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 20. Juni 2019

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 its user. As such an operating system, to manage a computer's physical resources and share them between an amount of user programs in a fair and safe way, is the central task. The part of operating systems providing this task is commonly referred as the *kernel*. Even if this distinction is widely uses, especially for Linux, there is hardly 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 that hard because it is not even consistent between different operating system architecture concepts. The lot of different goals and use-cases for operating systems which have a big impact to the architectural design, and with this to the term *kernel* too. But for a majority of them, at least the central *device drivers* are considered as 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 monolith and the microkernels. *Fuchsia*, respectively *Zircon*, as the kernel is named, is a new microkernel operating system which is developed from scratch by *Google*. The project is developed in public since 2016[1], but since it gone public, there has been no official announcement about the purpose of the system. Nevertheless, there are lots of reasons to study *Fuchsia* and especially to its kernel *Zircon*. The first one is clearly an architectural one, as *Zircon* is a microkernel. Of course are there some industrial used microkernel architectures, but the most and 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

[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/`

this very important aspect, the POSIX compatibility which provides a global standard (DIN 9945, IEEE1003.2) for UNIX-like operating systems[53]. 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[50].

Linux is in many issues 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 at all[2]. Since Linux was published, it is actively developed by a divers community and is not subject to the interests of a single company. Presumably, that is a powerful 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*, are based 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 admit the theoretical superiority of microkernels over monolithic ones[51]. But some years later, in 2006, the Linux contributer Richard Gooch justified the question of why *Linux* won't change to a microkernel architecture with performance drawbacks of them compared to monolithic ones due to privilege barriers[18].

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

1.1 Research Objectives

This work's aim 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 only 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 unattended in this work. Of course, they are essential, but there exist that much different concepts and implementations for filesystems to form an own research. For this particular work is the specific filesystem implementation not crucial due to the focus on kernel and device drivers. As a result, the operating systems top layers, neighter Google's *Fuchsia* in its entirety nor the *Linux* userland or desktop environments shall be a part of this thesis.

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

- Introducing basic concepts of operating system kernels and examine how they

[2]netmarkedshared.com, visited on 13.02.2019 `https://bit.ly/2SOs4lM`

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 to 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 view 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 temporal evolving 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*[51] in 1992. Besides the discussion is quite interesting from today's view on different operating systems, the forecasts on their future and the actual development, are both, TANENBAUM and TORVALDS underpinning their arguments with the origins of their implementations *MINIX* and *Linux* in different problems. And roughly spoken are exactly such different problems which needs to be solved with an operating system 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 many ways of working and architectures. As with the debate, it is rather difficult to impossible to find one architectural concept or implementation which is clearly superior to the other ones 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 still 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 especially for 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 are architectural decisions for operating systems 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 outcome. According to GLATZ[17] are some of them:

- 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*. That means an operating system design should provide a clear distinction between the *mechanism*, that means the capable abilities 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 is that requirements may change over some time and such a distinction makes it easy to adjust the *policy* via some 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 before, 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[50]. 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 utilize ring 3 for user-space applications, 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. Actual operating system implementations may not use all of them. Linux for example utilize ring 3 for user-space applications, shared libraries and ring 0 for system calls and the 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. Just 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 ring 3, running in the *user mode*, are not allowed to them so directly, they have limited privileges and a limited instruction set. As named 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*.

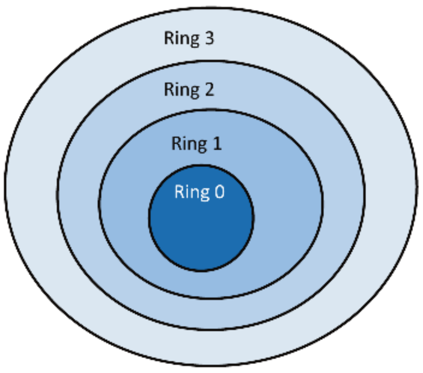


Figure 2.1: The Rings of the x86's security concept[17]

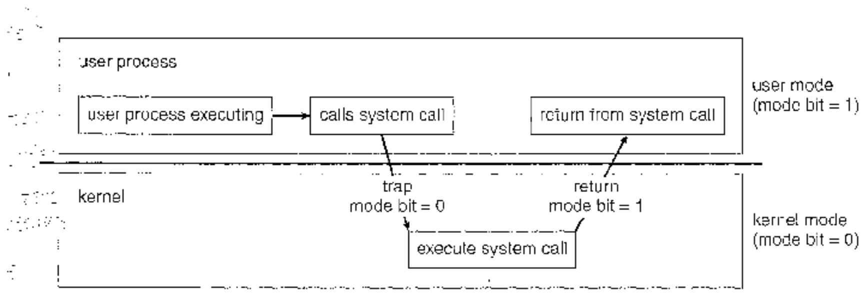


Figure 2.2: A system calls sequence including the mode switches[17]

The CPU's operating mode is usually controlled by a specific bit in the Program Status Word (PSW)[50]. PSW is term on x86 CPU architectures. It is corresponding to the Current Program Status Register (CPSR) on ARM architectures[1]. But since in literature is always spoken of the PSW regardless of the concrete CPU's architecture, this should also be done in this work. It influences the state of each CPU core itself in a

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

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 have an influence on the whole system, including device driver development but also on 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. On the contrary, this work will not take a particular look on special purpose operating system architectures such as ones for loosely coupled multi-processor systems known from processing clusters. Today, even the majority of general purpose computing systems are driven by more than one CPU cores and most of 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 but also the hardware, without any limitation, e.g. regarding the access of kernel functionalities of another component or hardware access. That means a function or procedure initial 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 could a crash in one single function or procedure crash the entire kernel or the resulting system may become difficult to understand and maintain[50], [44]. The missing memory protecting 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 (in common) dynamically loadable 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], [50]. Just like ordinary kernel functions or procedures, (malicious) programming errors in extensions may lead to a kernel crash or manipulate or damage other components. Contrary, the modular concept provides some advantages over regular monolitic kernels. It allows slimming down the actual kernel by providing the chance to reload only the actually needed functionality dynamically and e.g. security patches within such an extension is possible without restarting the entire system[5]. 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. Thus, concept of modules is quite popular for basically monolithic operating systems like *Linux* or *Solaris*[5], [44].

2.1.2 Microkernel Architectures

The microkernel architecture focuses on very opposite design goals compared to the monolithic one. Some of them are to cope the complexity, rather poor maintainability and susceptibility to errors by a massively modular approach. To archive 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 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[50].

What is part of this core functionality differs between miscellaneous 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]. In academic microkernel approaches, all components in user mode run within an own userspace process as small, well-defined modules, while the communication is done through copious message passing via the actual kernel[50], [26]. Since the restrictions by the CPU's dual mode still apply for microkernel based operating systems is it not allowed to 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 substitional. But thus, the kernel is able to check the action and whether the driver is authorized to executed them. Resulting, 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[50]. Equally, a crash in a userspace system component like a driver is not able to crash the entire kernel in such an approach. And as an additional advantage facilitate the microkernel architecture porting the operating system kernel to another target architecture as the most hardware dependencies are part of the small kernel[44], [26].

With all the named advantages microkernels offer, the question remains why microkernels are only spread in real-time, avionics or military but not for desktop operating systems. One reason is that all these advantages are bought at the high price of microkernel message costs. For the named application areas, especially the reliability that comes with the microkernel architecture is more desirable than the performance costs of the lot more context switches in comparison to monolithic architectures[50]. 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 trigger 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 displaced from the cache and the CPU is largely blocked in the meantime. By rapidly switching back to the previous context, as 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 as *Linux* it is[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, are pure microkernels 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[50].

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 cooperating between the layers or rings is regulated by clearly defined interfaces[5]. This is usually accompanied by the fact that the lower layers or inner rings are more privileged as the outer ones. However, there is no uniform and universally accepted approach for division in layers and their count according to this pattern[17], [50]. 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 considered properly to get a clean architecture, it that can 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. is the MINIX userspace 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], [50].

2.1.4 Hybrid Architectures

Hybrid operating system architectures based on monolithic, microkernels and may be the layered ones are 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[54], [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*[54].

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's, university, it becomes strongly related to its archetype, *Unix*[10]. They share fundamental design goals, just like being capable of multiple processors and user at the same time, but Linux is not based on the origin Unix source code[50]. The overall architecture of the Linux kernel is, as already named, monolithic and also inspired by Unix[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 to 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 to 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], [50]. 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 to the 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 part 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], [9]. 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 which is also a part of the Zircon Team at Google[33]. 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[33].

The mirokernel architecture is justified by having security, safety, reliability and modularity as major design goals for Zircon. According to Travis Geiselbrecht was the architecture a conscious tradeoff between the named goals and performance[16]. They try avoid costly context switches as much as possible, speed up the remaining ones and take advantage from 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[23]. The Zircon kernel itself is splitted 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.

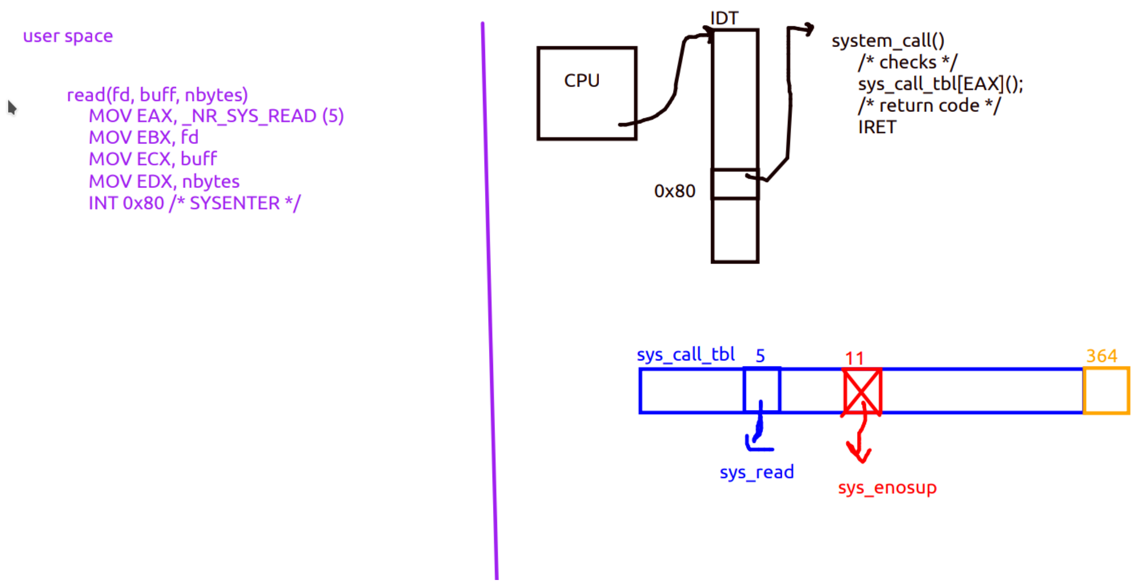
Zircon is for the most parts written in C++ and less in C. It provides native support for device drivers in both languages but due to the fact Zircon provides an Interface Definition Language (IDL) which defines a contract for in-process drivers, other languages are conceivable as well. In fact, support 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, this special calls are needed for the interaction with basic hardware devices like the CPU, the memory, peripherals or filesystems and for invoke 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 substition, it has to use one for switching the operating mode[17], [50]. 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 2.3), another one is to store them on stack, according the C/C++ calling 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 an other 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], [50]. To be exact, it is the interrupt vector number of the trap instruction which is responsible for the switch. They are `080` on Linux as pictured in 2.3 and `02e` on Windows systems[17]. But right before switching it is needed to save the Program Status Word (PSW), which contains the actual processors state includes 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 table, IDT) which contains the start address of the system service dispatcher routine (compare to 2.3). This tables content, the system service dispatcher routine, is loaded as next instruction to the new PSW[5]. 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 simplified pictured in 2.3[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 stack containing the bit for CPU's user mode execution. The user application has to clean up the stack like for each procedure call at the very end[50].

Figure 2.3 shows a simplified version of the system call implementation on x86 for Linux kernel before 2.5[3]. More modern versions use the special instructions `SYSENTER` and `SYSEXIT` (Intel) or `SYSCALL` and `SYSRET` (AMD) instead of the slower trap interrupts[3].



```
read(fd, buff, nbytes)
   MOV EAX, _NR_SYS_READ (5)
   MOV EBX, fd
   MOV ECX, buff
   MOV EDX, nbytes
   INT 0x80 /* SYSENTER */
```

```
system_call()
   /* checks */
   sys_call_tbl[EAX]();
   /* return code */
   IRET
```

Figure 2.3: System Call Implementation[26]

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 comply them, 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], [50]. 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[50], [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

In fact, 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 depending on the CPU's architecture. While a 32-bit Linux kernel in version 4.8 for the x86 architecture offered 379 calls, the 64-bit version for x86_64 offered only 328[26]. The *man-pages* project documents gives 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 spread 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 the most of them are using a handle which describes the kernel object to work on[32]. 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 handles parameters refers to one existing within the calling process's handle table and
- a handle has the necessary rights for the triggered action[32].

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]`https://www.gnu.org/software/libc/`

[3]`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[8]. Thus, the Zircon vDSO is the only way to perform system calls[31], 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 and the kernel's system call implementation[32]. Also in contrast to Linux does Zircon respectively Fuchsia 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[23]. The libzircon.so does not support directly I/O operations. They are performed by the *fdio.so* library which overwrites weak symbols of the libzircon[23]. 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, may 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 good 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[50]. 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 pseudoparallel execution of multiple programs based thereon[44], [50]. 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[50], [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[5]. 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 need to be stored and the way it is done slightly depends on an effective system's design and its implementation. In general, these informations are stored in a structure called *process control block* in literature[50]. 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[50], [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 2.4. 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 on 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, so it changes its state to *waiting* or *blocked.* Only 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 *terminate*. The operating system destroys the process and frees related resources[44], [28].

3.1 Process Concept 103

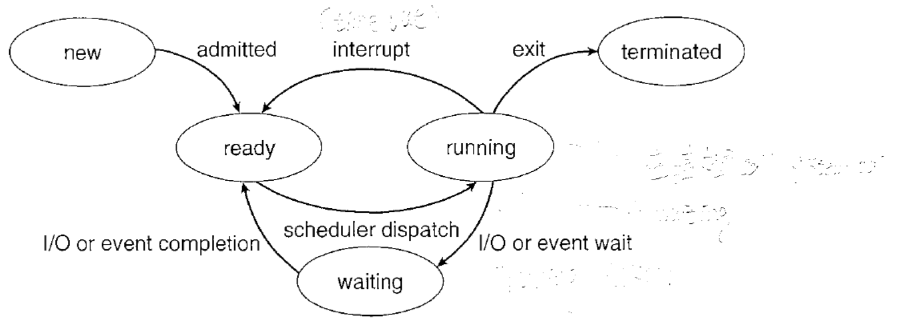


Figure 3.2 Diagram of process state.

Figure 2.4: Lifecycle of a process[44]

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[50].

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 is loaded 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[50].

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 are numerous processes created to execute parts of the system itself. The user perceives very few of them directly, since the most of them are *background processes* performing specific tasks like accepting incoming mails. Often, long running processes without user interaction are referred as *deamon processes*[17], [50]. They are mostly generated from the *init 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[50], [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]. An example in Unix-like systems is calling `exec()` on an already running process to replace the current program with another one[17], [50]. By *forking a process*, a second one that is, at least in the beginning, a copy of the original one. Both remains exiting and share the same environment such as program code, address space and resources[50]. As a result, both processes have access to the exactly 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[50]. In contrast, Windows offers a third way the *process creation* which combines them in a single instruction to start a second, independent process (`CreateProcess()`)[17].

Are 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*. Is it the case, e.g. for the once 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 one 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* takes 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 2.4), a process terminates itself regularly and voluntary (*normal exit*)[50]. 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[50]. Besides, a process may gets involuntary terminated or killed by another process. Killing another process is in common a privileged task which require an authorization or extended rights[50].

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)[50], [5].

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[50]. This means processes only group the resources together while *threads* represent the actual execution units on the CPU[50]. 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[50]. A thread's lifecycle is equivalent to process's ones, but in contrast to them are threads 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[50]. Another issue are competing threads that try to write the same global data. They cause a *race condition* which possibly result in inconsistent or wrong data[5].

Nevertheless, the advantages of using *threads* (multithreaded programming) predominate. The most important one is resource sharing. The fact multiple threads using the same resources by default, e.g. files, in pseudoparallel 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 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], [28]. 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)[28].

How *light-weight* they actually are depends on a lot to 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) (according to PCB) for each single process[28], [50]. 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[50], [5]. 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 benefits 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 according to the process table[50], [28]. 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[50].

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[50]. 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[28]. 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 splitted up[50], [28]. 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 new 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. Childs get 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[50]. 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[28].

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 are application using the clone call directly not portable[50].

Nevertheless, *clone()* is a very interesting concept in Linux. In each case, a process or thread created with clone gets an empty private stack and executes directly a new program which is given as an argument to the call. The decisions 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[50].

The flags listed show only a partial amount of the possibilities *clone()* offers. A full list 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[28].

2.3.4 Processes and Threads in Zircon

Similar to Linux, runnable entities of the Zircon kernel are called *tasks* objects. 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 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. According to Linux, jobs built 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 is managing

[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/`.

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 in this place. 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 an own VMAR to built 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 thread exits, it is impossible to create a new thread within this process. Processes which are composed to 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 like known from the general section are Zircon *threads* the actual runnable computation entity living within a process. They are created within the *process* context 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 voluntary 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 involuntary

- if the parent process is terminated,

- if someone calls `zx_task_kill()` on the thread's handle and
- if an exception was generated for which 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. In this context mentioned already the previous section 2.3.2 that the ability of threads to access shared data not only has advantages. 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. May 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 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*[50]. The section 2.4.1 deals with 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 the definition, the term *thread* refers in this section 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 voluntary[44]. In such a case 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 is it 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 exclude the chance that two threads try to change a shared resource at the same time[50]. But only certain areas in the program code are critical, the ones a shared resource is processed. They are called *critical regions* or *critical section*[50]. Other regions do not harm if they 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*[50], [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[50].

While there are some approaches to pure software solutions such as the algorithms of Peterson or Dekker (see [50] or [44] for further information) are hardware enabled solutions common today[50]. 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[50]. 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[50]. 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*[50]. 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 typ 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 programmer. 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[50]. For a semaphore, the fact updates on its value must be performed in an atomic operation is essential[44]. Neighter 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. These particular case of a binary semaphore is also referred as *mutex*. A *mutex* guarantees mutual exclusion for a critical section protected with its related operations as *lock* (P, *down*) and *unlock* (V, *up*)[50]. 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 is a blocking mechanism 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[50]. futexs are not a pure userspace construct even if the name suggests. They still need a wait queue in kernel to manage the contenting 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. An 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 to the queue. But if there is no contention is there also no need to involve the kernel[50]. 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 coordinate and 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[50]. 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 stoven 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 alternating implemented on the basis of blocking semaphores[17].

2.4.2 Inter Process Communication

While the previous section 2.4.1 focuses on threads, the actual one 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 performance are design aspects and their effects an essential component to be considered.

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 allocated the memory and show it in the address space of both processes[5]. The shared memory region is the responsibility of the processes, must be managed and secured against incorrect access by themselves[5], [17]. As known from thread communication, the advantages from 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, breaks this solution the isolation between processes. A variant of this idea is using *files* or similar resources

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

as a communication medium. In this case, the process isolation remains, but manual synchronization, as described in the previous section, is still necessary[5].

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 provides by the operating system respectively its kernel via two primitives: *send()* and *receive()*[50].

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 quantify of data which is defragmented on 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 resources 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[5], [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[5].

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], [5]. Only if the buffer is completely filled, the sender has to be blocked and vice versa the receiver if the buffer is emptied. 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[50].

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 alternate act 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*[5], [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 small amount of data to transmit. Is only worthwhile if the reliable transmission between designated instances is more significant as 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 neccessarily 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], [5]. 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 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], [50].

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 the concept of futexes to provide an efficient locking mechanism[50]. 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 one processes 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 in common send between the kernel and a procces, e.g. to indicate illegal instructions or memory access, arithmetic errors, expiring timers and much more. Only `SIGUSR1` and `SIGTERM` are send between processes. The 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 grateful 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 is it convention to raise the same signal with its default action again right after the custom handling finished[1]. Each Linux process is aware to receive 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 in Linux wide-spread 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 another than the creating process. Thus, pipes are commonly used between a parent process and its childs or between childs. The parent creates a pipe which is inherited to forked childs[44]. For further reading, Achilles gives 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 previous. In contrast to them are named pipes even between unrelated processes available. Several writers or readers on a single pipe are possible. Indeed, named pipes are capable for 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 influences the pipe just as little as a system reboot[44]. That is not the fact for ordinary pipes as they are bounded to the creating process and its childs. 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 bounded 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 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[32], [34]. The Zircon futex implementation is very efficient. It does not enter the kernel space or allocate any kernel resources in the uncontested case. In contrast to the Linux futex implementation do Zircon futex operations key off of the virtual address of a userspace pointer while Linux differentiate private futex actions from the ones shared across the address spaces of more than one process. Futher, Zircon futex operations do not modify the futex from the kernel, such an action is not required in the current implementation[34].

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*[30]. 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 signal 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 so-called *Virtual Memory Objects (VMOs)* and *VMARs* which represents a contiguous region of virtual memory respectively of virtual memory address space. Both go together and are mixed up sometimes. Generally, the term VMAR is used for any operating according to address regions, e.g. allocate them, map and unmap VMOs to processes, but also to protect and destroy regions[14]. The term VMO is used for actions on the memory object, such as create one and read or write it[2]. A VMO represents a set of physical memory pages on the one hand, but also the potential for pages on-demand via lazy creation on the other. Such objects are available on both, kernel and user-space and can be mapped into the address spaces of multiple, distinct processes[2], [32]. VMARs in Zircon are randomized by default and additionally allows adjusting the permissions of mapped pages[14]. The mapped VMOs are not only directly writable and readable. VMOs also provide an one-shot mechanism to avoid the cost of mapping them more often into address spaces than necessary. In such a case, a VMO is created and initially written by one process and subsequently handed over to the control of another process[32]. Thus, they are very suitable for efficient IPC via shared memory.

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 on end of the channel. Channels provide a datagram oriented message based communication between arbitrary and unrelated processes[37], [32]. 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[32]. 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. In contrast to pipes on 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[32], [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[32], [42]. Just as channels, sockets allow a bidirection 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[32].

FIFO The last IPC object Zircon provide, is a FIFO. There is no much documentation about yet, neighter in the usual documentation version used nor in Google's master repository ones. 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[5], [28]. 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[28]. 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[28]. 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[5]. 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 read to run to a CPU core[5], [28].

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[28]. 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 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 are 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[28], [5].

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[5].

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 voluntary hands over the control back to the system, e.g. because it is waiting for a resource or an event or it reaches its completion[28]. 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, but for batch jobs. For these systems, a preemptive scheduling strategy with generally interruptible jobs is more appropriated. 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 a certain policy[5]. This makes such a system much more responsible.

Whether a scheduling policy should be aware to 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 reactive system by prioritize User Interface (UI) related jobs or to ensure the fastest possible processing of a security-critical task in a realtime system[28]. In general, prioritized jobs are available on both, preemptive and non-preemptive policies[5].

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[5], Mandl[28], 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[5]. A global component named *load balancer* can step in if the load on an individual core is much higher as 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 putted in is also the first one how 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], [28]. 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[5]. Is the slice too small becomes the overhead caused by switching the process contexts superior, is its size chosen too long, the resulting strategy evokes on FCFS more likely. The calculation of the next job to schedule is very cheap, using RR as a strategy[5].

Dynamic Priority Round Robin (DPRR) is a priority aware modification of the usual RR scheduling strategy. The only point that differs is an additional queue in front of the known one. Each job arrives in the first queue with a priority and stay there until this value reaches a certain threshold. The priority value of each job in the preceding queue is increased after a time slices 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 little more management effort[5].

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[5], [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*[5], [28]. 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 become processed first. The ones in next lower priority are not touched until the higher ones are emptied. As a consequence are situations 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 appropriated 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 this strategy is allowed to stay active until the worst ratio is calculated for another one[28].

Earliest Deadline First [28], [5] (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[28], [5]. Nevertheless, is EDF a common algorithm for hard realtime systems and an ideal strategy on single core systems with up to 100% CPU utilization[5].

Priority Inversion

Using locking mechanisms like semaphores respectively mutexes may lead to dangerous behavior in context of priority based scheduling. For a basic example are 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 contested 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[50]. 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 runned before one with a high priority. Such a situation is referred as *priority inversion*[50]. 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 becomes waiting for a resource held by the L job. When H blocked, the M job was scheduled and had to 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 may 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 systems 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 tree 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 on by itself. As scheduling in Linux is based on time slices, is it the mission 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,

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

- the scheduler becomes invoked and finds another job that has to be scheduled according the current policy[26].

Linux uses one scheduler per CPU core and a global load balancer to move jobs between them, just as named 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 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 finding a decision 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 were they 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.
- Replace common interrupt handlers with preemptive kernel thread so that they can be scheduled and sleep[26].

The scheduling in Linux is very finely granular adjustable to the system and its applications needs. It is not only possible to select the grade of preemptibility and make it aware for 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 Bandwith 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 provide 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 section. The scheduling is done for each CPU core at 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[7]. 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. The 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 prioity. 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[7].

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 an one level upgrade when it unblocks after waiting for a resource or sleeping.

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

  - A job is penalized with an one level downgrade if it voluntary give up the control. The minimum priority value in this situation is 0.
  - A job is penalized with an 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 inherit one[7]. 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 is it 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[7].

A special role belongs to the *idle thread*. It lives besides the normal FIFO queues and is runned 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[7].

At the time of writing, an additional CFS is in work. Its API is compatible to the one Linux uses. Is it 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

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

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 is nearly equivalent to registers in terms of access time and costs, and thus, very small. After two to four cache levels follows by main memory or Random Access Memory (RAM). If it is spoken about memory 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][28]. 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 together with the needs of processes, i.e. the isolation between the address spaces of distinct processes but also a way to share memory for IPC (see 2.3 for the details)[28], [5] A pure static and direct memory management would not be sufficient for these purposes. In current systems, this is mostly realized by using the mechanisms of *address spaces* and *virtual memory*, which already have been mentioned several times.

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*[28]. Distinct parts within this layout are protected against each other, especially the system kernel itself[5]. 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[28].

The term *address space* is used for the memory region of applications, through, which is effectively located within the main memory's address space. Their intern 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[28].

However, the pure implementation of divided address spaces does not yet solve all the needs to be named above. Although the address space at its whole is now subdivided into various regions, it is not ensured that the address space of a process

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

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[28]. Furthermore, data should be aligned to addresses which are divisible by four to make accessing main memory from the CPU more efficient[28], [5].

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 programmer, only see a contiguous memory region starting with the address 0 but not the actual fragmented main memory used by various processes from distinct users[28], [5]. Such a virtual view to the actual main memory also enables pretending more available memory 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 which 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 in hard drives. As a result, the overall memory consumption may become greater than the physical available main memory and virtual memory is not neccessarily mapped to physical memory[28]. 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 current needed ones, and
- a *cleaning policy* trying to keep some main memory free at any time[28].

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 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[5]. 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[5].

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 the memory management take advantage from sophisticated concepts like *demand paging*, i.e. a page is only loaded if there is a need for, and *prepaging* with use of the so-called *locality effects*. The two most important ones are the *temporal locality* and the *spartial* 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[28], [5].

Above all, virtual memory enables the sophisticated memory management requirements needed for a 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[5] 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 calculates the physical one according a defined algorithm 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 CPU 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 the currently used policy[28].

The TLB works together with the MMU. It is an additional fast cache to speed up the assignment between virtual and physical memory addresses. As described previously is accessing a cache still considerably faster than accessing the main memory itself. Similar to common L1 and L2 CPU caches is there an own TLB per CPU. If the needed address translation is already stored in the TLB, no further access to the main memory is needed, if not must the MMU be invoked. 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[28] or Brause[5].

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 is (virtual) memory management in Linux 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 correspond 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 wide-spread 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 2.5 illustrates the context.

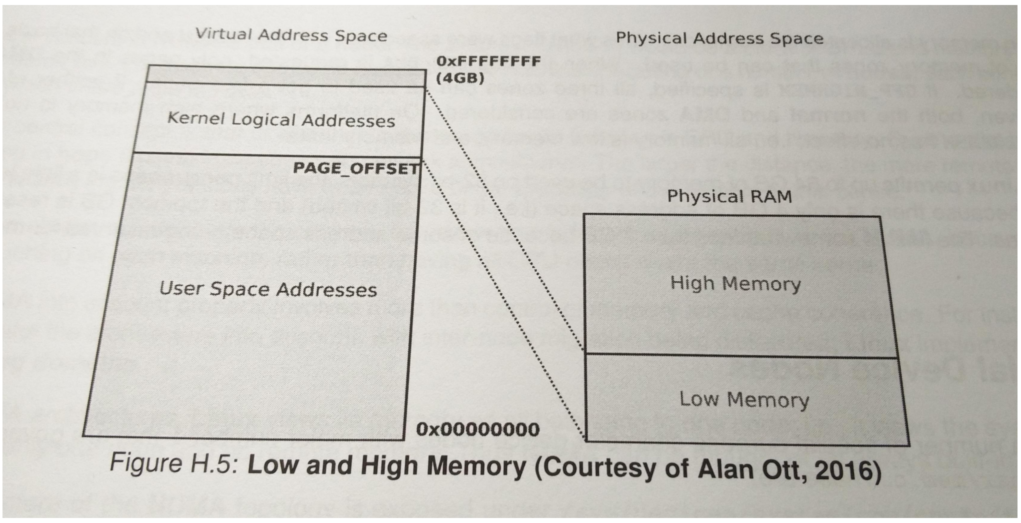


Figure 2.5: The relation between virtual and physical memory in 32-bit x86 architectures[26]

Actually, the division in high and low memory just one from several ones. Linux uses additionally a *zone allocator* memory algorithm to partition its address space more fine granular. Each zone differs in its usage and comes with own methods to perform basic memory operations like allocate and free 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:

- **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, on this place, 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].

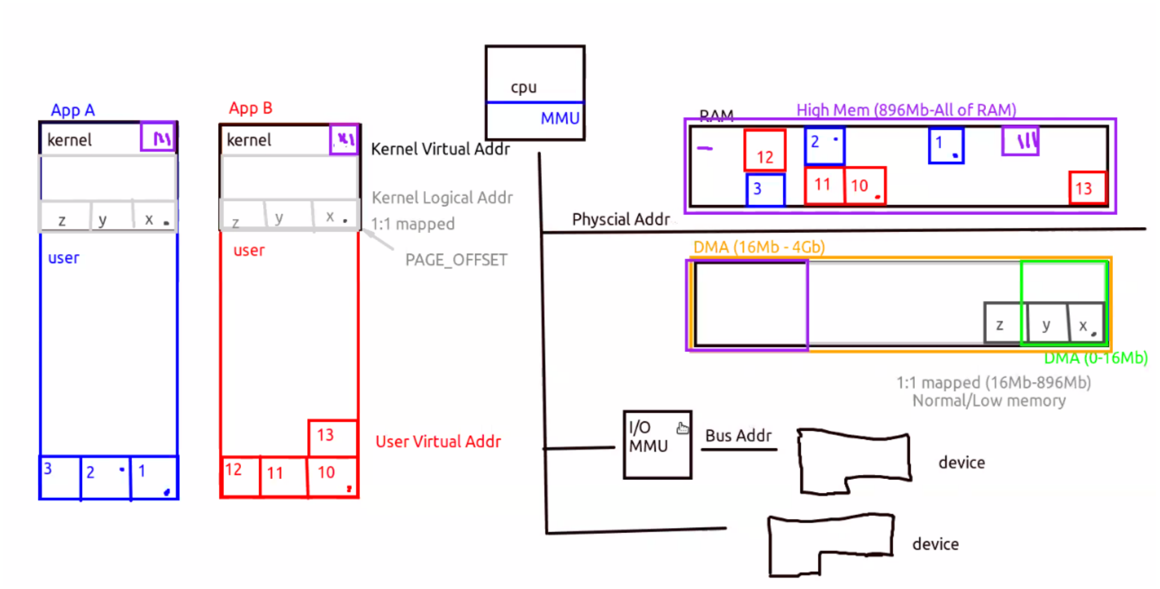


Figure 2.6: Linux memory management overview[26]

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 is the virtual or logical address 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 2.6 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 blue 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 grey and the *high zone* or high memory in *purple*. 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 are some kernel pages marked in grey 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 named, 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 according 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 eighter less complex than in Linux, even if the implemented concept itself is basically the same, or the available documentation provides just a little insight. All available texts on this topic includes 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 belongs 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 to regulated 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 there 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*[32], [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 that 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 is it possible to set specific cache policies for a VMO[32], [2].

2.7 I/O

The term Input/Output (I/O) is commonly used to describe the interactions between a computer itself and several devices types such as disks, keyboards, displays, clocks and much more which are usually referred as *peripheral devices*. Thus, it enters the actual topic of this work: device driver development. The majority of device driver development deals with I/O devices and related issues, such as control the device via a defined interface, catch and process interrupts from a device and handle interrupts. 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[50]. Nevertheless, this section will focus on the communication mechanisms between I/O devices and operating systems, but not about 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 in order that different devices from the same type, for example a few different mices, can be controlled in an uniform way[50].

Peripheral devices are divided into two categories all in all. *Block devices* are in common used to store information in fixed size blocks. Like internal memory has each location within such a device 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[50]. 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 consider any block structure. The data is not precise 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[50]. But some devices does 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 abstract 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 give 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[50], [17].

2.7.1 Memory Mapped I/O

The first and also wide-spread 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 programmer's view is accessing such a control register 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 the 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[50], [5].

2.7.2 Direct Memory Access

Another way to interact with a peripheral I/O device is Direct Memory Access (DMA). In earlier days, it was used on CPUs where *memory mapped I/O* were not available. 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 external device and memory. As a prequisit needs this controller 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. If the whole transfer is done sends the controller an interrupt to the CPU and return 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[50], [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 to signal 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 are interrupts popular for keyboards or hardware clocks[50], [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 this actions, but these are, especially on Zircon, on a system level even a driver developer does not or rarely enter. 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[29].

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 is proceeded 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 and passable environment. If anyone is in need for a certain concept can it 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, but not open to receive contributions from outside. Even if the concepts used do not differ that much, the implementation does. This is justified on the microkernel architecture and its effects but also with the used programming languages and above all with learnings from concurring operating system, especially the ones used in Google's environment like Android or Chromeium 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 but the focus is 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, probable 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 a good knownledge about the peripheral device and its controller, the hardware interface between device and computer and the target operating system from the programmer. 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 this questions is a valid entry point to the actual case study about the driver models in Linux and Zircon and the exemplarily device driver development.

A device driver's main purpose is providing an abstraction between user applications and peripheral devices[17]. A common programmer 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 the are managed, it would require too much knowledge in applications and make them 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,
- do the connection between applications and a certain peripherals,
- initialize the peripheral controler 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], [50].

Driver development is very operating system dependent and has wide-ranging consequeces. While the decision if drivers are located in user or kernel space goes alongside with the choice for an architecture, remain some further questions. 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, e.g. during runtime[50]. 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 typ or standardized for all drivers[50]. Furthermore, in almost every operating system specific driver model, it has to be ensured a driver is *reentrant*, 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[50]. In order to avoid re-developing or duplicating driver parts that remain the same for devices of the same type are device drivers often modelled and implemented as a hierarchical or layered model. A possibility is to split into a logical and a physical layer. The logical one contains driver functionality which remains consistent between same typed devices while the physical layer only takes care of device specific functionality[50]

3.1 Linux Driver Model

Driver Types

As already mentioned, drivers do most of all an abstraction of the communication with I/O peripherals. To do so, Linux provides more ways than widely known. The first one is via *direct hardware access* as *user-space driver*. As known from the previous sections are the most I/O respectively driver related operations privileged. But Linux offers a way for common user applications (userspace) to access hardware without a classical driver using this possibility. It is, above all, used for video drivers which are incorporated into the *X.Org display server*. In order to obtain the neccessary rights are two systemcalls needed: `iopl()` to change the privileged level and `ioperm()` to set the I/O port permissions. These calls can only be done by a privileged Linux user, the *root* user. Compared to other options to access peripheral hardware, this one has the disadvantage that interrupts are to available at all and the user software may run into issues with demand paging. So this way is maybe slower than a device driver in kernelspace, but for some tasks, like the already named *X.Org server* is it more meaningful 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/` are devices in those filesystems categorized in a structured way, according their types[17]. Linux differentiates drivers in the already known block devices, character devices and network devices, but internally are drivers structured in *subsystems* of similar device functionality like usb, network, bluetooth, gpio and many others[40].

As indicated above do *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 require not 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 the most cases assigned by the *udev* mechanism today. A closer look to it and its relation to the `/sys/` filesystem follows.

Character devices, no matter in which subsystem they are arranged, have in common that they are well represented as data streams. They provide only sequential access to their data and can be considered as file including the standard file operations[26]. The same applies to *block devices*. In contrast to character devices are block devices read and written only in multiples of their block-size. Linux enables devices of these 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 mount 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 different at all. 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 version, only a static integration was possible. That means all may 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 is it needed to recompile and reboot the kernel. Current Linux kernels allow additionally the dynamically loadable modules with optional parameters. Reload a driver during runtime without a reboot does not only saves 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 to exactly the same kernel module 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 a 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, beeing a part of the kernel's address space brings drivers into a special responsibility. There is no isolation between parts of the kernel, an errornous implemented driver may crash the entire system as a result. For the implementation of a driver, the fact of it should be used as built-in driver or as module has just little impact. It does require little or no changes on the source at all to switch between them as it is most of all a build configuration[26].

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

Driver Interfaces

As for device drivers in general need the ones in Linux functions to include themself into the system, 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 tiggered 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 behaviour a device, it is preferred to implement only the meaningful operations, leave the default behaviour for the others one and switch to a better suited interface for the device. Besides the standard I/O API, there are e.g. communication or multimedia specific APIs which suits better to devices of these kinds. Often, these interfaces are defined in Linux but built on basis of the `ioctl()` call which is technically a part of the file operationsi[40]. I/O control is a universal interface to define own, device specific commands. An `ioctl()` command is usually made from a made from a number and the type of optional arguments. However, the preferred way is to utilize the macros Linux provide to define beside the number and the arument types also the size of the transfered 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, this 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 needed to declared 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 is the communication with the user a 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, with different address spaces, different virtual memory addresses types and different rights. Addresses in the one address space are not neccessary meaningful in the other and additionally are user space buffers may 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 do the transition between the address spaces[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 direct access to kernel memory buffers which may also include memory regions of a device controller, by mapping it into the application's address space. Memory mapping affords 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 been accessed from kernelspace. Thus, they are not suitable for data exchange between kernel and userspace[26].

File operations are not the only option for an 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 generated during runtime as a virtual filesystem and spans a tree of device objects with the system bus on 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, like Peripheral Component Interconnect (PCI) or Universal Serial Bus (USB). Thus, the *udev* mechanism is invoked. It is a mechanism to create entries for devices in the */dev/* directory. Without udev, it would be neccessary 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 rather a common situation than an exception one driver instance has to manage more than one fitting 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 purpose. It must be *reentrant*, 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 neccessary step there, the allocation of a correct device number, is done by udev using information exported there. Besides the basically needed information which are exposed just by register the driver within sysfs is a driver 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, 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 accessable at all. The access permissions for a sysfs file entry are fine granular, based on *group permissions*. As 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 neccessary for drivers built as a module, but also allowed when compiling as a built-in driver. Thus, the most driver implementations does not require any code changes and the decision which variant is built depends only on a value in a configuration file. This additional functions do specific initializations which are only needed on modules. In common, the `init()` function itself and corresponding init data are specially marked to be discarded after initialization, while especially the `exit()` to clean up initializations 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 neigher the driver is initialized and ready nor is a device connected at this point. Figure 3.1 pictures these in a drivers sequence context. As the implementations of `init()` and `exit()` often only consists of registering the actual driver and thus abstractable boilerplate code, are they often replaced by a macro, e.g. `module_i2c_driver(<driver_struct_name>)` for an Inter Integrated Circuit (I2C) driver. Regardless of whether `init()` or a corresponding macro is used, the initialization consists mostly of publishing a driver structure to the kernel. In common this struct contains at least function pointers to neccessary driver entry points like `probe()` and `remove()` and to a sub-structure which covers 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 I2C typed[26], [40], [6]. Also, `probe()` is the first driver function that must be *reentrant*. It is called each time a matching device is detected and there should not be a artifical limit how much devices a driver can handle. Thus, the information needed for each distinct device should be stored in a private per-device datastructure. Allocate the memory for this structure and fill 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 an user wishes to unload the module. Only afterwards, the `exit()` function can be called. Within `exit()`, the device should be putted in a suitable condition prior to the initializations done in `probe()` are revoked[26], [40]. The `remove()` function is *reentrant*, too. It is called per device.

The right information needed to deregister device and the driver's entries for this

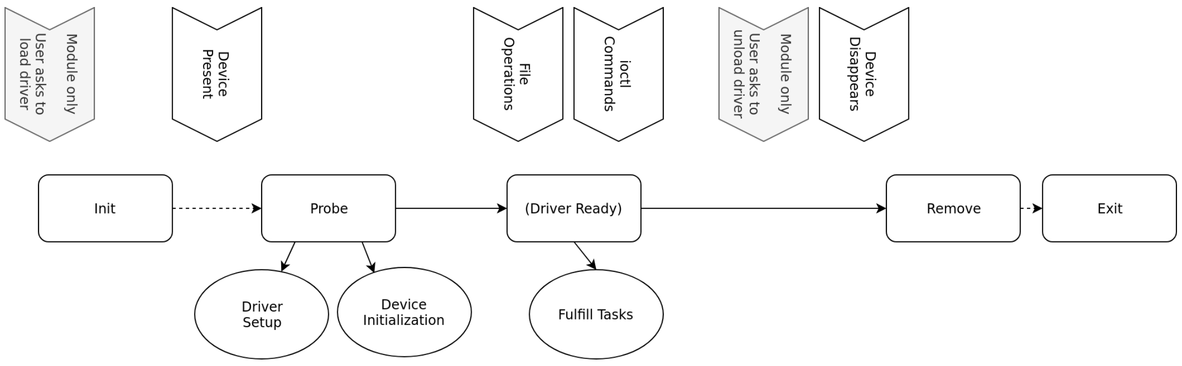


Figure 3.1: Simplified Lifecycle of a Linux Device Driver

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 an above 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 *reentrant*, 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, translate it in a command for the device controller and take care of the physical transmission to the device. Depending on the request, this may includes 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 converns 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 hided[35].

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 belong each part of the kernel including device drivers 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 an 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[35]. Thus, Zircon's idea differentiate from the textbook approach and group a number of related drivers together in so-called *device host* processes[35]. A driver itself is in Zircon compiled to a 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 cration 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 soften 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], [35]. Branches of this tree are represented by device host processes which consist of devices. At 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 seperate 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 future, this policy will may evolve to 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[35], [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 *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 discription itself, the binding program also contains pointers to the most neccessary 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 initializations. While its pedant 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]. Thus, it is not pictured in the simplified Zircon driver lifecycle in figure 3.2.

Similar to Linux' `probe()` function follows in Zircon the `bind()` method in a drivers life. It is invoked by the device coordinator how 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 or the device to succeed[**zircon-gettingstarted**]. 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 pictures 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 the fewest 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 never invoked currently because once a driver is loaded, it remains loaded for the lifetime of a device host process. Nevertheless, it should be implemented.

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()`. 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

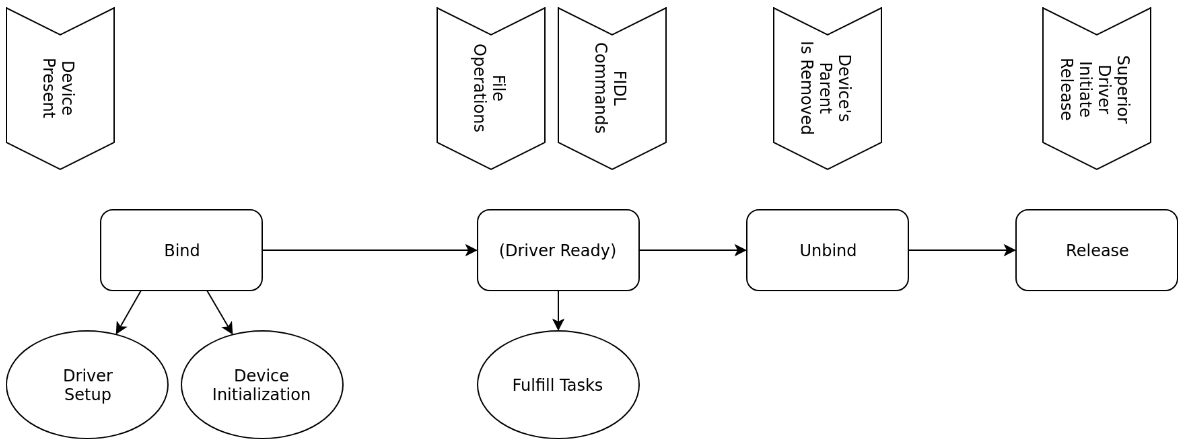


Figure 3.2: Simplified Lifecycle of a Zircon Device Driver

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 exemplarily tear down sequence on all children[15].

3.3 Test Setup

3.3.1 Hardware Issues

3.4 General Driver Concept

3.5 Linux

3.5.1 Driver Implementation

Interface

3.6 Zircon

3.6.1 Driver

C

C++

FIDL

3.6.2 PDev Driver

C

C++

FIDL

3.6.3 User Application

3.7 Evaluation

Chapter 4

Conclusion

Chapter 5

Outlook

Bibliography

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

[2] Christopher (cja) 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] Raghu Bharadwaj. *Mastering Linux Kernel development : a kernel developer's reference manual.* Birmingham, UK: Packt Publishing, 2017. ISBN: 978-1-78588-305-7.

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

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

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

[8] 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`.

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

[10] David Diamond and Linus Torvalds. *Linus Torvalds: Just For Fun. Wie ein Freak die Computerwelt revolutionierte. Die Biographie des Linux-Erfinders.* ngerman. Titel der amerikanischen Originalauflage: Just for Fun – The Story of an Accidental Revolutionary. HarperCollins, New York, 2001. Aus dem Amerikanischen von Doris Märtin. 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] Alan Holt. *Embedded operating systems : a practical approach*. London: Springer, 2014. ISBN: 978-1-4471-6602-3.

[23] 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.

[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] John Madieu. *Linux deivce drivers development : develop customized drivers for embedded Linux*. Birmingham, UK: Packt Publishing, 2017. ISBN: 978-1-78528-000-9.

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

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

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

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

[32] 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`.

[33] 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`.

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

[35] Bruce Mitchener. *Getting Started*. Oct. 2018. URL: `https://github.com/Allegra42/zircon/blob/i2c-grove-lcd/docs/ddk/getting_started.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. Aufl. 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. *Moderne Betriebssysteme.* Ed. by Herbert Bos. 4., aktualisierte Auflage. Always learning. Hallbergmoos: Pearson, 2016. ISBN: 978-3-86894-270-5.

[50] 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.

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

[52] Andrew S. Tanenbaum and Albert S. Woodhull. *Operating systems : design and implementation.* 3. ed. The MINIX book. Upper Saddle River, NJ: Pearson Prentice-Hall, 2006. ISBN: 978-0-13-142938-3; 0-13-142938-8; 0-13-142987-6.

[53] 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.

[54] 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	62
API	Application Programming Interface	11
BTI	Bus Transaction Initiator	52
CBS	Constant Bandwith Server	41
CFS	Completely Fair Scheduler	40
CPSR	Current Program Status Register	7
CPU	Central Processing Unit	1
DMA	Direct Memory Access	47
DPRR	Dynamic Priority Round Robin	
DSO	Dynamic Shared Object	62
EDF	Earliest Deadline First	
ELF	Executable and Linking Format	15
FCFS	First Come First Serve	37
FIDL	Fuchsia Interface Definition Language	62
FIFO	First In First Out	30
futex	fast user-space mutex	26
GCC	GNU Compiler Collection	12
I2C	Inter Integrated Circuit	60
IDL	Interface Definition Language	13
I/O	Input/Output	1
IP	Internet Protocol	32
IPC	Inter-Process Communication	9
LWP	Light-Weight Process	20
MMU	Memory Management Unit	45
PC	Program Counter	18
PCB	Process Control Block	19
PCI	Peripheral Component Interconnect	59

PFN	Page Frame Number	48
PID	Process Identifier	21
POSIX	Portable Operating System Interface for UniX	1
PSW	Program Status Word	7
RAM	Random Access Memory	43
RPC	Remote Procedure Call	62
RR	Round Robin	
SMP	Symmetric Multiprocessing	8
SRTF	Shortest Remaining Time First	
SSD	Solid State Drive	43
TAS	Test and Set	25
TCB	Thread Control Block	20
TCP	Transmission Control Protocol	32
TLB	Translation Lookaside Buffer	45
TSL	Test and Set Lock	25
UDP	User Datagram Protocol	32
UI	User Interface	36
USB	Universal Serial Bus	59
vDSO	virtual Dynamic Shared Object	15
VMAR	Virtual Memory Address Region	23
VMO	Virtual Memory Object	33
XCHG	Exchange	25

List of Listings

List of Figures

List of Tables