kind of calculation in kernel space, other than sheer curiosity. It just would have been - and already has been - an interesting journey, which is not worth to be continued.

9.9 Hypotheses & their realization

Three study hypotheses were introduced in 1.2.1. Given the performance results in 9.2, the study hypothesis #1 seems to hold. All three drivers, the NVMe driver, null block driver and Android Binder seem to have a little worse performance with Rust when the driver is compute limited. The Rust has a little more overhead compared to C, because it does some run-time checks like bounds checking. This performance deterioration is acceptable considering the benefits Rust provides for kernel development.

Based on the sampling in section 9.1 the study hypothesis #3 seems to be fulfilled. The executable code block size in the object file seems to be notably larger - two to threefold - for the Rust driver compared to the C version of the same driver. The debug section is even larger, but it can be omitted as it most likely could be optimized away in the production version. However, the sampling is small, only two kernel modules, out of which one is merely a wrapper to backing infrastructure. Comparing more kernel modules would be needed to confirm this.

Hypothesis #2 can’t be verified unambiguously, as it’s made even more difficult by its subjectiveness. Section 5.6 endorses this hypothesis, as the author of Asahi Linux mentioned they had almost no debugging after implementing the GPU driver in Rust. Traditionally, a lot of time had to be used to debug memory and synchronization errors. Also, the higher abstraction layer Rust provides speaks for faster development time. But because neither a driver was implemented in this thesis nor has any data been found regarding other drivers development times, this hypothesis is left unconfirmed.

10 Conclusions

In the systems programming languages front there hasn't been that much development, C has been around since the 1970s and is still widely in use in projects such as the Linux kernel. As Rust is a *new kid in town* in this realm, it was only a matter of time when it would get interesting enough to be included into Linux. Rust has been getting a lot of attention in the industry lately, as big technology companies, such as the ISRG (Internet Security Research Group), are supporting the project *Rust for Linux* with financial help from Google. Countless other big tech companies are also investing by educating developers to use Rust.

In this thesis a literature review of the current status of the project *Rust for Linux* was made. Rust provides major advantages with its automatic compile-time memory management, which helps fighting memory errors and synchronization problems. It is also desired for the higher abstraction layer in programming to entice new developers into developing Linux. The disadvantages mainly regard design choices in Linux, which don't fit well into the design philosophy of Rust and therefore some workarounds need to be made.

Two major obstacles are still hindering *Rust for Linux*. The first one being the pinned initialization problem, where an object must be pinned into memory at the time of it's construction. This problem projects to the required `unsafe` code for using a mutex. The most realistic solution for this problem is the *in-place constructor*, but the migration would not be easy. The problem with *pinned-init* will most likely be solved soon.

The other problem is that it's not possible with the kernel Rust API to check, if there are any sleeping calls inside atomic contexts. Because of the design in Linux, sleeping in an atomic context may not only cause deadlocks but is also able to cause *use-after-free* memory errors. For Rust it is paramount to ensure that sleeping calls are not done within atomic contexts. For those reasons a new project, `klint`, was created. It is used like `rustc` but it also checks for atomic context violations.

Five existing and two possible future projects were discussed. The only project, which already is in the Linux mainline, is the network PHY driver for ASIX AX88796B ethernet controller. It's not the most interesting piece of hardware, but a lot of infrastructure was provided with its driver to the kernel, which will benefit future networking drivers. The Android Binder driver seems quite useful - it already implements the whole functionality of the Binder C version, but is incomplete regarding debugging facilities. The driver demonstrated outstanding performance and Rust assisted in creating abstractions regarding ownership semantics.

The NVMe and null block driver are used as a proof-of-concept to demonstrate the capabilities of Rust in a block device driver. The drivers don’t implement all features already addressed by the C version, which makes them crude and unsuitable for production. The PuzzleFS kernel file system driver is still work-in-progress, but aims also to benefit from Rust as a block device driver. Currently it suffers from missing file system abstractions in the kernel Rust API.

The Asahi Linux project ports Linux on new Apple hardware. Most importantly for this thesis, the GPU drivers are written in Rust. What makes it special is how the new GPU hardware works - the GPU is commanded by a separate coprosessor. Both Apple and Nvidia use this new design principle which offloads complicated, timing critical API calls to the coprocessor. This new design makes the interface to the operating system simpler. The developer of the open source driver Nouveau for Nvidia’s graphics cards discusses about rewriting the driver in Rust. Also, Red Hat has began a new project, Nova, which aims to be the next-generation open source Nvidia driver for the GPUs with a GSP. In summary, both the Apple and Nvidia drivers benefit from Rust’s powerful macro system for creating drivers, which are supposed to run on multiple GPU interfaces simultaneously.

A comparison of the object files text sections between Rust and C drivers was made for two different device drivers, the ASIX AX88796B and the Android Binder. The comparison pointed out, that the Rust code is somewhat bigger than the C counterpart but more sampling would still be needed.

The project *Rust for Linux* has a long journey ahead, and it’s not easy to take it into use until more interfaces are developed. However, no new interfaces are developed until someone needs them, so the project advances after necessity.

Bibliography

[1] “Android Open Source Project”. Accessed: Nov. 16, 2023. [Online]. Available: https://source.android.com/

[2] “Interprocess Communications”. Accessed: Nov. 15, 2023. [Online]. Available: https://learn.microsoft.com/fi-fi/windows/win32/ipc/interprocess-communications

[3] K. N. Billimoria, *Linux Kernel Programming*, 0th ed., vol. 0. Packt Publishing, 2021.

[4] “What is MEMS”. Accessed: Dec. 11, 2023. [Online]. Available: https://www.memsnet.org/mems/what_is.html

[5] “Generic Mutex Subsystem”. Accessed: Oct. 17, 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/locking/mutex-design.html

[6] “A Collection of Open Standards”. Accessed: Apr. 02, 2024. [Online]. Available: https://nvmexpress.org/about/

[7] “RAII”. Accessed: Oct. 21, 2023. [Online]. Available: https://doc.rust-lang.org/rust-by-example/scope/raii.html

[8] “Linux Device Driver Tutorial – Part 1 | Introduction”. Accessed: Aug. 22, 2023. [Online]. Available: https://embetronicx.com/tutorials/linux/device-drivers/linux-device-driver-part-1-introduction/

[9] B. Roch, “Monolithic kernel vs. Microkernel”, [Online]. Available: https://web.cs.wpi.edu/~cs3013/c12/Papers/Roch_Microkernels.pdf

[10] Paul Mckenney, “Multi-Core Systems Modeling for Formal Verification of Parallel Algorithms”. Accessed: Apr. 02, 2024. [Online]. Available: https://www.researchgate.net/figure/Schematic-of-RCU-Grace-Period-and-Read-Side-Critical-Sections_fig2_228576235

[11] A. Hindborg, “Linux Rust NVMe Driver Status Update presentation slides”. Accessed: Sep. 29, 2023. [Online]. Available: https://lpc.events/event/16/contributions/1180/attachments/1017/1961/deck.pdf

[12] “What is Device Driver?”. Accessed: Apr. 03, 2024. [Online]. Available: https://www.sharetechnote.com/html/Linux_DeviceDriver.html

[13] K. N. Billimoria, *Linux Kernel Programming Part 2 - Char Device Drivers and Kernel Synchronization : Create User-Kernel Interfaces, Work with Peripheral I/o, and Handle Hardware Interrupts*, First edition. Birmingham, England: Packt Publishing, 2021.

[14] “Character device drivers”. Accessed: Feb. 27, 2024. [Online]. Available: https://linux-kernel-labs.github.io/refs/heads/master/labs/device_drivers.html

[15] “Email Binder IPC”. Accessed: Nov. 10, 2023. [Online]. Available: https://lore.kernel.org/rust-for-linux/20231101-rust-binder-v1-0-08ba9197f637@google.com/

[16] “Rust for Linux: Adding support for the Rust language to the Linux kernel”. Accessed: Aug. 08, 2023. [Online]. Available: https://github.com/Rust-for-Linux/linux

[17] “null block driver”. Accessed: Sep. 20, 2023. [Online]. Available: https://rust-for-linux.com/null-block-driver

[18] A.-A. Miculas, “Puzzlefs - The Next-Generation Container File system”. Accessed: Nov. 01, 2023. [Online]. Available: https://www.youtube.com/watch?v=OhMtoLrjiBY

[19] “Overview: Safe Pinned Initialization”. Accessed: Dec. 07, 2023. [Online]. Available: https://y86-dev.github.io/blog/safe-pinned-initialization/overview.html

[20] “Klint: Compile-time Detection of Atomic Context Violations for Kernel Rust Code”. Accessed: Aug. 08, 2023. [Online]. Available: https://www.memorysafety.org/blog/gary-guo-klint-rust-tools/

[21] “Yocto releases”. Accessed: Dec. 12, 2023. [Online]. Available: https://wiki.yoctoproject.org/wiki/Releases

[22] P. J. Salzman, M. Burian, O. Pomerantz, B. Mottram, and J. Huang, “The Linux Kernel Module Programming Guide”. Accessed: Aug. 09, 2023. [Online]. Available: https://sysprog21.github.io/lkmpg

[23] “miscdevice.h”. Accessed: Feb. 14, 2024. [Online]. Available: https://elixir.bootlin.com/linux/latest/source/include/linux/miscdevice.h

[24] “What is RCU? -- "Read, Copy, Update"”. Accessed: Oct. 13, 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/RCU/whatisRCU.html

[25] “klint source code”. Accessed: Oct. 23, 2023. [Online]. Available: https://github.com/Rust-for-Linux/klint

[26] “How to use klint”. Accessed: Feb. 06, 2024. [Online]. Available: https://rust-for-linux.zulipchat.com/#narrow/stream/291565-Help/topic/How.20to.20use.20klint

[27] “PuzzleFS”. Accessed: Nov. 01, 2023. [Online]. Available: https://github.com/project-machine/puzzlefs

[28] “Rust for Linux”. Accessed: Aug. 08, 2023. [Online]. Available: https://rust-for-linux.com/

[29] “SocHub”. Accessed: Apr. 27, 2024. [Online]. Available: https://sochub.fi/

[30] J. Corbet, G. Kroah-Hartman, and A. Rubini, *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc, 2005.

[31] D. P. Bovet, *Understanding the Linux kernel*, 3rd ed. Sebastopol (CA): O'Reilly, 2006. [Online]. Available: https://raw.githubuserconten

t.com/AugustTan/documents/master/Understanding%20the%20Linux%20Kernel%2C%203rd%20Edition.pdf

[32] “What is the Linux kernel?”. Accessed: Aug. 11, 2023. [Online]. Available: https://www.redhat.com/en/topics/linux/what-is-the-linux-kernel

[33] I. Haikala, *Käyttöjärjestelmät*, 2. uud. p. in Valikko-sarja. Helsinki: Talentum, 2004.

[34] “Dirty COW (CVE-2016-5195) is a privilege escalation vulnerability in the Linux Kernel”. Accessed: Apr. 01, 2024. [Online]. Available: https://dirtycow.ninja/

[35] “GNU Hurd”. Accessed: Aug. 24, 2023. [Online]. Available: https://www.gnu.org/software/hurd/

[36] A. Hindborg, “Linux Rust NVMe Driver Status Update presentation”. Accessed: Sep. 29, 2023. [Online]. Available: https://www.youtube.com/watch?v=BwywU1MqW38

[37] “What is a device driver?”. Accessed: Aug. 16, 2023. [Online]. Available: https://learn.microsoft.com/en-us/windows-hardware/drivers/gettingstarted/what-is-a-driver-

[38] “Linux vs. Windows device driver model: architecture, APIs and build environment comparison”. Accessed: Aug. 16, 2023. [Online]. Available: https://www.xmodulo.com/linux-vs-windows-device-driver-model.html

[39] “FUSE”. Accessed: Jan. 24, 2024. [Online]. Available: https://www.kernel.org/doc/html/latest/filesystems/fuse.html

[40] “devices.txt”. Accessed: Jan. 24, 2024. [Online]. Available: https://www.kernel.org/doc/Documentation/admin-guide/devices.txt

[41] “[RFC] Rust support”. Accessed: Oct. 10, 2023. [Online]. Available: https://lkml.org/lkml/2021/4/14/1023

[42] “Rust for Linux: Industry and academia support”. Accessed: Aug. 08, 2023. [Online]. Available: https://rust-for-linux.com/industry-and-academia-support

[43] Z. Li, J. Wang, M. Sun, and J. C. Lui, “Securing the device drivers of your embedded systems: Framework and prototype”, in *ACM International Conference Proceeding Series*, 2019.

[44] “The LLVM Compiler Infrastructure”. Accessed: Apr. 03, 2024. [Online]. Available: https://llvm.org/

[45] “GCC, the GNU Compiler Collection”. Accessed: Apr. 03, 2024. [Online]. Available: https://gcc.gnu.org/

[46] “Rust GCC”. Accessed: Oct. 11, 2023. [Online]. Available: https://github.com/Rust-GCC/gccrs

[47] “Rust for Linux: Rust version policy”. Accessed: Mar. 31, 2024. [Online]. Available: https://rust-for-linux.com/rust-version-policy

[48] “Rust for Linux: Unstable Features”. Accessed: Mar. 31, 2024. [Online]. Available: https://rust-for-linux.com/unstable-features

[49] “Ferrocene”. Accessed: Oct. 10, 2023. [Online]. Available: https://ferrous-systems.com/ferrocene/

[50] “Rust for Linux: Branches”. Accessed: Jan. 09, 2024. [Online]. Available: https://rust-for-linux.com/branches

[51] “Linux kernel source tree”. Accessed: Feb. 22, 2024. [Online]. Available: https://github.com/torvalds/linux

[52] Accessed: Mar. 16, 2024. [Online]. Available: https://lore.kernel.org/rust-for-linux/CALNs47uSVXk60BYHMd0kQ95p7RAuTe6ooQVTSD_eGu1i5caiPQ@mail.gmail.com/

[53] “Rust for Linux: Backporting and stable/LTS releases”. Accessed: Jan. 17, 2024. [Online]. Available: https://rust-for-linux.com/backporting-and-stable-lts-releases#backporting-and-stablelts-releases

[54] “Rust for Linux: Out-of-tree modules”. Accessed: Jan. 09, 2024. [Online]. Available: https://rust-for-linux.com/out-of-tree-modules

[55] “The Cargo Book”. Accessed: Sep. 15, 2023. [Online]. Available: https://doc.rust-lang.org/cargo/

[56] “Rust for Linux: Third-party crates”. Accessed: Jan. 09, 2024. [Online]. Available: https://rust-for-linux.com/third-party-crates

[57] “Third-party crates support: proc-macro2, quote, syn, serde and serde_derive #1007”. Accessed: Mar. 08, 2024. [Online]. Available: https://github.com/Rust-for-Linux/linux/pull/1007

[58] “Foreign Function Interface”. Accessed: Feb. 11, 2024. [Online]. Available: https://doc.rust-lang.org/rust-by-example/std_misc/ffi.html

[59] “bindgen”. Accessed: Feb. 11, 2024. [Online]. Available: https://github.com/rust-lang/rust-bindgen

[60] “The bindgen User Guide”. Accessed: Feb. 11, 2024. [Online]. Available: https://rust-lang.github.io/rust-bindgen/introduction.html

[61] “Alternative representations”. Accessed: Feb. 14, 2024. [Online]. Available: https://doc.rust-lang.org/nomicon/other-reprs.html

[62] “Foreign Function Interface”. Accessed: Feb. 11, 2024. [Online]. Available: https://doc.rust-lang.org/nomicon/ffi.html

[63] “Bindgen as a workspace #2284”. Accessed: Mar. 08, 2024. [Online]. Available: https://github.com/rust-lang/rust-bindgen/pull/2284

[64] “The First Rust-Written Network PHY Driver Set To Land In Linux 6.8”. Accessed: Feb. 16, 2024. [Online]. Available: https://www.phoronix.com/news/Linux-6.8-Rust-PHY-Driver

[65] “NVMe driver”. Accessed: Sep. 25, 2023. [Online]. Available: https://rust-for-linux.com/nvme-driver

[66] “null block device driver”. Accessed: Sep. 20, 2023. [Online]. Available: https://www.kernel.org/doc/html/latest/block/null_blk.html

[67] A. Hindborg, “[RFC PATCH 00/11] Rust null block driver”. Accessed: Sep. 24, 2023. [Online]. Available: https://lore.kernel.org/all/20230503090708.2524310-1-nmi@metaspace.dk/

[68] “Linux kernel source tree for NVMe driver and null block driver”. Accessed: Mar. 08, 2024. [Online]. Available: https://github.com/metaspace/linux

[69] A. Hindborg, “[LSF/MM/BPF TOPIC] blk_mq rust bindings”. Accessed: Sep. 20, 2023. [Online]. Available: https://lore.kernel.org/all/87y1ofj5tt.fsf@metaspace.dk/

[70] “OCIv2 Proposal Brainstorm”. Accessed: Nov. 01, 2023. [Online]. Available: https://hackmd.io/@cyphar/ociv2-brainstorm

[71] A. Sarai, “The Road to OCIv2 Images: What's Wrong with Tar?”. Accessed: Nov. 02, 2023. [Online]. Available: https://www.cyphar.com/blog/post/20190121-ociv2-images-i-tar

[72] “Android Binder Driver”. Accessed: Oct. 11, 2023. [Online]. Available: https://rust-for-linux.com/android-binder-driver

[73] “Linux kernel source tree for Android Binder”. Accessed: Mar. 08, 2024. [Online]. Available: https://github.com/Darksonn/linux/tree/master

[74] “Asahi Linux”. Accessed: Oct. 11, 2023. [Online]. Available: https://asahilinux.org/

[75] A. Lina, “Tales of the M1 GPU”. Accessed: Nov. 23, 2023. [Online]. Available: https://asahilinux.org/2022/11/tales-of-the-m1-gpu/

[76] “Speaker support in Asahi Linux”. Accessed: Nov. 27, 2023. [Online]. Available: https://github.com/AsahiLinux/docs/wiki/SW:Speakers

[77] “Hardware Reverse Engineering with the m1n1 Hypervisor”. Accessed: Nov. 30, 2023. [Online]. Available: https://asahilinux.org/2021/08/progress-report-august-2021/#hardware-reverse-engineering-with-the-m1n1-hypervisor

[78] “Red Hat's Long, Rust'ed Road Ahead For Nova As Nouveau Driver Successor”. Accessed: Mar. 21, 2024. [Online]. Available: https://www.phoronix.com/news/Red-Hat-Nova-Rust-Abstractions

[79] “Linux kernel source tree for Asahi Linux”. Accessed: Mar. 08, 2024. [Online]. Available: https://github.com/AsahiLinux/linux/tree/asahi

[80] “A Nouveau graphics driver update”. Accessed: Feb. 04, 2024. [Online]. Available: https://lwn.net/Articles/953144

[81] “Nova and staging Rust abstractions”. Accessed: Mar. 25, 2024. [Online]. Available: https://lore.kernel.org/dri-devel/Zfsj0_tb-0-tNrJy@cassiopeiae/

[82] “Rust Bindings Posted For KMS Drivers, VKMS Ported To Rust”. Accessed: Mar. 23, 2024. [Online]. Available: https://www.phoronix.com/news/Linux-Rust-KMS-RVKMS

[83] “Rust for Linux: The Safe Pinned Initialization Problem”. Accessed: Aug. 08, 2023. [Online]. Available: https://rust-for-linux.com/the-safe-pinned-initialization-problem

[84] “RFC: Field projection”. Accessed: Dec. 28, 2023. [Online]. Available: https://github.com/rust-lang/rfcs/pull/3318

[85] “How do I make character device?”. Accessed: Mar. 03, 2024. [Online]. Available: https://rust-for-linux.zulipchat.com/#narrow/stream/291565-Help/topic/How.20do.20I.20make.20character.20device.3F

[86] “Rust Kernel Module: Character Device Driver”. Accessed: Mar. 08, 2024. [Online]. Available: https://wusyong.github.io/posts/rust-kernel-module-02/

[87] “Release 3.1 (dunfell)”. Accessed: Dec. 12, 2023. [Online]. Available: https://docs.yoctoproject.org/migration-guides/migration-3.1.html

[88] “Release notes for 4.0 (kirkstone)”. Accessed: Dec. 12, 2023. [Online]. Available: https://docs.yoctoproject.org/migration-guides/release-notes-4.0.html

[89] “Release notes for 4.2 (mickledore)”. Accessed: Dec. 12, 2023. [Online]. Available: https://docs.yoctoproject.org/next/migration-guides/release-notes-4.2.html

[90] “Release notes for 4.3 (nanbield)”. Accessed: Dec. 12, 2023. [Online]. Available: https://docs.yoctoproject.org/dev/migration-guides/release-notes-4.3.html

[91] “Release 5.0 LTS (scarthgap)”. Accessed: Apr. 03, 2024. [Online]. Available: https://docs.yoctoproject.org/dev//migration-guides/migration-5.0.html

[92] “Linux kernel source tree for PuzzleFS”. Accessed: Mar. 08, 2024. [Online]. Available: https://github.com/ariel-miculas/linux/tree/puzzlefs

[93] “Nova project kernel tree”. Accessed: Mar. 25, 2024. [Online]. Available: https://gitlab.freedesktop.org/drm/nova

[94] “Giving Rust a chance for in-kernel codecs”. Accessed: May 13, 2024. [Online]. Available: https://lwn.net/Articles/970565/

[95] “Convert between Julian day numbers and Julian & Gregorian calendars”. Accessed: Feb. 21, 2024. [Online]. Available: https://github.com/jwodder/julian-rs

[96] Oliver Montenbruck, *Grundlagen der Ephemeridenrechnung*, 2nd ed. Sterne und Weltraum Dr. Vehrenberg GmbH München, 1985.

[97] “Primitive Type f64”. Accessed: Feb. 21, 2024. [Online]. Available: https://doc.rust-lang.org/std/primitive.f64.html

[98] “A port of musl's libm to Rust.”. Accessed: Feb. 21, 2024. [Online]. Available: https://github.com/rust-lang/libm

[99] “Mentorship Session: Setting Up an Environment for Writing Linux Kernel Modules in Rust”. Accessed: May 24, 2023. [Online]. Available: https://www.youtube.com/watch?v=tPs1uRqOnlk

A Setting up the development environment

In this chapter the *Rust for Linux* development environment installation is walked through. This chapter is created by following the instructions at [99].

A.1 Dependencies

The development environment is installed on a virtual machine running Fedora Linux 37. At the time, Fedora 38 is also out but the Linux kernel code needs still Clang 15 to compile. Fedora 37 uses at the time of writing this Clang 15.0.7 whereas Fedora 38 seems to have Clang 16.0.5.

Text listing 13: Dependencies

```
dnf install -y \
    make \
    llvm \
    clang \
    flex \
    bison \
    lld \
    ncurses-devel \
    elfutils-libelf-devel \
    openssl-devel \
    qemu-kvm
```

`openssl-devel` was added after build of Linux 6.6.0-rc1 with defconfig failed.

A.2 Source code

Please clone the source code for Linux and Busybox as shown in Text listing 14:

Text listing 14: Cloning the source code for Linux and Busybox

```
mkdir -p ~/src/
cd ~/src/
git clone --recurse-submodules -j$(nproc) \
    https://github.com/Rust-for-linux/linux.git
git clone https://github.com/mirror/busybox.git
```

A.3 Scripts

We will be using a few scripts so we don’t need to remember everything by heart. The author placed the scripts under `/home/elias/src/scripts/kernel/` but it should not matter where they are - except for the script which is used to build the kernel shown in Code listing 17, since it also calls the script for creating the initramfs shown in Code listing 21.

We compile Linux using `LLVM=1` because using Rust in the kernel does not yet work with GCC alone. The script Code listing 15 is used to configure the Linux kernel with default values. Target architecture is set to `x86_64`.

Code listing 15: Script for configuring the kernel with default values

```
#!/bin/bash
# Location: ~/src/scripts/kernel/configure.sh
set -euo pipefail
cd ~/src/linux
make LLVM=1 ARCH=x86_64 allnoconfig qemu-busybox-min.config rust.config
```

For manual configuration of the kernel we use a tool called `menuconfig`. It can be invoked using script Code listing 16.

Code listing 16: Script for configuring the kernel with menuconfig

```
#!/bin/bash
# Location: ~/src/scripts/kernel/menuconfig.sh
set -euo pipefail
cd ~/src/linux
# make LLVM=1 ARCH=x86_64 MENUCONFIG_COLOR=blackbg menuconfig
make LLVM=1 ARCH=x86_64 menuconfig
```

After the kernel is configured, it can be compiled using the script Code listing 17:

Code listing 17: Script for building the kernel

```
#!/bin/bash
# Location: ~/src/scripts/kernel/build.sh
#     https://stackoverflow.com/questions/19333918/dont-add-to-linux-kernel-version
# LOCALVERSION= is needed so that the kernel version is not appended with a '+'.
set -euo pipefail
cd ~/src/linux
make LOCALVERSION= LLVM=1 ARCH=x86_64 -j`nproc`
~/src/scripts/kernel/mkinitramfs.sh
```

For running the kernel, we need something to test it with. Once the initramfs has been created with the script Code listing 21, the kernel can be run using the script Code listing 18.

Code listing 18: Run kernel with Busybox

```
#!/bin/bash
# Location: ~/src/scripts/kernel/run.sh
set -euo pipefail
qemu-system-x86_64 \
    -nographic \
    -kernel ~/src/linux/vmlinux \
    -initrd ~/src/busybox/ramdisk.img \
    -nic user,model=rtl8139
```

Linux has a strictly specified Rust version so it's important to check it each time, the kernel source code is updated using `git pull`. The correct Rust version can be set using Code listing 19.

Code listing 19: Update `rustc`

```
#!/bin/bash
# Location: ~/src/scripts/kernel/update_rustc.sh
set -euo pipefail
cd ~/src/linux
rustup override set $(scripts/min-tool-version.sh rustc)
cargo install --locked --version $(scripts/min-tool-version.sh bindgen) bindgen-cli
rustup component add rust-src
# Check if Rust is available
make LLVM=1 ARCH=x86_64 rustavailable
```

The Busybox will be used as an initial RAM based file system, initramfs. It can be created using the script Code listing 21. The initramfs is just a compressed image of a root file system containing Busybox. Optional rust-analyzer bindings may be created using Code listing 20.

Code listing 20: Setup `rust-analyzer`

```
#!/bin/bash
set -euo pipefail

cd ~/src/linux/
make LLVM=1 ARCH=x86_64 rust-analyzer
```

Code listing 21: Create initramfs for Busybox

```
#!/bin/bash
set -euo pipefail

# make modules_install could be used, however it tries
# to install them in the host os /lib/modules/<version>
# so I can't use it for this project
#
KERNEL_VERSION="6.9.0-rc1"
BUSYBOX_INSTALL_PATH="${HOME}/src/busybox/_install"
INITRAMFS_PATH="${HOME}/src/busybox/ramdisk.img"
MODULE_PATH="${BUSYBOX_INSTALL_PATH}/lib/modules/${KERNEL_VERSION}"

# Clear old modules
echo "Clearing old modules"
rm -rf "${BUSYBOX_INSTALL_PATH}/lib/modules"

# modprobe wants the modules to be in this directory.
# The kernel version can be obtained by, e.g., `uname -r`
mkdir -p ${MODULE_PATH}

# Copy kernel modules into initramfs
cd ~/src/linux
find -iname "*.ko" -exec cp {} ${MODULE_PATH} \;
cd ~/src/busybox/_install
find . |cpio -H newc -o |gzip > ${INITRAMFS_PATH}
```

A.4 Busybox

Dependencies for Busybox shown in Text listing 15 should be installed after which Busybox can be configured using Text listing 16.

Text listing 15: Installing dependencies for Busybox

```
sudo dnf install -y \
    glibc-static-2.37-1.fc38.x86_64 \
    make \
    gcc \
    ncurses-devel
```

Text listing 16: Configuring Busybox

```
cd ~/src/busybox

# Configure
make defconfig
make menuconfig

# Inside Menuconfig, please select (enable)
# Settings
#     -> Build Options
#         -> Build Static Binary (no shared libs)
# This should set CONFIG_STATIC=y in the configuration file .config

make -j`nproc`
make install
```

The initramfs containing Busybox can be created using Code listing 21. Busybox can be configured by writing configuration files under its `etc`, for instance Text listing 17. Note that every time it is modified, the initramfs must be recreated to include the new configuration files.

Text listing 17: Busybox configuration at `init.d/rcS`

```
mkdir -p /dev /proc /sys
mount -t devtmpfs none /dev
mount -t proc proc /proc
mount -t sysfs sysfs /sys
mkdir -p /var/log
ifconfig lo up
```

Next the Linux kernel needs to be compiled, before it can be run with the Busybox initramfs.

A.5 Building Linux and running it with Busybox

In this section Linux is compiled using native tools. If some dependencies can't be satisfied, a Docker container can also be used for compiling Linux like described in section A.6. First, Rust must be installed as described in Text listing 18:

Text listing 18: Installing Rust

```
curl --proto '=https' --tlsv1.2 -sSf https://sh.rustup.rs | sh
source "$HOME/.cargo/env"
```

Next, the kernel needs to be configured with some default values using the script Code listing 15.

We also need to install Bindgen and set the rustc version. We also need to add `rust-src` for Linux as shown in Text listing 19. Please note, that for older kernel versions (6.3.0, branch `rust`) the library to be installed should be `bindgen` instead of `bindgen-cli` as explained in section 4.5.

Text listing 19: Installing Rust

```
cd ~/src/linux
cargo install --locked --version \
    $(scripts/min-tool-version.sh bindgen) bindgen-cli
rustup override set $(scripts/min-tool-version.sh rustc)
rustup component add rust-src
```

Rust should now be available in our toolchain, it can be checked using `make LLVM=1 rustavailable`. If everything seems good, the kernel can now be configured further using the script Code listing 16 and compiled using the script Code listing 17. Linux can be run with Busybox in QEMU using Code listing 18. To exit QEMU, first press `ctrl + a` and `x`.

A.6 Optional: compiling Linux in a Docker container

Optionally, Linux can also be compiled in a Docker container. This was developed to be able to compile Linux inside Fedora 38, however it was not easy to use `rust-analyzer` on the host VM from within the container. Podman was used since it was already installed on Fedora 38.

The Linux source code must be mounted inside the container both when building the container and when compiling the kernel. The built kernel image, `vmlinux`, will be written in the source code directory.

Code listing 22: Dockerfile for Linux build container

```
FROM fedora:37

# Dependencies
RUN dnf install -y \
  make \
  llvm \
  clang \
  flex \
  bison \
  lld \
  ncurses-devel \
  elfutils-libelf-devel

# Things that are not in Fedora Docker
RUN dnf install -y \
  diffutils \
  bc

# Rust
RUN curl https://sh.rustup.rs -sSf | bash -s -- -y
ENV PATH="/root/.cargo/bin:${PATH}"

# Rust-based dependencies. Note, these require the Linux source files
# to be mounted into /src/linux on build time!
WORKDIR /src/linux
RUN cargo install --locked --version $(scripts/min-tool-version.sh bindgen)
\
  bindgen
RUN rustup override set $(scripts/min-tool-version.sh rustc)
RUN rustup component add rust-src

CMD ["/bin/bash"]
```

The build-container can be created from Dockerfile shown in Code listing 22 using the script in Code listing 23.

Code listing 23: `build_container.sh`: creating a Podman image for the build container

```
#!/bin/bash
set -euo pipefail
podman build \
  --file Dockerfile \
  --volume ${HOME}/src/linux:/src/linux:Z \
  --tag kernelbuild-f37
```

The build-container can be run with the script in Code listing 24 by providing the command to be run as an argument, e.g., `make menuconfig`. To compile the kernel, please provide `make -j$(nproc)` as an argument. Without any argument, the build container will drop to a shell.

Code listing 24: `menuconfig.sh`: run `make menuconfig`

```
#!/bin/bash
# Location: ~/src/scripts/kernel_containerized_build/menuconfig.sh
set -euo pipefail
podman container run --rm -ti --detach-keys="ctrl-z,z" \
  --volume ${HOME}/src/linux:/src/linux:Z \
        --env ARCH=x86_64 \
        --env LLVM=1 \
  localhost/kernelbuild-f37 \
  make menuconfig
```

B Other tools

B.1 `klint`

Text listing 20: Commands for installing klint [25]

```
sudo dnf install -y sqlite-devel
cd ~/src/
git clone https://github.com/nbdd0121/klint.git
cd klint
cargo install --path .
```

Text listing 21: Commands for running klint [26]

```
#!/bin/bash
set -euo pipefail

export LD_LIBRARY_PATH=$(rustup run 1.75.0 bash -c "echo \$LD_LIBRARY_PATH")
make LOCALVERSION= LLVM=1 ARCH=x86_64 RUSTC=${HOME}/src/klint/target/release/
klint
```

B.2 `puzzlefs`

Text listing 22: Commands for installing and using puzzlefs [27]

```
sudo dnf install -y capnproto
cd ~/src/
git clone https://github.com/project-machine/puzzlefs.git
cd puzzlefs

# Create image with some test files
mkdir -p /tmp/example-rootfs/algorithms
touch /tmp/example-rootfs/lorem_ipsum.txt
touch /tmp/example-rootfs/algorithms/binary-search.txt
cargo run --release -- build /tmp/example-rootfs /tmp/puzzlefs-image puzzlef-
s_example

# Mount image
mkdir /tmp/mounted-image
cargo run --release -- mount /tmp/puzzlefs-image puzzlefs_example /tmp/mounted-
image
```

C Sample kernel module in C

Code listing 25: Simple kernel module written in C [22]

```c
/*
 * hello-4.c - Demonstrates module documentation.
 */
#include <linux/init.h> /* Needed for the macros */
#include <linux/module.h> /* Needed by all modules */
#include <linux/printk.h> /* Needed for pr_info() */

MODULE_LICENSE("GPL");
MODULE_AUTHOR("LKMPG");
MODULE_DESCRIPTION("A sample driver");

static int __init init_hello_4(void)
{
    pr_info("Hello, world 4\n");
    return 0;
}

static void __exit cleanup_hello_4(void)
{
    pr_info("Goodbye, world 4\n");
}

module_init(init_hello_4);
module_exit(cleanup_hello_4);
```

D Sample kernel modules in Rust

A minimal kernel module example from the *Rust for Linux* repository is shown in Code listing 26. Code listing 27 shows printing examples in a Rust kernel module.

Code listing 26: `rust_minimal.rs` [16]

```rust
use kernel::prelude::*;

module! {
    type: RustMinimal,
    name: "rust_minimal",
    author: "Rust for Linux Contributors",
    description: "Rust minimal sample",
    license: "GPL",
}

struct RustMinimal {
    numbers: Vec<i32>,
}

impl kernel::Module for RustMinimal {
    fn init(_module: &'static ThisModule) -> Result<Self> {
        pr_info!("Rust minimal sample (init)\n");
        pr_info!("Am I built-in? {}\n", !cfg!(MODULE));

        let mut numbers = Vec::new();
        numbers.try_push(72)?;
        numbers.try_push(108)?;
        numbers.try_push(200)?;

        Ok(RustMinimal { numbers })
    }
}

impl Drop for RustMinimal {
    fn drop(&mut self) {
        pr_info!("My numbers are {:?}\n", self.numbers);
        pr_info!("Rust minimal sample (exit)\n");
    }
}
```

Code listing 27: `rust_print.rs` [16]

```rust
// SPDX-License-Identifier: GPL-2.0
use kernel::pr_cont;
use kernel::prelude::*;
module! {
    type: RustPrint,
    name: "rust_print",
    author: "Rust for Linux Contributors",
    description: "Rust printing macros sample",
    license: "GPL",
}
struct RustPrint;
fn arc_print() -> Result {
    use kernel::sync::*;
    let a = Arc::try_new(1)?;
    let b = UniqueArc::try_new("hello, world")?;
    pr_info!("{}", a);
    pr_info!("{:?}", b);
    let a: Arc<&str> = b.into();
    let c = a.clone();
    dbg!(c);
    pr_info!("{:#x?}", a);
    Ok(())
}
impl kernel::Module for RustPrint {
    fn init(_module: &'static ThisModule) -> Result<Self> {
        pr_info!("Rust printing macros sample (init)\n");
        pr_emerg!("Emergency message (level 0) without args\n");
        pr_alert!("Alert message (level 1) without args\n");
        pr_crit!("Critical message (level 2) without args\n");
        pr_err!("Error message (level 3) without args\n");
        pr_warn!("Warning message (level 4) without args\n");
        pr_notice!("Notice message (level 5) without args\n");
        pr_info!("Info message (level 6) without args\n");
        pr_info!("A line that");
        pr_cont!(" is continued");
        pr_cont!(" without args\n");
        pr_emerg!("{} message (level {}) with args\n", "Emergency", 0);
        pr_alert!("{} message (level {}) with args\n", "Alert", 1);
        pr_crit!("{} message (level {}) with args\n", "Critical", 2);
        pr_err!("{} message (level {}) with args\n", "Error", 3);
        pr_warn!("{} message (level {}) with args\n", "Warning", 4);
        pr_notice!("{} message (level {}) with args\n", "Notice", 5);
        pr_info!("{} message (level {}) with args\n", "Info", 6);
        pr_info!("A {} that", "line");
        pr_cont!(" is {}", "continued");
        pr_cont!(" with {}\n", "args");
        arc_print()?;
        Ok(RustPrint)
    }
}
impl Drop for RustPrint {
    fn drop(&mut self) {
        pr_info!("Rust printing macros sample (exit)\n");
    }
}
```

E Linked list with memory error

Code listing 28: `list_head` implementation in C with a memory error in function `add_element` [19]

```c
#include<stdio.h>

struct list_head {
    struct list_head* next;
    struct list_head* prev;
};

void init_list_head(struct list_head* head) {
    head->next = head;
    head->prev = head;
}

void add_list(struct list_head* head, struct list_head* element) {
    element->next = head->next;
    head->next->prev = element;
    head->next = element;
    element->prev = head;
}

void debug_iter_list(struct list_head* head) {
    struct list_head* start = head;
    do {
        printf("%p\n", head);
        head = head->next;
    } while(head != start);
}

struct list_head add_element(struct list_head* head) {
    struct list_head element;
    init_list_head(&element);
    add_list(head, &element);
    return element;
}

int main() {
    struct list_head head;
    init_list_head(&head);
    struct list_head element = add_element(&head);
    printf("iterating list...\n");
    debug_iter_list(&head);
}
```

F `objdump-parse.py`

Code listing 29: Python script for parsing the output of `objdump -h`

```python
from collections import defaultdict
import fileinput
import json
import sys

# Elias Meyer, 2024-03-14
# Tool for parsing `objdump` output
# Usage: `objdump -h obj/rust_binder.o | python objdump-parse.py`

def main():
    content = sys.stdin.readlines()

    # Search first line
    for line in content:
        if line.strip().startswith("Idx Name"):
            break

    fields = {}
    for line in content:
        s = line.strip().split()
        if s is not None and len(s) >= 3 and s[0].isdigit():
            fields[s[1]] = int(s[2], 16)

    result = json.dumps(categorize(fields), sort_keys=True, indent=4)
    print(result)

def categorize(d):
    fields_text = [ ".exit.text", ".init.text", ".text" ]
    fields_data = [ ".data", ".exit.data", ".init.data" ]
    fields_bss = [ ".bss", ]
    fields_rodata = [
        ".rodata", ".rodata.str1.1", ".rodata.cst8",
        ".rodata.cst16", ".rodata.cst4", ".rodata.cst32"
    ]

    result = defaultdict(int)
    for f in d:
        if f in fields_text:
            result["text"] += d[f]
        elif f in fields_data:
            result["data"] += d[f]
        elif f in fields_rodata:
            result["rodata"] += d[f]
        elif f in fields_bss:
            result["bss"] += d[f]
        elif "debug" in f:
            result["debug"] += d[f]
    return result

if __name__ == "__main__":
    main()
```