Diplomarbeit

A Policy-Free System-Call Layer for the Hedron Microhypervisor

Philipp Schuster

Matr-Nr.: 4506091

01. April 2022

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

1. Hochschulbetreuer:	Dr.-Ing. Nils Asmussen
2. Hochschulbetreuer:	Prof. Dr.-Ing. Horst Schirmeier
Betrieblicher Betreuer:	Dipl.-Inf. Julian Stecklina

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 01. April 2022

Philipp Schuster

Danksagung

Ich möchte mich recht herzlich bei allen Menschen bedanken, die mich beim Erstellen dieser Arbeit in jedweder Form unterstützt haben. Dieser Dank gilt insbesondere der ganzen Cyberus Technology GmbH und meinem Vorgesetzten Tor Lund-Larsen, die mir das Thema dieser Diplomarbeit im Wesentlichen ermöglicht haben. Ganz besonderer Dank darüber hinaus geht an meinen betrieblichen Betreuer Julian Stecklina. Außerdem möchte ich mich bei meinem universitären Betreuer Dr. Nils Asmussen für seine gute Unterstützung bedanken. Obendrein gilt mein Dank meinen Freunden, meiner WG, meiner Familie, meinen Kommilitonen und allen sonstigen Menschen, die meine Studentenzeit zur bisher besten Zeit meines Lebens gemacht haben.

Aufgabe

Gegenstand dieser Arbeit ist das Design und die Implementierung eines generischen Mechanismus, der die Ausführung von Hybrid- und unmodifizierten Fremdanwendungen unter dem Mikrokern Hedron, einem Fork von NOVA, ermöglicht. Fremdanwendungen sind Programme, die gegen eine für Hedron fremde Systemaufrufschicht entwickelt wurden. Hybride Anwendungen beinhalten darüber hinaus zusätzlich Hedron-native Systemaufrufe, die sich von den fremden Systemaufrufen unterscheiden. Unmodifiziert bedeutet, dass die für das Fremdsystem kompilierten Anwendungen nicht nachträglich für Hedron angepasst werden müssen und somit eine Binärkompatibilität besteht.

Es soll untersucht werden, inwieweit eine erhöhte Produktivität beim Entwickeln neuer Software für Hedron erreicht werden kann, wenn man dafür beispielsweise die existierenden Toolchains für Linux benutzt. Das soll dem Mehraufwand durch eine Toolchainanpassung für das Entwickeln von Software für Hedron gegenübergestellt werden. Die generelle Funktionstüchtigkeit des Systems soll mit der erfolgreichen Ausführung einfacher unter Hedron laufender Linux-Programme, wie einem Dateisystemmikrobenchmark, aufgezeigt werden. Zudem sollen die Auswirkungen auf die Performance der fremden Systemaufrufe untersucht werden. Die Lösung soll entsprechend der Mikrokernphilosophie keine Policy im Kern implementieren. Im Kern wird nur der minimal notwendige Mechanismus realisiert, alles andere liegt in der Verantwortung der Laufzeitumgebung im User Space.

Kurzfassung

Das Entwickeln für Nicht-Standard-Targets, wie den Mikrokern Hedron, ist mit einer geringeren Produktivität verbunden, da typische Toolchains angepasst werden müssen. Beispielsweise ist die Standardbibliothek auf dem Zielsystem nicht vorhanden oder es wird eine spezielle Version benötigt. Das Entwickeln für etablierte Standard-Targets ohne Anpassung einer Toolchain ist hingegen einer großen Zahl an Entwickler:innen bekannt. Von einem Standard-Target abweichende Toolchains müssen pro Softwareprojekt angepasst und gepflegt werden. Die Erfahrung zeigt, dass dieser Prozess viele Ressourcen bindet.

In dieser Arbeit werden Änderungen an Hedron sowie ein zugehöriges Laufzeitsystem vorgestellt. Im Zusammenspiel ermöglichen diese Komponenten die parallele Ausführung von Hedron-nativen Anwendungen und unmodifizierten Fremdanwendungen am Beispiel von Linux-Programmen. Das ermöglicht, die etablierten Toolchains wiederzuverwenden und neue Software für Hedron mit diesen zu entwickeln. Ferner werden auch Linux-Programme unterstützt, die zusätzlich selbst Hedron-native Systemaufrufe enthalten (Hybridanwendungen). Daraus resultiert die Möglichkeit, zusätzlich direkt mit Schnittstellen der Laufzeitumgebung zu kommunizieren.

Fremdanwendungen laufen dabei als First-Class Citizens, also Seite an Seite mit Hedron-nativen Anwendungen. Das ist im Kontrast zu Lösungen wie *L4Linux*, die Binärkompatibilität über einen (para-)virtualisierten Gast ermöglichen. In dieser Arbeit wird Binärkompatibilität über eine Emulation der Fremdschnittstelle im User Space erreicht. Dies ermöglicht eine enge, funktionale und performante Integration in das bestehende Laufzeitsystem, wie den Zugriff auf existierende Datei- und Speicherverwaltungsdienste.

Ich zeige auf, dass die Lösung eine hohe Produktivität von Entwickler:innen ermöglicht und die Kosten der Emulationsschicht bei beispielsweise großem Datendurchsatz im Dateisystem mit Linux mithalten können.

Task

The subject of this work is the design and implementation of a generic mechanism that allows the execution of hybrid as well as unmodified foreign applications under the microkernel Hedron, a fork of NOVA. Foreign applications are programs which were developed against a system-call layer that is incompatible to Hedron's. Hybrid applications are foreign applications that additionally contain Hedron-native system calls, which differ from the foreign system calls. Unmodified means that the applications compiled for the foreign system do not need to be modified for Hedron, thus binary compatibility is provided.

The aim is to investigate to what extent productivity increases in the development of new software for Hedron if, for example, the existing toolchains for Linux can be used. This should be compared to the additional effort caused by a customized toolchain that is required for a non-standard target, such as Hedron. The general functionality of the system should be demonstrated with the successful execution of simple Linux programs running under Hedron, such as a file-system microbenchmark. The impact on the performance of foreign system calls needs to be investigated. In keeping with the microkernel philosophy, the solution must not implement any policy in the kernel. Only the minimum necessary mechanism is implemented in the kernel, everything else is the responsibility of the runtime environment in user space.

Abstract

Developing for non-standard targets, such as the Hedron microkernel, is associated with lower productivity, since typical toolchains have to be adapted. For example, the standard library is not available on the target system or a special version is required. Developing for established standard targets without adapting a toolchain, on the other hand, is known to a large number of developers. Toolchains that deviate from a standard target must be adapted and maintained for each software project. Experience shows that this process ties up many resources.

This paper presents modifications to Hedron and an associated runtime system. Together, these components enable the parallel execution of Hedron-native applications and unmodified foreign applications using Linux programs as an example. This allows to reuse the established toolchains and developing new software for Hedron with them. Furthermore, Linux programs are supported that contain additionally Hedron-native system calls (hybrid applications). This makes it possible to communicate directly with interfaces of Hedron's runtime environment.

Foreign applications run as first-class citizens, i.e., side-by-side with Hedron-native applications. This is in contrast to solutions like *L4Linux* which enable binary compatibility via a (para-)virtualized guest. In this work, binary compatibility is reached via an emulation of the foreign interface in user space. This allows a tight, functional, and performant integration into the existing runtime system, such as access to existing file and memory management services.

I show that the solution enables high developer productivity and that the cost of the emulation layer can keep up with Linux, for example, with large file-system throughput.

Contents

List of Figures

List of Tables

List of Listings

1 Introduction

The availability of software is a key factor for the success of an Operating System (OS). When a new kernel with a new runtime system is created, it is usually difficult to find developers for it, because the majority is always going to use what they are familiar with. Writing software is not just about writing code but about having a good mental model regarding the system, a situational awareness what tooling is applicable, and an understanding about how problems can be debugged. Undeniably, Windows and *UNIX*-like OSs (*Android*, *macOS*, *Ubuntu*, ...) are what the majority is used to and the target platforms for all known and convenient tooling. This tooling includes Integrated Development Environments (IDEs), compilers, build systems, and debuggers. For example, IDEs offer auto-completion for functions from the standard library and quick navigation to library functions. This convenience usually goes at least partially away with a custom build setup for a non-standard runtime environment. Newer languages, such as Rust with its ecosystem, make it easier to build for non-standard targets. Especially the configuration and set-up process of the build system is simplified. Still, the convenience is worse than when developing for standard targets. Furthermore, the amount of usable libraries (without further modification) is limited for non-standard targets. This thesis focuses on improving the developer experience for application programming for *Hedron* by giving developers the ability to use convenient *Linux* tooling paired with Hedron-specific functionality. Furthermore, it will lower the learning curve for new developers in their on-boarding process.

In this thesis, I introduce changes to Hedron and a corresponding new runtime environment for it. Both in combination allow the execution of applications compiled for other operating systems. As a proof of concept, I implemented the necessary functionality in user space for a correct execution of simple Linux programs. Although not implemented, the mechanisms described in this thesis will work for *Microsoft Windows* and macOS programs as well because it makes no specific assumptions.

In this chapter, I introduce the reader to goals and non-goals of this thesis. In Chapter 2 on page 25, I discuss relevant technical background knowledge. Chapter 3 on page 39 presents my design for a policy-free system-call layer for Hedron. In Chapter 4 on page 57, I focus on concrete implementation decisions. The evaluation in Chapter 5 on page 67 discusses the benefits, limitations, and possible performance impacts of my work. In Chapter 6 on page 83, I discuss related work and how certain solutions compare to my work. Chapter 7 on page 89 sketches possible

future improvements, and Chapter 8 on page 91 finally summarizes all findings. The appendix in Chapter 9 on page 93 shows further information, such as additional interesting implementation challenges.

Since Hedron uses abstractions that differ from typical UNIX abstractions, I especially recommend checking the corresponding Section 2.3.3 on page 29 in the background chapter, where all important Hedron abstractions and kernel objects are explained in more detail. In addition, it may be helpful to look at the glossary on Page 101 for a brief overview of important terms.

My work uses the term *foreign application* to describe applications that were developed for another OS, i.e., with a non-Hedron system-call Application Programming Interface (API), such as Linux. A "Hello World"-program compiled for Linux is an example of a foreign application. The term *hybrid application* will be used to refer to foreign applications that contain Hedron-native and foreign system calls side-by-side inside the same binary. When I talk about an application that is executing under Hedron, I mean the application executes under Hedron and the runtime system presented in this thesis.

All technical discussions and measurements of this work focus on the `x86_64`-architecture since Hedron is only used on that platform. The source code of my contributions can be found in the appendix in Section 9.1 on page 93.

1.1 Motivation

For a custom and not widely known OS, new developers need to learn about the kernel and its API first before starting to write software for it. Taking Linux as example, developers are familiar with the *libc* standard library. Thus, it is convenient if developers can reuse this interface to develop software for Hedron.

Cyberus Technology[1] has identified a need for a pragmatic solution to execute Linux applications under Hedron. The focus is not necessarily on making existing software usable, but primarily on writing new software with the same experience as developing for Linux. Specifically, established and well-known toolchains become feasible to be used for new Hedron applications. The result is a great developer experience with rapid progress and high productivity. Hybrid applications are a way to achieve this because typical workloads can be handled with known calls to libc whereas special functionality of Hedron or the runtime system can be triggered with an additional library.

[1] `https://www.cyberus-technology.de`

1.2 Goals

For this work I have set the following important objectives.

1. A generic mechanism is required to execute unmodified binaries for another OS under Hedron that allows a binary compatibility. Unmodified means that the executable files are supported in the state as they are produced by their default toolchain. These applications may be hybrid, i.e., Linux applications with some Hedron-native system calls side by side to regular ones. This combines the convenient and well-known experience of developing for existing software ecosystems with the easy integration of Hedron-related functionality into newly written programs.

2. The mechanism must enable tight integration of foreign programs into the existing runtime environment. For example, if a foreign application wants to open a file with the UNIX system call `open()`, this action should be handled by the normal file-system functionality of the runtime system.

3. The amount of new code in user space to emulate a relevant portion of foreign system calls should be within reasonable limits. If it requires millions of lines of code, hence a nearly full reimplementation, solutions such as *L4Linux* are a better option [18].

4. The changes to Hedron should be minimal and not introduce policies inside the kernel. It should be permitted to have competing implementations in user space. The microkernel should still be a microkernel afterwards.

5. Developer productivity and developer experience for a new (hybrid) Hedron application must be similar to developing a new program for example for Linux. Typical IDEs and debugging tools must be applicable.

To validate the successful execution of simple foreign applications, static Linux executables produced from programs written in *C*, *Rust*, and *Zig* will be tested. All further functionality, i.e., supporting more system calls for complex programs, is a matter of diligence, whereas the necessary basic mechanism around for the system-call intercepting and handling stays the same. To verify the functionality of the hybrid part, the “Hello World”-program written in Rust will be slightly modified to include Hedron-native system calls in addition.

With the solution I present in this paper, which meets the goals I set, developers will be able to write a new Hedron program using existing Linux tooling. This means, typical setups and toolchains to build applications are applicable, *valgrind* can be used to find memory leaks, and typical unit testing frameworks will integrate easily. Additionally, the program may contain code to act as a Virtual Machine (VM) under Hedron when it has full access to all Hedron-native system calls. There is no need to set up and support custom toolchains. Note that unit testing and memory checkers, such as valgrind, can only cover non-Hedron functionality. The test might needs to mock Hedron-calls when tests are executed under pure Linux.

1.3 Scope of This Thesis

It is not a goal that foreign applications will see their typical, complete runtime environment including specific services of the given foreign OS. Hence, taking Linux as example, a foreign Linux application will not see interfaces to runtime (sub-)systems, such as *ALSA*, *X11*, or *Wayland*. It is also not a goal to enable desktop environments with a rich Graphical User Interface (GUI).

Cyberus Technology's usage of Hedron was never meant for rich, multimedia applications and also will not cover this domain in the near future with its corresponding runtime environment. Thus, with the implementation in Chapter 4 on page 57, foreign applications will not be able to draw to the screen or playback audio data. Although, the generic mechanism proposed by my work is not in contrast to such functionality and is a step forward to achieve that.

To implement an emulation layer for foreign system calls in user space, I need a basic runtime system that provides basic functionalities such as memory allocations, logging facilities, and basic process management. These topics are well understood and already established in research and products [11, 19]. Hence, the discussion in this work will only focus on the relevant parts of the runtime system that enable the major contributions of my work. It is out of scope to discuss how a runtime system should be designed for Hedron or how maximum performance can be reached.

Finally, this work proves the functionality of the mechanism by successfully executing static Linux applications. Although it is technically possible to run static and dynamic applications with my proposed mechanism, supporting dynamic applications is out of scope.

1.4 Code Examples

Section 9.2 on page 93 shows multiple code examples that give the reader an idea about what kind of foreign and hybrid applications can be executed with the introduced foreign system-call mechanism for Hedron in Chapter 3 on page 39 and the corresponding policies implemented in user space described in Chapter 4 on page 57. As my current implementation only supports static Linux binaries, the code examples must be linked statically (for example against the *musl* library). However, this is a limitation of my runtime environment and not a limitation of the proposed policy-free system-call layer.

2 Technical Background

This section provides an overview of all relevant topics and areas covered in this thesis. The thesis assumes that the reader is familiar with the fundamentals of programming and basic OS concepts [38, 25]. Nevertheless, it gives a short introduction to a few generally known terms to recapitulate important facts briefly. For a quick overview of relevant terms, please also refer to the glossary on Page 101.

2.1 Operating System and Runtime Environment

An OS consists of the kernel and necessary runtime services running in user space. The kernel forms the runtime environment together with all relevant user-space components that provide the expected functionality of the system. There is no clear definition for relevant user-space components as they depend on the system, or more precisely, on the use case. A typical Microsoft Windows user for example sees the complete desktop environment with all pre-installed system applications as OS whereas a shell-only user of a headless Linux distribution needs much less functionality and still considers the system as a full OS. One way to classify OSs from a technical perspective is to look how they distribute responsibility between kernel space and user space.

2.2 Different Kernel Architectures

In the real world multiple designs for kernels architectures exist. Some OSs include drivers primarily inside the kernel whereas others include drivers primarily in user space. There are also mixed versions. Each has advantages and disadvantages that are discussed in the following.

2.2.1 Monolithic Kernels

Linux-based distributions and Microsoft Windows are among the most used OSs in the world, and both combined have a total dominance in consumer- and server-markets [40]. Although various systems share a monolithic design, there are differences at multiple levels, for example, in the handling of files and devices.

These systems are popular because the majority of software exists for them. Users tend to continue using systems they understand instead of switching to new ones,

although they may have better safety and security guarantees. Because of that fact, new software is developed for the established monolithic systems. It becomes hard to break out of the loop to establish new system architectures that are interesting to a significant amount of users.

Both kernels, Linux and *Windows NT*, grew over the last decades. Although code bases are improved over time, bugs might exist for over 15 years [15]. This applies for all software and not only monolithic kernels but the attack surface of each (silent[1]) bug is significantly more dangerous in kernel space. Although many monolithic kernels have support for user-space drivers, the default is still to add new and large drivers into the kernel code bases [41, 1]. Hence, these kernels still suffer from several security and safety issues caused by their monolithic design.

2.2.2 Microkernels

A microkernel follows the concept that as little code as possible runs in kernel space with a few reasonable exceptions.

> *"A concept is tolerated inside the microkernel only if moving it outside the kernel, i.e., permitting competing implementations, would prevent the implementation of the system's required functionality."*
> – Jochen Liedtke [23]

Jochen Liedtke said a microkernel provides only mechanisms but no policies. However, many existing microkernels differ in functionality they provide inside kernel space. For example, scheduling is still part of most L4 microkernels because no efficient user-space solution exists for that [8]. Thus, Liedtke's famous quote is a strong hint but no absolute rule. Different implementations take different design decisions.

For L4 microkernels, the following applies: The code of the kernel is only related to bootstrapping the system, managing memory and address spaces, providing an Inter-process Communication (IPC) mechanism, and a small set of drivers [23]. All other functionality runs in user space.

The small amount of code in kernel space reduces a possible attack surface significantly.

2.2.3 Comparison of Microkernels and Monolithic Kernels

In Figure 2.1 on the facing page, we can see a comparison of how an application requests system resources, such as memory, under different operating system architectures. On the left a monolithic system is shown. The application performs a system call and receives the result from that. This process is quick as the memory subsystem lives inside the kernel. Jumping into the kernel only require a lightweight context switch, i.e., no full address-space switch. The kernel usually lives in the

[1] A bug that can live in the code base for years without being discovered.

same address space as a user application, which makes the context switch to the kernel faster. This is only partially true if several mitigation techniques for vulnerabilities such as Meltdown are active. For example, techniques such as kernel page table isolation will add further latency but new smart mitigation strategies can keep the overhead of mitigations small [9].

On the right side of the figure an application that requests a system resource under a microkernel-based system is shown. All important subsystems, such as memory management, run in user space. The microkernel does not answer a request for a system resource directly but only offers a IPC mechanisms to a corresponding user-space destination. Hence, the costs of requesting system resources are usually higher compared to monolithic designs. To request a system resource, it does not only take a context switch into the kernel, but it takes two expensive context switches between processes. They are expensive because the context switches include an address-space switch.

The upper half of the figure (red) displays the kernel space, i.e., code running in privileged mode. The lower half (green) shows the user space. We can see that the amount of code running in privileged mode (kernel space) on a microkernel-based system is significantly smaller compared to a monolithic system.

In both cases, the requests must pass through the kernel as it is mandatory that the kernel checks for the correct permissions. Permissions can be modeled through capabilities, as explained in Section 2.3.1 on the next page.

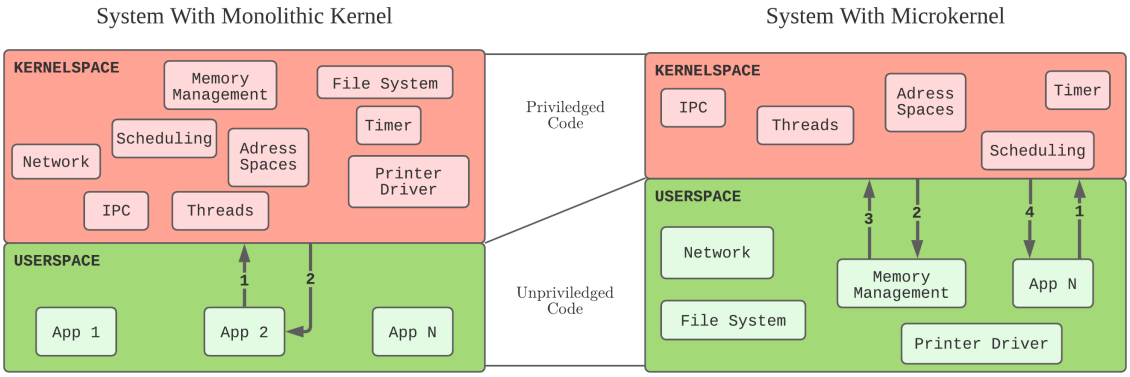

Figure 2.1: Comparison of a request to system services (such as memory) on a microkernel-based system (on the right) vs. a system with a monolithic kernel (on the left). On the left, the memory management system is in kernel space. An application can communicate with it via a single system call. On the right, an application needs to perform an IPC request guarded by the kernel to another user-space component, which implements the requested functionality.

2.3 Hedron Microhypervisor

The Hedron microhypervisor is a fork of *NOVA* maintained by Cyberus Technology. NOVA was publicly introduced in 2010 [36] and integrates itself into the long history of *L4* microkernels [36, 8]. The last shared commit of Hedron with NOVA dates back to 2015 [12]. Both projects share the same basic principles and large parts of the code base. The NOVA paper uses the terms microhypervisor and hypercall [36]. For consistency, I use the terms microkernel for microhypervisor and system call for hypercall.

2.3.1 Capabilities

Hedron uses capabilities to enforce security, manage system resources, and enable the principle of the least privilege. A capability is the right to use, alter, obtain access, or revoke access a certain system resource [32, 20].

Capability selectors are used to refer to capabilities from user space. They are an integer and similar to file descriptors in UNIX. Therefore, the application must keep knowledge about what capability selector belongs to which kernel object. This is similar to the following C example targeting an UNIX interface:

```
// FD is similar to a capability selector:  keep track of it
int config_file_fd = open("/foo/bar.txt", O_RDONLY)
```

2.3.2 Functionality Inside Kernel Space

As mentioned in Section 2.2.2 on page 26, microkernels aim for as little code in privileged mode as possible. For example, NOVA had only 9,000 Source Lines of Code (SLOC) when it was released [36].

However, the L4 history proved that a certain amount of functionality must be in the kernel for sensible functionality and performance. Inside kernel space, Hedron provides IPC primitives, address spaces, hardware-assisted virtualization, resource management via capabilities, delivery of interrupts, scheduling, and timers. Furthermore, Hedron is capable of bootstrapping the relevant parts of the underlying hardware, such as initializing and starting all CPUs to take full benefit of all cores from a multicore processor. It understands the page table mechanism as well as the Local Advanced Programmable Interrupt Controller (LAPIC) of each core. In short, it initializes the x86_64 platform. There is also a small portion of code to initially extract and dispatch the roottask (an Executable and Linking Format (ELF)-file), which Hedron expects as a *GRUB* boot module during boot. The role of the roottask is explained in Section 2.4 on page 32.

2.3.3 Kernel Objects

A kernel object is a data structure, usually with a mutable state, that the kernel manages to fulfill the promised functionality. It is the base for certain programming primitives of the system. Hedron knows five distinct kernel objects that are explained in Table 2.1. They correspond to the kernel objects of NOVA [36]. A Protection Domain (PD) in combination with Execution Contexts (ECs) is similar to what we know as process in UNIX. As it can have multiple ECs, a PD with multiple ECs can be understood as a process with multiple threads. The PD is a resource container for other kernel objects and memory capabilities. It holds the address space of a process.

ECs split into the two types: local ECs and global ECs. While global ECs execute in their own time slice provided by their dedicated Scheduling Context (SC), local ECs wait for a request until they execute and use the time slice of the caller.

One typical scenario for the usage of kernel objects is the following: If a PD wants to export functionality, like an interface to allocate memory, it can create a Portal (PT) that is bound to one of its local EC. The capability to that PT then needs to be delegated to the target PD that wants to access this service. When a PT (or another object) is granted to a PD, code running in that PD can access it through the corresponding capability selector. Through a `call` system call, the delegated PT can be reached. Hedron ensures that a caller has a capability to the specified PT to perform IPC.

Kernel Object	**Description**
Protection Domain (PD)	A resource container that holds an address space and capabilities to other kernel objects.
Execution Context (EC)	Bundles the execution state, such as instruction pointer and stack pointer, and belongs to a PD. There are local ECs and global ECs.
Scheduling Context (SC)	Gives a global EC a time slice.
Semaphore (SM)	Object used for synchronization of producer-consumer scenarios, such as interrupts.
Portal (PT)	An IPC endpoint bound to a local EC.

Table 2.1: Overview of Hedron's kernel objects.

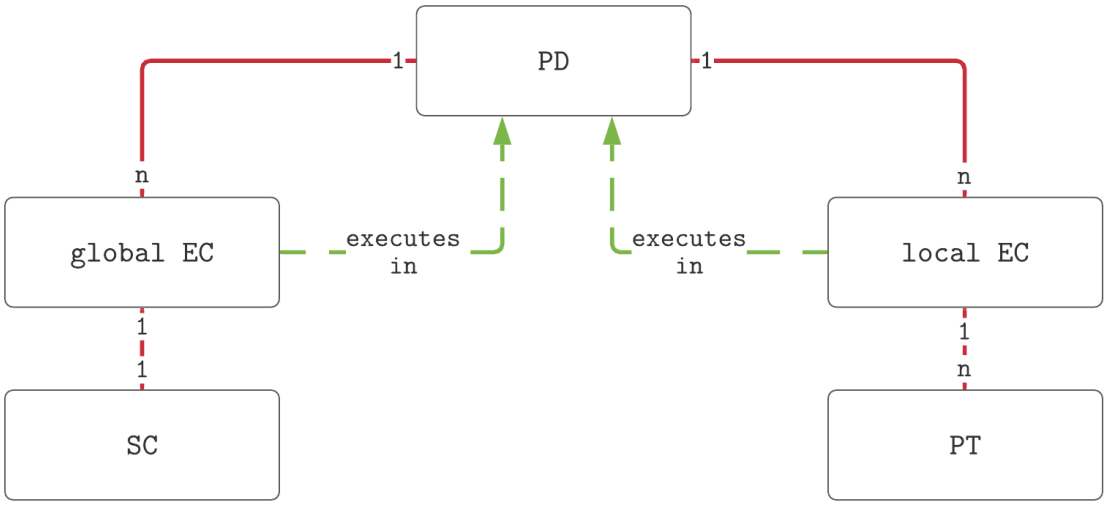

Figure 2.2: Typical relation between a Protection Domain, global Execution Contexts, and local Execution Contexts to run programs. The left side shows that a PD can have one or multiple global ECs. Each global EC needs one SC to receive a time slice. The left side shows what is similar to a process in UNIX with multiple threads. The right side shows the role of PTs. Each PT belongs to one local EC in which it executes. The figure only shows a subset of potential relations.

Figure 2.2 shows a subset of possible relations between those kernel objects. It shows the typical relation to enable a multithreaded program that also exports functionality via PTs. It shows how global and local ECs are used and that they all run inside the address space of their corresponding PD. The figure omits that a PD can have capabilities to other kernel objects.

All ECs live and execute inside the address space of the owning PD. A global EC needs a SC to run. A local EC can only run if a PT that is bound to it is called. As local ECs uses the time slice from the caller, only global ECs can initiate an IPC call to PTs. Nevertheless, local ECs can trigger further IPC calls, as long as the time slice of a caller is available. This donation of time slices also prevents priority inversion scheduling problems [36, 31].

Semaphores (SMs) are used for synchronization of producer-consumer scenarios, such as interrupts. A PD may use an arbitrary number of SM objects. The vCPU is a special form of an EC used to run operating systems in virtualized contexts. It offers software an interface to run under Hedron from a virtualized context with the downside of much higher costs when it communicates to Hedron (VM exits). The vCPU is not relevant for this work and is not discussed further. Details about it can be found in the original NOVA paper [36].

2.3.4 IPC and the UTCB

Under Hedron, processes (or to be specific: ECs) communicate via memory pages. For IPC, each EC has its own memory page called the User Thread Control Block (UTCB). All UTCBs are always mapped in Hedron. During IPC, Hedron copies data from the UTCB of the caller to the UTCB to the callee. Thus, Hedron uses kernel memory[2] IPC instead of register based IPC as other L4 microkernels [8].

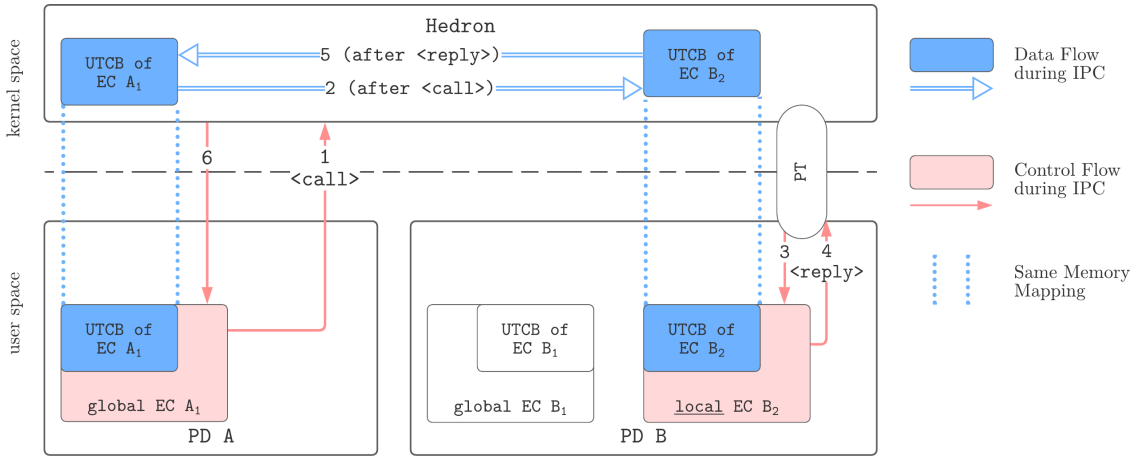

Figure 2.3: The global EC A_1 of PD A shown on the bottom left wants to communicate with a service provided by PD B. It copies relevant payload into its UTCB and performs a `call` system call to the target PT [STEP 1]. Hedron copies all payload from the UTCB of the caller to the UTCB of the callee [STEP 2]. Afterwards, control flow continues on the right side in PD B [STEP 3]. When the handler is done, it performs a `reply` system call [STEP 4]. Any result stored in the UTCB of local EC B_2 is transferred back to the UTCB of the original caller [STEP 5] and control flow continues in EC A_1 [STEP 6]. The blue parts inside the figure show all data flow operations whereas the red parts represent control-flow operations.

Figure 2.3 gives a detailed view of the role of the UTCB in IPC under Hedron. It shows how a caller on the left calls a service on the right and how control flow and data flow work. Additionally, the right side shows the global EC B_1 to demonstrate that each EC has its own UTCB. Thus, there is no shared UTCB between two ECs.

Exception IPC

Hedron delivers exceptions, such as invalid opcode (`0x6`) or page fault (`0xe`), by using the existing PT-based IPC mechanism. It uses a well-defined layout ("UTCB exception layout") to store the failed EC's CPU state, such as general purpose

[2]The message buffer is always mapped in the kernel and the kernel transfers data from buffer A to buffer B during IPC.

registers, into the UTCB of the corresponding exception PT. If the user-space handler receives an exception, the handler finds the original CPU state of the caller in its UTCB and start handling the exception. The Message Transfer Descriptor (MTD), which is assigned to each PT, specifies what information Hedron should store inside the UTCB exception layout. The MTD is a bitfield that tells whether general purpose registers, the instruction pointer, and other information about the CPU state should be taken into account in that process. When the exception IPC is replied, the (maybe from the userland altered) MTD inside the UTCB defines what fields from the UTCB should be put into the CPU state of the caller that originally triggered the exception. For example, this way the stack pointer can be altered.

2.4 Roottask and Runtime Environment

A roottask is the initial software component in a microkernel-based system that runs in user space. It can also be called the init process. The difference to the init process on monolithic systems, such as *systemd*[3], is that the roottask also decides how all drivers are bootstrapped and how they are exposed to the runtime environment.

The microkernel hands over control to it when the mandatory setup of the kernel successfully finishes. The roottask is bound by the kernel's limits and API, but apart from that it can create and enforce various policies and conventions for the runtime system. A runtime system includes all important services and IPC endpoints inside the roottask and other processes. Examples are a central logging gate (stdout), an allocator service, or a file-system service.

2.5 Application Binary Interface (ABI)

An Application Binary Interface (ABI) is a special form of an API that is relevant in low level system programming contexts. The following quote gives a good definition:

> *"Whereas an API defines a source interface, an ABI defines the low-level binary interface between two or more pieces of software on a particular architecture. It defines how an application interacts with itself, how an application interacts with the kernel, and how an application interacts with libraries. An ABI ensures binary compatibility, guaranteeing that a piece of object code will function on any system with the same ABI, without requiring recompilation."* – From the book: *Linux System Programming* [24]

[3] `https://www.freedesktop.org/wiki/Software/systemd/`

There exist a variety of ABIs in the world of software, such as the *System V ABI* [37]. It specifies how parameters are passed to functions on several platforms. As example, on x86_64 the first function argument usually is passed through register `rdi`.

2.6 System Call ABI

The system-call ABI describes the way values are passed to the kernel from user space. On `x86_64` system calls are done with the `syscall` or the `sysenter` instruction, which makes a fast transition to Ring 0, hence to kernel space. Due to legacy reasons, x86_64 also supports system calls via an `int 0x80`-interrupt. This is not covered in this thesis.

General purpose registers, such as `rax`, `rdi`, and `rsi`, retain their state during a system call, although there are some hardware limitations. For example, the `syscall` instruction saves `rip + 2` in `rcx`, i.e., the return address when the system call finishes, and *RFLAGS* in `r11`.

It is an implementation detail of the OS to create a calling convention that specifies what value is expected in what register. Hence, each kernel has its own specific ABI. Table 2.2 shows the system-call ABIs of a small selection of well-known kernels.

ABI	**Syscall Num**	**Arg 0**	**Arg 1**	**Arg 2**	**Arg 3**	**Arg 4**	**Arg 5**
Linux	`rax`	`rdi`	`rsi`	`rdx`	`r10`	`r8`	`r9`
Hedron	`rdi[8..0]`	`rdi[63..9]`	`rsi`	`rdx`	`rax`	`r8`	-
Xen	`rax`	`rdi`	`rsi`	`rdx`	`r10`	`r8`	`r9`
XNU	`rax`	`rdi`	`rsi`	`rdx`	`rcx`	`r8`	`r9`

Table 2.2: Overview of common system-call ABIs on x86_64. There are similarities between all listed systems. Except for Hedron (and NOVA), all use `rax` for the system call number. Furthermore, Hedron has one less parameter, than the others.

A kernel can use any general purpose register to report success or output parameters, as long as the register is not modified by the hardware on the transition back to user space. UNIX-like systems usually return the system call result in `rax`. NOVA and Hedron use `rdi` to store the main result and for some system calls `rsi` to store a second output value.

2.7 How Linux Runs Binaries

This section gives a short overview of relevant aspects about how Linux runs applications. This covers topics such as relevant initial data passed to applications through the stack and how signals are delivered.

When Linux starts an application, it parses the ELF-file and loads all *LOAD-segments* into memory with the proper page-table rights. It needs to set up the stack and to set the instruction pointer to the entry address specified in the ELF-file.

If the application starts running, several actions happen before the `main`-symbol is called. The entry point of the application does not point to `main()`, but to the `_start` symbol. This code is provided by the libc and performs initialization steps before it eventually calls `main`. The initialization steps rely on a specific initial stack layout from Linux that needs to be provided.

2.7.1 Initial Linux Stack Layout

When the CPU starts executing a freshly started Linux-application, the stack register `rsp` points to the bottom address of a data structure called the initial Linux stack layout. The layout is a few hundred to a few thousand bytes long and contains the argument vector (`argv`), the environment variables (`envp`), and the Auxiliary Vector (AuxV). The libc will fetch the pointers to `argv` and `envp` from the data structure and pass the information as arguments to `main()` eventually.

`argc`, `argv`, and `envp`

`argv` and `envp` are `null`-terminated arrays of pointers to C-strings. Additionally, the C-standard requires passing `argc` as first argument to `main`. This information is also in the initial stack layout as dedicated entry. The pointers in the arrays `argv` and `envp` point into higher addresses of the same initial stack layout, as Figure 2.4 on the facing page shows.

Auxiliary Vector

The AuxV is a data structure inside the initial stack layout that holds information about the ELF-file and about the environment, such as the platform and hardware capabilities. Implementations of libc may use information provided by the vector for dynamic linking or to retrieve information about the platform on which it is running.

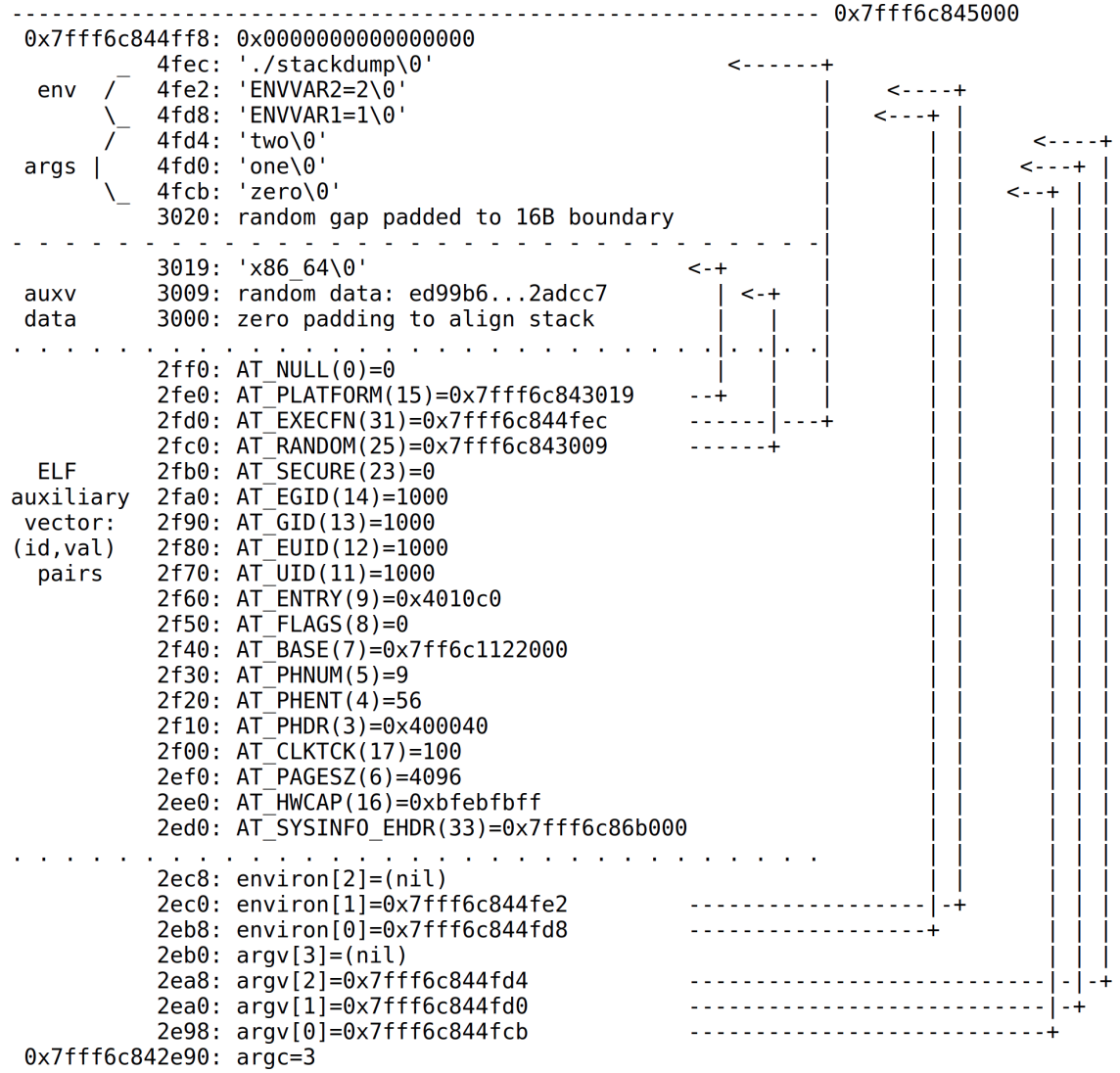

Figure 2.4: Example of the initial Linux stack layout for a Linux application. At the beginning of the execution of a just started program, the stack pointer points to the bottom of the structure. The code behind the `_start` symbol from libc needs to parse the structure to find the relevant information it needs to provide for `main()`, namely `argc`, `argv`, `envp`, and the Auxiliary Vector (AuxV) (Section 2.7.1 on the preceding page). The figure shows a screenshot taken from `lwn.net` [7].

2.7.2 Signals in Linux

A signal in UNIX-like systems, such as Linux, is a message that notifies the receiving process about an event. When a process switches from kernel- to user mode, Linux checks if there are pending events that should be signaled. If so, and a signal callback handler (a sigaction) is specified, control flow is handed over to the callback. For this, Linux prepares a special stack frame that contains contextual information about the signal. There are only a few exceptions that will never be delivered to user space, such as `SIGTERM` [34].

2.8 Static and Dynamic Binaries

Applications under Linux but also many other OSs, know a concept of being dynamic or static. This concept is also called dynamically or statically linkage. The executable file format may instruct the program loader to load additional libraries and link them into memory in order for the binary to fulfil its function. Such programs are dynamically linked. Static binaries, on the other hand, contain all code inside the file, which needs fewer steps to load a program into memory and execute it eventually.

Large and common libraries, such as a system's standard library (e.g., libc), large graphic libraries (e.g., *GTK*[4] or *DirectX*[5]), are usually dynamically linked against applications. This enables to bring bug fixes to software that is already in the field and saves space on disk. On the other hand it requires more functionality of the runtime system, such as a file system where dynamic libraries are located.

Linking against a well-defined interface, such as libc, also enables a basic porting mechanism. To name an example: Many simple applications written for Linux that use basic UNIX functions can compile and run on macOS without modifications to the source code and vice-versa. Figure 2.5 on the next page shows that. A application that expects a well-defined standard interface can work on multiple platforms, if a matching implementation for that interface is available. Static binaries need a recompilation to work on other systems. Dynamic binaries can work without recompilation as long as the program loader supports the executable file format and has a matching library.

[4] `https://www.gtk.org/`
[5] `https://docs.microsoft.com/en-us/windows/uwp/gaming/directx-programming`

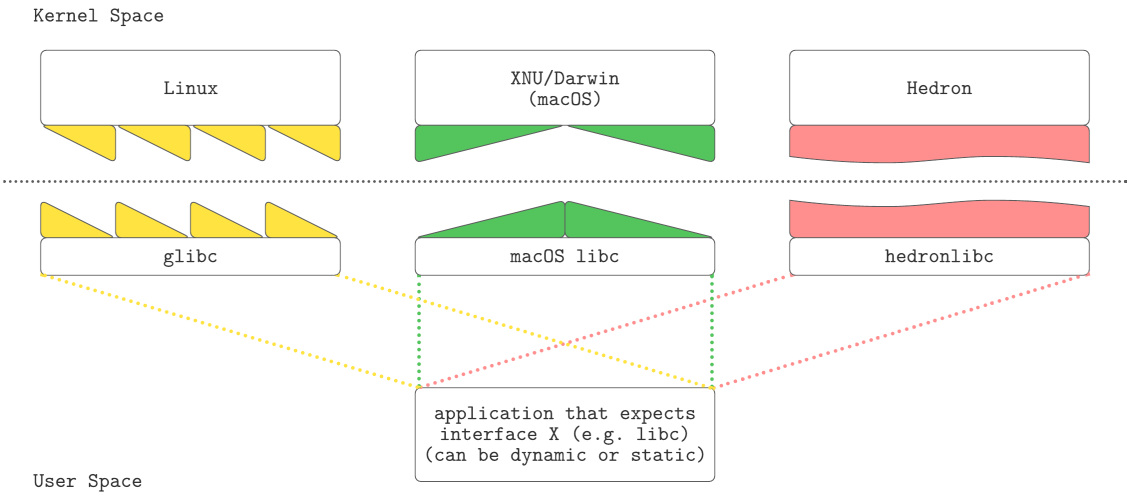

Figure 2.5: The figure shows an application that uses a standard library interface that abstracts all system calls. In the shown case, the application is programmed against the libc standard library. This enables to easily port the binary to other systems by recompiling it and linking the correct libc against it when the program is loaded. The *hedronlibc* does not exist and is also not presented in this work. It is only given as example here. This benefit exists for static and dynamic binaries, but static binaries need a recompilation to contain the correct libc implementation.

2.9 The Rust Programming Language

Rust is a system programming language but Rust is not limited to low-level systems programming. It is expressive and ergonomic enough to make CLI apps, web servers, and many other kinds of code quite pleasant to write. This section gives a brief introduction about some basic Rust abstractions used in several code examples of this thesis. `Rc<T>` is a **r**eference **c**ounted wrapper around a generic type T. It is similar to a shared pointer in *C++*. A `Weak<T>` can be constructed from a `Rc<T>`: it is a non-owning reference. A `Rc<T>` can be used if a parent object references (and owns) a child, whereas a child should reference its parent only with a `Weak<T>`.

`Result<S, E>` is a generic enum with the variants `Ok(S)` and `Err(E)`. Enums in Rust work like a typed union in C. Functions may return the `Result<S, E>` type if they are usually expected to succeed.

If the method `.unwrap()` is called on a result that holds `Err(E)` instead of `Ok(S)`, the Rust program panics. Panicking is a mechanism used for non-recoverable errors in Rust. It helps to prevent undefined behaviour by terminating the program in those cases. Rust programs can be configured to perform a certain action during a panic. They can either loop forever or write to stdout and perform a system call to terminate the program. The latter is the default option of the Rust standard library.

2.10 Summary

In this chapter, I introduced important concepts of the Hedron microhypervisor, such as its kernel objects (Protection Domain (PD), Portal (PT), Semaphore (SM), Execution Context (EC), Scheduling Context (SC)) and the UTCB. Furthermore, I described the Linux system call ABI and what parts are especially important in an emulation layer.

3 Design

This chapter presents multiple design approaches to enable foreign applications under Hedron, i.e., how to reach binary compatibility with a policy-free system-call layer (Section 3.1). After discussing the advantages and disadvantages of those designs, I explain why I chose a particular design. Next, the chapter discusses and compares design approaches to enable hybrid applications under Hedron while taking constraints of the chosen design from Section 3.1 into account (Section 3.2 on page 49). Finally, the chapter discusses the design to emulate a relevant portion of Linux under Hedron that is influenced by the chosen approach to reach binary compatibility (Section 3.3 on page 52).

While the design presented in this chapter is intended to be generic, the implementation and the evaluation focuses on Linux as an example. Linux can be used interchangeably with other operating system's kernels, as long as I discuss the policy-free system call layer in Section 3.1 and not the emulation of Linux functionality in user space in Section 3.3.

The design exploration space spans over two major domains. First, a mechanism to enable foreign system calls under Hedron and thus allow competing implementations in user space, and second, enabling Hedron-native system calls from foreign applications, i.e., enabling hybrid applications. To the best of my knowledge, support for hybrid applications is unique in the context of microkernels.

3.1 Enabling Foreign Applications

This section focuses on the execution of foreign applications under Hedron without hindering or breaking existing Hedron-native applications. It examines whether applications should run in a VM or as a first-class citizen[1] and presents the implications and limitations of those approaches.

Running software with an initially incompatible interface on a host system is nothing new in computer science and well explored since the occurrences of VMs [13]. For example, VMs became widely popular with hardware support on popular architectures to accelerate virtualization in the second half of the 2000s years [17]. Other approaches such as *L4Linux* [18] (2001), the *Windows Subsystem for Linux* (WSL) [4, 35] (2016), *X-Containers* [33] (2019), or

[1] First-class citizens in the context of this work describe applications that run on an equal level to native applications.

the proposed changes to *Fuchsia* [30] (2021) try to achieve the overall same goal but all of them focus on fully/pure foreign applications without hybrid parts.

3.1.1 Reach Binary Compatibility

Since I target unmodified foreign applications, I need a design that allows binary compatibility. Furthermore, the design must enable a functional integration into existing runtime services. For example, a UNIX `open()` system call must map to the interfaces of the runtime system's file-system service.

The major design question is how to achieve binary compatibility with low maintenance costs but also high support of functionality. This can be achieved at multiple levels, namely by running the foreign applications in a virtualized context or by executing them as first-class citizens. In the following, I present several design ideas and discuss them. Afterwards, I show what design I have chosen for this work.

The following list gives a short overview of design ideas to reach binary compatibility. They are discussed in detail on the next pages.

Design Ideas

1. Run foreign applications in a virtualized context (VM):
   a) VM with unmodified guest OS
   b) Para-virtualized VM with a modified guest OS (e.g. L4Linux [18] or *Xen* [2])
   c) "Lightweight VM" with forward kernel provided by the runtime system (e.g. *ELK Herder* [29] or *gVisor* [16])
2. Run foreign applications as first-class citizens:
   a) Extend Hedron's system-call ABI to reach binary compatibility for the foreign applications.
   b) Provide a *POSIX* emulation layer through a custom standard library that maps functions of the standard library to proper runtime functionality (e.g., to `call` system calls).
   c) Add a mechanism to Hedron that catches and forwards foreign system calls to a userland component that implements the policy to correctly emulate the behaviour of the foreign OS.

Approach **1.a)** can be excluded from further discussion because it isolates the application from the runtime system's services. Approach **2.a)** can also be ignored because Hedron is a microkernel and should stay one, as defined in the goals of my thesis in Section 1.2 on page 23. The other approaches are discussed in detail in the following.

Running Foreign Applications in a Para-virtualized VM

This section corresponds to approach **1.b)**. Since Hedron is a microhypervisor, it already brings the necessary functionality for hardware-assisted virtualization on `x86_64`. The use of para-virtualized VMs is well suited for isolation between tenants on a shared system because the virtualized Linux kernel gives tenant applications binary compatibility while the performance overhead that guest applications face is kept low. However, the existing and default setups prevent any interaction from the VM with runtime services of the main system.

To enable interactions with the runtime services one could think of a modified Linux in a VM running under Hedron that has a para-virtualized interface to the main runtime environment. This interface acts as a bidirectional communication channel between the main runtime environment and the para-virtualized Linux. Commands sent through this channel can instruct Linux for example to start or stop certain Linux applications or to create memory mappings. Running foreign applications in a para-virtualized VM requires changes to the guest kernel because it requires a para-virtualized interface to Hedron. Parts of the logic can be implemented in the guest's user space that uses this para-virtualized interface.

The advantage of this approach is that foreign applications can use the full feature set of the guest kernel. But keeping the para-virtualized Linux kernel up to date requires a significant maintenance effort. OSs are complex software projects and the maintenance is an individual process per OS. In the worst case, multiple versions of foreign kernels with multiple corresponding user-space daemons have to be maintained per supported foreign OS at the same time. This is against the goal that the mechanism plus corresponding policy implementations should be easily maintainable.

Since a tightly coupled integration into the services of the main runtime environment, such as logging or accessing the file system, is desired, running foreign applications in a (para-virtualized) VM triggers numerous expensive VM exits followed by VM entries. As a lively communication between the Hedron-world and the VM-world can be expected for hybrid applications, the costs increase compared to pure VMs-internal processing.

Running Foreign Applications in a Lightweight VM (Forward Kernel)

This section corresponds to approach **1.c)** of Section 3.1.1 on the facing page. A more lightweight but still virtualized approach is a foreign application that runs with hardware-assisted virtualization features but not on top of the guest OS. Instead, the Hedron userland provides a forward kernel for virtualized foreign applications that forwards system calls to the Hedron userland. This approach is similar to Compute Node Kernels (CNK) [14].

Using a forward kernel enables the typical isolation guarantees of VMs but does not offer the binary compatibility enabled by reusing the guest OS. To reach binary compatibility, Linux system calls can be partially implemented in the forward

kernel, such as in *gVisor* [16, 42]. However, system calls not handled by the forward kernel must be implemented by the host. Solutions such as *ELK Herder* [29] forward all system calls to the main system.

ELK Herder shows a general performance overhead of 2.2% [29]. This overhead can be measured if the tested workload does not require many system calls. System-call intensive workloads on gVisor for example come with a 50% overhead in system call throughput. These costs originate from expensive VM exits and reentries.

The maintenance overhead is reduced compared with approach **1.b)** (Section 3.1.1 on the previous page) because only the forward kernel and a component in Hedron's user space (the OS personality) needs to be maintained. However, this approach requires regular updates to a dedicated user-space component that implements foreign system calls. Depending on the OS, this may be less complicated, as maintainers have full control over their build setup and do not have to deal with foreign code bases. Using a forward kernel is similar to **2.c)** (which is discussed shortly) because binary compatibility is reached via a dedicated user-space component.

Reaching a functional integration into the existing runtime environment with a forward kernel is less flexible than with approach **1.b)** because in **1.b)** one can provide user-space daemons in the modified guest OS. Such a daemon can use a para-virtualized gate to the outer world. This does not apply to a virtualized environment that only consists of the foreign application and the forward kernel. With a forward kernel, the only possibility to enable a functional integration of the foreign application into the main runtime system is to map the used system-call interface onto corresponding functions of the runtime system.

It might be possible to trigger Hedron-native system calls from the foreign application so that the forward kernel or the responsible user-space component can distinguish them from foreign system calls, but for all virtualized approaches, i.e., **1.b)** and **1.c)** [this section], the performance penalty of frequent VM exits and entries is undesirable. This is the reason why I did not choose a virtualized approach. In the following, I discuss approaches that enables foreign applications as first-class citizens without the overhead of VM exits and entries.

First-Class Citizens With a POSIX Emulation Layer

This section corresponds to approach **2.b)** of Section 3.1.1 on page 40. One approach to enable foreign applications as first-class citizens is to link a matching standard library like a POSIX emulation layer. Such a layer forwards libc functions to corresponding PTs of the Hedron runtime environment (however, there are differences between libc implementations and POSIX, but they are irrelevant here). As explained in Section 2.8 on page 36, an advantage of applications that dynamically or statically link against a standard library (or a well-known interface such as POSIX) is that they are easily portable. Dynamic applications can be ported without recompilations in general (depending on the complexity and the executable

file formats the loader supports). In this section, *POSIX* emulation layer can be read interchangeable with other well-defined layers, such as *win32*.

For Microsoft Windows and UNIX-like systems it is the default to dynamically link applications. However, replacing the dynamically-linked standard library with a modified version that maps function calls to the corresponding runtime services is not sufficient. First, applications can be statically linked, so that the standard library cannot be replaced (once compiled). Second, applications can always contain hard-coded system calls, which would also not be covered by a POSIX compatibility layer. Besides these problems, applications might require different versions of the standard library, which would all have to be maintained.

One could patch hard-coded system calls inside the binary to point to PTs that fulfill the promised functionality. This patching can either happen when the program is loaded into memory or at install time. A patching mechanism allows the replacement of existing Linux system calls with function calls that implement the desired functionality. A similar approach is taken by X-Containers [33]. The infrastructure required to patch machine code is tricky and outweighs the costs of the small modification to Hedron for approach **2.c)** that I describe in the upcoming section.

With this approach, which relies on applications are programmed against well-known compatiblility layers, maintainers of Hedron and its runtime system have to maintain a standard library/compatibility layer at possibly different versions, which is rather complex, if more software is supported.

Providing a POSIX compatibility layer supports dynamic Linux binaries through the dynamically linked standard library, static binaries by linking them statically against a copy of the library, and all hard-coded system calls outside the standard library can be patched as described above. Static binaries can only be supported by recompilation, whereas already produced dynamic binaries can also be supported as long as they are linked against a proper library. Since with a POSIX compatibility layer static binaries require toolchain adjustments and because one goal of this is work to prevent them for application software, this design approach is not optimal. Therefore, I decided for another approach, which I discuss in the following.

First-Class Citizens With a Foreign System-Call Policy In User Space

This section corresponds to approach **2.c)** of Section 3.1.1 on page 40. Since Hedron is a microkernel and should remain one, Hedron cannot emulate foreign system calls itself. Instead, Hedron can forward foreign system calls to a user-space component. This component is called emulation layer or OS personality in the following. If Hedron catches all system calls for native and foreign applications directly, which is in contrast to the approach **2.b)** described earlier, both static and dynamic unmodified foreign applications can be supported. Furthermore, hard-coded foreign system calls outside the standard library are supported as well. This unifies any maintenance work to catch foreign system calls to one single interface: the system call handler in Hedron.

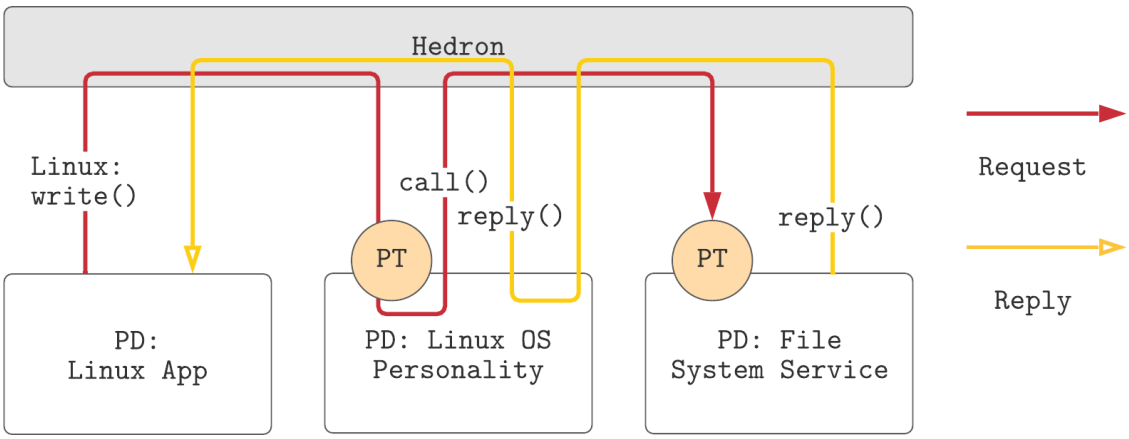

Figure 3.1: Overview of a foreign application running as first-class citizen that performs a Linux `write()` system call, which is caught by Hedron and delivered to user space. The communication path follows my discussed design **2.c)**. The Linux emulation component handles this call and replies eventually. For each IPC step a context switch is required.

Supporting foreign system calls in Hedron needs a mechanism that enables the discovery of foreign system calls and distinguish them from native ones. If a foreign system call is detected, Hedron has to forward it to a user-space component that implements a policy. Figure 3.1 shows how the communication path for a foreign system call looks like. It visualizes a simplified version of how a Linux application running as first-class citizen writes to a file and how the write system call is handled. Inside the figure, the Linux application performs a `write()` system call. Hedron can be modified to know if a system call originates from a "foreign" Protection Domain (PD). The system call is forwarded to the Linux OS personality. The Linux OS personality sees that it should forward the request to the file-system service. The latter replies to the OS personality and the personality finishes the foreign system call with a final `reply()` system call eventually.

The existing IPC mechanism of Hedron can be reused for the forwarding of the system call to the OS personality so that a foreign system call is handled similar to exception IPC. Relevant CPU state, such as all the general-purpose registers, can be transferred through the UTCB, as described in Section 2.3.4 on page 31.

The PT callback that handles the foreign system call can point to any component inside the runtime system. When the emulation layer handles a foreign system call it knows whether the foreign application made a Linux, a Microsoft Windows, or a macOS system call because the emulation layer knows the application's context. For different OS ABIs it might take multiple dedicated OS personality components. Once the OS personality receives the request, the request can be forwarded further, for example to the file-system service, as shown in Figure 3.1.

In contrast to the discussed para-virtualized VM design that reuses the guest OS, reaching binary compatibility with this design is challenging. On the other

hand, benefits are that there are no costly VM exits or entries, and it does not require maintaining Linux patches or patches to a POSIX compatibility layer. The changes to Hedron to detect foreign system calls are onetime costs whereas the OS personalities will require frequent maintenance to keep up with the latest upstream changes. Since the implementation goal of this work is not full Linux binary compatibility, but the ability to run simple Linux applications, and foreign application software can be supported as it is, i.e., unmodified, I chose approach (**2.c)**.

In the next sections, I discuss concrete design decisions that build on the approach of running foreign applications as first-class citizens with a foreign system-call policy in user space.

3.1.2 Modifications to the PD-Object in Hedron

In this section, I discuss changes to the Protection Domain (PD) kernel object that enables the system call handler to distinguish "foreign" from "native" system calls. The PD models in combination with global ECs and corresponding SCs what is known as process. It acts as a resource container. Since the binary becomes a process during runtime, the PD kernel object can be modified to know whether it is "foreign" or "native". This information can be transferred on PD creation (`create_pd()` system call).

The control flow in Hedron's system call handler can be altered with the information if the PD that belongs to the EC that triggered a system call is "foreign" or "native". If Hedron sees that it should not handle a system call natively because it originates from a foreign application, Hedron should forward the system call to a specified Portal (PT) via kernel-initiated exception IPC instead. Exception IPC includes relevant CPU state in the UTCB, as described in Section 2.3.4 on page 31. The PD needs a new property to know which PT should be used for that. This information can also be transferred during object creation (`create_pd()` system call).

The OS personality, the target of that IPC, must implement the policy in user space that emulates the expected behaviour of the foreign OS. The mechanism to use exception IPC to handle system calls is similar to the proposed approach in Fuchsia with *starnix* [30].

3.1.3 Handling Foreign System Calls in User Space

In the previous section, I discussed what changes enable Hedron to forward a foreign system call to the userland. In this section, I discuss how the userland has to cope with this request.

When the OS personality starts its execution, it needs to figure out what foreign system call was executed. It will receive the whole CPU state of the original system caller including relevant information, such as a system call number and arguments, from the UTCB. Afterwards, it needs to map the desired functionality onto available runtime services. When the user space handler is done, it can finish the foreign

system call with a `reply()` system call eventually. Hedron updates the CPU state of the caller with new values from the UTCB.

The foreign system-call handler component can use all existing runtime services, such as the memory allocator or the logging service, before it replies to a foreign system call. Hence, the foreign system-call handler running in user space is the place where the integration into the existing runtime system happens. As example, a Linux `mmap()` or `brk()` system call can be mapped onto the memory allocation service of the runtime environment.

3.1.4 Need for Mediators

On a high level, my work aims to translate between the world of Hedron with its runtime environment and several incompatible foreign worlds. With my chosen design (**2.c)**; see Section 3.1.1 on page 43), the entry into the user space for a system call is specified by a Portal (PT). This PT must belong to a mediator that handles the foreign system call, as described in Section 3.1.3 on the preceding page. Thus, the OS personality respectively the emulation layer component fulfils the role of this mediator. Depending on the foreign system call ABI this translation may vary in complexity.

Mediators are expected to hold relevant state about foreign applications. For example, Linux uses file descriptors. Runtime services, such as the file-system service, may have internal data structures to keep track of open files too. However, this is not necessarily compatible with the file descriptor model used by Linux or whatever the corresponding foreign OS is. Hence, the role of the mediator is not only to translate system calls from one world to another but also to map management structures from one world to another and keep track of them.

Additionally, the mediator must bridge the gap between the privilege-model of the foreign system and the capability-based world of Hedron. A practical solution might be that the mediator receives all capabilities necessary but enforces an internal policy with corresponding bookkeeping to hand out permissions to foreign applications as needed. The OS personality must be trusted to ensure a simultaneous execution of multiple foreign application within its responsibility without tampering with them.

A big variety of OSs, such as UNIX, expects user-memory addresses as arguments for many system calls. The calling application relies on that the kernel reads from these addresses or writes to them. An example is the UNIX system call `write()` [39].

```
sys_write(unsigned int fd, const char __user *buf, size_t count)
```

A runtime system under Hedron will be constructed in a way that most interfaces expect relevant IPC payload embedded inside the UTCB for better performance. This is intended by Hedrons IPC design. There might be reasonable exceptions where usually larger amounts of data are expected, such as file transfers. In this

case, shared memory is a more performant solution for a service interface. Whatever the interface looks like, the mediator can bridge the gap between those two worlds. To name one example, parameters from a Linux `write()` do not necessarily map 1:1 to the corresponding functionality inside the runtime environment. The mediator sees what parameters a foreign application passes, translates them if necessary, and forwards the request to the right runtime service.

To enforce strong safety guarantees, the mediator should never map memory pages from the application to the service. If the application performs a write system call and the file-system service offers a similar interface, the mediator should introduce a dedicated receive and send window that is page-aligned. Memory mappings can only be granted at the granularity of a page. Without a dedicated window, it might happen that the file-system service gains access to the stack of an application because the data of the write operation lies in the middle of a stack page.

Figure 3.1 on page 44 shows the practical relevance of the OS personality acting as mediator. A Linux application (shown left) wants to write to the file system. The figure shows the communication path of that request. If the mediator is a dedicated PD as shown in the figure then it takes four cross-PD context switches from the beginning of the request to the delivery of the response.

Possible Optimization

This overhead can be reduced with the following approach. Functionality from the mediator component can be moved into a library. The mediator can map this library into the address space of a foreign application on program startup. This happens transparently during runtime and foreign applications will never know about this. The library contains the PT entry point to which Hedron will forward foreign system calls. As a result, it requires two less expensive cross-PD context switches, as you can see in Figure 3.2 on the following page. They are replaced with lightweight PD-internal context switches (no need to switch the address space).

The proposed optimization works as long as the mediator library can maintain all important state by itself. Once state from other Linux applications is required, for example when writing to a pipe, it is necessary to talk with other components of the system, such as the Linux mediator or a pipe service. This depends on the implementation of the runtime system.

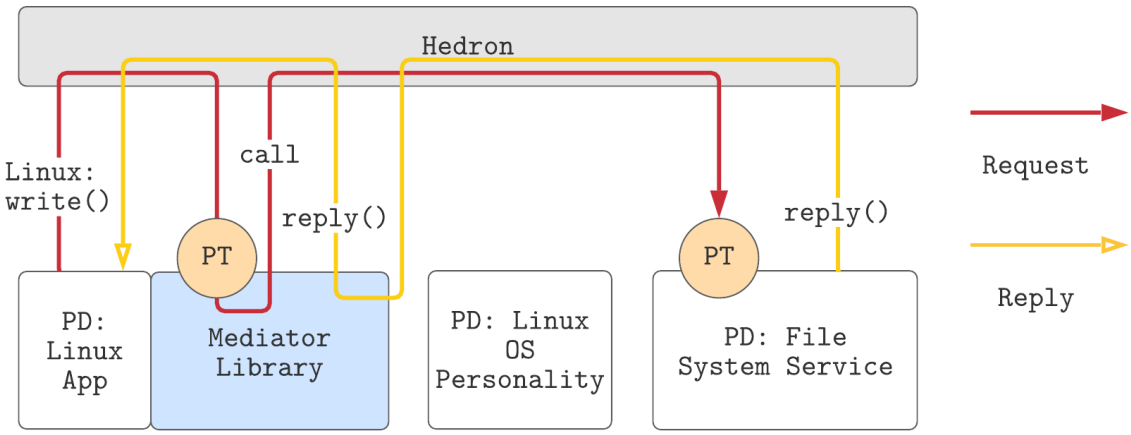

Figure 3.2: Optimized version of the architecture shown in Figure 3.1 on page 44 where two less expensive context switches are required because the mediator library lives inside the same address space (highlighted in yellow). An expensive cross-PD context switch is still required when talking with the file-system service. In this example, the Linux mediator component that is running as dedicated PD, is not involved. However, it may be called by the mediator library in some cases e.g. to retrieve or update state.

3.1.5 Implications and Limitations for Foreign Applications

Taking Linux es example, not all libc interfaces are suited for a 1:1 mapping to Hedron functionality if they violate certain security concerns of a microkernel-based systems. Thus, the OS personality may not emulate certain system calls. An example might be the libc function `fork()`. The following quote summarizes problems of it:

> *Fork is an anachronism: a relic from another era that is out of place in modern systems where it has a pernicious and detrimental impact. As a community, our familiarity with fork can blind us to its faults. Generally acknowledged problems with fork include that it is not thread-safe, it is inefficient and unscalable, and it introduces security concerns. Beyond these limitations, fork has lost its classic simplicity; it today impacts all the other operating system abstractions with which it was once orthogonal.* – Quote taken from "A fork() in the road" [3]

Thus, a runtime system for Hedron will never be designed to be 100% libc compatible for security reasons and boundaries of Hedron. Application developers need to cope with that and refactor some parts of their applications to use functions from a Hedron library instead of functions such as `fork()`. An alternative may be the libc function `posix_spawn()` which implements fork-like semantic.

It depends on the use case of a runtime system how the trade-offs between convenience/compatibility and security are weighted. As I focus on the execution of simple Linux applications, I leave this question for future work.

3.2 Enabling Hybrid Applications

In Section 3.1 on page 39, I decided for an approach where Hedron's system-call handler is responsible for foreign system calls. In this section, I discuss several strategies to solve the other major design challenge of my work: Enabling hybrid parts inside foreign applications. The discussed strategies are compatible with the design decisions I made so far. After the discussion, I explain what strategy I chose for this design challenge and why I made this decision.

Enabling native system calls from foreign applications, i.e., hybrid applications, means that Hedron's system-call handler must distinguish foreign system calls from native ones, if they come from a foreign PD. As discussed in Section 3.1.2 on page 45, Hedron knows if a system call originates from a native or a foreign PD. Hence, this mechanism only needs to do work if the PD is a foreign PD. Figure 3.3 on the next page shows the required system-call control flow inside Hedron for native, foreign, and hybrid applications.

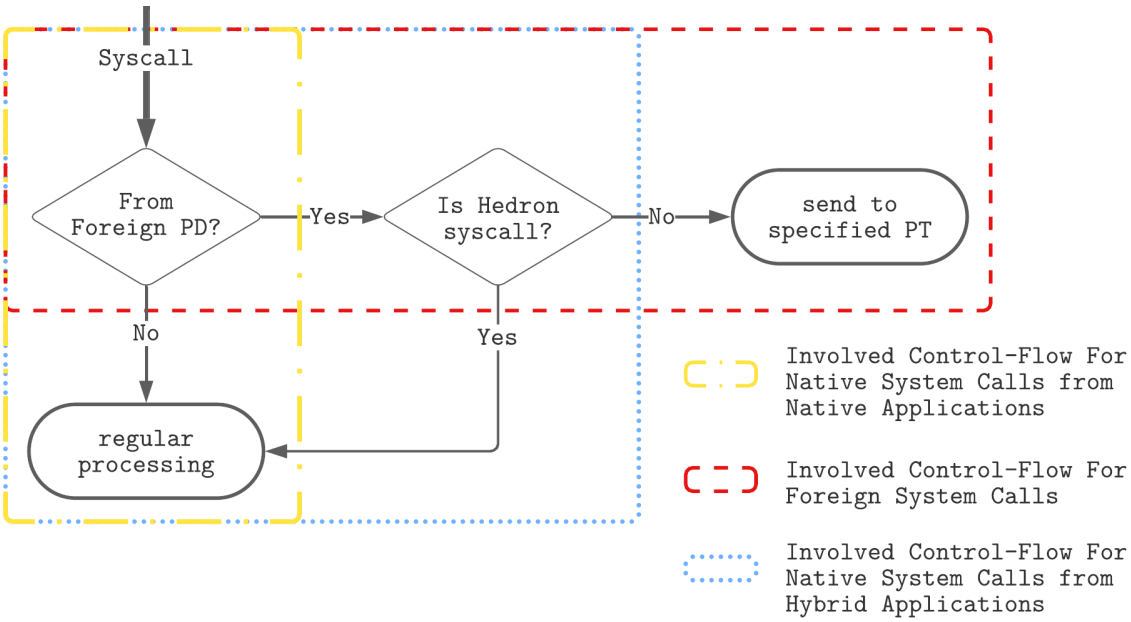

Figure 3.3: Flow chart of modified system-call handling inside Hedron for native, foreign, and hybrid applications. First, the kernel checks if the system call originates from a foreign PD. If this is not the case, Hedron processes the system call in the regular way (yellow box). Otherwise, Hedron checks if the foreign application made a native system call. If so, it forwards the request to the regular processing (blue box). If it is a foreign system call, the handler forwards the request to the specialized foreign system-call PT (red box).

3.2.1 Identify Hedron System Calls from Foreign Apps

A generic solution to distinguish native system calls from foreign applications is challenging because system-call ABIs can be overlapping (see Table 2.2 on page 33). This section discusses several approaches how this can be achieved.

Register-based

System-call ABIs might overlap and one register that is unused in one ABI might be used by another. Hence, using a magic value in a specific register is not generic enough. One could instruct Hedron to check one register of a hard-coded selection of registers for a magic value. However, this is not generic. It might happen that an application stores the magic value as part of its working set of variables in that register while it performs a foreign system call. Hedron will falsely assume it is a native system call in such a situation. A register-based approach is not solid enough.

Identify System Call Origin By Address

After a system call is triggered with the `syscall` instruction on x86_64 the register `rcx` contains `rip + 2`. This is the instruction pointer at the address that caused the system call plus two bytes which is the length of the `syscall` instruction. Thus, Hedron can find at which address a system call originates. If the system call was triggered from a foreign PD and is within a special range, this special range can mark all system calls as native system calls.

This is similar to an approach proposed for Linux to distinguish Microsoft Windows system calls from Linux system calls in the *WINE* project [5] and also similar to a mechanism in *OpenBSD* [28].

However, identifying system calls by their origin address requires work from the dynamic linker when a hybrid application is loaded because all native system calls must be behind a dynamically linked library. The loader can load this library into a specific address range known to Hedron. However, this may clash with addresses the program needs for itself. This can be solved by making the library code position independent and telling Hedron at which address range the library that triggers native system calls is loaded. This adds complexity to the linker. Furthermore, developers of hybrid applications must dynamically link against a "hedron standard library". This approach is complicated and restricts application developers.

A Field Inside The UTCB

In Hedron each EC has its dedicated UTCB, as shown in Figure 2.3 on page 31. The UTCB is always mapped inside Hedron, so Hedron can always access it during a system call. A foreign application will not know about its UTCB but the hybrid part of a hybrid application can. The UTCB header can be modified to contain a new field used for flags that affect the system-call behaviour. Before the hybrid code performs a Hedron-native system call, the hybrid application must set a bit of this field to true. During a system call, this bit tells Hedron to handle a system call as native one even if it comes from a foreign PD. Otherwise, it is a foreign system call (see Figure 3.3 on the facing page). The toggling of the bit can be hidden behind a "hedron standard library".

The UTCB is an already existing infrastructure that can easily be accessed in user space and kernel space. Each EC has a dedicated UTCB, as explained in Section 2.3.4 on page 31. Using a flag in the UTCB outweighs the negative aspects of the register-based approach and the one that checks the origin of system calls. The UTCB head must be modified in a way so that it includes a flag that Hedron can check. This flag indicates whether a system call is a native one even if it comes from a foreign application. I refer to this flag from now on as Native System Call Toggle (NSCT).

3.2.2 Implications and Limitations for Hybrid Applications

The hybrid parts of foreign applications must activate the NSCT before each Hedron-native system call and deactivate it afterwards. Otherwise, undefined behaviour will happen, i.e., unexpected system calls or exceptions are triggered.

Foreign applications can use the typical testing suites and utilities available for their respective platforms, except for the hybrid part, i.e., Hedron-native system calls. A test infrastructure for this is not covered by this thesis. Developers need to deactivate the hybrid part or emulate the Hedron behaviour manually when they execute tests or run the binary on the native foreign platform.

3.3 Emulating a Relevant Portion of Linux

In Section 3.1 on page 39 and Section 3.2 on page 49, I described the design of a policy-free system-call layer that fulfills the goals specified in Section 1.2 on page 23. The design decisions I made have an influence on how to reach binary compatibility. In this section, I discuss relevant design to emulate a relevant portion of Linux in user space, i.e., how to design a Linux OS personality. The section covers required functionality and environmental requirements to start and execute Linux applications.

3.3.1 Important System Calls

At first, I investigate relevant Linux system calls that are required to start and run several basic "Hello World"-programs. By using the *strace* utility, I can deduce the minimum functionality the runtime environment has to provide for the system-call layer. Table 3.1 on the next page summarizes the observed findings. It shows which system calls several "Hello World"-programs compiled with different compilers for different languages use at runtime. Each binary was build as static binary. The programs written in Rust and C are both explicitly linked against musl.

As Table 3.1 on the facing page shows, many system calls are triggered during execution, although the programs just print "hello world" to the screen. The reasons for this is that all programs have a small runtime which is not visible in source code but attached to the program during the linking step. On Linux, this startup routine comes from the libc. This is the code that initially starts executing and eventually calls `main()`. Rust programs under Linux are also linked against libc. Rust builds its own runtime between the one from libc and the actual entry point into a Rust program.

The table shows that the runtime of Go uses complex system calls, such as `clone` and `futex`. Therefore, Go is out of scope of this work.

Syscall	C	Go	Rust	Zig
arch_prctl	1x	1x	1x	1x
brk	-	-	2x	-
clone	-	3x	-	-
close	-	1x	-	-
exit_group	1x	1x	1x	1x
ioctl	1x	-	-	-
fcntl	-	3x	-	-
futex	-	1-4x*	-	-
gettid	-	1x	-	-
mmap	-	8x	2x	-
mprotect	-	-	1x	-
munmap	-	-	1x	-
read	-	1x	-	-
readlinkat	-	1x	-	-
poll	-	-	1x	-
prlimit64	-	-	-	1x
rt_sigaction	-	114x	5x	-
rt_sigprocmask	-	6-10x*	3x	-
set_tid_address	1x	-	1x	-
sched_getaffinity	-	1x	-	-
sigaltstack	-	2x	3x	-
write	-	1x	1x	1x
writev	1x	-	-	-
Compiler version	9.3.0	1.13.8	1.55.0	0.81
libc version	musl@1.2.2	bundled go libc	musl@1.2.2	bundled zig libc

Table 3.1: Overview of required system calls from several static “Hello World”-binaries on Linux. There are differences between binaries because each language comes with its own runtime. Results marked with * mean the occurrence is variable, thus, changes from run to run. The count of each system call does not necessarily increase the complexity in the emulation layer and stands here for informational completeness. The table might look slightly different if another libc implementation than musl is used.

3.3.2 Constructing the Initial Linux Stack Layout

In Section 2.7.1 on page 34, I described the initial stack layout that Linux creates for applications. The layout is tricky to create because the creation happens from the loaders address space, whereas the absolute pointers inside the layout have to be valid within the loaded application's address space. This can be solved by having a data structure that keeps track of pointers in both address spaces, which ensures the right pointer lands at the expected place. For example, that the `argv[0]`-pointer points to the correct address in the address space of the application.

3.3.3 Sending Signals

As described in Section 2.7.2 on page 36, Linux checks during a context switch to a process if there are pending signals. The runtime environment does not have the ability to intercept at this point because scheduling and context switches are in the responsibility of Hedron. Hence, sending signals to Linux processes is not easy to solve.

Signals are complex and out of scope of this work. In the following, I discuss two approaches how signals might be implemented with the existing Hedron mechanisms. The discussion gives a few pointers to relevant problems but the solution of these problems is future work.

Approach A: Use Recall-Exception

A naive approach may be to use the `ec_ctrl_recall()` system call, i.e., raise a recall-exception, to handle signals. This will cause the global EC that was specified in the corresponding `ec_ctrl_recall()` system call to trigger its recall-exception. The corresponding handler will be invoked via exception IPC. This handler must live inside the Linux emulation layer. For example, if a `SIGINT` is raised for a foreign process, the Linux emulation layer performs a `ec_ctrl_recall()` system call to force the target global EC specified by the system call into its recall-exception handler. The global EC must belong to the PD of the Linux process. Since Linux delivers signals per process but not per thread, the Linux OS personality can select any of the available global ECs. When the emulation layer handles the recall-exception, it can alter the CPU state as described in Section 2.3.4 on page 31. For example, it prepares a special stack frame required for signal handling and set the instruction pointer to a corresponding signal handler that the user application specified beforehand.

This approach comes with two problems. First, it prevents a foreign Linux application from handling recall-exceptions itself. Second, if all available global ECs are blocked by a system call, the signal cannot be delivered until one unblocks. Hedron is as it is at the time of this thesis not capable of finding if and why a certain global EC is blocked. Thus, currently user-space components including the roottask have no way to find out which global EC is not blocked, i.e., available, at the moment of

signal delivery. This is in contrast to Linux which can unblock and abort certain system calls and return to userland with an `EINTR` error code. Implementations of libc handles this and may restart the aborted system call.

Let us assume that modifications to Hedron enable user-space mechanisms that allow a detection of blocked global ECs. Still, to support aborting of Linux system calls all included userland components involved in a foreign system call must either be capable of aborting certain system calls and a subsequent restart or be idempotent regarding their operations.

Approach B: Use a Dedicated Global EC

Instead of using the recall exception to deliver signals, another approach for sending and handling signals is to use a dedicated global EC per foreign application. This global EC receives its time slice from a corresponding Scheduling Context (SC). It needs a small portion of initialization code that is independent from the foreign application. This code would have to put itself into sleep by using a Semaphore (SM). The Linux OS personality may unblock the semaphore and tell the handler information about a signal through shared memory. This solution is applicable because signals make no assumptions about the thread that handles a signal. However, a problem which arises from this approach is that a single threaded Linux application may not synchronize access to global data structures in its signal handler because it relies on that two units of execution never operate on the same data in parallel. Thus, this approach breaks existing semantics.

Signals That Are Not Sent to Linux Apps (SIGKILL)

Some signals such as `SIGKILL` cannot be caught by a Linux application. Instead, Linux uses it to immediately kill a process. With the current design of Hedron, resources cannot be freed when they are blocked. If an EC of the application that should be killed is blocked by a semaphore, the EC cannot be freed. Hedron has no mechanism that enables the release of blocked resources. Hence, if a EC is blocked, corresponding kernel objects, at minimum the blocked EC and its corresponding PD, cannot be freed because there will be references inside the kernel to them. Thus, the Linux application are not killable by a signal.

3.4 Summary

In this chapter, I introduced multiple approaches about how to enable foreign applications under Hedron. I decided to enable them as first-class citizens side-by-side with native applications. Hedron catches all system calls at its system-call entry, which enables static as well as dynamic unmodified foreign binaries next to native ones. Hedron detects foreign system calls and forwards them to an user-space handler which implements the policy to handle those system calls. Multiple OS personalities can exist concurrently. In real world scenarios, applications might need to avoid using certain functions for security reasons, such as the `fork()` primitive.

If a foreign application has hybrid parts, i.e., a hybrid application, it must set the Native System Call Toggle (NSCT) inside the UTCB to tell Hedron to treat the next system call as native system call. The userland is responsible to set up the relevant environmental conditions for foreign applications, such as the initial stack layout for Linux applications.

4 Implementation

This section provides a detailed overview of various aspects of my implementation. At first, I describe the modifications required for Hedron. Afterwards, I talk about the custom runtime system in Rust that includes the Linux OS personality.

4.1 Changes To Hedron

In Section 3.1.2 on page 45, I discussed that the PD-object needs modifications to distinguish foreign PDs from native ones. In Section 3.2.1 on page 51, I talked about the Native System Call Toggle (NSCT) that is required for detecting native system calls from hybrid applications. This section discusses my changes to Hedron that implement these design ideas.

I modified Hedron so that it can recognize foreign system calls and deliver them to a user-space component via IPC. These changes require less than 30 additional or modified SLOC (without comments and empty lines). In this process, I added two new properties to the PD kernel object:

- `bool is_foreign_application`
- `mword syscall_handler_pt_base`

To transfer the system call with its parameters to a user-space component that enforces a policy, Hedron checks whether a PD is foreign and if the NSCT is false inside its system-call handler (see Figure 3.3 on page 50). If so, it calculates the capability selector for the destination PT and sends a kernel-initiated IPC message to it. The full CPU state of the caller is transferred via the UTCB. The user-space component will handle the PT call similar to regular exceptions. The heart of the flexible policy-free system-call layer is shown in Listing 4.1 on the next page. Inside `sys_foreign_syscall()`, Hedron finds the right capability selector of the destination PT. It does this by adding the two values:

`current()->pd->syscall_handler_pt_base` and `current()->cpu`.

If a foreign system call is recognized, Hedron sends a message to the PT that is responsible for foreign system calls on the right CPU (PTs are CPU-local). Since the special exception data layout is copied into the UTCB (Section 2.3.4 on page 31),

```
void Ec::syscall_handler()
{
    bool foreign_pd = current()->pd->is_foreign_pd;
    bool nsct = current()->utcb->nsct();

    if (EXPECT_FALSE(foreign_pd && !nsct)) {
        // sends IPC msg to userspace; function doesn't return
        sys_foreign_syscall();
    }

    switch (current()->sys_regs()->id()) {
    // regular syscall procedure
    // ...
```

Listing 4.1: Snippet from Hedron's system call handler with my modifications. It shows the relevant additions that check if a certain system call is a foreign system call or a native one. Afterwards, it invokes the proper next handler function.

Hedron makes sure that the OS personality has relevant access to the CPU state of the caller when it receives the IPC call.

The user space must make sure that a proper Message Transfer Descriptor (MTD) is assigned to the syscall handler PTs when they are created, as described in Section 2.3.4 on page 31.

Furthermore, I modified Hedron's `create_pd()` system call to carry a new additional argument. This argument encodes if the new PD is a foreign PD and if so, what the base selector for foreign system-call IPC is.

4.2 Runtime System

The major engineering effort was the runtime system, which I wrote from scratch in Rust. There were no specific reasons to either keep or drop the existing runtime system written in C++, but the opportunity to use Rust with *Cargo* as productive and developer-friendly build environment resulted in the decision to start from scratch. The runtime system includes the Rust libraries *libhedron* and *libhrstd* as well as the roottask that provides several runtime services.

libhedron contains all important constants, type definitions, and system call wrappers for Hedron. libhrstd takes the role of the standard library. It contains common abstractions for memory, UTCB handling, IPC, and more. It can be used by Hedron-native Rust applications or by hybrid applications written in Rust. Applications can locate PTs of well-known runtime systems with it and communicate

with them. libhedron enables to serialize arbitrary data into the UTCB and deserialize data from there. libhrstd provides abstractions to either embed IPC payload or to transfer a user pointer with the corresponding length of the data structure.

4.2.1 Well-Known Runtime Services

My runtime system offers an allocator service, a logging service, an in-memory file-system service, a foreign system-call handler service (the Linux OS personality), and an exception handler service. For simplicity, all services are located inside the roottask and exposed via PTs to other PDs.

Furthermore, my runtime system offers two services called "echo" and "'raw' echo". They are useful to measure several IPC metrics and are discussed later in the evaluation in Section 5.3 on page 75.

4.2.2 In-Memory File-System Service

The current implementation of the roottask also contains the in-memory file-system service. This design choice was made by myself for simplicity. Its interface is close to the typical file interface of POSIX. This means that `open()`, `read()`, `write()`, `lseek()`, and `close()` use the same parameters as Linux and other UNIX-like and POSIX-compliant systems. I chose this design because this interface is well-known to software developers. Furthermore, it simplifies the mapping from the Linux world to the Hedron world.

4.2.3 Process Management

libhrstd contains convenient structs that are wrappers around Hedron kernel objects, namely Protection Domains (PDs), Portals (PTs), Semaphores (SMs), Execution Contexts (ECs), and Scheduling Contexts (SCs) (explained on Section 2.3.3 on page 29).

With properties of type `Rc<ObjType>` and `Weak<ObjType>` all structs can hold references to sub and parent objects. I described these Rust abstractions briefly in Section 2.9 on page 37.

A process is an abstraction of my runtime environment that wraps a PD object. Additionally, it contains information, such as the ELF file, the name of the executable, and its parent process. This relation is shown in Figure 4.1 on the next page. The roottask knows a struct `Process` that wraps around the struct `PdObject`. With the relations shown in the figures, all relevant information about related objects can always be obtained. For example, if a PT was called, this model can be used to determine which PD it was delegated to. My process model assumes that PTs are not used by multiple PDs but created individually for each PD.

Figure 4.1 on the following page omits that a PD can contain capabilities to all kinds of kernel objects for simplicity but the important delegation of PTs to PDs is shown on the right side.

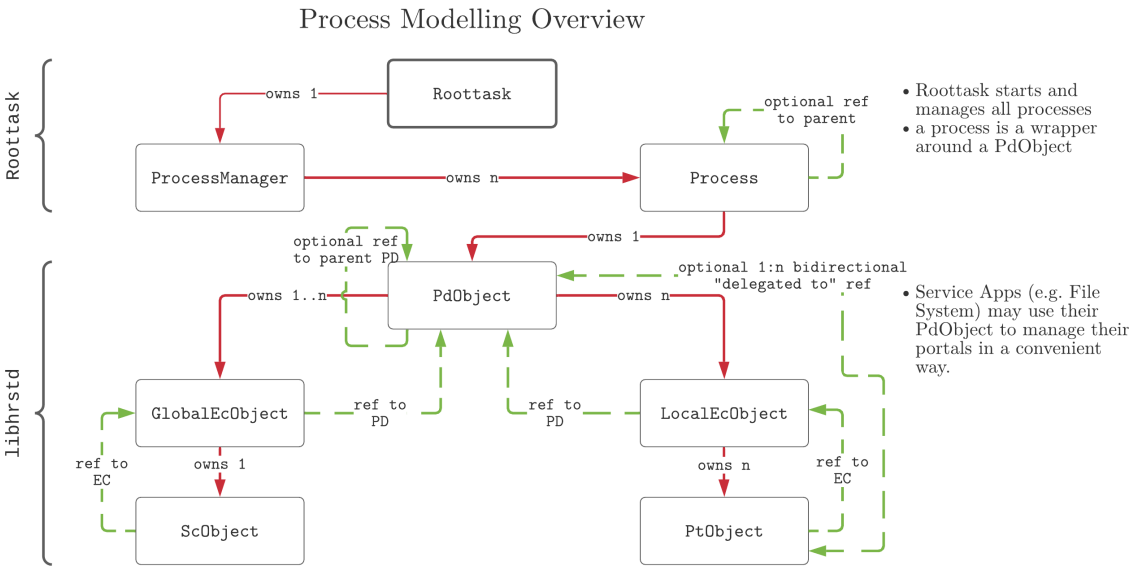

Figure 4.1: Process modeling inside the Rust runtime environment. The roottask manages the process manager, which manages processes. A process is a convenient wrapper around a `PdObject`. `{Pd,{Global,Local}EC,SC,Pt}Objects` are Rust types that are convenient runtime helpers for the underlying Hedron kernel objects plus corresponding system calls. Via smart pointers all objects can refer to other objects in a useful way. For example, this modeling helps to find the process where a PT was delegated to, when it was called. Hence, the calling process of an IPC.

4.2.4 Identifying the Origin of Portal Calls

Each `PtObject` has two important properties in my runtime system: a unique 64-bit long ID and a context object. The ID is used to identify the `PtObject` of a PT call. Hedron passes this ID as first argument to PT handlers. The roottask uses one generic entry for all PT calls. The ID is used to look up the actual `PtObject` in the `ProcessManager`. From that, I can check to which `PdObject` it was delegated to. I assume that a PT is only delegated to at most one other PD and only used by that one. From the context object that is attached to each `PtObject` I can check which specialized IPC handler should be invoked. Listing 4.2 on the next page gives a closer look at this context object.

The generic portal callback accumulates the right mutable UTCB reference, the calling process object, and some other relevant information. It passes all relevant data structures to the specialized IPC handler.

```
pub enum PtCtx {
    Exception(u64),
    Service(ServiceId),
    ForeignSyscall,
}
```

Listing 4.2: The snippet shows the context enum that is attached to each PtObject. The exception-variant carries the number of the exception, for example `0x6` for invalid opcode. If the portal references a service, the context object holds the ID of the requested service (e.g. allocator, stdout). In the case of foreign system calls, the request is forwarded to the OS personality that must figure out which foreign system call was triggered (by using the data provided by the exception data layout in the UTCB.).

4.3 Handle Foreign System Calls

The roottask knows if an application that it starts is a foreign or a native one. When it creates the PD, it passes that information to Hedron. On a foreign system call, the foreign system-call handler within the roottask is invoked by the generic PT entry handler discussed in Section 4.2.4 on the facing page. It checks the `abi`-property[1] of the corresponding `Process` object. The foreign system-call handler then forwards the request to the specialized OS-personality (also inside the roottask). The Linux OS-personality constructs a `GenericLinuxSyscall`-struct from the CPU state stored in the UTCB. This struct reads the register values and helps to access the system call arguments in correct order (as in Table 2.2 on page 33). The Linux OS-personality further process the desired Linux system call. It updates the registers as needed, read and write to user memory as needed, and returns to the generic PT entry function. For example, the result code is stored in register `rax`. Finally, `reply()` is executed by the generic portal entry and Hedron updates the CPU state of the original caller.

Section 9.4 on page 99 lists the supported Linux system calls of the current implementation of my OS personality.

[1] Currently, the implementation only knows Linux as foreign system call ABI. However, my implementation is extendable.

4.4 Hybrid Parts in Foreign Application

Foreign Rust applications can include the libhrstd library with a special Cargo build-time feature that tells the library to use the bindings for foreign applications. Therefore, an application can make Hedron-native system calls and thus become hybrid. The OS personality provides a special environment variable for foreign applications which allows them to detect if they are running under Hedron. Thus, the code of hybrid applications can run under Linux without requesting Hedron functionality but unfold all its functionality under Hedron. Listing 9.4 on page 97 shows an example of such a hybrid application.

The runtime environment itself will never know if a foreign application invokes Hedron-native system calls because this is entirely handled between Hedron and the code running inside the corresponding PD. The userland starts hybrid applications in the same way as foreign applications.

Currently, hybrid applications need to toggle the Native System Call Toggle (NSCT) before and after Hedron-native system calls. A possible simplification is that Hedron resets the NSCT after each system call. This adds a policy to Hedron and therefore I decided against it. However, if this mechanism is developed further until it lands in a product eventually, I see no reason why Hedron should not reset the flag on each system call if it is set. This operation is cheap and it prevents errors in user space caused by forgetting to reset the NSCT. Resetting the flag in Hedron will not increase complexity in kernel space and will not prevent flexibility in user space.

4.5 Communication Path: Native vs Foreign

In this section, I give a short overview about the communication path that native and foreign applications take to access system resources in my current runtime system. This summarizes many design choices and implementation steps. Figures 4.2 to 4.3 on the facing page visualizes the communication paths by native applications and foreign applications through their corresponding default file interface. Both figures show an application that uses the file-system service to write to a file. The first figure shows that for a native application whereas the latter shows it for a foreign Linux application. It is visible that a foreign system call is handled like regular IPC to a PT.

In a real world scenario, the implementation will be split into separate components, i.e., not all services are provided by the roottask. However, to show that my mechanism works, I took this shortcut. Figure 3.1 on page 44 provided earlier presents how the communication path looks like if the file-system services runs in its own PD.

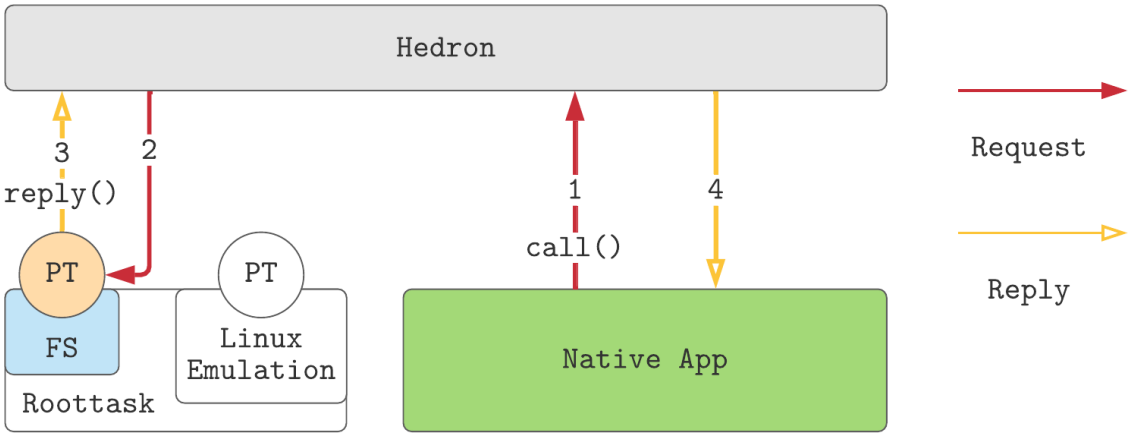

Figure 4.2: Overview of a native application that interacts with the file-system service. The app shown on the right side calls the PT of the file-system (FS) service to write data. If a hybrid application interacts directly with the Hedron runtime system, it will look similar. However, this is not necessarily intended or beneficial from a code point of view but technically possible.

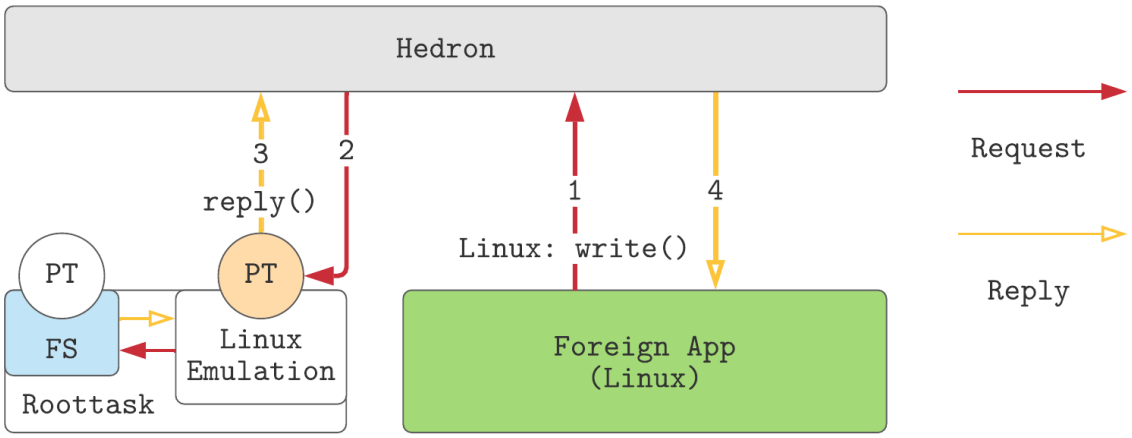

Figure 4.3: Overview of a foreign Linux application that interacts with the file-system service. The app shown on the right side executes a `write()` system call. Hedron recognizes the foreign system call and forwards it to the Linux OS emulation component. The Linux OS personality performs the file descriptor multiplexing, sees that the file descriptor belongs to a real file, and forwards the request to the file-system service.

4.6 Implementation Challenges

In this section, I briefly present selected challenges that I faced during the engineering of this work.

libhrstd is a Rust library intended for native applications and hybrid applications. By activating the right Cargo build-time feature, the library either wraps system calls with the Native System Call Toggle (NSCT) for foreign applications or ignores the toggle for native applications. I point out a programming challenge I have encountered in the engineering of this part.

libhedron exports all system calls as functions. Each system-call function from libhedron returns a Rust `Result<S, E>`-type. Inside libhrstd, I provide wrappers that enable the usage of them in hybrid applications. Let us take a look at Listing 4.3 on the facing page. `wrap_hedron_syscall` is a function that executes the passed system call function (from libhedron) wrapped by toggles of the NSCT. If the (Rust) code from Listing 4.3 on the next page code looked like

```
wrap_hedron_syscall(|| sys_create_pd(/*...*/).unwrap())
```

instead of

```
wrap_hedron_syscall(|| sys_create_pd(/*...*/))).unwrap()
```

(notice the ’)’-parentheses), a failed system call will cause a Rust panic before the NSCT is deactivated again. The reason for this is that Rust’s standard library for Linux handles a panic by writing a message to *stderr* before it finally terminates the program with an `exit_group` system call. Hence, an `.unwrap()` on an `Err` variant results in undefined behaviour and errors for the Linux application. This pitfall can completely be abstracted from application developers with the libhrstd. A user of the public API does not have to cope with this complexity.

```rust
/// Part of a library that the hybrid/foreign app can use.
fn wrap_hedron_syscall<T, R>(actions: T) -> R
where T: Fn() -> R {
    utcb().set_nsct(true);
    // NO foreign syscall from here
    let res = actions();
    // NO foreign syscall until the next line
    utcb().set_nsct(false);
    res
}

/// Part of a library that the hybrid/foreign app can use.
fn hedron_create_pd(parent_sel: u64, cap_sel: u64) -> PdObject {
    // sys_create_pd() comes from libhedron
    wrap_hedron_syscall(
        || sys_create_pd(parent_sel, cap_sel)
    ).unwrap()
}

// Actual foreign application.
fn main() {
    // hybrid part
    let parent_pd_sel = ...;
    let cap_sel = ...;
    let pd_obj = hedron_create_pd(parent_pd_sel, cap_sel);
}
```

Listing 4.3: Code snippet that shows how to enable and disable the Native System Call Toggle (NSCT) before and after native system calls in hybrid applications. This complexity is hidden behind the libhrstd library. For convenience, some library functions are shown inside the same listing.

4.7 Breaking Changes to Hedron API

The changes required for Hedron introduced a breaking API change in the `create_pd()` system call as it was extended with a new argument. All existing applications that use this system call must set the register of the new argument to zero, otherwise Hedron might think the new PD is a foreign PD. This means there is a small engineering effort for existing software necessary.

Although Hedron is open source, Hedron is mainly used by Cyberus Technology. Thus, this code migration is simple to solve if the modifications to Hedron are merged and used with the existing runtime environment. If my new runtime environment is used in the future, it already includes the proper system-call wrappers.

4.8 Summary

In this chapter, I talked about the basic implementation steps I had to take to implement the discussed design in Chapter 3 on page 39. I described how Hedron recognizes foreign system calls and how the runtime environment handles them. Since I have written a new runtime system, I have included additional interesting implementation details as appendix in Section 9.3 on page 97.

5 Evaluation

In this chapter, I explain and evaluate how the proposed changes to Hedron, paired with the new runtime environment, fulfill the goals specified in Section 1.2 on page 23. Furthermore, I examine how my work contributes to Hedron in terms of performance and developer productivity.

5.1 Functionality and Reliability

Figure 5.1 on the next page shows a screenshot of the serial output from the runtime environment executing a Linux application. The application is written in Rust and uses Rust's standard library. It prints its output to the screen by writing to stdout. This is handled by the Linux OS personality, which internally uses the *stdout* service. Thus, the personality connects write operations from the foreign application to the stdout service because it sees that the write operation targets file descriptor 1. The fact that the Linux application comes to this point proves that my runtime system and its Linux personality are capable of executing several Linux system calls. See Table 3.1 on page 53 for a list of system calls that are involved here. For example, the OS personality can perform memory mappings (Linux `mmap()` and `brk()`) requested by the application, and read from user memory (Linux `write()`).

The current implementation of the runtime system enables the successful execution of simple Linux applications. They can request heap memory or talk to the file system with adequate performance. The performance analysis of the file-system interface and the whole foreign system call mechanism follows shortly in Section 5.3 on page 75. Code examples with useful workloads that are supported can be found in Section 9.2 on page 93.

As of now, my implementation is not capable of handling complex Linux system calls, such as *clone* or *futex*. Thus, only a small subset of Linux is available at this time. This is not a limitation of my mechanism for the policy-free system-call layer.

Running a Linux OS personality as a user-space program improves overall liveliness of the system because, in the worst case, a failure affects only all running processes managed by that personality. The security of the whole system does not decrease from my proposed policy-free system call layer, as long as the runtime system follows typical microkernel principles, such as an isolation of dedicated components from each other and the principle of the least privilege.

```
[ INFO] libroottask:src/roottask_exception.rs@91: created local ec for exception ha
[ INFO] roottask_bin:src/main.rs@153: Rust Roottask started
Hello world from Rust!
my args are: [
    "./executable",
    "first",
    "second",
]
my envs are: [
    (
        "FOO",
        "BAR",
    ),
    (
        "LINUX_UNDER_HEDRON",
        "true",
    ),
]
```

Figure 5.1: Screenshot of the output of a Linux application running under Hedron that is written in Rust and uses Rust's standard library. It prints out its arguments and its environment variables. The same unmodified binary also works under Linux similar and produces similar output.

5.2 Developer Productivity

In this section, I discuss how my changes affect developer productivity. Developer productivity is influenced by initial efforts and long-term maintenance costs required to build software. Furthermore, availability of applicable tooling affects developer productivity. Questions arise such as how well does my IDE provide sensible auto-suggestions/completions or how usable is the build system and what similarities to existing standard build environments exist?

5.2.1 Scope

In this consideration, I assume that developers want to use a diverse set of (existing or upcoming) applications and libraries for Hedron and that they aim to keep the projects fresh with the corresponding upstream repositories. Furthermore, I look on writing new software from scratch. I focus on the programming languages C, C++, and Rust as examples.

To achieve this, I compare the approach presented in my work with two alternatives. Alternative approach A is about providing a POSIX compatibility layer (or whatever the default standard library of the foreign system is). It is discussed in Section 5.2.2 on the next page. Afterwards, I present the alternative approach B in Section 5.2.3 on page 71 where I discuss how "non-standard" software is build and what the limitations of this approach are. Finally, Section 5.2.4 on page 74 bridges the gap from the presented alternative approaches A and B to the support of hybrid and unmodified foreign applications presented in my work. The final

comparison examines how my presented design solves the inconveniences described in the upcoming paragraphs.

5.2.2 Approach A: Providing a POSIX Compatibility Layer

Let us assume that developers want to port existing libraries or binaries to Hedron. A possible approach is to provide a custom libc, i.e., POSIX compatibility layer. This also works with other well-defined standard libraries such as win32. In Figure 2.5 on page 37, I showed how a custom libc bridges the gap between an application and an (exchangeable) kernel.

A well-defined standard library layer helps to port existing software to a system such as Hedron without many application code modifications. If a developer uses this approach, the software must still to be linked against a special Hedron library in order to be able to use all functionality of Hedron and its runtime system. An adapted libc is not enough to cover all cases.

Examples for that are the Hedron system calls `pt_ctrl()` and `ec_ctrl()`. There are no suiting libc functions that can be mapped to them. Furthermore, the *libpthread*-concept is mappable to global ECs but not to local ECs. To name one example, code inside a local EC has to restore its stack manually before a `reply()` system call, otherwise the EC will overflow its stack eventually. libpthread does not know a concept to differentiate between different kinds of threads. libpthread is usually tied to the libc implementation because it needs access to mechanisms such as thread local storage [27]. So even if an adapted POSIX compatibility layer is available (including an adapted version of libpthread), local ECs cannot be covered sensibly.

Additionally, there are further restrictions that I already discussed in Section 3.1.5 on page 49. As a brief recap, developers may need to find alternatives for `fork()`.

The statements above are the reason why an additional library for Hedron is required to access all functionality of Hedron and its runtime system. A libc alone is not enough. To include such a library additional steps need to be taken. For C and C++ projects there are two options. For example, the library's header files and the shared object can be bundled in a packaged format, such as a debian package, and be provided for application developers. This package needs to be installed on the developer's system. The content of the package can be referenced as dependency through additional compiler- and linker flags. Another approach to include a Hedron library is to provide the hedron library as header-only library which allows to copy & paste the library into projects easily. This adds additional work for developers. For Rust projects, a Hedron library can be easily included in the `Cargo.toml`[1] file via `crates.io` or through a public git repository.

[1] The project file for Rust projects. It includes meta information and specifies dependencies that may be available on `crates.io` or in public git repositories. Cargo downloads these dependencies automatically.

What functionality will be provided by a custom libc is dependent on the implementation of the runtime environment running under Hedron. However, challenges to include this library into the default build setup, such as local setups and continuous integration, is challenging and needs individual work per project. So, in the following, I discuss how one can achieve that.

Using a custom libc is a toolchain adjustment. One needs to modify the linker (flags) to link against a different shared object. Popular build tools for C or C++ projects are *GNU Automake*[2] and *CMake*[3]. These build tools enforce no policies about how one should build their software project but only provide tooling to compose an individual build system. As a consequence, some projects include header definitions and code side by side and some separate them into dedicated `inc/` and `src/` folders. This is only one example that shows the variety of possibilities to build a C- or C++-based project. As a consequence, porting existing code bases to a new libc is a challenging and individual process. For every update to the upstream repositories that touches the build system, these steps need to be carefully rebased onto the latest upstream changes, which is error-prone.

For Rust this is a less inconvenient because Rust projects use Cargo as its default build tool. Cargo makes strong assumptions about the software project structure and unifies how additional changes to the build setup, such as special linker flags, can be provided. This simplifies using "non-standard" build targets, such as when using a custom libc.

The difference between Rust and C/C++ regarding dependencies is that Cargo can download dependencies by itself whereas they need to be installed manually by the systems package manager to be recognized by typical CMake-setups, for example. Another approach is that the libraries source code is copied into the project. This adds additional steps for local setups and continuous integration, which leads to inconvenience.

To link Rust code that uses Rust's standard library against a custom libc, a custom compiler target must be defined that links to the specific libc implementation. This is simple with Cargo because the compiler target specification is modeled by a JSON-file. This target definition is passed to Cargo through an additional parameter.

The existing Rust standard library with its bindings for Linux, i.e., libc bindings, can be reused in that case without changes. This works because the implementation of the Rust standard library maps its functions to a lower level system library, such as the libc.[4]

[2] `https://www.gnu.org/software/automake/`

[3] `https://cmake.org/`

[4] Background: If the compiler target specification specifies the field "os" as "linux", Cargo will take all compile-time feature branches inside the standard library that conditionally depend on Linux during the compilation of the binary. Thus, the existing Linux bindings for the Rust standard library to libc functions can be reused and Rust code uses the custom libc instead. `https://github.com/rust-lang/rust/blob/1.59.0/compiler/rustc_target/src/spec/linux_base.rs#L5`

For C and C++ projects the library files need to be provided manually and changes to the CMake- or Make-setup are required. The latter is an individual process per software project whereas it is roughly similar for multiple Rust projects.

All these custom steps require toolchain changes that need to be maintained. For Rust these are easy to manage and the necessary steps per software project stay the same. For C and C++ this is an individual maintenance process per project that is hard to maintain.

In this section, I discussed how a custom libc affects application developer productivity. In the following, I discuss how the experience for application developers looks like when no standard library is used but instead raw/pure "non-standard" software is produced.

5.2.3 Approach B: Developing "Non-Standard" Software

A "non-standard" build setup does not include the system's default standard library. In a Rust crate[5] a `#![no_std]`[6] attribute tells that the application should not be linked against `libstd` but `libcore` instead. Cargo recognizes this during builds. Such a crate does not link against libc when compiled on Linux and only allows functions operation on pure data. There are no interfaces to the outer world without additional library functions.

With C and C++ a "non-standard" build setup is in the responsibility of the build tools/the build system and not defined as an attribute in source code, as in Rust. These build setups are discussed in the following.

Changes to Build Tools (Toolchain Changes)

Rust uses Cargo as its default build tool. For `#![no_std]` libraries the Cargo setup needs no changes. For `#![no_std]` binaries Cargo needs additional configuration to use a custom linker script and, depending on the target, possibly a custom Cargo compiler target definition. There are no more steps involved. If we look at C and C++ this is different. C and C++ projects using *gcc*[7] for example need the additional compiler flags `-nostdinc`, `-nostdlib`, and `-ffreestanding`. Makefiles or CMake-files achieve that by specifying additional flags for the compiler. This itself is not standardized and there are multiple ways to do it. For example, the compiler flags can be assembled in multiple steps or at once. They can depend on certain conditions. For example, a project can produce a convenient library based on libc and a lightweight "non-standard" version without libc. These changes to the build tools, i.e., to provide special flags for the compiler, are changes to the

[5] A Rust library or binary.

[6] `#![no_std]` is a crate-level attribute that indicates that the crate will link to the core-crate instead of the std-crate. The libcore crate in turn is a platform-agnostic subset of the std crate which makes no assumptions about the system the program will run on. – `https://docs.rust-embedded.org/book/intro/no-std.html`

[7] `https://gcc.gnu.org/`

default toolchains. Experience shows that this is hard to maintain over time for multiple projects.

Initial and Ongoing Maintenance Costs

In this section, I discuss initial and ongoing maintenance costs of writing new "non-standard" software and porting existing software to be "non-standard"-compatible. I also discuss keeping ported software up-to-date with the corresponding upstream repositories.

In the discussion about the build tools and toolchain adjustments, I said that a Rust crate needs the `#![no_std]`-attribute. This applies for libraries. However, for `#![no_std]`-binaries you also need to configure Cargo to use a custom linker script, specify the `#![no_main]`[8] attribute, and provide a custom compiler target definition. Experience shows that this process is rather simple compared to the C and C++ world (will be discussed shortly). The steps mentioned above are one-time costs.

Long-time costs emerge when existing software for a standard target, i.e., one that relies on the standard library, is ported. Every new functionality in upstream repositories needs to be ported carefully. Calls to the standard library that, may talk to libc, need to be removed and replaced with calls to a Hedron library.

For C and C++ projects the long-term efforts mostly face the same challenges as for Rust. At least, if only new code is added. When the build system is modified in the upstream repositories then the costs to keep the altered version for Hedron up to date are higher. As I already discussed, each C and C++ projects de-facto have an individual build system which makes it hard to keep them up to date. The efforts and costs are individually for each project that needs to be ported.

Additional initial costs arise from new developers not being familiar with a Hedron library that provides a different interface than libc. Developers need to maintain a mental model about the runtime system and its runtime services and a situational awareness how specific behaviour may cause errors during runtime.

Convenience Through Available Tooling

The most important tool for a developer is the editor respectively the IDE. Examples are *Vim*[9], *Visual Studio Code*[10], and *CLion*[11]. A IDE helps with auto-suggestions that fit into the context one is just typing. Hence, the IDE completes the code where applicable to improve developer experience. In this process it takes available functions and modules from the standard library and included external libraries into account.

[8] Attribute that prevents Rust from emitting a "main" symbol. Otherwise, Rust expects a a function called `main()`.

[9] `https://www.vim.org/`

[10] `https://code.visualstudio.com/`

[11] `https://www.jetbrains.com/clion/`

To improve this, IDEs must have knowledge about the type of software one is writing. This is the reason why there are specialized editors and IDEs for Android app development, web technology development, or low level systems programming. There are also generic editors and IDEs that achieve that by providing specialized plugins. In the following, I focus on CLion to discuss developer convenience with the help of personal experiences I made in the past and during the work on this thesis. However, the implications I name in the following are similar for other tooling. CLion offers built-in support for C and C++ and offers support for Rust with the *IntelliJ Rust*[12] plugin.

At first, I discuss the convenience of CLion + IntelliJ Rust for "non-standard" Rust projects and afterwards the convenience of "non-standard" C/C++ projects with CLion. A Rust projects is classified through its `Cargo.toml` file that can be easily detected by an IDE. It is trivial to detect the `#![no_std]`-attribute in the Crate's main file (`lib.rs` or `main.rs`). If IntelliJ Rust detects it, CLion completes my code with imports from `core::*` or `alloc::*` instead of `std::*` - as expected and desired.

My runtime environment consists of Hedron-native, i.e., "non-standard", libraries and binaries, such as the roottask. Thus, this project itself is affected by a lack of developer experience resulting from "non-standard" targets. During my work on this project however, I regularly experienced situations in which `CLion` included types from `std::` instead of `core::`, even through it works sometimes. I experienced this as negative because I had to fix the IDE that is supposed to help me. Rust's design and ecosystem enables IDEs an easy detection for the absence of the standard library. This works well in most situations but not always.

I experienced that one is limited as a developer without the standard library, and I had to provide many functionality by myself that are included by default otherwise. Examples are an allocator or a way to print text to the outer world.

The experience for a "non-standard" build setup with C and C++ is different. My runtime system itself does not use "non-standard" components written in C or C++ but I discuss how the convenience looks like with them. As I already mentioned above in the discussion about build systems, effectively each C and C++ project has an individually customized build system. There is no standardized way to mark a binary or library as "non-standard" in the C/C++ world. This makes it hard for IDEs to discover the absence of the standard library. Instead, IDEs tend to complete functions such as `open()` and add "`#include <fcntl.h>`" on the fly.

In a minimal C++ demo project in CLion to verify developer convenience, I added the line "`add_compile_options(-nostdinc -ffreestanding -nostdlib)`" to a `CMakeLists.txt`-file. CLion did not further suggest including header files from the standard library and also did not complete functions from there. However, this setup is minimal and far from the complexity of real world scenarios. As soon as this statement is wrapped with an if-condition in the CMake-configuration that

[12] `https://www.jetbrains.com/rust/`

Clion cannot trivially resolve, the convenience is gone and the developer experience decreased.

In a minimal C++ demo project in CLion based on a trivial Makefile CLion was not capable of recognizing the absence of the standard library. Thus, developers do not face a convenient programming environment in such a case.

5.2.4 Comparison to My Presented Work

I assume approach A as suitable for porting existing libraries and binaries to Hedron whereas approach A and B are suited for new software. Approach B also works for existing software but the need to replace all libc calls with calls to a Hedron library is huge.

Above I discussed two approaches how software for Hedron can be build when the solution presented by my thesis would not exist. Approach A discussed the developer experience for building software with a custom POSIX compatibility layer, i.e., a custom libc, whereas Approach B discussed the use of "non-standard" build setups without a standard library. Afterwards, I deduced implications to developer convenience and maintenance costs from them. In the following, I discuss how the mechanism introduced by my thesis avoids the problems I discussed above.

One objective of my work is to enable unmodified foreign applications and avoid toolchain adjustments for application development. With my solution, application developers do not need any additional effort to enable their foreign application under Hedron. If they need access to native Hedron system calls, application developers have to link against a Hedron library additionally. For this, they need to learn about the facilities of Hedron's runtime system. Application developers can focus on the actual software and not on the build process. At least, they do not have to focus more on the build process that is beyond the typical setup for standard binaries. However, developers need to include an additional library to enable hybrid code inside their foreign applications.

The discussion above is not only about Rust vs C/C++. Rust has a standardized way for "non-standard" software witch is a major benefit. However, many projects that exist for decades are based on C and C++. It is not applicable to rewrite everything in Rust. This will not solve the problems of "non-standard" builds effectively because the rewrite to Rust itself is an immense effort. Furthermore, existing C and C++-projects target more Hardware-platforms because they use gcc instead of LLVM.

Building new "non-standard" software with Rust, which either includes no standard library or a custom standard library, has major benefits and simplifications compared to C and C++-based setups. However, the simplest solution for application developers is to use the default toolchains as they are. My work enables this successfully.

5.3 Performance

On microkernel-based systems, the performance of applications that require interaction with other system components is influenced by IPC costs. If no shared memory is used but instead Hedron's message-passing mechanism, context switches are required. These context switches include cheap PD-internal context switches and expensive cross-PD context switches. The first only require to update the stack and the instruction pointer whereas the latter additionally requires an address-space switch. With each IPC operation the total IPC costs grow. These IPC operations therefore affect the performance of workloads that use them frequently. One example is a foreign Linux application whose system calls are forwarded via IPC to a mediator and might require additional IPC operations to communicate with the corresponding OS service.

This section studies the overhead of foreign system calls and analyses the impact of IPC on its performance. All benchmarks were executed with 10,000 warmup rounds and 100,000 measurement rounds. The measurement numbers in the diagrams and show the average number of clock ticks during the execution of a workload. I executed each benchmark multiple times and visualize the variance for each measurement inside the diagrams along with the data.

Each binary, i.e., Hedron, the roottask, and user applications, was built in release mode and was compiled for the *IvyBridge* micro-architecture. All measurements were made on an i7-1165G7 *TigerLake* Intel CPU. The host operating system is Ubuntu 20.04 with a Linux 5.13 kernel. The Hedron measurements were taken inside *QEMU* at version 6.2 running on the host. QEMU was invoked with the `-cpu=host` parameter, thus runs directly on the CPU with activated hardware virtualization features enabled. The Linux measurements were taken inside the same QEMU setup but with a virtualized Ubuntu 20.04 with a Linux 5.13 kernel. Thus, overhead caused by nested paging is relative and affects both measurements in the same way.

5.3.1 Pure System-Call Performance

In this section, I discuss the raw system call performance of Hedron. I measured the costs of the `sysenter`, the callback inside the kernel, and the succeeding `sysret`. I used a cheap `pt_ctrl()` system call for that. Figure 5.2 shows the measured system-call costs from a native application and a hybrid application. Both are running under Hedron. Since the check for the Native System Call Toggle (NSCT), which was introduced in Section 3.2.1 on page 51, adds a branch instruction, one can see a minor growth in the system-call costs. For comparison with Linux, I included the costs of a cheap `set_tid_address()` Linux system call under native Linux which you can see on the right.

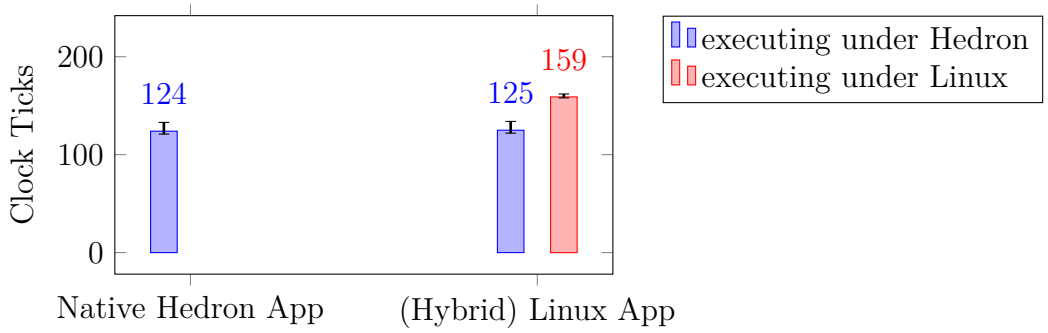

Figure 5.2: The figure shows an overview of the pure system-call costs from two binaries that execute Hedron-native system calls. Under Hedron the cheap `pt_ctrl()` system call was used to test the pure system call handler costs. For comparison to Linux, the right side shows the costs of a cheap `set_tid_address()` system call under Linux.

5.3.2 PD-internal and Cross-PD IPC Performance

As described earlier, an important factor influencing the performance of a microkernel-based system are the IPC costs. Therefore, it is reasonable to assume that all measurements that include IPC, such as the transmission of foreign system calls to a user-space component, are related to these numbers. Figure 5.3 on the facing page shows the costs of several IPC variants. I made four measurements for the properties {Cross-PD IPC, PD-internal IPC}×{Raw, Regular}. By “raw” I mean the pure Round-Trip Time (RTT) without any actions by the userland except for an immediate `reply()`. These are the pure costs of IPC accountable to Hedron and the hardware. By “regular” I mean IPC where the control flow goes through the Portal (PT) multiplexing mechanism introduced in Section 4.2.4 on page 60. You can see in Figure 5.3 on the facing page that the need to look into several data structures adds an overhead. The increase from “raw” to “regular” takes roughly 25% more clock ticks. Cross-Protection Domain IPC is 50% more expensive compared to PD-internal IPC because of the address-space switch. My

discussed modifications to Hedron in Section 4.1 on page 57 do not affect these numbers.

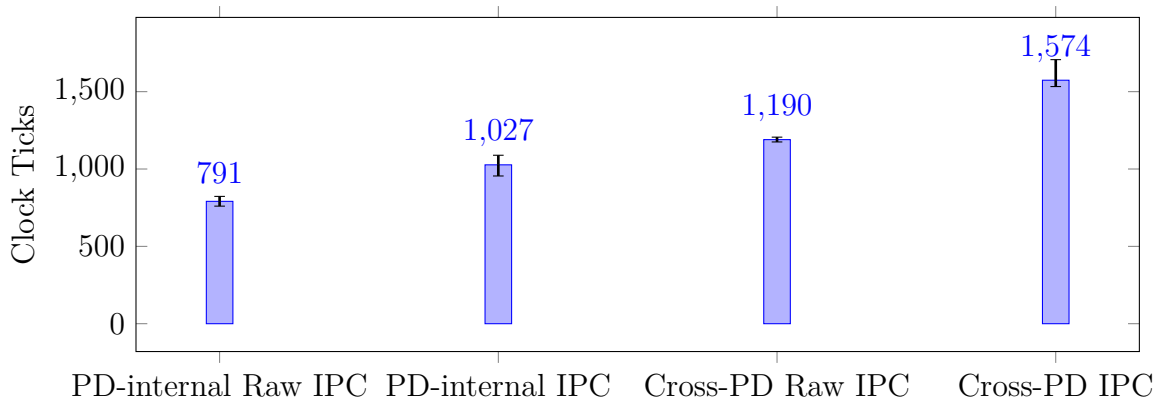

Figure 5.3: Overview of IPC costs in my runtime system. I measured the Round-Trip Time (RTT) of a `call()` system call with a subsequent `reply()` system call. Cross-PD IPC is roughly 50% more expensive than PD-internal IPC because the address space needs to be switched. By "raw" IPC I mean the pure cost of the system calls combined with Hedron's context switches without any additional actions of my runtime system. The non-"raw" IPC measurements includes the path through my Portal (PT) multiplexing mechanism described in Section 4.2.4 on page 60 that add an overhead of roughly 25%.

5.3.3 Foreign System-Call Performance

In this section, I show several metrics regarding foreign system calls. The costs of a foreign system call includes the summed times of a native system call, the duration of the IPC, and the actual emulation of the OS personality. I looked at three different Linux system calls with varying complexity. `set_tid_address()` updates a property inside the process-management structure inside Linux. `fstat()` fetches information about a file from the in-memory file system (`/tmp` in Linux). The `open()` system call in this example is used to open an existing file (inside `/tmp` in Linux). Figure 5.4 summarizes my observations. As expected, Linux is the fastest because the control flow path for system calls is shorter in a monolithic system. The difference decreases for expensive system calls. Linux is more than ten times faster for a cheap system call but only 60% faster for the `open()` call. Since my implementation includes all services inside the roottask as described in Section 4.2 on page 58, I simulated the costs of a mediator library as described in Section 3.1.4 on page 47 additionally. The diagram show that influence side-by-side.

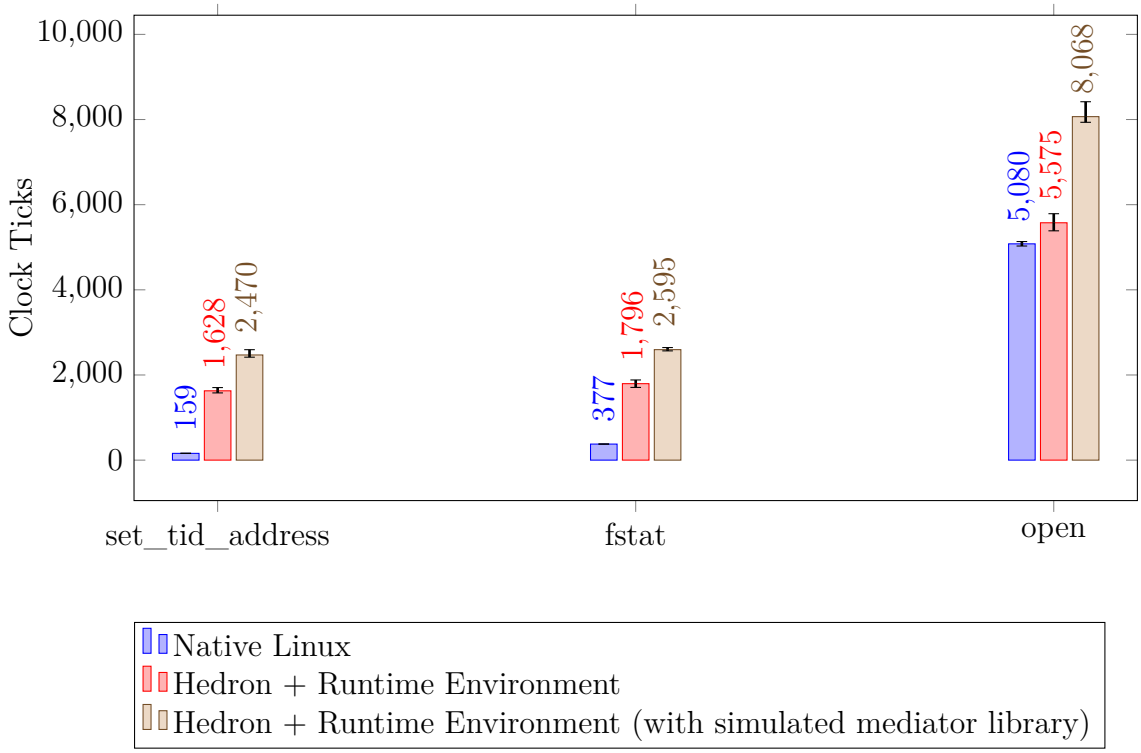

Figure 5.4: Overview of foreign system-call costs from a Linux application. The additional costs of foreign system calls caused by IPC costs are presented: the bars showing the time under Hedron are noticeable higher, which means the execution time is longer.

5.3.4 File-System Microbenchmark

After I showed several basic performance metrics above, I now present results of a microbenchmark against the file system. Under Linux the benchmark uses the `/tmp` file system. Under my runtime environment it uses the default file-system implementation that lives in memory.

The benchmark measures `read()` and `write()` operations on native Linux and on my runtime system under Hedron. It reads and writes a file and executes this multiple times to calculate the average costs. Read and write operations are measured separately. Furthermore, it checks how a growing buffer size for the read and write operations influences the overall costs per read or write. The results are given in Figures 5.5 to 5.6 on pages 80–81. All results include a simulated call to the mediator library as described in Section 3.1.4 on page 47. Furthermore, I simulated a dedicated page-aligned receive and send window between the Linux OS personality and the file-system service. In a real-world scenario it is sensible to build a setup like this because otherwise the file-system service might obtain access to the stack of the Linux application or other memory regions[13] I executed the benchmark with file sizes of 64 KiB and 1 MiB.

From the results, we can observe that the implementations of my Linux OS personality and my in-memory file system are competitive with Linux for large file sizes and large buffers. For small workloads the overhead of a system call has a large share regarding the total time of the operation. If the share of the overhead of the system calls decreases, which is the case for large buffer sizes on large file sizes, then my runtime system is equally fast or even outperforms Linux. The write performance is constantly better than under Linux. The read performance is constantly slower but the performance approaches that of Linux as the buffer size increases for large files.

The reason for the better write performance under my runtime system is that the control flow through my Linux operating system personality models only a small part of what Linux is capable of. It can be expected that if the complexity of the Linux OS personality grows over time and control-flow operations increase this lead against Linux decreases. Furthermore, the implementation of my in-memory runtime system is not a full UNIX-like file system but a `HashMap` in memory. This further improves the lookup speed of certain structures of the in-memory file-system implementation. Another aspect might be that the allocation mechanism of my file system performs faster. However, the focus of this benchmark is not the comparison of file-system implementations but only the mechanism to access files and transfer data.

[13] Memory mappings can only be given at the granularity of a page. However, it might happen that the pointer of a Linux write or read system call points into the middle of the heap or the stack. In such a case, a malicious service can manipulate the behaviour of a Linux app. Thus, a dedicated and page aligned send and receive window is a solution to that. In this case, only the Linux personality needs to be trusted.

File-System Microbenchmark With a File Size of 64 KiB

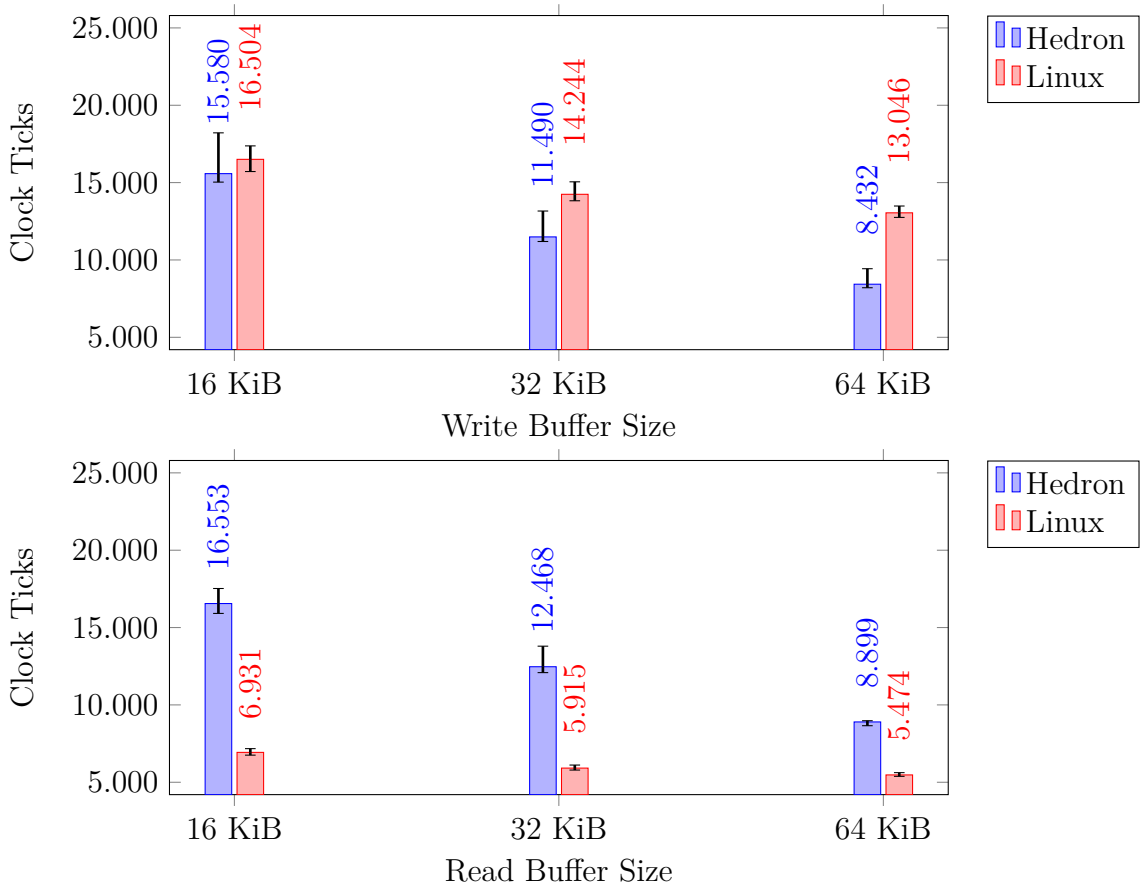

Figure 5.5: File-system microbenchmark against the in-memory file system with a file size of 64 KiB. The diagram shows the costs of the write operations on the top and the costs of the read operations on the bottom. A lower bar means a shorter execution time and thus a higher performance/throughput. The costs are influenced by the amount of read/write system calls required to transfer the whole file size. The buffer size influences the amount of required system calls. Hedron's runtime system outperforms Linux for write operations in any case. Read operations are slower but the difference decreased with a growing buffer size.

File-System Microbenchmark With a File Size of 1 MiB

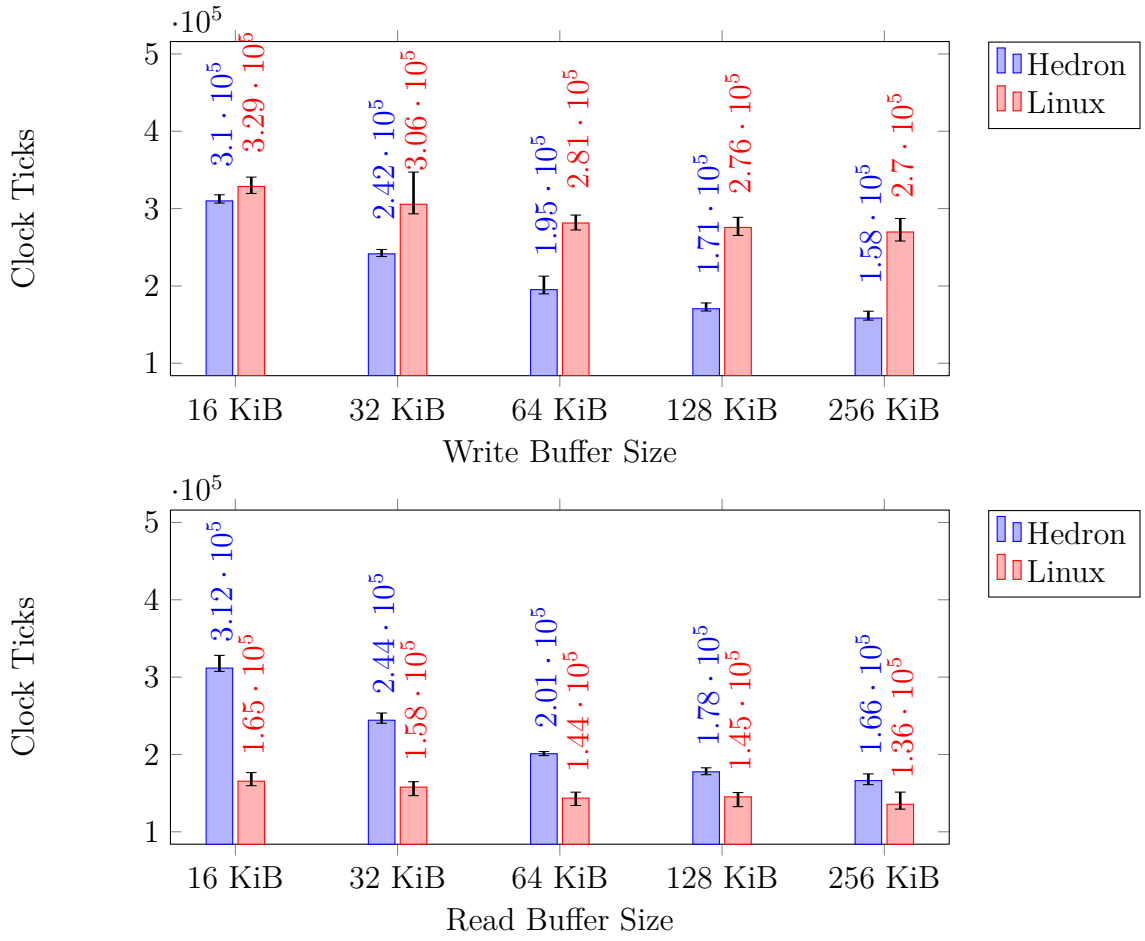

Figure 5.6: File-system microbenchmark against the in-memory file system with a file size of 1 MiB. The diagram shows the costs of the write operations on the top and the costs of the read operations on the bottom. A lower bar means a shorter execution time and thus a higher performance/throughput. The costs are influenced by the amount of read/write system calls required to transfer the whole file size. The buffer size influences the amount of required system calls. Hedron's runtime system outperforms Linux for write operations in any case. Read operations are constantly slower but read performance approaches that of Linux as the buffer size increases for large files.

5.4 Summary

In this chapter, I evaluated my work against the objectives I set in the introduction. It shows that the developer productivity is increased by enabling hybrid applications with Linux as example under Hedron. I showed that IPC costs negatively influences the performance of foreign system calls compared to native Linux. If the call itself is expensive, the overhead of the mechanism only has a small share. With a microbenchmark against the file system I presented that performance is competitive with native Linux in read and write operations for large file sizes and large buffer sizes. To be precise, read operations are close to the performance of Linux if the buffer sizes is higher than 256 KiB. Write operations are constantly faster but this might change with a more complex file-system implementation.

6 Related Work

This thesis covers the area of software reuse including reuse of existing toolchains to keep up a high developer productivity. A variety of external research groups, companies, and individuals came up with different solutions for reuse of software stacks, applications, and whole operating systems in the last decades. Their solutions differ in details and specific goals while all aim for a common higher objective. This chapter reviews those designs and how they relate to my thesis.

6.1 VM-based Software Reuse

A prominent example for software reuse is the use of VMs which are mainly relevant in scenarios where multiple tenants share a common infrastructure. This is useful when a provider offers infrastructure where multiple tenants share the resources of multiple compute nodes for an optimal resource utilization regarding energy efficiency, hardware resource minimizing, and good performance. The tenants should not modify their software in any way and see a world where they are the only software running on a system. Virtualization enables a strict isolation between multiple guest operating systems. Para-virtualization can be used to accelerate the VM. Furthermore, virtualization can be accelerated with hardware-accelerated virtualization features, such as *Intel VT* [17]. Hardware-assisted virtualization improves performance but targets the same level of abstraction.

A virtualized environment tricks a software into thinking it owns the full hardware. Thus, it can bootstrap the system and run its own applications. There is no functional interaction with other guests, hence, every tenant is strictly isolated. VMs reach binary compatibility by reusing the original kernel in a virtualized environment.

6.1.1 Reuse Original Operating System

L4Linux [18, 26] (2001) and *Xen* [2] (2003) are examples for para-virtualization solutions. Taking Linux as an example, both solutions modified Linux to run on a virtual architecture, i.e., on virtual hardware. This virtual architecture uses interfaces provided by either Xen or the L4 runtime environment (*L4Re*). In the case of L4Linux, the virtual hardware maps to system services of L4Re [19].

It is quite difficult to achieve a full functional integration into the existing runtime environment. Application under L4Linux run on a vCPU. One vCPU executes the code of Linux and its user applications [21]. If a user application performs an IPC call to the outside L4Re-world, the vCPU blocks and thus L4Linux is blocked, which is undesirable. We can solve this with decoupling [22]. Both solutions, Xen and L4Linux, reach binary compatibility through reuse of the original kernel that needs modifications for the para-virtualization.

X-Containers [33] (2019) follows an approach called *Library OS* (libOS). The host runs a small exokernel whereas the guests applications have a libOS in their address space. The approach requires no hardware virtualization and reaches binary compatibility through the corresponding libOS. Specifically, they reuse Linux's existing Xen para-virtualization support. Thus, Linux can run without the need for privileged mode in user space and be linked as libOS into the container. They patch binaries so that system calls are replaced by function calls into the corresponding libOS. Technically, X-Containers reaches binary compatibility the same way as Xen and L4Linux but the applications are executed with a slightly different model.

The Windows Subsystem for Linux (WSL) [4, 35] is an approach that is more similar to my work. In WSL 1 Linux is emulated whereas WSL 2 reaches binary compatibility with a modified para-virtualized Linux kernel [6]. The WSL (in both versions) aims for a functional integration into the existing runtime environment. This means, existing and unmodified Linux application can execute and access for example the same file system. This allows the usage of tools such as the *GNU Compiler Collection* on Microsoft Windows while the code editor might run as native Microsoft Windows application. Both can operate on the same directory structure. This is the same functional goal that my work targets.

My work reaches binary compatibility by providing a custom OS emulation layer instead of using para-virtualization with a modified guest kernel. This is in contrast to the approaches listed above. Whereas X-Containers, Xen, and L4Linux do not aim for a full functional integration, the WSL partially enables this with for example the file system. From a functional point of view, the latter enables a similar user and developer experience as my work.

6.1.2 Provide Forward Kernel

ELK Herder [29] virtualizes an application with a technique that can be described as compute kernel or forward kernel (see Section 3.1 on page 39). The main objective of ELK Herder is to enable fault tolerance by replicating Linux runtime processes. With ELK Herder a program can run virtualized multiple times and these instances execute simultaneously. Integrity is guaranteed at certain checkpoints across all running instances. In each VM, a tiny forward kernel forwards all requests to the host Linux.

The main objective is to discover hardware-faults. Binary compatibility is reached because all system calls from the VMs are forwarded to the host kernel.

However, this is not an approach that enables reuse of existing software from other operating system and only targets Linux.

Another example with a forward kernel approach but different goals than ELK Herder is gVisor [16]. gVisor is a solution that combines the two worlds of VMs and containers. VMs have a large overhead because they require a dedicated kernel. Containers are faster but the isolation is no longer done by the hardware but the software. gVisor tries to build the bridge between these two worlds. It is a forward kernel (they call it application kernel) that implements some Linux system calls by itself whereas others are forwarded to the host. This concept is a trade-off between maintainability costs, hardware-assisted isolation, low overhead, and performance. System call intensive workloads on gVisor for example come with a 50% overhead in system call throughput. These costs originate from expensive VM exits and reentries.

gVisor also aims for isolation from the existing runtime services to allow multiple tenants on the same hardware while achieving a lower overhead than with traditional VMs. It also enables benefits of containerized environments. gVisor reaches binary compatibility by its mixture of the small application kernel, which implements some system calls by itself, and the forwarding of several system calls to the host Linux. gVisor only works for Linux which is in contrast to my work that aims to support all foreign OSs.

6.2 System-Call Interception/Emulation

Fuchsia with starnix [30] (2021) takes an approach that shares the most similarities with my work. Although Zircon, the kernel of Fuchsia, applies many of the concepts popularized by microkernels, it does not strive to be minimal. For example, it includes multiple dozens of system calls [43]. Instead, the architecture of Zircon enables Fuchsia to reduce the amount of trusted code running in the system to a few core functions. A RFC from 2021 proposes changes to Fuchsia that allows the execution of unmodified Linux binaries. They plan to intercept system calls the same way as I discussed in Section 3.1 on page 39 inside the kernel. If a foreign system call is recognized, an exception IPC is send to an IPC endpoint. The binary compatibility is reached by implementing an OS personality called starnix. starnix fulfills the role of the mediator which I discuss in Section 3.1.4 on page 46. In my implementation, the roottask takes the role of that mediator. The RFC does not discuss the possible optimization I proposed in Section 3.1.4 on page 47 but technically it would be applicable with their architecture.

Zicron shares similarities to Hedron. For example, there exist an exception for the startup of processes [10]. Thus, starnix handles the startup of processes to set up the initial CPU state similar to the OS personality in my design.

The WSL in version 1 also intercepts and emulates Linux system calls [4, 35]. Microsoft uses a dedicated Linux interface in front of their kernel. This interface

translates Linux system calls to Windows NT system calls. Thus, unlike in WSL2 (discussed above), the support for system calls is limited.

6.3 Visual Comparison

Table 6.1 categorizes the related work projects I discussed above regarding certain properties. Each of the solutions mentioned above have in common that they do not require toolchain modifications for existing software. They are similar to my work in that regard. The existing differences are shown in the table below:

	Functional Integration	**(Para-) virtualization**	**First-Class Citizen**	**Foreign Apps**	**Hybrid Apps**
classic VMs		✓		✓	
WSL 1	✓		✓	✓	
WSL 2	✓	✓		✓	
Xen		✓		✓	
L4Linux		✓		✓	
X-Containers		✓		✓	
gVisor	✓		✓		
ELK Herder	✓		✓		
Fuchsia with starnix	✓		✓	✓	
Hedron with hybrid apps [This Thesis]	✓		✓	✓	✓

Table 6.1: Comparison between existing solutions and my thesis regarding several characteristics.

By *Functional Integration*, I mean that the execution model allows the interaction with existing runtime components, such as the same files. *(Para-)virtualization*, I refer to designs that use virtualization features to execute programs. By *First-Class Citizens*, I refer to designs that execute programs side-by-side with regular ones. By *Foreign Apps*, I classify solutions regarding if they allow the execution of software programs different to the ones of the host platform. By *Hybrid Apps*, I refer to the definition provided in Section 1.2 on page 23 where my work is to my best knowledge the only existing solution in the field of microkernels.

7 Future Work

In Chapter 3 on page 39, I propose a design where a Protection Domain (PD) knows whether it is "foreign" or "native". The control flow for system calls from foreign PDs works as shown in Figure 3.3 on page 50. To allow hybrid applications Hedron checks the kind of the PD and uses the Native System Call Toggle (NSCT) to distinguish native from foreign system calls.

In a next step, I would revoke the changes to the PD object described in Section 4.1 on page 57 and only use the NSCT for all PDs. When the runtime system starts a Hedron-native application then the flag defaults to true. Otherwise, when a foreign application is started, the flag should default to false. If a system call is recognized and the flag indicates false then a *Foreign Syscall Exception* must be triggered.

Right now, Hedron knows 32 different exceptions for regular PDs. Usually, these are the regular exceptions from the hardware with their corresponding offset with two deviations. Exception index `NUM_EXC - 1` and exception index `NUM_EXC - 2` are used for special Hedron exceptions namely *Recall Exception* and *Startup Exception*. I suggest adding the new special Foreign Syscall Exception at index `NUM_EXC - 3`. Enhancing the existing exception mechanism unifies it with my proposed design and reduces necessary code changes in Hedron.

In Section 3.3.3 on page 54, I discussed signals for Linux applications and outlined possible approaches. However, signals are out of scope of this work. It is future work to find mechanism for Hedron that allow the unblocking of blocked resources, such as ECs, and a clean release of resources that are no longer in use.

8 Summary and Conclusion

In this chapter, I briefly summarize the goals and major objectives of my thesis and discuss how well I achieved them and what problems, if any, occur and why.

In Section 1.2 on page 23, I presented the goals of this work. My task was to create a policy-free system-call layer for Hedron that allows competing policies in user space. This mechanism should enable the execution of unmodified foreign binaries with Linux as example. Additionally, hybrid applications should be supported.

In Section 3.1 on page 39, I discussed several strategies how to reach binary compatibility for foreign applications. I decided to use an approach that enables foreign applications as first-class citizens. This means that they run side-by-side with native Hedron applications. It is in Hedron's responsibility to catch foreign system calls and deliver them to a user-space destination. By reusing existing code inside Hedron and only minor additions, Hedron can discover if a foreign application made a foreign system call and forward it to a user space component, i.e., the OS personality, as exception IPC. This enables the transfer of the CPU state. If the OS personality replies that request it can alter the CPU state of the caller.

Afterwards, in Section 3.2 on page 49, I discussed strategies to allow native system calls from foreign applications, i.e., hybrid applications, that are compatible to the mechanism for foreign applications. I introduced the Native System Call Toggle (NSCT) flag inside the UTCB header which the hybrid application may use to mark the next system call as native one.

Chapter 4 on page 57 presented implementation details of my proposed design. Finally, in Chapter 5 on page 67 I showed that my implementation that follows my design choices enables the successful execution of several simple Linux binaries. These binaries may be hybrid. I showed that the costs of foreign system calls are noticeable and caused by IPC costs. However, if the system call itself is expensive, such as a large write operation, this overhead is small and my runtime system can even outperform Linux in several operations.

The source code of my work is provided in the appendix in Section 9.1 on page 93.

9 Appendix

9.1 Source Code of Main Contributions

The main contributions of my thesis, i.e., the modifications to Hedron and my runtime environment, are publicly available on GitHub. The corresponding README files guide interested readers to successfully build and run everything.

Modifications to Hedron

`https://github.com/phip1611/hedron`

This repository contains a fork of Hedron that includes relevant changes for my policy-free system-call layer.

Runtime Environment & Run Scripts

`https://github.com/phip1611/diplomarbeit-impl`

This repository contains my runtime environment and references my Hedron-fork as git submodule. It contains several code examples and a convenient setup to test and play around. The example code can bootstrap several native, hybrid, and foreign applications. Please refer to the README of those projects.

9.2 Code Examples

This section shows multiple code examples that give the reader an idea about what kind of foreign and hybrid applications can be executed with the introduced foreign system-call mechanism for Hedron in Chapter 3 on page 39 and the corresponding policies implemented in user space described in Chapter 4 on page 57. As my current implementation only supports static Linux binaries, the code examples must be linked statically (for example against the musl library). However, this is a limitation of my runtime environment and not a limitation of the proposed policy-free system-call layer.

```rust
use std::env::args;
use std::f64::consts::PI;

// This binary takes the first argument as radius and calculates
// area and circumference of a circle. If no argument is
// provided it falls back to 42.
fn main() {
    println!("Hello World from a Rust App compiled for Linux");
    let mut args = args().skip(1);
    let radius = args
        .next()
        .map(|x| x.parse::<f64>().ok())
        .flatten()
        .unwrap_or(42.0);
    println!("Circle");
    println!("  Radius       ={:6.2}cm", radius);
    println!("  Area         ={:6.2}cm²", PI * radius.powi(2));
    println!("  Circumference={:6.2}cm", 2.0 * PI * radius);

    let args = std::env::args().collect::<Vec<_>>();
    println!("my args are: {:#?}", args);
    let envs = std::env::vars().collect::<Vec<_>>();
    println!("my envs are: {:#?}", envs);
}
```

Listing 9.1: Linux application written in Rust to calculate certain metrics of a circle. It uses Rusts standard library to parse the program arguments and perform calculation based on them.

```c
#include <stdio.h>
#include <stdlib.h>
#include <inttypes.h>

const uint32_t DIM = 3;
int main() {
    // Two 3*3 matrices on the Heap with values: 0, 1, 2,...
    uint32_t * matrix1 = malloc(sizeof(uint32_t) * DIM * DIM);
    uint32_t * matrix2 = malloc(sizeof(uint32_t) * DIM * DIM);
    for (uint32_t i = 0; i < DIM; i++) {
        for (uint32_t j = 0; j < DIM; j++) {
            uint32_t num = i * DIM + j;
            matrix1[num] = num;
            matrix2[num] = num;
        }
    }

    // Target matrix on Heap; matrix multiplication
    uint32_t * res_matrix = malloc(sizeof(uint32_t) * DIM * DIM);
    for (int i = 0; i < DIM; i++) {
        for (int j = 0; j < DIM; j++) {
            uint32_t sum = 0;
            for (int k = 0; k < DIM; k++) {
                sum += matrix1[i * DIM + k] * matrix2[k * DIM + j];
            }
            res_matrix[i * DIM + j] = sum;
        }
    }

    // print whole matrix to screen
    printf("[\n");
    for (int i = 0; i < DIM; i++) {
        printf("  [");
        for (int j = 0; j < DIM; j++) {
            printf("%u,", res_matrix[i * DIM + j]);
        }
        printf("]\n");
    }
    printf("]\n");
    free(matrix1);
    free(matrix2);
    free(res_matrix);
}
```

Listing 9.2: A program written in C that multiplies two matrices and prints the result to the screen.

```rust
use std::fs::OpenOptions;
use std::io::{Read, Seek, SeekFrom, Write};

// Simple file operations using Rusts standard library (std::fs).
fn main() {
    println!("Hello World from a Rust App compiled for Linux");
    let mut file = OpenOptions::new()
        .create(true)
        .truncate(true)
        .read(true)
        .write(true)
        .open("/tmp/foobar")
        .unwrap();
    let write_msg = "Hello World; it works!!";
    file.write_all(write_msg.as_bytes()).unwrap();
    file.seek(SeekFrom::Start(0)).unwrap();
    let mut read_msg = String::new();
    file.read_to_string(&mut read_msg).unwrap();
    assert_eq!(write_msg, read_msg);
    println!("File content is: '{}'", read_msg);
}
```

Listing 9.3: Linux application that uses Rusts standard library (`std::fs`) to perform basic file operations.

```rust
fn main() {
    // Linux: write()
    println!("Hello, world!");

    // Check Env var; only execute under Hedron
    if var("LINUX_UNDER_HEDRON").is_ok() {
        // Linux: write()
        println!("This Linux binary executes under Hedron");
        // Hedron: create_pd()
        let pd_obj = libhrstd::PdObject::create_pd(/*...*/);
        // ...
    } else {
        // Linux: write()
        println!("This Linux binary executes under native Linux");
    }
}
```

Listing 9.4: Hybrid Linux application written in Rust. If it runs under Hedron, it also performs Hedron-native system calls. They are abstracted behind the libhrstd library.

9.3 Additional Implementation Details

As discussed in Section 1.3 on page 24, the runtime system is not part of the scientific aspect of this work. In this section, I would like to present further challenges I had to solve during my implementation of the runtime system.

Roottask and Userland Tarball

When the roottask starts execution, it sets up its stack, its heap allocator, logging facilities, and the runtime services that live inside it. This includes the in-memory file-system service, the allocator service, and the logger service. Furthermore, it initializes the process manager.

To start further programs the roottask needs to access the files that include the machine code somehow. One possibility is to use Multiboot boot modules. During the implementation I figured out that the build setup as well as the code become simpler if I only provide one Multiboot boot module that contains all files. I decided to use a tarball (a Tar-archive) for that.

The Multiboot-compliant bootloader provides the tarball to Hedron and Hedron makes it accessible by the roottask. Figure 9.1 on the next page shows this. To access the tarball's data, the Hypervisor Information Page (HIP) is searched for Multiboot modules. If the module with the tarball is found, the roottask maps itself

the memory to that module. Afterwards, the roottask can parse the tarball and extract the files. The tarball contains a flat hierarchy of ELF files. The file names the roottask expects in there are currently hard-coded. However, the tarball can contain a configuration file that includes what programs should be bootstrapped and how they are called in the future.

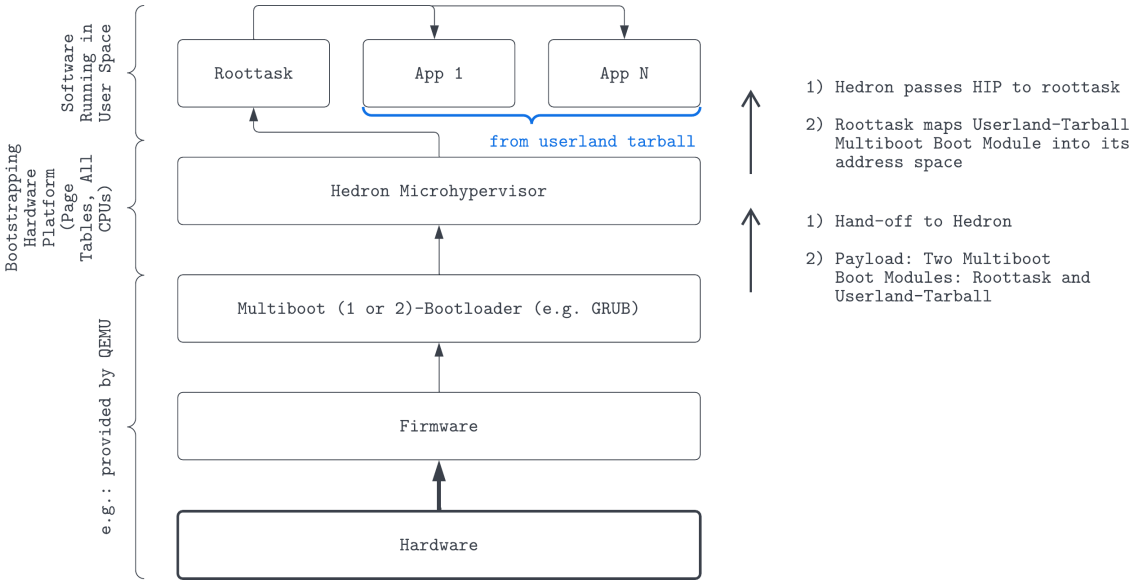

Figure 9.1: The figure shows the bootstrapping flow of Hedron to the running user apps. It starts with the hardware on the bottom. It shows that the Multiboot bootloader passes payload to Hedron as Multiboot boot modules. Hedron stores references to these boot modules inside the Hypervisor Information Page (HIP) so the roottask can find the memory of them. The roottask can then map itself the corresponding memory region and finally read the userland tarball.

9.4 Supported Linux System Calls

The following list shows the Linux system calls that my runtime environment (introduced in Chapter 4 on page 57) currently supports. Not all Linux system calls are implemented with 100% feature completeness but still have enough functionality to enable the successful execution of various simple Linux programs written in C and Rust. For now, this is only a small subset of Linux system calls. This is no limitation by my mechanism.

- `arch_prctl`
- `brk`
- `close`
- `fcntl`
- `fstat`
- `ioctl`
- `lseek`
- `madvise`
- `mmap`
- `munmap`
- `open`
- `poll`
- `read`
- `rtsigaction`
- `rtsigprocmask`
- `set_tid_address`
- `signalstack`
- `unlink`
- `write`
- `write_v`

9.5 Side Contributions

During my work on this thesis and the implementation much code had to be written. In this process, I have created a few open source libraries and contributed them to the Rust ecosystem because there were no satisfying solutions yet. *linux-libc-auxv*[1] is a library to build and parse the initial Linux stack layout. *tar-no-std*[2] is a library that can extract files from tarballs in contexts without a standard environment and without heap allocations. *Simple Chunk Allocator*[3] is a combination of a next-fit and best-fit heap allocator for "no_std" Rust programs that uses static memory as backing memory. I use it in my roottask.

Furthermore, I made smaller contributions to third party open source Rust projects, such as `https://crates.io/crates/elf_rs`.

I would also like to thank Adam Lackorzynski from *Kernkonzept*[4]. After a personal discussion in October 2021, Adam upstreamed an important patch[5] to the QEMU project that significantly improved my developer experience during the engineering of the runtime environment. The patch brings major improvements to the startup time of QEMU when large files (more than two MiB) are used with QEMUs multiboot functionality. It is included in QEMU 6.2 and above[6].

[1] crates.io: `https://crates.io/crates/linux-libc-auxv`
GitHub: `https://github.com/phip1611/linux-libc-auxv`
[2] crates.io: `https://crates.io/crates/tar-no-std`
GitHub: `https://github.com/phip1611/tar-no-std`
[3] crates.io: `https://crates.io/crates/simple-chunk-allocator`
GitHub: `https://github.com/phip1611/simple-chunk-allocator`
[4] `https://www.kernkonzept.com/`
[5] `https://gitlab.com/qemu-project/qemu/-/commit/48972f8cad24eb4462c97ea68003e2dd35be0444`
[6] `https://gitlab.com/qemu-project/qemu/-/commits/v6.2.0`

Glossary

Most of these terms are explained thoroughly in Chapter 2 on page 25. The glossary only gives a brief overview.

Hedron-specific Terms

- **Capability**: The granted ability of a PD to access a certain resource, such as a memory page or a kernel object.
- **Capability Selector**: A numeric index into the capability space of a PD. Usage is similar to a file descriptor in UNIX.
- **Execution Context (EC)**: A kernel object similar to a thread in UNIX. The entity that receives CPU time. There are two relevant kinds: global ECs and local ECs.
  - **Global Execution Context**: An EC with a dedicated SC, i.e., time slice. Used to execute main program functionality. Same role as a thread in UNIX.
  - **Local Execution Context**: An EC without a dedicated time slice is never scheduled automatically. Instead, PTs are attached to it and it runs in the time slice of the caller when such a PT is called.
- **Foreign Application**: Application with non-Hedron-native system call interface (e.g. Linux application).
- **Foreign System Call**: System calls triggered by foreign applications.
- **Hybrid Application**: A foreign application that contains a portion of Hedron-native system calls in addition. The application therefore can use two system-call ABIs simultaneously.
- **Hypervisor Information Page (HIP)**: A page provided by Hedron for the roottask that holds relevant information about the system.
- **Kernel Object**: A data structure, usually with a mutable state, that the kernel manages to fulfill the promised functionality. It is the base for certain programming primitives of the system.

- **Message Transfer Descriptor (MTD)**: Hardware-dependent word-width bitmap that specifies what data should be stored into or loaded from the UTCB when an exception occurs or is replied.
- **Native System Call Toggle (NSCT)**: A flag I introduced in the UTCB header that tells Hedron a system call is a native one even if it comes from a foreign application.
- **Protection Domain (PD)**: A kernel object used as resource container to manage memory capabilities, kernel object capabilities, and port I/O capabilities. A thin abstraction that is similar to a process in UNIX.
- **Portal (PT)**: A kernel object used as IPC entry point. Bound to a local EC. Specifies the instruction pointer that is loaded into register `rip` after a portal call.
- **Portal Call**: An IPC call to a PT with the intention to request the desired functionality, e.g. logging a message or allocating memory.
- **Portal Context (PTCtx)**: A property in implementation of my runtime system to retrieve contextual information about a portal that was called. For example, if the portal handles exceptions, service calls, or foreign system calls.
- **Scheduling Context (SC)**: A time slice for a global EC with a priority. One SC belongs to exactly one global EC.
- **Semaphore(SM)**: A kernel object used for (cross core) synchronization. Effectively, this is an one bit IPC.
- **User Thread Control Block**: Special memory region that is pinned inside the kernel and used to transfer data for kernel-to-user or user-to-user IPC. It has the size of a page (4096 bytes).

Other Terms

- **Application Binary Interface**: An API on binary level, i.e., without high level bindings. The API that a functionality has after the compilation step.
- **Application Programming Interface (API)**: A software interface that exports some service or functionality in a well-defined format.
- **Binary Large Object (BLOB)**: A large object mapped 1:1 to memory. For example, a file.
- **Boot Module**: A *Multiboot* boot module, or also called a GRUB boot module, is a Binary Large Object (BLOB) in memory that is passed to a loaded application, such as a kernel, by the bootloader.

- **Dynamic Binary**: Means that a binary has dependencies to one or multiple dynamic library and a program loader needs to provide them during load time.
- **Inter-process communication (IPC)**: An exchange of data between different processes.
- **First-Class Citizens**: First-class citizens in the context of this work describes applications that run on an equal level to native applications.
- **Foreign Function Interface (FFI)**: A mechanism by which a program written in one programming language can call routines or make use of services written in another.
- **Kernel space**: All code running in privileged mode on hardware. On x86_64, this refers to code running in ring 0.
- **Hypervisor**: Sometimes used as synonym of a VMM. In microkernel contexts this refers to the kernel space part of the virtualization infrastructure.
- **L4Linux**: Modified Linux kernel that runs on an L4 microkernel as user-space process.
- **libc**: Originally developed as standard library for programs written in C on UNIX systems. Today the de-facto standard API on Linux distributions and other UNIX-like systems to the kernel. Examples are *glibc* and *musl*.
- **Library OS (libOS)**: A concept where the functionality expected from the OS system call interface is packaged as library and linked into the address space of a program. System calls are replaced with function calls into the corresponding libOS.
- **Microhypervisor**: A microkernel with the main goal of enabling virtualized environments on the given hardware platform.
- **Microkernel**: A kernel with minimal code running in privileged mode on the hardware.
- **Pinned Memory**: Pinned memory has a fixed location and is always mapped and present.
- **POSIX**: A subset of the UNIX functionality specified by the IEEE.
- **Process (UNIX/Windows)**: Encapsulates an address space with multiple threads. The abstraction used to run programs.
- **Static Binary**: Means that a binary has all relevant libraries included and does not require the program loader to provide dynamic libraries during load time.

- **Thread (UNIX/Windows)**: The unit of execution. One thread belongs to one process. It knows its current instruction pointer and its own stack. Despite the stack, all memory is shared with other threads within the same process.
- **Toolchain**: A combination of tools that in combination produce usable binaries or libraries. Includes the compiler, the linker, and mandatory runtime libraries. It depends on the programming language, the ecosystem of the programming language, the target hardware platform, and the target OS, as well as the host OS on which the toolchain is executed.
- **UNIX**: UNIX was an OSs developed in the 1970s which principles live until today in many OSs and APIs. The principles are also known as the UNIX philosophy.
- **User space**: All code running in user mode (non privileged) on hardware. On x86_64 this refers to code running in ring 3.
- **Userland**: The entirety of software running in user space that fulfills the promised functionality of the software system. Typically, this includes service processes, such as the file-system service, the network stack, and other user-space components.
- **Virtual Machine Monitor (VMM)**: A VMM is a component that emulates the necessary environment for a VM. Sometimes this is used as synonym for a Hypervisor. In microkernel contexts this refers to the userland part of the virtualization infrastructure.

Acronyms

ABI Application Binary Interface.

API Application Programming Interface.

AuxV Auxiliary Vector.

BLOB Binary Large Object.

EC Execution Context.

ELF Executable and Linking Format.

GUI Graphical User Interface.

HIP Hypervisor Information Page.

IDE Integrated Development Environment.

IPC Inter-process Communication.

LAPIC Local Advanced Programmable Interrupt Controller.

MTD Message Transfer Descriptor.

NSCT Native System Call Toggle.

OS Operating System.

PD Protection Domain.

PT Portal.

RTT Round-Trip Time.

SC Scheduling Context.

SLOC Source Lines of Code.

SM Semaphore.

UTCB User Thread Control Block.

VM Virtual Machine.

Bibliography

[1] *[PATCH v13 00/10] NTFS read-write driver GPL implementation by Paragon Software - Konstantin Komarov*. URL: https://lore.kernel.org/lkml/20201120160944.1629091-1-almaz.alexandrovich@paragon-software.com/ (visited on Jan. 3, 2022).

[2] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the art of virtualization". In: *ACM SIGOPS Operating Systems Review* 37.5 (Oct. 2003), pp. 164–177. ISSN: 0163-5980. DOI: 10.1145/1165389.945462. URL: https://doi.org/10.1145/1165389.945462 (visited on Dec. 23, 2021).

[3] Andrew Baumann, Jonathan Appavoo, Orran Krieger, and Timothy Roscoe. "A fork() in the road". In: *Proceedings of the Workshop on Hot Topics in Operating Systems*. HotOS '19. New York, NY, USA: Association for Computing Machinery, 2019, pp. 14–22. ISBN: 978-1-4503-6727-1. DOI: 10.1145/3317550.3321435. URL: https://doi.org/10.1145/3317550.3321435 (visited on Mar. 24, 2022).

[4] benhillis. *WSL Release Notes*. de-de. URL: https://docs.microsoft.com/de-de/windows/wsl/release-notes (visited on Jan. 2, 2022).

[5] Jonathan Cobert. *Emulating Windows system calls in Linux [LWN.net]*. en-US. June 2020. URL: https://lwn.net/Articles/824380/ (visited on Dec. 30, 2021).

[6] *Comparing WSL 1 and WSL 2*. URL: https://docs.microsoft.com/en-us/windows/wsl/compare-versions (visited on Feb. 15, 2022).

[7] David Drysdale. *How programs get run: ELF binaries [LWN.net]*. Feb. 2015. URL: https://lwn.net/Articles/631631/ (visited on Jan. 3, 2022).

[8] Kevin Elphinstone and Gernot Heiser. "From L3 to SeL4 What Have We Learnt in 20 Years of L4 Microkernels?" In: *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*. SOSP '13. Farminton, Pennsylvania: Association for Computing Machinery, 2013, pp. 133–150. ISBN: 9781450323888. DOI: 10.1145/2517349.2522720. URL: https://doi.org/10.1145/2517349.2522720.

[9] Sebastian Eydam. *Mitigating Processor Vulnerabilities by Restructuring the Kernel Address Space*. URL: https://fosdem.org/2022/schedule/event/seydam/ (visited on Mar. 27, 2022).

[10] *Fuchsia Exception Handling*. URL: https://fuchsia.dev/fuchsia-src/concepts/kernel/exceptions (visited on Feb. 15, 2022).

[11] *Genode*. URL: https://genode.org/ (visited on Feb. 10, 2022).

[12] *GitHub - Cyberus Technology - The Hedron Microhypervisor*. URL: https://github.com/cyberus-technology/hedron (visited on Jan. 20, 2022).

[13] Robert P. Goldberg. "Survey of virtual machine research". In: *Computer* 7.6 (June 1974). Conference Name: Computer, pp. 34–45. ISSN: 1558-0814. DOI: 10.1109/MC.1974.6323581.

[14] Gerhard Goos, Juris Hartmanis, Jan van Leeuwen, David Hutchison, Takeo Kanade, Josef Kittler, Jon M Kleinberg, Friedemann Mattern, John C Mitchell, Moni Naor, Oscar Nierstrasz, C Pandu Rangan, and Bernhard Steffen. "Lecture Notes in Computer Science". en. In: (2016), p. 1228.

[15] Guru. *15 Years Old Linux Bug Let Attackers Gain Admin Privileges*. Cyber Security News. Mar. 16, 2021. URL: https://cybersecuritynews.com/15-years-old-linux-bug/ (visited on Jan. 20, 2022).

[16] *gVisor*. URL: https://gvisor.dev/ (visited on Jan. 24, 2022).

[17] *Intel® Virtualization Technology (VT) in Converged Application Platforms*. en. 2007. URL: https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/virtualization-tech-converged-application-platforms-paper.pdf (visited on Jan. 3, 2022).

[18] *L4Linux*. URL: https://l4linux.org/ (visited on Jan. 2, 2022).

[19] *L4Re*. URL: https://l4re.org/ (visited on Feb. 10, 2022).

[20] Adam Lackorzynski and Alexander Warg. "Taming subsystems: capabilities as universal resource access control in L4". In: *IIES '09*. 2009. DOI: 10.1145/1519130.1519135.

[21] Adam Lackorzynski, Carsten Weinhold, and Hermann Härtig. "Combining Predictable Execution with Full-Featured Commodity Systems". In: (), p. 6.

[22] Adam Lackorzynski, Carsten Weinhold, and Hermann Härtig. "Decoupled: Low-Effort Noise-Free Execution on Commodity Systems". In: *Proceedings of the 6th International Workshop on Runtime and Operating Systems for Supercomputers*. ROSS '16. New York, NY, USA: Association for Computing Machinery, June 1, 2016, pp. 1–8. ISBN: 978-1-4503-4387-9. DOI: 10.1145/2931088.2931095. URL: https://doi.org/10.1145/2931088.2931095 (visited on Feb. 26, 2022).

[23] Jochen Liedtke. "On µ-kernel construction". In: *Symposium on Operating System Principles*. ACM, 1995.

[24] Robert Love. *Linux System Programming [Book]*. Jan. 1, 2007. ISBN: 978-0-596-00958-8. URL: https://www.oreilly.com/library/view/linux-system-programming/0596009585/ch01.html (visited on Feb. 8, 2022).

[25] Marshall Kirk McKusick, George V. Neville-Neil, and Robert N. M. Watson. *The Design and Implementation of the FreeBSD Operating System*. Google-Books-ID: aY1pBAAAQBAJ. Addison Wesley, Aug. 2014. 926 pp. ISBN: 978-0-321-96897-5.

[26] Frank Mehnert, Michael Hohmuth, Sebastian Schönberg, and Hermann Härtig. "RTLinux with Address Spaces". In: (), p. 5.

[27] *Musl design concepts*. URL: https://wiki.musl-libc.org/design-concepts.html (visited on Mar. 24, 2022).

[28] *OpenBSD system-call-origin verification [LWN.net]*. URL: https://lwn.net/Articles/806776/ (visited on Dec. 30, 2021).

[29] Florian Pester. "ELK Herder - Replicating Linux Processes with Virtual Machines". MA thesis. Technische Universität Dresden, Feb. 2014.

[30] *RFC-0082: Running unmodified Linux programs on Fuchsia*. en. Feb. 2021. URL: https://fuchsia.dev/fuchsia-src/contribute/governance/rfcs/0082_starnix (visited on Jan. 2, 2022).

[31] L. Sha, R. Rajkumar, and J.P. Lehoczky. "Priority inheritance protocols: an approach to real-time synchronization". In: *IEEE Transactions on Computers* 39.9 (Sept. 1990). Conference Name: IEEE Transactions on Computers, pp. 1175–1185. ISSN: 1557-9956. DOI: 10.1109/12.57058.

[32] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. "EROS: a fast capability system". In: *ACM SIGOPS Operating Systems Review* 33.5 (1999), pp. 170–185. ISSN: 0163-5980. DOI: 10.1145/319344.319163. URL: https://doi.org/10.1145/319344.319163 (visited on Feb. 7, 2022).

[33] Zhiming Shen, Zhen Sun, Gur-Eyal Sela, Eugene Bagdasaryan, Christina Delimitrou, Robbert Van Renesse, and Hakim Weatherspoon. "X-Containers: Breaking Down Barriers to Improve Performance and Isolation of Cloud-Native Containers". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, Apr. 2019, pp. 121–135. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304016. URL: https://doi.org/10.1145/3297858.3304016 (visited on Aug. 5, 2021).

[34] *signal(7) - Linux manual page*. URL: https://man7.org/linux/man-pages/man7/signal.7.html (visited on Jan. 7, 2022).

[35] Prateek Singh. "Getting Started with WSL". en. In: *Learn Windows Subsystem for Linux: A Practical Guide for Developers and IT Professionals*. Ed. by Prateek Singh. Berkeley, CA: Apress, 2020, pp. 1–17. ISBN: 978-1-4842-6038-8. DOI: 10.1007/978-1-4842-6038-8_1. URL: https://doi.org/10.1007/978-1-4842-6038-8_1 (visited on Dec. 23, 2021).

[36] Udo Steinberg and Bernhard Kauer. "NOVA: A Microhypervisor-Based Secure Virtualization Architecture". In: *Proceedings of the 5th European Conference on Computer Systems*. EuroSys '10. Paris, France: Association for Computing Machinery, 2010, pp. 209–222. ISBN: 9781605585772. DOI: 10.1145/1755913.1755935. URL: https://doi.org/10.1145/1755913.1755935.

[37] *System V ABI: x86 psABIs / x86-64 psABI*. URL: https://gitlab.com/x86-psABIs/x86-64-ABI (visited on Jan. 20, 2022).

[38] Andrew Stuart Tanenbaum. *Modern Operating Systems, 4th Edition*. ISBN: 978-0-13-359162-0. URL: https://www.pearson.com/content/one-dot-com/one-dot-com/us/en/higher-education/program.html (visited on Jan. 20, 2022).

[39] Linus Torvalds. *GitHub: torvalds/linux*. original-date: 2011-09-04T22:48:12Z. Sept. 2021. URL: https://github.com/torvalds/linux/blob/35776f10513c0d523c5dd2f1b415f642497779e2/include/linux/syscalls.h#L132 (visited on Sept. 13, 2021).

[40] *Usage share of operating systems*. de. Page Version ID: 1042708909. Sept. 2021. URL: https://en.wikipedia.org/w/index.php?title=Usage_share_of_operating_systems&oldid=1042708909 (visited on Sept. 8, 2021).

[41] Jack Wallen. *Linux kernel 5.15: NTFS support gets a significant boost*. en. URL: https://www.techrepublic.com/article/ntfs-support-gets-a-significant-boost-in-linux-kernel-5-15/ (visited on Jan. 3, 2022).

[42] Ethan G Young, Pengfei Zhu, Tyler Caraza-Harter, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. "The True Cost of Containing: A gVisor Case Study". In: (), p. 6. URL: https://www.usenix.org/system/files/hotcloud19-paper-young.pdf.

[43] *Zicron System Calls*. URL: https://fuchsia.dev/fuchsia-src/reference/syscalls (visited on Mar. 29, 2022).