University of Pforzheim
School of Engineering

Master Program Embedded Systems

Master Thesis

The Zircon Kernel
A Consideration of a Microkernel Approach and its Effects on Driver Development

Examiner	Prof. Dr.-Ing. Rainer Dietz
Second Supervisor	Prof. Dr.-Ing. Martin Pfeiffer
Author	B.Sc. Anna-Lena Marx
Matriculation Number	317593

Submission Date 20.06.2019

Eidesstattliche Erklärung

Ich erkläre hiermit, dass ich die vorliegende Master-Thesis selbstständig, ohne Hilfe Dritter und ohne Benutzung anderer als der angegebenen Hilfsmittel angefertigt habe. Die aus fremden Quellen direkt oder indirekt übernommenen Gedanken sind als solche kenntlich gemacht. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt und auch nicht veröffentlicht.

Pforzheim, den 19. Juni 2019

Kurzfassung

Monolithische Betriebssystemkerne waren lange Zeit der Stand der Technik in beinahe allen Anwendungsgebieten. Für speziellere Bereiche, beispielsweise für Echtzeit-Betriebssysteme, sind dagegen seit einigen Jahren vor allem Mikrokernel-basierte Systeme auf dem Vormarsch. Nun ist mit Googles Fuchsia beziehungsweise seinem Kernel Zircon ein weiteres Mikrokernel-basiertes Betriebssystem veröffentlicht worden, bei dem zwar der Einsatzbereich bislang unbekannt, aber entsprechendes Potenzial, verbreitete monolithische Betriebssysteme zu verdrängen, vorhanden ist. Die vorliegende Arbeit untersucht daher die Unterschiede zwischen Linux als monolithischen Vertreter und Zircon als Mikrokernel bezüglich grundlegender Konzepte der Betriebssysteme-Theorie und deren Umsetzung mit einem Schwerpunkt in der Treiber-Entwicklung. Zu diesem Zweck wurde im zweiten Teil der Arbeit eine Fall-Studie anhand einer exemplarischen Gerätetreiber-Entwicklung durchgeführt, die aber auch den Hauptkritikpunkt an Mikrokernel-basierten Betriebssystemen, die Leistungsfähigkeit, mit einbezieht. Die Betrachtung kommt zum Ergebnis, dass die Kernel-Architektur selbst kaum Einfluss auf die Treiberentwicklung hat, sich dafür aber positiv, wenn auch verborgen, auf die Handhabung und Stabilität eines Betriebssystems auswirkt. Allerdings kann auch die verminderte Leistungsfähigkeit gegenüber monolithischen Betriebssystemen aufgrund einer simplen Analyse bezüglich der Dauer einer Systemaktion aus einem Treiber heraus bestätigt werden. Ein bemerkenswerter Vorteil des Zircon Kernels gegenüber Linux liegt in zahlreichen Implementierungsdetails, die einige historisch bedingte, umständliche Programmierkonstrukte in der Linux Treiberentwicklung obsolet machen. Die Arbeit richtet sich vor allem an System- und Gerätetreiber-Entwickler, die an den Unterschieden zwischen monolithischen und Mikrokernel-basierten Systemen interessiert sind und eventuell sogar Inspiration zur Verbesserung von bereits bestehenden Kernels suchen.

Schlagwörter Betriebssysteme, Monolith, Mikrokernel, Gerätetreiber, Zircon, Linux

Abstract

Monolithic operating system kernels were the state of the art in almost all fields of application for a long time. For more specific areas, such as real-time operating systems, microkernel based systems have been gaining ground for some years now. With Google's Fuchsia, respectively its Zircon kernel, another microkernel based operating system has been released which has the potential to replace monolithic systems, even if the field of application is still not known right now. Therefore, the present work examines the differences between Linux as a monolithic representative and Zircon as a microkernel with respect to basic concepts of operating system theory and its implementation with a focus on device driver development. The second part of this thesis therefore includes a case study based on an exemplary device driver implementation, which analyzes the main issue of microkernel based operating systems as well, the performance. The analysis concludes that the kernel architecture itself is hardly influencing the driver development, although it has a positive but hidden effect on the handling and stability of an operating system. However, the reduced performance compared to monolithic operating systems can be confirmed by a simple test examining the duration of a system action from a driver as well. An additional, notable advantage of the Zircon kernel over Linux is caused by numerous implementation details that make some, for historical reasons, cumbersome programming constructs in Linux driver development obsolete. The work is primarily aimed at system and device driver developers who are interested in the differences between monolithic and microkernel based operating systems and may even seek inspiration to improve existing kernels.

Keywords operating systems, monolith, microkernel, device driver, Zircon, Linux

Contents

Chapter 1

Introduction

Modern application Central Processing Units (CPUs), as used for e.g. desktop computers or smartphones, are highly complex circuits consisting of at least two independent actual processing units. It is hardly possible to master these multicore processors, associated memory and Input/Output (I/O) devices in their full complexity nor take advantage of their actual computing power or security mechanisms without using an operating system as an abstraction layer between the actual hardware and their users. As such this goal of an operating system, to manage a computer's physical resources and share them between a multitude of user programs in a fair and safe way, is the central task. The part of operating systems providing this task is commonly referred to as the *kernel*. Even if this distinction is widely used, especially for Linux, there is no clear and global definition which specific tasks of an operating system are part of the *kernel*. Albrecht Achilles named process management, memory management and basic I/O operations as such ones in his book[1]. Actually, this distinction between a whole operating system and its kernel itself is so difficult to draw because it is not even consistent between different operating system architecture concepts. There are a lot of different goals and use-cases for operating systems which have a big impact on the architectural design, and therefore the term *kernel* too. But for a majority of them, at least the central *device drivers* are considered as a part of the kernel.

With *Linux* and *Fuchsia*, two fundamentally different representatives of the two best-known operating system concepts will be considered in this work, the microkernels and the monolithic ones. Both are representatives for very opposing kernel architecture approaches. A monolithic kernel runs entirely within the CPU's kernel mode, a privileged execution mode, while microkernels try to reduce their share in kernel mode and swap as much functionality as possible into user mode. *Fuchsia*, respectively *Zircon*, as the kernel is named, is a very new microkernel operating system which is developed from scratch by *Google*. The project is developed in public since 2016[1], but since it has gone public, there has been no official announcement about the purpose of the system. Nevertheless, there are a lot of reasons to study *Fuchsia* and especially its kernel *Zircon*. The first one is clearly an architectural one, as *Zircon* is a microkernel. Of course there

[1] androidpolice.com, visited on 13.02.2019 `https://www.androidpolice.com/2016/08/12/google-developing-new-fuchsia-os-also-likes-making-new-words/`

are some industrially used microkernel architectures, but the most successful ones are Portable Operating System Interface for UniX (POSIX) compatible and targeting real time tasks, such as *QNX* or *FreeRTOS*. But apart from the unknown use-case, *Zircon* differs in this very important aspect, the POSIX compatibility which provides a global standard (DIN 9945, IEEE1003.2) for UNIX-like operating systems[51]. That is a very brave decision as POSIX compatibility is a common part of public tenders for software. Beside this, one further point making *Zircon* an interesting research topic is *C++* as preferred programming language for the kernel, which is a rather unusual choice[49].

Linux is in many issues the very opposite to *Zircon*. It was started as a hobby project by Linus Torvalds in 1991 and grew over the years to a widely spread, powerful kernel. Today, it is, due to its use in *Android*, the most used operating system kernel overall[2]. Since Linux was published, it is actively developed by a diverse community and is not subject to the interests of a single company. Presumably, that is a major factor for its spread besides its licensing. Nevertheless, *Linux* is a grown software project. Architectural decisions, for example the one for its implementation as a monolithic kernel and its relation to *UNIX*, were taken in the very beginnings of *Linux* as a personal hobby project and affect it still today. In a well-known mailing list discussion with Andrew S. Tanenbaum in 1992, Torvalds even admitted the theoretical superiority of microkernels over monolithic ones[50]. But some years later, in 2006, the Linux contributor Richard Gooch justified the question of why *Linux* won't change to a microkernel architecture with performance drawbacks compared to monolithic ones due to privilege barriers[18].

Along with the guess that *Zircon* will target the same sector as *Linux*, especially the mobile and embedded sector, both of them make for an interesting comparison to examine the impacts of kernel concepts on an essential part of them, the *device driver* development. In this process, the date of origin should be taken into account as an additional factor. It is to be assumed the *Zircon* team is aware of historic, rather poor decisions in *Linux* and other operating systems, and grasps the opportunity of a completely new product.

1.1 Research Objectives

This work's goal is to evaluate the differences between monolithic and microkernel architectures for operating systems and how they influence the development of device drivers. For this purpose, this work shall focus solely on the operating system's kernels, the *Zircon* kernel and the *Linux* kernel. In most literature, filesystems are also listed as an essential part of an operating system, but shall be out of scope of this work. Of course, they are essential, but they form their entirely own field of research. For this particular work, the specific filesystem implementation is not crucial due to the focus on kernel and device drivers. As a result, neither Google's *Fuchsia* nor the *Linux* userland or desktop environments as the users of the operating system kernel shall be a part of this thesis.

[2] netmarkedshared.com, visited on 13.02.2019 `https://bit.ly/2S0s4lM`

Contrarily, the following topics are to be examined within the scope of this work:

- Introducing basic concepts of operating system kernels and evaluating how they are implemented in the specific realizations, *Linux* and *Zircon*.
- Examining the effects of the architectural differences of the two well-known approaches *monolithic* and *microkernel* on device driver development using the example of the non-academic, specific implementations, *Linux* and *Zircon*. This includes:
  - The analysis of the driver models and how they are integrated into the kernel.
  - Examining the driver development itself on both architectures with the aid of a hands-on case study.
  - The inspection of the driver development workflow, especially with a focus on available tooling.
  - A consideration on the way the drivers are used by the kernel itself, especially by other kernel components, and the actual users.
- Evaluating the results of these examinations based on the theoretical concepts and its actual realizations with respect to the historic timeline of *Linux* and *Zircon*.

Chapter 2

Modern Operating System Concepts

This thesis' introduction already picked up the discussion about which operating system architecture is the superior one by referring to the *Tanenbaum-Torvalds debate*[50] in 1992, a discussion being quite interesting from today's point of view on different operating systems and their future. Tanenbaum and Torvalds, both are underpinning their arguments with the origins of their implementations *MINIX* and *Linux* in different problem spaces. And roughly spoken, exactly these different problem spaces which need to be addressed by an operating system are one reason for their diversity. Over the years, they had to fit in solving very different kinds of problems on very different kinds of hardware, which resulted in very different implementations and architectures. As with the debate, it is impossible to find one architectural concept or implementation which is clearly superior to the other in every use-case. Nevertheless, it is a reasonable question why a majority of the operating system kernels which were developed from scratch in the last few years are based on a microkernel concept. In 2009, the official statement of the Linux kernel developer Richard Gooch was that monolithic kernels are superior for performance reasons[18]. So the question remains what changed during the last ten years to promote this change, and specifically in the context of this work, how this affects device driver development. For this reason, this chapter is dedicated to the basic components and functionalities of operating system kernels and how they are implemented in the real world examples *Linux* and *Zircon*.

2.1 Operating System Architectures

As already pointed out, architectural decisions for operating systems are commonly influenced by the issues they are intended to solve. By giving priority to some design objectives that are pertinent to the underlying issue, different concepts and architectures are the most appropriate choice. According to Glatz[17], some of them are:

- Providing a reliable, crash-proof environment.
- Providing a portable operating system.
- Providing a scalable operating system, e.g. in terms of processing cores.

- Providing an extensible operating system, e.g. in terms of adding additional functionality to the kernel.
- Providing real-time capabilities.
- Providing an efficient design in terms of resources and performance.
- Providing a secure environment for user applications.
- Providing a maintainable operating system, e.g. by the division of policy and mechanism.

In addition, operating systems should also pay attention to the common software design issue *mechanism vs. policy*, i.e. an operating system design should provide a clear distinction between the *mechanism*, the capabilities that can be performed (how is something done) and the *policy*, which controls how the available capabilities are used (what is done)[26], [44]. An example for driver development could be controlling the number of processes that can use a device at once. In this case, the driver should provide the mechanism, *how* such a limitation could be done, but not *what*, the actual number of allowed processes. The idea behind being that requirements may change over time and such a distinction makes it easy to adjust the *policy* via parameters without touching the underlying *mechanism*[44].

How these design principles fit into the known operating system architectures will be considered in the following sections. But first, the terms *kernel mode* and *user mode* will be explained as they are fundamental for this work.

Dual-Mode Execution

Modern general purpose CPUs provide a ring based, hardware enabled security model which had its origin in the Intel x86 processor architecture[49]. It is usually made of four different security levels, the rings 0 to 3 which are illustrated in Figure 2.1. In this theoretical model, ring 3 is the least secure level, used for common user applications (even if started with extended privileges (*root* for the UNIX-like world)), while ring 2 is used for libraries shared between user applications and ring 1 is for system calls[17]. Actual operating system implementations may not use all of them. Linux for example utilizes ring 3 for user-space applications and shared libraries and ring 0 for system calls and the kernel. System calls provide the transition to ring 0, the one with the topmost security level, which is used for the operating system kernel. As a crucial part of an operating system, system calls will be discussed in more detail later in this work. Directly related to this model is the *dual-mode* execution mode of modern CPUs. It is a hardware enabled security concept to provide a distinction between the user applications in ring 3 and the actual operating system kernel in ring 0. Only the kernel in ring 0, running in the *kernel mode* (or *privileged mode, supervisor mode or system mode*), has direct and privileged access to memory, hardware, timers or interrupts, e.g. for performing I/O operations or memory mappings[26]. User applications in

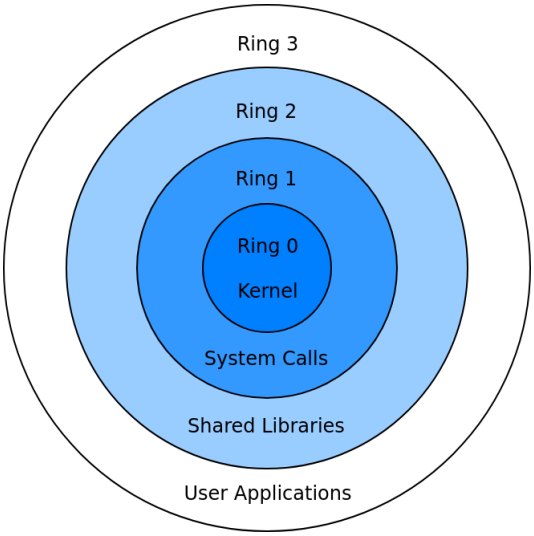

Figure 2.1: The Rings of the x86's security concept according to [17]

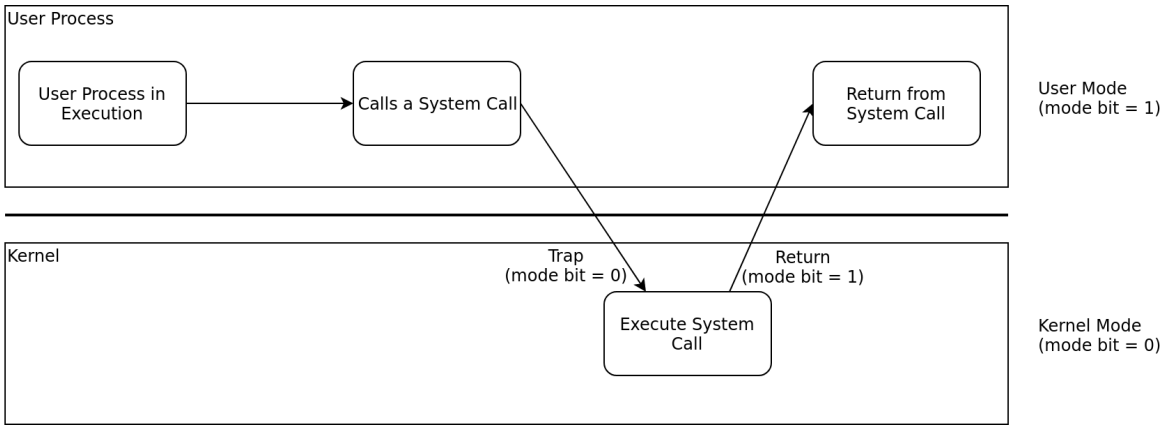

Figure 2.2: A system calls sequence including the mode switches according to [17]

ring 3, running in the *user mode*, are not allowed to use them directly, they have limited privileges and a limited instruction set. As mentioned above, they need to use a mechanism called *system calls* to transfer the execution to the *kernel mode* where the privileged actions are performed. Lastly, the execution is transferred back to the calling user process and with this, the mode changes back to *user mode*. Figure 2.2 pictures the operating flow of a system call including the mode switches between *user* and *kernel mode*.

The CPU's operating mode is usually controlled by a specific bit in the Program Status Word (PSW)[49]. PSW is the term on x86 CPU architectures. It is corresponding to the Current Program Status Register (CPSR) on ARM architectures[1]. But since the literature always uses the PSW regardless of the concrete CPU's architecture, this should also be done in this work. This bit influences the state of each CPU core itself

[1]http://infocenter.arm.com, visited on 08.03.2019
`http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dui0473f/CHDFAEID.html`

in a multi-processor system, but not the operating system kernel. As a result, different CPU cores may be in a different execution mode[26]. With this separation, any privileged instruction is forbidden in *user mode* and will not be executed.

Based on the dual mode execution on the CPU, different architectural concepts for operating systems evolved. They differ e.g. in the share of the operating system respectively the operating system's kernel actually running within the CPU's *kernel mode*. Thus, they influence the whole system, including device driver development but also performance and security issues.

With this basic knowledge about the CPU's operating modes, the next section researches a selection of different operating system architectures. Special attention should be paid to the most common ones, the *monolithic* and the *microkernel* architectures and their implementation in Linux and Zircon. Importantly, this work will not discuss special purpose operating system architectures such as e.g. Integrity OS for realtime applications or the Cray Linux Environment which is a Linux adaption for supercomputers. Today, even the majority of general purpose computing systems are driven by more than one CPU core and most common modern operating systems are designed to provide support for the de facto standard for tightly coupled systems, Symmetric Multiprocessing (SMP).

2.1.1 Monolithic Architectures

Some sources, such as Glatz[17] or Silberschatz[44], suggest monolithic operating systems do not have a well-defined structure at all. As they are indeed most commonly grown structures, started in a completely different scope (MS-DOS, the original UNIX), it is not an incorrect claim. But it does not necessarily have to be the case. Above all, monolithic operating system (kernel) architectures have in common that they form one single binary program which is running entirely in kernel mode. User programs, running in user mode, interact with the kernel only through a well-defined set of *system calls*[26]. Within the kernel itself, all parts are free to use and access each other even the hardware, without any limitation, e.g. regarding the access of kernel functionalities of another component or hardware access. So a function or procedure initially developed for scheduling processes could be used in a completely different context if its functionality is useful to solve another issue. In fact, there is no information hiding between kernel functions or procedures. Any function in this kernel context has full access to the hardware, such as I/O devices, timers, interrupts and even to the memory. There is no memory protection or validation between different components of a monolithic kernel. Of course, this leads to some serious disadvantages in this architecture, for example, a crash in one single function or procedure could crash the entire kernel or the resulting system may become difficult to understand and maintain[49], [44]. The missing memory protection within the kernel could also be a source for crashes or attacks. But in contrast, this design also enables a very efficient kernel design without any unneeded communication overhead or hardware inefficiencies[26].

An extension of the monolithic architecture are the so-called **modular operating systems**. They provide additional, defined interfaces for (usually) dynamically load-

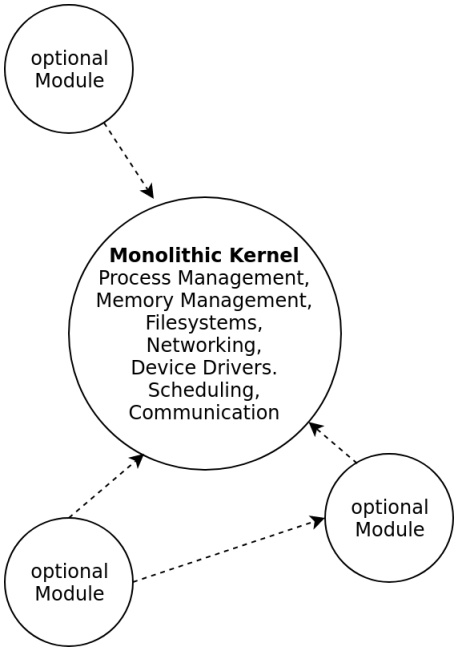

Figure 2.3: A Monolithic Kernel Architecture according to [26]

able and unloadable extensions, e.g. for device drivers or filesystems. Sometimes, such extensions or modules are just allowed to use a limited function set of the operating system, but they are still running as a part of the kernel in kernel mode[26], [49]. Just like ordinary kernel functions or procedures, (malicious) programming errors in extensions may lead to a kernel crash, manipulation or damage to other components. On the other hand, the modular concept provides some advantages over regular monolitic kernels. It allows slimming down the actual kernel by providing the ability to load only the actually needed functionality dynamically and e.g. security patches within such an extension are possible without restarting the entire system[4]. As the extensions become a part of the operating system running in kernel mode, no additional communication effort between the actual kernel and the modules is required. Figure 2.3 pictures an exemplary architecture for a monolithic kernel using optional modules. Such a concept of modules is quite popular for monolithic operating systems like *Linux* or *Solaris*[4], [44].

2.1.2 Microkernel Architectures

The microkernel architecture focuses on very different design goals compared to the monolithic one. Some of them are to cope with the complexity, rather poor maintainability and susceptibility to errors by a massively modular approach. To achieve this, the core idea behind microkernels is to provide only a very small kernel running in kernel mode which only provides the core functionalities while all the other important functions of an operating system are running in user mode. Thereby, the microkernel architecture is excellently suited to implement a proper division of mechanism and policy. The kernel provides just the most basic mechanisms needed for an operating

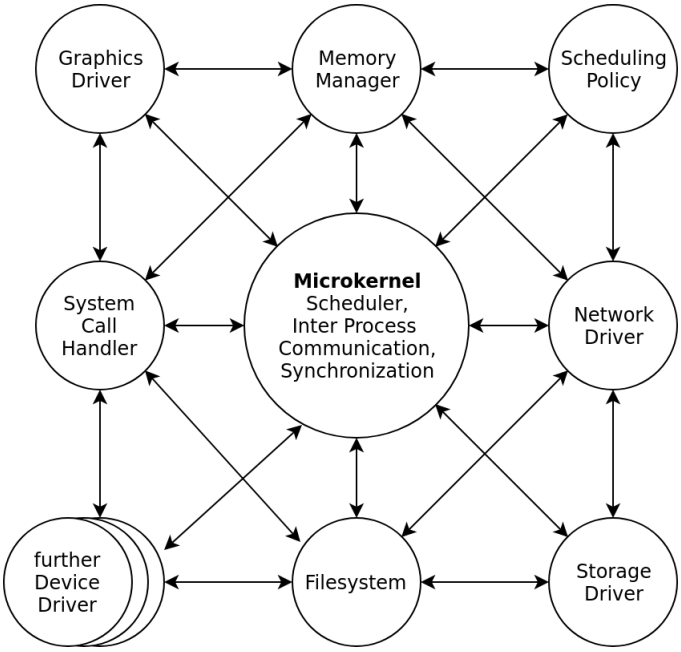

Figure 2.4: A Microkernel Architecture according to [26]

system, while the userspace modules implement the policy. This decoupling makes it easier to change the policy in userspace for altering requirements without touching the actual kernel[49]. What is part of this core functionality differs between the various sources, but all considered ones are in agreement that a simple mechanism for process scheduling is as well a core functionality as providing an Inter-Process Communication (IPC) mechanism[26], [44], [17]. In contrast, the sources disagree whether memory management and virtualization, device drivers or synchronization facilities are a part of the actual kernel. The *Mach* microkernel, which formed the first generation of microkernels in 1985, named process and thread administration, an extensible and secure IPC mechanism, virtual memory management and scheduling as its core tasks, while everything else needed has to run in usermode[41]. Functional enhancements of the system do not require changes to the kernel itself, too. This concerns, depending on the exact realization, device drivers, memory management, system call handlers and even more system components[26], [44]. Figure 2.4 tries to picture this situation. In academic microkernel approaches, all components in user mode run within their own userspace process as small, well-defined modules, while the communication is done through copious message passing via the actual kernel[49], [26]. Since the restrictions by the CPU's dual mode still apply for microkernel based operating systems it is not allowed for device drivers running in user mode to have direct physical access to I/O ports as a consequence. A device driver has to invoke the actual kernel to perform the needed action in substitution. But thus, the kernel is able to check the actions and whether the driver is authorized to execute them. As a result, the microkernel design is more reliable and secure as such a division enables the kernel to intercept erroneous actions such as accidental memory writes to important regions[49]. Equally, a crash

in an userspace system component like a driver is not able to crash the entire kernel in such an approach. And as an additional advantage, the microkernel architecture facilitates porting the operating system kernel to another target architecture as most hardware dependencies are part of the small kernel[44], [26].

With all the described advantages of microkernels, the question remains why microkernels are primarily used in real-time environments, avionics or military, but not for desktop operating systems. One reason is that all these advantages are paid by the high price of microkernel message costs. For the mentioned application areas, especially the reliability that comes with the microkernel architecture is more desirable than the performance costs of the more frequent context switches in comparison to monolithic architectures[49]. Since a lot of the operating system's functionality has been moved to the userspace, microkernel architectures need to perform noticeable more context switches to invoke the actual kernel for privileged actions. The performance losses are not only caused by the large amount of context switches themselves, but also by the fact that modern CPUs, particularly the caches, are not designed for them. Every context switch causes cache misses which triggers that the required data has to be loaded from the slower main memory and cached. The data of the previous context (e.g. the user mode context) will be dismissed from the cache and the CPU is largely blocked in the meantime. By rapidly switching back to the previous context, as it is usual for e.g. a short kernel invocation to perform an I/O operation on microkernel architectures, the cache is no longer suitable for the new context and has to be replaced[26].

First of all, the *L4* kernel, a second generation microkernel was able to get close to the performance of a monolithic kernel like *Linux*[21]. This has been achieved by improving the Inter-Process Communication (IPC) mechanism, a fundamental component of microkernel architectures on which other communication mechanisms are based. Nevertheless, pure microkernels are mainly used for systems with high reliability requirements but unusual for desktop application. Some industry examples are *Integrity*, *QNX* and *seL4*, a mathematically verified version of the *L4* kernel[49].

2.1.3 Layered Architectures

Layered operating system architectures are usually organized in hierarchical layers, but sometimes the chosen model is described as a series of concentric rings. Each layer or ring provides a group of functionality while it is only allowed to use the functions of the one directly below. The cooperation between the layers or rings is regulated by clearly defined interfaces[4]. This is usually accompanied by the fact that the lower layers or inner rings are more privileged than the outer ones. However, there is no uniform and universally accepted approach for division in layers and their count according to this pattern[17], [49]. In fact, a meaningful division is not that easy. Functionalities may have to be divided artificially and the harmonious arrangement can have its pitfalls caused by the access requirements this architecture is based on. Is the layered access model applied properly to get a clean architecture, only then it is able to unfold its advantages. These are for example the interchangeability of the layers if they and their interfaces were properly designed or the resulting concept for debugging. As the layers

are constructed on top of each other, it is possible to debug and verify each one for its own, starting at the lowest layer up to the top most one[44]. But also the costs for system calls are comparatively high, because they have to be passed though all layers while each one adds overhead to such a call[44].

In general, layered operating system architectures are related to monolithic ones, but it is conceivable to adopt the idea for microkernel approaches, e.g. the MINIX userspace is divided into layers. Examples for this architecture are *OS/2* or newer *Unix* variants, while *Multics* is one for a concentric ring based model[17], [49].

2.1.4 Hybrid Architectures

Hybrid operating system architectures based on monolithic concepts, microkernels and maybe layeres are a common approach to combine the advantages of these concepts. They try to pair the performance of the monolithic design with the modularity and reliability of microkernels[52], [44]. How both worlds interact is very different depending on the exact implementation. SILBERSCHATZ explains one of them in his book *Operating System Concepts*[44] using Apple's *OS X* (today named *macOS*) as an example. Depending on the exact implementation and the share of the architectures, most disadvantages of monolithic architectures still apply for the hybrid systems. Further examples for hybrid architectures are *Windows NT* and *BeOS*[52].

2.1.5 The Linux Kernel's Monolithic Architecture

Linux is the perfect example for an extremely grown operating system. Starting as a pure hobby project to learn about a specific CPU and connect to the Unix computers at LINUS TORVALDS, its initial author, university, it became strongly related to its archetype, *Unix*[10]. They share fundamental design goals, just like being capable of multiple processors and users at the same time, while not being based on the origin Unix source code[49], [26], [10]. It is entirely running in kernel mode and all built-in layers have full access to the internal kernel Application Programming Interface (API) using common function calls like in C. A sophisticated concept of kernel modules which can be dynamically loaded into a running kernel makes a limited number of microkernel advantages available for Linux. Modules in Linux are only allowed to use a restricted (exported) set of functions to use, but once loaded into the running kernel, they become a part of the monolith running in kernel mode[26]. Linux is largely compatible to the POSIX standard which was initially created for Unix. Initially, it was because TORVALDS could not get a version of the standard, while today it is rather a conscious decision[10], [49]. Also, the decision for the monolithic architecture is today consciously supported by the kernel community and justified with its performance and efficiency over microkernels due to the *privilege barrier* between user and kernel mode which has to be passed quite often in microkernel architecture[18], [26]. The Linux kernel itself is divided in five essential tasks which are also reflected in its source code. They are:

- Process Management,
- Memory Management,
- Filesystems,
- Device Management and
- Networking[26].

By structuring this tasks and further components into *subsystems* like *drivers/*, *fs/* (filesystems), *net/* or *kernel/*, the Linux kernel remains comprehensible and in some ways modular. A closer look at most of them in general but also their implementation in Linux is done in the following sections.

The Linux kernel is mainly written in C but some very hardware dependent parts are in Assembly. Additionally, especially the Assembly parts were strongly dependent on the GNU Compiler Collection (GCC). Today, there are some efforts to reduce the share of Assembly for maintenance and readability since modern compilers do not generate less efficient code as hand-written Assembly is[19]. This also reduces the dependency to GCC and enables the use of alternative compilers, especially Clang[11], [8]. Nevertheless, the Linux kernel's principal language is C and it only provides support for C drivers.

2.1.6 The Zircon Kernel's Microkernel Architecture

In contrast to the Linux kernel, Zircon is not a grown structure. Started in 2015, it was largely developed from scratch by Google for a so far undisclosed field of application[16]. Nevertheless, Zircon emerged from a branch of *Little Kernel* (LK) by Travis Geiselbrecht who is also a part of the Zircon Team at Google[32]. Despite its origin, Zircon is very different to Little Kernel. It targets powerful devices such as modern computers and phones and provides for this reason only 64-bit support, first class user-mode support and a capability-based security model. In contrast, Little Kernel is designed for embedded applications and amongst others used as bootloader for *Android* and as *Android Trusted Execution Environment (Trusty TEE)*[45]. It has 32-bit support, but none of the more sophisticated features Zircon has[32].

The microkernel architecture is justified by having security, safety, reliability and modularity as major design goals for Zircon. According to Travis Geiselbrecht, the architecture was a conscious tradeoff between the named goals and performance[16]. They try to avoid costly context switches as much as possible, speed up the remaining ones and take advantage of Symmetric Multiprocessing (SMP), but it is not the focus of Zircon. Alike, Zircon does not focus on performing I/O operations or process management which are the key tasks POSIX was designed for[16]. As a result, Zircon does not claim to be or to become POSIX compatible, they just support a very basic subset of the standard[22]. The Zircon kernel itself is split up into the actual microkernel running in kernel mode (*kernel/*) and services, drivers and core libraries running in user mode (*system/*)[46]. The kernel part provides the basic operating system mechanisms:

- Process Management,
- (virtual) Memory Management,
- Inter-Process Communication and
- Synchronization Mechanisms[46].

The part running in user mode contains core services for, amongst others, booting, device management and networking, device drivers respectively hardware related code and user libraries.

Zircon is for the most parts written in C++ and less in C. It provides native support for device drivers in both languages but due to the fact that Zircon provides an Interface Definition Language (IDL) which defines a contract for in-process drivers, other languages are conceivable as well. In fact, support for Rust drivers is currently being worked on[16]. Unlike Linux, Zircon provided support for both, the GCC and the Clang compiler, from the beginning caused by the sophisticated tools around Clang and LLVM.

2.2 System Calls

System calls were already marginally mentioned in this work as the mechanism to switch the program execution between user and kernel mode, because applications running in user mode have only restricted rights. Thus, these special calls are needed for the interaction with basic hardware devices like the CPU, the memory, peripherals or filesystems and for invoking the actual operating system's kernel for management operations like process management[26]. System calls are to a high degree hardware dependent and differ between various operating system implementations.

A system call has its origin in an application running in user mode. If the application has to invoke the operating system kernel, e.g. to perform an action on memory in substitution, it has to use one for switching the operating mode[17], [49]. As switching the CPU's execution mode also means a new context, a system call to the kernel differs from a common procedure call. In user mode, the so-called entry code stores the system call's parameters in a defined way. One is to store the parameters and the call's number in defined registers (see 2.5), another one is to store them on stack, according the C/C++ calling convention in reverse order[44], [17]. The exact one depends on the actual CPU architecture but also on the operating system kernel itself. Linux, for example, has in general another convention for system calls than Windows. The following instruction triggers a special software interrupt containing the order to switch the context. It is also named *trap instruction*[17], [49]. To be exact, it is the interrupt vector number of the trap instruction which is responsible for the switch. They are `0x80` on Linux as pictured in 2.5 and `0x2e` on Windows systems[17]. But right before switching it is needed to save the Program Status Word (PSW), which contains the actual processors state including the mode bit, to the stack. The same number is used in kernel mode as an index within the interrupt vector table (or Interrupt Descriptor

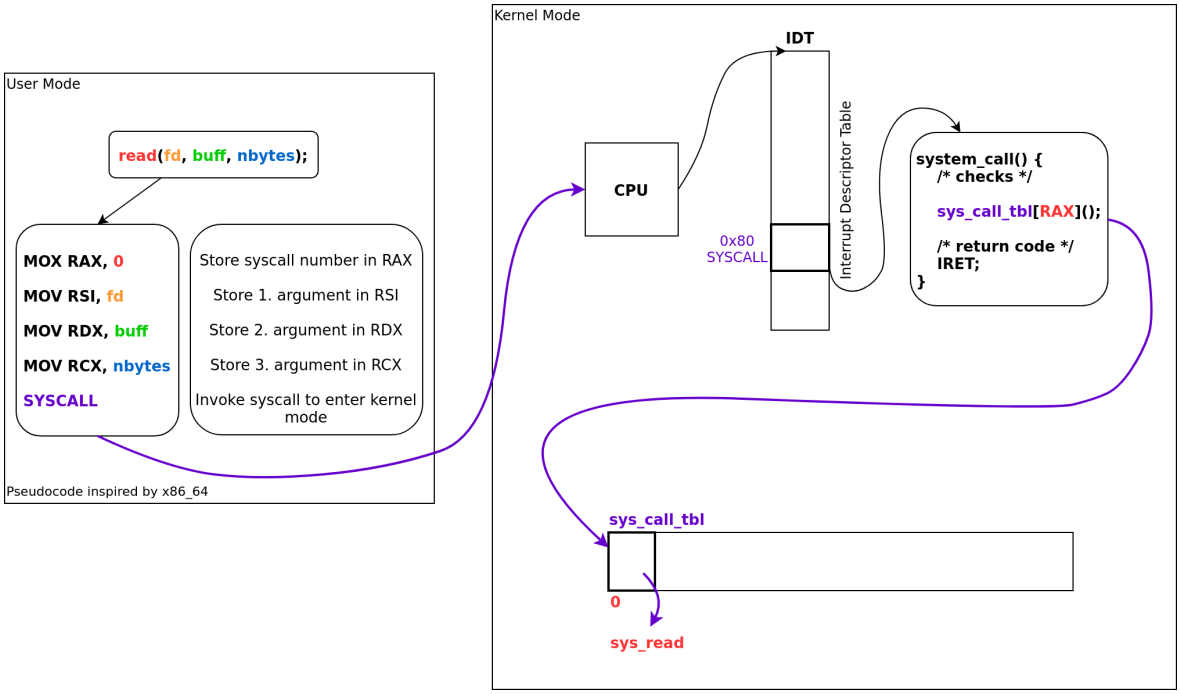

Figure 2.5: System Call Implementation inspired by [26]

Table (IDT)) which contains the start address of the system service dispatcher routine (compare to 2.5). This table's content, the system service dispatcher routine, is loaded as next instruction to the new PSW[4]. Jumping to this routine, the system call's parameters and its number are restored to examine the actual call and invoke the matching service routine from the system service dispatching table which finally fulfills the requested action as pictured in simplified form in 2.5[17]. Conclusively, the control flow jumped back to the system service dispatcher which hands the control back to user space including switching back to user mode in the common way to return[17]. The previous PSW is restored from the stack containing the bit for the CPU's user mode execution. The user application has to clean up the stack like for each procedure call at the very end[49].

More modern versions use the special instructions `SYSENTER` and `SYSEXIT` (Intel) or `SYSCALL` and `SYSRET` (AMD) instead of the slower trap interrupts[3]. Figure 2.5 uses a x86_64 like assembly and thus, this modern syntax for invoking the systemcall while the IDT also pictures the older number 0x80.

POSIX

The basic idea behind the Portable Operating System Interface for UniX (POSIX) standard is to define a stable interface between the user-space and the operating system kernel to achieve portability for applications on systems meeting this standard. But even if an operating system complies, it may differ in the type and number of available system calls. POSIX is only an API definition but not one for system calls. An operating system may implement the POSIX standard functions within a library which

often involve system calls, but it has not to do so[26], [17]. System calls are rarely used directly by user applications without an abstraction layer such as libraries. An example is the *libc* on Unix-like systems[26], [49]. Equally, a system is not constrained to a single API and may implement different ones using its existing set of system calls[17].

POSIX is a very common standard which is for example implemented in UNIX, macOS, MACH and partly in Linux[49], [17]. Another example for an API is Win32 used by Windows to abstract their system calls, but as a result, applications targeting the Windows Win32 API are not portable to systems implementing the POSIX standard.

2.2.1 System Calls in Linux

The way system calls are working in Linux was already described as part of the general section. The exact mechanism, the calling convention but also the number of system calls is highly dependent on the CPU's architecture. While a 32-bit Linux kernel in version 4.8 for the x86 architecture offeres 379 calls, the 64-bit version for x86_64 offeres only 328[26]. The *man-pages* project documents give an overview (`man 2 syscall`) about architectural differences and the calling conventions. How far Linux is actually compatible to the POSIX standard is not only related to the kernel and the number of system calls itself, but for the most part to its abstraction layer, the used POSIX/C standard library. One of the most widespread ones, the *glibc* (GNU C Library) aims to follow POSIX.12008 amongst other standards[2] while the *musl* library does not implement it in complete[3].

2.2.2 System Calls in Zircon

In Zircon, system calls are bounded to the concept of *handles*, a construct which allows applications running in user mode to reference an object in kernel mode[47]. Interactions between user applications and kernel objects are still done using system calls but most of them are using a handle which describes the kernel object to work on[31]. Handles are checked by the kernel each time a system call is triggered. For additional security, the kernel checks whether

- a handle has the correct type for the system call,
- a kernel handle's parameters refer to an existing within the calling process's handle table and
- a handle has the necessary rights for the triggered action[31].

In contrast to Linux, Zircon provides just one library for system calls and the standard C implementation, the *libzircon.so*. It is a virtual Dynamic Shared Object (vDSO) directly provided by the kernel and not stored as a physical Executable and Linking Format (ELF) file on disk. For the reason that vDSOs are accessible from both, kernel

[2] gnu.org, visited on 15.06.2019 `https://www.gnu.org/software/libc/`

[3] repo.or.cz, visited on 15.06.2019 `https://repo.or.cz/w/musl-tools.git/blob_plain/HEAD:/tab_posix.html`

and user mode, without switching the context, they are a perfect concept to implement system calls in a very performant way[7]. Thus, the Zircon vDSO is the only way to perform system calls[30], which is a very elegant solution to cope with performance issues in a microkernel architecture.

The system calls are defined by using an abstract definition syntax and the matching tool *abigen* which generates header files and code for the libzircon's and the kernel's system call implementation[31]. Also in contrast to Linux, Zircon respectively Fuchsia does not aim for POSIX compatibility. It implements only a very limited subset of POSIX consisting of basic I/O operations and pthreads. Zircon does not support Unix-like signals, symbolic links and much more[22]. The libzircon.so does not support directly I/O operations. They are performed by the *fdio.so* library which overwrites weak symbols of libzircon[22]. All available system calls in Zircon from the version this thesis is working on are documented in `https://github.com/Allegra42/zircon/tree/i2c-grove-lcd/docs/syscalls`.

2.3 Processes and Threads

Modern operating systems are characterized by their ability to perform various tasks at the same time. So far, this fact has simply been accepted within this work, but it was not questioned what is special about it and how this multitasking is achieved by an operating system. In general, parallelization of tasks can take place on different levels, e.g. as hardware or software parallelism. While the physical resources for the first case are actually available to execute the tasks really simultaneously, it only seems to be so for software parallelism[17]. Hardware parallelism is realized by independent, maybe specialized execution units such as multiple processing cores or controllers which are able to perform particular parallel to the main CPU. This could be USB, network or graphics controller, for example[17]. In order to create the impression of parallelism on a single CPU and to use their available computing power as efficient as possible, an abstraction to provide pseudo concurrency for the execution of several tasks is required. According to Tanenbaum, this concept called *process model*, is the most central one of operating systems[49]. The term *process* is often defined as a program in execution[1]. The process model is complemented by the thread model which provides a simplified version for parallelism within a process or an application[17]. The following sections take a closer look at these concepts and answer the remaining questions, in particular about the process and thread model, synchronization mechanisms, inter process communication and their implementation in Linux and Zircon.

2.3.1 Processes

The term *process* describes an instance of a program in execution, including all its required resources to enable a model for the pseudo parallel execution of multiple programs based thereon[44], [49]. Each program in execution is modeled as a separate process which is assigned to a, from its point of view, private CPU including CPU registers, above all the program counter (PC) and a virtual private address space[49], [17].

Especially by using virtual private memory, processes are not only modeled individually, but are also isolated from each other in reality in order to prevent errors or deliberate attacks between programs[4]. But since in reality more processes, respectively programs, have to be executed than CPUs exists, a mechanism is needed to switch between processes and allow the execution of all them. This mechanism behind, the *multiprogramming model* (formerly also known as *time-sharing model*), and the policies called *process scheduling* will be considered hereafter.

However, for a change between the execution of programs to succeed, a process must contain information about its state at the moment it is interrupted to run another one. Which information needs to be stored and the way it is done slightly depends on an effective system's design and its implementation. In general, this information is stored in a structure called *process control block* in literature[49]. Depending on the implementation, it contains for example

- a process id,
- the process's state in the process lifecycle (described in the following section),
- the CPU's register contents, especially the program counter (PC), the stack pointer (SP) and the processor status word (PSW),
- the process's address space including the program code, its stack and heap and data segments,
- information about resources allocated by the process like open files, or I/O devices,
- CPU scheduling and accounting information like a priority, the maximum and actual computing time for this process,
- a reference to the parent process and
- the process's rights[49], [17], [1], [44].

Process Lifecycle

The state in the list above refers to a process's lifecycle. It is quite easy and in most cases described as a state chart like pictured in figure 2.6. Once a process was created, it is managed by the operating system and changes its state to *ready*. The process is ready to run, but has to wait until the system allocates a CPU to it. Is that the case, the process changes its state to *running* and performs its calculations. If a process is interrupted (preempted) by the operating system without having fulfilled its task completely, there can be two reasons for this which result in different subsequent states. Was the process interrupted just to run another one, the process's state changes back to *ready*. It only lacks CPU time to run the process again. If the reason was instead that resources - like the process is waiting for an external event or required I/O devices - are used by another process, then it changes its state to *waiting* or *blocked*. Only

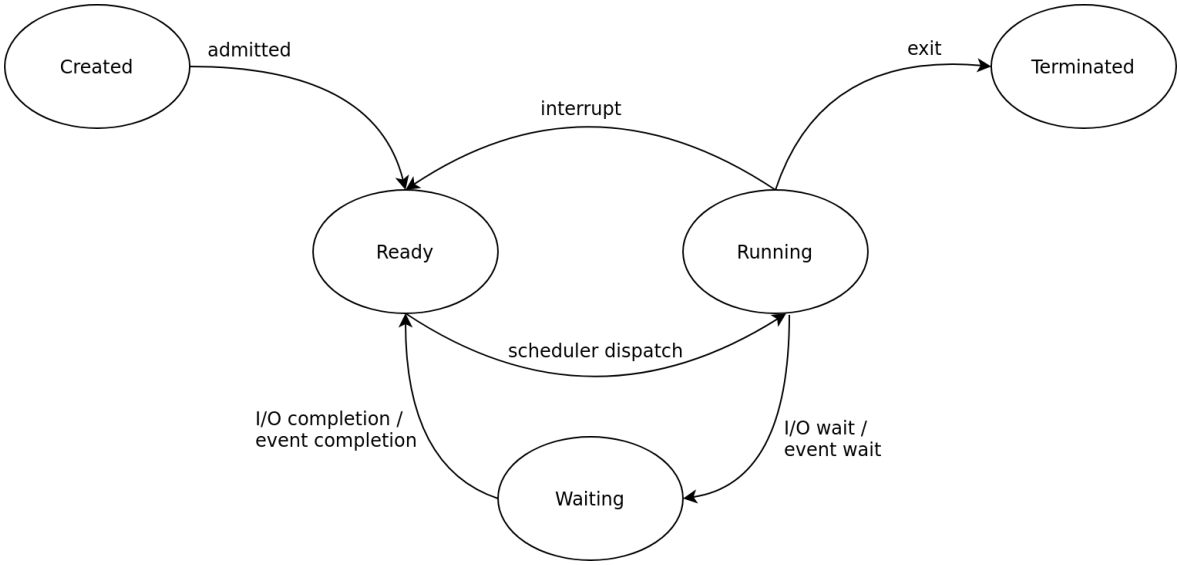

Figure 2.6: Lifecycle of a process according to [44]

when the blocking resource needed by the process is available again, the operating system changes the status of the process to *ready* again. The process is prepared to get reassigned to a CPU. A running process finishing its task during its computing time has reached the end of its life, its state changes to *terminated*. The operating system destroys the process and frees related resources[44], [27]. The model described applies to each individual process. For the coordination of the entirety of processes, the scheduler, an operating system kernel's component is responsible. It is a good example for the separation of mechanism and policy. The mechanism behind is based on the *multiprogramming* model as the switching of the CPU between programs is called. The policy, that is the strategy to decide which process should run next, should be specified apart from the mechanism. It not only allows the impression of parallelism on a single processing core, but also increases its utilization, as many processes spend a lot of time waiting for external events such as keyboard inputs[49].

Principally, the mechanism behind scheduling does not differ fundamentally from the treatment of an ordinary interrupt. First, the Program Counter (PC) is saved prior to the new PC loading from the interrupt vector and the CPU's registers are saved. After a new stack was setup, the actual interrupt service runs before the scheduling policy decides which process is to run next and the selected one is started[49].

Process Creation

There are several reasons why a new process should be started and scheduled at all. The most basic one is the systems start-up. To initialize an operating system, numerous processes are created to execute parts of the system itself. The user perceives very few of them directly, since most of them are *background processes* performing specific tasks like accepting incoming mails. Often long running processes without user interaction are referred to as *daemon processes*[17], [49]. They are mostly generated from the *init*

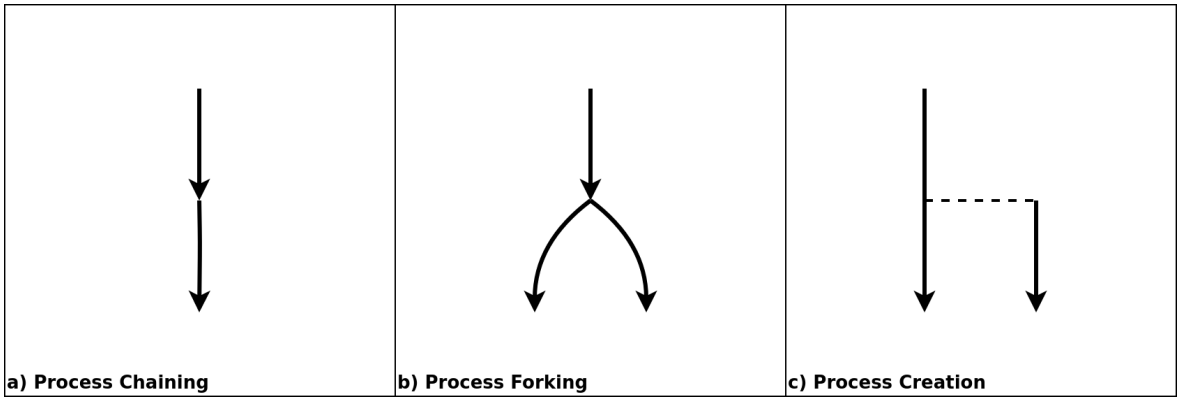

Figure 2.7: Ways of Process Creation according to [17]

process, the first one running and bringing up the system. Foreground processes, the ones a user interacts with, are more often started on behalf of the user. Besides, a process can also be started by a system service call from an existing process. The last option that a process is started to execute a batch job is rare today apart from scientific high performance computers[49], [17].

Also for the way how a new process can be started there are several options. Using *process chaining*, a running process starts an independent new one, a new Process Control Block (PCB), and destroys itself[1]. This option is pictured in figure 2.7, a. An example in Unix-like systems is calling `exec()` on an already running process to replace the current program with another one[17], [49]. By *forking a process* (figure 2.7, b), a second one is created that is, at least in the beginning, a copy of the original one. Both keep exiting and share the same environment such as program code, address space and resources[49]. As a result, both processes have access to exactly the same resources like opened files or I/O devices[1]. The Unix implementation of `fork()` is an example for this way of process creation. Process *forking* and *chaining* are usually used combined in Unix-like systems to create a new, independent process which is running in parallel to the first one[49]. In contrast, Windows offers a third way of *process creation* (figure 2.7, c) which combines them in a single instruction to start a second, independent process (`CreateProcess()`)[17]. As the newly created process and its parent share the same program code, the new one can simply inherit all needed data from its parent like it is done by *process forking*[1]. Each process has its own virtual address space if it is not shared through inheritance.

As a result, the data cannot be copied between processes without further ado. This is not only necessary for the multiprogramming model, but also a security mechanism that encapsulates and protects applications in execution against each other, called *process isolation*. In this case, i.e. for the one created via *process creation*, an operating system needs a mechanism to pass data to the created process. As a solution a mechanism for the communication between processes with which the parent process communicated the new data to the child via a buffer is just as conceivable as the use of initial parameters which are transmitted during the process creation, e.g. via a system call[17]. The first option is used by the `popen()` call of Unix-like systems while

the mechanism behind the *IPC* plays a major role in modern operating systems and therefore is treated in a separate section. The second one is more often used for *threads* which are the topic of the following section.

Process Termination

Reaching the end of its lifecycle (see figure 2.6), a process terminates itself regularly and voluntary (*normal exit*)[49]. In addition, a process can terminate itself prematurely for various scenarios or be terminated by the operating system or other processes. For example, a process may detect an internal error and voluntary terminates itself ahead of time or the operating system detects such an error and terminates the process involuntary to avoid major damage[49]. Besides, a process may get involuntarily terminated or killed by another process. Killing another process is usually a privileged task which requires an authorization or extended rights[49].

2.3.2 Threads

The current process concept, as previously introduced, provides only a single thread of execution. However, this an issue as soon as a resource needs to be edited in parallel. If a user edits a file for example, the process has to wait for the user's input repeatedly. It is nearly impossible to check this input for errors at the same time, since a second process is not allowed to access resources allocated by another process without an explicit action of programmers (like explicit Inter-Process Communication)[49], [4].

The *thread model* should solve this issue. Its basic idea is to equip a process with several execution threads that work on the same resources running in quasi-parallel[49]. This means processes only group the resources together while *threads* represent the actual execution units on the CPU[49]. A program is still abstracted as a process while (parallel) sequences within the program correspond to threads. As a result, *threads* can be considered as slimmed down processes sharing the same address space and physical resources. Each thread has its own PCs, register set, stack and state, but the address space, global variables, open files and accounting information are shared[49]. A thread's lifecycle is equivalent to process' ones, but in contrast to them threads are not isolated and protected against each other[17]. As they share the same address space, one thread can read, write or even destroy another thread's private stack[49]. Another issue are competing threads that try to write the same global data. They cause a *race condition* which possibly results in inconsistent or wrong data[4].

Nevertheless, the advantages of using *threads* (multithreaded programming) predominate. The most important one is resource sharing. The fact that multiple threads are using the same resources by default, e.g. files, in pseudo parallel are the biggest advantage but at the same time the biggest problem of this concept. Threads enable responsive, interactive applications and increase the performance especially on multiprocessor architectures in that they can run truly parallel[44]. Another reason is an economic one. While allocating an address space and physical resources to create a process is an expensive operation, creating a thread is not as they inherit exactly these components from their corresponding process[44], [27]. Threads only need to set up

their own PC, registers, stack and a state within a process. That's why they are also known as Light-Weight Process (LWP)[27].

How *light-weight* they actually are depends on a lot on their implementation in an operating system. Conceivable options are thread implementation in userspace or kernel space, but also hybrid ones. Implementing them as user library requires an own mechanism to schedule threads within a process which is often realized as a kind of runtime environment including a Thread Control Block (TCB) (corresponding to PCB) for each single process[27], [49]. There is no need to invoke the operating system to switch a thread, but as a result all threads within a process block become blocked if a single one waits for a resource[49], [4]. The operating system's schedule just do not know about the runtime environment and possible other runnable threads within a process. The implementation in kernel space is not that light-weight, but multicore architectures benefit more from this variant. If the operating system's kernel has the control over threads, these are managed just like processes. Instead of one TCB within each process, the kernel collects the TCBs for all threads in the entire system in a thread table corresponding to the process table[49], [27]. But this also allows the kernel to recognize a blocked thread and schedule another one of the same process instead of blocking the whole process[49].

2.3.3 Processes and Threads in Linux

As in the general model, a process in Linux corresponds to a program in execution. Initially, a process contains exactly one thread of execution but further ones can be created as soon as the process has started[49]. Processes are organized in a tree-like structure with the so-called *init* process on its top. Starting from init, each new process to be created is split off from its parent using the *fork()* system call. Forking a process, the newly created child process inherits the whole environment of its parent including environmental variables, opened files and network connections. Furthermore, it gets a copy of its parents address space including the data and code sections. As both processes share the same data and resources at this time, synchronization is needed if both try to access the same one[27]. This issue is considered in the following section 2.4.1. The distinction of parent and child is done via their Process Identifier (PID). The fork call returns 0 to the child and a non-null PID to its parent. Based on this, the execution paths can be split up[49], [27]. Even if parent and child differ only by the PID directly after the fork, changes in the parent are not visible to the child and vice versa. To take advantage of this fact to actually use the newly created process independently of the parent, in Unix-like systems the fork call is in most cases directly followed by the call *exec()*, which replaces the complete process environment with the one from another, new program. As copying the process data is quite costly and needless if an exec follows directly, modern Linux systems use a *copy-on-write* strategy for process forking. Children get their own page tables pointing to the parent ones. The child still can read the parent's environment data without a copy operation needed. Only when the child or the parent tries a write access on this data, a *protection fault* is thrown and the page to modify is copied to the child process[49]. Such a fork

is known as *vfork()* and used especially in situations an *exec()* follows directly to avoid the expensive and needless copy of the parent process's environment[27].

Within the Linux kernel, the difference between processes and threads (or heavy-weight and light-weight processes) is not as important as described in the general section about processes. It is rather spoken of *tasks*. The reason for this is that the Linux kernel offers a possibility for a fine-granular process respectively thread creation via the *clone()* call on kernel level. In addition, Linux respects POSIX threads[26]. But as the clone call is a unique feature of Linux and not generally available on other Unix-like systems applications using the clone call directly are not portable[49].

Nevertheless, *clone()* is a very interesting concept in Linux. In every case, a process or thread is created with clone it gets an empty private stack and executes directly a new program which is given as an argument to the call. The decision whether a process or a thread should be created, which resources should be shared or copied and which process is actual the parent are based on the following flags:

- if `CLONE_VM` is set, a new thread is created, if not, the call results in a new process.
- if `CLONE_FS` is set, the newly created thread or process shares *umask*, *root* and *working directories* with its parent. They are not shared at all if the flag is not set.
- if `CLONE_FILES` is set, the parent shares file descriptors with its child. If not, they are copied to the child.
- if `CLONE_PARENT` is set, the child's parent is the same as the calling process ones. If the flag is not set, the calling process becomes the parent[49].

The flags listed show only a partial amount of the possibilities *clone()* offers. A full list of the glibc wrapper to the corresponding systemcall is available as a man-page using `man 2 clone`. Both, processes (tasks) and threads based on the clone call are kernel constructs. The POSIX thread implementation on Linux internally also uses the clone call with the special flag `CLONE_THREAD`.

In contrast to the process lifecycle presented in the generic section, Linux tasks have one extra state called *zombie* which is entered on process termination until the parent process is informed about[27].

2.3.4 Processes and Threads in Zircon

Similar to Linux, runnable entities of the Zircon kernel are called *tasks*. This term includes the kernel objects *jobs*, *processes* and *threads*. As the whole Zircon kernel is object-based, the user interacts with kernel objects like the ones mentioned above via handles. The Zircon documentation describes *handles* as "kernel constructs that allows user-mode programs to reference a kernel object"[38], containing the reference to this object, the corresponding rights and the user-space process it is bounded to. Thus, a handle can reference each object type listed in the *Zircon Kernel objects* reference[25], including kernel objects for drivers like *interrupts*, *resource* objects and *Log* objects.

The Zircon systemcall `zx_status_t zx_task_kill(zx_handle_t handle)` is such an interaction between a user application and the kernel. It refers to a *task* object handle and thus to all objects for which *task* is a generic term[20].

Jobs as an organizational unit are Zircon-specific and not known from the general or the Linux-specific section on processes and threads. A *job* manages a group of processes but possibly other (child) jobs, too. Corresponding to Linux, jobs build a tree structure as every process must belong to a single job but jobs can be nested. Except the root job, which corresponds to the init process on Linux, each one has only one parent. A Job object consist of a reference to a single parent job, a number of child jobs and a set of member processes. In future[4], they will also contain a set of policies[36]. Jobs are used to track the privileges needed to perform kernel related operations like systemcalls but also to track and limit the consumption of basic computing resources such as memory, CPU time or I/O devices[36]. The idea behind this concept enables managing applications composed of more than one process as a single entity, both from a resource and permission view as well as from lifetime control[38].

A Zircon *process* itself is an instance of a program as defined in the general section about processes 2.3. It consists of the program code as a set of instructions to be executed by one or more *threads* and a collection of resources. Threads are just as much a part of a process' resources as *handles* and *Virtual Memory Address Regions (VMARs)* are[38]. Strictly speaking, it is not necessary to mention VMARs as an own point. They are kernel objects itself. But as the documentation mentions them explicitly, they should also receive special attention at this point. A VMAR represents a contiguous part of virtual memory address space used by the kernel as well as by the user-space. Each process starts its own VMAR to build up its address space. VMARs have a hierarchical permission model, so a process with a read only address space cannot create a readable and writable one, and are randomized per default[14].

The lifetime of Zircon processes differs from the general but also from the Linux model. In Zircon, a new process is created via `zx_process_create()` but the execution starts not before `zx_process_start()` is called. Was a process already started and its last running thread exits, it is impossible to create a new thread within this process. Processes which are composed of a job are threatened as a single entity from a lifetime control's point of view. The lifetime of Zircon processes (or jobs) ends if

- the last thread within the process or job exits or is terminated,
- the process or job itself calls `zx_process_exit()`,
- the parent job terminates the process or
- the parent job is destroyed[38].

Just as described in the general section Zircon *threads* are the actual runnable computation entity living within a process. They are created within the *process* context

[4]Unless otherwise noted, the status of the Zircon documentation cited in this thesis corresponds to the forked and frozen source code repository on which basis the driver development takes place. See `https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/`.

via `zx_thread_create()` but as it is done with Zircon processes, the actual execution is not started until `zx_thread_start()` **or** `zx_process_start()` is called. The `main()` function or entry point of an application should be the first one started via `zx_process_start()`. But returning from such an entry point does not terminate a thread's execution. This must be done manually by voluntarily calling

- `zx_thread_exit()`,
- `zx_vmar_unmap_handle_close_thread_exit()`,
- `zx_futex_wake_handle_close_thread_exit()`

on the thread itself[39]. Even if the last handle to a thread is closed, the thread is not terminated automatically. Instead, the thread must be killed explicitly after the handle was restored via calling `zx_object_get_child()` on the parent. On the other hand, a thread can be terminated involuntarily

- if the parent process is terminated,
- if someone calls `zx_task_kill()` on the thread's handle or
- if an exception was generated for which there is no handler or the handler terminated the thread[39].

In contrast to Linux threads and some library thread implementations, Zircon threads are always *detached* from each other. An operation like `join()` which waits for an undetached thread to complete and allows a clean termination is not needed in Zircon itself. Libraries and runtime environments on top of Zircon may require such an operation to reach e.g. POSIX compatibility[39].

2.4 Synchronization and Inter Process Communication

The term Inter-Process Communication (IPC) is not only used for the pure communication between different processes but also for the synchronization between processes, threads and data shared between them. As already mentioned in the previous section 2.3.2, the ability of threads to access shared data has not only advantages in this context. As they share storage, e.g. main memory, two threads can read and write the same value. But what happens if both of them try to access the same, shared value? Two threads try to read and modify a shared value are given for example. Both read the value and calculate based on the read a new one to store. There is no problem as long as they just read the same value, but it occurs on updating the value. Maybe the first thread is scheduled first and updates the value, directly following the second one is scheduled and updates the value based on the value read before the first thread was executed. In this case, the update of the first thread is lost. The data possibly becomes inconsistent. Such a situation is called *lost update problem*[17]. Unless the scheduling

of the threads is predictable in each situation, it remains impossible to predict the exact outcome. Therefore, one speaks of a *race condition*[49]. The section 2.4.1 discusses corresponding problems, especially for threads, and indicates ways to deal with them. Since it is not sufficient to allow only the exchange of data respectively communication between threads of a single process, section 2.4.2 shows mechanisms that overcome the process isolation and enable the communication between multiple ones.

2.4.1 Synchronization

In the context of this section, it should be assumed that *threads* are implemented as a part of the operating system's kernel rather than based on a runtime system. According to the definition, the term *thread* refers in this section to both, execution threads within a single process and the ones within independent processes. A competitive situation can occur for both variants. The threads of one process rather rival regarding a shared variable while the threads of different processes rather compete for resources such as a printer. Race conditions are a topic each time several processes or threads work on the same data in any way. This also applies to an operating system's kernel, except for non-preemptive kernels which can guarantee that only one thread is active at a time on a single-core system[44]. In contrast to preemptive kernels, non-preemptive ones do not allow a thread running in kernel mode to be interrupted. The thread runs until it exits kernel mode, blocks or yields the control of the CPU voluntarily[44]. In such a case it can be guaranteed that a kernel and its data structures are free of race conditions. However, preemptive kernels are more common because they are more responsive and better suited for real-time tasks due to their interruptibility[44]. Furthermore, with today's multicore systems it is rather unlikely to fulfill the requirement of only one active thread in kernel space. For this reason, a mechanism to prevent race conditions on both, user and kernelspace, is needed. The basic idea behind is to prevent the possibility that two threads try to change a shared resource at the same time[49]. But only certain areas in the program code are critical, the ones where a shared resource is processed. They are called *critical regions* or *critical section*[49]. It does not harm if other regions are interrupted. If only one thread at a time is allowed in critical sections and another one is not allowed to enter the region until the competing access is completed, this is called *mutual exclusion*[49], [17].

The easiest way to achieve *mutual exclusion* is to completely disable the system's interrupts immediately after entering a critical region and re-enable them just before leaving it. In the meantime, all incoming interrupts are collected and processed as soon as they become re-enabled[1]. However, this method is not suitable for modern multicore systems since the interrupts can only be locked for the current CPU core. The competing thread can still modify the resource if executed on a different core. This solution is not ideal for single core systems, too, because even the clock interrupt is disabled and with this the process scheduling. Does the currently running thread not re-enable the interrupts, the whole system is blocked[49].

While there are some approaches to pure software solutions such as the algorithms of Peterson or Dekker (see [49] or [44] for further information) hardware enabled

solutions are common today[49]. Modern multicore CPUs usually offer an instruction which is referred as Test and Set Lock (TSL) or Test and Set (TAS) in literature. It is an atomic, not interruptible operation, usually used to modify a shared variable which controls the access to a shared memory region[49]. The atomicity of a TSL instruction is in common achieved by locking the memory bus to prevent other CPUs from accessing the memory until the operation is done. As a big advantage of this solution, the CPU cores are not obstructed. Common calculations are not impeded but memory accesses are prevented[49]. A variant is the Exchange (XCHG) instruction which exchanges the content of two memory locations in one atomic operation[44].

All mechanisms mentioned so far have one problem: they require busy waiting. The thread waiting is still active and waists CPU time. For short waits, this is perfectly fine. Switching to another thread and back would be more expensive in such a situation[17]. Thus, there are locking mechanisms that implement busy waiting very efficiently, called *spinlocks*[49]. But longer active waits are very inefficient. In this case, it is better to use blocking waits and bring another thread in execution until the reason for the blocking is removed, e.g. the desired resource is freed again.

Semaphores and Mutexes

Dijkstra suggested in 1965 a possibly blocking lock mechanism called *semaphores* based on a CPU's TSL instruction and easier to use for application developers. A *semaphore* is a new integer type with two related operations called P and V in the original paper or *down* and *up* in some literature and implementations[17]. The semaphore is initially initialized with a value greater 0, the exact one depends on a system's implementation or can be defined by a developer. If the P or *down* operation is executed on a semaphore, it is checked whether the value is greater than 0, decrements the counter if it is the case and continues. If not, the thread is put to sleep. The operation remains unfinished until a V or *up* operation was executed and incremented the semaphores value. One of the possibly multiple threads sleeping on a semaphore is randomly or by a certain rule chosen by the operating system and gets the clearance to complete the P or *down* operation[49]. For a semaphore, the fact that updates on its value must be performed in an atomic operation is essential[44]. Neither the decrementing of the value performed in the P operation nor the incrementing in V may be interrupted.

Semaphores are a generic approach to control access to a *critical section* for a number of threads, but the often required mutual exclusion is only given if the semaphore is initialized with the value 1. This particular case of a binary semaphore is also referred to as *mutex*. A *mutex* guarantees mutual exclusion for a critical section protected with its related operations as *lock* (*P, down*) and *unlock* (*V, up*)[49]. Except for the initial value and the operation's names, mutexes do not differ from semaphores. Just like semaphores, they can be implemented using the TSL or XCHG instruction.

But if semaphores are actually blocking depends on the implementation. As mentioned earlier a blocking mechanism is not always advantageous, e.g. for very short waits or multicore systems with real parallelism. In these cases, its is potentially more efficient to use busy waiting with spinlocks[17].

Futexes

Another variant of semaphores is the fast user-space mutex (futex). It targets the issue that neither busy waiting nor block and reschedule another thread is very efficient on modern multicore CPUs. On a very parallel system, there are many contentions for resources which would require frequent switching of the active threads, but scheduling another thread respectively process requires as well expensive switches to the kernel[49]. Futexes are not a pure userspace construct even if the name suggests. They still need a wait queue in kernel to manage the contending threads waiting on a lock, schedule another thread (futexes are blocking by design) and putting a thread into the queue requires a system call as well. A user library tests a lock variable using a TSL instruction. If the lock is already held by another thread, the library has to put the thread into the queue. But if there is no contention there is also no need to involve the kernel[49]. The less actual competitive situations occur, the more efficient are futexes compared to common mutexes. Mutexes are for example used as part of the Linux kernel. In this context, Ingo Molnar published an article[5] as part of the Linux kernel documentation covering the implementation of robust futexes and its impact on the system performance.

Barriers

So far, the synchronization of resource accesses has been discussed. Sometimes, this is not an issue but to coordinate and to synchronize the sequence flow of two or more threads. If threads within an application are divided into self-contained phases, it must be ensured that the next phase is not started until the current one has been completed by all threads involved[49]. The synchronization points where the threads have to wait for each other to enter the next phase are called *barriers*. An example for such a situation could be cooking. The stove must not be switched on until all ingredients have been purchased. In contrast to the general implementation of barriers, a common real world stove does not block itself should this not be the case. In most instances, barriers are implemented on the basis of blocking semaphores[17].

2.4.2 Inter Process Communication

While the previous section 2.4.1 focuses on threads, this section explicitly deals with the communication between different processes and their synchronization. As IPC is a central element in operating systems, especially in microkernel architectures, which significantly influences the system's behavior and its performance. Both of them are crucial aspects in each operating system and thus, their efficiency is a decisive factor for a systems acceptance.

[5] git.kernel.org, visited on 15.06.2019 `https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/Documentation/robust-futexes.txt`

Memory Based Communication

Since the communication of threads of one process via its shared address space is already known, it is obvious to think about a comparable technique between processes as well. But in contrast to threads within a process, the concept of process isolation must be considered here. As a security feature, it is not possible for processes to share their address space. The basic idea to implement a corresponding idea nevertheless involves the idea of using an *external shared memory region* which is requested from the operating system. It allocates the memory and shows it in the address space of both processes[4]. The shared memory region is the responsibility of the processes, must be managed and secured against incorrect access by themselves[4], [17]. As known from thread communication, the advantages of this solution are the performance and transparency of raw memory access, but it also needs additional (manual) synchronization to obtain and maintain consistent data[17]. Nevertheless, this solution breaks the isolation between processes. A variant of this idea is using *files* or similar resources as a communication medium. In this case, the process isolation remains, but manual synchronization, as described in the previous section, is still necessary[4].

Message based communication

Message based communication methods are more wide-spread. In contrast to the memory based ones, the synchronization usually does not have to be done manually. The message exchange and with it the synchronization are provided by the operating system respectively its kernel via two primitives: *send()* and *receive()*[49].

Basically, there are three types of messages:

- A *Message* is characterized by a delimited amount of data within a single communication.
- *Streams* can theoretically transport an unlimited amount of data. In practice, there is a limitation not visible for the sender and the receiver.
- *Packets* organizes the data to transport in standardized formats, the communication protocols. They allow fragmented transfer of a large quantity of data which is defragmented on the receiver side. The system implementation hides this fact, so the packets itself are not visible for an application[17].

Regardless of the type of message, there are two basic operations available for message-based communication: *send()* and *receive()*. These operations are commonly supported by the operating system and enable not only the system-wide but also the cross-system exchange of messages and data[17]. Especially the opportunity of cross-system communication offers a significant advantage over memory-based communication, as this is the foundation for network communication. But also the synchronization between the involved processes is simplified. The concept of messages avoids race conditions and makes the manual use of semaphores for application developers obsolete,

as the operating system realizes the transmission of the messages and their synchronization. However, this concept is not as efficient as memory-based IPC concepts in terms of resource consumption and performance[17].

Synchronous and Asynchronous Communication One of the most basic design decisions within message-based communication is whether to send or receive messages synchronously or asynchronously. When transmitting messages *synchronously*, both sender and receiver must be ready for the exchange at the same time[17]. Is this not the case for one instance of them, e.g. the receiver, the other one (the sender) must block until the first one, the receiver, becomes ready. An additional, manual synchronization is not necessary. Even if the literature does not describe a general way to transfer messages between processes, race conditions cannot occur as long as a date is only sent from a writing process to the address space of an initially reading process and can only be modified there[4], [17]. However, within the individual processes, the synchronization corresponds to the one known from threads (see 2.4.1). But the close coupling of the processes within the synchronous communication leads to the fact that the program flow of both is no longer possible independently of each other. Accordingly, the parallelism for processes in such a relationship is also limited[4].

The *asynchronous* approach tries to decouple this by inserting a buffer between which is also known as *mailbox* or *message queue*. So sending is possible even if the receiver is not available at one moment, as long as the buffer is not completely filled. The receiver can read equally if the sender is not ready, but the buffer still contains messages. Even different processing speeds between the processes involved can also be compensated in this way[17], [4]. Only if the buffer is completely filled, the sender has to be blocked and vice versa the receiver if the buffer is empty. But an operation only blocks when running in a *blocking operation mode*. In a *non-blocking operation mode*, a call returns immediately in such a situation but reports a full respectively empty buffer which has to be handled on application level[17]. *Asynchronous* communication allows an independent program flow and simplifies parallelism thereby. Nevertheless, an additional buffer with limited size is needed. The exact one depends on the implementation. If this buffer is filled and the receiver is blocked, the situation is the same as before with synchronous communication. But the main problem with this approach is that it is not possible to predict how long a message transfer takes and when a response can be expected[17]. The issue is to decide whether a message is lost, the communication only takes a long time or the receiving process has crashed[49].

Connection-Oriented and Connectionless Communication Especially when the reliability of data transmission has a great significance, a *connection-oriented* approach for message-based IPC is chosen. This allows to guarantee the message sequence, monitor the transmission time of them and to intervene if necessary[17]. Until a (logical) connection or *channel* between the communication partners has been established, the initial message receiver must be unambiguously known. As soon as this is the case, the identification is not longer of relevance. The established channel provides a reliable end-to-end transmission which may have the following properties:

- An *unidirectional* connection allows message transfers only in one direction at a time.
  - If the roles of sender and receiver never change, one says the transmission is *simplex*.
  - If the participants act alternatingly as sender or receiver, the transmission is also called *half duplex*.
- An *bidirectional* connection allows message transfers in both directions. Both participants can act as sender and receiver at the same time. The transmission is *full duplex*[4], [17].

If a connection is no longer needed, it must be disconnected and destroyed. The construction and dismantling of a connection means an additional and noticeable overhead especially for a small amount of data to transmit. It is only worthwhile if the reliable transmission between designated instances is more significant than the connection costs[17]. One example for connection-oriented communication is a telephone call.

But there are also good reasons for *connectionless* communication. For example if the overhead of the connection setup is too costly for little data to be transferred or rather few data is sent to a large amount of receivers. A real world example for *connectionless* communication is radio. The order of messages cannot be guaranteed for this connection type and also the loss of them is possible, but this has not necessarily further impact on the application. For a multimedia application, a retry could be a greater damage than the package loss. But for other application purposes, a retry is maybe needed and must be triggered from the application itself[17], [4]. Nevertheless, the recipient must be addressed anew in every single message.

Receiver Addressing Message receivers respectively their message queues are commonly addressed by a symbolic name which is managed in a system-wide or cross-system directory. When addressing them, one speaks depending on the number of receivers and the liability of the transmission of:

- an *unicast*, if there is a 1:1 relationship between sender and a certain receiver addressed.
- a *multicast*, if there is a 1:m relationship between sender and a defined group of receivers addressed.
- an *anycast*, if there is a 1:1..n relationship between sender and group of receivers. At least one recipient from the group must receive the message.
- a *broadcast*, if there is a 1:n relationship between the sender and all detectable receivers, whether the message actually arrives does not matter[17].

Priority In literature, asynchronous message-based IPC mechanisms are usually modeled without prioritizing messages. A buffer, message queue or mailbox that works according to the First In First Out (FIFO) principle is suitable for this purpose. In order to enable a prioritized message exchange, it is, iter alia, conceivable to use separate buffers or queues for each priority[17], [49].

2.4.3 Synchronization and IPC in Linux

Basic Synchronization

Modern Linux based operating systems provide a wide range of basic thread synchronization mechanism from a user's point of view. Examples are POSIX realtime semaphores, POSIX mutex and System-V semaphores[17]. The underlying implementation in Linux versions starting from 2.6 is based on the concept of futexes to provide an efficient locking mechanism[49]. While in userspace the grown structure of Linux, its openness, but also standards like POSIX allows finding nearly every variant of basic synchronization mechanisms, the kernel itself is much more limited.

Within the actual Linux kernel, only *spinlocks* and *mutexes* are available besides *atomic data operations* like `atomic_set()`, `atomic_inc()`, `atomic_dec()` and others[1]. As mentioned in the general section 2.4.1, a spinlock uses busy waiting while the kernel's implementation of mutexes allows to schedule another task. Both are blocking binary semaphore versions that only differ in terms of waiting behavior[43]. Linux' versions compiled for single processing core systems and without preemption support do not provide spinlocks. They are not needed in this case. Mutexes are still available to ensure a shared resource is only accessed from one process at a time. Preemptive single core systems provide spinlocks, but they only disable preemption during a critical region[43].

Signals

Signals are a special concept of UNIX and Unix-like operating systems, located somewhere between process synchronization and IPC. They are used to transmitting events or information between the kernel and processes similar to hardware interrupts[17]. The most known usecase of signals is to interrupt or terminate a process from another one, but actually they are also used for signaling and handling errors, I/O events or to trigger even user-defined actions[1]. Signals are usually sent between the kernel and a process, e.g. to indicate illegal instructions or memory access, arithmetic errors, expiring timers and much more. Only `SIGUSR1` and `SIGTERM` are sent between processes. Most of them are ignorable and/or handable by the receiving process, except from the `SIGKILL` and `SIGSTOP` signals[1].

Signal handlers are commonly used to gracefully terminate a process or handle the indicated situation. They provide a default behavior which can be overwritten to handle specific signals tailored to the particular process. In this case it is convention to raise the same signal with its default action again right after the custom handling

finished[1]. Each Linux process is aware of receiving signals, but if a signal is ignored, the result is not predictable in most cases.

Inter Process Communication

Pipes Pipes were one of the first IPC mechanisms in Unix-like systems besides shared memory between threads within a process and additional memory regions shared between processes. They provide a stream-based communication mechanism using datagram-typed messages. Pipes are also widespread in Linux and especially known for process chaining like it is done in the command line instruction `ps -A | sort | more`. This shell command takes the output of the first command, `ps -A` (show all processes), and uses it as input to the second one, `sort`, which is used as input for the last one, `more`, again. In general, pipes can provide unidirectional or bidirectional communication between processes but ordinary Linux pipes provide only a unidirectional but buffered communication between the *write-end* to the *read-end* of a pipe[17], [44]. A bidirectional communication requires two inverse pipes on Linux[44]. As a pipe is buffered, both processes are decoupled to some degree, which is influenced by the actual buffer size, a system and architecture dependent value. Using a pipe synchronizes the involved processes implicitly. A reading process is blocked on an empty pipe, a writing one on a full pipe[17]. Ordinary Linux pipes are not accessible by any other than the creating process. Thus, pipes are commonly used between a parent process and its children or between siblings. The parent creates a pipe which is inherited to forked children[44]. For further reading, Achilles presents a good introduction to the implementation of Linux pipes and their usage in his book[1].

Named Pipes (FIFOs) Named pipes or FIFOs are an enhancement to the ordinary pipes described previously. In contrast to them named pipes are even available between unrelated processes. Several writers or readers on a single pipe are possible. Indeed, named pipes are capable of a bidirectional communication in half-duplex mode, too. But in fact, bidirectional IPC is done via two distinct pipes[44]. Named pipes are a special type of file which can be created with a freely selectable name at any point in a filesystem. A writing as well as a reading process can open and use a named pipe by its given name just like an ordinary file[17]. Once, a named pipe was created, it exists until it is explicitly deleted. The termination of the process which created the named pipe influences the pipe just as little as a system reboot[44]. This is not the case for ordinary pipes as they are bound to the creating process and its children. Both, ordinary and named pipes are only available between processes on the same system in Linux[44].

Berkeley Sockets While both types of pipes are bound to a single system, a corresponding mechanism for cross-system communication respectively network communication is needed. The best known mechanism to do this in Linux and other Unix-like systems are Berkeley sockets. They provide cross-system communication comparable to pipes[17]. Sockets are *communication endpoints*, put on top of the Transmission

Control Protocol (TCP) (connection-oriented) respectively the User Datagram Protocol (UDP) (connectionless) as a transportation layer and identified by an Internet Protocol (IP) address together with a port number[44]. This work will not examine the transportation layer and physical networking more closely. Using TCP, the resulting socket is called *stream socket*. The resulting communication is stream-based. In contrast, using UDP as transportation protocol, the socket is a *datagram socket*[44]. A communication between two processes is implemented using an own socket from the same type for each one, working according to a client-server model. Sockets provide a bidirectional communication between the processes owning the endpoints with the basic operations `send()` and `recv()`. Connection-oriented sockets using TCP are additionally in need to establish a connection and shut it down as soon as it is not longer needed[17].

2.4.4 Synchronization and IPC in Zircon

Synchronization with Futexes

In Zircon, all low level synchronization mechanisms are based on futexes. High level implementations of mutexes and semaphores such as `pthread_mutex_t` are built on top. Thus, normal users and user applications are rarely directly in touch with futexes[31], [33]. The Zircon futex implementation is very efficient. It does not enter the kernel space or allocate any kernel resources in the uncontended case. In contrast to the Linux futex implementation do Zircon futex operations key off of the virtual address of an userspace pointer while Linux differentiates private futex actions from the ones shared across the address spaces of more than one process. Further, Zircon futex operations do not modify the futex from the kernel, such an action is not required in the current implementation[33].

Events and Event Pairs

Events are described as *signalable event for concurrent programming*[13] and *user signalable kernel objects*[13]. A variant of events are event pairs. They are described as *mutually signalable linked pairs of signalable objects for concurrent programming*[29]. Unfortunately, there is not any more documentation about them and their use at all, not even in Google's zircon master repository.

Signals

Signals in Zircon are e.g. used for channels to indicate it contains messages to read. They are information exposed to applications by waitable kernel objects. Each object can expose a number of signals including generic ones for all objects but also ones that are specific to the object type such the signal described in the example above. Signals can be observed by processes respectively applications by `zx_object_wait_one()` for example. It is possible to wait for specific signals on several objects[48].

Inter Process Communication

Shared Memory Zircon provides an object based model for shared memory which is built on top of its virtual memory implementation. It is randomized per default and does not differ in the way the address space layouts of kernel and userspace are handled[2], [31]. Except from permission issues, they are always handled the same[14]. A virtual shared memory object can be mapped into the address spaces of one or more distinct processes. The Virtual Memory Object (VMO) is readable and writable but it allows a *one-shot* mechanism as well. This mode hands the control over a VMO over to another process directly after it was created and initially written[31]. Its underlying idea is to avoid the cost of mapping them more often than necessary into address spaces. As a result, this mechanism allows very efficient IPC via VMOs as shared memory implementation[31].

Channels Channels are Zircon IPC objects, comparable with Pipes in Linux. But in contrast to Linux, they are bidirectional, ordered message queues with two handles, each one refers to one end of the channel. Channels provide a datagram oriented message based communication between arbitrary and unrelated processes[37], [31]. The maximum number of handles referring to a channel respectively its ends and its message size are limited. In common, there are two steps in sending a message over a Zircon channel. At first, a message is atomically written into the channel. Afterwards, the ownership of all handles in the context of the message is transferred into the channel[37]. As a result, object handles which are written into a channel are removed from the sending process. Handles which are read from a channel become added to the receiving process. During the transmission within the channel, the handle still exists to ensure the life of related objects[31]. Threads can block on a channel until messages are available. Short reads or writes are not supported. This means, messages read from or written to a channel must fit, or they can not been read from or written to. In contrast to pipes in Linux, Zircon channels provide a special operation called `channel_call()` which bundles the common situation a message is sent in one direction and its response is awaited right after. The sending process dequeues the matching response identified by its first four bytes by itself[31], [37].

Sockets There are two main differences between Zircon channels and sockets. At first, sockets offer stream-oriented message passing instead of the datagram oriented one known of channels. Besides, sockets can not transfer object handles[31], [42]. Just as channels, sockets allow a bidirectional communication with two ends. The data is written to one handle and read from the other one. Both ends can fulfill both jobs, but they can also been shutdown for reading or writing[42]. Unlike channels, sockets provide short writes if the internal message buffer is almost filled and short reads if more data than available is requested[31].

FIFO The last IPC object Zircon provides, is a FIFO. There is not much documentation currently, neither in the usual documentation version used nor in Google's master

repository. FIFOs are described as IPC queues with an advantage on read and write operations. They are intended to be a top-level interface for shared memory based IPC which enables the performance advantages over channels and sockets. Nevertheless, are FIFOs on Zircon more limited on the size of elements and buffers than the previously named mechanisms[24].

2.5 Scheduling

The term *scheduling* was already part of this work in the context of processes and threads. It was used to describe situations in which the actual executed processes and threads must be replaced at certain intervals to provide the impression of parallelism. In most cases, there are much more processes ready for execution than the actual number of available execution units, the CPU cores. Although this work assumes modern multicore CPUs and thus real parallelism, the number of processes in interactive or realtime operating systems is much higher than the number of cores, too. A mechanism to share the resource *computing time* between all processes being in need of it is required accordingly.

Scheduling is generally used as an umbrella term for the actual *scheduling*, i.e. the planning of the process execution and allocation of resources like computing time on the basis of certain criteria, and the *dispatching*, i.e. the mechanism which provides the assignment of a process and its data to a CPU core and executes their exchange[4], [27]. Since the concept of scheduling is already divided into strategy (policy) and mechanism, it is obvious to do this also in the implementation. The mechanism of the dispatcher is familiar from context switching. Except changing the CPU's operating mode, pure process switching follows the same rules. Basically scheduling can refer to processes in the whole or individual threads[27]. The differences and effects of both variants have already been discussed as part of the section 2.3. Scheduling strategies presented as part of this section apply to both, regardless of whether they are applied on processes or threads managed by an operating system or threads managed by a runtime environment. Thus, it is often generally referred to as job scheduling[27]. Furthermore, only *dynamic scheduling* is considered in the following. In contrast to *static scheduling*, changing tasks, as known from interactive or dialog driven systems, can be taken into account there. Operating systems with a fixed and static scheduling list which are setup in advanced and not changed during runtime are rare[4]. In addition, a distinction is made between *long term scheduling* which describes the management of all tasks arriving in an operating system, especially the skillful execution of long-running jobs at times the system is not actively in use, and *short term scheduling*, the strategy to assign the processes which are currently ready to run on a CPU core[4], [27].

Scheduling goals

Having the common case that more processes or, more generally spoken, jobs are ready to be executed, there is a need of a policy to find the next one to run. Such a strategy is influenced by various design goals, depending on the operating system's

type, its purpose and the actual application[27]. The goals which can be included in an individual strategy are as follows:

- *Fairness*, i.e. a minimum computing time is guaranteed for each job and no one should be preferred if not explicitly part of the strategy.
- The *CPU utilization* should generally be as high as possible. It is the most limited resource in a computing system.
- *Minimize response times* is especially important in interactive systems as a delayed reaction to a user input is more noticeable than a longer runtime of an already long-running background job.
- *Minimize wait times* could also be an interest. But this fact depends not only on a scheduling strategy and the related resource synchronization but also on I/O operations not directly influenced by a job scheduler.
- *Maximize throughput* refers to the number of settled jobs per time.
- *Minimize the turnaround time*, i.e. the complete time a job needs for its execution. As with the related point "minimize wait times", a scheduling strategy only impacts the time in a waiting queue.
- A *Predictable process execution* is especially a requirement for the use in realtime operating systems[27], [4].

But there is no single scheduling strategy that fits all situations and goals equally well. In some cases, the goals named above even contradict each other. However, the decoupling of mechanism and policy enables the user to change the latter one according the current needs and interests[4].

Scheduling policies

One of the most important questions in the selection of a suitable scheduling strategy is whether it should be preemptive or not. This term was also already marginally mentioned in the context of process and thread synchronization (see section 2.3). The focus there was rather on the impact the interruptibility of a process or thread has on itself and shared resources. In the context of job scheduling, a non-preemptive job implies the job runs until it voluntarily hands the control back to the system, e.g. because it is waiting for a resource or an event or it reaches its completion[27]. Depending on the exact policy, a high maximum computation time is often given nevertheless in such a case. It prevents faulty or malicious processes from blocking an entire system[1]. Even on modern multicore CPUs is a pure non-preemptive scheduling strategy rather not suitable for interactive, multiuser and realtime operating systems, except for batch jobs. For these systems, a preemptive scheduling strategy with generally interruptible jobs is more appropriate. The whole available computing time is sliced down into equal sized time slots which are assigned to the jobs. If a job does not finish within such a

slot, it needs to be rescheduled according to a certain policy[4]. This results in a much more responsive system.

Whether a scheduling policy should be aware of prioritized jobs follows directly after. There are a lot of reasons to give priority to certain jobs, e.g. to create the impression of a notably more reactive system by prioritising User Interface (UI) related jobs or to ensure the fastest possible processing of a security-critical task in a realtime system[27]. In general, prioritized jobs are available on both, preemptive and non-preemptive policies[4].

The intention of this work is not to provide a complete enumeration of scheduling policies but a selection of relevant ones with respect to Linux and Zircon. As a consequence, preemptive strategies for interactive multiuser and realtime systems are considered in particular. More complete considerations are available in the books of Brause[4], Mandl[27], Achilles[1] and others. For the introduction to scheduling strategies within this work, the individual ones are examined on basis of a single CPU core. In fact do modern multicore processing systems using SMP manage an own job scheduling for each core separately. As a result, competing situations for processes and threads such as discussed in the previous section 2.4.1 arise again. An optimal scheduling strategy for multicore systems should therefore also take the required resources into account to avoid unnecessary blocking or context changes[4]. A global component named *load balancer* can step in if the load on an individual core is much higher than the average[1].

Round Robin (RR) is intended for the use in interactive multiuser systems. It is an enhancement of the so-called *First Come First Serve (FCFS)* strategy used in batch-oriented systems. RR is applied for non-priorizing preemptive timesharing systems instead[1]. All reaching jobs are put into a FIFO based queue. The first one which was put in is also the first one which gets assigned to a CPU core from the scheduler. In contrast to FCFS as a non-preemptive strategy for batch jobs, the maximum time a job is allowed to use the resource CPU is limited. Does a running job not voluntary release the CPU within its time slice, an interrupt to cut it off is triggered. The job loses its CPU core which becomes reassigned to the next job in the queue. If the replaced job was not finished yet, it is enqueued again[1], [27]. The size of the time slice, also called *quantum*, is the same for each process, but the difficulty in this strategy is determination of its effective size[4]. Is the slice too small the overhead caused by switching the process contexts becomes predominant, is its size chosen too long, the resulting strategy resembles FCFS. The calculation of the next job to schedule is very cheap, using RR as a strategy[4].

Dynamic Priority Round Robin (DPRR) is a priority aware modification of the usual RR scheduling strategy. The only aspect that differs is an additional queue in front of the known one. Each job arrives in the first queue with a priority and stays there until this value reaches a certain threshold. The priority value of each job in the preceding queue is increased after a time slice is finished. As a result a job with the priority 8 is executed prior to one with a priority of 4, even if it arrived shortly after

the lower prioritized one. The subsequent RR strategy is not touched at all to add priority support, but the management of the additional queue requires a little more management effort[4].

Shortest Remaining Time First (SRTF) is an algorithm for preemptive scheduling with time slices, too. The decision for the next job is, as described by the name, made based on the shortest remaining computing time needed. As a result, each time a slice ends and a new job needs to be assigned to a CPU core, the remaining time for each upcoming job must be recalculated. Thus, the selection of the succeeding job is quite complex every time[4], [17].

Multilevel Scheduling is a strategy to schedule various types of jobs with different importance. The jobs are often split up into *system processes* with the highest priority, followed by *interactive jobs* and *general jobs*. The lowest priority is given to CPU-intensive and long-running *batch jobs*[4], [27]. The idea for the multilevel scheduling algorithm is to hold a distinct queue for each priority level available. Jobs in the queue with the highest priority will be processed first. The ones in the next lower priority are not touched until the higher ones are emptied. As a consequence situations are possible where low priority processes never become scheduled. One speaks of *starvation*[17]. Multilevel scheduling also allows some variants. The actual scheduling strategy for each queue itself is variable. RR can be used to provide a fair scheduling between all jobs on a single priority level. For realtime systems with one or just a few jobs in the highest level, *FCFS* could be more appropriate in certain cases. Generally is an additional reassignment of the CPU core as soon as a high prioritized job enters its queue reasonable in realtime system using multilevel scheduling as a strategy[17].

Fair Share Scheduling is a strategy to guarantee the same share of the CPU to each single job. Having n jobs in the whole system, each one gets a $1/n$ share of the whole computing time allocated. As a result, the computing time used so far must be tracked along with each job. This time is compared with the actual overall time the CPU worked. The decision which job is assigned to the CPU next is made on basis of the worst ratio between already consumed computing time and the overall available one. A job scheduled according to this strategy is allowed to stay active until the worst ratio is calculated for another one[27].

Earliest Deadline First (EDF) is a realtime scheduling strategy. Since certain critical tasks have to be completed at a given time to avoid more or less serious consequences, this algorithm uses this deadlines to decide on their basis which job must be executed next. The strategy itself is rather complex but useless in situations where several critical jobs must be completed at the same time[27], [4]. Nevertheless, is EDF a common algorithm for hard realtime systems and an ideal strategy on single core systems with up to 100% CPU utilization[4].

Priority Inversion

Using locking mechanisms like semaphores respectively mutexes may lead to dangerous behavior in context of priority based scheduling. For a basic example two jobs, H (high priority) and L (low priority) are given. Both jobs want to use the same resource. The H job is scheduled as soon as it enters the *ready* state. The problem arises if the L job starts first while H is waiting for e.g. another resource. L enters the critical section and occupies the contended resource. Just in this situation, the H job becomes ready and is scheduled according to the priority based policy. If H is busy waiting for the resource, L is never rescheduled, can not end its critical section and free the resource[49]. Both jobs are blocking each other, they are in a *deadlock*[17]. If H blocks and hands over to L again, the priorities got inverted. A lower prioritized job is running before one with a high priority. Such a situation is referred to as *priority inversion*[49]. The situation described here is only dangerous if the H job misses a critical deadline, but there is a very similar and serious real world example of the mars spacecraft *Pathfinder*. It also uses priority based scheduling with three jobs, H (high priority), M (medium priority, long runtime) and L (low priority). Just as explained for the situation above, the H job is waiting for a resource held by the L job. When H blocked, the M job was scheduled and had to be finished until the L job had the chance to free the resource needed by H. But the high prioritized task of H was to reset a watchdog to prevent a system reset accompanied by the loss of data[17], [44]. A more detailed description of the situation is given by SILBERSCHATZ in [44].

The most common technique to prevent dangerous *priority inversion* is *priority inheritance*, a special property of semaphores which needs to be implemented within the operating system. If a higher prioritized process or thread is waiting for a resource occupied by a less prioritized, the priority of the second job will be raised to the one of the higher prioritized. This ensures that the actual low prior job frees the resource as soon as possible and the truly high prioritized task can run next. The original priority becomes restored, once the resource has been freed[17].

An alternative mechanism that can act without operating system support is *priority ceiling*. Whenever a process locks a semaphore or mutex, its priority is raised to a higher level than any other process maybe waiting for the resource. As soon as the semaphore respectively mutex is unlocked, the former priority is restored[17].

2.5.1 Scheduling in Linux

As usual, modern Linux kernels offer various opportunities to adjust even scheduling to the needs of its intended application. While code running in user space may be preempted at any time, common modern Linux kernels itself can be compiled in three variants:

- As a no-forced preemption version, suitable for the use in servers.
- With voluntary kernel preemption for the use in desktop systems and

- as a fully preemptive kernel for desktop systems with low latency requirements[26], [6].

Older kernels before version 2.6.23 were not preemptive at all. They were only disturbed by explicit or implicit sleep requests, e.g. if a required resource was not available, or, if enabled, on interrupts. After the disruption, the previous task was resumed on the CPU core.

Back on modern kernel versions, one task can not interrupt another one by itself. As scheduling in Linux is based on time slices, it is the duty of the scheduler to assign another job to a CPU core as soon as a slice ends. Summarized, a job scheduled in Linux is allowed to run until

- it needs to wait for a resource to be freed or a system event to complete,
- it reaches the end of its lifetime and exits,
- its time slice expires,
- the scheduler becomes invoked and finds another job that has to be scheduled according to the current policy[26].

Linux uses one scheduler per CPU core and a global load balancer to move jobs between them, just as described in the general section. However, both try to schedule a job on the same CPU to minimize cache thrashing[26].

Before kernel version 2.6.23, the default scheduler in Linux was named *O(1)* scheduler. It is named after the *Big-O notation* of its algorithm which classifies the runtime or memory requirements of algorithms according to its inputs. This means in case of the O(1) scheduler that the time it needs to make the decision which task is to be scheduled next is independent of the number of tasks in the system at all. It was using a priority ordered queue for each CPU core[26]. Since kernel 2.6.23, the Completely Fair Scheduler (CFS) became the new default scheduler. It is a fair share scheduler, similar to the strategy described in the general section. The decision which task must be scheduled next is made on the basis of the amount of time a task has been waiting to run, divided by the number of running tasks. The resulting time of each task is compared with the actual time received so far. The one with the worst ratio is selected. Thus, the decision process is rather complex in this strategy[26].

Linux' origin is an interactive operating system kernel, but there was the desire to use it in realtime environments as well. For this use, the Linux kernels needed lower latencies, other preemption methods and realtime capable schedulers. As the needed changes affect wide parts of the kernel they were initially developed as a patch set called *rt_preempt*. They include amongst others:

- Replacing spinlocks with preemptive *rtmutexes* in the whole kernel.
- Implementing priority inheritance for locking mechanisms.

[6] See also "`man 7 sched`" on Linux systems as an additional source for this section.

- Replace common interrupt handlers with a preemptive kernel thread so that they can be scheduled and sleep[26].

The scheduling in Linux is very fine granular adjustable to the system and its applications' needs. It is not only possible to select the grade of preemptibility and make it aware of realtime tasks, but also to set a scheduling policy for each single thread. Three normal and three realtime policies are available for this purpose. The normal ones are:

- `SCHED_OTHER` respectively `SCHED_NORMAL`, as it is called within the kernel, which describes the default timesharing scheduling with a static priority level.
- `SCHED_BATCH` which is used to schedule CPU-intensive and non-interactive tasks. It has to be used with the static priority 0.
- `SCHED_IDLE` which is used for tasks with a very low priority, even lower than batch jobs.

For realtime tasks, Linux provides:

- `SCHED_FIFO`, a First In First Out policy just as described in the general section. It can be used with static priories greater 0. A runnable thread using this policy will immediately preempt any running task using the normal policies mentioned above.
- `SCHED_RR` just as mentioned in the general section is Round Robin even in Linux a time slicing version of the FIFO policy.
- `SCHED_DEADLINE` is the latest scheduling policy in Linux. It is available since version 3.14 and provides a variant of the Earliest Deadline First policy mentioned in the general section combined with the Constant Bandwidth Server (CBS) policy. EDF is used for the actual deadline scheduling while CBS guarantees non-interferences between threads. This strategy targets hard realtime requirements. Thread scheduled with it are the ones with the highest priority which can be controlled by a user.

A much more detailed explanation of thread specific scheduling policies and their use via `sched()` is given as part of the man-pages project using the command `man 7 sched` on a Linux system[7].

2.5.2 Scheduling in Zircon

The Zircon scheduler was developed out of Little Kernel's one and provides just a minimal realtime aware scheduling mechanism instead of the variety of possibilities known from Linux. But the basic mechanism behind is also already known. Zircon scheduling is a kind of multi-level scheduling with time slicing as stated in the general

[7]The man-pages version used is *2019-03-06* on a kernel in version 5.0.2.

section. The scheduling is done for each CPU core on its own and each core has its very own set of queues. As Zircon supports 32 priority levels right now, this makes 32 FIFO queues per core[6]. Each job gets the same time slice. If it is not finished with its work, it is enqueued again. The scheduler starts with the queue with the highest priority and executes the jobs for the maximum time of a time slice. Only when this queue is empty, the next one is considered. If a job blocks, it is completely removed from the CPU's queue and added to a distinct waiting queue. It stays there until the required resource becomes available and thus its status changes to runnable again. The job is put back to, whenever possible, the previous CPU core's queue with the right priority. Was the job blocked before its time slice was finished, it becomes rewarded by being inserted at the front of the queue to resume as soon as possible[6]. The priority of jobs in Zircon is determined on the basis of three criteria:

- A base priority from 0 to 31.
- A priority boost.
  - A job is rewarded with a one level upgrade when it unblocks after waiting for a resource or sleeping.
  - A job is penalized with a one level downgrade if it voluntarily gives up the control. The minimum priority value in this situation is 0.
  - A job is penalized with a one level downgrade if it is preempted after using up its entire time slice. In this situation, the priority value may become negative.
- Priority inheritance is a special criteria. If a job controls a resource which is needed by one with a higher priority, the job's priority is temporary boosted up to the priority level of the other job to prevent priority inversion (see 2.5).

As a result, the effective priority is either calculated from the base value in combination with the boost or it is the inherited one[6]. This policy is designed for an interactive system. Rewarding short and non-CPU intensive jobs affects most of all UI related tasks which are often blocked by waiting for user interaction, while CPU intensive background tasks are penalized for using the whole time slice and hindering the ability to react. Zircon is per design capable for realtime tasks. Within the scheduler it is considered by a special flag for them: `THREAD_FLAG_REAL_TIME`. Such a job is allowed to run within its priority level without being preempted until it blocks, yields or is manually rescheduled[6]. A special role belongs to the *idle thread*. It lives besides the normal FIFO queues and is run by the scheduler if there are not any other jobs ready to be assigned. The purpose of the idle thread is to keep track of idle times, but some platform implementions may use it to implement a low power consumption wait mode[6].

At the time of writing, an additional CFS is in the works. Its API is compatible to the one Linux uses. It is not merged into the standalone Zircon repository yet,

but to the development master branch of the Zircon kernel within Fuchsia. Further information is available in the according documentation within this repository[8].

2.6 Memory Management

Memory management is an essential part within an operating system. Without going into detail, some terms and principles belonging to this topic were already part of previous sections. For example, the topics *process isolation* or *IPC via shared memory* involved sophisticated concepts for the use of the main memory in particular. The concepts of *virtual memory* and private *address spaces* had a special significance in this context. Thus, this section will focus on them.

The reason for the use of such sophisticated memory management mechanisms over direct memory allocation is motivated by the need for process isolation, shared memory and similar approaches but also by the nature of a computer's memory. In common, the memory is a hierarchical system with CPU registers at its top. Registers are implemented in the same technology as the CPU itself. As a result, they are very limited and expensive but also very fast. The next layer are caches. Current systems use a hierarchical cache system itself. The topmost layer, the layer 1 cache is nearly equivalent to registers in terms of access time and costs[49]. It is always a part of the CPU to reduce the signal paths. But there is not unlimited space and thus, it is very small sized[49]. After two to four cache levels follows the main memory or Random Access Memory (RAM). If memory is mentioned within this work, the main memory is meant in the majority of cases. In common, it is implemented as *Dynamic* RAM. Accessing the main memory is 10 to 50 times slower than accessing caches, but the capacity is significantly larger (e.g. Intel Core i7-7820HQ; L1 caches are divided into data and instruction caches with 32 kilobytes each, while the total main memory installed is 16 gigabyte[9][27]. The data of running processes and the system itself is generally held in the main memory. The last kind of memory which is important as a part of this thesis are common hard drives and newer Solid State Drives (SSDs), used for files, not running applications and miscellaneous other data.

The memory hierarchy is a result of often not having enough space. Not even the main memory is always capable to store the data of all running applications. But there is a desire for allowing processes to use more memory than physically available and for a unified contiguous view of the memory area of each process. A solution for this issue, the virtual memory, merges with the requirement of *process isolation* between the address spaces of distinct processes. For real, the main issue for virtual addresses is process isolation but this also creates a need for efficient Inter-Process Communication (IPC) and ways to share memory between processes (see 2.3 for the details)[27], [4]. In current systems, this is mostly realized by using the mechanisms of *address spaces* and *virtual memory*, which already have been mentioned several times.

[8]Zircon Fair Scheduler, visited on 11.04.2019 `https://fuchsia.googlesource.com/fuchsia/+/refs/heads/master/zircon/docs/fair_scheduler.md`

[9]Benchmarks from an Intel Core i7, 7th Generation.

2.6.1 Address Space

The term *address space* covers the summary of all possible memory addresses within the main memory from 0 to $2^{32} - 1$ on a 32-bit system or $2^{64} - 1$ on a 64-bit one. It is divided into reserved parts for the system, applications and others. How this division is done is defined by the operating system in an *address space layout*[27]. Distinct parts within this layout are protected against each other, especially the system kernel itself[4]. But the layout distinction does not provide process isolation. The system is indeed divided from application processes using such a layout, however those are not separated from each other within their sections[27].

The term *address space* is used for the memory region of applications. It is effectively located within the main memory's address space. Their internal layout is specified by the compiler or a runtime based on language conventions or standards. A C application for example, is divided into different sections like the program code itself, constants, static variables and initialized data, not initialized data, and dynamic sections for heap (data which is allocated at runtime) and stack[27].

However, the pure implementation of divided address spaces does not yet solve all the needs to be mentioned above. Although the address space as a whole is now subdivided into various regions, it is not ensured that the address space of a process within is always contiguous. From a technical point of view, this is not always possible because the size of process address spaces differs and as a result, gaps which have to be filled as well arise[27]. Furthermore, data should be aligned to addresses which are divisible by four to make accessing main memory from the CPU more efficient[27], [4].

2.6.2 Virtual Memory

The abstraction needed to reach the named requirements for the physical main memory is called *virtual memory* or *virtual address space*. Its basic idea is to provide abstracted, contiguous views on the actual memory to processes, which was one of the goals. A process, respectively a developer, only sees a contiguous memory region starting with the address 0 but not the actual fragmented main memory used by various processes from distinct users[27], [4]. Such a virtual view of the actual main memory also enables pretending that more memory is available than actually present to an application. Since the entire memory region of a program is rarely needed in main memory at a time, the concept of virtual memory together with a sophisticated mechanism to load certain memory regions from disk enables the execution of processes, even if they can not be completely loaded into the main memory. The operating system tries to hold currently needed data in the main memory while the other information is stored on hard drives. As a result, the overall memory consumption may become greater than the physical available main memory and virtual memory is not necessarily mapped to physical memory[27]. An operating system unit called *memory manager* visualizes each process for itself. Virtual memory is only a logical construct until the memory manager maps it to physical available one during runtime. The purpose of this unit is, besides the already mentioned, implementing

- a *fetching policy*, which describes a strategy to ensure a certain memory region is present in the main memory when needed,
- a *placement policy*, which manages where newly added regions are placed within the memory,
- a *replacement policy*, i.e. the strategy that comes in when there is no more free memory available and data must be removed to load the currently needed ones, and
- a *cleaning policy* trying to keep some main memory free at any time[27].

For the use within the memory management and the actual implementation of virtual memory are virtual as well as physical main memory regions divided into fixed size blocks. The blocks in the virtual address space are referred to as *pages* while the ones within the physical address space are named *page frames*. Size and location from virtual pages and physically available page frames differ from each other. The addresses of the virtual pages are managed in so-called *page tables*. One distinct page table represents a program which is available in the main memory[4]. A physically available page frame can be mapped into several virtual page tables, that means into several applications. Shared libraries and shared memory IPC mechanisms are implemented in this way[4].

Pages, respectively page frames, represent the memory blocks which are transported between hard drives and the main memory by the *memory manager*. The policies implemented by the memory management take advantage of sophisticated concepts like *demand paging*, i.e. a page is only loaded if there is a need for it, and *pre-paging* with use of the so-called *locality effects*. The two most important ones are the *temporal locality* and the *spatial* or *memory locality*. They state that pages which were often needed in a temporal proximity in the past will continue to be needed together with a high probability in the future, respectively that virtual pages located close to each other are often used together. In accordance, the memory manager tries to load pages which are needed with a high probability in near future in advance[27], [4].

Above all, virtual memory enables the sophisticated memory management requirements needed for an efficient and secure operating system with process isolation, shared memory and a contiguous view to a program's address space for example. Memory protection mechanisms like process isolation and access rights to a process's address space are implemented based on virtual memory, too (refer to Brause[4] for more details). But thus, a translation between virtual addresses and the physical equivalents which considers the fact that a physical memory region may be mapped into several virtual address spaces, is needed. To speed up this costly translations, current application-oriented CPUs are supported by two on-chip hardware mechanisms, the Memory Management Unit (MMU) and the Translation Lookaside Buffer (TLB).

The MMU is a part of the processor itself and responsible to map virtual addresses to physical ones. Thus, accessing the raw main memory and regions with a special meaning is in common a privileged action done within the operating system kernel. The CPU sends a virtual address to the MMU which disperses the address to a physical one using the according page table and sends the result via the address bus to the

main memory. If a needed page is not available there, the MMU raises a special interrupt named *page fault*. It interrupts the running process so the kernel can try to fetch the needed address from disk. Even if only a single address is missing, the memory manager always loads whole pages to the main memory. If the virtual memory address as a resource is available, the interrupted process is rescheduled according to the currently used policy[27].

The TLB works together with the MMU. It is an additional fast cache to speed up the lookup between virtual and physical memory addresses. As described previously accessing a cache is still considerably faster than accessing the main memory itself. Similar to common L1 and L2 CPU caches there is an own TLB per CPU core. If the needed address translation is already stored in the TLB, no further access to the main memory is needed, if not the MMU itself must be invoked and not only its caching component, the TLB. Just like caches and the main memory, a replacement strategy for the TLB is needed to ensure its efficient use. A more detailed introduction to these hardware support mechanisms and related strategies can be consulted at Mandl[27] or Brause[4].

2.6.3 Memory Management in Linux

Linux also uses a virtual memory system, on modern MMU-enabled CPUs, but there is a Linux kernel variant for MMU-less architectures used for very small embedded systems, called *uCLinux*. While a usual Linux kernel with virtual memory system needs an address translation, *uCLinux* uses raw physical memory addresses without further protection or other sophisticated mechanisms. But more common and focused in this work is the use of virtual memory enabled systems[26]. In general (virtual) memory management in Linux is a very complex and highly architectural dependent matter. As *32-bit x86* was the most common architecture for a long time, it is often used to describe the mechanisms, but it is also very specific in a way that even the memory management for *64-bit x86* architectures differs a lot. Nevertheless, this work focuses on *32-bit x86* due to the spread of the corresponding terms and mentions important differences in 64-bit x86 and ARM architectures.

The Linux virtual memory management mechanism divides the available memory in various different sections. The most basic and widespread terms in this context are *high memory* and *low memory*. Both do only matter on 32-bit systems. There is no such division in 64-bit systems at all. And in fact, the meaning of the terms is not complicated. The whole available address space in Linux is divided into virtual kernel and user addresses. *Low memory* is a term for a physical memory region in the main memory, which corresponds to the high virtual addresses used for the kernel. Kernel virtual addresses differ from common virtual addresses and will be considered in the next section. *High memory* describes the higher physical memory addresses which maps to the virtual addresses used within user applications. As described in the general section, Linux provides an own protected address space with virtual addresses for each process, too, which are generally mapped into high memory[26]. Figure 2.8 illustrates the context.

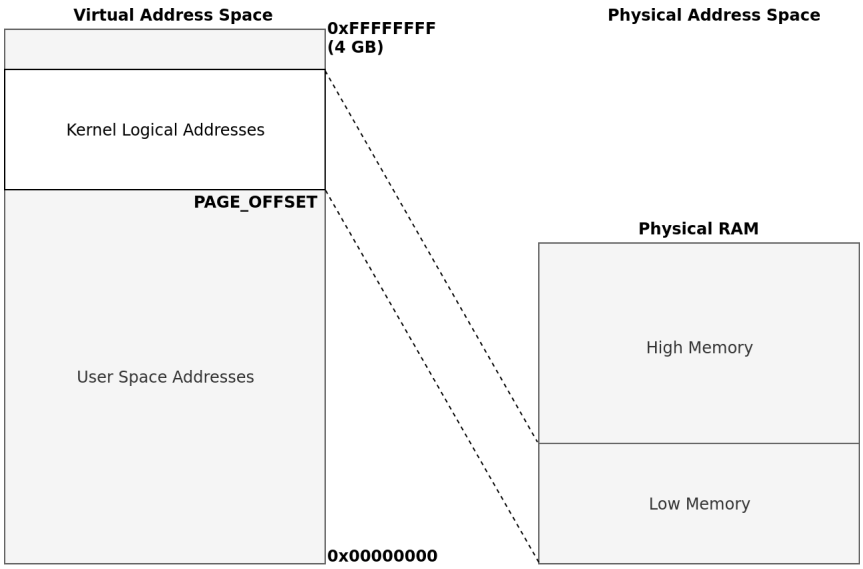

Figure 2.8: The Relation between Virtual and Physical Memory in 32-bit x86 Architectures according to [26]

Actually, the division in high and low memory is just one from several ones. Linux uses additionally a *zone allocator* memory algorithm to partition its address space in a finer granular way. Each zone differs in its usage and comes with its own methods to perform basic memory operations like allocating and freeing memory pages[26]. On 32-bit x86 architectures, the zones are:

- the **Direct Memory Access (DMA) zone** is located from 0 to 16 MB in both, 32-bit and 64-bit x86 architectures. It must be used for DMA data transfers on devices with 24-bit addresses.

- the **DMA32 zone** is also on both variants located from 16 MB up to 4 GB. It must be used for DMA transfers on devices with 32-bit addresses as suggested by its name.

- the **normal zone** is from 16 MB to 896 MB on 32-bit systems respectively it fills the whole remaining RAM on 64-bit systems and overlaps with the DMA32 zone. The normal zone is used for kernel and user data, but kernel addresses in this zone are calculated divergent from user addresses.

- the **high zone** is a 32-bit only facility. It starts from 896 MB and fills the remaining RAM up to 64 GB. This is also the reason for this special zone: it enables 32-bit x86 systems running on Linux to access more than the usually addressable 4 GB of physical main memory[26].

As suggested, Linux combines its memory management with different types of addresses. The available types depend on the architecture, too. This work will focus on 32-bit x86, as usual. The types are:

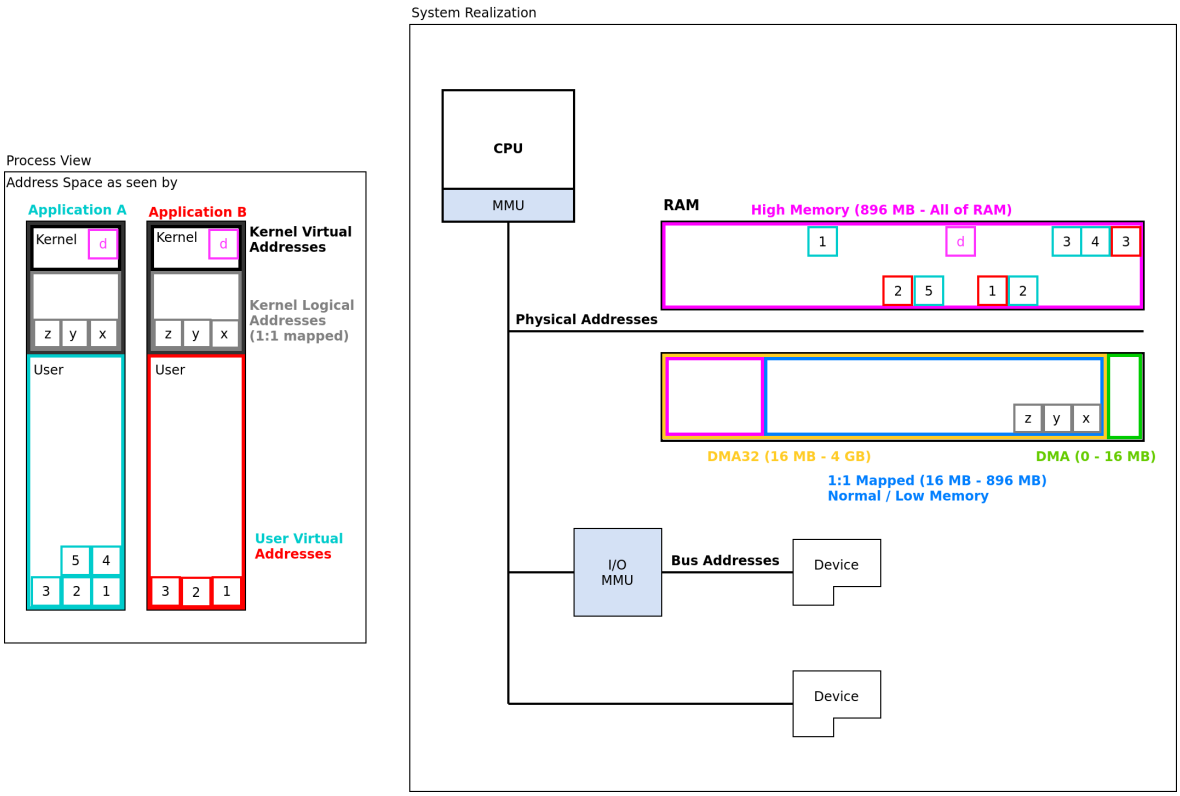

Figure 2.9: Linux memory management overview according to [26]

- **user virtual addresses** as seen by user space applications. They can be 32 or 64-bit long, even on 32-bit systems. Each process has its own virtual address space. As a result, two identical virtual addresses may refer to distinct addresses in the physical memory.

- **physical addresses**, i.e. the actual physical memory addresses which are used between the CPU and the main memory. Also, the addresses may be 64-bit long, even on a 32-bit system.

- **bus addresses**, which are physical addresses used between the main memory and peripheral buses in x86 architectures. In other architectures or even newer variants, this fact may change.

- **kernel logical addresses**, which are actual physical addresses shifted by an offset. They are always stored in low memory and only used within the Linux kernel itself.

- **kernel virtual addresses**, which are kernel addresses with no need for a direct mapping to physical memory addresses. It is a 32-bit x86 architectural anomaly and not available on 64-bit systems. Kernel virtual addresses which are not logical addresses, too, reside in high memory[26].

As also described in the general section, Linux uses fixed sized pages within memory management. The actual size of a page depends on the CPU architecture. Pages in x86 are in common 4 KB sized while ARM based architectures may use 4, 16 or 32 KB sized ones. Thus, application code should never depend on page sizes. Linux manages the pages using page tables, too. For the kernel related pages the virtual or logical address are decoded to the offset and a *virtual Page Frame Number (PFN)*. This PFN is stored in the page table together with access control information and a *valid* flag. At this point, it becomes apparent that the entire memory handling in Linux and also the protection of memory regions, e.g. against writing and/or code execution, is done on page level. The *valid* flag indicates if a virtual page is effectively available in main memory. If a page is not valid, i.e. swapped out to disk or not even yet loaded to the main memory, a *page fault* is raised if the page is requested. The kernel is invoked to load the proper page into main memory using demand paging and the hardware facilities *MMU* and *TLB*[26].

Figure 2.9 illustrates the most important terms and concepts of memory management in Linux at a glance. The graphic is inspired by the one done by John Bonesio during a *Developing Linux Device Drivers* Linux foundation training class. On the left-hand side of the graphic, the view to the (virtual) memory from two distinct applications A and B is pictured. Both see their own virtual address space using *user virtual addresses*, marked in turquoise and red. The view on the kernel's address space is the same for both of them. On the right-hand side, the graphic pictures the connection between CPU and MMU (including but not pictured the TLB), the physical main memory divided into *low* and *high memory* and the peripherals as well as the address types used in between. The physical memory is divided into the known memory zones, the *DMA* zone in green, the *DMA32 zone* in orange, the *normal zone* or low memory in blue and the *high zone* or high memory in *pink*. Additionally, the graphic symbolizes how pages in different contexts are mapped into the physical main memory. Both user processes have numbered pages which can also be found in the right-hand side RAM representation. Furthermore, some kernel pages are marked in gray with the characters *z, y and x* which are 1:1 mapped into the normal memory (kernel logical addresses) and a page in purple (kernel virtual address), mapped into the high memory zone.

Linux memory management is, as already mentioned, a very complex topic. This section is only an introduction to the most important terms and concepts. It is not possible to give a complete and detailed summary on this topic as part of this work. Thus, a very good and detailed resource to memory management in Linux on different architectures and the effects on kernel programming is the *LFD430 Developing Linux Device Drivers* course script[26] respectively the corresponding Linux foundation training class, which was also used as main source for this section.

2.6.4 Memory Management in Zircon

The memory management in Zircon is either less complex than in Linux, even if the implemented concept itself is basically the same, or the available documentation provides just little insight. All available texts on this topic include the terms *Virtual Memory*

Address Region (VMAR) and *Virtual Memory Object (VMO)* which already were part of the section 2.3.4 about processes and threads in Zircon. As seen in this section, but also in the documentation, both terms are interlocked and belong together in most cases.

The basic object for virtual memory management in Zircon is the Virtual Memory Address Region. A partitioning of the physical address space as done in Linux is not described for Zircon at all. The VMAR is an abstraction to manage a process's address space, more accurate a contiguous region of a virtual memory address space and the allocation of an address space. A distinction between the handling of kernel and user processes is not done at all. VMARs are per process objects. Along with the creation of a process, a root VMAR is created to span the entire virtual address space for this process. This root VMAR can be divided into a number of non-overlapping subareas, e.g. to group related parts together in a sub-VMAR. Altogether, a subsection of the origin root VMAR can also represent a virtual memory mapping or a gap as well as a child VMAR such as mentioned above. VMARs support a hierarchical permission model in order to regulate allowed mappings. Thus, it is impossible that a child VMAR has more permissions than its parent. In Zircon, the address space spanned by VMARs and all allocations are randomized by default. It is an additional security mechanism which is also available and enabled per default in Linux but not mentioned explicitly in the previous section due to the scope of Linux' memory management. On creation, the caller can choose a certain randomization algorithm for a process, otherwise the default algorithm is chosen. It tries to spread the allocations widely across the available size of the VMAR. Additionally, VMARs support an optional fixed offset mapping mode, comparable with kernel logical addresses in Linux, which is called *specific mapping*[31], [14].

The second part in Zircon's memory management are Virtual Memory Objects (VMOs). They represent a set of physical memory pages or the potential for pages, i.e. a contiguous region of virtual memory may can be mapped into one or multiple process address spaces (VMARs). VMOs are used from both, kernel and userspace and represent virtual memory pages as well as physical memory. A VMO is created as needed on demand, i.e. virtual pages are also allocated on demand. As this object is based on pages, its size is only given in whole pages. Permissions of mapped pages within a VMO can be adjusted on page level, too. VMOs are the standard method to share memory between processes in Zircon. Thus, they support basic I/O operations on the memory, e.g. a VMO can directly read and written. For the use within device drivers or other processes with special purposes it is possible to set specific cache policies for a VMO[31], [2].

2.7 I/O

The term Input/Output (I/O) is commonly used to describe the interactions between a computer itself and several device types such as disks, keyboards, displays, clocks and much more which are usually referred to as *peripheral devices*. Thus, it enters the actual topic of this work: device driver development. The majority of device driver de-

velopment deals with I/O devices and related issues, such as controlling the device via a defined interface, reacting to an interrupt from a device and handling it. Finally, an I/O system, respectively device drivers for I/O devices as part of an operating system, have to provide simple and consistent abstractions between the actual device and the user[49]. Nevertheless, this section will focus on the communication mechanisms between I/O devices and operating systems, but not on the physical connection between computer and peripherals or protocols. The actual implementation of I/O drivers in both considered systems follows in the next chapter of this work. But goals for I/O hardware itself and drivers are often the same or similar, e.g. reaching a certain degree of decoupling the exact device and the corresponding control interface so that different devices from the same type, for example a few different mices, can be controlled in an uniform way[49].

Peripheral devices are divided into two categories all in all. *Block devices* are usually used to store information in fixed size blocks. Like internal memory every location within such a device has its own address. Specific addresses are accessible and seek operations are feasible. Examples for such a peripheral type are hard disks or USB memory sticks[49]. But the majority of peripheral devices belongs to the category of *character devices*. Those are rather stream data oriented, i.e. they deliver or accept information without considering any block structure. The data is not precisely addressable and seek operations on the device's data are not possible. Peripheral devices in this category are manifold. Their types range from printers over network interfaces to sensors[49]. But some devices do not fit in the scheme at all. A hardware clock is also a peripheral device for example, but it just raises interrupts in certain intervals instead of exchanging data.

All types of peripheral I/O devices are in common equipped with an electrical controller in form of a chip or primitive processor. These are basically programmable or controllable via special purpose registers. The purpose of such a controller is managing and abstracting the communication with the actual peripheral device and the computer it is attached on. In common, the actual device is managed via control and state registers and in some cases data buffers on the controller. This means in general writing certain values into defined control registers on the controller triggers actions of the peripheral devices while reading them return a state information but also results, e.g. sensor values after a measurement was triggered previously via the registers. The I/O device controller is attached to the computer's bus system, which finally provides various ways to access the peripheral device's control registers[49], [17].

2.7.1 Memory Mapped I/O

The first and also most widespread method to interact with the peripheral device's control or data registers, mostly from *character devices*, is mapping these few registers into the address space of the actual computer, i.e. into the main memory of the system. This results in accessing an address, mapped to a control register of a peripheral I/O device, is not longer accessing a storage location within the main memory but the control register outside. From a developer's view accessing such a control register is

nothing special using *memory mapped I/O*. It is a common memory operation which can be written in C or another high-level language without a need for specialized and architecture dependent assembly instructions. Beside the simple and transparent use of *memory mapped I/O* to access a peripheral device, it can also score by the fact that a basic protection is already given. Accessing the main memory directly is already a privileged operation within most systems but also mapping I/O regions to a user process invokes the operating system itself and the MMU to check the needed permissions[49], [4].

2.7.2 Direct Memory Access

An optimized way to interact with a peripheral I/O device is Direct Memory Access (DMA). In earlier days, it was commonly used on CPUs where *memory mapped I/O* was not available, but the so-called I/O ports, which are not further considered in this work. Today, the purpose of DMA is rather speeding up large data transfers between a peripheral device and the systems main memory. The idea behind is moving the rather costly I/O transfers from the CPU to a distinct but specialized controller, which allows direct interaction between an external device and the memory. As a prerequisit this controller needs direct access to the processor bus, i.e. it must be able to take the control over this bus from the CPU. In the first step, the CPU is still needed. It must setup the DMA controller's internal registers which specify the start address of the data to transfer, a byte count and the transfer direction amongst other things. The controller takes over the control and initializes and executes the data transfer by issuing requests to the I/O device instead of the CPU. After a single partial request, e.g. the write of a memory address, is done, the DMA controller increments the address, decrements the byte count and re-executes the transfer until the byte count is 0. Once the whole transfer is done the controller sends an interrupt to the CPU and returns the control over the bus. The DMA controller may work on physical or virtual addresses. For the latter ones, the MMU must be involved. As the CPU is bypassed during DMA transfers, caches may become inconsistent and require additional handling to prevent inconsistent data between them and the main memory[49], [17].

2.7.3 Interrupts in I/O Devices

Interrupts were already part of this work in different places. Within I/O operations, they are commonly used by the I/O peripheral devices e.g. to signal data is ready to be read or a given action is finished. To do so, the I/O device asserts a signal on the assigned bus line. The signal is detected by an interrupt controller chip within the computer itself respectively the CPU which decides how to handle it, e.g. by activating the right interrupt handler routine. In the context of I/O devices interrupts are popular for keyboards or hardware clocks[49], [17].

2.7.4 I/O in Linux and Zircon

I/O operations at the level described in this section are very hardware dependent and less influenced by the system. Mappings, e.g. used for memory mapped I/O are in fact the same as known from mapping a shared memory region into a process's address space. Both, Linux and Zircon have their own APIs to perform these actions, but these are, especially on Zircon, on a system level which even a driver developer does not or rarely enters. Thus, rather the Zircon side of DMA is of interest because of the object-based kernel approach, while DMA but also memory mapped I/O in Linux is pretty much as described in general.

Memory mapped I/O in Zircon does only differ by using VMOs to represent memory. It is possible to create a special kind of VMO, a physical VMO, representing physical memory regions for the I/O device registers. This VMO type can be mapped into process VMARs just like any other common VMO. However, the rights to create a *physical VMO* are very limited. After all, only the bus driver should create such an object and hand it over to ordinary device drivers[28].

DMA in Zircon is controlled via a Bus Transaction Initiator (BTI) object. It represents the ability of a device to take the control over the bus and with this to perform DMA operations. A BTI object can also be used to granting a device access to memory. It can pin memory used in VMOs and the given physical device addresses can be used to initiate memory transactions, i.e. DMA transactions which proceed as described in the general part[12].

2.8 Summary - Concepts in Linux and Zircon

The main difference between Linux and Zircon remains the basic architecture and about 25 years passed time. A lot of general concepts stay largely the same on both systems but the implementation differs due to architectural concepts, used programming languages, but also due to different design goals. Linux is a grown kernel which implements nearly every operating system theoretical concept. It is a very open environment. If anyone is in need of a certain concept it can be integrated into the Linux kernel or an own fork. Beside the very basic kernel itself, Linux is highly configurable to specific needs. As a result, there is not *one* single Linux kernel but a number of variants depending on the chosen configuration. Zircon is more radical in many aspects. It is developed in public by a team from a large company for a certain, but at the moment not further specified purpose. Yet, it is not open to receive contributions from outside. Even if the used concepts do not differ that much, the implementation does. This is justified by the microkernel architecture and its effects but also by the used programming languages and above all by learnings from other operating systems, especially the ones used in Google's environment like Android or Chromium OS which are based on Linux. Zircon is considerable less broadly positioned than Linux but the fewer concepts used and their implementation are very well selected and executed to suit the needs and design goals of the overlying Fuchsia system. It is basically one resulting kernel with only a few configurations and the focus lies on modularity, security, safety and stability anyway[16]. Thus, the Zircon kernel is designed from scratch for the use in an interactive or realtime operating system, probably for a new mobile OS, where exactly these design goals predominate the advantages of a Linux kernel. However, the focus on the few carefully selected concepts offers the opportunity to optimize them to a high degree.

Chapter 3

Case Study: Driver Development in Linux and Zircon

Device drivers are an integral part of operating systems and require an in-depth understanding of the peripheral device and its controller, the hardware interface between device and computer abstraced by the target operating system from the developer. The fundamental operating system principles and their realization were already established in the previous chapter. But the question of what a device driver actually is and its responsibility was not discussed so far. Therefore, answering these questions is a valid entry point to the actual case study about the driver models in Linux and Zircon and the exemplary device driver development.

A device driver's main purpose is providing an abstraction between user applications and peripheral devices[17]. A typical developer should not have to think about the way a specific device is controlled. Especially, as even devices from the same type differ in the exact way they are managed, it would require too much in-depth knowledge from application developers and make the resulting applications very error-prone. Thus, it is the task of a driver as a part of an operating system to

- define an abstraction of a device to the system,
- build and maintain the connection between applications and certain peripherals,
- initialize the peripheral controller and the device if it is needed,
- query the device state from the controller,
- log events,
- provide a consistent API for all devices from the same type to the user,
- receive abstract application requests and translate them to commands which can be submitted to the device and
- transfer data from and to the device[17], [49].

Driver development is very operating system dependent and has wide-ranging consequences. While the decision if drivers are located in user or kernel space some further questions remain with the choice of an architecture, some further questions remain. Some of them are about the way and point in time a driver is attached to the operating system. For example if the driver must be known at compile time or if it is possible to attach it later during runtime[49]. In any case, each system should provide a unified but extensible device driver interface which supports various device types, even for those that were non-existing at the time the operating system was designed[17]. The design of the interface could be specific to each device type or standardized for all drivers[49]. Furthermore, in almost every operating system specific driver model, it has to be ensured a driver is *re-entrant*, which means a running driver must be safe if it is called from several processes at the same time, and also still safe if peripheral devices become added or removed while the computer is still running[49]. In order to avoid re-developing or duplicating driver parts that remain the same for devices of the same type, device drivers are often modeled and implemented as a hierarchical or layered model. A widespread pattern is to split the implementation into a logical and a physical layer. The logical one contains driver functionality which remains consistent between devices of the same type while the physical layer only takes care of device specific functionality[49]

3.1 Linux Driver Model

Driver Types

As already mentioned, drivers most importantly implement an abstraction of the communication with I/O peripherals. To do so, Linux provides more ways than commonly known. The first one is via *direct hardware access* as *user-space driver*. As known from the previous sections most I/O related drivers and its operations are privileged. But Linux offers a way for normal user applications (userspace) to access hardware without a classical driver. This way of accessing the hardware is mostly used for video drivers which are incorporated into the *X.Org display server*. In order to obtain the necessary rights two systemcalls are needed: `iopl()` to change the privileged level and `ioperm()` to set the I/O port permissions. These calls can only be performed by a privileged Linux user or the *root* user. Compared to other options to access peripheral hardware, this one has the disadvantage that interrupts are not available at all and the user software may run into issues with demand paging. So this option is possibly slower than a device driver in kernelspace, but for some tasks, like the already named *X.Org server* is this kind of a accessing hardware more appropriate than a kernel driver[26], [17].

Another widely unknown and CPU specific way to perform I/O operations is using *minimal operating system support* to access serial interfaces on x86-based CPUs. In doing so, the kernel does not know anything about the exact device but about its I/O interface[17]. Thus, this variant is not handled further. Instead, this work will focus on I/O drivers in *kernel-space*, the most common way device drivers in Linux are written. This kind of driver is a part of the kernel's address space, running in the

CPU's kernel mode and thus privileged. User applications can access kernel drivers and thus the devices via common file operations like `open()`, `close()`, `read()` or `write()` as they are shown virtually as *special files* in the device filesystem `/dev/`. Together with the optional entries in the `/sys/` filesystem and the older ones in `/proc/` devices in those filesystems are categorized in a structured way, according to their types[17]. Linux differentiates drivers in the mentioned block devices, character devices and network devices, but internally drivers are structured in *subsystems* of similar device functionality like usb, network, bluetooth, gpio and many others[40].

As indicated above *character* and *block device drivers* in Linux have filesystem entries which are associated with them. Such a file node is the basic way to communicate with the driver from userspace. The fundamental `/dev/` directory but also the `/sys/` and `/proc/` are virtual. They do not require more disk space than the needed inodes. A device is identified by a *device number* which is composed of a *major number* to identify the device itself and the *minor number* to count the existing device instances[26]. Device numbers are in most cases assigned by the *udev* mechanism today. A closer look at it and its relation to the `/sys/` filesystem follows.

Character devices, no matter in which subsystem, have in common that they are well represented as data streams. They provide only sequential access to their data and can be accessed just like a regular file including the standard file operations[26]. The same applies to *block devices*. In contrast to character devices block devices are read and written only in multiples of their block-size. Linux enables devices of this type to behave similar to character devices and transfer any number of bytes per time, too[26]. Random data access is also allowed and the access to their data is usually cached. One characteristic of block devices is the fact a *filesystem* can be mounted on the device. Thus, file operations are of course available on them. Examples are hard drives or USB memory sticks[26]. The third device class, *network devices*, are entirely different. They transfer *packets* of data. Network devices are not mapped as files or provide file operations. Instead, they are most often identified by name (`eth0`, `wlan0`) and accessed via the *Berkeley socket* interface[26].

Driver Build Types

Device drivers in Linux can be a static part of the kernel or a dynamically loadable *module*. In older versions, only a static integration was possible. That means all the possibly needed drivers must be present at compile time. Thus, the kernel size increases but some drivers will not be used at all and to add a new driver a recompilation and a reboot is needed. Current Linux kernels additionally allow dynamically loadable modules with optional parameters. Reloading a driver during runtime without a reboot does not only save space, it is also useful for development. It is only needed to recompile and reload a single module instead of the entire kernel including a system reboot[40]. However, the module has to be built for exactly the same kernel version as running in order to be loaded. A module which is dynamically loaded via `insmod` respectively `modprobe` is, like built-in drivers, a part of the kernel's address space and running in kernel mode. But as this module does a dynamic binding to the kernel's symbol table, it is only allowed to use a, in comparison to built-in drivers, restricted API[17]. However,

being a part of the kernel's address space assigns drivers a special responsibility. As there is no isolation between parts of the kernel, an errornously implemented driver may crash the entire system as a result. Whether a driver should be build as a built-in driver or as a module has little impact on its implementation. It does require little or no changes to the source at all to switch between them as it is mostly a build configuration[26].

Module drivers are indeed useful, but not realizable for every device type. Some drivers, e.g. specific CPU controllers, must be present at a very early system stage to enable Linux to read from hard drives. Writing such a driver as a module which needs to read from a hard drive to be loaded dynamically into the system is therefore not realizable[40].

Driver Interfaces

Device drivers in general need to call Linux system functions to include themself into the kernel, i.e. methods to initialize respectively deinitialize the driver and the associated device, operating system triggered functions, e.g. interrupt handler functions, as well as user application triggered routines to enable the communication between user and device[40]. The latter ones also include the driver-side implementations for the standard I/O API, the file operations. It should be implemented in a device-specific way for the operations that are meaningful for the device. If there is no such behavior, it is preferred to implement only the meaningful operations, leaving the default behavior for the others and switching to a better suited interface for the device. Besides the standard I/O API, there are e.g. communication or multimedia specific APIs which are better more appropriate for devices of these kinds. Often, these interfaces are defined in Linux but built based on the `ioctl()` call which is technically a part of the file operations[40]. I/O control is an universal interface to define its own, device specific commands. An `ioctl()` command is usually made from a number and the type of optional arguments. However, the preferred way is to utilize the macros Linux provides and use them to define a command consisting of the number, the argument types, the size of the transferred data and their transfer direction[40]. This is the best way to verify `ioctl()` calls to a certain extent. The calls defined for the use in `ioctl()`'s must be known in both, kernel and userspace. Thus, these interface descriptions and corresponding datastructure definitions must be accessible from both sides. Typically, they are found in `linux/include/uapi/linux/`[40].

To make the drivers' implementations of these standard I/O functions callable for the system and users, it is required to declare them to the operating system kernel via specific calls. They take structures with function pointers to the driver implementations as an argument. Functions that do not have a meaning for a specific driver are denoted with a null pointer[17],[40].

Data Exchange

Besides controlling the actual device the communication with the user is the main task of a driver and done as part of common calls like `read()`, `write()` or `ioctl()`. This requires data exchange between processes. IPC was already a topic of this work, also the way it is done in Linux, but the communication between driver and user application is different. It is not an exchange between processes in userspace but between kernel and userspace. This includes a change of the CPU's execution mode and thus, different access rights. Addresses in one mode are not necessarily meaningful in the other and additionally user mode buffers may be swapped out from RAM to disk. The Linux kernel helps in this situation with the built-in functions `copy_from_user()` and `copy_to_user()` which provide a validation[26], [17].

Another way to exchange data between both worlds is the use of *memory mapping* via the `mmap()` call, a standard POSIX systemcall. It enables user applications to direct access kernel memory buffers which may also include memory regions of a device controller, by mapping it into the application's address space. Memory mapping requires a longer setup time than `copy_to/from_user()`, but once the mapping is ready, the access is faster and does not need further systemcalls[26], [17]. Normal files should never be accessed from kernelspace. Thus, they are not suitable for data exchange between kernel and userspace[26].

File operations are not the only option for a user interface to drivers. Another one, the *system filesystem (sysfs, /sys/)* is closely tied to the *unified device model*. It is a framework to handle all devices attached to a computer system in a unified scheme with similar data structures and functions. The representation of this model, of the current state of devices and corresponding drivers in a running Linux system, is the virtual sysfs. It gets generated at runtime as a virtual filesystem and spans a tree of device objects with the system bus at its root as a system representation. A driver's interaction with the model itself is most often limited. It is only needed to register the driver on bus type the corresponding device is physically attached to, like Peripheral Component Interconnect (PCI) or Universal Serial Bus (USB). Then, the *udev* mechanism is invoked. It is a mechanism to create entries for devices in the */dev/* directory. Without udev, it would be necessary to create a corresponding node there manually using a device number consisting of a major number for the device type and a minor number to enumerate the device instance. Drivers match on defined devices and it is a rather common situation that one driver instance has to manage more than one corresponding physical device. The minor number is used to map exactly this situation without any mix-ups between the devices. To comply, the driver implementation must also be designed to handle this scenario. It must be *re-entrant*, i.e. one implementation must be able to handle a number of matching physical devices without mix-ups[26], [40].

For drivers registered in the sysfs, this necessary step, the allocation of a correct device number, is done by udev using information exported there. Besides the necessary information which is exposed just by register the driver within sysfs is allowed to expose further *virtual files* underneath the devices node entry. Those files provide an interface to the driver and can be made readable to expose information and writeable, e.g. to change device buffers or enter a defined command. It is also possible to combine both

operations or make the file not accessible at all. The access permissions for a sysfs file entry are fine granular, based on *group permissions*. The implementation for reading or writing of such a file with a freely selectable name is not further limited and a possible alternative to *ioctl()*. In contrast to *ioctl()* calls, it is easier to access driver information via *sysfs* as it only requires *read()* or *write()* calls and these can also be issued from a terminal[26], [40].

Driver Lifecycle

The sequence of a Linux device driver differs marginally depending on its build variant. The driver entry points `init()` and its corresponding `exit()` function are only necessary for drivers built as a module, but also allowed when compiling as a built-in driver. Thus, most driver implementations do not require any code changes and the decision which variant is built depends only on a value in a configuration file. These additional functions do specific initialization which are only needed on modules. Usually, the `init()` function itself and the corresponding init data are specifically marked to be discarded after initialization, while the `exit()` function is not needed for built-in drivers because they are not unloaded at all[26]. The `init()` function of a module is called as soon as a privileged user loads it to the kernel using `insmod` or `modprobe`. It will add and initialize the module but neither the driver is initialized and ready nor is a device connected at this point. Figure 3.1 pictures this in a driver's sequence context. As the implementations of `init()` and `exit()` often only consist of registering the actual driver and thus abstractable boilerplate code, they are often replaced by a macro, e.g. `module_i2c_driver(<driver_struct_name>)` for an Inter Integrated Circuit (I^2C) driver. Regardless of whether `init()` or a corresponding macro is used, the initialization consists mostly of publishing a driver structure to the kernel. Usually, this struct contains at least function pointers to necessary driver entry points like `probe()` and `remove()` and to a sub-structure which consists of driver specific data as its name and a table with specifications of matching devices[40], [26].

Using this specification, a sophisticated mechanism within the Linux kernel calls the given `probe()` of the matching driver as soon as such a device appears. The `probe()` function is used by the driver to test if the device given by the system really matches the driver and if it is the case, to initialize the device's controller and register itself properly at all needed kernel facilities as illustrated by figure 3.1. The signature of `probe()` is not unified for all drivers. It depends on the device type, e.g. if the device is PCI, USB or an I^2C typed[26], [40], [5]. Also, `probe()` is the first driver function that must be *re-entrant*. It is called each time a matching device is detected and there should not be an artifical limit how many devices a driver can handle. Thus, the information needed for each distinct device should be stored in a private per-device data structure. Allocating the memory for this structure and fillng it with relevant information is also a part of `probe()`[26], [40].

The `probe()` function's counterpart is `remove()`. It is called if a device is to be removed from the system or already was removed without announcement, e.g. due to an electrical error. Additionally, `remove()` is called for all devices that are controlled by a module driver at the time a user wishes to unload the module. Only afterwards,

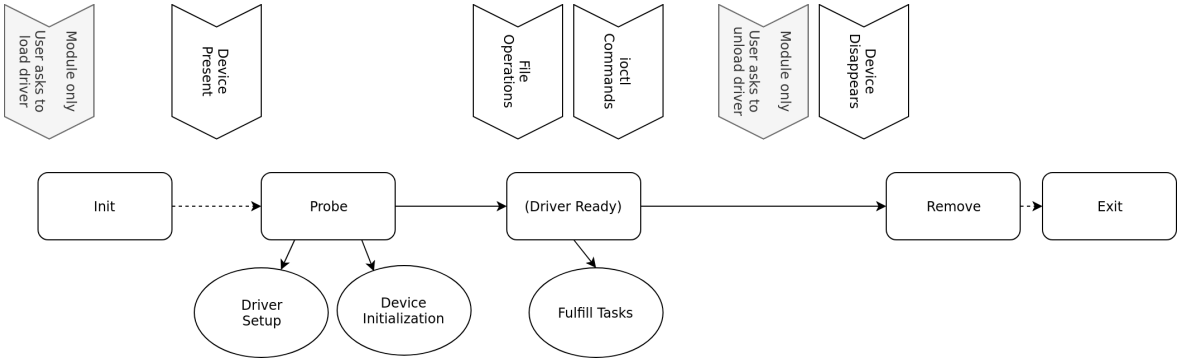

Figure 3.1: Simplified Lifecycle of a Linux Device Driver

the `exit()` function can be called. Within `exit()`, the device should be put in a suitable condition prior to the initialization done in `probe()` is revoked[26], [40]. The `remove()` function is *re-entrant*, too. It is called per device. The right information needed to deregister a device and the driver's entries for this specific instance should be stored as part of the private per-device datastructure. Figure 3.1 illustrates this situation.

After probing the device, the driver for this instance is ready for use. Different interfaces to a driver in Linux were already mentioned in a previous section. Figure 3.1 only pictures the file operations and `ioctl()` as a special situation within them. Regardless of the used interface, all corresponding implementations in the driver must be *re-entrant*, but not only in terms of different device instances using the same driver code, but also for a single instance receiving multiple requests, e.g. from different users. The implementations task is decoding the user's request, translating it in a command for the device controller and taking care of the physical transmission to the device. Depending on the request, this may include sending requests for actions, commands and data to the device but also fetching answers, status codes and e.g. processed data from it[40].

3.2 Zircon Driver Model

The driver model in Zircon differs a lot compared to Linux due to the influences of the microkernel approach. This concerns in particular mechanisms and corresponding terms which are used by the system to manage device drivers and enable them in userspace. Nevertheless, a driver in Zircon has the same purpose as a Linux one: providing a uniform interface to a specific device while its implementation details are hidden[34].

Device Model

Zircon's model for devices and drivers is a direct result of choosing a microkernel approach and at the same time a rejection of the situation in Linux. There, device drivers live in the kernel's address space with privileged access to the whole kernel

memory and other resources. As a result, each part of the kernel including device drivers belong to the same process. A fault isolation within the Linux kernel is not given and a bad driver may break the entire kernel. In contrast, a pure textbook approach for a microkernel would run each single driver in its own process to reach the maximum possible isolation. Even if some real-world microkernel implementations do so, it is not an efficient approach as it requires a great amount of context switches and IPC[34]. Thus, Zircon's idea differs from the textbook approach and groups a number of related drivers together in so-called *device host* processes[34]. A driver itself in Zircon is compiled to an Executable and Linking Format (ELF) shared library, a Dynamic Shared Object (DSO).

Another related mechanism in the Zircon kernel is the *device manager process (devmgr)*. It contains the *device coordinator*, a piece of software that keeps track of drivers and devices. The device coordinator manages the discovery of drivers and devices and is responsible for the creation of device host processes. A DSO driver is loaded into a *device host (devhost)* process and lives there, maybe together with other related drivers, to reduce needed context switches without softening the microkernel concept too much. In addition, the coordinator maintains the *device filesystem (devfs)* as a mechanism that enables userspace applications to access a driver and thus, the device too. Similar to the unified device model in Linux, the Zircon device coordinator views devices as a part of a unified tree structure[15], [34]. Branches of this tree are represented by device host processes which consist of devices. In the current state of Zircon, the policy used to decide which drivers are grouped together for performance reasons and which ones should be placed into separate device host processes is made based on the underlying physical system. As a result, each device that is able to represent a physical bus master becomes a device host process and all corresponding child devices are placed into this process. In the future, this policy will possibly evolve into a more sophisticated concept[15].

In Zircon, device drivers may implement *protocols*, that means C Application Binary Interfaces (ABIs). A protocol is a strict interface definition and defines a set of functions a driver must implement. Protocols are specific to classes of devices. As a result, all devices from a type, e.g. PCI devices must implement the same protocol and thus, the same functions. Zircon differentiates rather in device protocol types such as *PCI, USB, block core or ethermac* than in block, character or network devices. A protocol is used by child drivers to interact with its parent drivers in a device specific manner. So it is an interface protocol between different driver layers, and thus commonly different device host processes, for a particular device type or between drivers in the same device host process[34], [15].

Additionally, a device can implement *interfaces*. They represent *Remote Procedure Call (RPC) protocols* which are used by userspace applications or services. Interfaces are for example the POSIX styled `open()`, `close()`, `read()`, `write()` or `ioctl()` functions but also own interfaces defined using the Zircon specific Fuchsia Interface Definition Language (FIDL)[15].

Within the device filesystem (devfs), Zircon devices respectively drivers are grouped in *classes*. A class represents in this situation a promise to implement certain protocols

and/or interfaces. Devices exist in devfs in a structured way under a topological path according to the scheme `/dev/class/device/drivername`, e.g. `/dev/pci/00:02:00/intel-ethernet`. At the time of writing, the names within the class directories, the device identifiers, are unique numbers in a certain pattern[15].

Driver Lifecycle

It is currently not possible in Zircon to built drivers in a different way than the built-in *ELF shared libraries* mentioned before. They are not loaded into a device host process until it is determined they are actually needed. This is done using the *binding program* which is a part of the driver. Within the driver, it is defined using system internal macros. The compiler moves this program into the *ELF NOTE* section of the binary where it can be inspected by the *device coordinator* without the need to fully load the driver into its own process. Besides the bind description itself, the binding program also contains pointers to the most important driver methods[15].

The first but less used method in the Zircon device driver lifecycle is `init()`. It is invoked when a driver is loaded into a device host process and used for any global initialization. While its equivalent in Linux is often replaced using macros to reduce boilerplate code, Zircon makes it optional to implement it. Typically, no implementation for `init()` is required but if the method is implemented and fails, the whole driver fails[15]. It is pictured in the simplified Zircon driver lifecycle as an optional operation in figure 3.2.

Similar to Linux' `probe()` function follows in Zircon the `bind()` method in a driver's life. It is invoked by the device coordinator which offers the driver a device to bind. This device matches the rules the driver has published as a part of its bind program. Within `bind()`, the driver has to initialize the device, setup interfaces to itself and publish one or more childs of the device to succeed[34]. Adding such a child device is done using `device_add()`. It creates a new device and adds it as a child to a provided parent device. This parent must either be exactly the device which is passed to `bind()` by the device coordinator or another device which already has been created by the same device driver. This method includes adding the newly created device to the device filesystem (devfs) which is maintained by the device coordinator. As soon as a device is added to devfs, the device operations, e.g. `read()`, `write()` or calls defined using FIDL, can be called by the device host. Figure 3.2 illustrates the simplified situation. If a device shall be added but not be accessed already, e.g. to do a longer initialization as a background thread, the device can also be added in an invisible mode using a specific flag. After the initialization is done, the device must be made visible to be accessed[15].

The device driver method `create()` is only invoked for platform or system bus drivers or proxy drivers. Thus, it concerns only very few drivers and is not further considered in this work or the related figure[15]. The driver's `release()` method is invoked right before the driver is unloaded and after all devices it may have created in `bind()` using `device_add()` have been destroyed. The method is currently never invoked because once a driver is loaded, it remains loaded for the lifetime of a device host process. Nevertheless, it should be implemented.

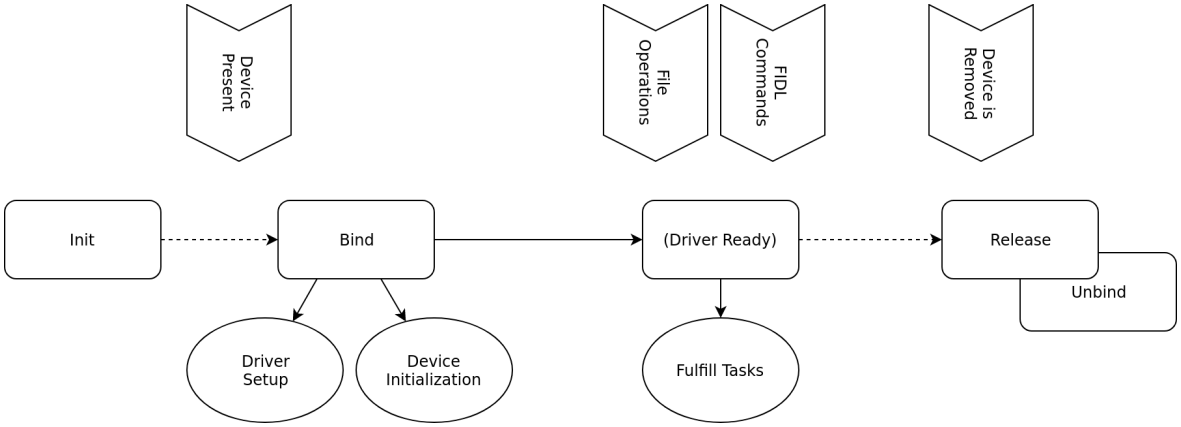

Figure 3.2: Simplified Lifecycle of a Zircon Device Driver

In theory, `release()` and the related `unbind()` method should be called e.g. if a parent device detects the corresponding device is removed and thus, calls `device_remove()` as pictured in figure 3.2. In consequence, the `unbind()` method is called on all child devices because the parent becomes removed. Unbind should remove all interfaces that were created in relation to the `device_add()` call. If a device still has work in progress when `unbind()` is called by the parent, the child device continues this first. Thus, the parent must ensure the device is not working anymore before it also calls `release()` as a last step in this exemplary tear down sequence on all children[15].

3.3 Development Setup

3.3.1 Hardware Selection

The hardware selection for the driver development case study is limited by the available development platforms for Zircon. The best known x86_64 platform is probably Google's own hardware, the *Pixelbook*, which is currently shipped with *Chrome OS*. Unfortunately, there is hardly anything known about suitable internal hardware for a manageable test driver for e.g. sensors.

Thus, the decision was made for an ARM64 based development board with an accessible expansion interface. The chosen **HiKey960** is pictured in figure 3.3. It is not only a development board for Linux but also officially listed as reference platform for Google's *Android* operating system. The HiKey is, amongst other things, equipped with[1]

- 4 ARM Cortex A73 and 4 ARM Cortex A53 CPU cores arranged in the big.LITTLE architecture,
- a ARM Mali G71 MP8 Graphics Processing Unit (GPU),
- 3 GB RAM,

[1]96boards.org, visited on 02.05.2019 `https://www.96boards.org/product/hikey960/`

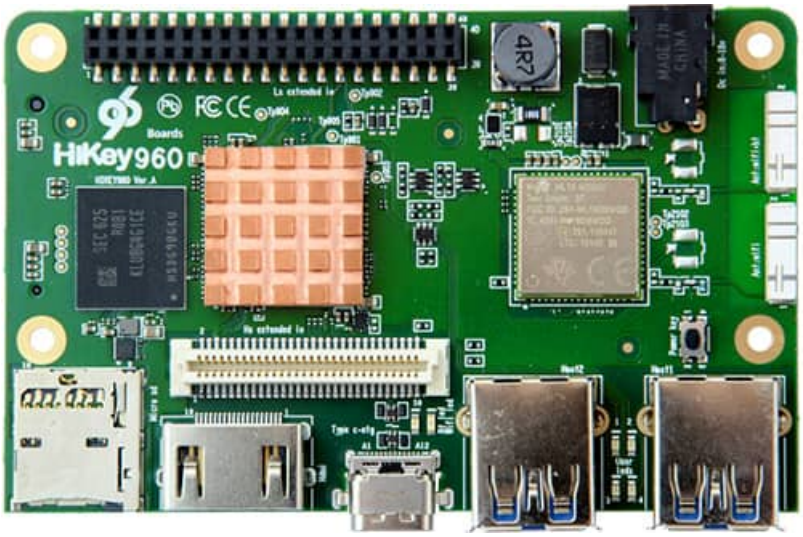

Figure 3.3: HiKey960

- 32 GB Flash Storage,
- an expansion interface, in particular consisting of
  - UART,
  - I²C,
  - SPI and
  - GPIO.

The peripheral device for which the driver is written, should provide a well-known interface that is already supported within the Zircon kernel. I²C matches this requirement and driver examples for C and C++ are available in the used Zircon source version. Of course, I²C support is available in Linux, too. As I²C is a very common hardware communication interface manifold devices equipped with this interface are available. Different sensor types are conceivable as well as actors like displays which is another argument for the decision to use I²C. To make the driver development more sophisticated, the **Grove-LCD RGB Backlight** peripheral device (see figure 3.4) was selected. The peculiar fact about the device is that the Liquid Crystal Display (LCD) and the RGB backlight are controlled by distinct controllers on the peripheral device. Using I²C enables this situation. It is a two wired master-slave bus working in a serial transfer mode. Several masters and slaves are allowed, but both roles can also be combined in one. A data transfer is initialized by a master reaching out for the desired slave using an address. For the Grove device, there are two distinct slaves with individual addresses which need to be controlled. A device's I²C address is usually set by the manufacturer but configurable to avoid conflicts on the bus. For the Grove device, the default slave addresses are `0x62` for controlling Red Green Blue (RGB) and `0x3e` for LCD. Two additional addresses are available on the combined device, but they are

Figure 3.4: The Grove-LCD RGB backlight peripheral device

used on startup and can not be addressed individually[2]. Thus, they do not matter for the driver development.

Hardware Issues

The HiKey960 works internally on +1.8V while the Grove-LCD RGB peripheral device is on +5V. As a result of not working at the same voltage level adjustments are needed. Unfortunately, common level shifters did not work in this situation, because the level ranges for detecting a logical *0* respectively for a logical *1* on the Grove differs between both I²C controllers. Thus, a sophisticated level adjustment to match both ranges was needed to solve this issue. The final resulting circuit is pictured in figure 3.5.

The practical part of this thesis was started with the development of the Zircon driver. Thus, the adjusting circuit was in particular designed for exactly one HiKey960 development board. This board was equipped with the necessary firmware to flash and boot Zircon while two other HiKey960 boards were prepared for booting Linux. After switching to Linux development and thus to another HiKey960, there were again issues with level adjustment which were expressed by an unreliable or not at all working LCD while the signals on the I²C bus captured by an oscilloscope were fine. The error search indicated that the output levels for I²C at all tree available HiKey960 development boards differ in the range of 0.5V. Accordingly, the two HiKey boards running Linux did not get proper levels to detect a logical *0* respectively a *1* depending on the direction of the deviation. As an adjustment for another board would have resulted in unreliableness on the Zircon side, the decision was made to port the already existing Linux driver to the Raspberry Pi development board (version 2 or 3). It is running on +5V per default which makes any level adjustment obsolete. A dynamic switching between Zircon and Linux on the exact same HiKey960 board was not possible. The diverging firmware needed to boot either Zircon or Linux was too error-prone in the setup. Likewise, a complete change to the Raspberry Pi as a development platform

[2]wiki.seeedstudio.com, visited on 15.06.2019 `http://wiki.seeedstudio.com/Grove-LCD_RGB_Backlight/`

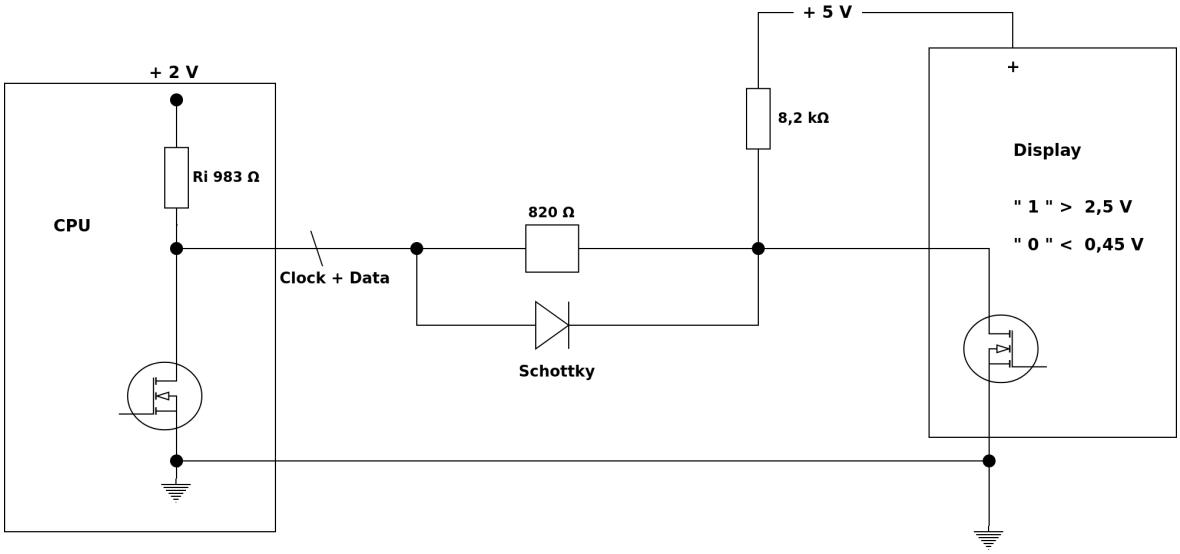

Figure 3.5: Level Adjustment for the Grove RGB LCD with the HiKey960 Development Board

was not possible, since Zircon does not support the board (anymore[3]). The final setups used for Zircon driver development and the remaining Linux development are pictured in figure 3.6 (Zircon) and figure 3.7 (Linux). This change does not influence the driver development in any manner.

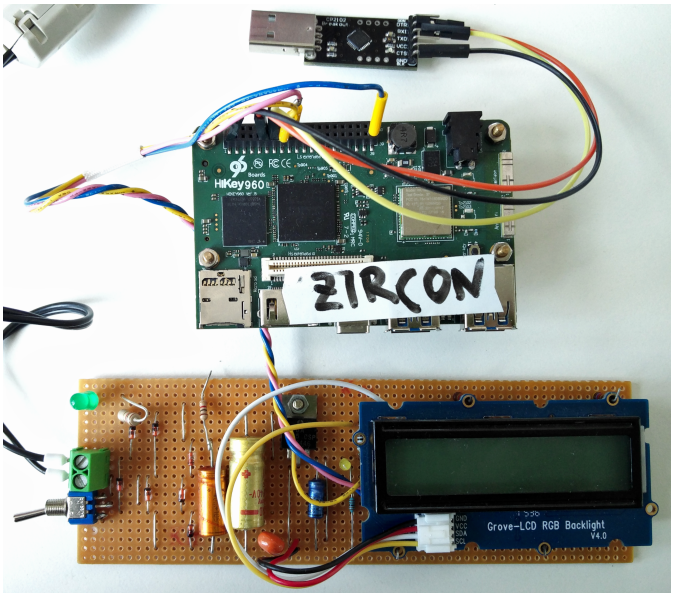

Figure 3.6: Final Development Setup for Zircon using the HiKey960 Development Board

[3] https://github.com/Allegra42/zircon/commit/5dc89c3f67808804c0c7d1bd9a0df3703d961ce6#diff-9bd3cb9d38dba050f310f03d18bbb2cf

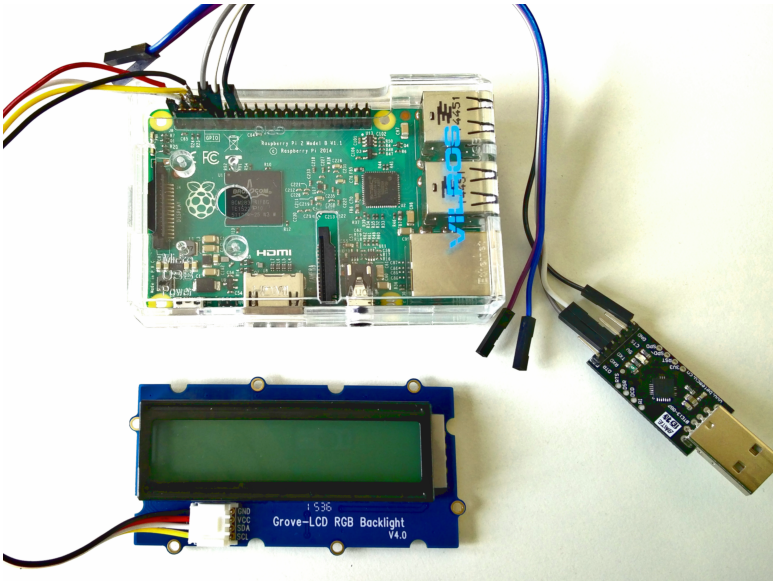

Figure 3.7: Final Development Setup for Linux using the Raspberry Pi Development Board

3.3.2 Software Development Setup

Linux

Initially, the Linux driver development was based on a mainline kernel in version 4.9. Due to the needed platform change, the already existing driver was ported to the Raspberry Pi Linux kernel tree in version 5.0. For a stable source code base over this thesis' duration, the kernel's source repository was forked and all development done in a branch of the fork. The repository is publicly accessible on GitHub[4].

The Raspberry Pi runs a standard Raspbian Linux, just the kernel respectively the driver module need to be changed during development. Corresponding scripts to build kernel and modules for a Raspberry 2 or 3 running Raspbian and flash the binaries to a prepared SD card are part of the repository, too. By switching to the specific Linux kernel tree for the Raspberry Pi, there is no default support for using **Clang** as compiler anymore. Some device specific changes in this kernel tree impede the Clang build and would need manual adjustments. The use of Clang related tools, especially *ClangFormat* is not affected.

Within this work, the Raspberry Pi runs without an attached display besides the Grove-LCD RGB peripheral. The development board is accessed using its physical Universal Asynchronous Receiver Transmitter (UART) interface with a USB to serial Transistor-Transistor Logic (TTL) adapter which allows the connection to a terminal session.

[4] `https://github.com/Allegra42/linux-rpi`

Zircon

The Zircon kernel consists of the actual kernel, a bootloader, system modules, third-party modules and scripts. *System modules* include necessary system facilities like the already mentioned device manager, but also the effective device drivers, or system-relevant user applications. Zircon is not built in versioned releases, yet. To work on a stable code base, nevertheless, the Zircon repository on GitHub was forked in December 2018. Today, due to restructurings from Google, the origin repository is no longer available and even the standalone Zircon code moved into the Fuchsia source tree on GoogleSource[5]. Accordingly, the driver development but also this work in general refer to the forked Zircon source tree[6].

In contrast to Linux, Zircon is booted as a standalone system for this thesis. The Fuchsia userland is not used at all. Thus, the tooling of the running kernel is limited and just a subset of the console commands and interactions known from Linux or Fuchsia is available. This concerns i.a. the Unix-like tool `cat` which is, for example, used to read from device files, while `echo`, which is used to write into device files, is available. As an alternative option, an implementation of the older Unix tool `dd` can be used within the pure Zircon kernel.

For the development of Zircon including drivers and operating system related userspace modules both compilers, GCC and Clang, are available. While GCC is default in most situations and scripts, e.g. in the built script for the HiKey960 which is a native part of the source tree, the surrounding tools like code formatting or linter are clearly based on Clang and may require a Clang build to work.

Just like Linux, the Zircon development setup is accessed via its UART interface. The setup is the same as for Linux, no matter whether using a Raspberry Pi or the HiKey960. Without Fuchsia as an userland, Zircon is not running a full graphics stack at all and thus, would not be able to show a Graphical User Interface (GUI) in contrast to Linux respectively Raspbian. As the HiKey960 is not equipped with a physical ethernet interface, UART is the only way to interact with Zircon.

3.4 General Driver Concept

In general, the driver concept for both, Linux and Zircon should be very similar because the Grove-LCD RGB backlight as peripheral device specifies them and stays the same on both platforms. However, both operating system kernels add some requirements, too.

Nevertheless, the first decision for the driver concept is caused by the Grove-LCD RGB backlight's nature of combining two distinct device controllers into a single peripheral device. Especially in Linux, two distinct I^2C devices, i.e. two controllers and thus two slave addresses, required distinct device drivers until kernel 4.9. Only later Linux kernel versions allow a combined driver from two or more related I^2C slave addresses by providing an appropriate API. Zircon also allows combined I^2C drivers in

[5] `https://fuchsia.googlesource.com/fuchsia/+/refs/heads/master/zircon/`
[6] `https://github.com/Allegra42/zircon`

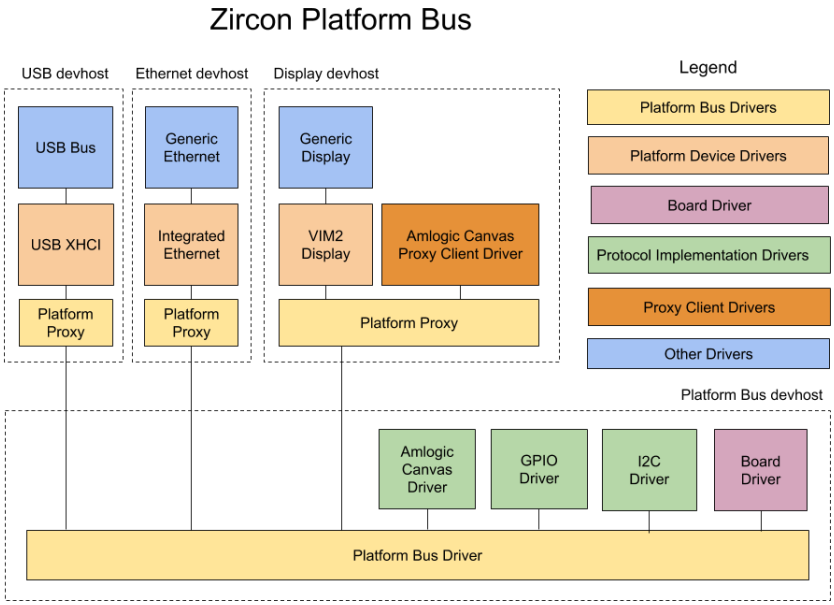

Figure 3.8: The setup of the Zircon Platform Bus on ARM64 based devices[35]

the used version level, but only on so-called *platform devices*. The corresponding term *platform bus* describes the Zircon driver for a framework that manages several low level drivers on ARM64 system architectures. An I²C driver is in this context a *protocol implementation driver* and thus a part of the *platform bus*. Such a driver is running in the same device host as the bus driver itself[35]. The situation is illustrated in figure 3.8 which is also a part of the official Zircon documentation[7][35].

However, the possibility to write a combined driver for both parts of the Grove is right now exclusively limited to platform devices. Indeed, the development setup using the HiKey960 and the Grove-LCD RGB backlight device fulfills this requirement but in contrast to Linux the needed API is not fully available.

For this work, the decision was made to use the new API for a combined driver on Linux. It is still new and rarely used right now but interesting, globally available but above all, it is more meaningful for devices which are consisting from more than one I²C controllers, such as the Grove RGB LCD backlight device. Perspectively, it is to be expected that more similar device driver implementations will switch to this API. One positive consequence of switching is the need of less boilerplate code. For example it is not longer needed to write two or more pretty similar initialization routines for each partial driver. In the device filesystem `/dev/`, the Grove peripheral also shows up as a single device which corresponds more closely to reality than distinct part devices. A further consequence of this decision results for an user's access to the driver. The standard file operations `read()` and `write()` are no longer meaningful for the device. Using distinct drivers, especially for the LCD, both of them are meaningful and thus useful, but already the meaning for the RGB part is more complicated. While it is

[7] `https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/ddk/platform-bus.png`

very clear that a string which is sent to a line display should be shown, to a RGB backlight device is not that meaningful. It requires a certain non-intuitive format to be understood. By implementing distinct drivers, this is to some degree possible without losing meaning and the intuitive nature of especially the `write()` call, but a combined driver would imply a sophisticated string parsing to differentiate between both devices. Thus, the usage of `write()` would not be intuitive for users anymore. As a result, implementing those calls does not make sense for this kind of driver. Instead, `ioctl()` commands or specifically defined pseudo-files in sysfs or procfs are a more conceivable solution for the Linux driver.

In Zircon, the situation is very similar to Linux. But the API for a combined driver is not available on x86_64 platforms. Using the combined *platform device API*, implementation standard file operations are, like in the Linux case, not meaningful. Besides, the way file operations are implemented in Zircon is not considered as known as it is for Linux. Thus, both variants, the two driver versions which are also usable on x86_64, but also the platform device way, are interesting for this work and should be considered. Using the HiKey960 as a hardware platform enables both. While the standard file operations are available on both, Linux and Zircon, the further interfaces differ between them. Zircon offers with a FIDL defined API a very sophisticated option for driver specific interfaces which is comparable to the also available I/O-controls but there are not equivalents to sysfs or procfs. To keep the drivers on both systems comparable and limit the development effort, they are not further considered in this work. Of course, even more comparable would be the use of `ioctl()` on both operating systems, but this would break the idea of writing idiomatic code for both platforms. According to the documentation are interface definitions using FIDL clearly preferred in Zircon[34]. Although the trend in Linux goes away from `ioctl()` to implement meaningful virtual file entries for sysfs the former ones are still accepted and widespread while Zircon clearly prefers the use of FIDL in such a situation. Furthermore, the virtual entries below a sysfs record are hardly comparable to I/O-controls or FIDL in terms of defining type-safe call signatures with several arguments and types.

Additionally, Zircon enables driver development in different languages. For the source code version used within this work, these are *C* and *C++*. Using C as a programming language for drivers implies good documentation and lot of examples within the code. C++ is less documented in the used version but also in Google's master repository. Nevertheless, documentation and code are clearly in favor of C++ drivers and more and more C drivers are moved to C++. Accordingly, both ways are very attractive for this work, C++ in particular as drivers in this language are a great difference to Linux ones.

The chosen compiler does not influence the actual driver development to a major degree, but it does for the system. Most of all, the error messages, diagnostics and some internal details are the obvious parts that may differ between GCC and Clang. They may also influence the output, but syntax errors are found on both of them. Hence, the possibility to compare two builds from different compilers is at all very useful, e.g. in cases of compiler errors, but also to compare code sizes, binary or build performances or code portability. On Zircon, Clang is slightly preferred as a respective build is needed

for surrounding tools while GCC is often default for build scripts. GCC, on the other hand, was very closely interlocked with Linux for a long time. But also in this case, the opportunity to compare the outputs of two compilers is helpful and the work that was done to decouple Linux and GCC to enable Clang is a positive one. Another reason for the use of Clang in Linux is, like for Zircon, the surrounding tools e.g. for style checking, linting, static and dynamic code analysis or address sanitizing[9]. However, without a closer look into the generated binaries a well-founded choice of the compiler for the pure driver development is rather difficult. Due to the change to the Raspberry Pi as a hardware platform for Linux driver development, Clang can not longer be used out of the box for compilation but before this change, both compilers and resulting kernels were used. The same is and may should still be done for Zircon, e.g. to see changes in the tooling of both compilers.

The last but important issue for a general driver concept is the question of the implemented functional scope. The Grove-LCD RGB backlight as a peripheral device is rather complex, not only because of its two controllers. It is also on account of its LCD controller and its complex and detailed internal instruction set. Already the Arduino library provided from the manufacturer for this device defines about 14 LCD methods. Implementing all of them as a part of the drivers in all Linux and Zircon variants with meaningful user interfaces would go beyond the scope of this work without adding a value. Thus, all drivers should implement the same basic functionality to show general driver concepts. Because of the decision to write only a combined driver in Linux, this variant lacks support for `read()` and `write()` but the additional defined interfacing options for the user, i.e. `ioctl()` for Linux and *FIDL* for Zircon, stay the same. As a result, the Zircon *platform device* drivers represent the same functionality as the Linux driver. Respectively, a combined driver should provide in each case a way to:

- set the backlight color,
- get the backlight color,
- clear the shown text on the LCD,
- write the first LCD line,
- write the second LCD line,
- read the whole LCD content and
- get the size of a LCD line.

The function arguments for these operations should be the same on both operating systems as far as it is permitted by the used interface mechanism. With these selected functions, the driver implementations should cover the exchange of the most important data types between user and kernel space in both directions. Not depicted is e.g. memory mapping or DMA as there is no meaningful implementation for the device.

The variants using two distinct driver implementations provide exactly the same functionality respectively the same interface definition but divided between both controllers. Thus, the first two named requirements belong to the RGB driver while the

remaining ones are part of the LCD driver. Additionally, these driver types should implement `read()` and `write()`. On the LCD side, they should write the user's input to the device file at the LCD's first line for `write()` and read the entire content of the LCD for `read()`. Hence, their implementation is very similar to the explicitly defined device operations above. In Linux, this would mean the implementation of e.g. writing the first line as a part of `ioctl()` would not differentiate more profoundly from the one for `write()`. Only the function signature and thus the copying of data between user and kernel would be slightly different but not the general file operation specific issues in Linux, the used helper functions and the actual invocation of the peripheral. This does not justify the development of a whole second driver variant in Linux, but the differences in these functions should be mentioned as a part of this work, nevertheless. These methods are more interesting for this work, especially in the Zircon C++ implementation.

3.5 Linux Driver Development

3.5.1 Prearrangements

To build a Linux driver, various options are available. It can be built as a permanent part of the kernel as a *built-in* driver or as a *module*. For this work, the first choice should not play a major role, but the source location. Carefully developed as a module within the kernel sources, the change to *built-in* is nothing more than a build configuration. Accordingly, the driver must be placed into the internal kernel structure in a meaningful manner. Linux is organized into *subsystems* to structure its monolithic architecture into functionally related modules. Within the *drivers* directory, there are several subsystems which are more or less meaningful for the Grove display. The probably most appropriate subsystem for this device is *auxdisplay*, i.e. auxiliary display which groups a small amount of drivers for small additional displays e.g. LCD's. Currently, the auxdisplay subsystem consists of only nine drivers. Each one is made from a single C code file and thus, not further divided into sub-directories for the drivers. The Grove driver must be to be inserted in this structure. Therefore, the driver source file is placed in `drivers/auxdisplay` and its build rules are added in the general `Makefile` for this subsystem. To configure the build, i.e. if the driver should be built as part of the kernel or as a module or none at all, it must also be registered in the subsystems `Kconfig` file. Listing 1 shows both entries in a combined view. The actual configuration is done via device specific configuration files, e.g. `arch/arm/configs/bcm2709_defconfig`. Mostly, these long files are not directly modified. Instead, the command `make menuconfig` is used as a front-end. It parses the `Kconfig` files within the whole kernel and represents them in their tree structure. The Grove driver for example, is part of the `auxdisplay` subsystem. Thus, general support for auxdisplays must be enabled to build also the Grove driver. The help text shown for the driver configuration shown in listing 1 mentions the path `include/uapi/linux/grove_ioctl.h`. It contains the custom `ioctl()` call definitions. They should be accessible from both, kernel and userspace. Thus, the definition

```
1  # Make Entry
2  obj-$(CONFIG_GROVE)                    += grove.o
3
4  # Kconfig Entry
5  config GROVE
6  tristate "Grove-LCD RGB backlight v4.0 Support"
7  default m
8  ---help---
9       Enable support for the Grove-LCD RGB backlight v4.0 via I2C.
10      The device is accessible via "/dev/grove" char device.
11      For the interface see "include/uapi/linux/grove_ioctl.h".
```

Listing 1: Build Configuration for the Grove-LCD RGB backlight driver

for this and other drivers are collected under this path.

The Linux driver discussed in this section is a complex software implementation. Even if its nature, consisting of two distinct device controllers, make the development more interesting, it is neither necessary nor possible to talk about the device specific implementation in all details within this work. Thus, only specific code snippets are discussed. The full Linux driver source code is available at `https://github.com/Allegra42/linux-rpi/blob/rpi-5.0.y/drivers/auxdisplay/grove.c`.

3.5.2 Driver Initialization and Exit

For the Grove driver, there is no need for additional one time initialization that is commonly done within the regular `module_init()` call. As already mentioned, the Linux kernel provides macros instead to register the driver without unnecessary boilerplate code. For an I²C driver as the Grove is, the matching macro is `module_i2c_driver()` (see listing 2, line 17). It takes an `i2c_driver` structure (see listing 2, line 8) as an argument which contains function pointers to the driver implementation of `probe()` and `remove()`, but also driver specific information like its name and a further pointer to an `of_device_id` table. This table defines a set of properties, e.g. so-called *compatible strings* (listing 2, line 2). As the Raspberry Pi is an ARM architecture, its actual hardware configuration and thus also firmly attached devices like the Grove are defined in a specific file, the *device tree*. Within the device tree, an entry which defines the relevant information about the Grove-LCD RGB backlight device is needed. Listing 3 shows this definition. From the system's view, this information is the superior I²C node the device is attached to and the device's addresses on the bus. Besides, for the matching between the actual device and the driver the *compatible string* is a crucial information. It is defined according to a defined scheme of the manufacturer and the device name (see listing 3, line 6). To take advantage from the new Linux API which enables a single driver for combined I²C devices from two or more bus slaves, their addresses and subnames must be defined in a specific manner shown in listing 3, line 7

```
1  static const struct of_device_id grove_of_idtable[] = {
2      { .compatible = "grove,lcd" },
3      {}
4  };
5
6  MODULE_DEVICE_TABLE(of, grove_of_idtable);
7
8  static struct i2c_driver grove_driver = {
9      .driver = {
10         .name = "grove",
11         .of_match_table = grove_of_idtable
12     },
13     .probe_new = grove_probe,
14     .remove = grove_remove,
15 };
16 %
17 module_i2c_driver(grove_driver);
```

Listing 2: Driver Initialization Sequence using `module_i2c_driver()`

and 8. Defining, in this case, both partial I²C devices within an own node would not work in the context of this API. Unfortunately, this special requirement is rarely documented right now, maybe because the use of this feature within the mainline kernel is still very limited.

To come back to the tasks of `module_i2c_driver()`, the focus returns to the way a module is registered in the system. Thus, the function pointers in listing 2 line 12 and 13 are also decisive. The structure given to `module_i2c_driver()` contains, besides the matching related entries, function pointers to the driver's implementations for `probe()` and `release()`. After registering the driver by the system using this structure and the macro, exactly this driver's `probe()` is called for a device with a matching compatible string. Indeed, the device tree used for ARM architectures is at first a static structure, but it can be extended at runtime using *device tree overlays* which provide a kind of *hotplugging* for device types like I²C on ARM. The last and unusual issue in listing 2 is the structure's entry for the `probe()` function in line 13. It is `probe_new()` instead of `probe()`, but the reason for this is simple. The current function signature for probing an I²C device will change in the future and `probe_new()` already takes a function pointer to the new one. As there is no need for the now deprecated one within this driver, the new function signature should be used.

Using the `module_i2c_driver()` macro also means `exit()` does nothing special and thus, it is not further mentioned.

3.5.3 Device Probing and Releasing

As already discussed, the `probe()` implementation in listing 4 already uses the new method signature. The previously used second argument, the

```
&i2c1 {
  pinctrl-names = "default";
  pinctrl-0 = <&i2c1_pins>;
  clock-frequency = <100000>;
  grovei2c: grove-i2c@3e {
    compatible = "grove,lcd";
    reg = <0x3e 0x62>;
    reg-names = "grovelcd", "grovergb";
  };
};
```

Listing 3: Device Tree Configuration for the Grove Peripheral Device

`struct i2c_device_id *id`, is not needed for this driver. As shown in figure 3.1 the driver's implementation of probe is roughly divisible into two parts: the driver's own setup including its registration within the system and the device specific initialization. The first part is very similar for many drivers. It includes i.a. allocating private memory regions and device numbers but also creating entries in sysfs and devfs. Often, a driver supports a few similar devices or device revisions which differ in their needs e.g. for setting them up or their instruction set. In such a case, the driver must check within `probe()` if the probed device is actually a supported one and how it must be handled. The grove driver only allows a single compatible string. A decision between two similar devices is not needed within this work. The whole setup is under control, a similar device which would require a distinction will not be added. Also, an additional testing if the device matches is redundant as long as the driver fails if the addresses defined within the device tree are not detected on the bus or the specific initialization failed.

The `probe()` function is the first function of this driver which must be *re-entrant* and aware of more than one device to handle. Thus, a structure for device specific and private data is needed. It is globally defined within this driver and should contain the needed information to restore a device and its state from each possible driver entry point. For the Grove peripheral, it is e.g.

- the device number `dev_t devnum`, which merges *major* and *minor* numbers,
- the character device `struct cdev cdev`,
- pointers to the I^2C device instances `struct i2c_client *client` for both, RGB and LCD and
- device specific private data to restore the device's state, i.e. a struct to store color information and char buffers for the LCD content.

The reasons for storing exactly these information within the struct defined as `struct grove_t` become clear in the course of this driver's analysis.

```
1  static int grove_probe(struct i2c_client *client)
2  {
3    int ret = 0;
4    struct grove_t *grove;
5    struct device *dev = &client->dev;  // Store I2C client device
6    struct device *device;      // Resulting device from creating a devfs entry
7
8    grove = kzalloc(sizeof(struct grove_t), GFP_KERNEL);
9    if (IS_ERR(grove)) {
10     dev_err(dev, "failed to allocate a private memory area for the device\n");
11     return -ENOMEM;
12   }
13
14   grove_class = class_create(THIS_MODULE, "grove");
15   if (IS_ERR(grove_class)) {
16     dev_err(dev, "failed to create sysfs class\n");
17     return -ENOMEM;
18   }
19
20   if (alloc_chrdev_region(&grove->devnum, 0, 1, "grove") < 0) {
21     dev_err(dev, "failed to allocate char dev region\n");
22     goto free_class;
23   }
24
25   cdev_init(&grove->cdev, &grove_fops);
26   grove->cdev.owner = THIS_MODULE;
27
28   if (cdev_add(&grove->cdev, grove->devnum, 1))
29     goto free_cdev;
30
31   device = device_create(grove_class, NULL, grove->devnum, "%s", "grove");
32   if (IS_ERR(device)) {
33     dev_err(dev, "failed to create dev entry\n");
34     goto free_cdev;
35   }
36
37   /* continued with device specific initialization */
38 }
```

Listing 4: Device Probing

As the items within the `grove_t` structure should be filled during the `probe()` for exactly the device it is called with, it is useful to allocate the needed memory on the heap right from the start using `kzalloc()` (see listing 4, line 8). Of course, the resulting pointer needs to be checked. In case of a failed memory allocation, the driver probing must be aborted with a corresponding error (see listing 4, line 9 to 12). The next step is creating a *class* entry within the sysfs for the Grove device. For example with the name *grove* as shown in listing 4, line 14. It is not only a good habit to create sysfs entries but also needed for the automated assignment of device numbers. If this

part is skipped, the entry in devfs including major and minor numbers must be created by hand. Of course, the resulting class pointer must be checked as well to avoid further errors, e.g. if the class could not be created for some reasons.

Accordingly, the following call to `alloc_chrdev_region()` actually registers a range of device numbers for the asking device using the same name as for the class creation (see listing4, line 20). For a single device, only one device number is needed. This is the meaning of the second and third argument in the call. The device number is unique for each device and thus helps to identify a device. For this reason, the resulting assigned number of the type `dev_t *`, is stored as part of the `grove_t` structure (see first method argument in listing 4, line 20). As before, a check of the operation's return value and error handling in the case of a failure during the allocation is needed.

In the following step, the character device structure `cdev` is initialized for this device using `cdev_init()` in listing 4, line 25. Its first argument is a reference to the empty struct which is a part of `grove_t`. The second argument references the `struct file_operations` which contains function pointers to the implementations of this driver's file operations, the driver entry points. The structure is not directly shown as part of a listing as there is nothing special about it. As defined in the general driver concept in section 3.4 the driver should be accessible via I/O-controls and thus, the struct contains a function pointer to its implementation. Additionally, it holds pointers for the owner but also for open and release. The `.owner` is pointing commonly to `THIS_MODULE`. The reason for implementing `open()` and `release()` even if there is no need for an access control for this device is related to restoring the right device instance if the driver is entered via a file operation like `ioctl()`, `write()` or `read()`. Its underlying issue is further discussed as a part of the following section. In the context of `cdev_init()` two further actions are needed: Defining the cdev's owner (see listing 4, line 26) and adding the cdev structure, and thus the represented character device as well, including its device numbers to the system (see listing 4, line 28).

As a last action within this part of `probe()`, an entry for the probed device is added within devfs (`/dev/` using `device_create()` (see listing 4, line 31). It takes the class created in sysfs as a first argument. The second one would be the parent device for the newly created one. As there is no matching one for the Grove-LCD RGB backlight device, the argument stays `NULL`. The creation of the device in devfs is only one place where the device number is needed in this function. Thus, it was necessary to create it before and save it as a part of the `grove_t` structure. The last arguments for this call are a format string to define the device' name within devfs and a number of strings or numbers to fill it. In this case, it is just the string `"grove"`. As before, this call needs error checking and handling as well.

The device specific part of `probe()` starts with saving the I^2C client which was delivered as an argument within the private `grove_t` structure for later use. They are needed to perform the I^2C transfers. Thus, it is essential to store them for later use in this driver. As listing 5 states in line 5, the client structure is saved as `grove->lcd_client`. But how is it known that the structure delivered represents the LCD and not the RGB controller? To answer this, the device tree definition shown in listing 3 is needed. The

```
1  static int grove_probe(struct i2c_client *client)
2  {
3    /* continued */
4
5    grove->lcd_client = client;
6    i2c_set_clientdata(client, grove);
7    ret = grove_init_lcd(grove);
8    if (ret) {
9          dev_err(dev, "failed to init LCD, free resources\n");
10         goto free_device;
11   }
12
13   grove->rgb_client = i2c_new_secondary_device(grove->lcd_client, "grovergb", 0x62);
14   if (grove->rgb_client == NULL) {
15         dev_info(dev, "can not fetch secondary I2C device\n");
16         goto free_device;
17   }
18   i2c_set_clientdata(grove->rgb_client, grove);
19   ret = grove_init_rgb(grove);
20   if (ret) {
21     dev_err(dev, "failed to init RGB, free resources\n");
22     goto free_device;
23   }
24
25   return 0;
26   /* jump marks for error handling (not listed here) */
27 }
```

Listing 5: Device Probing (Grove Specific Part)

order of the controller addresses as defined within this description affects how the device is passed to the driver's `probe()` implementation. The LCD's address 0x3e was defined at first, thus it is passed as major device to `probe()`. All following devices are not directly passed. The driver handles them as so-called *secondary devices* and references them using their address as well as their given name defined in the device tree.

But first, the major I²C device must be initialized before adding the secondary one. Before doing this, the `grove_t` structure which contains now the I²C client for the LCD controller as well, is set to the `void *driver_data` pointer of the `struct device` which is itself a part of `struct i2c_device`. Some calls to this driver contain exactly this structure as an argument which allows to restore the device for use. Using global variables to store the data of all possible devices is avoided in this way. Additionally, it is a much more flexible way. Global definitions would not allow dynamically changing quantities of devices controlled by a driver and in the worst case, unnecessarily allocated memory. The assignment to `void *driver_data` is done via the `i2c_set_clientdata()` call shown in listing 5, line 6.

```
static int grove_init_lcd(struct grove_t *grove)
{
    int i = 0;
    int ret = 0;

    struct i2c_cmd_t cmds[] = {
      { LCD_CMD, 0x01 },  // clear display
      { LCD_CMD, 0x02 },  // set cursor position to home
      { LCD_CMD, 0x0c },  // enable display | show no cursor
      { LCD_CMD, 0x28 },  // enable two line mode
    };

    mutex_lock(&grove_mutex);
    for (i = 0; i < (int)ARRAY_SIZE(cmds); i++) {
        ret = i2c_smbus_write_byte_data(grove->lcd_client, cmds[i].cmd,
              cmds[i].val);
        if (ret) {
            dev_err(&grove->lcd_client->dev,
                "failed to initialize the LCD\n");
            goto fail;
        }
    }

    char init[] = "@Init";

    ret = i2c_master_send(grove->lcd_client, init, sizeof(init) - 1);
    if (ret < sizeof(init - 1)) {
        dev_err(&grove->lcd_client->dev,
            "failed to initialize the LCD\n");
        goto fail;
    }
    ret = 0;

fail:
    mutex_unlock(&grove_mutex);
    return ret;
}
```

Listing 6: Controller-specific LCD Initialization

Its counterpart, `i2c_get_clientdata()` is used within this driver as well, as a part of the opposite of device probing: the `release()` function. The device initialization itself is outsourced to an own function. Within this thesis, only the LCD initialization is described. The RGB one is straight forward using the commands of the respective controller and as well as the LCD initialization rather device specific than relevant for driver development consideration.

The LCD init function takes the `grove_t` structure as an argument. This is needed to access the I^2C client stored within which is needed for the actual communication with the device. Of course, it would be possible to use `struct i2c_client` as an argument

as well, but in this case, the function call would change between LCD and RGB initialization. As `grove_t` stores both I²C client structures, it is a comparable way. Within the function itself, the initialization sequence for the LCD is traversed. The sequence is made from specific values that need to be written to specific registers on the controller. The exact sequence depends on the device and is in general specified in the corresponding documentation. For the Grove LCD, several options for the sequence are inspired by the manufacturer's documentation for using the device e.g. the Raspberry Pi and Python[8]. The way the commands are defined and sent to the device is instead already inspired by the way it is done in *Zircon* (see listing 6, line 6 to 22). There, a structure is defined which describes the way a command is constructed, in this case, it is made from two `uint8_t`, the command register and the value which is to be set. Using such a struct enables easy readability but also to loop over the commands in a graceful manner to send them to the device. The communication between driver and device is a *critical section*. It is not meant to be entered by more than one process at the same time. The initialization should be only entered once per device, indeed, but is still secured with a mutex as a precaution. Within the loop, the actual transfer between driver and device is done using the kernel internal API `i2c_smbus_write_byte_data()`. It writes exactly one byte of usable data, the value within the struct, to a specified controller register. For all of these setup commands, the register is 0x80 what the name `LCD_CMD` stands for. As a result, the loop iterates over all tuples made from a register and a value and sends them to the `struct i2c_device`pointer given as a first argument of the call (see listing 6, line 14 to 22). For the LCD, this way is only used to setup the device or prepare other writes, but not for the actual text writes themselves. Using structures for text has *padding effects* as a result. Thus, a more sophisticated way is to encode the target register on the controller as a part of the string to be sent and use so-called *block writes* instead of sending single bytes at a time. It instrumentalizes the effect a block write like `i2c_master_send()` interprets the first sent byte as a controller register. In case of the Grove device, the corresponding register is 0x40 which is interpreted as an @ in American Standard Code for Information Interchange (ASCII). For this initialization, writing a string to the LCD is not necessarily needed. This part could also been skipped, thus, but it illustrates the way strings are written inside the whole driver. The call to the kernel API `i2c_master_send()` is as well slightly different from the one used previously (see listing 6, line 26). It also takes the `i2c_device` structure as a first argument, but the second one is a pointer to the char buffer which is to be sent. In this context, it is `char init[]` defined in listing 6, line 24. The last argument is the number of bytes which are to be sent. Usually, this would be the string size including `\n`. But because the LCD would try to print this character as well, it is cut of by decreasing the size by one. Unlike the previous call `i2c_master_send()` does not return a status code but the number of successfully transferred bytes. Thus, the error handling must check for this. In each case, the mutex must become unlocked before the function is left. Using jump marks for error handling simplifies this. In contrast to ordinary application development they are gladly used within Linux drivers because

[8] wiki.seeedstudio.com, visited on 12.05.2019 `http://wiki.seeedstudio.com/Grove-LCD_RGB_Backlight/`

error handling becomes more readable than by using e.g. if-else constructs. However, only linear onward jumps are allowed to ensure this.

After the device initialization itself was considered for the LCD as an example for both parts of the Grove-LCD RGB backlight peripheral, the focus should come back to the anomaly of this driver, the handling of the second I²C controller. While the `struct i2c_client` for the LCD is given as an argument of `probe()` (see listing 6, line 5), the one for the RGB part must be requested using the rather new `i2c_new_secondary_device()` API of the Linux kernel (see listing 6, line 13). It is a helper function to fetch the second defined address in the device tree and create the associated device representation. Its first argument is a pointer to the primary I²C client, in this case, the LCD. The second one is the symbolic name given in the device tree (see listing 3). Usually, the slave address is parsed from the corresponding address to this name, but if there is an issue, the helper function tries to create the secondary device using the default address specified as a third argument. The resulting `struct i2c_client` is stored as a part of `grove_t` as well. Further steps for the RGB device are the same as for the LCD one (see listing 6), but the exact implementation of the device specific initialization is of course different and meaningful for the RGB device.

3.5.4 Driver Interfaces

The Linux driver is, as designed in section 3.4, accessed via I/O controls and the section about this driver's prearrangements 3.5.1 already mentions they are located besides the actual source code, within `include/uapi/linux/grove_ioctl.h`, to be accessed from both sides, user and kernel. Essentially, I/O control call definitions are nothing more than numbers and using pure numbers on both sides works as well. But it is error-prone and does not even allow Linux to perform the basic checks, e.g. if the size of transferred data matches the definition for these calls. However, a well engineered `ioctl()` interface needs special care from the programmer. The Linux kernel provides macro definitions and a scheme to define meaningful and checkable `ioctl()` calls with non-conflicting numbers, but thus, they must be used. The way used within this work is shown in listing 7. It corresponds to the recommendation given in *Developing Linux Device Drivers*[26], the accompanying script to a Linux Foundation class. To encode the number including the data transfer direction and the call parameters the macros `_IO, _IOR, IOW` and `_IORW` are available. For example, `_IOR` means the user reads data from the kernel. Each I/O control definition is made from a *type*, a *number* and, if data is transferred, from the *size* of those data as well. The *type* is a kind of magic number which is used throughout the driver for creating the I/O control numbers. In this case, it is defined in listing 7, line 1 as `MAGIC 'M'`. The character is interpreted as its ASCII value, 0x4D, and represents a base value for the macro usage. According to the concept specified in section 3.4 the operations, which should be provided by the driver, are listed and enumerated from line 3 to 9 in listing 7. This number is the second argument needed for the usage of the macro. But the actual `ioctl()` numbers are not defined until line 12 in this listing. Only there, the actual macros are used.

```
1  #define MAGIC 'M'
2
3  #define SET_COLOR           0x01
4  #define GET_COLOR           0x02
5  #define CLEAR_LCD           0x03
6  #define WRITE_FIRST_LINE    0x04
7  #define WRITE_SECOND_LINE   0x05
8  #define READ_LCD            0x06
9  #define GET_LINE_SIZE       0x07
10
11 #define GROVE_SET_COLOR         _IOW (MAGIC, SET_COLOR, struct color_t)
12 #define GROVE_GET_COLOR         _IOR (MAGIC, GET_COLOR, struct color_t)
13 #define GROVE_CLEAR_LCD         _IO  (MAGIC, CLEAR_LCD)
14 #define GROVE_WRITE_FIRST_LINE  _IOW (MAGIC, WRITE_FIRST_LINE, struct string_t)
15 #define GROVE_WRITE_SECOND_LINE _IOW (MAGIC, WRITE_SECOND_LINE, struct string_t)
16 #define GROVE_READ_LCD          _IOR (MAGIC, READ_LCD, struct string_t)
17 #define GROVE_GET_LINE_SIZE     _IOR (MAGIC, GET_LINE_SIZE, uint8_t)
```

Listing 7: I/O Control Call Definitions

If the transferred data consists of more than one argument, as it is e.g. the case for setting and getting the Grove's backlight color consisting of a red, green and blue ratio, often a structure is used to describe the situation. Within this work, the template definition of the struct is passed as a size to the macro as well. In cases the transmitted value is a single one, it is enough to use only its type.

If no transfer at all is needed, e.g. if the command should only trigger a specific action without parameters or return values, the `_IO(type, number)` macro should be used. It does not define a *size* to transfer in contrast to the other ones (see listing 7, line 11 to 17).

Before diving right into the implementation of `ioctl()`, the question is answered why this driver needs to implement the file operations `open()` and `close()` even if there is no need for a sophisticated access control. The reason is in the function signature of I/O control, `ioctl(struct file *file, unsigned int cmd, unsigned long arg)`, more precisely in the `struct file`. It does not contain an entry for a private device data structure such as `grove_t` is or any other entry which enables the restoration of the related instance of this structure for the called device. Even if the signature slightly differs for `read()` and `write()`, it is the same issue with them. However, each time before one of these basic file operations is called, `open()` must be invoked as the user must open the file descriptor which represents the device. If it is not explicitly implemented by a driver, a default one is used. But the fact `open()` is called can be exploited to solve the underlying issue. Its call signature, `open(struct inode *inode, struct file *file)`, enables restoring `grove_t` via its entry `struct cdev *i_cdev`. To do so, the `grove_t` structure needs to contains a `cdev` entry as well.

```
1  static int grove_open(struct inode *inode, struct file *file)
2  {
3      struct grove_t *grove;
4
5      grove = container_of(inode->i_cdev, struct grove_t, cdev);
6      file->private_data = grove;
7
8      return 0;
9  }
10 static int grove_release(struct inode *inode, struct file *file)
11 {
12     file->private_data = NULL;
13     return 0;
14 }
```

Listing 8: Implementation of `open()` and `release()`

Having such a situation, the *magic macro* `container_of(ptr, type, member)` can be applied to restore the instance of the `grove_t` structure for the called device (see listing 8, line 5). The way this macro is implemented is not discussed as part of this work, but the blog article *The Magical container_ of Macro*[9] by Radek Pazdera gives a sound explanation. To make this `grove_t` instance effectively available within `ioctl()` (and/or `open()` and `close()`), it must be stored in a way that is accessible from these calls. As all of them share `struct file *file` as a call argument. It is the only possible location as well. The file structure contains an entry `void *private_data` which is intended for exactly this use-case (see listing 8, line 6). By using this trick, the pointer must be invalidated in `release()`, the counterpart of `open()` right before the file structure is destroyed by the kernel (see listing 8, line 12).

Within the `ioctl()` implementation, the device's instance of `grove_t` is restored as a first step (see listing 9, line 5). Its contents, especially the `i2c_client` instances for both controllers, are essential for this function respectively for the actual I^2C transactions. Before starting with the implementation of the individual commands, memory is allocated for the structures `struct color_t` and `struct string_t` as well as a preparation (see listing 9, line 7 and 11). These structures represent the backlight color, consisting of a red, green and blue ratio, and the LCD's content. They may become filled either from kernel or from user space depending on the actual command and thus, the data transfer direction and of course, they must be freed before leaving this function, even if it is not shown in the corresponding listing 9.

It is neither possible nor purposeful for the question of this work to consider all commands defined within the concept in detail. As the idea of writing to the LCD was already discussed during device probing, `ioctl()` is focusing on setting the RGB backlight respectively returning its state to the user. However, it is obvious that the setting of the LCD's content is not exactly the same as it was during `probe()`.

[9] radek.io, visited on 13.06.2019 `https://radek.io/2012/11/10/magical-container_of-macro/`

```c
static long grove_ioctl(struct file *file, unsigned int cmd, unsigned long arg)
{
  /* Definition of variables */

  grove = file->private_data;
  dev = &grove->lcd_client->dev;
  color = kzalloc(sizeof(struct color_t), GFP_KERNEL);
  if (IS_ERR(color))
    return -ENOMEM;

  string = kzalloc(sizeof(struct string_t), GFP_KERNEL);
  if (IS_ERR(string))
    return -ENOMEM;

  switch (cmd) {
  /* Some cases are skipped for this listing */
  case GROVE_SET_COLOR:
    mutex_lock(&grove_mutex);
    if (copy_from_user(color, (const void *)arg, sizeof(struct color_t))) {
      dev_err(dev, "copy from user failed\n");
      mutex_unlock(&grove_mutex);
      break;
    }

    struct i2c_cmd_t cmds[] = {
      { RED, color->red },
      { GREEN, color->green },
      { BLUE, color->blue },
    };
    for (i = 0; i < (int)ARRAY_SIZE(cmds); i++) {
      ret = i2c_smbus_write_byte_data(grove->rgb_client, cmds[i].cmd, cmds[i].val);
      if (ret) {
        dev_err(dev, "set new color failed\n");
        mutex_unlock(&grove_mutex);
        break;
      }
    }
    grove->color = *color;
    break;

  case GROVE_GET_COLOR:
    mutex_lock(&grove_mutex);
    if (copy_to_user((void *)arg, (const void *)&grove->color, sizeof(struct
    ↪  color_t))) {
      mutex_unlock(&grove_mutex);
    }
    break;

  /* Default case */
  /* Free allocated memory */

  mutex_unlock(&grove_mutex);
  return 0;
}
```

Listing 9: I/O Control Implementation

In this context, a static string was set which already contained the encoded target register on the controller. The related `ioctl()` calls instead receives user defined strings and thus, string operations to do a respective encoding. For the now considered RGB backlight device, the implementation is much more straight forward. Its first command defined in the concept and thus in the according Linux header is `GROVE_SET_COLOR`. In case the function's argument `unsigned int arg` matches the number behind this symbolic name, the associated implementation in the switch-case construct is called. As the commands are decoded as numbers, using switch-case to implement them is a common pattern. In each case, the error handling and ending of a command's realization must be done properly to avoid unattended effects from fall-through's if they are not explicitly desired in a situation. However, all of these drivers commands contain at least one critical command, either a data transfer from or to a user and/or an I^2C write transaction.

Therefore, the according instructions must be locked with a mutex. In cases an error occurred, it must be ensured the mutex is unlocked in any situation and the switch-case but also the function itself is left properly. The task of `GROVE_SET_COLOR` is to receive user input in a certain format defined by the structure `color_t` and set the device' backlight color accordingly. Both subtasks, but especially the operation on the I^2C bus, must be locked.

Nevertheless, the data transfer between user and driver needs special attention as well. As discussed in section 2.6.3 Linux use different address types between them. Thus, the call's argument `unsigned long arg` which is interpreted as an address to the user's buffer, is meaningless within kernel. Using it without further ado is possibly even dangerous. Instead, the buffer must be transferred to the kernel buffer `color_t color` which was allocated at the beginning (see listing 9, line 7). The `ioctl()` header definition specified the size of this structure as transfer size of this call and should be used as a reference for the maximum size to copy. For the actual transfer, the Linux kernel offers helpers. The call `copy_from_user()` which is used for this command, and its counterpart `copy_to_user()` that will be used as part of the next considered transfer. It takes the given function arg as an address in userspace and copies a given number of bytes, i.e. the defined structures size, to a specified memory region in kernel (see listing 9, line 19). According to the command definition in the header file (see listing 7), the transferred buffer should be of type `color_t`, and thus it can be interpreted accordingly. Within I/O control, a careful error handling is very important. If the transfer failed, it is necessary to unlock the mutex and leave the entire `ioctl()` implementation properly.

On a successful transfer, the received data is taken to set the color on the actual device. Similar to the LCD's setup known from device probing, a structure consisting of the target register's address and the value to set is defined. Using this structure as a base type, an array is created. Each RGB color ratio is set within an own register and thus, as an own entry within this array. Following the actual I^2C transfer is done as known from initializing the LCD during `probe()`. A loop iterates over the array and transfers each partial command tuple to the respective device controller (see listing 9, line 30 to 37).

Unfortunately, the controller does not allow reading the current values directly from the device. Thus, they must be stored alongside the private device instance data. The LCD ratios the same issue and thus, a private buffer to store its state as well must be used. Respectively, updating the device implies updating this state information, but it must not be changed before the I²C transferred was finished successfully to avoid wrong or inconsistent states (see listing 9, line 38). The next considered command, `GROVE_GET_COLOR` takes advantage of this state information. In this situation, there is no I²C read transfer in this I/O control command, but the buffer must be copied into user mode accessible addresses. As mentioned previously, the kernel helper `copy_to_user()` is invoked for this task (see listing 9, line 43). It differs from its counterpart only in the transferred direction. The `ioctl()` call argument `arg` is interpreted as an address in userspace again, but this time as a target address, while the `struct color_t` which is stored as a part of the instance data acts as source.

Even if the individual I/O control commands are implemented in a row within this function usually only a single on is executed each time `ioctl()` is called by a user. Thus, the developer must ensure the `break;` command is set correctly each time it is needed. The listing shown as a part of this work only consists of a snippet of the full implementation listed in the related Github repository[10], but is a recognizable complex task if the number of commands or the complexity of their implementation increases. To outsource them into their own functions might be helpful in these cases. Nevertheless, an `ioctl()` implementation requires a high degree of care and attention from a developer to avoid errors.

3.6 Zircon Driver Development

By allowing different programming languages for device drivers but also the variety of available and legit ways result in the fact that Zircon drivers can not be considered in every detail. In Linux, the situation was similar, but the options and resulting drivers in Zircon are noticeable more varied. This section will focus on the *platform device* driver variants, especially the C one to hold this discussion comparable to the Linux one, but the C++ driver implementation and its anomalies are considered as well because it is special compared to Linux. Nevertheless, the *two driver* variant should be mentioned, too. The differences between both ones are rather caused by the way the device is defined and initialized and thus, not crucial. Similar to Linux, this section will focus on selected driver aspects as well. The full implementations for all written driver variants are available in the corresponding GitHub repository[11].

[10] github.com, `https://github.com/Allegra42/linux-rpi/blob/rpi-5.0.y/drivers/auxdisplay/grove.c`

[11] github.com, `https://github.com/Allegra42/zircon/tree/i2c-grove-lcd/system/dev/display`

3.6.1 Prearrangements

Zircon drivers are always built as a part of the kernel and accordingly user system components at the moment. Within the corresponding driver directory `zircon/system/dev` are several subdirectories to group them based on their tasks. As there is no comparable one to *auxdisplay* on Linux, the best matching one is probably `display`. Each driver variant is located in `display` as an own subdirectory consisting of its source file and a `rules.mk`, the makefile. A dedicated and sophisticated build configuration as known from the Linux `Kconfig` system is not available. If a driver directory contains a valid `rules.mk`, it is built. Working with four different driver variant for the same device, this situation leads to expectable issues. Thus, the `rules.mk` of each currently not needed driver must be taken off the build, e.g. by commenting its content out.

As already discussed Zircon uses the Fuchsia Interface Definition Language (FIDL) rather than I/O controls. All of these definitions which refer to a module located within `zircon/system`, i.e. a system component that is not part of the actual microkernel, are collected within the directory `zircon/system/fidl`. According to the way a driver is modularized, one or more FIDL definitions are grouped in a subdirectory together with a `rules.mk` file. A special compiler generates interface definitions and bindings in the needed languages from the generic FIDL files. At the moment, these are most of all C bindings for both sides, a set for the use in drivers and a corresponding one for the use in user applications. Even if a driver is written in C++ there are currently only the C bindings available. The resulting effects are considered as a part of the following section.

3.6.2 Device Definition

In Linux, the way an I^2C device is defined for the system was discussed as part of driver initialization. A corresponding driver function exists but is hardly ever used and the whole binding mechanism in Zircon works slightly different. Zircon relies on so-called *board files* to define a computing platform such as the Hikey960 is. Linux previously did the same but changed to the already known *device tree* representation for ARM-based computing devices because the former concept did not scale well with the number of supported boards. The device tree files are a textual representation in an own syntax while board files are written in C on both kernels. For the actual use device tree files but board files as well are compiled to a binary representation. However, the device tree can be enhanced during runtime using so-called *device tree overlays*. Today, a comparable mechanism for Zircon board files is not known.

For the actual definition of the Grove-LCD RGB device for the Hikey960 running Zircon, the targeted driver language is not decisive but the kind of the driver. If both controllers are to be used in standalone driver implementations, e.g. necessary on non platform devices, they must be defined independently from each other as well. After all, a driver must be able to pick exactly the desired device controller. In cases a combined *platform device* specific driver is requested, both controllers must be accessible from the same binding sequence.

```
1  static const pbus_i2c_channel_t i2c_grove_rgb_channels[] = {
2    {
3      .bus_id = 0,
4      .address = 0x62,
5    },
6  };
7
8  static const pbus_i2c_channel_t i2c_grove_lcd_channels[] = {
9    {
10     .bus_id = 0,
11     .address = 0x3e,
12   },
13 };
14
15 static const pbus_i2c_channel_t i2c_grove_pdev_channels[] = {
16   {
17     .bus_id = 0,
18     .address = 0x62,
19   },
20   {
21     .bus_id = 0,
22     .address = 0x3e,
23   },
24 };
25
26
27 static const pbus_dev_t i2c_grove_lcd_dev = {
28   .name = "grove-lcd-i2c",
29   .vid = PDEV_VID_SEEED,
30   .pid = PDEV_PID_SEEED,
31   .did = PDEV_DID_SEEED_GROVE_LCD,
32   .i2c_channel_list = i2c_grove_lcd_channels,
33   .i2c_channel_count = countof(i2c_grove_lcd_channels),
34 };
35
36 static const pbus_dev_t i2c_grove_pdev_dev = {
37   .name = "grove-lcd-i2c",
38   .vid = PDEV_VID_SEEED,
39   .pid = PDEV_PID_SEEED,
40   .did = PDEV_DID_SEEED_GROVE_PDEV,
41   .i2c_channel_list = i2c_grove_pdev_channels,
42   .i2c_channel_count = countof(i2c_grove_pdev_channels),
43 };
```

Listing 10: Device Definition in Zircon Boardfiles

Listing 10 shows the way the devices are defined for both kinds of drivers. Line 1 to 13 shows the pure definition of single handled devices on the I^2C bus. They are described as an array of a structure consisting of the bus ID and the respective slave address. The definition for the combined device is shown in the same listing from line

15 to 24. It does not differ from above but combined both of those definitions in a single one, i.e. the array contains both structure elements instead of only a single one. More interesting is the way the device is actually defined in order to be bindable. The definitions listed so far do not contain any information that allows a matching between device definition and driver. Those are added in a second structure which contains a device name, but also entries for a *vendor ID (vid)*, a *product ID (pid)* and a *device ID (did)*. These IDs are decisive for the binding between device and driver. In order to include the device to be addressed in this structure as well, a pointer of the respective structure type is added alongside the number of actual device entries in the previous list (see listing 10, line 27 to 34 and 36 to 43). As a result, the device definitions for a single driver only contain one element in this `i2c_channel_list` while the device definition for the pdev driver holds two of them. Listing 10 shows only the LCD definition for the first type. The one for RGB is defined accordingly. But the idea behind this second device definition goes further. It does not only allow one or more I^2C devices but additional ones as well. This allows creating complex device descriptions, e.g. a device consisting of I^2C parts, General Purpose I/O (GPIO)'s, interrupts and others, and makes exactly this combination of devices accessible to a single driver. Within the Hikey960 board files, such an approach is for example used to describe the on-board GPU which is made from Memory Mapped I/O (MMIO), interrupts and BTI for DMA transfers.

Thus, the device description in Zircon is much more detailed but very descriptive. Unfortunately, there is hardly any documentation available on this topic and the binding mechanism in general. However, the pure definition of the devices is not enough. If these descriptions are not attached to the object representing the *platform bus* for the Hikey960 using the `pbus_device_add()` API, no binding is possible at all. The device is not known without it.

3.6.3 Driver Binding

The pure binding or matching mechanism between device definition and driver is the same for C and C++ in both variants as well because in this situation only C language bindings are available at the moment. Thus, the way a driver defines a matching device is considered for both language versions while the actual `bind()` implementation is discussed in the specific sections. Within C driver implementations, the necessary code lines are a part of the common driver source code, in C++ they are usually outsourced in a single `bind.c` file containing these lines and an extern definition of the C++ function signature for `bind()`. While the way for C drivers is well-defined and documented, the implementations of C++ `bind.c` files differ a lot within the available sources. This work only discusses the way used within this case study. The listing 11 shows such a binding definition, to be accurate, the one for the C++ LCD driver. It only differs by the one used for the C one by line 2 which is only needed for C++ and the C++ specific naming within the code. While Linux uses compatible strings for the matching between driver and device Zircon offers a more sophisticated mechanism based on macros. Listing 11 shows a rather basic example in the lines 10 to 14.

```
1  /* C++ specific line */
2  extern zx_status_t grove_lcd_bind(void* ctx, zx_device_t* parent);
3
4  /* Shared part */
5  static zx_driver_ops_t grove_lcd_cpp_driver_ops = {
6    .version = DRIVER_OPS_VERSION,
7    .bind = grove_lcd_bind,
8  };
9
10 ZIRCON_DRIVER_BEGIN(grove-lcd-cpp, grove_lcd_cpp_driver_ops, "grove-lcd-cpp", "0.1",
   ↪  3)
11   BI_ABORT_IF(NE, BIND_PLATFORM_DEV_VID, PDEV_VID_SEEED),
12   BI_ABORT_IF(NE, BIND_PLATFORM_DEV_PID, PDEV_PID_SEEED),
13   BI_MATCH_IF(EQ, BIND_PLATFORM_DEV_DID, PDEV_DID_SEEED_GROVE_LCD),
14 ZIRCON_DRIVER_END(grove-lcd-cpp)
```

Listing 11: Driver Binding for the LCD Driver C/C++

The binding definitions are surrounded by two macros, the `ZIRCON_DRIVER_BEGIN()` and `ZIRCON_DRIVER_END()`. Within this macros, the actual binding rules are defined. Zircon allows much more complex sequences to describe the device to be bound than Linux does. The corresponding listing 11 only shows two types of them. A full definition is available in `system/public/zircon/driver/binding.h`. The first used type is `BI_ABORT_IF()`. In combination with its first argument `NE`, the lines 11 and 12 instruct the device coordinator to abort the binding process if the third argument which must be from the same type specified as a second argument, does not match the one defined for a device (see 10, lines 29 to 30 and 38 to 39). This type of rule is used within all drivers written as a part of this thesis to ensure the vendor and product ID matches the ones defined for the Grove device definitions. All of them share exactly these symbolic names shown in this listing as vendor and product IDs. Only the product ID, and thus, the last binding rule shown in line 13 is different for the particular device types, the LCD, the RGB and the combined PDev device. The corresponding binding rule to differentiate between them is `BI_MATCH_IF()` with `EQ` as a first argument. As a result, the whole binding sequence means a device matching to the shown rule must define `PDEV_VID_SEEED` as a vendor ID, `PDEV_PID_SEEED` as a product ID and `PDEV_DID_SEEED_GROVE_LCD` as a product ID. Accordingly, the shown rule is for a Grove LCD driver, no matter if it is written in C or C++. To change it for the RGB or PDev one, only the symbolic name in listing 11, line 13 and related naming definitions must be changed. In common, these are the names used as arguments for `ZIRCON_DRIVER_BEGIN/END()`. But these macros do not only contain the driver's name as an argument, the `ZIRCON_DRIVER_BEGIN/END()` takes a reference to a driver operations structure as well. The structure (see listing 11, line 5 to 8) is not visible as such one at first sight because Zircon allows type definitions in contrast to Linux. Within this `zx_driver_ops_t`, only two entries must be defined. The version code must be set to `DRIVER_OPS_VERSION`. It is a pre-defined symbolic name and a

requirement for the driver. The second entry in line 7 is a function pointer to the driver's *bind* implementation, i.a. the corresponding function to Linux' `probe()`. If the binding sequence matches a device, this function is the one to be called. Similar to Linux, its task is setting up the driver and the device for use.

As already mentioned, the second line, the extern definition of the `bind()` function signature, is only needed for C++ drivers as the binding routine and the actual implementation of `bind()` are done in different source files and programming languages. In a pure C driver, it is not needed at all.

One further special situation about these binding rules is the fact they are defined as a part of the driver but located in a defined binary segment which allows the device coordinator to access them without loading a whole driver into its address space. The full driver is not loaded until a matching device is found and its `bind()` function referenced in `zx_driver_ops_t` is actually called.

3.6.4 C-Driver

Driver Binding and Release

After a match between driver and device is found by the device coordinator, the complete driver is loaded into the device host process of a superior device. At this point, the driver's execution starts by calling its `bind()` function with this parent device as an argument.

Zircon driver code seems unfamiliar at first sight if switching from Linux, e.g. because of naming, function signatures and coding style. But the structure within is similar. As a first step, memory for instance specific driver data is allocated. The previous section mentioned already that the coding style used in Zircon prefers type definitions over structures for C drivers. Thus, the type `grove_t` is introduced for this situation. The data stored within is comparable to the one stored in the Linux equivalent. It contains, besides other entries that are considered as soon as they are needed, ones for the I^2C objects but also entries to store the device's current state. Unlike Linux, the actual memory allocation for this type is done via `calloc()`, a standard C library function (see listing 12, line 4 to 7). Obviously, error handling in form of checking if the pointer to the resulting memory regions is valid, is needed in Zircon as well.

The next call is kind of platform device specific. It fetches the protocol device for this driver, in this case for the `PDev` protocol (see listing 12, line 9 to 13). This will turn this driver implementation into a platform device specific one. A driver which is not associated to platform devices would fetch the `ZX_PROTOCOL_I2C` instead, in this situation. The result of this call is a `pdev_protocol_t` typed object which represents the platform device instance. It is needed to fetch associated devices on the platform bus. For this driver, these are both I^2C devices of the Grove-LCD RGB backlight. They are allocated using the call `pdev_get_protocol()` shown in listing 12, line 16 to 25. The call is very similar to the previously used one to fetch the platform device. It uses the fetched platform device instance as a parent instead of the one given as a call argument to the bind function.

```c
static zx_status_t grove_bind(void* ctx, zx_device_t* parent) {
    grove_t* grove = calloc(1, sizeof(*grove));
    if (!grove) {
        return ZX_ERR_NO_MEMORY;
    }

    if (device_get_protocol(parent, ZX_PROTOCOL_PDEV, &grove->pdev) != ZX_OK) {
        free(grove);
        zxlogf(ERROR, "Failed to fetch the PDEV protocol for the grove lcd rgb
        ↪  driver - not supported\n");
        return ZX_ERR_NOT_SUPPORTED;
    }

    size_t actual;
    if (pdev_get_protocol(&grove->pdev, ZX_PROTOCOL_I2C, 0, &grove->i2c_rgb,
    ↪  sizeof(grove->i2c_rgb), &actual) != ZX_OK) {
        free(grove);
        zxlogf(ERROR, "Failed to fetch the I2C protocol (rgb) for the grove lcd rgb
        ↪  driver - not supported\n");
        return ZX_ERR_NOT_SUPPORTED;
    }
    if (pdev_get_protocol(&grove->pdev, ZX_PROTOCOL_I2C, 1, &grove->i2c_lcd,
    ↪  sizeof(grove->i2c_lcd), &actual) != ZX_OK) {
        free(grove);
        zxlogf(ERROR, "Failed to fetch the I2C protocol (lcd) for the grove lcd rgb
        ↪  driver - not supported\n");
        return ZX_ERR_NOT_SUPPORTED;
    }

    device_add_args_t args = {
        .version = DEVICE_ADD_ARGS_VERSION,
        .name = "grove-pdev-drv",
        .ctx = grove,
        .ops = &grove_device_protocol,
        .flags = DEVICE_ADD_INVISIBLE,
    };

    if ((status = device_add(parent, &args, &grove->device)) != ZX_OK) {
        free(grove);
        return status;
    }

    thrd_t thrd;
    int thrd_ret = thrd_create_with_name(&thrd, grove_init_thread, grove,
    ↪  "grove_init_thread");
    if (thrd_ret != thrd_success) {
        status = thrd_ret;
        device_remove(grove->device);
        free(grove);
    }

    return status;
}
```

Listing 12: Implementation of the `bind()` Function within the Zircon Platform Device Driver in C

The fetched protocol is, of course, `ZX_PROTOCOL_I2C` and the resulting object is stored as an `i2c_protocol_t` within the `grove_t` structure. But to fetch the desired device from both possible ones defined in listing 10, an additional call argument is needed. It is a number to describe the position within the `pbus_i2c_channel_t` array (see listing 10, line 15 to 24). Unlike Linux's `probe()`, Zircon's `bind()` does not need so much kernel internal driver setup and registration in various infrastructure elements. The only similar action a Zircon driver must perform is adding a device below the parent via the `device_add()` call in listing 12, line 35. This call adds this device instance to the system using the arguments defined in line 27 to 33. For this reason, the structure contains a pointer to the driver operations (line 31), i.a. the FIDL operations on this platform device driver. The issue known from Linux, making the private driver instance data available through the whole driver including the file operations and other interfacing options, is solved more sophisticated in Zircon. Already this arguments structure contains it as a so-called *context pointer* `ctx`. It is a void pointer and thus, capable of its own type definitions.

Beside these entries, the structure contains a version code, the driver name to be shown and maybe additional flags. These flags influence, among other things, the way a device is added to the system. The implementation shown uses `DEVICE_ADD_INVISIBLE` as a flag. It should be used for device drivers with long running initialization. The idea behind is to add a device representation already to the system and leave the `bind()` function but hide it from users until the initialization is done. One or more of such long running operations are usually done as in parallel running threads, if possible. If the initialization finishes successfully, the driver implementation must switch the device state to visible while a failed one will not be spotted by users at all.

The device initialization for both Grove parts are not that long and it would probably be fine to initialize them using a sequential function. However, Zircon does not define precise rules for this situation. For this reason, the actual device initialization is implemented in the thread version in C and the sequential one in C++. The code shown in listing 12, line 40 to 46 uses a single thread for both initializations which is considered in detail within the following section. As the thread creation takes the driver context as an argument as well, it enables the access to the needed I^2C instances without further ado. The used Zircon driver implementation returns a status code which is needed for a proper error handling. However, if the device initialization fails, only the situation occurs in which a call must be revoked within the whole binding function. The already added device must be removed and, as always, the `grove_t` instance must be freed (see listing 12, line 42 to 46).

As shown, Zircon does not need a sophisticated driver setup as known from Linux and as a result, not that much memory allocations in `bind()`. At the same time, it is not needed to undo most calls done during this function. As a result, the error handling is, in this case, less complex than in Linux. Using jump marks for clean up is not needed, thus, but valid in Zircon as well.

```
1  static int grove_init_thread(void* arg) {
2      grove_t* grove = arg;
3
4      mtx_lock(&grove->lock);
5      /* RGB initialization is skipped */
6
7      i2c_cmd_t setup_cmds[] = {
8          {LCD_CMD, 0x01},
9          {LCD_CMD, 0x02},
10         {LCD_CMD, 0x0c},
11         {LCD_CMD, 0x28},
12     };
13
14     for (int i = 0; i < (int)ARRAY_SIZE(cmds); i++) {
15         status = i2c_write_sync(&grove->i2c_lcd, &setup_cmds[i].cmd,
           ↪  sizeof(setup_cmds[0]));
16         if (status != ZX_OK) {
17             zxlogf(ERROR, "grove-lcd: write to i2c device failed\n");
18             goto init_failed;
19         }
20     }
21
22     status = i2c_write_sync(&grove->i2c_lcd, "@Initialized", 12);
23     if (status != ZX_OK) {
24         zxlogf(ERROR, "grove-lcd: write to i2c device failed\n");
25         goto init_failed;
26     }
27
28     mtx_unlock(&grove->lock);
29     device_make_visible(grove->device);
30     return ZX_OK;
31
32 init_failed:
33     zxlogf(ERROR, "grove init thread failed\n");
34     mtx_unlock(&grove->lock);
35     return ZX_ERR_IO;
36 }
```

Listing 13: Device Initialization in a Zircon Platform Device Driver (C)

Accordingly, the `release()` implementation is very simple and short as well and thus, not shown here. The device should be removed and the context data must be freed. However, it is rare this function is called according to the driver lifecycle discussed in section 3.2. The mentioned actual device initialization is very similar to the way it is done in Linux. Thus, this section focuses on the Zircon specific parts and considers only the LCD initialization. The RGB one is part of this thread function as well, but skipped in this context. Data transfers on the I^2C bus are also critical sections in Zircon, so it is needed to lock them just like in Linux (see listing 13, line 4). It is managed as a part of the `grove_t` instance.

Overall, in Zircon the use of global variables is avoided, while static defines, e.g. for fixed values are allowed. Everything else should be a part of a device instance context. After the section is locked, the way I²C transactions are done does not differ from Linux except the available kernel API calls. It is because the Linux implementation was inspired by Zircon. Defining a structure respectively a data type for the register and corresponding value tuples, building an array containing the needed sequence (see listing 13, lines 7 to 12) and sending it using a loop over this array (lines 14 to 20) was discovered as a commonly used pattern in Zircon.

It is better readable but as well more flexible than writing the commands hard coded into the I²C APIs and reduces redundant code at the same time. Using this pattern, an additional step in the initialization sequence requires only an extra entry in the `i2c_cmd_t` array but no change in the write logic. The used loop determines the array's size and does the action, the `i2c_write_sync()` for all tuples within. Thereby the I²C write call is similar to the one invoked on Linux. Indeed, there is no API to write a single byte value into a controller register, but the call shown in listing 13, line 15 takes the desired I²C representation as a first argument as well. The following argument differs. It is the address to an element in the array, which represents the target register on the controller. From this point, the call works like the one used to send strings in Linux. The last argument is not a single value to be written in the named register but a number of bytes to be transferred. As already known from Linux is the first one interpreted as a target register while all following ones are written in it. To leave the loop correctly in case of an error this function implementation takes advantage of jumps. The error routine must unlock the mutex and exit the thread with a certain error code which allows a proper tear down of the failed driver in the calling function (see listing 13, lines 18 and 32 to 35).

If the initialization of both partial devices was successful so far, an example text is set to the LCD. It shows to a user that the display is initialized and ready for use. With the available Zircon API for writing I²C transfers, the idea for texts stays the same as already pictured for Linux. The target register is encoded as an ASCII character and prefixes the actual text to be shown. As discussed previously the terminating, non-printable character of a string must be cutted off at sending to prevent display errors.

However, the probably most important task of this function is, after the initialization was finished successfully, to change the device visibility in Zircon's device filesystem. This is done using the call `device_make_visible()`, shown in listing 13, line 29. From this moment on, its representation and thus the actual Grove device, is accessible by users via the interfacing options defined via the `.ops` field in the `device_add_args_t` (see listing 12, line 31).

Driver Interfaces

In accordance with the concept drawn up previously, the Zircon platform device driver receives only a FIDL interface. The common file operations `read()` and `write()` are considered in the following section for the two-driver version.

```
1  library zircon.display.grove.pdev;
2
3  [Layout="Simple"]
4
5  interface Pdev {
6      1:SetColor(uint8 red, uint8 green, uint8 blue);
7      2:GetColor() -> (uint8 red, uint8 green, uint8 blue);
8      3:ClearLcd();
9      4:WriteFirstLine(uint8 position, string:32 line);
10     5:WriteSecondLine(uint8 position, string:32 line);
11     6:ReadLcd() -> (string:32 content);
12     7:GetLineSize() -> (uint8 linesize);
13 };
```

Listing 14: FIDL Definitions for a Zircon Platform Device Driver (C)

All FIDL definitions for the Grove-LCD RGB backlight driver variants are defined within the directory `system/fidl/zircon-display-grove/` in the Zircon sources. It contains three single definitions, one for the platform device variant and two of them for the single controllers. Thereby is the first variant split into the controller specific parts for the LCD and RGB definitions. The defined function signatures do not change between both variants. Besides the FIDL files, the directory contains a `rules.mk` file as well. It bundles the make rules for all of them in a single file. To reach distinct outputs, the built modules get a name extension. Usually, they are named according to the current directory but to build all FIDL files in a single directory with only one `rules.mk`, the module names are extended with e.g. `.pdev` for the platform device definition or `.rgb` for the RGB specific one. The compiled results are available in the build directory under the path `build-arm64/system/fidl/zircon-display-grove.pdev` for the platform device. The other ones are named according to their extensions. Important for the further driver development is the generated header file with the function signatures for driver and user. It must be included in the source file and the whole directory must be referenced in the driver's `rules.mk` by adding the additional line `MODULE_FIDL_LIBS := system/fidl/zircon-display-grove.pdev` in order to be linked correctly. Within the source file, it is sufficient using `zircon/display/grove/pdev/c/fidl.h` as an include path. It is the path to the header definition starting in the mentioned directory's `gen/include/zircon` record.

The actual interface definitions are shown in listing 14. It's C-like syntax is simple to use, but with some pitfalls. The definition for interface number 3 in line 8 is very simple. It is a void call and thus without call arguments. More complex ones as number 1 in line 6 take one or more primitive data types as arguments. Return values, e.g. as defined for interface number 2 in line 7 are defined using a `-> (<type>)` after the actual call interface. The return value might consist of more than one value. An anomaly is the way strings are defined as in or out parameters in the FIDL syntax.

```
1  static zx_protocol_device_t grove_device_protocol = {
2      .version = DEVICE_OPS_VERSION,
3      .release = grove_release,
4      .message = grove_fidl_message,
5  };
6
7  static zx_status_t grove_fidl_message(void* ctx, fidl_msg_t* msg, fidl_txn_t* txn) {
8      zx_status_t status = zircon_display_grove_pdev_Pdev_dispatch(ctx, txn, msg,
       ↪  &fidl_ops);
9      return status;
10 }
11
12 static zircon_display_grove_pdev_Pdev_ops_t fidl_ops = {
13     .SetColor = grove_fidl_set_color,
14     .GetColor = grove_fidl_get_color,
15     .ClearLcd = grove_fidl_clear_lcd,
16     .WriteFirstLine = grove_fidl_write_first_line,
17     .WriteSecondLine = grove_fidl_write_second_line,
18     .ReadLcd = grove_fidl_read_lcd,
19     .GetLineSize = grove_fidl_get_line_size,
20 };
```

Listing 15: Driver Interfaces via FIDL in a Zircon Platform Device Driver (C)

Using the *simple* layout needed for device drivers, a string as datatype definition must be followed by its maximum transfer size (see listing 14, lines 9 to 11). However, the well-defined input and output arguments lead to some advantage over Linux's I/O controls. The generated code allows for example strict type safety and according checks. At the same time the use of an abstract interface definition language enables the generation of bindings in different target languages. For the Zircon version used during this work, only idiomatic C bindings are available for drivers.

Within the driver, the FIDL calls are declared via a `.message` function pointer in the already mentioned `zx_protocol_device_t` structure. Beside this message entry, it contains as well a version string and the function pointer to the previously discussed driver release implementation (see listing 15, lines 1 to 5). The `message()` implementation always follows the same scheme. It calls a dispatcher function which is generated as a part of the FIDL definitions and forwards the message call arguments to this function besides a reference to `zircon_display_grove_pdev_Pdev_ops_t` (see listing 15, lines 7 to 10). This datatype consists of a structure of function pointers to the FIDL definitions for the included one and is generated in this context as well. All generated helpers and function signatures are available in the previously mentioned and included header file. The `zircon_display_grove_pdev_Pdev_ops_t` structure holds the function pointers to the actual implementations of the defined interfaces within this driver. A FIDL call reaches the driver as a marshaled message which is dispatched into the actual call via the provided `dispatch()` call and the list of actual implementations in the driver in `zircon_display_grove_pdev_Pdev_ops_t`.

```c
static zx_status_t grove_fidl_set_color(void* ctx, uint8_t red, uint8_t green,
    uint8_t blue) {
    grove_t* grove = ctx;
    mtx_lock(&grove->lock);

    i2c_cmd_t cmds[] = {
        {RED, red},
        {GREEN, green},
        {BLUE, blue},
    };

    for (int i = 0; i < (int)ARRAY_SIZE(cmds); i++) {
        zx_status_t status = i2c_write_sync(&grove->i2c_rgb, &cmds[i].cmd,
            sizeof(cmds[0]));
        if (status != ZX_OK) {
            zxlogf(ERROR, "grove: write to i2c device failed\n");
            mtx_unlock(&grove->lock);
            return status;
        }
    }
    grove->color.red = red;
    grove->color.green = green;
    grove->color.blue = blue;

    mtx_unlock(&grove->lock);
    return status;
}

static zx_status_t grove_fidl_get_color(void* ctx, fidl_txn_t* txn) {
    grove_t* grove = ctx;
    zx_status_t status = zircon_display_grove_pdev_PdevGetColor_reply(txn,
        grove->color.red, grove->color.green, grove->color.blue);
    return status;
}
```

Listing 16: Illustrative Implementation of two FIDL Calls in a Zircon Platform Device Driver (C)

The actual implementations of the FIDL calls are similar to the Linux ones. As there, the set and get function for the RGB controller are considered in the Zircon specific listing 16. Besides the changing API, the way data is transferred between user and driver is the main difference and characterized by FIDL and the fact Zircon drivers are located in userspace. Call arguments are applicable without further ado and each function signature is enhanced with a context pointer by the FIDL compiler. As a result, it is not necessary to use `open()` and `close()` to access the device instance data as it is done in Linux. Both functions are not needed at all, if no specific needs must be matched. Thus, the actual `grove_fidl_set_color()` implementation follows the same sequence as the Linux one discussed before. Of course, the code shown in listing 16, lines 1 to 27, is idiomatic for Zircon and uses these APIs which were already considered during the `bind()` implementation.

The `grove_fidl_get_color()` implementation is more interesting. It's initial FIDL definition had no input but three output arguments which must be transported back to the calling user process which invokes the IPC communication. The generated function signature contains the familiar context pointer but as well an argument of the type `fidl_txn_t`. Furthermore, the FIDL compiler generated a `reply()` call specific to this interface definition (see listing 16, line 31). It must be called with matching arguments but the underlying IPC mechanism is abstracted. Thus, especially this implementation is very simple. The `fidl_txn_t` pointer `txn` is forwarded to the reply call while the RGB component values are taken from the currently saved instance state, the context.

Further FIDL call implementations are neither discussed nor shown as part of this work. The full source code for this driver variant is available in the corresponding GitHub repository at `https://github.com/Allegra42/zircon/tree/i2c-grove-lcd/system/dev/display/grove-c-pdev-driver`.

3.6.5 Further Driver Variants

Two Driver Variant

The first difference between the C driver variants is in the bind implementation of the drivers. Section 3.6.4 already mentioned the difference between the I²C device allocation in both C driver variants, the platform device and the common one. The platform device variant fetches the `ZX_PROTOCOL_PDEV` using the parent device given as a call argument of `bind()` (see listing 17). Following, it uses the resulting `pdev_protocol_t` instance with a platform device specific API to allocate the I²C devices (see listing 17). Instead, the common version, here shown for the RGB C driver, allocates the I²C device directly by fetching a `ZX_PROTOCOL_I2C` (see listing 18). For the actual use of the fetched I²C instances the underlying mechanism does not matter at all. However, a single driver implementation for a composed device is currently only available on platform devices using the related API.

```
1   if (device_get_protocol(parent, ZX_PROTOCOL_PDEV, &grove->pdev) != ZX_OK) {
2     /* error handling */
3   }
4
5   if (pdev_get_protocol(&grove->pdev, ZX_PROTOCOL_I2C, 0, &grove->i2c_rgb,
    ↪  sizeof(grove->i2c_rgb), &actual) != ZX_OK) {
6     /* error handling */
7   }
8   if (pdev_get_protocol(&grove->pdev, ZX_PROTOCOL_I2C, 1, &grove->i2c_lcd,
    ↪  sizeof(grove->i2c_lcd), &actual) != ZX_OK) {
9     /* error handling */
10  }
```

Listing 17: Allocation of an I²C device in a Zircon Platform Device Driver (C)

```
1   if (device_get_protocol(parent, ZX_PROTOCOL_I2C, &grove_rgb->i2c) != ZX_OK) {
2     free(grove_rgb);
3     zxlogf(ERROR, "Failed to fetch the I2C protocol for the grove lcd rgb driver -
      ↪  not supported\n");
4     return ZX_ERR_NOT_SUPPORTED;
5   }
6 }
```

Listing 18: Allocation of an I²C device in a common Zircon Device Driver (C, RGB)

The implementation of the FIDL calls is the same, no matter which driver type is used. As a single difference, there is a division of the total calls to the corresponding controller specific driver parts. But in contrast to a platform device driver, it is more or less meaningful to implement the `read()` and `write()` systemcalls for those drivers. In general, they are done similar to the unattended Linux implementation, except already known Zircon specifics and details. The discussion about these calls is about the RGB specific driver within this work. It does not only illustrate the calls itself but also the issue with `write()` for such a device and why FIDL or respectively `ioctl()` on Linux, is more suitable in some situations.

The `read()` implementation shown in listing 19 is absolutely fine and valid for the RGB device except the representation of its results. It is a composite value made from a red, green and blue ratio. Of course, the driver is free to define the output format but it is not that intuitive and comparable compared to reading e.g. files. Apart from this problem, the implementation is rather simple and follows the standard. A `read()` call in Zircon has, as discussed before, a context pointer as an argument. Thus, an additional procedure to make the instance data available as in Linux is not necessary (see listing 19, lines 1 and 3).

```
1  static zx_status_t grove_rgb_read(void* ctx, void* buf, size_t count, zx_off_t off,
   ↪  size_t* actual) {
2    if (off == 0) {
3      grove_rgb_t* grove_rgb = ctx;
4      *actual = snprintf(buf, count, "Grove LCD RGB Status:\nRed: %x\nGreen: %x\nBlue:
       ↪  %x\n",
5          grove_rgb->color.red, grove_rgb->color.green, grove_rgb->color.blue);
6    } else {
7      *actual = 0;
8    }
9    return ZX_OK;
10 }
```

Listing 19: Implementation of the `read()` call in a Zircon Device Driver (C)

Without different execution modes between driver and user, there is no need for checking the buffer and its permissions with sophisticated helpers. The data to be read is stored using the `snprintf()` function in line 4 with a defined format. As a matter of fact, the resulting string should be prepared with this call. However, the maximum size to copy is specified in the argument `count` and the user buffer is accessible without further ado, thus the call `snprintf()` can execute the transfer to the user without any additional steps (see listing 19, line 4). The original logic behind is again similar to Linux except the names, but both specify an offset at which should be read. An example for such an implementation is given in [23]. Another call argument is the read offset `off`. It specifies the start address to read from. For a character device like the considered RGB backlight device is such an offset not meaningful at all and thus, ignored. If an offset is specified, no data at all is copied as a result, the value of `actual` is set to *EOF (End Of File)*. Is the offset zero, then as much data as possible, specified in the already mentioned `count` value, is transferred.

The call argument `actual` is in fact an output argument which must be set with the number of actually transferred bytes. For this reason, the return value of `snprintf()` in line 4 is stored within. Independent from this value, the `read()` function must always return `ZX_OK` (see listing 19, line 9).

Even for the `write()` function shown in listing 20, the buffer given as a call argument could be used directly, without any validation checking. However, a buffer with RGB values can not been written like a text. The color ratios must be parsed from the given buffer and sent to specific and distinct registers. Thus, the input text needs a robustly defined format to be parsed in this situation. Implementing the `write()` call for RGB is rather a demonstration of the way it is realized and the issues with writing such data than a valid use case. FIDL calls are considerably better suited for this situation. Because the input buffer must be parsed, it makes sense to copy it to a temporary one where the parsing is done. As the temporary buffer can take the size of the input buffer specified by the `count` value, the whole content can be copied at once (see listing 20, lines 4 and 6).

```c
static zx_status_t grove_rgb_write(void* ctx, const void* buf, size_t count,
    zx_off_t off, size_t* actual) {
  grove_rgb_t* grove_rgb = ctx;
  char delim[] = " ";
  char tmp[count + 1];

  snprintf(tmp, count, "%s\n", (char*)buf);
  *actual = count;

  mtx_lock(&grove_rgb->lock);

  char* ptr = strtok(tmp, delim);
  while (ptr != NULL) {
    if (i == 0 && ptr[0] == 'r') {
      red = atoi(++ptr);
    } else if (i == 1 && ptr[0] == 'g') {
      green = atoi(++ptr);
    } else if (i == 2 && ptr[0] == 'b') {
      blue = atoi(++ptr);
    } else {
      /* handle wrong input format */
    }
    ptr = strtok(NULL, delim);
    i++;
  }

  i2c_cmd_t cmds[] = {
    {RED, red},
    {GREEN, green},
    {BLUE, blue},
  };

  for (int i = 0; i < (int)ARRAY_SIZE(cmds); i++) {
    status = i2c_write_sync(&grove_rgb->i2c, &cmds[i].cmd, sizeof(cmds[0]));
    if (status != ZX_OK) {
            zxlogf(ERROR, "grove-rgb: write failed\n");
            goto fail;
    }
  }
  grove_rgb->color.red = red;
  grove_rgb->color.green = green;
  grove_rgb->color.blue = blue;

fail:
    mtx_unlock(&grove_rgb->lock);
    return status;
}
```

Listing 20: Implementation of the `write()` call in a Zircon Device Driver (C)

Similar to the `read()` function, the number of copied bytes is stored in `*actual` as an output argument. Why string parsing is not desired in a driver is pictured in the lines 11 to 24. It is complex, error-prone and hard to maintain, even in Zircon which enables the use of the standard C library. Furthermore, the user must know about the specific syntax. In the shown and shortened function, the information about this syntax is skipped. It is only printed to the user in the case a wrong format was sent.

The actual I²C transfer is done similar to the previously considered ones. A structure is used to store the tuples of register addresses and its matching color ratio parsed from the input buffer and a loop steps over these tuples to send them via `i2c_write_sync()` (see listing 20, lines 26 to 38). Even if the Grove RGB backlight is modified using `write()`, the internal color state must be updated after a successful update since it is not possible to read it directly from the device.

C++ Driver Variants

The C++ driver variants are considered at once with a focus on the platform device driver. Nevertheless, important differences in the two-driver variant are mentioned. As a first and fundamental difference are Zircon C++ drivers truly object-oriented. A Driver is represented by a common class with contructor, destructor as well as public and private member variables and functions. Thus, the driver is organized differently from a pseudo object-oriented C driver. A per device context data structure is not needed anymore because a driver object itself is instantiated per device and the member variables, especially the private ones, adopt the task of it. Further, the widely used function pointers are not idiomatic in C++ and thus avoided when possible. Instead, especially the driver operations like `read()`, `write()` or the `message()` function which dispatches FIDL calls, are integrated as C++ **mixin**s. Unfortunately, neither C++ drivers nor the way mixins are instrumented is well documented in this context. Probably the best information about it is written as a comment in the header file `system/ulib/ddktl/include/ddktl/device.h`. It contains a list about the available mixins as well. Because the platform device driver must support FIDL defined function calls, the corresponding mixin `ddk::Messageable` must be invoked. For the two-driver variant it is additionally `ddk::Readable` and `ddk::Writeable`. Listing 21 shows the way the `ddk::Messageable` mixin is included into the Grove platform device C++ driver in line 2. Line 3 shows the situation for the two-driver variant. The mixin functions must be declared in the public section with a defined name and signature as shown in the lines 14 and 15 for the `DdkRelease()` function which is obligatory for each driver and for `DdkMessage()`.

Very different in contrast to a C driver is the way the `pdev` instance is allocated in a C++ driver. It is represented as a private member variable within the `GroveDevice` class. The platform device itself is represented as a `PDev` name class from the namespace `ddk` (see listing 21, line 22) while its corresponding `pdev` object is initialized using the *initializer list* of the `GroveDevice`. It calls the constructor of `PDev` with `parent` as an argument while the driver class itself is constructed (see listing 21, lines 7 and 8). The same mechanism is used to instantiate the I²C representation `I2cChannel` in the ordinary variant with two drivers.

```
1  class GroveDevice;
2  using DeviceType = ddk::Device<GroveDevice, ddk::Messageable>;
3  /* using DeviceType = ddk::Device<GroveDevice, ddk::Messageable, ddk::Readable,
   ↪  ddk::Writeable>; */
4
5  class GroveDevice : public DeviceType {
6  public:
7      GroveDevice(zx_device_t* parent)
8          : DeviceType(parent), pdev(parent) {}
9      ~GroveDevice() {}
10
11     static zx_status_t Bind(zx_device_t* parent);
12
13     /* Mixin methods */
14     void DdkRelease();
15     zx_status_t DdkMessage(fidl_msg_t* msg, fidl_txn_t* txn);
16
17     /* Definitions for FIDL (shortened) */
18     static zx_status_t SetColor(void* ctx, uint8_t red, uint8_t green, uint8_t
       ↪  blue);
19     static zx_status_t GetColor(void* ctx, fidl_txn_t* txn);
20
21 private:
22     ddk::PDev pdev;
23     std::optional<ddk::I2cChannel> i2c_rgb;
24     std::optional<ddk::I2cChannel> i2c_lcd;
25     fbl::Mutex i2c_lock;
26     /* Variables holding the device state */
27
28     struct I2cCmd {
29         uint8_t cmd;
30         uint8_t val;
31     };
32
33     /* Definitions for internally used functions, e.g. init functions */
34 };
```

Listing 21: Header Definition for a C++ Platform Driver in Zircon

For the pdev one, the I^2C devices are instantiated during the common device initialization written in the C++ source file. The last anomaly which is worth mentioning is the `fbl` namespace which contains e.g. the Zircon C++ mutex implementation shown in listing 21, line 25. For C++ drivers, Zircon provides specific libraries in contrast to its C drivers and bundles them in exactly this `fbl` namespace. Besides the mutex implementation, it contains for example a specific secured string class including helper functions for the use in C++ drivers as well as other re-implementations of the C++ standard library.

```
1  zx_status_t GroveDevice::Bind(zx_device_t* parent) {
2    fbl::AllocChecker ac;
3
4    auto dev = fbl::make_unique_checked<grove::GroveDevice>(&ac, parent);
5    if (!ac.check()) {
6      return ZX_ERR_NO_MEMORY;
7    }
8
9    auto status = dev->DdkAdd("grove-pdev-drv", DEVICE_ADD_INVISIBLE);
10   status = dev->PdevInit();
11
12   __UNUSED auto ptr = dev.release();
13
14   return status;
15 }
```

Listing 22: Implementation of `Bind()` in a Zircon Device Driver (C++)

The `Bind()` function shown in listing 22 looks unfamiliar when moving from C driver development as well, but at a closer look, it does not differ except from the language and its effects. It is just the counterpart of a C driver in idiomatic C++. As such one a manual memory allocation for data structures is not longer needed. It is done implicitly as part of the driver object creation (see listing 22, line 4). But calling the driver's constructor does not only build a driver respectively a device instance. Looking back to the corresponding header file it does invoke the mixins, in this case `ddk::Messageable` for FIDL, as well as the constructor for the `PDev` instance. Thus, constructing the driver object does a bulk of a C driver in a single call plus the according header definitions. This results in less code but is significantly harder to read and to understand if not familiar with C++. To reach the same functional range as the C driver one needs to allocate the I^2C channel objects, add the device with meaningful flags and initialize the actual Grove device. Similar to the C driver, the device is added *invisible* as long as the initialization is not done, yet (see listing 22, line 9).

The C++ version of this call, the `DdkAdd()` requires only a name and flags as arguments. It is neither needed to provide a version code nor a context or driver operations. The context is not needed because the instance specific data is stored as a part of the driver object in C++ while the operations are invoked via the *mixins* specified in the header file. However, the actual device initialization which is invoked in line 10 of this listing, is not done using a thread as known from the C driver. But as the initialization of this device does not require a long running threaded setup, it is only a variation without further meaning for driver development.

For a better modularization the initialization of the actual devices is split into three functions: the `PdevInit()` pictured in listing 23, the `LcdInit()` in listing 24 and the not shown `RgbInit()`. The first one, `PdevInit()` instantiates the I^2C devices and invokes their actual initialization routines sequentially. The platform device object was already created during the driver class constructor.

```
1  zx_status_t GroveDevice::PdevInit() {
2    if (!pdev.is_valid()) {
3      goto error;
4    }
5
6    i2c_rgb = pdev.GetI2c(0);
7    if (!i2c_rgb) {
8      goto error;
9    }
10   i2c_lcd = pdev.GetI2c(1);
11   if (!i2c_lcd) {
12     goto error;
13   }
14
15   if ((status = RgbInit()) != ZX_OK) {
16     goto error;
17   }
18   if ((status = LcdInit()) != ZX_OK) {
19     goto error;
20   }
21
22   DdkMakeVisible();
23   return status;
24
25 error:
26   return ZX_ERR_NO_RESOURCES;
27 }
```

Listing 23: Implementation of the Device Initializations in a Zircon Device Driver (C++, shortened)

Thus, this method must only check if it was created correctly (see listing 23, lines 2 to 4). As for the C driver, the `pdev` object is needed to create the I²C ones. The C++ API to do this is easier than in C. It is needed to call `GetI2c()` with the item location within the I²C definition array on the `pdev` (see listing 23, lines 6 and 10). If both objects are created without errors, the initialization routines are called sequentially. The device state is not switched to *visible* until both of them run successfully. If that is the case, the visibility can be switched using the call `DdkMakeVisible()` shown in line 22 and make the device accessible, e.g. via the FIDL calls.

The LCD initialization shown in listing 24 is already known from previous Zircon driver variants, but implemented using some C++ specific concepts. One is using the `fbl` implementation of C++ strings. In comparison to C strings this class provides a wide range of very comfortable helpers. As a result, the required string operations, e.g. to add the `@` in front of a non-static string, become easier. Another concept that is currently available for C++ only are *for-each loops* (see listing 24, line 12). Iterating over the LCD setup sequence becomes better readable and maintainable. However, an I²C write changes, too.

```
1  zx_status_t GroveDevice::LcdInit() {
2    fbl::String init("@Init");
3
4    I2cCmd cmds[] = {
5      {LCD_CMD, 0x01},
6      {LCD_CMD, 0x02},
7      {LCD_CMD, 0x0c},
8      {LCD_CMD, 0x28},
9    };
10
11   fbl::AutoLock lock(&i2c_lock);
12   for (const auto& i : cmds) {
13     status = i2c_lcd->WriteSync(&i.cmd, sizeof(cmds[0]));
14     if (status != ZX_OK) {
15       zxlogf(ERROR, "grove-pdev-cpp: i2c.WriteSync failed\n");
16       goto error;
17     }
18   }
19
20   status = i2c_lcd->WriteSync(reinterpret_cast<const uint8_t*>(init.c_str()),
     ↪  static_cast<uint8_t>(init.length()));
21   if (status != ZX_OK) {
22     goto error;
23   }
24   line_one = init;
25   return status;
26
27 error:
28   return ZX_ERR_IO;
29 }
```

Listing 24: Implementation of the LCD Initializations in a Zircon Device Driver (C++, shortened)

In C++, it is a member function of `I2cChannel` objects and thus, it must be called on such one (see listing 24, line 13). Accordingly, it is no longer needed to name the I²C instance as a call argument. Furthermore, the I²C write for a `fbl::String` is no longer that simple as before. C++ strings are represented as a class with internal character pointer holding the actual data. To match the `uint8_t*`, which is required in `WriteSync()`, this internal data must be casted. In C strings or character arrays are implicitly interpreted as the unsigned `uint8_t*`, while C++ requires sophisticated cast operations to cope with the situation (see listing 24, line 20). This results in a more complex and harder to read LCD write operations, but the idea behind stays the same.

```
1  zx_status_t GroveDevice::DdkMessage(fidl_msg_t* msg, fidl_txn_t* txn) {
2    return zircon_display_grove_pdev_Pdev_dispatch(this, txn, msg, &fidl_pdev_ops);
3  }
4
5  zircon_display_grove_pdev_Pdev_ops_t fidl_pdev_ops = {
6    .SetColor = GroveDevice::SetColor,
7    .GetColor = GroveDevice::GetColor,
8    .ClearLcd = GroveDevice::ClearLcd,
9    .WriteFirstLine = GroveDevice::WriteFirstLine,
10   .WriteSecondLine = GroveDevice::WriteSecondLine,
11   .ReadLcd = GroveDevice::ReadLcd,
12   .GetLineSize = GroveDevice::GetLineSize,
13 };
14
15 zx_status_t GroveDevice::ReadLcd(void* ctx, fidl_txn_t* txn) {
16   auto& self = *static_cast<GroveDevice*>(ctx);
17
18   fbl::String tmp = fbl::String::Concat({self.line_one, "\n", self.line_two});
19
20   return zircon_display_grove_pdev_PdevReadLcd_reply(txn, tmp.c_str(),
     ↪  static_cast<size_t>(tmp.length()));
21 }
```

Listing 25: FIDL in a C++ Zircon Device Driver

The `DdkMessage()` method is included as a mixin but the implementation is very much the same as in C, except the context pointer is replaced with `this`. Unfortunately, this means as well the generated `dispatch()` call is not specific to C++ (see listing 25, lines 1 to 3) because generating C++ bindings for drivers from FIDL definitions is not possible with the chosen Zircon version. Thus, the FIDL calls are not invoked like `DdkMessage()` itself, as *mixins*, but as a structure of function pointers (see listing 25, lines 5 to 13). Thereby, the actual implications of using C calls in a C++ driver become visible in the FIDL implementations, for example in the shown `ReadLcd()` (see listing 25, lines 15 to 21). The generated function signature delivers a context pointer as an argument but in C++, an object of the driver class `GroveDevice` is needed. There is no documented solution for this issue, but research in other drivers yielded to the pattern shown in line 16. It was commonly used in other Zircon drivers to cope with similar situations. The context passed to `dispatch()` is not a common C context pointer, but a `this` pointer of the current `GroveDevice` instance. Thus, it can be cast back to a `GroveDevice` object and used as such one during the FIDL call implementation (see listing 25, lines 2, 15 and 16).

The actual implementation of `ReadLcd()` is nothing special at all, but it pictures the simple string handling using the Zircon C++ `fbl::String` class (see listing 25, line 18). On the other side are the issues with those strings and C APIs illustrated as well. A conversion to the C string representation is needed for the generated C function `reply()` in line 20. The remaining FIDL call implications are done accordingly.

3.7 Performance Comparison

One of the most frequently mentioned disadvantages of a microkernel architecture in comparison to a monolithic kernel is performance loss and of course, it is an issue in Zircon as well. Even Travis Geiselbrecht, one of the main developers behind Zircon, noted that it is difficult to reach the same performance as traditional monolithic operating systems[16]. But since this fact is well-known and accepted for the resulting advantages like maintainability, security and modularity, the question about the actual cost remains. For this work, it is fascinating how the microkernel architecture affects driver development. Therefore, the assumption is made that especially the additional context switches into the actual microkernel running in kernel mode which are e.g. needed to perform hardware actions, influence the overall driver and system performance. The resulting idea is to measure the elapsing time during a corresponding call, for example toggling GPIOs. A more complete performance analysis of both kernels is hardly practicable as part of this work and thus, the corresponding results can of course not give a complete picture of the kernel performanc in every situation. But at least, they can convey an idea of the performance loss in Zircon in contrast to Linux.

To make the comparison as meaningful as possible, as many influencing parameters as possible are excluded. Using different development boards was fine for driver development, because it did not impact driver development as the factual topic of this work at all. For a comprehensive kernel performance test it is not suitable at all. Differing factors, for example the CPUs, its architectures, board wiring and various components would influence the results in an unpredictable way. For Zircon, the choice of development boards is pretty limited to the *HiKey960*, but the previous issues with this board are neglectable in this situation. Instead, it would rather distort results to consider external hardware and measure the performance there due to physical limitations and setup times. However, it is not reasonable to use exactly the same *HiKey960* for both measurements. Linux requires a completely different board configuration which can not be easily replaced. Whether this has an effect on the measurement at all, and if so how, can not be determined without further investigation.

The time elapsed must be measured internally for meaningful results without physical implications of the underlying hardware controller which is accessed by the test call. On the driver side is e.g. toggling a GPIO only a memory operation on a mapped register while the physical action performed by a corresponding controller takes a longer time. Both operating systems provide internal high resolution timers with a nanosecond resolution which can be used for the comparison, but particularly the quality of the Zircon timer is difficult to verify. Nevertheless, these timers are probably the best idea to measure the elapsed time during toggling GPIOs. Except for scheduling influences drivers are executed sequentially and without interruptions. Taking the elapsed nanoseconds since system start right before and after the considered systemcall and subtracting the start time from the end time should produce a comparable call duration.

As a result of these considerations the performance measurement is carried out under the following basic conditions:

- The operating systems to be compared run on a HiKey960 development board, each on a separated one.
- Both systems are in an idle state. Apart from a driver implementation performing the measurement and the basic operating system itself no further application is running.
- Both system specific driver implementations measure the required time to toggle a GPIO pin using the APIs:
  - `gpio_set_value()` on Linux and
  - `gpio_write()` on Zircon.
- The elapsed time is calculated on both systems using the calls
  - `getrawmonotonic()` on Linux and
  - `zx_clock_get_monotonic()` on Zircon.

  Both return the value of a monotonic counter running since system start with a nanosecond resolution. They are called twice, once before the GPIO toggle is done and once right after. Their delta is the elapsed time during the call. Possibly, it is influenced by scheduling, different scheduling strategies, the actual board and as well, the quality of the timer implementation.

The resulting values for Linux and Zircon are pictured as a *box-plot* in figure 3.9. This diagram type allows a comparable statistical representation of the elapsed time in nanoseconds. Each colored box has the size of the interquartile range (IQR) for the given values. It contains the average 50% of the datapoints limited by the upper and lower quartile. The bigger line within a box represents the median for the given range. The lines starting at the box are named *whiskers* and picture the range from the last values within the $1.5 * IQR$ below and above the box. Values within this range are also named as mild outlier values in contrast to the ones outside, pictured as dots.

Figure 3.9 shows a significant difference in the time needed to toggle a GPIO. The overall span of measured results but especially the box height in Linux is much smaller than in Zircon. All in all the Linux values represent a range between 520 and 7291 ns with a median on 1562 ns. For Zircon, the overall span starts higher, on 8854 ns which is clearly above the outlier values on Linux. At the same time the box height as well as the span represents a wider range with a median on 15625 ns and extreme outliers on up to 32291 ns. Considering the median values Zircon's `gpio_write()` is approximately 10 times slower than Linux's `gpio_set_value()`. However, the way this difference in performance is exactly influenced by various factors such as the used development board revisions, the scheduling strategies and the additional context changes in the microkernel approach, which is assumed to be a major factor, can not be deduced

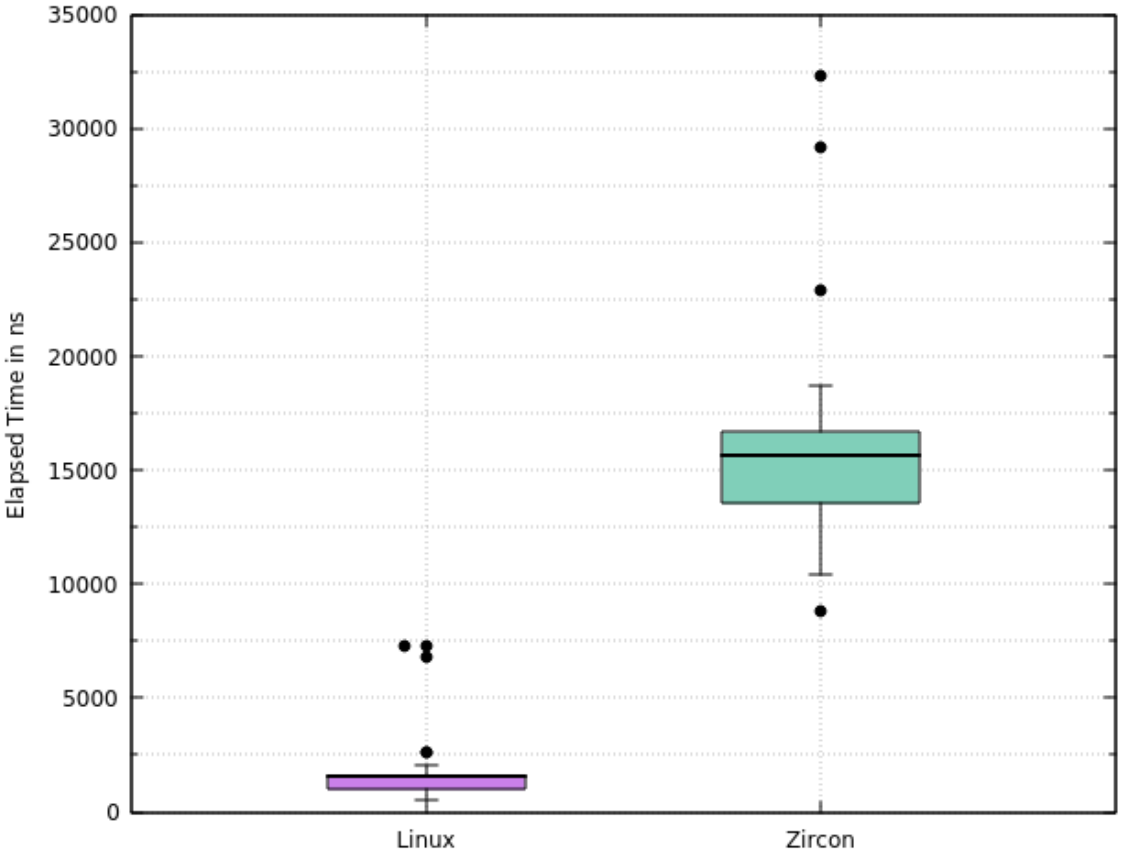

Figure 3.9: Performance Comparison between Linux and Zircon

from the measurement. Based on the results of this measurement, it can be assumed that the overall performance of both systems behaves similarly in comparison. But for reliable statements more in-depth and comprehensive tests are necessary.

Chapter 4

Conclusion

The present work and its evaluation can be divided into a theoretical and a practical part. While the former deals with general operating system concepts and their use in Zircon as a recently implemented microkernel approach in comparison to the older monolithic Linux kernel, the second part focuses more on their differences in driver development. The architectures and objectives of microkernel and monolith systems are fundamentally different, whereas the concepts used for basic operating system tasks are often similar. Overall, however, these are optimized for the use in their respective architecture. Especially Zircon presents an implementation idea that demonstrates a great deal of knowledge and experience in the field of operating system development and clearly shows that the developers have learned from aspects that are not necessarily optimally solved in Linux, e.g. for historical and/or compatibility reasons. Examples for these are, amongst other things, the handling of system calls or the memory management. Due to the lack of POSIX compatibility, Zircon enables a unencumbered design of systemcalls and adapts them to modern needs, but the enforcement of vDSOs and the use of `libzircon.so` for all systemcall implementations is interesting from a security aspect, as well. Linux is very architecture dependent regarding memory management. The multitude of architectures and their associated terms and concepts makes them difficult to understand and use correctly, even for experienced developers. In this context Zircon is, as far as it can be read from the documentation, much more stringent and as well, simpler. Without a division into memory areas with explicit meaning for different tasks, both, understanding and development becomes easier. On the other hand, there are also concepts used in Zircon, which, based on the experiences with Linux, can be assumed to fail for maintainability reasons with a possible spread of Zircon. An example is the way Zircon deals with board definition files which is discussed in section 3.6.

The circumstances are similar for the actual driver development as well. Conceptually, the influences of the microkernel become slightly visible in the lifecycle of a driver, for example when binding the driver to a device and for the selection of the superior device host process. Within the driver development itself, the differences to Linux are rather caused by the collective experiences and the selection of more modern approaches. Neither the break with the POSIX standard nor the introduction of

FIDL or the support of a wider range of programming languages in kernel and drivers are direct consequences of an operating system kernel architecture, but a reaction to learnings from Linux and other operating systems. Nevertheless, exactly these aspects make the development on Zircon truly pleasant.

Because of the well-written documentation, the transition to C drivers in Zircon is very smooth, only the documentation for the definition of devices and for board files as a whole are missing. The change to C++ drivers is more difficult, because there is hardly any documentation. By taking advantage from advanced C++ concepts, such as mixins these drivers become much more difficult to understand while the corresponding situation is obvious in a C implementation. Furthermore, C and C++ APIs are often mixed within already existing C++ drivers in the used version level. On the one hand this is motivated with missing C++ bindings for certain tasks in the used version level, as it is e.g. the situation for FIDL, on the other hand simple inconsistencies are probably an issue. But if the connections between C and C++ drivers are clear, C++ becomes very interesting and sufficient for driver development. In this context, it is also interesting to see how the development and language prioritization in Zircon will change when the support for *Rust* is ready for use. Considering the current Zircon development, the general trend is to replace C drivers with C++ ones. Rust could be even more interesting from a security point of view. As a programming language it is not as mature and wide-spread as C and C++. Overall, the development workflow for Zircon appears well thought-out and coherent, no matter which programming language or compiler is used. However, most of the accompanying tools in Zircon are based on LLVM and therefore, require a Clang build. Linux slowly allows the use of Clang and LLVM as well, justified with the advantages of the same tools as known from Zircon. But none of these toolchains had a noticeable influence on the development which is why a more detailed compiler analysis was skipped.

The microkernel was able to play off its strengths mainly in its actual use. While a bug in Linux might paralyze the whole system, Zircon, as a microkernel, can react much better. Unfortunately, the exact behavior in such situations is not yet documented. But during the development, two situations have caught the attention. First, it is the behavior of the driver respectively of the device coordinator in case of failed initializations during binding, second, it is the handling of incorrectly typed FIDL calls. In the first situation, a device can not be initialized and the `bind()` method of the corresponding driver ends with certain status codes which signal the device coordinator to try a number of initializations in the hope that a temporary issue can be solved. In the other one, the strong typing of FIDL calls becomes obvious. While a faulty buffer in an `ioctl()` call can potentially cause a lot of damage in Linux, FIDL call arguments are already checked at the beginning. If these do not match the defined call, the whole connection between user and the FIDL interface of a driver will be reset. Indeed, it is necessary to reconnect to the driver again, but wrong parameters can only cause little to limited damage by this check and the concept of distinct device hosts at all. In the attempts regarding the use of FIDL calls within the case study typically it was only possible to identify crashes and errors on the basis of informative log messages from the console. No further effects were observed and the driver could be used as usual.

The observed situations both belong to the desired effects of the microkernel or rather FIDL, even if they are unfortunately not documented at all. However, these positive effects also result in the significant performance losses discussed as part of the previous chapter. Whether one would accept these depends very much on the application of a kernel respectively a system. For Zircon, however, these losses are clearly accepted in favor of the advantages. Even though Zircon's impression in this thesis is very positive, this does not automatically imply that Linux is worse. The fundamentally different architectural concept is clearly visible but very much influenced by Linux's origins. It can also be easily seen that the developers of Zircon have considered problems and unpleasant issues in Linux and consciously implemented them in a different manner. Examples are, of course, FIDL, the context pointers and the whole break with the POSIX standard, as well. This development can also be traced back to the history of both systems and especially Linux. In addition, Zircon is currently not used for production. Thus, there is much more space for experiments. Of course, it would be very desirable that some of these experiments and concepts find their way into Linux over time. Especially in driver development, these could be context pointers and FIDL call definitions instead of I/O controls. For both of them a realization in Linux is technically conceivable, independent of the underlying operating system kernel architecture, but how likely a realization is, remains open. Accordingly, a kernel architecture has less influence on driver development than the use of modern concepts that address issues of other kernels such as Linux. However, the architecture and resulting effects of a kernel have a significant effect on its intended use.

Chapter 5

Outlook

At the present time, there are no reliable statements about the future of Zircon or Fuchsia as an overall operating system, but there are many speculations[1]. Whether and if so, when, Zircon will actually step out of the status of a research project remains open. Currently, speculation about the use of it as an operating system for larger Internet of Things (IoT) devices, possibly even with a display, seems to be the most promising. The concept of the microkernel including the present scheduling policy would fit. If the shown performance disadvantage compared to Linux is problematic in such an application remains uncertain, but interesting is as well the question if this aspect can be further optimized over time.

Important for the question about the future of Zircon and Fuchsia is, as well, the topic of this work. That Zircon can easily replace a widespread system like Android is doubtful, just because of the incompatibility of the kernels. Although the Fuchsia project is working on compatibility layers at application level for other important systems, the hardware and driver support is probably even more important. But there is no such compatibility layer for drivers. They would have to be re-developed from scratch. For Android devices, however, a variety of different hardware and therefore also drivers are used. Google as the main developer of Android has no control whatsoever. At this point, device manufacturers are often dependent on System on a Chip (SoC) manufacturers and so-called board support packages with necessary drivers as well. While the switch from Linux to Zircon is uncomplicated from a driver development point of view, economic interests in particular could be an obstacle here, even if there might be advantages for resulting products. It is more probable that Google will be testing and evolving Zircon respectively Fuchsia on their own products which are completely under their own control before attempting a major market launch.

Difficult to judge is, as well, the question of how Zircon respectively Fuchsia would be accepted by independent developers. The issue is less on application development as on the acceptance of the Zircon kernel itself. There are already enough approaches for compatibility in application development, so that hardly any difficulties are to be

[1] googlewatchblog.de, visited on 03.06.2019 `https://www.googlewatchblog.de/2019/05/fuchsia-daempfer-ist-googles/`

expected at this point[2]. However, the attraction of Linux and probably a reason for its widespread use is based on its open development. Zircon's development is open source as well, but not completely open as Linux. Currently, only employees of Google are involved in the active development. Minor error corrections from independent parties might become accepted according to official contribution documentation, but major changes are not. This could be a disadvantage compared to Linux and maybe also favor branches of Google's Fuchsia respectively Zircon in order to be able to integrate functionality needed by some interest groups.

In summary, the future of Zircon is still very open and daring. The kernel itself is fascinating and follows a well-designed concept for a modern, user-centered operating system. The effects of its architecture on driver development are mainly visible in a positive manner during usage, while driver development itself is rather influenced by elaborate implementation details. The switch from Linux or similar operating systems to Zircon is basically more influenced by corporate policy and economical considerations than by technical difficulties. Due to the good documentation, the change between C drivers becomes very easy, C++ drivers require currently a longer learning period due to missing information. In this case, the transition is rewarded by efficient, secure programming as well as the advantages of the microkernel architecture, but also the associated performance losses. Obviously, there are suitable application areas, but if the kernel will find widespread distribution remains a speculation at the moment.

[2]googlewatchblog.de, visited on 03.06.2019 `https://www.googlewatchblog.de/2018/04/kompatibilitaet-nachfolger-android-bruecke/`

Bibliography

[1] Albrecht Achilles. *Betriebssysteme : eine kompakte Einführung mit Linux*. Berlin: Springer, 2006. ISBN: 978-3540238058.

[2] Christopher Anderson et al. *Virtual Memory Object*. Oct. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/objects/vm_object.md.

[3] Mojtaba Bagherzadeh et al. "Analyzing a Decade of Linux System Calls". In: *Empirical Software Engineering* (Oct. 2017). DOI: 10.1007/s10664-017-9551-z.

[4] Rüdiger Brause. *Betriebssysteme : Grundlagen und Konzepte*. Berlin: Springer Vieweg, 2017. ISBN: 978-3-662-54099-2.

[5] Jonathan Corbet. *Linux device drivers*. Beijing Sebastopol, CA: O'Reilly, 2005. ISBN: 0-596-00590-3.

[6] Ed Coyne and Bruce Mitchener. *Zircon Scheduling*. Dec. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/kernel_scheduling.md.

[7] Matt Davis. "Creating a vDSO: the Colonel's Other Chicken". In: *Linux Journal* (Feb. 2012). URL: https://www.linuxjournal.com/content/creating-vdso-colonels-other-chicken.

[8] Nick Desaulniers et al., eds. *Compiling the Linux kernel with LLVM tools*. Feb. 2019. URL: https://fosdem.org/2019/schedule/event/llvm_kernel/.

[9] Nick Desaulniers, Greg Hackmann, and Stephen Hines. *Compiling Android userspace and Linux Kernel with LLVM*. Oct. 2017. URL: https://llvm.org/devmtg/2017-10/slides/Hines-CompilingAndroidKeynote.pdf.

[10] David Diamond and Linus Torvalds. *Linus Torvalds: Just For Fun. Wie ein Freak die Computerwelt revolutionierte. Die Biographie des Linux-Erfinders*. München: Deutscher Taschenbuch Verlag, 2002. ISBN: 3-423-36299-5.

[11] Jake Edge. "Building the kernel with Clang". In: Linux Plumbers Conference. Sept. 2017. URL: https://lwn.net/Articles/734071/.

[12] Todd Eisenberger. *Bus Transaction Initiator*. Apr. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/objects/bus_transaction_initiator.md.

[13] Todd Eisenberger, Roland McGrath, and George Kulakowski. *Event*. Sept. 2017. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/objects/event.md.

[14] Todd Eisenberger et al. *Virtual Memory Address Region*. Oct. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/objects/vm_address_region.md.

[15] Brian Swetland George Kulakowski Bailey Forrest. *Zircon Device Model*. Aug. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/ddk/device-model.md.

[16] Travis (travisg) Geiselbrecht et al. *IRC Chat Discussion About the Zircon Microkernel (Architecture, Design Goals, Language Support)*. Freenode IRC. Feb. 2019.

[17] Eduard Glatz. *Betriebssysteme : Grundlagen, Konzepte, Systemprogrammierung*. Heidelberg: Dpunkt, 2015. ISBN: 978-3864902222.

[18] Richard E. Gooch. *Why is the Linux kernel monolithic? Why don't we rewrite it as a microkernel?* Oct. 2009. URL: http://vger.kernel.org/lkml/#s15-4.

[19] Richard E. Gooch et al. *Section 15 - Programming Religion*. Oct. 2009. URL: http://vger.kernel.org/lkml/#s15.

[20] Scott Grahm and George Kulakowski. *Task*. Nov. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/objects/task.md.

[21] Hermann Härtig et al. "The performance of -kernel-based systems". In: *Proceedings of the sixteenth ACM symposium on Operating systems principles - SOSP '97*. ACM Press, 1997. DOI: 10.1145/268998.266660. URL: https://doi.org/10.1145/268998.266660.

[22] Nick Kralevich, Roland McGrath, George Kulakowski, et al. *Fuchsia's libc*. Mar. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/libc.md.

[23] Rob Krten. *Simple Drivers*. Dec. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/ddk/simple.md.

[24] George Kulakowski. *FIFO*. Apr. 2017. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/objects/fifo.md.

[25] George Kulakowski et al. *Zircon Kernel objects*. Feb. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/objects.md.

[26] *LFD430 Developing Linux Device Drivers*. v4.19. The Linux Foundation. The Linux Foundation, 2018.

[27] Peter Mandl. *Grundkurs Betriebssysteme: Architekturen, Betriebsmittelverwaltung, Synchronisation, Prozesskommunikation, Virtualisierung (German Edition)*. Springer Vieweg, 2014. ISBN: 9783658062170.

[28] Travis Geiselbrecht Anna-Lena Marx. *IRC Chat Discussion About I/O in Zircon*. Apr. 2019.

[29] Roland McGrath and George Kulakowski. *Event Pair*. Sept. 2017. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/objects/eventpair.md.

[30] Roland McGrath, George Kulakowski, and Garret Kelly. *Zircon vDSO*. Aug. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/vdso.md.

[31] Roland McGrath, George Kulakowski, Brian Swetland, et al. *Zircon Kernel Concepts*. Dec. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/concepts.md.

[32] Roland McGrath, Carlos Pizano, and Bruce Mitchener. *Zircon and LK*. Oct. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/zx_and_lk.md.

[33] Roland McGrath et al. *Futex*. Dec. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/objects/futex.md.

[34] Bruce Mitchener. *Getting Started*. Oct. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/ddk/getting_started.md.

[35] Mike Voydanoff Christopher Tam Bruce Mitchener. *Zircon's Platform Bus*. Oct. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/ddk/platform-bus.md.

[36] Carlos Pizano, Scott Grahm, and George Kulakowski. *Job*. Nov. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/objects/job.md.

[37] Carlos Pizano et al. *Channel*. Dec. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/objects/channel.md.

[38] Carlos Pizano et al. *Process*. Dec. 2017. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/objects/process.md.

[39] Carlos Pizano et al. *Thread*. Nov. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/objects/thread.md.

[40] Jürgen Quade and Eva-Katharina Kunst. *Linux-Treiber entwickeln : eine systematische Einführung in die Gerätetreiber- und Kernelprogrammierung - jetzt auch für Raspberry Pi*. 4., aktualisierte und erw. Auflage. Heidelberg: dpunkt-Verl., 2016. ISBN: 9783864902888; 3864902886.

[41] Richard Rashid et al. "Mach: A System Software Kernel". In: (Sept. 1992).

[42] James Robinson et al. *Socket*. Nov. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/objects/socket.md.

[43] Rusty Russell. *Unreliable Guide To Locking*. 2016. URL: https://kernel.readthedocs.io/en/sphinx-samples/kernel-locking.html.

[44] Abraham Silberschatz. *Operating system concepts*. Hoboken, NJ: J. Wiley & Sons, 2009. ISBN: 978-0470233993.

[45] Brian Swetland et al. *Introduction*. Oct. 2017. URL: https://github.com/littlekernel/lk/wiki/Introduction.

[46] Brian Swetland et al. *Zircon*. Oct. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/README.md.

[47] Brian Swetland et al. *Zircon Handles*. Jan. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/handles.md.

[48] Brian Swetland, Roland McGrath, and George Kulakowski. *Zircon Signals*. Feb. 2018. URL: https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/signals.md.

[49] Andrew S. Tanenbaum and Herbert Bos. *Modern operating systems*. 4. ed. Boston: Prentice Hall, 2015. ISBN: 978-013-359-162-0; 978-1-2920-6142-9; 0-13-359162-X.

[50] Andrew S. Tanenbaum, Linus Torvalds, et al. *LINUX is obsolete*. Jan. 1992. URL: https://groups.google.com/d/topic/comp.os.minix/wlhw16QWltI/discussion.

[51] Jürgen Wolf. *C von A bis Z : das umfassende Handbuch ; [inkl. CD-ROM mit Openbooks und Referenzkarte mit wichtigen Befehlen*. Bonn: Galileo Press, 2009. ISBN: 978-3-8362-1411-7.

[52] Jason Wu, Dan Williams, and Hakim Weatherspoon. *Microkernels: Mach and L4*. 2010. URL: https://www.cs.cornell.edu/courses/cs6410/2010fa/lectures/06-microkernels.pdf.

List of Abbreviations

ABI Application Binary Interface . . . 64
API Application Programming Interface . . . 12
ASCII American Standard Code for Information Interchange . . . 83
BTI Bus Transaction Initiator . . . 54
CBS Constant Bandwidth Server . . . 42
CFS Completely Fair Scheduler . . . 41
CPSR Current Program Status Register . . . 7
CPU Central Processing Unit . . . 1
DMA Direct Memory Access . . . 48
DPRR Dynamic Priority Round Robin
DSO Dynamic Shared Object . . . 64
EDF Earliest Deadline First
ELF Executable and Linking Format . . . 16
FCFS First Come First Serve . . . 38
FIDL Fuchsia Interface Definition Language . . . 64
FIFO First In First Out . . . 32
futex fast user-space mutex . . . 28
GCC GNU Compiler Collection . . . 13
GPIO General Purpose I/O . . . 92
GPU Graphics Processing Unit . . . 66
GUI Graphical User Interface . . . 71
I²C Inter Integrated Circuit . . . 62
IDL Interface Definition Language . . . 14
IDT Interrupt Descriptor Table . . . 14
I/O Input/Output . . . 1
IoT Internet of Things . . . 119
IP Internet Protocol . . . 34

IPC Inter-Process Communication . . . 10
LCD Liquid Crystal Display . . . 67
LWP Light-Weight Process . . . 22
MMU Memory Management Unit . . . 46
MMIO Memory Mapped I/O . . . 92
PC Program Counter . . . 19
PCB Process Control Block . . . 20
PCI Peripheral Component Interconnect . . . 61
PFN Page Frame Number . . . 50
PID Process Identifier . . . 22
POSIX Portable Operating System Interface for UniX . . . 2
PSW Program Status Word . . . 7
RAM Random Access Memory . . . 44
RGB Red Green Blue . . . 67
RPC Remote Procedure Call . . . 64
RR Round Robin
SMP Symmetric Multiprocessing . . . 8
SoC System on a Chip . . . 119
SRTF Shortest Remaining Time First
SSD Solid State Drive . . . 44
TAS Test and Set . . . 27
TCB Thread Control Block . . . 22
TCP Transmission Control Protocol . . . 33
TLB Translation Lookaside Buffer . . . 46
TSL Test and Set Lock . . . 27
TTL Transistor-Transistor Logic . . . 70
UART Universal Asynchronous Receiver Transmitter . . . 70
UDP User Datagram Protocol . . . 34
UI User Interface . . . 38
USB Universal Serial Bus . . . 61
vDSO virtual Dynamic Shared Object . . . 16
VMAR Virtual Memory Address Region . . . 24
VMO Virtual Memory Object . . . 35
XCHG Exchange . . . 27

List of Listings

List of Figures