ETH zürich

ETH Library

Linux as a universal boot loader for new operating systems

Master Thesis

Author(s):
Walter, Jan Nino

Publication date:
2022-05

Permanent link:
https://doi.org/https://doi.org/10.3929/ethz-b-000583404

Rights / license:
In Copyright - Non-Commercial Use Permitted

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Master's Thesis Nr. 391

Systems Group, Department of Computer Science, ETH Zurich

Linux as a universal boot loader for new operating systems

by

Jan Nino Walter

Supervised by

Daniel Schwyn
Prof. Dr. Timothy Roscoe

November 2021 – May 2022

D INFK

Abstract

Booting a non-Linux operating system on ARM-based systems is difficult. The booting infrastructure developed targeting solely Linux, Linux in turn evolved to be bootable on a wide range of such systems. As a result, it is simple to boot Linux, but difficult to boot a non-Linux OS on an ARM-based system. In this thesis, we use Linux as a boot loader to boot a target, non-Linux kernel. The idea is to leave the boot and initialization process to Linux, then take over the initialized system. We focus on making booting simpler for new research kernels in development. To make the development of a new kernel simpler, we let Linux continue its execution after boot in order to support the new kernel.

In this thesis, we show the feasibility of such a boot loader and describe a concrete implementation. This boot loader implementation is able to boot a simple kernel. The boot process never turns a CPU off, thus leaving it in an initialized state. A virtual address space with an identity map is created for the booted kernel. The boot environment can be modified to suit specific use cases. We show that it is possible to use virtio to let the booted kernel use a device through Linux, by letting it use a virtio console in that way. The device type specific implementation of virtio is decoupled from the specific physical device, so by implementing it once, all physical devices of that type can be used. The boot loader itself profits from the hardware and software support of Linux as well, enabling it to use devices, file systems and networking out of the box.

Acknowledgements

I want to thank my advisors Daniel Schwyn, Timothy Roscoe and David Cock for their support. Their guidance and ideas helped me out a lot and I learned much thanks to them.
I also want to thank the other members of the Systems Group for their help.
Finally I want to thank my family and friends for their support.

Contents

1 Introduction **1**

2 Background **3**
2.1 Linux Kernel Virtual Memory Layout 3
2.2 Kexec 3
2.3 Virtio 5
2.3.1 Virtio Standard 6
2.3.2 Hypervisor-less Virtio 7

3 Related Work **8**
3.1 Kexec based Linux Boot Loaders 8
3.2 Popcorn 9

4 Implementation **11**
4.1 Resource Restriction 11
4.2 Memory Management 12
4.2.1 Memory Assignment 13
4.2.2 Map Memory 13
4.2.3 Detect Restricted Memory 15
4.3 Load Kernel Code 16
4.4 Boot Kernel 16
4.4.1 Boot Process 17
4.4.2 Verification 18
4.4.3 Old Implementation using a Kernel Thread 18
4.5 Boot Environment 19
4.5.1 Virtual Address Space 19
4.5.2 Modifying the Boot Environment 20
4.6 Accessing Devices 21
4.6.1 Access a Device Directly 21
4.6.2 Virtio Overview 22
4.6.3 Shared Memory 22
4.6.4 MMIO 23
4.6.5 Notifications 25
4.6.6 Implementing the Virtio Console 26
4.7 Kernel Module Interface 27
4.7.1 Kernel Instances 28

5 Kernel Module Interface **30**
5.1 Ioctl 30
5.2 General Interface 30
5.3 Instance Interface 31

6 Evaluation **34**
6.1 Basic Boot 34
6.2 Consecutive Boot 35
6.3 Access UART 36
6.4 Linux Memory Usage 36
6.5 Virtio Console Overhead 37

7 Future Work **40**
7.1 Linux Support after Boot 40
7.2 Setting up Registers 41
7.3 Exception Vector Table 41
7.4 Share Memory between Rust Kernels 42
7.5 Release a Kernel Instance 42

8 Conclusion **44**

1 Introduction

Booting an operating system that is not Linux on an ARM-based system is difficult. There are many different loaders which are all designed to solely boot Linux. At the same time, the Linux kernel evolved to be able to boot on a wide variety of ARM-based systems. This results in the situation where it is simple to boot Linux on an ARM system, but very difficult to boot a non-Linux OS. It gets even worse if the non-Linux OS should be bootable on a range of such systems. As a consequence, academic research in OS design and implementation targets x86 for the most part. If it does target ARM, then just a single system with no hope of wide portability.

This thesis explores the idea of using Linux as a boot loader to boot a non-Linux OS. Instead of dealing with booting, why not leave it to Linux which is already able to boot on a wide range of ARM systems. After booting Linux, it could load the non-Linux OS kernel and boot it. Using Linux as a boot loader has the benefit that any system that boots Linux can in theory boot a non-Linux OS. Another advantage is that such a boot loader can take advantage of existing software support of Linux. Whether device drivers, network protocols or file system support, they are all supported and maintained by Linux, so the boot loader can just use them.

To make ARM-based systems more appealing for operating system research, we put the focus on booting new operating systems that are in development. The system should be handed over to such a new OS in an initialized state, the less initialization the new OS has to perform the better. When developing a research OS, one might want to explore a specific part of OS design, but require functionality outside of that focus. To give an example, such functionality could be storing a file, sending a network packet or simply printing a message. Instead of implementing such functionality in the new OS, it could be provided by Linux. Offloading functionality that involves interaction with devices makes the new OS more portable, since you can make Linux take care of the device interaction. So, in contrast to regular boot loaders, we keep Linux running after boot in order to support the new OS.

The goal of this thesis is to show the feasibility of using Linux as a boot loader that keeps executing after booting a kernel in order to support it. The intention is to make a boot loader suited for OS development. The boot process should not reset the initialization performed by Linux and keep the system in an initialized state. Linux should support the booted kernel by letting it use a device through it. As Linux and the booted kernel execute in parallel, they need to be restricted in the resources they use to prevent interfering with each other.

The concrete target platform of this project are ARMv8-based systems and OS kernels written in Rust. We will call the kernel that is booted by Linux the Rust kernel in this thesis. This makes it clearer whether we refer to the Linux kernel or to the kernel being booted. The kernel does not need to be written in Rust, our boot loader just expects that the kernel is an ELF executable. The modifications to Linux should be contained inside a Linux kernel module

if possible. A kernel module is easier to use and debug compared to modifying the Linux kernel itself. A modified kernel needs to be compiled in its entirety, a kernel module can simply be loaded into a running kernel. Porting a kernel module to a newer kernel version is simpler as well, since all changes are contained in one place. In this thesis, we assume that the machine is able to boot Linux in some way. How Linux boots is not relevant for this thesis, as we are always working with a machine running Linux and try to boot a Rust kernel from there.

We will now give a brief overview on how this boot loader works. The boot process to boot a Rust kernel on a CPU tricks Linux into believing that the CPU is turned off, while it is actually executing the Rust kernel. This is achieved by disabling the part in Linux's CPU shutdown process that actually shuts down the CPU. Believing that the CPU is shut down, Linux does not interact with the CPU in any way, thus it does not interfere with the Rust kernel. Since the CPU is never turned off, it stays in an initialized state. The boot loader allows the Rust kernel to use a console device through Linux. Using the console through Linux is implemented using virtio. Linux sets up a virtual device and forwards requests made by the Rust kernel to that virtual device to the actual physical device. This level of indirection makes the Rust kernel independent of the specific physical device.

2 Background

In this section, background knowledge is provided that this thesis builds on. It first describes two Linux specific concepts, the kernel virtual memory layout of Linux and kexec, a mechanism to boot a new kernel from Linux. Then it describes virtio, which is used by our boot loader to let the Rust kernel use a virtio console through Linux.

2.1 Linux Kernel Virtual Memory Layout

Virtual memory is split into two parts, the lower half belonging to user space and the upper half belonging to kernel space. The Linux kernel structures the kernel address space into regions. The virtual memory layout is shown in Table 1 for the arm64 Linux kernel with 48-bit virtual addresses and `4kB` pages [1].

The relevant memory regions for this thesis are the *kernel logical memory map* and the *vmalloc region*. The Linux kernel maps the entire memory in the *kernel logical memory map* during boot. In this region each virtual address correspond to the physical address plus a constant offset. This makes it simple to switch between virtual and physical addresses and memory is physically contiguous. The *vmalloc region* is used for allocating virtually contiguous memory that is not necessarily physically contiguous. This allows to allocate large virtually contiguous chunks of memory, but it is slower as the mappings need to be created. Switching between virtual and physical addresses is slower as well.

In ARM-based systems there are two registers to store the currently active page tables: TTBR0 and TTBR1 (Translation Table Base Register). TTBR0 points to the user address space page table and TTBR1 points to the kernel address space page table. Which page table is used for resolving an address is determined by the first bits of the address. For 48-bit virtual addresses, it is determined by the first 16 bits. If they are all zero, then the address is resolved in the TTBR0 page table, if they are all one, then in the TTBR1 page table.

2.2 Kexec

Kexec is a mechanism in Linux that allows to boot a new kernel from the currently running kernel. It does so without handing control to the firmware and restarting the system. The currently running Linux kernel is replaced by the new kernel and stops execution. Several projects that use Linux as a boot loader use kexec at their core to perform the boot process. In the beginning of this project we were considering to use kexec as well, but decided against it later on. This section describes how kexec works.

Booting a new kernel with kexec is performed in two steps, first loading a new kernel with the system call kexec_load, then executing the new kernel with the system call reboot, passing the argument *LINUX_REBOOT_CMD_KEXEC*. Kexec can be used in two ways, one is to reboot into a new kernel, the other is to prepare a crash kernel that takes over once the main kernel crashes. The

Region	Start	End
user space	0x0000000000000000	0x0000ffffffffffff
kernel logical memory map	0xffff000000000000	0xffff7fffffffffff
[kasan shadow region]	0xffff600000000000	0xffff7fffffffffff
bpf jit region	0xffff800000000000	0xffff800007ffffff
modules	0xffff800008000000	0xffff80000fffffff
vmalloc	0xffff800010000000	0xfffffbffefffffff
fixed mappings (top down)	0xfffffbfff0000000	0xfffffbfffdffffff
[guard region]	0xfffffbfffe000000	0xfffffbfffe7fffff
PCI I/O space	0xfffffbfffe800000	0xfffffbffff7fffff
[guard region]	0xfffffbffff800000	0xfffffbffffffffff
vmemmap	0xfffffc0000000000	0xfffffdffffffffff
[guard region]	0xfffffe0000000000	0xffffffffffffffff

Table 1: Virtual memory layout of Linux kernel on arm64.

intention behind booting into a crash kernel is to debug the crashed main kernel. In the following we describe these two ways of using kexec.

When using kexec to reboot into a new kernel, kexec_load loads the new kernel code into kernel memory. This location is not where the new kernel expects to be loaded, the kernel code is not contiguous and cannot be executed in its current form. The reason for loading the new kernel in this way is that kexec ultimately allows loading of the new kernel at any physical memory location. This includes locations where the currently executing kernel resides. So at the current stage the new kernel cannot be loaded at its eventual location in order to not interfere with the currently running kernel. After kexec_load completes, the user will at some point make the reboot system call. This initiates the kexec reboot, which shuts down all CPUs other than the bootstrap processor and tries to shut down all devices, such that they are in a defined and inactive state. Then it makes a copy of a small piece of code, we call it the copy-code, turns off the Memory Management Unit (MMU), disables the data caches and finally executes the copy-code. The copy-code copies the code of the new kernel to its final memory location. This copy process typically overwrites the old kernel, so it is necessary to place the copy-code in a safe location that is unaffected. Figure 1 illustrates how the memory layout is changed by the copy process. At the end of the copy process, the copy-code jumps into the new kernel code. This concludes the kexec portion of the boot process and hands control over to the new kernel.

The behavior of kexec is different when preparing a crash kernel. The goal is to collect debugging information from the main kernel when it crashes. As a consequence, the crash kernel has to be loaded differently, as overwriting the crashed main kernel deletes its state, defeating the purpose of a crash kernel intended for debugging. So when loading a crash kernel with kexec_load, it is placed into an area of memory that is reserved at boot time with the kernel parameter `crashkernel`. After kexec_load completes, the crash kernel is in an

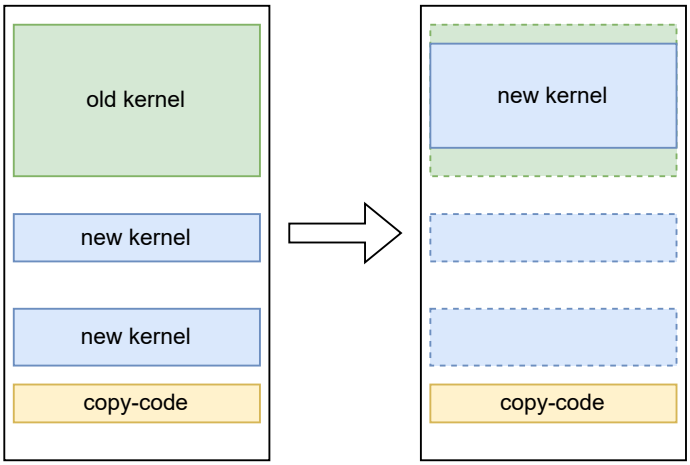

Figure 1: Kexec copy process before booting the new kernel. The copy-code copies the new kernel to its final memory location. On the left is the memory layout before the copy process, on the right afterwards.

executable state. When the main kernel crashes, the kexec mechanism simply passes control to the crash kernel.

2.3 Virtio

Virtio was introduced to provide a standardized solution for paravirtualized devices [2]. It allows the guest to access devices controlled by the host. How such accesses work is illustrated in Figure 2. If the guest wants to send a request to a physical device, then the virtio driver running on the guest sends it to the virtio device running on the host. The virtio device is just a virtual device, it uses Linux primitives like file descriptors to forward the request. Linux forwards the requests to the physical device driver, which performs the actual communication with the physical device and sends the request to it. The interface between virtio driver and virtio device is specified by the virtio standard and depends on the device type. The virtio device is independent of the specific physical device driver, since it uses Linux primitives to interact with it. The Linux primitives being used depend on the device type as well. So each device type needs a different pair of virtio driver and virtio device. But the same pair can be used to interact with any physical device driver and physical device of the same device type.

We use virtio to allow the Rust kernel to use devices through Linux. By implementing the virtio driver for the Rust kernel and the virtio device for Linux for one device type, the Rust kernel can use all physical devices of that device type. The device type that we implemented is the virtio console. We will first describe the virtio interface between virtio driver and virtio device in

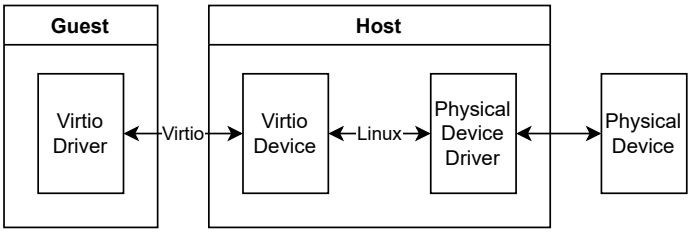

Figure 2: Interaction between components when virtualizing a physical device with virtio.

more detail, then we look into the hypervisor-less virtio project, from which we use the virtio device implementation.

2.3.1 Virtio Standard

This section contains the information on virtio that this thesis builds on. We use the virtio version 1.1 standard [3] with the legacy interface.

There are two actors in virtio: the virtio driver, running on the guest, and the virtio device, running on the host. Virtio offers three different transport modes: Peripheral Component Interconnect (PCI), Memory-Mapped I/O (MMIO) and channel I/O. We use the MMIO transport mode, so communication between virtio driver and virtio device happens through MMIO registers. If the virtio driver writes or reads an MMIO register, it traps and passes control to the virtio device, allowing it to react accordingly. However the actual data transport does not happen through MMIO registers, instead it goes through the main data structure in virtio, the virtqueue. There can be one or more virtqueues, each virtqueue is identified by a queue number. The number of virtqueues depends on the device type. A console device for example contains at least two virtqueues, one for receiving and one for transmitting characters. Data is sent through a virtqueue in buffers. Every buffer is allocated by the virtio driver and is either read-only or write-only for the virtio device. Buffers can be chained together. To send a buffer to the virtio device, the virtio driver adds it to the virtqueue and marks it as available in the virtqueue. The virtio device processes the buffer and marks it as used when it is done. Buffers are always exchanged in this way, whether the virtio driver sends data to the virtio device or the other way around. To inform the other party that a buffer is used or available, they send a notification. The virtio driver does so by writing into the *QueueNotify* MMIO register, the virtio device by causing an interrupt.

When the virtio driver writes into an MMIO register, then it can change the value that other MMIO registers will return. For example, writing the virtqueue number into the *QueueSel* register allows to select the specified virtqueue. This changes the value that will be read of four other registers, namely *QueueNumMax*, *QueueNum*, *QueueAlign* and *QueuePFN*. But because accessing MMIO registers blocks the virtio driver until the virtio device is done with the opera-

tion, this does not cause any race conditions.

2.3.2 Hypervisor-less Virtio

Virtio is usually used in the context of virtualization, but we want to use virtio between two cores without virtualization. This does not affect the virtio driver much, because to the virtio driver the virtio device looks like a regular device. But it does affect the virtio device, which expects to run on a hypervisor. In most cases, the virtio device implementations are provided by the user space hypervisor program, for example by QEMU or the Linux KVM tool. These virtio device implementations are not intended to be used stand alone without a virtual machine. But we want to avoid making a virtio device implementation from scratch, since there are already several implementations. Luckily, there is a project from OpenAMP called hypervisor-less virtio [4][5], which modifies the Linux KVM tool [6] to run a virtio device without being a hypervisor. We will call this modified version of the Linux KVM tool the hypervisor-less KVM tool from now on.

Hypervisor-less virtio targets Asymmetric Multiprocessing (AMP) systems where running one OS instance across all cores is not possible or does not satisfy real-time, concurrency or safety requirements. For example, one subsystem could run a general-purpose OS and another subsystem could run a real-time OS. Hypervisor-less virtio provides a means of communication and resource sharing between such OSes. It does so by adapting virtio to this hypervisor-less setting. It uses the MMIO transport mode of virtio, since it can not only be used to communicate downwards to a hypervisor, but also to communicate laterally between OSes. The MMIO registers are put into shared memory, accessible by both OSes. Notifications are implemented through a character device. The hypervisor-less KVM tool, which acts as the virtio device, makes system calls on the character device file to send and receive notifications. A write system call sends a notification, whereas a poll system call waits for and receives a notification. The character device implementation [7] of the hypervisor-less KVM tool works with a specific Inter Processor Interrupt (IPI) controller and is platform dependent. According to the virtio standard, a virtio device does not receive notifications from a virtio driver. But the hypervisor-less KVM tool requires them to inform the virtio device of accesses to MMIO registers by the virtio driver.

All data shared between the hypervisor-less KVM tool and the virtio driver are inside the shared memory region. This includes the MMIO registers, the virtqueues and the buffers of data. The hypervisor-less KVM tool defines the location and size of the shared memory region. It passes this information to the virtio driver through MMIO registers. These registers are at a memory location in the MMIO range that is currently not used by the virtio standard.

3 Related Work

This section describes works that are related to our boot loader and compares them. First we describe Linux boot loaders that use the kexec mechanism for booting. Then we look into Popcorn, which runs multiple kernels on a system and thus needs to solve similar problems like our boot loader.

3.1 Kexec based Linux Boot Loaders

The idea of using an operating system as a boot loader is not new. The paper "Give your bootstrap the boot: Using the operating system to boot the operating system" [8] from 2004 discusses the idea and examines three mechanisms that allow Linux to boot an OS. The mechanism of the three that is still relevant today is kexec. Since it is included in the Linux kernel, it is an appealing option for using Linux as a boot loader. In this section we will describe kexec based boot loaders, with the example of kboot, and make the comparison to our boot loader.

There are several projects that use Linux as a boot loader that are based on kexec. One of them is kboot, a proof-of-concept implementation of a Linux boot loader, which is described in the paper "kboot - A Boot Loader Based on Kexec" [9] from 2006. Other similar projects that are still active are Petitboot [10] and kexecboot [11]. We will refer to kboot in this section, but most statements apply to the other projects as well. The motivation behind kboot is that there is an overlap in features between a boot loader and Linux. Different storage devices, file system formats and network capabilities are all supported by Linux. But on the boot loader, they either need to be re-implemented and maintained, or they are not supported, dropping features and limiting the flexibility of the boot loader. By using Linux as the boot loader, one can get all this support for free. An additional advantage is that if the boot process fails, then the user can try to debug the cause in a familiar Linux environment.

Kboot is a user space program which is combined with a minimal Linux OS. This minimal Linux OS needs to be booted itself, this can happen through a boot loader like GRUB or it can be loaded directly by the EFI firmware as an EFI executable [12]. Once the minimal Linux OS is booted, kboot launches a user interface, which allows to select or modify a kernel configuration to boot. Such a kernel configuration typically contains a kernel, kernel parameters and an initramfs. The boot itself is done by the kexec mechanism, which stops the currently running minimal Linux OS and boots the target kernel. Every kernel that can be booted with kexec can be booted by kboot.

We will now compare kexec based Linux boot loaders, like kboot, with our boot loader. Both have a similar concept by using Linux as a boot loader, taking advantage of Linux's software and hardware support. The main difference is that kboot stops its own execution when it boots the target kernel, whereas our boot loader keeps running to support the target kernel. Kboot uses kexec for booting, which is implemented and maintained by the Linux kernel developers. Our boot loader does not use kexec, instead it uses its own booting mechanism,

which is implemented in a Linux kernel module. The boot environment for the target kernel is different as well. Kexec turns off the MMU, disables data caches and tries to shut down devices. In contrast, our boot loader does none of the above, it leaves them enabled to simplify the initialization process for the target kernel. In conclusion, while these boot loaders have a similar approach, they have different purposes. Kboot is intended to replace the boot loader, being feature rich and maintainable. Our bootloader is intended to reuse existing booting infrastructure for Linux to boot a non-Linux kernel and support it afterwards.

3.2 Popcorn

In this section we look into Popcorn, a replicated-kernel OS based on Linux [13]. Popcorn boots multiple Linux kernel instances, each kernel instance has its own set of CPUs, memory and physical devices. These kernels communicate directly with each other to replicate a common OS state, which is where the name "replicated-kernel OS" stems from. While Popcorn is not a boot loader, it shares some of the main tasks with our boot loader: booting a new kernel instance while keeping the previous instance running, partitioning resources between kernel instances and sharing devices among kernel instances. For this reason we describe Popcorn in this section and compare it to our boot loader.

The goal of Popcorn is to improve the performance of systems with heterogeneous Instruction Set Architecture (ISA) processors [14]. Each group of CPUs that has the same ISA runs its own Linux kernel, compiled for the ISA. Applications can be scheduled among the different CPU groups, where Popcorn Linux tries to schedule the given piece of code on the group best suited to execute it. Popcorn boots new kernel instances through a modified version of kexec. The modified kexec loads the kernel, the boot parameters and the initramfs at the target locations. Then it sets the remote CPU's instruction pointer to the kernel entry point and sends an IPI to wake the CPU up. Similar to regular Linux, a kernel instance boots on one CPU and later in the boot process starts the remaining CPUs assigned to it. The resources of the system are partitioned between the kernel instances, these partitions do not overlap. The resource partitioning scheme defines these assignments and is required prior to booting. Memory is restricted with the `memmap` kernel parameter, which excludes memory from the memory map provided by the BIOS. Devices are assigned to one owner kernel. They can still be accessed by kernels that do not own them, this happens through inter-kernel message passing. This inter-kernel message passing is implemented with shared memory and IPIs. There are two ways to access a device that is owned by another kernel: the access can be proxied by the owner kernel or ownership is passed. Which method is chosen depends on the device type. So if an application running on a local kernel makes a request to a device owned by a different kernel, the request is either forwarded to the owner kernel or gets locally staged, waiting for the ownership to pass to the local kernel.

Even though Popcorn and our boot loader have entirely different goals, they solve similar problems. We will now compare these solutions to each other.

Popcorn boots a new kernel instance by sending an IPI to an offline CPU. Our boot loader keeps the target CPU running, only making Linux believe that it was shut down. In Popcorn's case the booted CPU is uninitialized, in our boot loader's case it is initialized. Whether it is better to boot an initialized or uninitialized CPU depends on the use case, Linux expects to be booted on an uninitialized CPU, whereas writing an experimental kernel is simplified if the CPU is already initialized. An advantage of our solution is that it is contained within a kernel module. Both Popcorn and our boot loader restrict memory through kernel parameters. Device sharing has a different purpose in the two systems. In Popcorn, the goal is to allow all kernel instances to access a device, while only one kernel has full control over a device at a time. In our boot loader, the goal is to let the Rust kernel use devices through Linux, simplifying its implementation and leveraging Linux's device support. To this end, handing a device around between the Rust and the Linux kernel is not necessary, if the Rust kernel can handle a device on its own then it should get exclusive access to it. Both systems allow to access devices controlled by another kernel, but they do so in different ways. Popcorn uses inter-kernel message passing to forward application requests, our boot loader uses virtio.

4 Implementation

This section describes the implementation of the boot loader. The boot loader consists of several software components, their relations are shown in Figure 3. The `kernel module` contains the main part of the boot loader: it performs the boot process, it manages and maps memory, it loads the kernel executable and it is involved in the notification process of virtio. The user space program `bootloader_user` operates the kernel module through the kernel module interface and starts the hypervisor-less KVM tool. While a user space program like `bootloader_user` is necessary to operate the kernel module, it does not add functionality on its own, so it is rarely mentioned in this thesis. The `hypervisor-less KVM tool` contains the virtio device implementation and allows the Rust kernel to use the virtio console through Linux. `Bare Rust` is the Rust kernel that we use to boot. It is a very simple kernel in the form of a bare metal Rust program. `Bare Rust` uses a virtio driver to use the virtio console provided by the `hypervisor-less KVM tool`.

The description of the implementation is structured into the following sections: how Linux is restricted from using all the resources in the system, how memory for the Rust kernel is managed by the kernel module, how the kernel code is loaded, how a Rust kernel is booted, what the boot environment looks like, how the Rust kernel can access devices, in particular the virtio console, and finally a description of the kernel module interface. The source code is contained in a git repository, all relative paths mentioned are relative to that repository.

4.1 Resource Restriction

The Linux kernel cannot use the whole system like it usually does, it has to be restricted to leave resources for the Rust kernel to use. Such a restriction on CPU and memory can be achieved with kernel parameters during Linux boot. Our boot process of a Rust kernel takes a CPU used by Linux, restricts the Linux kernel from using it and boots the Rust kernel on it. So all CPUs can be handed to Linux when it boots. This is not the case for memory, we restrict the memory Linux accesses through the kernel parameter `mem`. `mem=1G` forces Linux to only use 1 GB of memory. The restricted, remaining part of memory is not mapped into the *kernel logical memory map* and the physical address range does not show up in `/proc/iomem`. It is possible to map memory in the restricted memory region into the *vmalloc region* using the kernel function `vmap`. It is necessary that we can map and access the memory we just restricted from the Linux kernel since we need to copy the Rust kernel there. Being able to do so does however raise the question whether there are other processes doing the same. To verify that this is not the case and our restricted memory is truly restricted, we wrote a QEMU TCG plugin that monitors all memory accesses of a QEMU virtual machine. The plugin logs any accesses that are within a certain physical memory range. There are many memory accesses in the restricted region when grub is booting, but none while Linux is running. If we initiate an access ourself, it gets properly detected by the plugin. This observation makes

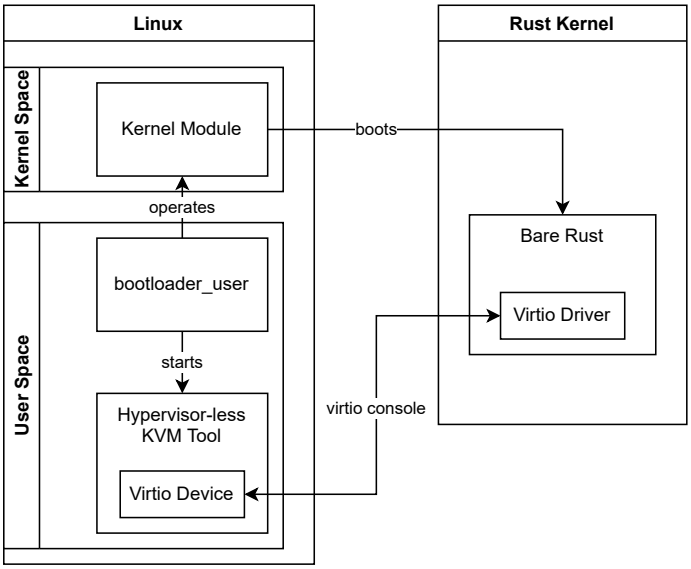

Figure 3: The software components of this boot loader and the relations between them.

us confident that restricting memory with the kernel parameter `mem` does work.

4.2 Memory Management

Linux is restricted from accessing the entire memory, this restricted memory is intended to be used by the Rust kernel and is managed by the kernel module. The main goal of this memory management is to prevent usage of memory that should not be used. The two types of memory that should not be used are memory managed by Linux, i.e. memory outside of the restricted memory, and memory used by other Rust kernels. Both cases have the same underlying problem: multiple kernels believing they have exclusive control over a memory range, while this is not the case. To prevent these cases, the kernel module allows to request memory for a Rust kernel. The memory is assigned to the Rust kernel if the memory is inside restricted memory and was not assigned before. Not assigning the same memory range multiple times prevents it from being used by two Rust kernels. But it can also help when booting a single Rust kernel. If a Rust kernel for example requires two memory ranges, one with the kernel binary and the other with some initialized data, then these ranges should not overlap. If the two memory requests accidentally do overlap, then the kernel module will reject the second request.

4.2.1 Memory Assignment

For a memory request to be accepted, it needs to be within the restricted memory and not yet assigned to a Rust kernel. So the memory management needs to know what memory is restricted and what memory was already assigned. The memory ranges that are restricted from Linux are passed by the user as a physical start and end address through a kernel module parameter. Kernel module parameters can be set when the module is loaded or, if it is built-in the kernel, through kernel parameters. It would be possible to change kernel module parameters after the module is loaded through sysfs [15], but we disallow that by setting the file permission to read-only to simplify the implementation. The kernel module stores the restricted memory ranges in a list of memory_region structs (defined in `kernel-module/memory.h`). Each memory_region struct holds a physically contiguous range of memory that is not allowed to overlap with other memory_region structs.

On a memory request, the kernel module checks whether the requested memory range is contained in a single memory_region struct. If a requested memory range is contained in two consecutive memory_region structs, the check will nonetheless fail. This simplifies the implementation and can be easily avoided by not adding consecutive restricted memory regions. Then the kernel module checks if the memory request overlaps with any of the already assigned memory ranges in the containing memory_region struct. This is done by traversing the list of assigned_memory structs (defined in `kernel-module/memory.h`) each struct memory_region has. If both of these checks succeed, then the memory request is successful and the memory range gets assigned. It is stored in the list of assigned_memory structs in the containing memory_region struct. The start and end address of the memory request do not have to be page aligned, the kernel module will extend the requested memory range to be page aligned. The difference from the requested start address to the assigned start address is stored in the field offset of struct assigned_memory. So the start and end address in struct assigned_memory are always page aligned. The kernel module fills each assigned memory range with zeros to prevent previously written content from reappearing, possibly causing bugs.

4.2.2 Map Memory

The memory management allows to map an assigned memory range in three different address spaces: in the Rust kernel's address space, in the Linux's kernel address space and in the user space program's address space. All assigned memory should be accessible to the Rust kernel, if this is not desired then one can use the regular memory allocation functionality provided by Linux. So every assigned memory range gets always mapped in the Rust kernel's address space. Because the kernel module needs to overwrite every assigned memory range with zero, it is also necessary to map them into the Linux's kernel address space. A mapping into the user space program's address space is optional and depends on how the memory range is used. Each kind of mapping is described

Linux Macro	Permission
PAGE_KERNEL	read write
PAGE_KERNEL_RO	read
PAGE_KERNEL_ROX	read execute
PAGE_KERNEL_EXEC_CONT	read write execute

Table 2: Memory protections for mapping into the Rust kernel's address space.

in more detail in the following:

Rust Kernel Address Space The function `map_into_mm()` declared in `kernel-module/memory.h` maps an assigned memory range into the new virtual address space of the Rust kernel. The kernel module creates this new address space and creates the mappings with this function before the Rust kernel is booted. For each mapping, it sets the virtual address to the same value as the physical address, resulting in identity mappings. The new virtual address space is described in more detail in Section 4.5.1. A mapping is made with a memory protection which is specified by the memory request. The memory protection determines the permission by which the memory range will be available in the Rust kernel. Since we use Linux functions to create and set up the page tables, we also use Linux macros to describe the memory protection. The memory protections the kernel module allows for the Rust kernel address space are listed in Table 2. Even though the kernel module allows the memory protections *PAGE_KERNEL_RO* and *PAGE_KERNEL_ROX*, they do not actually prevent the Rust kernel from writing into such memory ranges. But it is important to choose an executable memory protection if one wants to load code that should be executed. Otherwise executing such code will result in a Linux kernel panic. We use the memory protection *PAGE_KERNEL_EXEC_CONT* because we always map physically contiguous memory.

The kernel module implementation currently does not provide a way to free the page tables of mappings in the Rust kernel address space. The reason is that Linux functions that unmap memory (like `vm_munmap()`) require the mappings to be tracked in vm_area_struct structs, however these structs are not set up with the functions we use to create the mappings. There is no Linux function that just creates such vm_area_struct structs, so creating them manually would require copying code from the Linux kernel. Alternatively one could use the Linux function `_install_special_mapping()` to create the mappings, which creates the required vm_area_struct structs. This function uses demand paging, so it would be necessary to populate the mappings before booting the Rust kernel. We did not implement mapping memory with this function due to time constraints.

Linux Kernel Address Space The function `map_with_vmap()` uses the Linux function `vmap()` to map into the kernel address space of Linux. `vmap()` chooses a free range of virtual addresses in the *vmalloc region* of the Linux kernel and maps the memory there. The new pointer pointing to the virtual address is stored in

the assigned_memory struct, this allows other parts of the kernel module to use that mapping and access the assigned memory. When the memory is released, the pointer is used to free the mapping. Apart from setting memory to zero when it is assigned, the kernel module currently uses these mappings in the notify device to access the shared polling memory (see Section 4.6.5) and when loading an ELF executable (see Section 4.3).

User Space Program Address Space An assigned memory range gets mapped into the address space of a user space program only when the program instructs the kernel module to do so via an mmap system call. The user space program has to choose the *MAP_SHARED* flag for the mmap system call, because the memory will always be shared with the Rust kernel. The flag ensures that writes change the underlying memory and are visible to other processes.

The kernel module does not unmap mmapped memory when the assigned memory is released. To unmap it, the lower part of virtual memory belonging to user space would have to belong to the user space program that initiated the mmap. This is not necessarily the case, especially in the current implementation where the only way to release memory is to unload the kernel module. So the user space part of virtual memory will always belong to an rmmod process. Since the kernel module does not unmap or track mmapped memory, the user space program is responsible of not using mmapped memory after the assigned memory was released. Otherwise it could interfere with a new Rust kernel that requested this memory.

4.2.3 Detect Restricted Memory

The kernel module currently gets the restricted memory range from the user via a kernel module parameter (as described in Section 4.2.1). It would be convenient if the kernel module would detect the restricted memory range on its own. However there are at least two ways to do so, through the device tree or the UEFI memory map, depending on how the system is set up. There might even be system setups where neither of these work. So we decided to let the user specify the restricted memory range, because this will always work and gives the most flexibility. The current implementation could be extended to offer an automatic restricted memory detection that works on some systems. We will sketch out how this could be done for a memory map defined by a device tree or by UEFI.

To detect the restricted memory range, the kernel module has to know the entire memory available in the system and subtract the memory available to the Linux kernel. The memory available to Linux can be found in the file `/proc/iomem`. The information of this file can be accessed from the kernel module with the Linux kernel function `walk_iomem_res_desc()`. How to get the entire memory available is dependent on how the memory map of the system is handed to Linux. If the memory address ranges are stored in a device tree, one can use `of_find_node_by_name()` to find the device tree node and `of_find_property()` to find the property of the node containing the address range. If the available

memory address are stored in an UEFI memory map, one can use the function `for_each_efi_memory_desc()` to iterate over the descriptors in the UEFI memory map. Some of these descriptors describe memory that is used after boot by the UEFI runtime, so one should check the memory type and only use those that are not reserved [16]. The Linux function `is_usable_memory()` is used by the Linux kernel to check whether a descriptor is usable.

4.3 Load Kernel Code

One step in booting a kernel is to load its code into memory. The kernel module provides the functionality to load an ELF executable. An ELF executable has segments that need to be loaded into memory. Each segment specifies a virtual address where it expects to be loaded. The kernel module requests memory at that virtual address, maps it into the kernel's *vmalloc region* and copies the segment into it. The Rust kernel gets its own virtual address space with an identity map (see Section 4.2.2), so the virtual addresses specified by the ELF executable will correspond to the physical addresses. As a consequence, the addresses where the ELF segments expect to be mapped need to be valid physical memory addresses that are not used by Linux or already requested from the kernel module. So it is not possible to load the same ELF executable twice, because it will request the same memory regions from the kernel module twice. The appendix of this thesis describes how to change the ELF base address in a Rust project in order to use valid memory regions. Creating ELF executables from the same code with non overlapping memory regions allows to load it multiple times.

Instead of changing the ELF executable's virtual addresses, one could forgo the identity map of the Rust kernel's virtual address space and instead allow the user to pick the virtual address. It does not matter if two ELF executables use the same virtual addresses as long as the physical addresses are distinct. Another solution would be to support Position Independent Executables (PIE) and relocate them to valid addresses.

The ELF loader that parses and loads the ELF executable is taken from Barrelfish [17]. The kernel module implements the memory allocator that provides the ELF loader with memory. The source code of the ELF loader is located in `kernel-module/elf/`. The ELF loader does not need to be in the kernel module, so it would actually be better if it was contained in a user space program. The reason why we put it in the kernel module is that we did not have a user space program at the time. We did not change it afterwards since moving the ELF loading functionality to the user space program does not bring additional functionality.

4.4 Boot Kernel

A kernel needs, like any piece of code, a CPU core to run on. To get a clean separation between Linux and the Rust kernel, we want to remove a core from Linux and dedicate it to the Rust kernel. Linux has to believe that the core is

unavailable, otherwise it could interfere with the Rust kernel. There is a mechanism to turn CPUs on or off at runtime in Linux called CPU hotplug. Linux will not interact with a turned off CPU and ignore it until it gets instructed to turn the core back on. Turning a CPU off with CPU hotplug does, among many other things, mark the CPU as offline and migrate tasks and IRQs away to other cores. So the idea is to use this turn off mechanism, without actually powering the CPU down. Linux will treat the core as offline, even though it is actually still running.

Apart from making Linux ignore the core, we also need to get the Rust kernel to execute on the core. At first we intended to spawn a kernel thread which will do the jump into the kernel image. To ensure the kernel thread was in control of the CPU at the time of the jump, we copied and altered code from the core shutdown process, which prevented a full core shutdown (see Section 4.4.3 for more details). So in the end we instead made the idle thread perform the jump, which allows us to do a clean core shutdown.

4.4.1 Boot Process

Linux running on arm64 uses the Power State Coordination Interface (PSCI) to shutdown a core. Two PSCI functions are used: CPU_OFF to power the calling CPU off and AFFINITY_INFO to verify that the CPU was actually turned off. Linux calls CPU_OFF on the idle thread of the target CPU, this should be the last statement to execute on this CPU, Linux expects this function call to never return. The function call is done through a function pointer to CPU_OFF stored in a struct, so we can replace the actual CPU_OFF function with our own function. This allows us to prevent the call to the PSCI and thus prevent the shutdown of the core. Additionally this also gives us the perfect opportunity to jump into the kernel image. Linux does not expect this function to return anyways and performed all statements it wants to perform on the core beforehand. Since we execute the jump on the idle thread instead of a kernel thread, there will not be a never exiting kernel thread leaving a task struct in the Linux kernel. The idle task is expected to be stopped by the CPU shutdown, its task struct would be reused for the new idle thread when the CPU is turned on again. The remainder of the CPU shutdown process takes place on another CPU. At some point it will use the PSCI function AFFINITY_INFO to check whether the target CPU was actually powered off. Again this function call is done through a function pointer, which we again replace with our own function that always reports a successfully powered off CPU. From there on Linux does the remaining part of shutdown process.

To summarize what we do, we replace the two functions CPU_OFF and AFFINITY_INFO with our own functions, then instruct Linux to shut down the target core. Afterwards we restore the original functions again. These relatively minor adjustments allow Linux to perform the complete shutdown process of a core and make that core execute a binary. Since we do not touch most of the shutdown process, this solution should be quite robust to changes made to it. As we instruct Linux to shut down the target core, it marks the

core as offline. The kernel module only allows to boot on cores that are marked online. Instructing Linux to turn on a core that is marked offline because we booted on it earlier will fail, as the core did not actually shut down and is still executing. If however the core did shut down because the Rust kernel called the CPU_OFF function itself, then Linux will be able to turn the core on again. It will be marked as online and can be used for future boots.

There is however an unclean part to this implementation, the struct that stores the PSCI function pointers cannot be directly accessed from a kernel module. We circumvent this restriction by using the function `kallsyms_lookup_name()` which returns the address of a symbol. While this method does allow us to access the struct, it is a bit of a hack, especially since the function is not exported to kernel modules from Linux 5.7 on. But even in newer Linux Kernel versions it is apparently possible to use kprobe to get access to `kallsyms_lookup_name()` from a kernel module.But it seems to be necessary to use `kallsyms_lookup_name()` since almost no functions and fields used for CPU hot-plugging are exported to kernel modules.

4.4.2 Verification

We tested the implementation by making it launch a small assembly program on the new CPU. It stores a value at a memory location and we observed the change from a kernel thread on Linux. While this observation shows that the code we jump to does execute, it does not show whether there is any other code from Linux executing on that CPU. To verify that this is not the case, we wrote a QEMU TCG plugin that logs all statements executed on the target CPU. Running the kernel module in a QEMU VM with this plugin showed that the only instructions executed on the CPU are the ones of the assembly program we launched. So we know that the implementation successfully isolates a core from Linux.

4.4.3 Old Implementation using a Kernel Thread

An earlier implementation jumps to the Rust kernel from a kernel thread instead of the idle thread. The kernel thread is spawned and bound to the target core. The kernel module runs on a different core and starts the CPU hotplug shutdown process. It follows the process up to the point where Linux would normally invoke the core local stopper thread to perform some shutdown tasks, one of them being migrating all threads in the runqueue to other cores. After the stopper thread is done, there are no other threads left on the CPU and the idle thread takes over and actually shuts the CPU down. So if we invoke the stopper thread, we will not be able to run the kernel thread we spawned in the beginning on the target core. So in order to pass control to the kernel thread, we do not invoke the stopper thread and instead let the kernel thread do the work of the stopper thread. After the kernel thread is done with that, it turns off the local IRQs and jumps into the Rust kernel. This prevents the idle thread from being run and shutting down the CPU.

This implementation has some downsides compared to the one we use, described in Section 4.4.1. We copied chunks of Linux kernel code in order to change the behavior, which results in code duplication and makes the kernel module harder to maintain, since every change in the CPU shutdown process of the kernel code would need to be copied into the kernel module. It also results in 15 function calls to functions not accessible to kernel modules. These functions and one struct have to be looked up with the `kallsyms_lookup_name()` function. This is additionally unstable since some of these functions are static, so if the kernel compiler decides to inline any one of them, then it cannot be found by `kallsyms_lookup_name()`. This implementation also deviates more from the actual shutdown process than the implementation we use, as the stopper and the idle thread are never invoked. As a consequence, we could not do the complete CPU shutdown process, so we stopped in an intermediate state. The kernel thread never correctly exits from Linux's point of view, which in turn prevented the kernel module from being unloaded and was maybe the reason why Linux printed warnings about the CPU being stalled by the kernel thread. This implementation does give more control than the implementation we use, as there are more places in the shutdown process where we could add code.

4.5 Boot Environment

We describe the boot environment of a freshly booted Rust kernel in this section. First we describe the general environment, then the virtual address space of the Rust kernel and finally how the boot environment can be modified in the Linux kernel module before control is handed over to the Rust kernel.

The boot environment after passing control to the Rust kernel is in general similar to the Linux kernel environment, since the jump is executed from the context of a Linux idle thread. The exception level is the same as the one Linux is running in. Interrupts are disabled, but the exception vector table is the same as the one used by Linux. As a consequence, if the Rust kernel causes a fatal exception, like a null pointer exception, then the Linux kernel panics. This is not necessarily bad, as a Linux kernel panic prints useful debugging information, like the type of exception and the values of registers. To prevent the Linux kernel from panicking and to handle exceptions itself, the Rust kernel will need to set up its own exception vector table. The MMU is enabled, with a new virtual address space created by the Linux kernel module. Caches are enabled as well. The stack pointer points to the kernel stack of the former idle thread. This stack is not used anymore since the Rust kernel overtook the idle thread during boot (see Section 4.4.1). However, since the stack is located in Linux kernel memory, the Rust kernel should refrain from using it and set up a proper stack in its own memory.

4.5.1 Virtual Address Space

The Rust kernel runs in its own virtual address space set up by the Linux kernel module. This virtual address space consists of identity mappings, so

each virtual address has the same value as the physical address it maps to. All memory assigned to the Rust kernel is mapped in this address space. The page tables are allocated by Linux using the `pte_alloc_map()` and friends functions (see `map_vaddr_to_page()` at `kernel-module/module_main.c`). These functions allocate memory from the buddy allocator with the `alloc_page()` function. So the page tables are in memory managed by the Linux Kernel, the kernel module does not track their location so the Rust kernel is not expected to access or modify the page tables. We think that this restriction is fine, a simple kernel can use the mapped pages without bothering with paging at all, whereas a more advanced kernel sets up its own page tables and discard the page tables set up by Linux when it is done. Alternatively, if the need arises to access the page tables allocated by Linux, one could rewrite the `pte_alloc_map()` functions to use memory managed by the kernel module and assign it to the kernel kernel.

The kernel module installs the page tables created by Linux in the TTBR0 register, so the address space is in the lower part of virtual memory usually used by user space applications. TTBR1 holds the value of the Linux kernel page table, so the upper part of virtual memory maps to Linux kernel memory. The reason for putting the Rust kernel's address space into TTBR0 is to make the jump into the Rust kernel code easier. Just before performing the jump, the kernel module sets TTBR0 to the new page table. The kernel module is running in Linux kernel virtual memory, so setting TTBR0 does not affect it. If we would instead change the value of TTBR1, then we would have to make sure that the currently executing instructions are mapped in the new address space as well, otherwise the program counter would suddenly point at a completely different location in memory, depending on where the new address space maps the virtual address. A kernel that sets up its own page tables can use the same approach the other way around: it prepares the page tables while running in the address space set up by the Linux kernel module. When it is done, it can set the TTBR1 register without affecting itself, since it is still running code mapped in TTBR0 page tables. Then it can jump to code mapped in TTBR1 page tables and stop using the old address space. However, this only works if the Rust kernel does not use any memory of the Linux kernel anymore. So it has to set up its own stack and exception vector in its memory.

4.5.2 Modifying the Boot Environment

Different kernels require different boot environments, so we cannot accommodate all of them. In this section we describe how the boot environment can be modified to suit a specific kernel. Such modifications could for example be setting registers to an address to pass boot parameters, setting up an exception vector, changing the configuration of timers or disabling the MMU. There are two places in the kernel module's code that are interesting for modifying the boot environment: `my_cpu_off()` and `boot_kernel_image()`.

The function `my_cpu_off()` performs the jump into the Rust kernel. It is called during the CPU hotplug shutdown process on the CPU on which the kernel will boot, in the context of the CPU's idle thread. So in this function

one could modify the state of the boot CPU just before handing control to the Rust kernel, like setting registers to certain values or disabling the MMU.

The other function `boot_kernel_image()` initiates the CPU hotplug shutdown process. It can execute on any of the CPUs that are currently marked as online in Linux. The Rust kernel's virtual address space is initialized at this point. While this function is less useful than the previous one for modifying the boot environment, it could be used for more general changes like modifying the Rust kernel's virtual address space.

4.6 Accessing Devices

After a successful boot, the Rust kernel will want to use devices present in the system. If the Rust kernel has a driver for a device, then it can use the device directly. But if it does not, then the driver would have to be implemented. Linux already has such a driver, so instead of implementing the driver again, the Rust kernel could use the device through Linux. By allowing the Rust kernel to use devices through Linux, it can use all devices that are supported by Linux. We decided to implement such a mechanism with virtio. We use virtio because there is no need for us to implement a similar mechanism from scratch and because there is a high potential for code reuse, since virtio is a known standard. As described in Section 2.3, the virtio driver and the virtio device are independent of the physical device. So by adding the generic virtio driver and virtio device for a device type, all physical devices of that type can be used. This makes the Rust kernel much more portable. As an additional benefit, virtio allows to share a device and use it from both the Rust kernel and Linux.

In this section, we will first briefly describe how a Rust kernel could access a device directly. The remainder of the section is dedicated to describe the virtio implementation that allows the Rust kernel to use a device through Linux.

4.6.1 Access a Device Directly

In this section, we describe how the Rust kernel could use a device directly if it has the corresponding device driver. This is faster than using the device through Linux, as the communication over virtio is not needed. However it is less portable than using virtio, the Rust kernel will need a driver for each device it should support.

If the Rust kernel can use a device directly with its own driver, then Linux has to be prevented from accessing the device. Drivers work under the assumption that they have exclusive access to a device. So if both Linux and the Rust kernel run a driver for one device, then they will interfere with each other. The device driver of Linux can be disabled by unloading its kernel module. Then the Rust kernel driver has exclusive access to the device and can operate it without interference. If the Rust kernel does not have a driver to operate a device, but it needs to use the device directly, then another option would be to simplify the implementation and leave the initialization to Linux. The device driver of Linux could be modified to not shut down the device when it is unloaded. Thus

the device stays in an initialized state, allowing a Rust kernel driver to skip initialization and operate the device straight away.

4.6.2 Virtio Overview

Virtio is usually used with a hypervisor host, running the virtio device, and a virtual machine guest, running the virtio driver. But we have two kernels running on the same machine on the same exception level, so we have to adapt virtio. In the following sections, we describe the shared memory region between virtio device and virtio driver and how MMIO and notifications are implemented. In the end we describe the changes that were necessary in order to support the virtio console.

There are already several virtio implementations, so we reduce the work by not implementing the standard from scratch, but rather reuse and adapt an existing implementation. For the virtio driver, running on the Rust kernel, we use the Rust virtio driver implementation [18] from the rCore OS project. For the virtio device, we use the hypervisor-less KVM tool from OpenAMP which we described in Section 2.3.2. Since we do not use virtualization, we will not refer to the two actors as *host* and *guest*, but instead refer to them by the name of the software that runs on them, namely *virtio device* and *virtio driver*.

The virtio device type we decided to implement is the virtio console. Having a console on the Rust kernel is useful for printing debugging information, especially in the early stages before a kernel can set up its own console. However, implementing a simple UART driver would be easier than implementing a virtio console. The virtio console has the advantage that implementing it once allows to use any UART device supported by Linux. In addition, the virtio console is more versatile, because it uses the virtio device's standard input (stdin) and standard output (stdout). This allows for example to use the virtio console over `ssh` or to redirect its output into a file.

4.6.3 Shared Memory

Virtio uses data structures that are shared between host and guest. Most of these data structures are allocated by the virtio driver. It can choose the location freely in its own memory, as the virtio device runs on the hypervisor and can access all of the guest's memory. The hypervisor-less KVM tool drops the assumption that the virtio device has full access to the virtio driver's memory. Instead it adds the concept of a shared memory region that is accessible by both. All shared data structures must be inside this shared memory region. We decided to keep this concept of shared memory, because it gives us a clean separation of memory shared between Linux and the Rust kernel and memory exclusively used by one of them. In regular virtio, the virtio driver can allocate data structures anywhere, for example on its stack, making it much harder to track which memory is shared and which is under exclusive control of the Rust kernel. In this section we describe this shared memory concept in more detail. This concept was created by the hypervisor-less KVM tool, we did not make any

changes to it. We only adapted the Rust virtio driver to support this concept as well.

The location and size of the shared memory region is set by the command line arguments of the hypervisor-less KVM tool. It maps the shared memory into its address space and allocates the requested devices. Figure 4 shows the memory layout of the shared memory region. Every device gets its own range inside the shared memory where all its data structures reside. The size of this range depends on the virtio device type, because the virtio device type determines how many virtqueues are used. At the beginning of each of these shared memory ranges, the hypervisor-less KVM tool initializes the MMIO registers. In order to pass the location and size of the shared memory range of the device to the virtio driver, it is written in four of these MMIO registers. These registers are not part of the virtio standard. They are located at addresses in the MMIO range that are currently not used by the virtio standard. The concrete layout of the MMIO registers used by the hypervisor-less KVM tool is defined in the header file `kvmtool/include/kvm/virtio-mmio.h` in `struct virtio_mmio_hdr`. The virtio driver sets up the number of virtqueues required by the virtio device type. As described in Section 2.3.1, the information that is passed through a virtqueue is contained in buffers. The virtio driver must place the virtqueues and their buffers inside the shared memory range of the device. It can decide where to place them inside the device's shared memory range, as long as they do not overlap with the MMIO registers.

The current implementation does not inform the virtio driver where the MMIO registers are located in memory. We currently just hard code the address of the virtio console MMIO into the Rust kernel. One way to avoid that and pass the MMIO address in a more elegant way could be to use the device tree shown at the top of the shared memory region in Figure 4. This device tree is set up by the hypervisor-less KVM tool. We did not inspect this device tree, so we do not know what information it contains, but it could contain the address of the MMIO registers of the virtio devices.

4.6.4 MMIO

The virtio standard defines three transportation modes: MMIO, PCI and channel I/O (see Section 2.3.1). We use virtio over MMIO, such that the Rust kernel does not need support for PCI. Alternatively, virtio over channel I/O could be used, we decided to use virtio over MMIO since we found more existing implementations that use it. As we are dealing with virtual devices, they do not have real MMIO. Instead we use regular memory in the shared memory region to emulate MMIO. Since we are not in a virtualization context, an access to MMIO cannot trap and pass control to the virtio device running on a hypervisor. So we use notifications to inform the virtio device of MMIO accesses. We will first describe the implementation, then point out how we modified the hypervisor-less KVM tool and the Rust virtio driver.

The virtio driver sends a notification after an MMIO access. The virtio device can then review the change to the MMIO registers and update its state

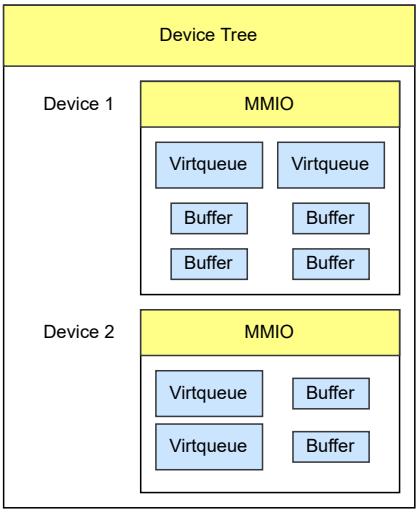

Figure 4: Memory layout of the shared memory region. Yellow items are set up by the hypervisor-less KVM tool, blue items by the virtio driver.

accordingly. To lower the number of notifications, we do not send one when reading the value of a register. As a consequence, the virtio device cannot react to an MMIO read, thus all MMIO registers must have the current correct value. So after every write to an MMIO register, the virtio driver sends a notification causing the virtio device to update its state and, if necessary, change the values of MMIO registers. There are cases where changing the value of one MMIO register requires the virtio device to change multiple other MMIO registers. One of these cases is the *QueueSel* register, which, as described in Section 2.3.1, causes four other MMIO registers to change their values. So the virtio driver has to block on a write notification to give the virtio device enough time to update the MMIO registers. Otherwise the virtio driver could read the old value of an MMIO register before it was updated by the virtio device, leading to a race condition. To inform the virtio driver that the MMIO registers are updated and it can continue its execution, the virtio device sends a ready notification. Therefore our implementation of emulating MMIO requires two new notifications not part of the virtio standard: one to inform the virtio device of a write to MMIO and the other to release the virtio driver from blocking.

We will now describe the changes we made to the hypervisor-less KVM tool and the Rust virtio driver. The Rust virtio driver is written for regular virtio, so it does not contain any of the described measures. So we added sending a notification after writing into an MMIO register and waiting for a ready notification. The function that updates the MMIO registers was already implemented in the hypervisor-less KVM tool, we just made a minor change to it to update an MMIO register correctly. The hypervisor-less KVM tool calls this MMIO update function when it receives a message from the virtio driver,

Variable	Sender	Description
host_notify	driver	driver wrote to MMIO and it should be updated
host_ready	device	MMIO is updated, driver can proceed
guest_notify	device	virtio notification to driver
guest_msg	driver	driver debug message (64 bit number)
guest_msg_ack	device	debug message was read by notify device

Table 3: Description of the variables used for notifications by polling.

however it does not send a ready notification back, so we added that. We are not sure whether the hypervisor-less KVM tool implemented a blocking mechanism as we did, but we did not find an equivalent to a ready notification that we send.

4.6.5 Notifications

The virtio standard specifies that the virtio device sends notifications to the virtio driver. It uses a dedicated interrupt signal to do so. We have not setup the exception vector on the Rust kernel, so it cannot receive interrupts. Instead our implementation sends and receives notifications by polling on shared memory.

In our implementation, it is not just the virtio device that sends notifications, but also the virtio driver. So both virtio device and virtio driver poll to receive notifications. The Rust kernel performs the polling itself, on the Linux kernel it is performed by the kernel module via a character device called notify device. We implement polling with a character device because the hypervisor-less KVM tool does so as well. As described in Section 2.3.2, the hypervisor-less KVM tool performs system calls on the character device file to send and receive notification. We cannot reuse the character device of the hypervisor-less KVM tool because it is platform dependent. But by implementing our own character device and using the same interface, we do not need to make changes in the hypervisor-less KVM tool for sending and receiving notifications. Another advantage is that encapsulating the notification mechanism in a character device makes it easier to add a new notification mechanism in the future, like IPI.

There are five variables that we use for polling, defined in the header file `kernel-module/notify_dev.h` in `struct notification_shared_mem`. They are described in Table 3. Each variable is exclusively written to by either the virtio device or the virtio driver, the other polls on it. To send a notification, the sender increments the variable. The receiver polls on the variable and reacts to a change of its value.

On the Linux kernel side, the notify device sends a notification on a write system call. If the count parameter of the write system call is 10, it increments the `host_ready` variable, otherwise it increments the `guest_notify` variable. We chose the value of 10 arbitrarily, we just needed a simple way to differentiate these two kinds of notifications. To poll on the `host_notify` and the `guest_msg` variables, the notify device uses a kernel thread. This kernel thread is spawned

when the notify device file is opened and stopped when the file is closed. The notify device file can only be open once at a time. The hypervisor-less KVM tool uses a thread to make a poll system call on the notify device file. The notify device puts the thread to sleep and wakes it up if there is a change in the `host_notify` variable. To prevent missing a notification, the notify device counts the number of received notifications. This number is increased when receiving a notification and decreased when waking up a thread that made a poll system call. So if there is currently no sleeping thread, then the next thread making a poll system call will not sleep but return immediately. The `guest_msg` variable is intended for debugging messages, a change of this variable causes the new value to be printed in the kernel log.

On the Rust kernel, the kernel has to poll for changes by itself, if it supports threads then it can poll from a thread as well, otherwise it has to use its main thread to do so. Bare Rust for example starts polling to receive console input with its main thread after printing a message to the virtio console. As described in Section 4.6.4, the virtio driver needs to block after writing to MMIO until the virtio device is done with updating the MMIO. So after a write to an MMIO register, the virtio driver increments the `host_notify` variable. Then it has to poll on the `host_ready` variable.

The poll memory is shared between Linux and the Rust kernel. However it is not in the shared memory region described in Section 4.6.3. The reason for not putting the poll memory into the shared memory region is that we wanted to isolate the polling notification mechanism from the hypervisor-less KVM tool. It should be possible to implement a different notification mechanism without needing any changes in the hypervisor-less KVM. If it was embedded in the shared memory region, then the hypervisor-less KVM tool would need to allocate it, thus depending on the notification mechanism.

4.6.6 Implementing the Virtio Console

In this section we describe the virtio console specific changes we had to make in order to support it. We made changes to the hypervisor-less KVM tool and the Rust virtio driver. We expect that these changes will have to be applied again when adding support for another virtio device type, so we describe them here to simplify the process.

Hypervisor-less KVM tool According to the virtio standard, during initialization of a virtqueue the driver can choose the size of the virtqueue, as long as it is smaller than the maximum size chosen by the device. However, the original hypervisor-less KVM tool implementation ignores any virtqueue size chosen by the virtio driver and uses a hard-coded size. The size of a virtqueue defines its memory layout, so they have to match in order for the virtqueue to work properly. One could either make the driver choose the size hard-coded in the hypervisor-less KVM tool or change the hypervisor-less KVM tool to use the size chosen by the driver. We used the second approach for the virtio console device since it follows the virtio standard more closely.

The function `virtio_mmio_notification_out()` updates the emulated MMIO registers in shared memory. The original hypervisor-less KVM tool implementation did not update the *QueuePFN* register correctly when a new queue was selected with the *QueueSel* register. So we modified the function to update the register properly.

As described in Section 2.3.1, each virtqueue is identified by a virtqueue number. The `get_vq()` function takes such a virtqueue number and returns the struct of the corresponding virtqueue. Due to changes we made, this function can be called with an invalid queue number, which is not handled in the original implementation and causes an array access outside of the array's bounds. We decided to change the `get_vq` function and check whether the virtqueue number is valid and return NULL if it is not.

Rust virtio driver As described in Section 4.6.3, we specified that the memory shared between virtio device and virtio driver must be in the shared memory region. Thus the virtio driver must allocate all shared data structures inside the shared memory region. We added the `SharedMemory` struct to the Rust virtio driver, which hands out pages in the shared memory region. It is initialized by the `VirtIOHeader.shared_memory()` function. So one can use the `SharedMemory` struct to allocate the virtqueues and their buffers.

4.7 Kernel Module Interface

The kernel module is designed to be used with an user space program. So it needs a kernel module interface that allows a user space program to pass control to the kernel module to perform actions that are not possible from user space, like mapping memory or booting a Rust kernel. Linux provides several ways to implement such an interface, so we briefly describe the options and then motivate why we chose to use a `procfs` based interface. Adding a new system call is not an option since we do not want to modify the kernel outside of the kernel module.

procfs A virtual file system usually mounted at `/proc`. A kernel module can create directories and files in `procfs` and define file operations for those files. A user space program can make system calls on these files which cause the corresponding file operation to be executed. `Procfs` allows to use general file operations, this makes it flexible by allowing to use special operations like ioctl and mmap.

sysfs and configfs Virtual file systems, usually mounted at `/sys` and `/config`. Both allow to show information and modify kernel objects. They complement each other, `sysfs` is used for kernel objects created by the kernel and `configfs` is used to allow user space to create and delete kernel objects. Files in these file systems represent attributes of objects, each file should only contain one value. User space can make read and write system calls on these files

to interact with them, other operations are not allowed. In `configfs` objects can be created with mkdir and rmdir.

netlink A socket based interface that allows to send messages between kernel and user space processes. A user space program uses the interface like a regular socket, a kernel module uses an internal kernel API.

We decided to make a `procfs` based interface. The main reason being that it is easier to set up and use than the other two options. It is also more flexible by allowing ioctl and mmap. The downside is that `procfs` should be used for process related information, it is encouraged to use `sysfs` for other information. An alternative would be to use a combination of `sysfs` and `configfs`. Using `sysfs` on its own would not work well because it is the user space program that initiates the creation of objects like a memory range. However using only `configfs` would make it hard for the kernel module to create objects visible to the user, for example creating memory ranges when loading an ELF executable. So we would need to use both `sysfs` and `configfs`, spreading the files to two different locations. A major downside is that neither of these file systems support mmap, so to access memory the user would have to perform a read or write system call on every access. Using mmap allows to map the memory into the user's virtual address space, so no further system calls are necessary to access the memory. `Netlink` would be similarly flexible as `procfs` with ioctl, however it is much more work to set up, one reason being that it would require us to define a message format.

4.7.1 Kernel Instances

The kernel interface should allow to setup and boot more than one kernel. To manage multiple kernels, the kernel module separates each kernel into a kernel instance. The user can create and perform actions on kernel instances. Each kernel instance represents one Rust kernel and has its own resources assigned. These resources are not allowed to overlap with other kernel instance's resources. The resources a kernel instance currently tracks is assigned memory and notify device. If the kernel instance is deleted, the assigned resources are released. Currently the only way to delete a kernel instance is to unload the kernel module, which causes the deletion of all kernel instances. One could add a ioctl to delete a single kernel instance in the future. Releasing a kernel instance's assigned memory can be dangerous, as the booted kernel could still be using that memory. The problem occurs when the memory gets assigned again, which sets it to zero. So if the old kernel still uses that memory, a null pointer or an invalid instruction exception will likely occur. If the Rust kernel did not replace the exception vector, then such an exception will cause a Linux kernel panic. To be safe, the Rust kernel has to stop execution by calling the PSCI function CPU_OFF before releasing memory. A convenient place to call CPU_OFF is inside the Rust kernel's panic handler, which needs to be defined in a bare metal Rust

program. CPU_OFF has to be called on the CPU to be turned of, so it has to be called by the Rust kernel.

5 Kernel Module Interface

The kernel module provides the functionality to boot a kernel. This section describes the interface through which a user space program can access said functionality. The userspace program interacts with the kernel module through files in the procfs file system. The kernel module creates these files and directories in `/proc/bootloader_lkm/`. For every kernel that should be booted, the user creates a kernel instance. The user can then instruct the kernel module to perform actions on this kernel instance, like adding memory to it, loading a kernel or booting the kernel. For each kernel instance, the kernel creates a directory, named after the instance, in `/proc/bootloader_lkm/`. This instance directory will contain the files relevant for this instance.

The kernel module interface can be divided into two parts: the general interface (Section 5.2) and the instance interface (Section 5.3). The general interface is independent of any kernel instance and controls the kernel module overall. The instance interface controls a specific kernel instance.

5.1 Ioctl

Both the general interface and the instance interface are mainly controlled through ioctl system calls. An ioctl is registered and identified by a command number, which in Linux is generated from three constants [19]. A user space program needs to use the same constants to end up with the same command number. The constants of each ioctl the kernel module supports are listed in the header file `kernel-module/procfs.h`. This header file also contains the definitions of the structs that are passed as arguments with the ioctl. We prefixed some fields of these structs with `"ret_"`. They are intended to be return values from the kernel module and get overwritten by it if the ioctl is successful. If an ioctl fails, then it could help to check the kernel log for messages printed by the kernel module. They are often more useful than the error code returned by an ioctl.

5.2 General Interface

The general interface allows to control the overall state of the kernel module. This can happen in two ways: passing parameters to the kernel module or making ioctl system calls. Ioctl system calls for the general interface target file `/proc/bootloader_lkm/module_control`, which is created when the kernel module is loaded. The general interface currently supports one kernel module parameter and one ioctl:

Parameter: Memory This parameter allows to specify the memory ranges reserved for the kernel module. It takes an array of pairs of memory addresses, the first address specifying the start and the second address specifying the end of a memory range. For example, if one wants to pass the two memory ranges `start1-end1` and `start2-end2`, the kernel module parameter would look like

this: `memory=start1,end1,start2,end2`. These addresses are physical addresses and must be page aligned, the end address points to the last byte in the memory range. Linux has to be restricted from accessing the memory ranges passed with this parameter. Section 4.1 describes how to use a kernel parameter to restrict Linux from using the entire memory. The memory ranges currently usable by Linux are the memory ranges named System RAM in file `/proc/iomem`. By booting Linux twice, once normally and once with memory restriction, and comparing the content of `/proc/iomem`, one can find out which memory ranges disappear in the memory restricted case. These memory ranges are the ones not accessible to Linux and can be passed to the kernel module.

Ioctl: Create Instance The ioctl *IOCTL_ROOT_CREATE_INSTANCE* creates a new kernel instance. The user has to provide the name of the new kernel instance and the length of the name. The maximum length is 255. The kernel module will create the instance directory in `/proc/bootloader_lkm/`, containing one file called `instance_control`. This file allows to control the new kernel instance, see the following Section 5.3. Creating two kernel instances with the same name will fail.

5.3 Instance Interface

The instance interface allows to control a specific kernel instance. Every kernel instance has a instance directory in `/proc/bootloader_lkm/`, this instance directory contains a file called `instance_control`. An instance can be controlled by calling ioctl system calls on this file. The following ioctls are available:

Request Memory The ioctl *IOCTL_INSTANCE_MEMORY_REQUEST* requests a memory range and assigns it to this kernel instance. All memory successfully requested will be mapped into the virtual address space of the kernel of this instance. The call requires three arguments: the start and the end address of the memory range and the memory protection. The start and end address are physical addresses. Since the virtual address space is a identity map, they will correspond to the instance's kernel's virtual addresses. The end address points to the last byte included in the memory range. The start and end address do not need to be page aligned. The memory protection determines the permission by which the memory range will be available in the new kernel. They are described by macros defined in <sys/mman.h>, the same macros that are used by the mmap system call. The permission combinations accepted by the kernel module are listed in Table 2. Note that even though the interface allows to choose a read-only protection, writes to such memory are not prevented.

The memory request will fail if the requested memory range is not inside one of the memory regions available to the kernel module. It will also fail if it overlaps with any of the previously requested memory ranges, including those requested by different kernel instances. If the memory request is successful, the kernel modules creates a file representing the new memory range in the instance

directory and returns the memory number. The name of this file will start with "mem-" followed by the memory number, so the user can append memory number to recreate the file name. Reading the file will print the start and end address of the memory range. If the user wants to write into the memory range, they can make an mmap system call on the file to map the memory range into the user space program's virtual address space. Writing into the memory range can be used to initialize data for the instance's kernel before boot. But it can also be used as shared memory to communicate with the instance's kernel after boot. The hypervisor-less KVM tool for example does this to make a shared memory range where the data structures of virtio are accessed and updated by itself and the instance's kernel. The mmap call has to be made with the flag *MAP_SHARED*. The user must always mmap the entire memory range. The kernel module does not track mmapped memory, so it does not ensure that it was unmapped when a kernel instance is released (currently the only way to release an kernel instance is to reload the kernel module). The user must not use mmapped memory after a kernel instance is released.

Add Notify Device The ioctl *IOCTL_INSTANCE_ADD_NOTIFY_DEV* creates a new notify device and adds it to this kernel instance. A notify device is a character device that sends and receives notifications in our virtio implementation (see Section 4.6.5). The ioctl takes the start address of the polling memory as argument. The kernel module will request one page of memory at that address. The notify device will use the beginning of this page for the struct `struct notification_shared_mem` which contains the polling variables. On success, the ioctl returns the minor device number of the newly created device. The character device file, named "notifydev-" followed by the minor device number, is located in `/dev/`.

Load ELF Executable The ioctl *IOCTL_INSTANCE_LOAD_ELF* loads an ELF executable for this kernel instance. It takes a file descriptor of an ELF executable as argument and returns, on success, the entry point of the executable. The kernel module automatically requests the required memory ranges and copies the segments of the ELF executable there. The ioctl will fail if any of the memory ranges are already assigned to this or another kernel instance. The addresses of the memory ranges are defined in the ELF executable. So if these addresses are not available, they need to be changed in the ELF executable (see appendix). If the ioctl fails, earlier successful memory requests will not be released.

Boot Kernel The ioctl *IOCTL_INSTANCE_BOOT* boots this kernel instance on a core. It takes the CPU and the kernel entry point as an argument. The kernel module checks whether the entry point lies within one of the memory ranges assigned to the kernel instance, if it is not then the call fails. If the specified CPU is not marked as online in Linux (see `/sys/devices/system/cpu/online`), then the call will fail as well. Before the kernel instance is booted,

its virtual address space is created and all its assigned memory gets mapped into it.

Usually this ioctl will be called last on a kernel instance. The other ioctls set up the kernel instance and this ioctl boots it. But this does not have to be the case, it is possible to make these ioctl calls when the instance kernel is already running. However, memory that is requested after this ioctl will not be mapped into the running kernel's virtual address space. The kernel module does assign the memory, so it is possible to add memory to a kernel after boot. The user will likely need a way to inform the running kernel instance of the newly added memory and the kernel instance will need to map it. The other two ioctls, *IOCTL_INSTANCE_ADD_NOTIFY_DEV* and *IOCTL_-INSTANCE_LOAD_ELF*, request memory as well, so the same applies to them. Calling another *IOCTL_INSTANCE_BOOT* ioctl on an instance can be used to add an additional CPU to a kernel instance. The kernel module will create the virtual address space again for the new CPU. Successful memory requests made after the boot of the first CPU but before the boot of the second CPU will be mapped in the virtual address space of the second CPU.

6 Evaluation

In this section, we evaluate the implementation of this boot loader. This boot loader was used on two different systems so far, a ThunderX Cavium machine and a QEMU virtual machine. All evaluations described in this section were performed on the ThunderX. An overview on the hardware of the ThunderX is shown in table 4. Each evaluation is performed with a different version of Bare Rust, our simple Rust kernel. These modified versions can be found on different branches on the git repository of this project. The `README.md` file describes which branch belongs to which evaluation. The branch specific `README.md` describes how the experiment can be launched. In this section, when we refer to the *Rust kernel*, then we refer to the version of Bare Rust belonging to the current experiment.

We discuss the following evaluations in this section. The first three experiments evaluate the functionality of this boot loader: performing a regular boot, booting Rust kernels consecutively and accessing the UART directly from a Rust kernel. The last two experiments measure the amount of memory that Linux requires and compares the virtio console to the UART in terms of printing throughput.

6.1 Basic Boot

In this experiment, we evaluate whether this boot loader can boot a single Rust kernel. In addition, we want to test whether the Rust kernel can use the virtio console to delegate console I/O to Linux.

The boot loader loads the target Rust kernel and boots it. To signal a successful boot, the Rust kernel sends a magic number to the notify device by writing it to the `guest_msg` variable (see Section 4.6.5). The notify device prints the value of this variable into the kernel log whenever it changes. Afterwards, the Rust kernel sends a test string over the virtio console, then it spin loops on the virtio console to receive characters. From Linux, the user observes and enters characters into the virtio console. For every character the Rust kernel receives, it sends the character incremented by one back over the virtio console.

After running the boot loader, the magic number appears in the kernel log. Then, the test string appears in the virtio console. For every character the user enters, the same character incremented by one appears in the virtio console.

ThunderX Cavium CN8890	
CPU	Cavium CN8890
Micro Architecture	ARMv8
CPU Sockets	2
Cores	48
Memory	500 GiB

Table 4: Hardware overview of the evaluation machine.

The Rust kernel boots successfully, as indicated by the magic number. Delegating console I/O to Linux works as well, both for printing and for receiving characters.

6.2 Consecutive Boot

This boot loader does not stop its execution after a boot, so generally it outlives the booted Rust kernel. After a Rust kernel terminates, it would be convenient if the boot loader could boot another Rust kernel without having to reboot the machine. This would be especially convenient during development of a Rust kernel, where Rust kernel reboots will be common. So the goal of this experiment is to evaluate whether multiple Rust kernels can be booted consecutively.

We perform four consecutive boots. The Rust kernel prints a test string over virtio. For each boot, we modify the test string being printed. This allows us to differentiate the Rust kernel instances and make sure that the next Rust was correctly loaded. Since these Rust kernels use the same memory ranges, the kernel module will only grant the memory requests of the first Rust kernel and deny the later requests, since the memory is already assigned. To allow requesting the same memory range multiple times, we reload the kernel module. We always reload the kernel module after the current Rust kernel prints its text string and before starting the next Rust kernel. We perform this experiment in two variants:

1. After printing the test string, the Rust kernel terminates by calling the PSCI function CPU_OFF.

2. The Rust kernel does not terminate and prints the test string in an infinite loop.

In the first variant, the first Rust kernel booted prints its test string. The next Rust kernel boots successfully and prints its own test string. The remaining boots are successful as well. In the second variant, the first Rust kernel booted prints its test string repeatedly. Booting the second Rust kernel however results in a Linux kernel panic. This kernel panic is caused by a invalid instruction exception.

If the Rust kernel properly terminates and shuts down its core, then consecutive boots succeed. If the Rust kernel is still executing, then consecutive boots fail. We think that the invalid instruction exception is caused by the memory requests for the second Rust kernel. When the kernel module assigns a memory range to a Rust kernel, it overwrites the range with zeros. Since the first and second Rust kernel use the same memory ranges, all of that memory gets overwritten with zeros, including the kernel binary. As the first Rust kernel still executes, it reads an invalid instruction and causes the exception. So to consecutively boot Rust kernels that use overlapping memory ranges, it is important that the previous Rust kernel stops its execution before booting the next Rust kernel.

Being able to boot Rust kernels consecutively is convenient, since it allows to perform the development of the Rust kernel on the same machine it boots on. Since this boot loader is just a Linux system with an additional kernel module, it is possible to use any software available on Linux. So one can modify the Rust kernel code with an editor, compile it and boot it on one machine.

6.3 Access UART

The goal of this experiment is to see whether the Rust kernel can print to a simple UART device that was initialized by Linux. The Rust kernel should not need to perform any initialization itself, but be able to use it directly.

As the UART is already configured by Linux, we only need the parts for printing of an UART driver for the Rust kernel. Our test machine has an *ARM PL011* compatible UART. We copy the driver parts for printing from the *PL011* UART driver from Barrelfish [21]. The experiment is performed in the following way: We request the memory page that holds the required registers of the UART, then we boot the Rust kernel which attempts to print a test string to the UART. We observe the output of the UART.

The memory request for the page holding the UART registers fails, the Linux kernel prevents a write to read protected memory and stops our kernel module. We modify the kernel module to not overwrite requested memory with zero. After the modification, the memory request is successful and the test string appears on the UART.

We can see that mapping device memory with our kernel module works. However, this is only the case if the memory is not overwritten with zeros. Overwriting arbitrary device registers with zero is not desirable. Doing so is either prevented by Linux, as it is in this experiment, or it can cause undesired device behavior, depending on the register. We disabled overwriting assigned memory with zeros as a temporary fix in this experiment, however this affects all assigned memory, not just the memory containing device registers. A better solution would be to adjust the kernel module interface and allow the user to specify whether a requested memory range should be overwritten with zeros or not. Apart from that, this experiment shows that it is possible to use a device initialized by Linux, at least for a simple device like a UART. We did not restrict Linux from using the UART, which worked fine in this case, because Linux was waiting for input on the UART and thus did not print to it. For other devices, Linux will need to be restricted from using it, in order to not interfere with the Rust kernel. We described in Section 4.6.1 how Linux could be restricted from using a device.

6.4 Linux Memory Usage

After successfully booting a Rust kernel, Linux keeps running alongside it. As a consequence, it uses some amount of memory that is not available to the Rust kernel anymore. The goal of this experiment is to find out what the minimum

amount of memory is that must be reserved for Linux such that it successfully boots a Rust kernel.

To find the minimum amount of memory necessary, we boot Linux with different amounts of reserved memory. The amount of reserved memory is set with the `mem` kernel parameter. Booting the Rust kernel with an amount of reserved memory is successful, if Linux itself boots successfully and then the Rust kernel boots and runs successfully. The memory Linux uses is not independent of the Rust kernel. Linux creates a new address space for the Rust kernel and maps all its memory into it. The page tables of this new address space are allocated inside the memory of Linux. So Linux's memory usage increases with the amount of memory assigned to the Rust kernel. Using the virtio console increases Linux's memory usage as well, since it requires Linux to run the hypervisor-less KVM tool. The Rust kernel of this experiment gets 34 MB of memory assigned and uses the virtio console. We decided to include the virtio console in this evaluation, because if there is no interaction between Linux and the Rust kernel, then there is no advantage in keeping Linux running. Apart from what we explicitly run, the amount of memory that Linux requires also depends on its configuration. While it would be possible to reduce the memory by using a minimal Linux configuration, we want to evaluate a more typical configuration. We use an Ubuntu 20.04.2 installation with the Linux kernel version 5.4.94.

The lowest amount of reserved memory for which Linux consistently boots the Rust kernel successfully is 513 MB. Reserving less memory often results in a kernel crash during the Linux boot process. But sometimes it boots successfully.

Linux runs out of memory and fails to boot before the Rust kernel does. So a Rust kernel with only a small amount of memory assigned like in this experiment boots successfully as long as Linux boots successfully. We are not sure why Linux sometimes manages to boot with less than 513 MB reserved memory, there seems to be some variability in the amount of memory Linux allocates during boot. But a bootloader that only sometimes boots successfully is not useful, so restricting the reserved memory for Linux that much should be avoided.

6.5 Virtio Console Overhead

The Rust kernel can use the virtio console to delegate console I/O to Linux. The goal of this experiment is to determine the overhead of using the virtio console compared to using the UART directly. We compare their performance in terms of writing throughput.

In this experiment, we measure the throughput of printing characters in two different scenarios: the Rust kernel using the virtio console to delegate printing to Linux and the Rust kernel using the UART to print. In the virtio console scenario, the actual printing is performed by the hypervisor-less KVM tool, which prints to its stdout. We do not redirect the hypervisor-less KVM tool's stdout to the UART, as this would just put an upper bound on the throughput of the virtio console. We measure the throughput by measuring the time it takes to print a long test string. The test string we chose is a 20 000 characters

long version of the Lorem Ipsum placeholder text. Both the virtio console and the UART print a string character by character. The virtio console puts the character in a buffer and sends it through the virtqueue. We measure the time from issuing the first print statement until issuing the last print statement. It might seem odd that we do not measure the time until the last print statement finishes. The reason is that in order to determine when the last character is printed on the UART, we would need to track its output. But from there we do not have direct access to the system time, which makes it harder to measure the time correctly. This is also the reason why we measure the throughput but not the latency of the console. So while the last character is not actually printed in the time measured, it only has a negligible impact on the throughput, due to the high number of characters printed. To measure the system time, we use the system counter, which is specified in the Arm Architecture Reference Manual[20] and which provides a uniform view of system time across cores. We read the system counter frequency in Hz from the `CNTFRQ_EL0` register. We divide the value read from the system counter by `CNTFRQ_EL0` $\cdot 10^{-6}$ to get the time in µs. To measure the throughput of printing to the UART, we need an UART driver. Since Linux already configured the UART, we can just copy the parts for printing of an existing UART driver. We copy these parts from the PL011 UART driver from Barrelfish [21]. We hard code the memory locations of the UART registers and the system counter, they have to be changed to run the experiment on a different machine. To map these locations into the Rust kernel's address space, we request them from the kernel module. This only works if we modify the kernel module to not fill requested memory with zero. Writing zero to a page containing registers is either prevented by Linux or causes undesired behavior, depending on the register. During the experiment, while the Rust kernel prints 20 000 characters to the UART, the UART driver on Linux is still running. It does not interfere with the experiment, because Linux is waiting for user input.

The virtio console throughput measurements are plotted in a histogram in Figure 5. We performed the virtio console measurement 1800 times. There are two accumulations in this histogram, the accumulation with the lower throughput is more frequent than the other. The UART throughput measurements are in the range from 11 521.56 B/s to 11 521.57 B/s. We performed the UART measurement 200 times. The first measurement of each series of consecutive measurements has a higher throughput. We remove them from the result as we account them to the warm-up phase of the UART. since the UART throughput is so stable compared to the virtio console throughput, we indicate its average in the histogram with a red vertical line.

The throughput of the virtio console is most of the time higher than the throughput of the UART. Passing the characters between cores through shared memory seems to be quicker than communicating with the UART device. So delegating the console to Linux does not result in a performance penalty in terms of throughput. We think that the two accumulations in the virtio console throughput histogram are caused by the scheduling of the Linux kernel thread that polls for virtio notifications. This kernel thread is described in Section 4.6.5.

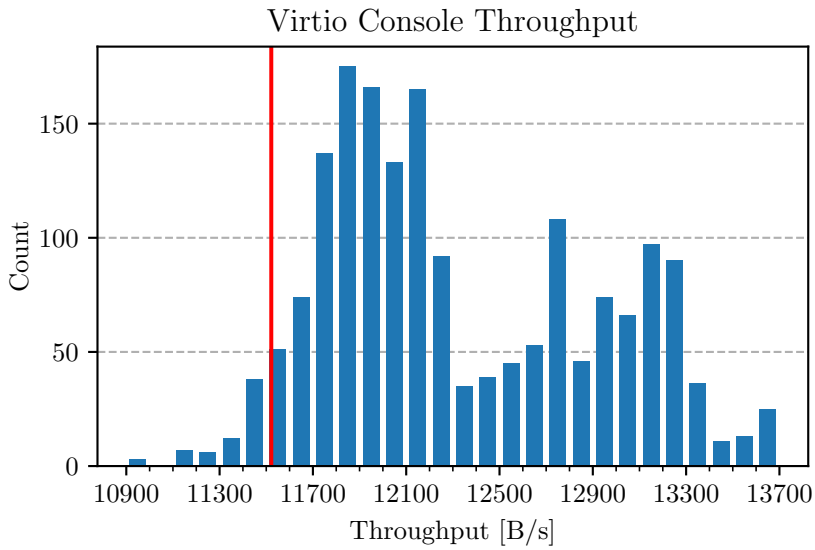

Figure 5: Histogram of the virtio console throughput. The red vertical line marks the average of the UART throughput.

How this kernel thread is scheduled influences the time a notification takes to reach the recipient. Since notifications from the Rust kernel are blocking, this can greatly influence the performance of the virtio console.

7 Future Work

There are several ways how this boot loader can be further developed and extended. In this section, we describe concrete ideas how this project could be improved. We will first discuss how Linux's support of the Rust kernel could be extended, followed by specific features that would improve the boot loader.

7.1 Linux Support after Boot

One of the main benefits of this boot loader is the support a Rust kernel can get from Linux after it was booted. This support is currently limited to the virtio console, so it would be great to extend it and offer more services. In this section we describe services that could be offered by Linux.

The easiest way to extend the support offered by Linux is to add support for more virtio device types, like network or block devices. Supporting a new virtio device type allows the Rust kernel to use devices of that type through Linux. The Rust kernel does not need drivers specific to the physical device, it just needs the generic virtio drivers corresponding to the device type. We expect that the changes necessary to support another device type are the same as the ones we made for the virtio console. These changes are described in Section 4.6.6, so we expect that adding support for another device type is straightforward.

Linux could offer other services to the Rust kernel apart from letting it use devices through it. But in order to let the Rust kernel request a service, it needs a means of communication with Linux. One possible solution is to use virtio again and implement a custom virtio driver and virtio device pair that exchanges messages. The virtio console could be used as template. The virtio device would, instead of printing a received message, take action based on it. If a message requires action from the kernel module, then the virtio device can trigger it through an ioctl. Alternatively one could implement a new communication channel between Linux and the Rust kernel. Depending on the communication channel, it could make sense to use and extend the notify device used by virtio (see Section 4.6.5).

A service that Linux could offer is to let the Rust kernel request additional CPUs. The Rust kernel would specify an entry point, and the Linux kernel boots an available CPU at that location. Alternatively the Rust kernel could be able to request more memory while it is running. Adding more CPUs or memory to a Rust kernel is already possible in the current implementation, but it has to be initiated from Linux with the corresponding IOCTL (see *Boot Kernel* in Section 5.3). Initiating such requests from the Rust kernel is more useful since it allows the Rust kernel to decide when it needs them and how much it needs. It would also allow to pass a response to the Rust kernel, for example the exact range of memory that was assigned or whether booting an additional CPU was successful.

Another service that Linux could offer is to let the Rust kernel use the file system through it. This is similar to implementing the virtio block device, but the virtio block device interacts directly with the block device and thus is not

aware of the file system. Being able to directly read or write files would be a convenient functionality for the Rust kernel to have. The Rust kernel would not need drivers to use the file system and could use any file system that Linux supports.

The last service we describe that Linux could offer is to manage the virtual address space of the Rust kernel. Currently the Linux kernel module maps all memory assigned to the Rust kernel into the virtual address space the Rust kernel boots with. Afterwards, it will not modify this address space in any way. If the Rust kernel wants to perform any changes to an address space or create a new one, then it needs to implement these operations itself. Linux could offer to do these operations for the Rust kernel. The Rust kernel could then for example request to map a range of pages at a virtual address in a certain address space. Another advantage of letting Linux manage memory mappings is that it can ensure to only map memory that is assigned to the Rust kernel. This would prevent accidentally mapping memory that is not assigned and thus not reserved for the Rust kernel. Since Linux manages the page tables, it would allocate them inside its own memory, like it currently does for the initial address space.

7.2 Setting up Registers

A way to pass boot arguments to a kernel is to set specific registers to certain values. Such a value could be a magic number or a number indicating whether the CPU is the bootstrap CPU or not. It could also be an address describing the location of a data structure. Such a data structure could for example contain boot parameters or a description of the hardware, like a device tree. Other possible use cases, specific to our boot loader, are to set a register to the address of the virtio console's MMIO registers (see Section 4.6.3) or to set the stack pointer (see Section 4.5). It would be convenient to have an ioctl that lets the user choose what value a specific register will have when the Rust kernel starts executing. If a user wants to pass a data structure to the Rust kernel, they could achieve that in the following way: request memory from the kernel module, use mmap to map it into user space, write the data structure into it and set the register to the address of the data structure. To implement this feature, the kernel module would have to set the registers to the specified values just before jumping into the Rust kernel image. In the current implementation, the user can manually set registers to certain values by modifying the kernel module. But we expect this to be a common use case, so it would be convenient to support it with an ioctl in the kernel module interface.

7.3 Exception Vector Table

In the current implementation, our boot loader does not set up an exception vector table for the Rust kernel. The Rust kernel is booted with the exception vector table used by Linux. So if the Rust kernel causes an exception, it is handled by Linux. Depending on the exception, this causes the Linux kernel

to panic. If the Linux kernel would not panic, then it could boot another Rust kernel without needing to reboot the machine. Currently, the Rust kernel has to replace the exception vector table to prevent the Linux kernel from panicking. Instead, our boot loader could set up a default exception vector table for the Rust kernel. This default vector table could use the virtio console to send messages to Linux to inform the user that an exception occurred. In case of a fatal exception which causes a panic, like a null pointer exception, it could print useful debugging information, like the type of exception or the address where the exception was caused. The Rust kernel could still modify or replace the default exception vector table provided by the boot loader if it needs to.

7.4 Share Memory between Rust Kernels

The kernel module ensures that a memory range is not assigned to multiple Rust kernels (see Section 4.2). The intention is to prevent the case where two Rust kernels are under the impression that they own a range of memory exclusively, but they actually share it with each other. But only assigning memory exclusively might be too restrictive. Booting multiple Rust kernels and letting them share memory could introduce interesting possibilities to use our boot loader, like building an OS that uses multiple Rust kernels, similar to Popcorn (see Section 3.2). Not much is needed to support sharing memory among Rust kernels, just a mechanism to circumvent the memory assignment check that disallows assigning the same memory range multiple times. One could add a parameter to the ioctl used to request memory that allows to specify the memory as sharable with other Rust kernels.

7.5 Release a Kernel Instance

In the current implementation, the only way to release a kernel instance and the resources assigned to it is to unload the kernel module. This can be used to start a Rust kernel again after it stopped: if the previous Rust kernel instance stops its execution, then reload the kernel module and boot the Rust kernel again. The new Rust kernel instance can use the same resources as its predecessor, as we saw in Section 6.2. However, every time the kernel module is unloaded, all kernel instances are released. So an ioctl could be added that allows to release a single kernel instance. The function `delete_proc_instance()` in `kernel-module/procfs.c` is responsible for releasing a kernel instance. So an ioctl that releases a kernel instance could just call this function. However, there are two issues with how this function releases memory in the current implementation. The first issue to be aware of is that it leaks memory, because the page tables created to map the memory into the address space of the Rust kernel are not freed. The second, more important issue is that if the Rust kernel instance, whose resources are being released, is still executing, then overwriting its memory with zeros will cause a null pointer or an invalid instruction exception, as described in Section 4.7.1. In order to safely free a kernel instance's memory, the Rust kernel has to stop its execution with the PSCI function CPU_OFF. The kernel module

could use the PSCI function AFFINITY_INFO to check whether the CPU used by a kernel instance is turned off. Adding such a check would make an ioctl to release a kernel instance safer. In addition to that, it would be useful if there was a way to instruct the Rust kernel to call the CPU_OFF PSCI function. If the Rust kernel sets up an exception vector table, then it could add an interrupt which Linux can raise to cause the Rust kernel to call CPU_OFF. Alternatively it could also be caused by a command sent over the virtio console. Adding these two mechanisms, a check that the Rust kernel is not executing and a method to make it stop its execution, would make an ioctl to release a kernel instance safer and more useful.

8 Conclusion

In this thesis, we describe a way how Linux can be used as a boot loader. What makes this boot loader special is that Linux keeps running after booting a Rust kernel, resulting in a system where multiple kernels run in parallel. To prevent the kernels from interfering with each other, resources need to be restricted and assigned to each kernel. Linux is restricted from using the entire memory with a kernel parameter, the unused memory can be assigned to Rust kernels. CPUs are all assigned to Linux in the beginning. Every boot invocation removes a CPU from Linux and hands it over to the booting Rust kernel. Devices are per default owned by Linux. They can either be removed from Linux's control by disabling the corresponding device driver or they can be shared. In the shared case, the devices are controlled by Linux and the Rust kernel can use them through Linux via virtio. The required software, virtio driver and virtio device, needs to be implemented once per device type, but allows the Rust kernel to use all devices of that type through the device support of Linux. The device type we implemented for this thesis is the virtio console. The boot process itself is implemented inside a kernel module. To boot a Rust kernel on a certain CPU, it lets Linux perform the entire CPU shutdown process, except for the part that actually shuts the CPU down. As a result, from the point of view of Linux, the CPU is shut down, so Linux will not touch it anymore. In the meantime, the CPU starts executing the Rust kernel. So in this boot loader, the boot process does not change the power state of a CPU, it just removes it from the control of Linux. For that reason, the CPU is in an initialized state when the Rust kernel starts executing. The MMU is enabled, all memory assigned to the Rust kernel is mapped into the current address space, an identity map which is set up by the kernel module prior to boot. To suit the needs of a specific Rust kernel, the boot environment can be modified before control is handed over to it. The kernel module of this boot loader allows to load a kernel binary in the form of an ELF executable. Other kernel binary formats can be loaded from user space if a corresponding loader is available. User space programs interact with the kernel module through the kernel module interface. It allows to operate the kernel module, like instructing a boot process, assigning memory to a Rust kernel or mapping that memory into user space. By mapping Rust kernel memory into user space, it is possible to initialize it before boot or communicate with the Rust kernel after boot.

We conclude that it is feasible to use Linux as a boot loader which boots a Rust kernel and runs in parallel with it while providing support. The support the current implementation provides is limited to console I/O, but we expect it to be easily extensible to support more device types. While not tested with a mature Rust kernel, we saw that this boot loader can boot a simple kernel, in the form of a bare metal Rust program. This kernel profits from the initialization done by Linux, it starts with paging and caching set up, fatal exceptions are reported by a Linux kernel panic and, as we saw with the UART device, initialized devices can be taken over. Together with the device support provided by Linux over virtio, this makes it easier to set up and run a new kernel in development.

The boot loader also improves the portability of a Rust kernel, in terms of the machines it boots on as well as the devices it supports through virtio. However, as we tested the boot loader only on two different systems so far, the preceding statement on portability has to be taken with a grain of salt and requires further evaluation. The boot loader itself profits a lot from Linux as well, being able to use networking, various devices and software written for Linux. One can easily connect to the boot loader over the internet, modify a Rust kernel, build it and then boot it, all on the same machine. As long as the previous Rust kernel terminates, the boot loader can reuse its resources and boot a new Rust kernel, without needing to reboot the machine every time. The current implementation does leak memory though, so Linux will eventually run out of memory. This boot loader requires no modifications to the Linux kernel itself, all instructions that are performed in kernel space are contained within a kernel module. This boot loader is still in its early stages, it can be improved in many ways. From features that make it more convenient to use or would be useful in specific cases to essential features like supporting more device types over virtio. Apart from improving the boot loader itself, it can be used as a stepping stone for other future projects in OS development.

Overall, this boot loader simplifies the development of new operating systems on ARMv8-based systems, it takes care of booting up the machine, leaves the system in an initialized state and shows that device interaction can be left to Linux to simplify the implementation and make it more portable.

Appendix

Build Rust Kernel

A bare metal rust program that runs on an 64 bit ARM-based system should be built with the `aarch64-unknown-none` target triple. The target triple and the base address of the ELF image can be set by adding a cargo configuration file `.cargo/config.toml` with the following content:

```
target = "aarch64-unknown-none"
rustflags = ["-Clink-arg=--image-base=0x100"]
```

This would set the base address to 0x100.

References

[1] "Memory Layout on AArch64 Linux - The Linux Kernel documentation." https://docs.kernel.org/arm64/memory.html, May 2022. [Online; accessed 10. May 2022].

[2] R. Russell, "virtio: Towards a De-Facto Standard For Virtual I/O Devices," *ACM SIGOPS Operating Systems Review*, vol. 42, no. 5, pp. 95–103, 2008.

[3] "Virtual I/O Device (VIRTIO) Version 1.1." https://docs.oasis-open.org/virtio/virtio/v1.1/virtio-v1.1.html, Dec. 2018. [Online; accessed 5. Apr. 2022].

[4] D. Milea, "Hypervisor-less virtio." https://www.openampproject.org/docs/blogs/HypervisorlessVirtioBlog_Feb2021.pdf, Mar. 2021. [Online; accessed 6. Apr. 2022].

[5] OpenAMP, "Hypervisor-less virtio GitHub README." https://github.com/OpenAMP/kvmtool/blob/master/README_RSL.md, Apr 2022. [Online; accessed 6. Apr. 2022].

[6] "KVM tool." https://github.com/kvmtool/kvmtool, Apr 2022. [Online; accessed 7. Apr. 2022].

[7] OpenAMP, "Hypervisor-less KVM tool - notification character device." https://github.com/OpenAMP/kvmtool/blob/master/user-mbox-rsld/user-mbox.c, Apr 2022. [Online; accessed 13. Apr. 2022].

[8] R. G. Minnich, "Give your bootstrap the boot: Using the operating system to boot the operating system," in *2004 IEEE International Conference on Cluster Computing (IEEE Cat. No. 04EX935)*, pp. 439–448, IEEE, 2004.

[9] W. Almesberger, "kboot - a boot loader based on kexec," in *Proceedings of the Linux Symposium*, vol. 1, pp. 27–38, Citeseer, 2006.

[10] open power, "Petitboot." https://github.com/open-power/petitboot, May 2022. [Online; accessed 22. May 2022].

[11] kexecboot, "kexecboot." https://github.com/kexecboot/kexecboot/wiki/About, May 2022. [Online; accessed 22. May 2022].

[12] "The EFI Boot Stub - The Linux Kernel documentation." https://www.kernel.org/doc/html/latest/admin-guide/efi-stub.html, May 2022. [Online; accessed 22. May 2022].

[13] A. Barbalace, B. Ravindran, and D. Katz, "Popcorn: a replicated-kernel os based on linux," in *Proceedings of the Linux Symposium, Ottawa, Canada*, 2014.

[14] A. Barbalace, M. Sadini, S. Ansary, C. Jelesnianski, A. Ravichandran, C. Kendir, A. Murray, and B. Ravindran, "Popcorn: Bridging the programmability gap in heterogeneous-isa platforms," in *Proceedings of the Tenth European Conference on Computer Systems*, pp. 1–16, 2015.

[15] "sysfs - The Linux Kernel documentation." `https://www.kernel.org/doc/html/latest/filesystems/sysfs.html`, Apr. 2022. [Online; accessed 22. Apr. 2022].

[16] "UEFI GetMemoryMap() Boot Services Function - ACPI Specification 6.4 documentation." `https://uefi.org/htmlspecs/ACPI_Spec_6_4_html/15_System_Address_Map_Interfaces/uefi-getmemorymap-boot-services-function.html`, Jan. 2021. [Online; accessed 26. Apr. 2022].

[17] BarrelfishOS, "Barrelfish - ELF loaderr." `https://github.com/BarrelfishOS/barrelfish/tree/master/lib/elf`, May 2022. [Online; accessed 19. May 2022].

[18] rCore OS, "virtio-drivers." `https://github.com/rcore-os/virtio-drivers`, Apr 2022. [Online; accessed 5. Apr. 2022].

[19] "ioctl based interfaces - The Linux Kernel documentation." `https://www.kernel.org/doc/html/latest/driver-api/ioctl.html`, May 2022. [Online; accessed 5. May 2022].

[20] Arm, "Arm Architecture Reference Manual for A-profile architecture." `https://developer.arm.com/documentation/ddi0487/ha`, May 2022.

[21] BarrelfishOS, "Barrelfish - pl011 UART driver." `https://github.com/BarrelfishOS/barrelfish/blob/master/kernel/arch/arm/pl011.c`, May 2022. [Online; accessed 19. May 2022].

Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Declaration of originality

The signed declaration of originality is a component of every semester paper, Bachelor's thesis, Master's thesis and any other degree paper undertaken during the course of studies, including the respective electronic versions.

Lecturers may also require a declaration of originality for other written papers compiled for their courses.

I hereby confirm that I am the sole author of the written work here enclosed and that I have compiled it in my own words. Parts excepted are corrections of form and content by the supervisor.

Title of work (in block letters):

Linux as a universal boot loader for new operating systems

Authored by (in block letters):
For papers written by groups the names of all authors are required.

Name(s):	**First name(s):**
Walter	Jan Nino

With my signature I confirm that

- I have committed none of the forms of plagiarism described in the 'Citation etiquette' information sheet.
- I have documented all methods, data and processes truthfully.
- I have not manipulated any data.
- I have mentioned all persons who were significant facilitators of the work.

I am aware that the work may be screened electronically for plagiarism.

Place, date	**Signature(s)**
Rifferswil, 27.05.22	Walter

For papers written by groups the names of all authors are required. Their signatures collectively guarantee the entire content of the written paper.