Evaluation of Rust for Operating System Development and Porting Key Components of the HermitCore Unikernel

Evaluierung von Rust zur Betriebssystementwicklung und Portierung von Schlüsselkomponenten des Unikernels HermitCore

Colin Finck
Matriculation Number: 314570

Master Thesis

The present work was submitted to
RWTH Aachen University
Faculty of Electrical Engineering and Information Technology
Institute for Automation of Complex Power Systems
Univ.-Prof. Antonello Monti, Ph. D.

Supervisor: Dr. rer. nat. Stefan Lankes

Eidesstattliche Versicherung

Ich, Colin Finck (Matrikelnummer 314570), versichere hiermit an Eides Statt, dass ich die vorliegende Masterarbeit mit dem Titel

Evaluierung von Rust zur Betriebssystementwicklung und Portierung von Schlüsselkomponenten des Unikernels HermitCore

selbstständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	Unterschrift

Belehrung

§156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit einer Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des §158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum	Unterschrift

Kurzfassung

Diese Arbeit bewertet die Tauglichkeit der neuartigen und auf Sicherheit und Nebenläufigkeit ausgelegten Programmiersprache *Rust* für die Entwicklung von Betriebssystemen. Dazu wird das in C geschriebene Library-Betriebssystem *HermitCore* nach Rust portiert. Eigenschaften von HermitCore und Funktionen von Rust werden im Detail erläutert. Schlüsselkomponenten wie die Speicherverwaltung, Hardware-Initialisierung und der Task-Scheduler werden neu geschrieben, um das System-Design zu verbessern und die Vorteile von Rust auszunutzen. Zudem werden generische Implementierungen von Datenstrukturen entwickelt, die aktuell nicht von Rust bereitgestellt werden und für eine Vielzahl von Anwendungsfällen einsetzbar sind. Das entstandene Betriebssystem wird auf mehreren Test-Rechnern verifiziert und mit der C-Version verglichen. Benchmarks zeigen, dass die Rust-Implementierung teilweise schneller als das Original ist und es wird argumentiert, dass die Rust-Version ebenfalls weniger anfällig für Programmierfehler ist. Die Arbeit endet mit einem Fazit und Ideen für zukünftige Verbesserungen von HermitCore und Rust.

Stichwörter: HermitCore, Rust, Unikernel, Multi-Kernel, Betriebssysteme

Abstract

This thesis evaluates the novel programming language *Rust*, which is tailored towards safety and concurrency, for operating system development. This is accomplished by porting the library operating system *HermitCore* written in C to Rust. Characteristics of HermitCore and features of Rust are presented in-depth. Key components like the Memory Manager, hardware initialization, and task scheduler are rewritten to improve the operating system design and leverage the advantages of Rust. Moreover, generic implementations of data structures are provided, which are currently unavailable in Rust and usable for a variety of applications. The resulting operating system is verified on multiple test systems and compared to the C version. Benchmarks prove that the Rust implementation is partially faster than the original and it is argued that the Rust version is also less prone to programming mistakes. The thesis ends with a conclusion and ideas for further improvements on HermitCore and Rust.

Keywords: HermitCore, Rust, Unikernel, Multi-Kernel, Operating Systems

Contents

1 Introduction

HermitCore[1] is a novel operating system kernel developed at the Institute for Automation of Complex Power Systems (ACS) since 2015. It is tailored for low system noise and predictable runtime behavior to facilitate High-Performance Computing (HPC) applications scaled across thousands of nodes [1]. Reducing system noise has become increasingly important in Symmetric Multiprocessing (SMP) systems as jobs are parallelized among more and more processors. Due to the burden of synchronization in parallel systems, noise on a single processor can delay the execution of other processors and have a huge negative impact on the overall performance [2].

To fulfill these goals, HermitCore has been designed as a single-address-space library operating system, also called *Unikernel* [3]. Application code is compiled together with the operating system code into a single lightweight binary, which can run directly on hardware or inside a virtual machine. Additionally, HermitCore can also work as a *Multi-Kernel* side by side to Linux. This enables running the performance-critical part of an application inside HermitCore while Linux provides a fully-weight kernel for pre- and post-processing. By providing support for the entire GNU Compiler Collection (GCC) and common programming models such as OpenMP, existing HPC applications can easily be built for HermitCore.

Rust[2] is a programming language invented by Graydon Hoare and sponsored by Mozilla Research since 2009 [4]. As a compiled systems language with deterministic memory management, it competes directly with C and C++, but puts a special emphasis on safety and concurrency [5]. Examples for such features include bounds-checked indexing, variables being private and immutable by default, guaranteed validity of memory references, and ensuring that only one mutable reference to a variable exists at the same time. These rules are checked at compile time, so they do not incur runtime costs and it is impossible to violate them accidentally. By employing these techniques, Rust tries to eliminate the most common programming mistakes such as buffer overflows, accessing invalid pointers, and data races in multithreaded code [6]. They rank among the top software security issues [7].

Rust originated out of dissatisfaction with C++ [8]. As such, it partly resembles its syntax and supports concepts like Generic Programming, Resource Acquisition Is Initialization (RAII), and Smart Pointers. However, it has also taken influences from newer languages, e.g. Haskell's Typeclasses or OCaml's Pattern Matching [9].

[1] `http://www.hermitcore.org`
[2] `https://www.rust-lang.org`

Within the scope of this thesis, the suitability of the Rust language and toolchain for writing entire operating systems for HPC applications is evaluated. This is done by rewriting the existing C code of HermitCore key components in Rust and integrating them into the library operating system. Whenever Rust offers a safer or more elegant way of implementing a feature, it is preferred over a direct translation of the C code. However, a focus also lies on maintaining compatibility to existing HermitCore applications, in particular the system call interface.

In the following chapter, the HermitCore operating system and Rust programming language are introduced in depth. Chapter 3 details the implementation of all ported key components and the integration into the existing build system and codebase. By the end of that chapter, Rust code has completely replaced the previous HermitCore implementation and only a few external libraries written in C remain. The resulting implementation is then benchmarked in Chapter 4 to measure the performance impact of the used Rust features. Identified advantages and disadvantages of the current Rust language and toolchain are presented. Answers are given to the questions whether Rust's safety measures really improve the code quality and whether they come at a major cost of performance. The thesis concludes with a brief summary as well as ideas for future work.

2 Basics

This chapter provides an overview about the HermitCore operating system and the Rust programming language. Important concepts are introduced, which are later applied in Chapter 3 for the implementation.

2.1 The HermitCore Operating System

The hierarchical organization of current HPC systems into thousands of nodes, which itself feature multiple processors, requires highly scalable system software. This complexity is increasing even further with hardware trends such as Non-Uniform Memory Access (NUMA) and Non-Uniform Cache Architecture (NUCA), so the software stack has to adapt to these changes. On the other hand, HPC application software often relies on stable operating system interfaces like the Portable Operating System Interface (POSIX) [1]. To fulfill all requirements, a customized version of Linux is usually used in the HPC space [10].

However, Linux has been designed as a general-purpose multi-user multitasking operating system. These properties add unnecessary complexity for HPC jobs and complicate customizations. Another major problem are background tasks executed by the kernel, which contribute to high system noise. Due to the burden of synchronization in parallel systems, noise on a single processor can delay the execution of other processors and have a huge negative impact on the overall performance. Tsafrir et al. have provided a probabilistic argument that the effect of system noise is linearly proportional to the size of the cluster [2]. Therefore, measures have to be taken to reduce the system noise to a minimum.

To address these problems, three different approaches are currently widespread:

1. Taking the fully-weight Linux kernel and removing all features not necessary for HPC.

2. Developing a lightweight kernel for HPC applications.

3. Developing a lightweight kernel (called *Multi-Kernel*) that runs an HPC application side by side to Linux and forwards all system calls to it.

Option 1 usually retains full compatibility with existing applications, but comes at the expense of maintaining a highly modified Linux fork. Option 2 promises an operating system tailored for the specific needs of HPC, but often lacks compatibility with well-established operating system interfaces like POSIX. Option 3 combines

advantages of both previous approaches. However, this requires the setup of two kernels and system call forwarding may hamper performance.

HermitCore represents a new approach to address the problems of HPC systems. On the one hand, it has been designed as a library operating system, also called *Unikernel* [3]. As such, application and kernel code are compiled together into a single optimized binary, which can run directly on hardware or inside a virtual machine. There is no user and privilege separation, so all code runs in the same memory address space. Consequently, system calls become simple function calls, whereas they often lead to expensive context switches in general-purpose operating systems. Moreover, the absence of periodic clock ticks highly reduces the system noise compared to Linux. All these factors guarantee maximum performance and predictable runtime behavior for HPC needs. By also providing support for the GCC toolchain, Message Passing Interface (MPI), OpenMP, and POSIX, existing HPC applications can easily be built for HermitCore.

On the other hand, HermitCore can also work as a *Multi-Kernel* side by side to Linux on dedicated processors. This enables running the performance-critical part of an application inside HermitCore while Linux provides a fully-weight kernel for pre- and post-processing. Some system calls can also be delegated to the Linux system, for example to provide access to I/O devices unsupported by HermitCore.

Due to its unique design, the HermitCore kernel has been written from the ground up and is not derived from any popular operating system family. However, some code is shared with former projects of the ACS and Lehrstuhl für Betriebssysteme (LfBS) institutes, such as eduOS[1] and MetalSVM[2].

In the following, individual aspects of the HermitCore kernel are presented in detail.

2.1.1 Architecture Support

HermitCore has been developed for the x86-64 architecture supported by processors of Advanced Micro Devices (AMD) and Intel, with a port to ARM Limited's AArch64 architecture in progress[3]. This makes it compatible with the Intel Xeon and AMD Opteron processor platforms as well as the Intel Xeon Phi accelerator cards dominant in the server and supercomputing space [11].

The x86-64 architecture is a Complex Instruction Set Computer (CISC) architecture, which means that it supports a large number of powerful instructions at the expense of a complex chip design. It has evolved as a 64-bit extension of the 32-bit Intel i386 architecture, with the goal of supporting 64-bit memory addressing and integer arithmetic, more registers, and cleaning up the feature set while maintaining compatibility to existing 32-bit applications at the same time [12].

[1] `https://github.com/RWTH-OS/eduOS`
[2] `http://www.lfbs.rwth-aachen.de/content/765.html`
[3] `https://github.com/RWTH-OS/HermitCore/tree/aarch64`

For computational-intensive applications, the x86-64 instruction set additionally incorporates a number of Single Instruction, Multiple Data (SIMD) extensions to allow a single integer or floating-point calculation to be performed on multiple data values simultaneously. However, using the latest SIMD extensions in applications requires the underlying operating system to save and restore the context of the SIMD registers during task switches. It also has to report this to the processor and application by setting the respective bits in the CR4 and XCR0 Control Registers.

As HermitCore has been designed with HPC applications in mind, it implements operating system support for all Intel SIMD extensions to date (including AVX2 and Xeon Phi's AVX-512 extensions [1]). Furthermore, it also detects the presence of Enhanced Intel SpeedStep Technology (EIST) power management in the processor and adjusts that to achieve the goal of maximum performance and predictable runtime behavior.

Even though HermitCore is engineered towards low system noise, it has to partially implement interrupt events to offer support for I/O devices (like network cards) and timers. This requires the initialization and handling of an interrupt controller chip. Due to its x86 heritage, a typical x86-64 platform comes with two different interrupt controller types:

- The Programmable Interrupt Controller (PIC), also called Intel 8259, has been available since the very first IBM Personal Computer (Model 5150) in 1981. It only supports 8 interrupt lines, but can be cascaded with another PIC to double the number of interrupt lines. With the growing number of peripherals attached to a computer, a second PIC has been introduced in the IBM PC/AT in 1984. Modern computers and servers still contain two PICs or an emulation thereof, however their general usage is discouraged. Compared to the alternative, they are slower, focused on single processor systems, and support less interrupt lines. It should be noted though that the first interrupt line of the first PIC in every x86 computer is connected to a Programmable Interval Timer (PIT) running at the constant frequency of 1193182 Hz.

- The Advanced Programmable Interrupt Controller (APIC) has been introduced in 1992 as a modern alternative to the legacy PICs [13]. Formerly an extra chip on the motherboard for every processor, it has been integrated into the processor itself since the Intel Pentium Pro in 1995 [14]. Compared to the legacy PICs, it supports 256 interrupt vectors (of which 24 can be interrupts of external I/O devices), memory-mapped registers for fast communication, and Inter-Processor Interrupts (IPIs) to communicate with up to $2^8 = 256$ processors of an SMP system. It also incorporates a programmable timer oscillating at the processor frequency. The APIC architecture has been refined further with the introduction of the xAPIC in the Intel Pentium 4 processor in 2000 and the x2APIC in Intel Nehalem-based processors in 2008. Latter one allows to address up to $2^{32} \approx 4$ billion processors and supports access to APIC registers through more efficient Machine-Specific Registers (MSRs).

Designed as an operating system for multiprocessor servers, HermitCore has to use the APIC for inter-processor communication. However, it also makes use of the APIC Timer for scheduling operating system events and handles interrupts through the APIC. If support for the newer x2APIC is detected, the use of MSRs is preferred over memory-mapped register access. The only exception is the early boot phase where the frequency of the processor and APIC Timer is unknown. In that case, HermitCore has to resort to the legacy PIC and its connected PIT oscillating at a known constant frequency to measure the processor and APIC Timer frequencies.

All x86-64 systems boot up with only a single processor enabled. In order to run applications on other processors in Symmetric Multiprocessing (SMP) operation, the operating system first has to enumerate the available processors. Two different methods provide this information in the x86-64 architecture:

- The original Intel MultiProcessor Specification of 1994 defines some default configurations of dual-processor systems. A server manufacturer can choose one of them and indicate it to the operating system, so it knows about the available processors, APICs, and interrupt assignments. The refined MultiProcessor Specification 1.4 of 1995 adds a *MultiProcessor Configuration Table* [15]. Since then, a server manufacturer can freely define the available number of processors, APICs, and interrupt configurations. This change has also enabled x86 servers with more than two processors.

- The Advanced Configuration and Power Interface (ACPI) has been introduced in 1996 and updated multiple times since then. Its goal is to provide a unified way for the operating system to access hardware, multiprocessor, and power management information and control these aspects of a computer [16]. The information is provided through tables that are loaded into memory when the computer boots up. However, many of these tables contain code in ACPI Machine Language (AML) instead of descriptive data. This requires an operating system to ship with an interpreter for the AML code when it wants to make use of ACPI.

Due to the complexity of the ACPI standard and the dependence on an AML interpreter, HermitCore relies on the traditional MultiProcessor Tables to enumerate available processors. Testing has shown that a large number of modern server systems still provides support for these tables.

The following sections cover implementation details of several HermitCore components. Whenever applicable, a reference to the architecture-dependent implementation for x86-64 is made.

2.1.2 Memory Manager

Managing the installed Random Access Memory (RAM) is one of the core features of every operating system. No matter if an application relies on statically allocated

memory at start-up or dynamically allocated memory at runtime, the operating system needs to keep track of used and free blocks of RAM, and who may access them.

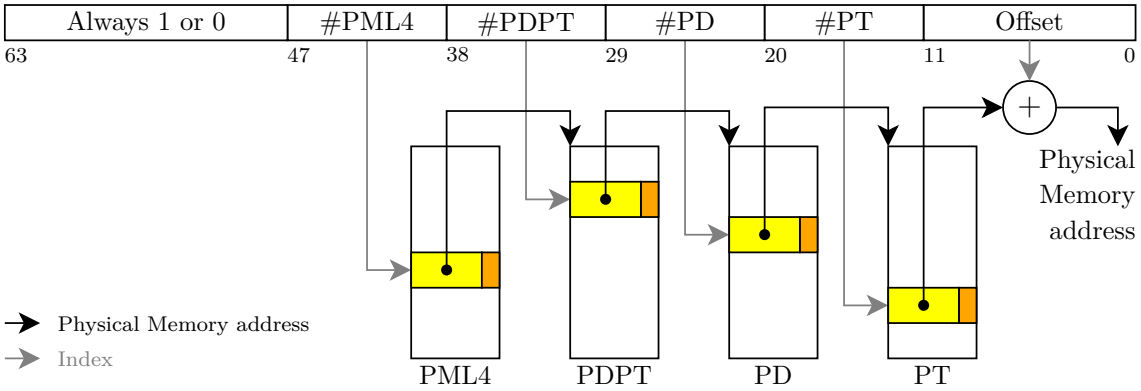
HermitCore currently focuses on the x86-64 architecture, which uses 64-bit addresses to reference memory bytewise and comes with a Memory Management Unit (MMU). An MMU is part of most modern computer architectures to provide support for Virtual Memory, Memory Protection, and Cache Control. When the MMU is enabled, the processor can no longer access physical memory directly. Instead, each memory address (called *Virtual Address*) is looked up in the *Page Tables* and translated into a *Physical Address* based on the information. These Page Tables also reside in RAM. To speed up the translation of frequently used memory addresses, the results of the most recent lookups are stored in the Translation Lookaside Buffer (TLB) cache. If no page table entry exists for an address or access is denied by the memory protection settings, a *Page Fault* exception is thrown, which can be freely handled by the operating system. General-purpose operating systems use the Virtual Memory feature to securely run multiple applications in parallel. Each application is presented with custom Page Tables that can map the same Virtual Addresses to different Physical Addresses in memory. This way, all applications can use the same memory addresses without disturbing each other and no application can maliciously overwrite memory of another application.

Page Tables in the x86-64 architecture manage memory in $2^{12}\,\mathrm{B} = 4\,\mathrm{KiB}$ granularity. These 4 KiB blocks are called *Pages*. As a single Page Table for all possible $\frac{2^{64}}{2^{12}} = 2^{52}$ 64-bit memory addresses in 4 KiB granularity would consume $2^{52} \cdot 8\,\mathrm{B} = 32\,\mathrm{PiB}$, the x86-64 architecture uses two tricks:

- Because no computer system can currently be equipped with anywhere near $2^{64}\,\mathrm{B} = 16\,\mathrm{EiB}$ of RAM, only the lowest 48 bits of a 64-bit Virtual Memory address are used. This provides support for up to $2^{48}\,\mathrm{B} = 256\,\mathrm{TiB}$ of memory. Current x86-64 processors accept only so-called *canonical addresses*, where bits 48 through 63 (counting from zero) replicate bit 47. This ensures that no software abuses these currently unused bits for own purposes and stays compatible with future processors that support a larger virtual address space. In fact, an extension to a 57-bit virtual address space is currently being prepared [17].

- Instead of using a single Page Table to look up the entire Virtual Memory address, the x86-64 architecture divides the used 48 bits of the address into 5 parts, starting from the Most Significant Bit (MSB). Each of these parts is 9 bits wide except for the 5th one, which is 12 bits wide. The 9-bit parts now represent indices to smaller tables that store $2^9 = 512$ entries and only consume $512 \cdot 8\,\mathrm{B} = 4\,\mathrm{KiB}$ each. Technically, these tables are called Page Map Level 4 (PML4), Page Directory Pointer Table (PDPT), Page Directory (PD), and Page Table (PT). As the tables manage memory in 4 KiB granularity, the last 12 bits of a stored memory address would always be zero. Therefore, this

otherwise unused 5th part is used for memory protection and caching flags. A special *Huge Page* flag can also be used to mark an entry in PDPT or PD as a (larger) Page rather than a pointer to the next table. This enables the Page Tables to manage memory in 2 MiB (Huge Page in PD) or 1 GiB (Huge Page in PDPT) granularity. The entire addressing scheme is illustrated exemplarily for memory mapped to a 4 KiB Page in Figure 2.1.



(a) Translating a Virtual Memory address to a Physical Memory address

(b) Format of a Page Table Entry

Figure 2.1: Addressing scheme for memory mapped to a 4 KiB Page in the x86-64 architecture

To address these specifics of the x86-64 MMU, HermitCore implements a Virtual Memory Manager, a Physical Memory Manager, a Paging component, and a Heap Allocator. The Physical Memory Manager uses a so-called *Free List* to keep track of free 4 KiB RAM blocks. This is a doubly-linked list whose entries mark the start and end of free memory regions. They are sorted from the lowest to the highest memory address. The Virtual Memory Manager in HermitCore uses a similarly sorted linked list, however that list keeps track of *used* Virtual Memory Pages. Each first entry in both lists is preallocated while the memory for further entries is allocated using the HermitCore Memory Manager components. The preallocation avoids a *chicken-egg problem*, because allocating a linked list entry needs memory, but memory can only be allocated with existing entries in the list.

None of the HermitCore Memory Manager components currently make use of Huge Pages, even though 2 MiB Pages would consume less entries in the TLB cache and therefore could improve performance. The only exception is the HermitCore Loader, which loads the HermitCore kernel and application to 2 MiB Huge Pages.

The Paging component traverses the Page Tables to map a Virtual Memory address to a Physical Memory address. As previously stated, a processor can

only access Virtual Memory addresses once its MMU has been set up. However, each entry in the Page Tables references Physical Memory addresses. If the Paging component modifies an entry in a Page Table, it needs to find out about a corresponding Virtual Memory address for a known Physical Memory address. To make this straightforward, the HermitCore Paging component employs a trick known as *Self-referencing Page Tables*: The last entry of PML4 is set to the Physical Memory address of PML4 itself. Due to the x86-64 architecture's symmetric partition of a Virtual Memory address into 9-bit indices, this makes the PML4 table available at the Virtual Memory address 0xFFFF_FFFF_FFFF_F000. In particular, the address translation of this Virtual Address works as follows:

1. Bits 39 through 47 of the Virtual Address are examined to determine the index in the PML4 table. They are all 1, and therefore the last entry in PML4 is looked up. This entry usually points to a PDPT, however it is the special self-referencing entry and points back to the same PML4 table. As both tables are equal in size, the MMU continues as usual and has no trouble interpreting the PML4 table as a PDPT.

2. Now bits 30 through 38 of the Virtual Address are examined to determine the index in the PDPT. As the bits are again all 1, the last entry in PDPT is looked up. However, as the last entry in PML4 (looked up in step 1) points to PML4 itself, the lookup is again happening in the same PML4 table.

3. The Virtual Address bits 21 through 29 are examined to determine the index in the PD, and they also point to the last entry. Again, the last entry in the PML4 is looked up.

4. Finally, bits 12 through 20 also point to the last entry in the PT. The PT in this address translation is again the same PML4 from the beginning. Consequently, the translation of the Virtual Memory address 0xFFFF_FFFF_FFFF_F000 has yielded a reference to manipulate the PML4 table.

In a similar fashion, individual PDPT, PD, and PT tables can be referenced with Virtual Memory addresses by using this self-referencing entry in the PML4. For a more detailed description of this technique, the reader is referred to [18].

While general-purpose operating systems provide a distinct PML4 for each application, HermitCore is a single-address-space operating system supporting only one application. Therefore, it is sufficient to set up one PML4 for the kernel and application during boot-up and keep that for the lifetime of the application. This also leads to a better usage of the TLB, which would otherwise be cleared on every task switch.

Finally, HermitCore's Heap Allocator enables the kernel to provide `kmalloc` and `kfree` functions for dynamic memory allocation. As the Paging component only manages memory in 4 KiB granularity, the Heap Allocator incorporates the *Buddy*

System to allocate and deallocate fractions of a Page. For a description of the Buddy System, the reader is referred to [18] and [19].

It should be noted that the Heap Allocator only manages dynamic memory allocations of kernel code. Application code usually calls the `malloc` and `free` functions from the C library, which itself use the POSIX system call `sbrk` to request memory from the operating system. As HermitCore supports only a single application, an area of Virtual Memory is reserved for the only application during boot-up. When the application calls `sbrk`, HermitCore just checks whether the requested additional amount of Virtual Memory is within the boundaries of that area, and grants access to it. Mapping the new Virtual Memory to Physical Memory happens later when the application first accesses the new memory. The access causes a Page Fault that is handled by HermitCore to request Physical Memory on demand.

While protections against malicious applications are less of a concern for HPC operating systems, HermitCore uses the No-eXecute (NX) memory protection feature of the x86-64 architecture wherever possible. This can be used to mark certain memory regions as pure data regions, which the processor will never interpret as code. On general-purpose operating systems, it is heavily used to guard against exploits of heap overflows. However, this feature is also useful for HermitCore to detect some possible programming mistakes early.

2.1.3 Scheduler

HermitCore is designed to run only a single application, and this application is supposed to launch as many threads as there are available processors for optimal hardware utilization. However, there may be situations where an application launches more threads than available processors. Furthermore, when network support is enabled, an additional thread runs on the boot processor to handle TCP/IP communication. Therefore, HermitCore implements a priority-based round-robin scheduler, supporting up to 32 different priorities [1].

Unlike general-purpose operating systems, the scheduler does not distinguish between processes and threads. This allows for a unified handling of both as *tasks*. Another characteristic of HermitCore's scheduler are the separate task queues for each processor. This avoids global scheduler locks when a new task is scheduled.

All modern general-purpose operating systems make use of *Preemptive Multitasking*. This concept involves a periodic interrupt that regularly suspends the current task of a processor and switches to another task. As a result, all tasks get a similar amount of processor time and a single task cannot negatively impact the scheduling of other tasks. Preemptive scheduling can also assist with fulfilling priority constraints, or meeting deadlines in Real-Time Operating Systems (RTOSs). In contrast to that, HermitCore is tailored for low system noise and therefore does not implement a periodic scheduler interrupt. Instead, the scheduler is only called when an application is blocked (through a timer or semaphore wait) or explicitly calls the `sys_yield` system call. However, no more than one task runs on a processor

anyway in the optimal usage scenario for HermitCore, so the drawbacks of this so-called *Cooperative Multitasking* are negligible.

HermitCore's scheduler is customized towards OpenMP applications. These usually come with a main *manager thread* and spawn several *worker threads* for parallel processing. Hence, HermitCore's `sys_clone` system call (used to spawn a thread for the current application) also implements a round-robin algorithm to map each thread to another processor. This mapping is static and does not change during the lifetime of a thread, which promises ideal performance and cache utilization for the optimal HermitCore usage scenario.

2.1.4 Timers

HermitCore belongs to the group of so-called *tickless* kernels. Traditional kernels use a periodic timer with a constant frequency to update the internal tick counter, check for elapsed deadlines, and regularly perform maintenance tasks (such as preemptive scheduling). A tickless kernel, on the other hand, refrains from using a timer of constant frequency. The operating system needs a different way of maintaining the internal tick counter and meeting deadlines. Without the regular interrupts through the periodic timer, the processor is woken up less often and therefore consumes less energy, which is especially important for mobile and embedded devices. Microsoft Windows only integrated tickless operation with Windows 8 in 2012 [20] whereas Linux does not support full tickless operation before version 3.10 in 2013 [21].

Avoiding periodic timers also has an advantage for HPC operating systems, since it reduces the system noise. Because HermitCore already relies on Cooperative Multitasking and does not need periodic calls into the scheduler, implementing tickless operation has been more straightforward than for general-purpose operating systems:

- An internal tick counter is maintained for each processor by reading its Time Stamp Counter (TSC) and calculating the corresponding value for a 100 Hz timer. The TSC is an internal register of every x86-64 processor, which counts the number of clock cycles since boot-up. It increases monotonically at a constant frequency provided that the processor runs at a constant frequency (which is guaranteed by HermitCore's EIST power management settings). Using the TSC eliminates the need for a periodic timer to count clock ticks.

- Deadlines by applications are met by using the APIC Timer in one-shot operation. Every time a deadline elapses, the one-shot timer interrupts program execution, the deadline is handled, and the APIC Timer is reprogrammed for the next deadline. Examples for such deadlines are calls to `sys_msleep` (pausing program execution for a certain time) and timing out when waiting for a semaphore.

2.1.5 Network Support

HermitCore offers to delegate certain system calls to a Linux kernel, for example to support additional I/O devices. Therefore, it needs a high-performance communication interface to exchange messages between both kernels. Currently, this communication is realized through TCP/IP, so HermitCore implements basic network support.

Originally, this feature has been designed for the Multi-Kernel mode, with HermitCore and Linux running side by side on the same computer. However, the implementation supports multiple network adapters and can universally be used to enable communication of HermitCore and Linux over any (virtual or real) network. The following network adapters are used based on the detected operating mode, virtual machine hypervisor, and Peripheral Component Interconnect (PCI) bus devices:

- For Multi-Kernel mode, HermitCore implements a virtual network interface known as `mmnif` and a counterpart in the Linux kernel. With HermitCore and Linux running on different processors of the same computer, communication over this interface happens through shared memory and APIC IPIs.
- If the *uhyve* hypervisor customized for HermitCore instances is detected, the `uhyve-net` driver is loaded. It communicates directly with its counterpart in the hypervisor through port writes and Direct Memory Access (DMA).
- For all other cases, HermitCore provides drivers for the Intel E1000 and Realtek RTL-8139 network interfaces. These are two of the most popular Ethernet chips used in computers and are also often emulated by virtual machine hypervisors. Finally, a driver for the virtual `vioif` interface implemented into many hypervisors is also provided. If any of these network cards is detected on the PCI bus, HermitCore uses the first one to enable networking in HermitCore.

2.1.6 Third-Party Components

HermitCore integrates several open-source third-party components to provide an HPC operating system with POSIX compatibility. Within the scope of this thesis, the following components had to be considered:

- The HermitCore toolchain includes the Binutils[4] and GNU Compiler Collection (GCC)[5] enhanced by a *x86_64-hermit* target to compile binaries for HermitCore. The compiler kit comes with support for the C, C++, Fortran, and Go programming languages. Particularly to support the latter one and its lightweight threads known as *goroutines*, HermitCore implements the uncommon POSIX functions `getcontext`, `makecontext`, and `setcontext`.

[4] `https://www.gnu.org/software/binutils`
[5] `https://gcc.gnu.org`

- Newlib[6] serves as the C library for HermitCore. Due to Newlib's design towards embedded systems without any operating system, HermitCore only needs to implement a low number of system calls to support a C library.
- A version of POSIX Threads for Embedded systems (PTE)[7] adapted to HermitCore provides the popular Pthreads interface. Apart from using OpenMP, this offers another way to write multithreaded code for HermitCore.
- The embedded Lightweight IP (lwIP)[8] stack is integrated to handle TCP/IP communication of a HermitCore instance.
- HermitCore uses the CMake[9] build system to compile the library operating system, dependencies, applications, and link them together into self-contained images.

2.2 The Rust Programming Language

C and C++ are two of the most popular programming languages in 2018 [22], and the dominant languages for writing low-level system software. C has been invented between 1969 and 1973 to provide a structured and typed language as an alternative to assembly for the then upcoming byte-oriented computers. Its history is closely tied to that of the Unix operating system, which has been rewritten in C in 1973. With the rising popularity of Unix and its proven portability to a variety of computer systems, C has become the language of choice for developing all kinds of software [23]. C++ has emerged between 1979 and 1985 as an attempt to extend C with high-level features such as classes and improved type checking while maintaining backward compatibility to C [24]. Today, C++ is often used to write complex applications (such as Adobe Photoshop), extend software originally written in C (such as Microsoft Windows), and it even provides the base for higher level frameworks (like Java or .NET).

Despite the popularity of C and C++, both languages are not without criticism. Their handling of memory as mere pointer addresses easily leads to buffer overflows and invalid memory accesses when used incorrectly. Such programming mistakes are common and do not just cause software malfunctions, but can also induce severe security issues [7]. Also C++ is frequently criticized for its complexity. Two programs written in C++ often use different subsets of the language, and when these subsets do not match, code from one program cannot be trivially imported into the other program [25]. Finally, the development of programming languages has not stopped with C and C++. Since then, new languages have emerged to improve on the mistakes of the existing ones without the need of staying compatible. The

[6] `https://sourceware.org/newlib`
[7] `http://pthreads-emb.sourceforge.net`
[8] `https://savannah.nongnu.org/projects/lwip`
[9] `https://cmake.org`

new concepts also include a native support of concurrency from the programming language, which is not provided by C or C++.

Rust is a modern programming language, which originated out of dissatisfaction with C++ [8]. Development began in 2006 and is sponsored by Mozilla Research since 2009 [4]. Its goal is to provide a language to build safe and concurrent systems while retaining full control over memory and other resources [5]. Unlike other popular languages of 2018, such as C#, Java, JavaScript, or Python, Rust is a language compiled for the target processor and suitable for programming it in a low-level fashion. Nevertheless, it also integrates many features for high-level programming. While resembling some C++ properties (like Generic Programming, RAII, and Smart Pointers), Rust has also taken inspiration from other modern programming languages.

The following sections explain some characteristics of Rust, along with their origins, benefits, and suitability for the task.

2.2.1 The Rust Toolchain

The first Rust compiler has been written in OCaml, but the compiler has since been rewritten in Rust itself, and this new compiler is self-hosting since 2011 [26]. This makes the Rust toolchain one of the largest projects currently using and influencing Rust, another notable one being the Mozilla Servo browser engine.

The Rust compiler `rustc` builds upon the open-source LLVM compiler framework. As such, it only needs to provide a single translator from Rust to LLVM Intermediate Representation (IR) and LLVM is responsible for translating IR to optimized assembly language for each supported target processor.

Most of the time, `rustc` is not called manually, but by the build tool and package manager `cargo`. This tool supports convenient configuration files for describing the build process and required dependencies of a Rust program. If desired, Cargo can automatically download open-source dependencies in the specified version from the public Rust repository Crates.io. For more complex situations, Cargo allows the build process to be controlled with Rust code itself. Often Cargo obviates the need for an external build tool and helps to keep the build process consistent among multiple Rust projects.

Apart from the build process, Rust also tries to keep the code documentation consistent for all projects. Therefore, it ships with `rustdoc`, a tool that automatically generates well-formatted HTML documentation from project source files. This is comparable to *Doxygen* for C/C++ and *Javadoc* for Java. Providing `rustdoc` as part of the basic toolchain advises developers to document their code and maintain a uniform comment style.

Finally, the Rust toolchain extends LLVM's debugger LLDB to provide support for source-level debugging of Rust code. When LLDB is started using the wrapper script `rust-lldb`, the debugger can render Rust-specific types (like `enum`) in a readable way. Rust also extends the well-known GNU Debugger (GDB) with a similar `rust-gdb` wrapper script. As GDB is known as a mature debugger, integrated into many

development environments, and its support for Rust is currently better than LLDB's according to [27], it has been preferred within this thesis.

Due to LLVM's generation of GNU-compatible object files, tools of the LLVM toolchain may be accompanied by GNU tools during the build process. This compatibility has been utilized within this thesis: The existing HermitCore C codebase (compiled by GNU GCC) could be extended by Rust code (compiled by LLVM-based `rustc`) and later the same GNU `ld` linker as for the C codebase has been used to link all components together.

The Rust project maintains multiple *release channels* for the toolchain, the most important ones being *stable* and *nightly*. Components can be downloaded from them using the `rustup` tool. Stable versions represent supported releases containing only mature features. These features are guaranteed to remain compatible with every new version of the Rust toolchain. Nightly versions incorporate the latest features under development, however these are subject to change. This also includes features important for operating system development, such as inline assembly and structure alignment. Therefore, a nightly version has been used within this thesis, precisely `rustc 1.22.0-nightly (17f56c549 2017-09-21)`.

2.2.2 Basic Safety Features

Rust's attempt at creating safer applications starts with safer defaults. Every declared variable is immutable by default, meaning that it can only be assigned once and not modified anymore. If that is required, an explicit declaration with `mut` is necessary. This way, the compiler can catch attempts where variables are modified by mistake and discourages reusing the same variable for different purposes.

A similar rule applies to variables inside structures and functions. By default, these are private and can only be accessed by code within the same module. Adding the `pub` keyword makes them public. This is again different to C/C++, which makes these elements public by default and provides the `static` keyword to mark one as private. The Rust defaults are safer, because private elements are less likely to be exposed by mistake.

Control structures like `if` and `while` always require braces to specify their start and end. A control structure that only encompasses a single line cannot be expressed without braces. This feature of C/C++ has hardly simplified development, but can cause logic errors when a one-line `if` or `while` statement is later extended by a second line. If the programmer forgets to add braces, the second line is always executed independent of the condition. This has already been the root cause of a serious security vulnerability in the recent past [28]. Always requiring braces guards against such mistakes.

Rust actively discourages the use of mutable global variables by marking them as *unsafe*. This happens due to Rust's goal of preventing data races: If two threads simultaneously modified the same global variable, the result would be undefined. Therefore, global variables are only safe if they are set once as immutable ones or

if guarded by a thread-safe lock that serializes access. In all other cases, any code that accesses a mutable global variable must be enclosed in an `unsafe` block.

2.2.3 Expressions and Statements

Both C/C++ and Rust belong to the class of *expression-based languages*. The entire code in expression-based languages consists of *expressions* and *statements*. An expression always yields a value, which can be assigned to a variable. For instance, every mathematical operation is an expression. In contrast to that, a statement never returns a value, but has side effects.

C/C++ has originally been designed with a clear distinction between both categories. First and foremost, control structures like conditionals (`if`), loops (`for`, `while`), and jumps (`goto`) are statements whereas all operators for variables are expressions. However, this also includes the assignment operator. While it is often used as a statement with the side effect of assigning a value to a variable, it also yields this value as a result. Therefore, C and C++ allow writing `x = y = 1` to assign the value `1` to both variables `x` and `y` simultaneously. On the contrary, this is also the reason why the compiler accepts both `if (x = 1)` and `if (x == 1)`, a common typing mistake. While the former variant assigns `1` to the variable `x` and always evaluates to true, the latter one correctly checks `x` for equality with `1`. Doing this mistake can have serious security consequences, such as the one illustrated in [29]. Another exception in the C/C++ language design is the ternary operator. It allows writing conditional expressions such as `x = y > 5 ? 1 : 0`. However, this duplicates the `if` statement, only for the sake of maintaining the distinction between expressions and statements. The same example code can be written using an `if` statement: `if (y > 5) { x = 1; } else { x = 0; }`

Rust features a clean design that puts a focus on expressions. Rust's only existing statement, in the sense that it returns no value at all, is the `let` keyword to introduce a variable and optionally assign it to an initial value. Everything else is an expression yielding a value. That value may be of the empty type `()` though, which is Rust's way of specifying expressions with no meaningful return value [30]. This design has many consequences:

- Conditionals and loops can be used just like their counterparts in C/C++, but also as expressions. The conditional expression example above in Rust would be written as `x = if y > 5 { 1 } else { 0 }`. Rust does not need an extra ternary operator for this.

- Rust can implement the assignment operator in a way that it yields the empty type `()`. This does not allow the `x = y = 1` syntax from C/C++ to assign two variables at once. However, it prevents assignments inside conditionals and therefore guards against mixing up the assignment operator `=` and the equality operator `==`.

- The last line of a function (or generally any scope) can return a value simply by specifying an expression. For example, `return x;` as the last line of a C/C++ function can be replaced by an even simpler `x` (without semicolon) in Rust. Anyway, the `return x;` syntax is still supported for returning early in a function.

It should be noted that both C/C++ and Rust allow expressions to be turned into statements by discarding their result. This can have security implications though when a return value of a function must be checked for success and this is forgotten. Therefore, Rust allows marking individual return types with the `must_use` attribute to trigger a warning when a result of that type is discarded. By default, this applies to the generic `Result` return type.

2.2.4 Arrays, Slices, and Strings

Just like in most other programming languages, Rust supports handling contiguous objects of the same type and calls these *arrays*. Elements of an array can be addressed by their index. However, while C/C++ simply calculates and accesses a memory address based on the index (valid or not), Rust first checks whether the desired index is within the boundaries of the array. If that is not the case, program execution is stopped with an unrecoverable error (called a *panic*). This security feature may incur a little overhead at runtime, but guards against one type of buffer overflows.

Slices are specific to Rust and reference a part of an array. The part is described by a memory reference to the starting byte and a length field. Having such a basic type at the language level has a substantial security advantage, because Rust can again use the length field to bounds-check each access and prevent buffer overflows.

Consequently, slices also form the basis for a safe low-level type to represent strings (called `str`). In contrast to that, traditional C strings are arrays of characters terminated by a NUL character. To write safe applications, C developers are required to perform length checks manually, which can be cumbersome calculations prone to errors. Forgotten or improperly done length checks are a frequent cause for security vulnerabilities through buffer overflows. Therefore, the overhead of bounds-checking slices and strings in Rust is generally tolerated for the added security. Several string classes have been implemented in C++ that are not prone to the security problems of C strings. However, with no mutually agreed standard at the language level, these are hardly universally usable in every project. They also rely on dynamic memory allocations, which are often avoided for developing operating system kernel code.

2.2.5 Generic Programming

Many programming languages support a technique called *Generic Programming*, which enables developers to write universal algorithms once that can later be instantiated for many types. A common example is an array that does not have a

fixed size, but dynamically grows with the number of elements added to it. The standard libraries of C++ and Rust provide such a universal container for any type under the name *vector*.

Both C++ and Rust support Generic Programming, however their implementations are vastly different. C++ provides the *template* system to develop universal functions and classes. When instantiating a template, the compiler tries to expand the type placeholders into the actual types. It may be unsuccessful at doing so, for example when code deep within the template performs mathematical operations, but the template is instantiated with a non-numeric type. In this case, the compiler outputs a complex error message referring to the code line of the mathematical operation. If the user only knows about the template specification, but not its concrete implementation, this error message can be confusing and makes it hard to track down the problem to the non-numeric type. C++ also supports *specialization* to let a developer provide a different or optimized template implementation for a specific type. Furthermore, the C++ template system does not only support placeholders for types, but also for values. Just like type placeholders, expressions given to these values are evaluated at compile-time. All these properties of C++ templates can be used together for Template Metaprogramming (TMP), a powerful technique to implement algorithms executed by the compiler during code generation. TMP is considered turing-complete, however it is a mere byproduct of the C++ template features and lacks decent debugging capabilities. Development and usage of TMP code is prone to complex and unhelpful error messages by the C++ compiler.

Rust's take on Generic Programming is inspired from the type checking in Haskell and ML languages [9]. In contrast to C++, the declaration of a generic function or structure has to specify each interface a type needs to implement. The compiler only accepts instantiations with types that implement all required interfaces and otherwise cancels the build process early with a descriptive error message. As a consequence, the generic code may also call no functions that depend on an interface not specified in the declaration. Stable releases of Rust do not yet support any way for generic code to provide a different or optimized function implementation based on type constraints. However, such a specialization feature is being worked on and already available in nightly builds of the Rust toolchain [31].

The interfaces are called *traits* in Rust. The simplest traits serve as markers and do not specify any other constraints. An example for this is the `Copy` trait. When a type implements `Copy`, it indicates that an independent duplicate of it can be created simply by copying its bytes. For instance, every primitive integer type implements `Copy`. A vector (`Vec`) does not, because a bytewise copy of it would duplicate memory references, so they still point to the original memory addresses afterwards. Traits may also specify functions a type needs to implement or constants it needs to provide. An example for both is the `Add` trait, which requires an `add` function to perform the addition and the resulting type of the addition from every implementer. This allows for the implementation of additions of different integer types, but also of more advanced structures (like `SystemTime` + `Duration`).

Overall, C++ and Rust are hardly comparable in the discipline of Generic Programming. The template system of C++ generally provides more features, among them metaprogramming, but these come at the expense of debuggability and type safety. On the other hand, generic code in Rust is based on precise interface requirements and the Rust compiler provides useful error messages during the development. Either system can provide advantages and disadvantages based on the situation.

2.2.6 Pattern Matching

To support a structured handling of multiple cases, Rust provides the concept of *Pattern Matching* using the `match` keyword. It is roughly comparable to the `switch` statement in C/C++, however Rust's Pattern Matching is more powerful and has its roots in a similar concept from OCaml and SML [9].

Rust's implementation of structured case handling starts with safer defaults. Whereas the code for a C/C++ case label is executed until a `break` statement, the handler for a case in Rust's `match` statement exits at the next case. This *fall-through* feature of C/C++ can be used to support handling multiple cases with the same code. On the other hand, an undesired fall-through due to a forgotten `break` easily happens, and has already been the cause of security vulnerabilities in the past [32]. As long as a developer does not generally forbid fall-throughs for a project, the C/C++ compiler cannot catch the erroneous ones.

Rust provides an extended syntax for structured case handling. To let a single code block handle multiple cases, these cases can be combined with the pipe symbol `|` (e.g. `1 | 2`) or by matching a range of values (e.g. `2 ... 7`). The compiler always checks whether a `match` block addresses all possible cases, and otherwise stops the build process. However, the underscore symbol `_` may be used to provide a default handler for all unmatched cases.

Another useful feature of Pattern Matching is the *destructuring* of compound types like `enum`s. This is exemplified in Listing 2.1 for the `Result` type. Rust's `Result` type is an `enum` usable as the return value of a function to either indicate success along with a value or failure along with error information. Destructuring is used to access the value and error information in a `match` block.

```
match result {
    Ok(val) => println!("Success and the value is {}", val),
    Err(e) =>  println!("Failure with error {}", e),
}
```

Listing 2.1: Exemplary destructuring of a compound type in a `match` block

Destructuring is not limited to `match` blocks though. It may also be used in `if` blocks together with a `let` keyword. Taking Listing 2.1 again, in case only the `Ok` result is interesting, it can be checked by writing `if let Ok(val) = result`.

Finally, Pattern Matching supports several additional features not covered here. For these, the reader is referred to [33].

2.2.7 Ownership, References, and Borrowing

On the one hand, Rust aims to be a compiled systems language with deterministic memory management and full control over stack and heap allocations, just like C and C++. On the other hand, a second goal of Rust is to guarantee the validity of memory references at all stages of a program in order to prevent common mistakes in C/C++ code. This latter goal has only been accomplished by languages like C# and Java in the past. However, they do that by using a Garbage Collector that looks for no longer used memory from time to time, incurring an overhead at runtime and making memory management non-deterministic.

To fulfill both goals at the same time, the Rust programming language features the unique *Ownership* model. This model guarantees memory safety without needing a Garbage Collector. Instead, it imposes three basic rules on the usage of variables, and these rules are checked at compile time [34]:

1. Each value in Rust has a variable that's called its *owner*.
2. There can only be one owner at a time.
3. When the owner goes out of scope, the value will be dropped.

The far-reaching implications of these rules are best illustrated by some examples. For instance, a dynamically sized vector of 32-bit integers called `vec1` with the two elements `42` and `1337` can be initialized by writing

```
let vec1: Vec<i32> = vec![42, 1337];
```

This allocates memory for the vector on the heap and defines the variable `vec1` as its owner. C programmers would now need to manually call `free` to deallocate the associated memory when it is no longer needed. They have to take care that `free` is only called once, but also in every exit path of the function. Otherwise, memory is either deallocated twice or not deallocated at all, resulting in possible heap corruptions with security implications or memory leaks. A Rust programmer does not need to care about this, because rule 3 guarantees that the associated memory is deallocated as soon as the owning variable goes out of scope. The deallocation takes place automatically at every exit path of the function. Due to rule 2, it is also guaranteed that the same memory is never deallocated twice. This is still comparable to the RAII pattern of C++, which also allows for an automatic resource deinitialization when a variable leaves its scope. However, in contrast to Rust, RAII is just one possible pattern in C++ and not enforced at the language

level. A usual consequence in C++ programs is that some variables are deinitialized automatically while others need to be freed manually.

In the Rust example code, the variable `vec1` may now be passed to another function `process_vector` declared as

```
fn process_vector(input_vec: Vec<i32>)
```

When this function is called as `process_vector(vec1)`, Rust's Ownership model causes the value of `vec1` to be *moved* to `input_vec` inside `process_vector`. In other words, `input_vec` *consumes* the variable `vec1` and the ownership for the underlying memory is transferred. Due to rule 2, the variable `vec1` cannot be used anymore after this ownership transfer, and the compiler stops with an error when attempting to do so. Unless the ownership is transferred yet another time in `process_vector`, all associated memory for the vector is deallocated when leaving `process_vector` and `input_vec` goes out of scope. This behavior is characteristic to Rust and different to what other programming languages do in such a situation. For instance, a similar `process_vector` function in C++ may implicitly allocate a new vector `input_vec` and copy all data from `vec1` into `input_vec`. This could be a very memory and time consuming operation if the vector contained many elements. C++ supports passing a reference to `vec1` instead by using the `&` character. However, this single character can easily be forgotten without any compiler warning, possibly resulting in huge performance penalties [35].

There are many cases though, where a variable shall stay usable after passing it to a function. Like C++, Rust supports the concept of *References* for these cases. By writing an ampersand in front of `Vec<i32>` in the function declaration above, the function takes a reference to a variable instead of consuming it. The call also changes to `process_vector(&vec1)`. Passing a variable by reference does not cause a transfer of ownership. Instead, the variable is *borrowed* from the caller until the called function returns. The variable `vec1` remains the only owner of the associated memory and is still responsible for deallocating it when the variable leaves its scope.

It is important to note that a borrowed variable passed like this is immutable, unlike a consumed one. By exchanging `&` with `&mut`, mutable references can be created. However, this triggers the *Borrow Checker* of the Rust compiler: This step verifies that there is never more than one mutable reference to a variable at the same time. It also prevents mutable references from being created while other immutable references to the same variable exist. These additional rules contribute to Rust's goal of preventing data races: With only one mutable reference to a variable at any time, two threads cannot modify the same value simultaneously. Users of immutable references can also be sure that the referenced variable does not change while the reference is held. In any case, multiple immutable references to the same variable are no problem, because none of the users can change the variable's value [36].

Rust's references share some similarities with raw pointers from C/C++. However, the strict rules enforced by the Rust compiler ensure that references always point to initialized memory of existing variables. On the other hand, pointers may be null, reference memory that has already been freed, or memory never initialized at all.

Finally, Rust also supports duplicating a variable to have two copies of it. This is called *cloning*. However, unlike C++, expensive cloning operations never happen implicitly in Rust. The programmer explicitly has to call the `clone` function to indicate that a copy of the variable is desired.

The examples above are deliberately based on the `Vec` type, which allocates heap memory and is expensive to clone. Rust's behavior of preventing implicit copies makes sense here. However, another category are the *primitive types* (such as `bool`, `char`, or `i32`), which are entirely allocated on the stack. Cloning them is a lightweight operation. The types indicate this by implementing the `Copy` trait. Therefore, passing such variables to a function or assigning one to another variable creates an implicit copy instead of transferring ownership. This obviates the need for explicit `clone` calls and makes handling these types no harder than in C and C++. The added simplicity comes at no expense to performance, because it is limited to lightweight types implementing the `Copy` trait.

Despite the advantages of Rust's rules, there are corner cases impossible to implement as long as these rules are enforced at compile time. A typical example is the doubly-linked list data structure. Every node of a doubly-linked list shall contain a mutable reference to the previous and the next element. Using two `&mut` references per node would violate Rust's rule of having no more than one mutable reference to any variable at the same time. Nevertheless, a doubly-linked list can be developed without resorting to unsafe Rust functions. For this situation, the Rust standard library provides the `Rc` container. It implements *reference counting*, meaning that every clone of it increments an internal counter and every dropped clone decrements that counter. When the counter reaches zero, the associated memory is automatically freed. Thread safety is maintained by allowing access to the referenced variable only through the provided `borrow` and `borrow_mut` functions. These implement the borrow-checking rules from above at runtime. When any rule is violated, for instance when two threads try to modify the same doubly-linked list node simultaneously, program execution is stopped with an unrecoverable error (called a *panic*). Borrow-checking at runtime incurs some overhead, however the added safety often outweighs that.

2.2.8 Foreign Function Interfaces

As Rust implements some features in a unique way (such as Generic Programming), it requires its own Application Binary Interface (ABI) to link different pieces of Rust code together into a single compiled binary. Currently, this ABI is not static

and subject to change with every Rust release[10]. Moreover, it is incompatible with the popular ABIs defined by the C programming language for each processor architecture. Nevertheless, Rust implements a Foreign Function Interface (FFI) to let Rust code interact with C code. This is especially relevant within this thesis, because existing HermitCore components written in C are gradually replaced by components written in Rust. These Rust components need to be called from C functions and vice-versa. For the former case, Rust provides two options:

- Declaring a Rust function with `extern "C"` compiles it using the C ABI defined for the current processor architecture. It can then be called by Rust and C code just like any other function written in C. While only a single C ABI (also known as *calling convention*) has been defined for the x86-64 architecture targeted within this thesis, multiple different ABIs exist for other architectures. For a complete list of the ones supported by Rust, the reader is referred to [37].

- Adding the `no_mangle` attribute to a function turns off the so-called *name mangling*. By default, the Rust compiler encodes the library name, module hierarchy, and a hash of the type information in an exported function name. For example, an exported function called `abort` defined in a module `scheduler` of the library `hermit_rs` gets the long and non-memorable mangled name `_ZN9hermit_rs9scheduler5abort17h25817f4396886865E`.

  When Rust code interfaces with other Rust code, name mangling happens in the background and is of no concern. It ensures that called functions accept the supplied parameters and prevents conflicts when two `abort` functions are exported from different modules. However, mangled names are impractical when a Rust function shall be called from C code. If `no_mangle` was used for the `abort` function above, this function could be called from C code by doing a usual `abort()` call.

Currently, not all possible C functions can be rewritten in Rust. For example, Rust does not yet support writing C-compatible functions that take a variadic number of parameters[11]. A typical representative of this class of functions is `printf`.

The FFI also supports calling C functions from Rust code. For that, every external C function needs to be declared with its parameters and return value in an `extern "C"` block. It can then be called almost like any other Rust function. The main difference is that calls to C functions always need to be enclosed in an `unsafe` block, because Rust cannot make any safety guarantees about such external code.

C functions do not support Rust references, but often deal with raw pointer addresses. These are also required to facilitate low-level hardware development in Rust. Therefore, Rust implements support for raw pointers as well. Every reference, or integer of type `usize`, can be freely converted into a raw pointer. However,

[10] `https://github.com/rust-lang/rfcs/issues/600`
[11] `https://github.com/rust-lang/rust/issues/44930`

dereferencing such a pointer can only happen in an `unsafe` block, because Rust cannot make any guarantees about the memory behind a pointer. When reading from a raw pointer in Rust, it is usually good practice to perform some sanity checks and then convert the pointer to a Rust reference in a short `unsafe` block. Further accesses can then happen by using the reference without additional `unsafe` blocks.

Finally, writing an operating system requires another class of functions to be implemented, namely *interrupt handlers*. This thesis targets the x86-64 processor architecture, so handling interrupts for this architecture is explained in the following: As soon as an interrupt occurs, program execution is transferred to the interrupt handler, without saving the context of the code currently being executed. Therefore, every interrupt handler needs to save the registers it modifies and restore them when completing its work. It also needs to return with the `iret` instruction as opposed to `ret` to indicate that interrupt handling has completed. Many programming languages require such code to be written in assembly language, because they lack native support for writing interrupt handlers conforming to these requirements. However, Rust implements the `abi_x86_interrupt` feature in nightly builds. When it is activated, functions can be declared with `extern "x86-interrupt"` to mark them as interrupt handler functions. Processor registers modified within such functions are correctly saved and restored, and `iret` is used for exiting.

2.2.9 Crates for Operating System Development

When developing code that is later going to be used by multiple programs, most programming languages offer a way to write it once without the need to duplicate it for every program. In C, this is known as creating a *library* while Java calls this a *package*.

Rust also supports this method and calls the result a *crate*. Crates can be kept private or published online for others to reuse. Many open-source crates are available on the Rust repository Crates.io and come with a well-defined and well-documented interface. Including such a crate in a Rust project is often as simple as adding a single line to the Cargo configuration file with the name of the crate and the desired version. It is then automatically downloaded and compiled during the build process. However, Cargo also supports using local crates.

Numerous crates are available on Crates.io, which aid in the development of an operating system in Rust. The following have been used within this thesis:

- The **bitflags** crate enables Rust code to comfortably manipulate bitmask flags in integer types. These are often needed when programming hardware or interfacing with functions written in C. A set of flags is defined through a `struct` containing each flag as a constant. Currently, this crate is among the top 10 most-downloaded crates on Crates.io, and also a popular dependency of other crates.

- The **lazy_static** crate works around a limitation of global variables in Rust. By default, they need to be initialized with a constant value at declaration.

This excludes dynamic memory allocations and any kind of code executed at runtime. This crate offers a way to initialize individual global variables at runtime on their first access, thereby enabling the use of dynamically computed values. At the time of writing, it is also among the top 10 most-downloaded crates on Crates.io.

- The **raw-cpuid** crate offers comfortable Rust functions to examine the result of the x86/x86-64 `cpuid` instruction. This is commonly used to gather information about the installed processor and its features. As such, it can be considered fundamental for developing an x86-64 operating system in Rust.

- The **x86** crate from the same developer offers several convenience functions and data structures to comfortably program a 32-bit x86 or 64-bit x86-64 processor at hardware level. This includes setting up the Global Descriptor Table (GDT) for memory segmentation and Interrupt Descriptor Table (IDT) for interrupts, Task State Segments (TSSs) for multitasking as well as Control Registers (CRs) and Machine-Specific Registers (MSRs). At the time of writing, a fork of it called **x86_64** is available, which exclusively targets the 64-bit x86-64 architecture and is under more active development. However, the former crate is more popular at the time of writing, has been deemed sufficient for this thesis, and is therefore used for HermitCore.

- Finally, the **spin** crate implements some synchronization primitives based on spinning. They do not need any operating system support and can therefore be used for developing an operating system. However, only the specialized `Once` and `RwLock` variants have been used within this thesis. The former one can be used to run a code path only once (e.g. for initialization) and skip it on further calls. The latter one provides a lock that enables multiple readers but only a single writer to access a resource. Further locks have been implemented manually within this thesis to account for the specific requirements of HermitCore.

3 Implementation

Based on the concepts introduced in Chapter 2, key components of the HermitCore operating system have been successively ported to the Rust programming language. This chapter details their Rust implementation and integration into the existing set of components written in C.

3.1 Goals

In order to benefit from the advantages of Rust, HermitCore components have not been ported by simply conducting a one-to-one translation of the existing C code. Instead, the functionality of each component has been studied to create a Rust implementation that is compatible to the required C interfaces on the outside, but may work differently on the inside. This often resulted in a total rewrite of each component rather than a simple port. To make these rewrites as advantageous as possible, the following additional goals have been defined:

- As a start, the Rust components shall only target the Unikernel mode of HermitCore. This mode can easily be tested in any virtual machine and it builds the foundation for all other features. Implementing and verifying support for the Multi-Kernel mode or specialized virtual machine hypervisors (like *uhyve*) is beyond the scope of this thesis. However, the Rust components shall be developed flexible enough to account for a later implementation of these features.

- The amount of Rust code marked as *unsafe* shall be kept as small as possible. The only exceptions are global variables, which are designed to be exclusively mutated by the single kernel thread on the boot processor. Guarding them by synchronization primitives would guarantee safe Rust code, but imply that they are mutated concurrently.

- Safe and maintainable code shall be preferred even if it could have a negative impact on performance or memory consumption. The actual performance of the final code is measured in Chapter 4. Additionally, as an operating system for HPC applications, HermitCore is not constrained by memory limitations like an embedded operating system would be.

- The Resource Acquisition Is Initialization (RAII) pattern shall be used wherever possible. Instead of relying on manual `kmalloc` and `kfree` calls, the

allocation and deallocation of kernel memory shall happen automatically by using the respective containers in Rust (such as `Box` or `Vec`).

- The x86-64 architecture-specific code shall refrain from implementing any code paths that are only relevant for older 32-bit x86 processors. Due to its MetalSVM heritage and traditional best practices, the HermitCore C code sometimes checks the availability of features that every x86-64 processor provides. It also implements some legacy features, which have been superseded by more modern equivalents on x86-64 processors.

- In case of error conditions, the operating system shall immediately halt with a Rust *panic* and a descriptive error message instead of propagating an error code to the caller. Without any form of user input supported in the kernel, most HermitCore errors are unrecoverable. As such, there is no added value but a loss of information when propagating an error code instead of halting directly.

3.2 Console Output

As the first step, the Rust code needs to be able to output messages on the console to give status information and aid with debugging. In HermitCore, this usually means transmitting these messages over a (physical or virtual) serial port with a terminal connected to the other end. Therefore, the HermitCore C code initializes the serial port early. It also integrates a standalone version of the popular `printf` function for kernel usage and implements some macros around it. These macros add support for different message levels (errors, warnings, information, etc.) and prepend each message with additional information, like an identifier of the current processor. The message level that shall be output by HermitCore is configured at compile-time.

Instead of an external `printf` implementation, the Rust code uses the built-in formatting features of the programming language. They support constructing a message out of a formatting string with placeholders and arguments similarly to `printf`. However, Rust already interprets the formatting string at compile-time. This prevents common mistakes of `printf`, like unresolved or incorrectly typed placeholders. Using the formatting features for transmitting messages only requires an empty structure implementing the `write_char` and `write_str` functions of the `fmt::Write` trait. The former function simply forwards to `uart_putchar` of the C serial port code while the latter function calls `write_char` for each character. At a later stage, the `uart_putchar` function as well as the entire serial port initialization code has also been replaced by Rust code. The Rust version always outputs messages on the first serial port (address 0x3F8) with the maximum baudrate of 115200 bps. In contrast to that, the C code scans the PCI bus for serial port cards and initializes the first one with 38400 bps. This is a less likely configuration and requires a more complicated virtual machine setup.

To allow for basic debugging of computers that do not have a serial port, the Rust version also implements a screen output of all messages. This is currently not provided by the HermitCore C implementation.

3.3 Build System

The CMake build system is responsible for building the HermitCore kernel, all dependent libraries, and HermitCore applications. As the Rust code is added next to the existing C code, the Rust toolchain needed to be integrated into a CMake environment. For all other programming languages, this has been done by writing CMake configuration files for their compilers and defining the source files of each module in CMake. However, to leverage the advantages of Rust's Cargo build tool, a `cargo` call has been integrated as a custom target *hermit_rs* in the CMake configuration files instead. The HermitCore Rust code is then compiled by Cargo and can therefore easily import crates like any other Rust project. After building the Rust code, the added CMake configuration merges the resulting library with the HermitCore C library into a single *libhermit.a* file. This library is then picked up by the CMake build process for building HermitCore applications.

At a later stage of development, the CMake build process has been enhanced by the addition of prebuild steps. One of them is assembling the SMP boot code and writing it into an array of a Rust source file, so it can be used for the initialization step in Section 3.5.5. This is accomplished by adding a custom command in CMake, which is defined to output that source file and set as a dependency of *hermit_rs*. While external code is usually imported through `use` statements, Rust also supports an `include!` macro to include the generated Rust source file and make use of the SMP boot code array.

Finally, another custom target *doc* has been added to the CMake configuration file. This target calls `cargo` with the `rustdoc` argument and additional parameters to generate a code documentation of the entire HermitCore Rust source code, including all private members.

3.4 Memory Manager

As a low-level part of HermitCore without any dependencies on other parts, the Memory Manager has been chosen as the first candidate to be ported to Rust, in particular its x86-64 Paging component. After that, the porting work has continued on the Physical and Virtual Memory management.

3.4.1 Paging

As illustrated in Section 2.1.2, the Paging component maps Virtual Memory addresses to Physical Memory addresses. Apart from an initialization function, it only makes the following functions available to other kernel code:

- `__page_map` maps a Virtual Memory address to a number of contiguous 4 KiB pages of Physical Memory at a specific address. If desired, it sends an IPI to all other processors to let them clear their TLBs.
- `page_unmap` removes such a mapping.
- `virt_to_phys` determines the Physical Memory address for a Virtual Memory mapping of a specific address.

From these functions, only `__page_map` has been considered important. Pages do not need to be explicitly unmapped, because an abandoned page mapping is of no concern in a single-address-space operating system without memory protection. Instead, abandoned mappings can simply be overwritten by the next `__page_map` call. Additionally, `virt_to_phys` is only used by network device drivers, which are not required for HermitCore's Unikernel mode.

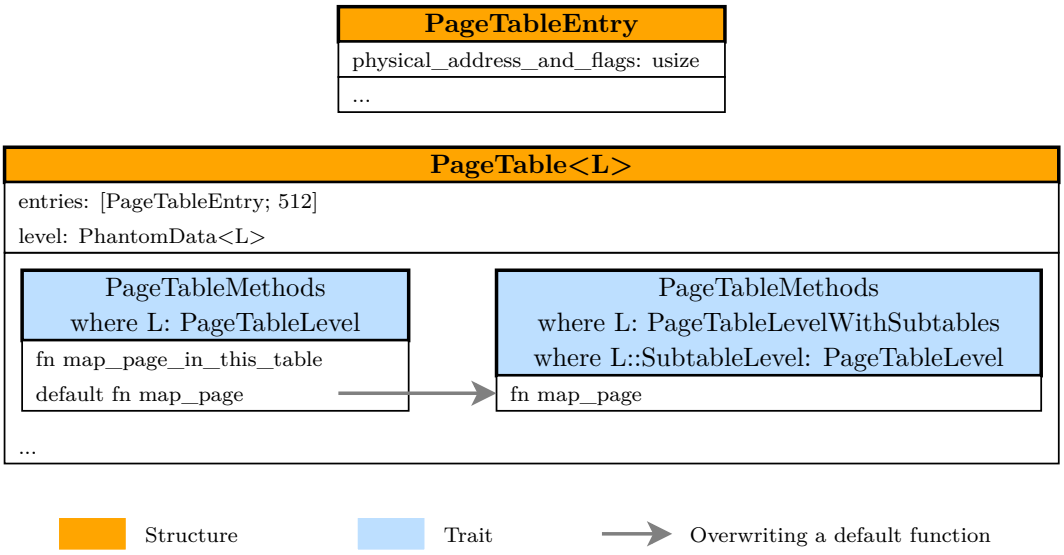
The C implementation of `__page_map` relies on the self-referencing entry in PML4. Due to that, all page tables appear consecutively in the Virtual Memory address space starting at 0xFFFF_FF80_0000_0000. The implementation makes use of this fact to treat all tables as a single large array of page table entries. An iterative algorithm is then used to walk through PML4, PDPT, PD, and PT. During this walk, memory is allocated for a new page table if needed until the mapping is finally set in the page table entry.

This algorithm is very short and efficient. However, treating all page tables as a single large array provides no protection against accessing non-existing table entries. Furthermore, the implementation only supports 4 KiB pages while larger pages are beneficial to HPC applications.

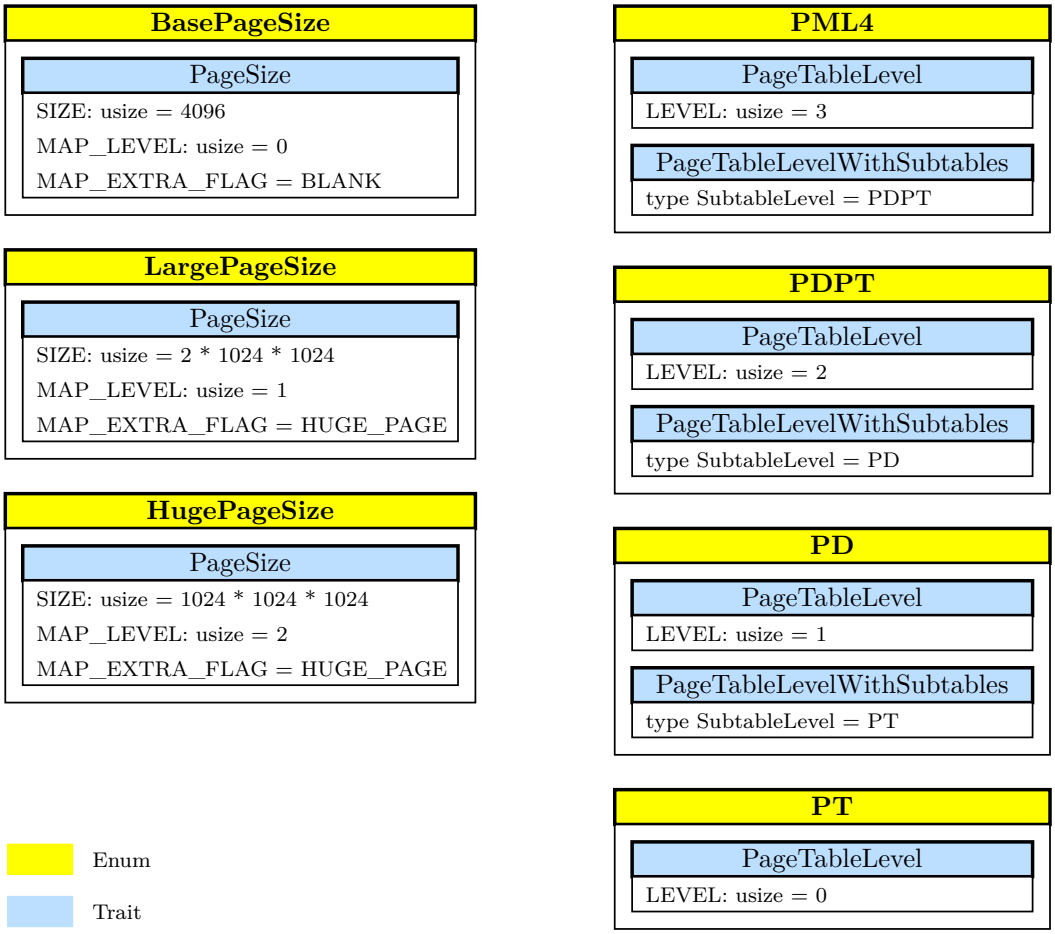
Therefore, the Rust implementation of the Paging component follows a different design. It is centered around the `PageTable` structure, which manages an array of 512 `PageTableEntry` elements. Using Rust's Generic Programming capabilities (cf. Section 2.2.5), `PageTable` can be used for PML4, PDPT, PD, and PT tables. These 4 types are implemented as empty `enum`s. To serve as the type parameter for `PageTable`, the `PageTableLevel` trait is implemented for each of them. PML4, PDPT, and PD also implement the `PageTableLevelWithSubtables` trait to indicate that they have a subtable and the type of that subtable. This trick of leveraging Rust's typing system for a safe handling of the page table hierarchy is further detailed in [38].

A similar Generic Programming design is used to support different page sizes. 4 KiB, 2 MiB, and 1 GiB pages are represented by `BasePageSize`, `LargePageSize`, and `HugePageSize` `enum`s. All of them have in common that they implement the `PageSize` trait.

Furthermore, a `PageTableMethods` trait is defined to specify functions that shall be implemented by all 4 variants of a `PageTable` structure. The trait is also needed to make use of Rust's experimental specialization feature. This allows for different function implementations depending on whether a table has subtables (additional



(a) Simplified `PageTable` and `PageTableEntry` structures with implemented `PageTableMethods` trait



(b) Empty enums with implemented traits as type parameters for the paging functions

Figure 3.1: Structure of the Paging implementation in Rust

`PageTableLevelWithSubtables` trait) or not (only a `PageTableLevel` trait). Two notable functions are presented in the following:

- `map_page_in_this_table` sets a page table entry for the Virtual Memory to Physical Memory mapping in the current table. This function only needs a single generic implementation for all page table types. It is internally used by `map_page`.
- `map_page` checks the current table and the page size to decide whether the page needs to be mapped in this table or in a subtable. In the former case, it just calls `map_page_in_this_table`. However, in the latter case, any not yet existing subtable is created and the `map_page` implementation for the subtable is called recursively.

  By using specialization, a default implementation is provided for all page table types that implement `PageTableLevel`. This implementation is overwritten for all types which also implement `PageTableLevelWithSubtables`. Consequently, there is one implementation shared by PML4, PDPT, and PD and another implementation just for the last page table PT. The latter implementation always calls `map_page_in_this_table`, because there is no other option at the last page table level.

The discussed elements of the paging implementation are illustrated in Figure 3.1.

Finally, `PageTable` provides the convenience function `map_pages` to call `map_page` in a loop for mapping a range of pages. It should be noted that all presented functions are generic over the page size. Therefore, this design of the Paging component enables a straightforward mapping of pages in the 4 KiB, 2 MiB, and 1 GiB page sizes supported by the x86-64 architecture. It can also easily be enhanced by a future 5th page table level as soon as processors support it. However, the mapping algorithm uses a recursive instead of an iterative approach. Whether this incurs a significant performance cost is to be determined in Chapter 4.

For compatibility with the existing HermitCore C code, `__page_map` has been implemented as an `extern "C"` function that calls `map_pages` with `BasePageSize` to always map 4 KiB pages. In a later stage of development, this compatibility is no longer necessary, so the `__page_map` function is replaced by a generic Rust function supporting all page sizes.

3.4.2 Physical and Virtual Memory Management

The management of Physical and Virtual Memory in doubly-linked lists has proven to be suitable for HermitCore. However, studying the C source code has revealed no reason why the Physical Memory Manager needs to keep track of *free* memory blocks while the Virtual Memory Manager needs to manage *used* blocks. Therefore, the Rust implementation of both memory management components is based on a generic `FreeList` structure that manages free blocks of memory. Each node of the `FreeList`

marks the start and end of a free memory region. Nodes are sorted from the lowest to the highest managed memory address. The implemented `FreeList` structure itself is based on a generic doubly-linked list data structure. Rust currently does not come with a suitable data structure for this task, so a generic `DoublyLinkedList` structure has also been implemented within this thesis. The design of both components is described in the following.

Apart from a way to initialize it with the managed memory address range, a generic Free List needs to provide two functions at minimum:

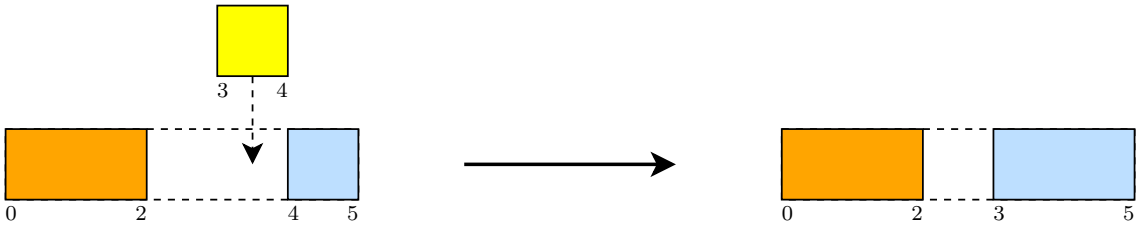
- The `allocate` function is used to request a contiguous range of memory of a specific size. The function traverses the sorted linked list from the lowest to the highest memory address. As soon as it finds a region of free memory large enough, this region is shrunken by the specified size, starting from the left. Mathematically expressed, the `start` field is incremented by the specified size. If the resulting size of the list node is zero afterwards (`start == end`), the node is removed from the list.

- The `deallocate` function does the opposite and is called when giving back a contiguous range of memory previously requested with `allocate`. It has to handle more cases though: The returned memory range may extend an existing list node to the left or right without gaps. In that case, the existing node can simply be adjusted (Figures 3.2a and 3.2b). This adjustment may lead to a node completely filling the gap between two previously separate nodes. Consequently, the function can reunite both nodes into a single large node (Figure 3.2c). However, if no node can be extended, a new node needs to be created and inserted into the list at the right position (Figure 3.2d).

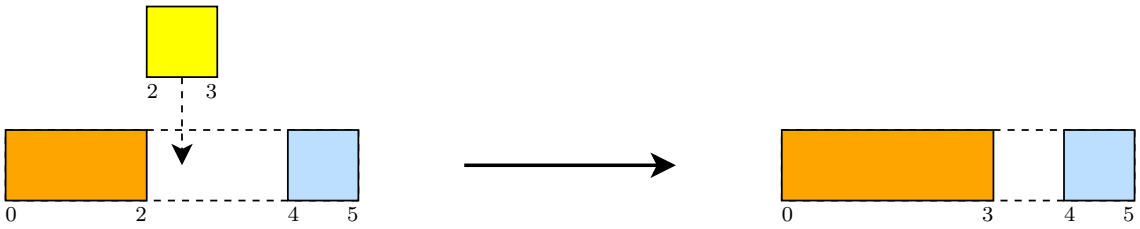
In addition to that, the `FreeList` structure implemented within this thesis also provides an `allocate_aligned` function to allocate memory aligned to a specified memory address boundary, and a `reserve` function to mark a memory region as reserved (e.g. by memory-mapped hardware devices) and unusable for memory allocations.

This design of the `FreeList` structure defines the requirements for the underlying list data structure. First of all, the list must be iterable in the order of the nodes. This is possible in $\mathcal{O}(1)$ by following the `next` references of a doubly-linked list. The same operation complexity also applies to most other data structures though.

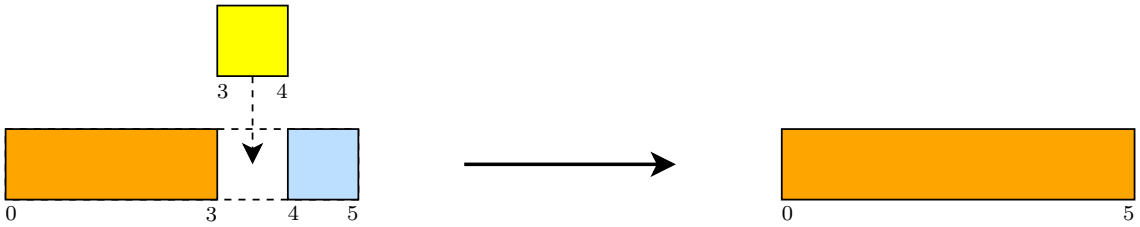
However, the more important part is the addition and deletion of nodes. The data structure must support adding a node before or after the current iterated node, as well as deleting the current iterated node. None of the data structures currently provided by Rust are optimized for inserting elements in the middle. Furthermore, removing individual elements either incurs another traversal of the data structure (Rust's `Vec` and `VecDeque`) or is not supported at all (Rust's `LinkedList`). Using a doubly-linked list, these operations are also possible in $\mathcal{O}(1)$. This is the reason for implementing a generic `DoublyLinkedList` structure instead of relying on Rust-provided data structures.



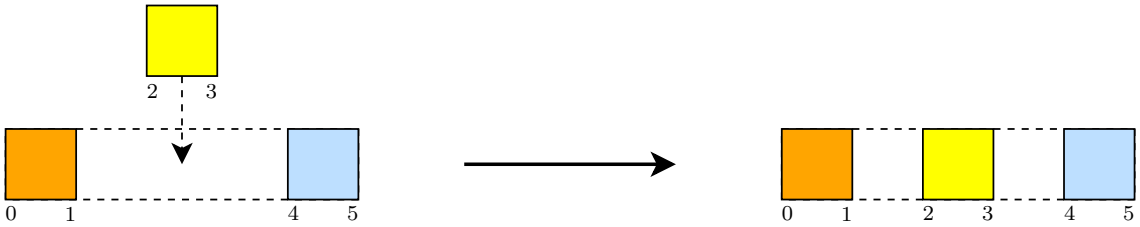
(a) Extending an existing node to the left.



(b) Extending an existing node to the right.



(c) Reuniting two nodes into a single large one when a new node completely fills the gap.



(d) Inserting a new node when it cannot extend any existing one.

Figure 3.2: Possible Free List cases when deallocating memory.

In a doubly-linked list, each node has one reference to the previous node and one reference to the next node, hence the name. To perform the required operations, these references need to be mutable ones. However, this violates Rust's rule of having no more than one mutable reference to any variable at the same time. To work around this limitation, the `Rc` container has been used throughout the `DoublyLinkedList` implementation as described in Section 2.2.7. This allowed for an implementation of a generic doubly-linked list structure without using a single `unsafe` block.

Using the generic `FreeList`, the implementation of the actual x86-64 Physical and Virtual Memory Managers has been straightforward. The Physical Memory Manager determines the available RAM in the computer and adds all usable regions to the Free List during initialization. The allocation and deallocation of Physical Memory in `BasePageSize` (4 KiB) granularity is then performed by forwarding calls to the `allocate` and `deallocate` functions of the `FreeList`.

The Virtual Memory Manager operates likewise. However, it is initialized to manage a single large region starting after the kernel image and ending at address 0x1_0000_0000. This encompasses a region of almost 4 GiB in size. It is important to note that the Virtual Memory Manager exclusively manages *kernel* memory in the Rust implementation of HermitCore, which is why a 4 GiB region should be sufficient for every case. As described in Section 2.1.2, the single HermitCore application requests memory by enlarging and shrinking its heap through the `sbrk` system call. Therefore, it only needs a single contiguous Virtual Memory region. By managing only Virtual Memory up to address 0x1_0000_0000, the region from 0x1_0000_0000 to 0x8000_0000_0000 is implicitly available as a single contiguous region. This amounts to approximately 128 TiB available for the application, which is far more than most current servers provide in RAM. The entire Virtual Memory Layout of the HermitCore Rust implementation is depicted in Figure 3.3.

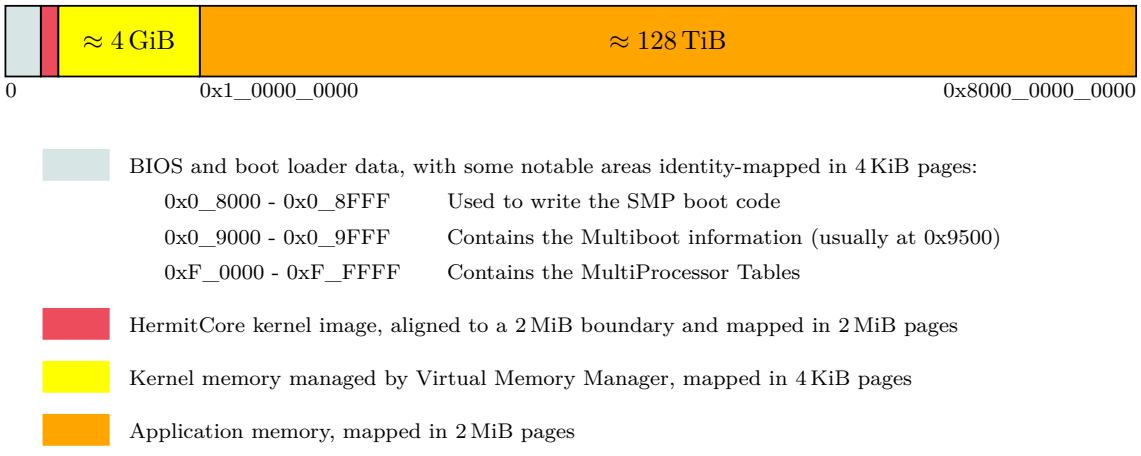


Figure 3.3: Virtual Memory Layout of the HermitCore Rust implementation

Finally, an architecture-independent Memory Manager is implemented and exposed to the rest of the kernel functions. To allocate kernel memory in 4 KiB granularity, it simply performs the following steps:

1. It verifies that no other processor currently uses the architecture-independent Memory Manager and then locks it to the current processor.
2. It allocates the required amount of Physical Memory using the architecture-dependent Physical Memory Manager.
3. It allocates the same amount of Virtual Memory using the architecture-dependent Virtual Memory Manager.
4. Finally, it maps the resulting Virtual Memory address to the resulting Physical Memory address using the architecture-dependent Paging component. This makes the memory available and usable by kernel code.

Deallocating kernel memory happens similarly.

3.4.3 Heap Allocator

The architecture-independent Memory Manager implemented in the last section already allows allocating and deallocating memory in 4 KiB granularity in theory. However, two tasks remain:

1. Rust components using the RAII pattern (like `Box`, `Rc`, and `Vec`) rely on an internal memory allocator provided by Rust to serve their memory requests. This allocator is only implicitly compiled into user-space applications written in Rust, but not when developing low-level kernel code. Therefore, the internal memory allocator must be implemented manually based on the architecture-independent Memory Manager from the last section. This is comparable to overloading the `new` and `delete` operators in C++.
2. The *chicken-egg problem* described in Section 2.1.2 is also relevant for the Memory Manager components implemented in Rust: As the used `FreeList` is based on a `DoublyLinkedList`, which itself uses an `Rc` container for each node, this container needs to be able to allocate memory. This is already the goal of Task 1. However, in addition to that, the `Rc` container also needs to allocate memory during the initialization of the Memory Manager for adding the first entries into the lists. A solution must be found to allocate these first entries without relying on the implemented Memory Manager.

To solve Task 1, the experimental `alloc`, `allocator_api`, and `global_allocator` features of Rust have been activated for the HermitCore code. They have then been employed to develop a Heap Allocator around the architecture-independent Memory Manager, which is used internally for the allocation and deallocation of

memory by Rust components. While the original HermitCore Heap Allocator in C comes with a *Buddy System* to efficiently manage fractions of a page, the current Heap Allocator implemented in Rust simply rounds up each allocation to the next 4 KiB boundary. This causes a waste of memory, especially for small allocations $\ll 4$ KiB. However, this has been considered acceptable within this thesis, because HermitCore is currently not constrained by memory limitations due to its design towards HPC systems (cf. Section 3.1). Furthermore, the majority of dynamic memory allocations from kernel code occur during the boot phase of HermitCore, so most *out-of-memory conditions* can be caught before the application is started. Apart from this, always allocating at least a single page with every dynamic memory request provides ideal conditions for testing the stability of the Memory Manager. This is further discussed in the next section.

The *chicken-egg problem* of Task 2 has been solved by enhancing the Heap Allocator developed for Task 1. During the early boot phase, Rust components now allocate memory from a preallocated static buffer instead of the list-based Memory Manager developed above. A global index into that buffer is incremented with every allocation. Requested memory blocks are not tracked, so deallocations do not return any memory. However, this simple Heap Allocator has the advantage of being available early during boot without any further initialization. Within this thesis, it has therefore been called the *Bootstrap Allocator*. When initializing the actual HermitCore Memory Manager, it is used for allocating the first entries of the Free Lists. As soon as the HermitCore Memory Manager is ready, the system switches over to it and further allocations no longer use the Bootstrap Allocator. Due to the short period of time, in which the Bootstrap Allocator is used, a size of 4 KiB has been deemed sufficient for its static buffer.

At this point, it would have been possible to implement the `kmalloc` and `kfree` functions based on the Heap Allocator to stay compatible with dynamic memory allocations done by the HermitCore C code. However, it has been decided to disable all C components doing dynamic memory allocations instead, and later reimplement these components in Rust. This decision has eventually lead to a complete rewrite of all HermitCore components in Rust.

3.4.4 Node Pool

An important detail of the `FreeList`'s implementation of `allocate` and `deallocate` in Section 3.4.2 needs special attention: If `allocate` finds a node that matches the requested size exactly, it removes that node from the list, thereby implicitly leading to a deallocation operation on the same Free List. The `deallocate` function is even more complicated in this regard: If a deallocated memory region cannot extend an existing region in the list, a node for a new region is allocated. Moreover, if a deallocated region perfectly fills the gap between two regions, one node is extended to encompass the entire large region while the other node is deallocated (cf. Figure 3.2c). However, the consistency of the Free Lists is not guaranteed in the middle of an allocation or deallocation operation. Therefore, additional allocations

and deallocations must be serialized and may not happen while a Free List operation is taking place.

This has led to the introduction of the *Node Pool*. The Node Pool is basically another doubly-linked list out of Free List nodes. Instead of allocating and deallocating new nodes in the middle of a Free List operation, these are drawn from or given back to the Node Pool. Allocations during Free List operations are prevented this way, because the nodes in the pool have already been allocated beforehand. Deallocations also do not happen, because nodes are not deleted, but unmounted from one Free List and pushed again to the Node Pool.

When allocating memory using the architecture-independent Memory Manager, the `allocate` function is called once for the Free List of the Physical Memory Manager and once for the Virtual Memory Manager. Depending on the state of the Free Lists, the Node Pool may contain 0, 1, or 2 additional nodes afterwards. As the pool is based on a doubly-linked list, it does not need any capacity adjustments and can accommodate an arbitrary number of additional nodes simply by chaining them together.

Deallocating memory using the architecture-independent Memory Manager leads to one `deallocate` call for the Physical Memory Manager's Free List and another one for the Virtual Memory Manager. In an extreme case, both deallocations perfectly fill the gaps between neighboring memory regions and 2 nodes are returned to the Node Pool. In the other extreme case, each deallocation needs a new node to track the deallocated regions. The Node Pool must provide at least 2 nodes to handle such a situation.

As a consequence, a `maintain` function has been implemented for the Node Pool. It ensures that the pool contains exactly 2 nodes. Missing nodes are allocated while surplus ones are removed and thereby deallocated from memory. This `maintain` function is called by the deallocation routine of the architecture-independent Memory Manager before any Free Lists are modified.

It should be noted that the decision not to implement a Buddy System in the previous section has helped to trigger these corner cases. With a Buddy System in place, most `allocate` calls would return a part of an already allocated page without modifying the Free Lists at all. However, the corner cases may still appear when an entire page is allocated or deallocated.

3.5 Hardware Initialization

After a Rust-implemented Memory Manager had been successfully developed and integrated into the HermitCore C code, the next step has been rewriting the fundamental hardware initialization code in Rust. The individual initialization steps and the characteristics of their implementations in Rust are presented in the following sections and later summarized in a diagram.

3.5.1 Processor Initialization

Step 1 of the hardware initialization is detecting the boot processor and its features. This has to happen as early as possible to adjust subsequent initialization steps to the detected processor, but also to report processor information on the console (see Section 3.2). A log file of the entire console output can then later be associated to the used computer system and assist in debugging hardware-specific problems.

The original HermitCore C code implements a single function to gather information about the features of the installed processor. It calls the x86-64 `cpuid` instruction directly and parses its output through bitwise operations until it is stored in a global structure variable. In contrast to that, the Rust code relies on the *raw-cpuid* crate introduced in Section 2.2.9. This crate internally calls `cpuid` and performs the required bitwise operations to gather feature information. The actual Rust code developed for HermitCore can then use descriptive crate functions instead of bitwise operations to check the existence of individual processor features. This makes it shorter and more readable than its C counterpart. Moreover, these comfortable abstractions are not expected to incur runtime costs due to the optimizations performed by the Rust compiler.

The processor detection is followed by the processor configuration. This is facilitated in Rust using the *x86* crate presented in Section 2.2.9. It mostly follows the HermitCore C implementation and basic x86-64 manuals like [39] to initialize the desired features. However, one of the goals in Section 3.1 is to avoid redundant code paths. Several such code paths exist in the HermitCore C code, partly because they used to serve a purpose on 32-bit x86 processors, but are no longer necessary for 64-bit x86-64 processors. This approach shall keep the resulting Rust code shorter and better maintainable than its original C equivalent. In particular, the following code paths during the processor configuration have been identified as redundant:

- **Checking support for the No-eXecute (NX) bit**
  This feature is guaranteed to be supported by all x86-64 processors. As a consequence, it is now enabled unconditionally and by the boot assembly code. This makes NX memory protection already available when mapping the first page with Rust code.

- **Checking the existence of a Floating-Point Unit (FPU) and basic SIMD extensions**
  All x86-64 processors come with an FPU and the MMX, SSE, and SSE2 SIMD extensions. This also includes the `fxsave`/`fxrstor` instructions to save and restore registers used by these extensions. There is no reason to ever use the older `fsave`/`frstor` pair, which does not support SSE.

- **Supporting two ways of setting the FS and GS registers**
  All x86-64 processors support setting the special FS and GS segmentation registers through MSRs from kernel code. Newer processors also implement the `wrfsbase` and `wrgsbase` instructions to let an application set these registers

from user mode. The HermitCore C code prefers the newer instructions over the older MSRs, however this has no advantages for a single-address-space operating system. Therefore, the Rust code always uses the MSRs.

Both the C and Rust version of the processor configuration code also adjust EIST power management settings. If support for these settings is detected, the processor is set to operate at maximum performance and a constant frequency. This guarantees a predictable runtime behavior for HPC applications and allows the use of the Time Stamp Counter (TSC) as a reliable clock source.

3.5.2 Global Descriptor Table

The next hardware initialization step is the setup of the Global Descriptor Table (GDT). The GDT contains settings for dividing the Virtual Memory address space between code and data as well as kernel and application. A basic GDT has already been set up by the boot assembly code just to enter the 64-bit mode of the processor. However, this table is recreated during the initialization of HermitCore with all needed settings for normal operation. This includes a single entry for kernel code and another one for kernel data. General-purpose operating systems usually create additional entries for application code and application data, but this is not necessary for a single-address-space operating system like HermitCore.

The GDT also contains Task State Segments (TSSs). HermitCore makes use of Task State Segments to define separate stacks for tasks and interrupt/exception handlers. An x86-64 processor automatically sets the internal stack pointer to such a separate stack when an interrupt or exception occurs and the original task stack pointer is restored when the interrupt/exception handler has finished. As a result, a handler cannot corrupt the stack of the current task and vice-versa.

While external interrupts shall only be handled by the boot processor, processor exceptions can occur on every processor. Therefore, the stacks need to be allocated per processor. The C version of the TSS setup code always allocates 3 stacks of 8 KiB for 256 processors statically. For the Rust implementation, the Memory Manager has already been initialized at this stage, so it is possible to allocate the stacks dynamically and only for each processor that is actually installed. This also removes one occurrence of a strict limitation to a maximum number of processors.

The GDT setup code has again employed functions of the *x86* crate to comfortably create the table. However, it originally lacked support for adding TSS entries on x86-64 processors. Therefore, this feature has been implemented into the *x86* crate within this thesis and submitted upstream.

3.5.3 Interrupts and Exceptions

After initializing the GDT and a TSS, hardware initialization continues with the setup of interrupt and exception handlers. An x86-64 processor manages them in

the Interrupt Descriptor Table (IDT). Setting up the IDT in Rust could again be done by means of the *x86* crate.

However, another topic is the implementation of the actual handler functions: Most C compilers have no specific support for writing x86-64 interrupt/exception handlers. Therefore, the original HermitCore implementation splits up the handler functions into multiple parts:

- A prolog written in assembly and duplicated for each interrupt and exception, which saves all registers and calls a common C routine with the number of the interrupt/exception
- A common C routine, which looks up the specific handler routine in a function pointer table (for interrupts) or prints the according exception message (for exceptions)
- An epilog written in assembly that restores the saved registers and returns with `iret`

This design has been revised for the Rust implementation of interrupt and exception handlers. As described in Section 2.2.8, Rust allows functions to be declared with `extern "x86-interrupt"` to mark them as interrupt or exception handler functions. These functions automatically save and restore used registers and return with `iret`. Consequently, assembly prologs and epilogs are no longer necessary, neither is an additional function pointer table. As a result, the code for an interrupt or exception handler is not just more comfortable to write in Rust, but also carries less overhead.

External interrupts in HermitCore are currently limited to timer interrupts, and network adapter interrupts for the Multi-Kernel mode. They are meant to be handled by the Advanced Programmable Interrupt Controller (APIC) as soon as it has been initialized (see Section 3.5.5). However, Section 2.1.1 describes that HermitCore may use the legacy Programmable Interrupt Controllers (PICs) and their connected Programmable Interval Timer (PIT) to determine the processor frequency. As the Rust implementation detects the processor frequency in the next initialization step, the PICs need to be prepared at this stage.

Due to a design mistake in the very first IBM Personal Computer from 1981, PIC 1 signals interrupt numbers 8 to 15 by default. However, any x86-64 or former x86 processor reserves the range from 0 to 31 for processor exceptions. When any of these interrupts occur, the operating system cannot reliably detect whether it is an external interrupt or a processor exception. Therefore, every modern operating system for a x86-64 processor remaps PIC 1 to report interrupts in the range 32 to 39, and PIC 2 to the range 40 to 47. This remapping code is common and well-documented, for example in [40]. For the HermitCore Rust code, it has been implemented similarly, but additionally masks all external interrupts by default. This prevents unexpected interrupts, for example by keystrokes. However, implementing an interrupt handler routine now also requires unmasking the respective interrupt beforehand.

After completing this initialization, the operating system enables interrupts and exceptions for the first time.

3.5.4 Processor Frequency Detection

In the next step, the Rust code detects and reports the frequency of the boot processor. Disabling power management in Section 3.5.1 ensures that this frequency is constant and does not vary depending on the load. Knowing about the processor frequency is important in order to use the TSC for the calculation of 100 Hz ticks. It is also used to calibrate the APIC Timer, as well as for timing calculations in the HermitCore port of *newlib*.

The original HermitCore C code detects the frequency using 3 methods. For the Rust version, another reliable method has been found, so it tries out the following detection methods in this order:

1. **The command-line parameter `-freq`**
   The `-freq` parameter allows specifying the processor frequency in MHz as a boot loader command-line parameter. If this parameter is found, the specified frequency is used without consulting the processor at all.

2. **The CPUID Frequency Information** (added for the Rust implementation)
   An appropriate `cpuid` call on newer processors returns multiple internal frequencies. If this feature is available, the detection code uses the *Base Frequency* as the processor frequency, which is the constant frequency when power management is disabled. However, testing has revealed that this feature is only available in Intel processors starting with the Skylake microarchitecture from 2015 [41]. Therefore, a different solution is required for older processors.

3. **The CPUID Brand String**
   The `cpuid` instruction on all x86-64 processors can also return a so-called *Brand String*. For x86-64 Intel processors, this string is guaranteed to contain information about the nominal frequency and parsing this information is described in [39]. The Rust implementation follows this guideline, but in contrast to it, only frequencies in the GHz range are considered, because no x86-64 processors in the MHz or THz range are currently known.

4. **Measurement**
   Finally, if all previous methods failed, the processor frequency is measured. This happens by configuring the PIT to interrupt with a timer frequency of $f_{\text{PIT}} = 100\,\text{Hz}$ and counting each interrupt handler call (*tick*). When starting the measurement, the current TSC value and PIT tick count is read. After $n_{\text{ticks}} = 3$ PIT ticks have elapsed, the TSC is read again and the difference is calculated. This difference n_{cycles} specifies the number of instructions executed within n_{ticks} PIT ticks. Hence, the processor frequency is calculated by

$$f_{\text{processor}} = \frac{f_{\text{PIT}} \cdot n_{\text{cycles}}}{n_{\text{ticks}}}$$

This method is implemented similarly in the C and Rust version of HermitCore. However, the Rust version should have a slightly better accuracy due to the lower overhead of the interrupt handler. This method is also the sole reason why the processor frequency detection may only happen after interrupts and the PICs have been configured.

All 4 methods make use of Rust's comfortable `Result` type. A successful frequency detection is reported by returning `Ok` whereas `Err` is returned otherwise. In contrast to simple boolean return values, this type triggers a warning when a caller does not check the return value. It also provides the comfortable `or_else` method to concatenate the calls to all 4 methods without writing a series of `if` clauses.

3.5.5 APIC and SMP

In the final hardware initialization step, the Local Advanced Programmable Interrupt Controller (APIC) of the boot processor is configured and the other processors (called *application processors*) are booted for Symmetric Multiprocessing (SMP) operation. This is considered one step, because both tasks are closely related: Their configuration information is derived from the same tables and application processors are initialized by sending Inter-Processor Interrupts (IPIs) using APIC functions.

Configuring the boot processor's Local APIC begins by reading these tables and storing information about the Local APIC IDs of all installed processors as well as the address of the memory-mapped Local APIC of every processor. The Local APIC ID is a unique number inside a computer system to address the Local APIC of a specific processor and therefore the processor itself. Due to the reasons outlined in Section 2.1.1, the HermitCore C implementation relies on the MultiProcessor Tables provided by a MultiProcessor Specification 1.4-compliant computer system to gather this information. This also applies to the Rust implementation. However, the Rust implementation encapsulates the entire parsing code for the MultiProcessor Tables in a single function. This design simplifies a later implementation of a parser for the more popular ACPI tables, should one become necessary in the future.

The configuration continues by determining the APIC operating mode. All x86-64 processors support the xAPIC mode, which exposes APIC registers on a Physical Memory address that needs to be mapped to a Virtual Memory address. Newer processors further support the x2APIC, which can address more processors and exposes its registers over more efficient MSRs. If support for x2APIC is detected, this mode is enabled and used in the following. Otherwise, a Virtual Memory page is requested and the address of the memory-mapped Local APIC is assigned to it.

A notable difference between the C and Rust implementation of the APIC code lies in the function that writes to a Local APIC register. The Rust version of that function can be universally used for xAPIC and x2APIC mode. It always takes

an x2APIC MSR address and a 64-bit value, which it can simply forward to the `wrmsr` instruction in x2APIC operating mode. If the Local APIC runs in xAPIC mode instead, the function automatically translates the x2APIC MSR address to an xAPIC memory address and adapts the value to the required format for xAPIC operation. The underlying code is presented in Appendix A.1. On the contrary, the C implementation often needs different code paths for x2APIC and xAPIC mode, making the code more complex and harder to maintain.

Configuring the Local APIC is concluded by registering the required interrupt handlers and disabling unused ones. As soon as the interrupt number for the spurious interrupt is set, the Local APIC is enabled and starts delivering interrupts to the processor.

After enabling interrupts, the APIC code configures the APIC Timer. HermitCore later uses this timer in one-shot operation to schedule the next deadline. In this operating mode, the counter of the APIC Timer is set to a start value and this value is counted down at a constant frequency until it reaches zero. At that point, an interrupt is signaled and the timer is deactivated until the counter is reset.

Unlike the PIT, the frequency of the APIC Timer is not known. However, the processor frequency is known at this stage, so the APIC Timer frequency can be measured by using the processor's TSC. As all operating system events are later scheduled relative to a 100 Hz timer, an appropriate counter start value $n_{\text{counter,start}}$ for a single tick of that timer shall be determined. To improve measurement accuracy, a value for $n_{\text{ticks}} = 3$ ticks is determined and later divided by n_{ticks}. The detailed implementation is as follows:

1. The APIC Timer is started by setting the counter to the maximum 32-bit value $n_{\text{counter,max}} = 2^{32} - 1$.

2. The current value n_{TSC} of the processor's TSC is read and used to calculate

$$n_{\text{TSC,end}} = n_{\text{TSC}} + n_{\text{ticks}} \cdot \frac{f_{\text{processor}}}{100\,\text{Hz}}$$

3. A busy-waiting loop is entered and constantly updates n_{TSC} with the current value of the processor's TSC. The loop is exited as soon as $n_{\text{TSC}} \geq n_{\text{TSC,end}}$.

4. The current value n_{counter} of the APIC Timer counter register is read. Finally, the counter start value is calculated by

$$n_{\text{counter,start}} = \frac{n_{\text{counter,max}} - n_{\text{counter}}}{n_{\text{ticks}}}$$

   HermitCore can now configure the APIC Timer to signal an interrupt in 100 Hz tick granularity by initializing the counter register with an integer multiple of $n_{\text{counter,start}}$.

It should be noted that the APIC Timer internally divides its counter frequency by a constant value. The HermitCore C version configures this divisor to 1 to get maximum accuracy from the timer. However, this results in a large $n_{\text{counter,start}}$, and multiples of it easily overflow the 32-bit counter register. Consequently, the maximum duration of an APIC Timer one-shot timeout is highly constrained in the C implementation. Testing has shown that the maximum supported divisor of 128 still provides enough accuracy for a single tick of a 100 Hz timer while supporting 128x longer timeouts. Therefore, the Rust implementation configures the divisor to 128 before measuring the APIC Timer frequency. The equations above are independent of the divisor as long as the divisor is set in advance and remains constant afterwards.

Finally, all prerequisites are met to boot application processors and enable SMP operation. A possible algorithm for the x86-64 architecture is standardized and described in [15]. However, this algorithm also considers 32-bit Intel486 and Intel Pentium processors, so it could be implemented in a simplified manner for the Rust version of HermitCore.

For each processor, an *INIT* IPI needs to be sent first. This IPI resets the processor to an initial state. After this IPI, Intel486 and Pentium processors already begin executing the Basic Input/Output System (BIOS) boot code. Consequently, the so-called *BIOS Reset Vector* needs to be programmed in advance to instruct the BIOS to jump to a custom SMP boot code. However, this behavior of the *INIT* IPI does not apply to later 32-bit processors and all 64-bit x86-64 processors. They are reset to a *halted* state and only begin executing code on the first *STARTUP* IPI. Furthermore, the address of the SMP boot code is sent directly with the *STARTUP* IPI. Therefore, the Rust code does not need to reprogram the BIOS Reset Vector. Moreover, it does not send a second *STARTUP* IPI either as stated in the standardized algorithm. Testing has shown that this is no longer necessary and the SMP boot code is already executed after the first *STARTUP* IPI. This is also confirmed by [42].

The HermitCore SMP boot code switches an application processor from 16-bit to 32-bit to 64-bit mode and then calls into the shared boot code for all processors. For the HermitCore Rust implementation, the shared boot code detects that it boots an application processor, skips some initialization steps that have already been performed by the boot processor, and finally calls a Rust entry point specific to application processors. Rust code now performs the remaining per-processor initialization steps. As a final step, the shared processor counter is incremented to let the boot processor know that the current application processor has finished booting. The boot processor then initializes the next application processor.

All installed processors are assumed to be equal models of equal frequency. Therefore, steps like processor frequency detection and APIC Timer measurement are only done once for the boot processor and then the results are applied to all application processors.

3.5.6 Boot Process Diagram

The diagram in Figure 3.4 summarizes the individual steps of the boot process and hardware initialization of the HermitCore Rust implementation.

3.6 Per-Processor Variables

HermitCore supports variables, which hold a different value on each processor. This is useful when a processor needs to store information that is only relevant for itself. Examples for such per-processor information are the own Local APIC ID, the internal tick counter, and a reference to the processor's task scheduler. Both the C and Rust versions of HermitCore support per-processor variables, however their implementations are fundamentally different.

The C version introduces a `.percore` section in the HermitCore image file. Every per-processor variable is declared to be part of the `.percore` section. During the build process, the linker script of the HermitCore toolchain instructs the linker to enlarge the `.percore` section to 256x its size (as determined by the space taken by all declared per-processor variables). As a consequence, memory for up to 256 processors to store different variable values is statically allocated when booting HermitCore. The kernel first initializes all per-processor values to the values of the boot processor and zeroes the boot processor's GS register. Each application processor sets its GS register to the offset of the `.percore` section reserved for it. The macros `per_core` and `set_per_core` then allow access to the per-processor value of a variable. They make use of the segmentation feature of x86-64 processors, which enables the FS and GS registers to be used in a `mov` instruction to add an offset to the given memory address. For example, the instruction `movl %gs:(core_id), eax` determines the memory address of the variable `core_id`, adds the offset stored in the GS register, and loads the 32-bit value stored at the resulting address into the EAX register.

In the beginning, this concept has been implemented similarly in Rust. However, it has since been rewritten entirely to make use of dynamic allocations and remove the limitation to 256 processors. The new implementation requires all per-processor variables to be fields of a global `PERCORE` structure. This structure is statically allocated once for the boot processor, and dynamically allocated every time a new application processor is booted. The GS register of each processor then stores the memory address to its `PERCORE` structure.

Each field of `PERCORE` is declared using the generic type `PerCoreVariable` with the actual variable type as a type parameter. The declaration and fields of `PERCORE` are given in Listing 3.1.

`PerCoreVariable` implements `get` and `set` functions that use a `mov` instruction with segmentation similar to the C version. As different `mov` instructions are needed based on the size of the variable type, Rust's specialization feature is used again. By default, the `get` and `set` implementations use the `movq` instruction to handle

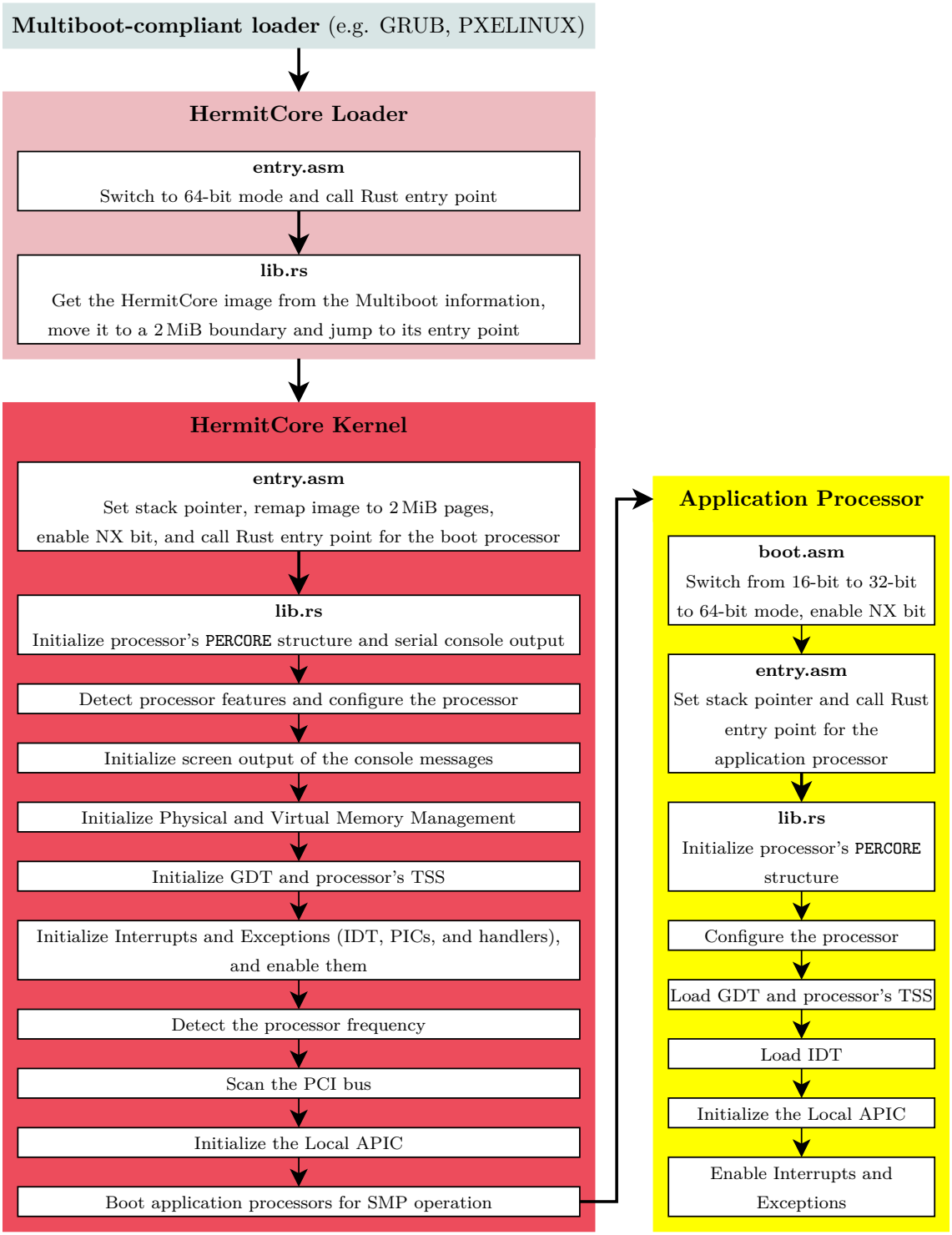


Figure 3.4: Boot process of the HermitCore Rust version including all hardware initialization steps

64-bit values. This covers 64-bit integer types and all pointers. 32-bit values such as the Local APIC ID are handled with a specialized implementation of `get` and `set`, which uses the `movl` instruction instead.

Due to the dynamic allocations, this implementation of per-processor variable support is not limited to a maximum number of processors. It does not need any toolchain support in the form of a `.percore` section either. Finally, the new version also ensures that only the value for the current processor can be accessed. The C version does not prevent accessing a per-processor variable directly by its name instead of using the `per_core` and `set_per_core` macros. This would always yield or modify the variable value stored for the boot processor.

It should be noted that either implementation highly depends on characteristics of the x86-64 architecture. Implementing support for a new architecture in HermitCore requires an architecture-specific implementation of per-processor variables as well.

```
pub static mut PERCORE: PerCoreVariables
    = PerCoreVariables::new(0);

pub struct PerCoreVariables {
    /// APIC ID of this CPU Core.
    core_id: PerCoreVariable<u32>,
    /// Scheduler for this CPU Core.
    scheduler: PerCoreVariable<*mut PerCoreScheduler>,
    /// Task State Segment (TSS) allocated for this CPU Core.
    pub tss: PerCoreVariable<*mut TaskStateSegment>,
    /// Value returned by RDTSC/RDTSCP last time the timer
    /// ticks were updated in processor::update_timer_ticks.
    pub last_rdtsc: PerCoreVariable<u64>,
    /// Counted ticks of a timer with the constant frequency
    /// specified in processor::TIMER_FREQUENCY.
    pub timer_ticks: PerCoreVariable<usize>,
}
```

Listing 3.1: Declaration and fields of the `PERCORE` structure

3.7 Scheduler

The design of the task scheduler of the HermitCore C version has been presented in Section 2.1.3. Instead of writing a new scheduler in Rust based on that C code, the existing scheduler from the eduOS-rs Rust operating system project could be imported. The eduOS-rs project[1] is currently being developed in parallel at the ACS and provides a basic educational operating system written in Rust.

The eduOS-rs scheduler is a priority-based round-robin scheduler centered around a `Scheduler` structure. During boot, a single instance of that structure is initialized along with a `Task` structure for the idle task. When the operating system has finished initialization, it enters an infinite loop that calls into the `schedule` function. All tasks ready to schedule are collected in the *ready queue* and sorted by task priority. The `schedule` function checks if a task with a higher priority than the current task is available, and then calls the `switch` assembly function to switch to that task. The assembly function saves the context of the current task, changes the stack pointers in the TSS, and finally restores the context of the new task. This context includes all registers and flags, including the instruction pointer. Therefore, the new task continues exactly at the position where it has last stopped before switching to another task. If a new task is running for the first time, the scheduler has manually created an initial context for it, with the instruction pointer referencing the task's entry point. Consequently, the `schedule` function can also be used for spawning new tasks. In case no task is available, the scheduler switches back to the idle task, which continues in the infinite loop.

This scheduler generally implements the required functionality for HermitCore. However, eduOS-rs is currently a single-processor operating system. Therefore, the imported scheduler code has been heavily adapted for SMP operation. Based on the general `Scheduler` structure of eduOS-rs, a `PerCoreScheduler` structure has been introduced, which is allocated for each processor during boot and only manages the tasks for that single processor. Each processor has a quick access to its `PerCoreScheduler` instance through a per-processor pointer variable (cf. Section 3.6). Additionally, all `PerCoreScheduler` instances are collected in a `BTreeMap` and sorted by the Local APIC ID corresponding to a processor. This enables one processor to schedule tasks on another processor, for example when spawning a new task or cloning an existing one.

Unlike eduOS-rs, the semaphore synchronization primitive implemented for HermitCore supports acquiring a semaphore with a timeout. Consequently, when a task cannot acquire the semaphore directly, it is blocked until either another task releases the semaphore or the timeout expires. To prevent both events from possibly occurring at the same time, they have been centrally serialized in the `blocked_tasks` queue of `PerCoreScheduler`. Blocking and unblocking a task can only happen through functions of the `blocked_tasks` queue. Unblocking a task is implemented by looking it up and removing it from the `blocked_tasks` queue before adding

[1] `https://rwth-os.github.io/eduOS-rs`

it back to the ready queue. This prevents unblocking a task twice by concurrent events.

When adapting the eduOS-rs Rust scheduler for SMP operation, a clear distinction needs to be made between variables accessed concurrently and requiring synchronization, and variables accessed only by a single processor. In the end, only 3 variables needed to be protected by interrupt-safe spinlocks:

- **The `state` field of `PerCoreScheduler`**
  This field contains the queue of ready tasks waiting to be scheduled and a boolean value indicating whether the processor is halted. It is important that both fields are guarded by a single lock. The reason for this is explained in the following.

- **The `blocked_tasks` field of `PerCoreScheduler`**
  This field contains all blocked tasks of this processor's scheduler, sorted by their timeouts (if any). This queue must be guarded by a lock, because a task releasing a resource may unblock a task on a different processor. Additionally, tasks may be unblocked by timeouts from the timer interrupt handler.

- **The global `TASKS` variable**
  This `BTreeMap` centrally collects all `Task` structures and their unique IDs in the operating system. Every new task on every processor updates this global variable, so a synchronization is inevitable.

It is important that the used spinlocks disable interrupts. Otherwise, the timer interrupt handler may be called and try to lock one of these variables while they are already locked by other scheduler code. The result would be a deadlock and an interrupt handler that never returns.

During the evaluation of the HermitCore scheduler in Rust, a severe problem has been uncovered: When only the idle task is available, both eduOS-rs and the HermitCore scheduler in C reenable interrupts with `sti` and later use the `hlt` instruction to halt the processor until the next interrupt. This way, the processor does not needlessly loop and can therefore reduce its power consumption. Another processor can wake it up by creating a new task for the halted processor and then sending an IPI. However, a race condition occurs when this IPI is received after the processor has reenabled interrupts but before it has called `hlt`. In that case, the processor is woken up while running and then halted anyway, although a new task is available. This problem can be solved by reenabling interrupts and halting the processor in one atomic operation. Intel guarantees that the exact sequence `sti; hlt` is executed atomically [43]. Therefore, the scheduler has been reworked to disable interrupts for the entire critical path and only reenable them with the `sti; hlt` sequence when switching to the idle task.

To minimize operating system noise from interrupts, a wake-up IPI is only sent when the processor has really been halted. Another processor could figure this out by checking the current task of the possibly halted processor. If that current

task is the idle task, a wake-up IPI should be sent. However, this would require a synchronization primitive around the `current_task` field of `PerCoreScheduler`. As most system calls access the current task, their performance would be hampered due to the required locking. Therefore, a boolean value `is_halted` has been introduced, which is set right before the `sti; hlt` sequence and locked together with the `ready_queue` field. Whenever a processor creates a task on another processor, it already locks the ready queue, so it can now check the `is_halted` value as well.

Finally, unsafe `Shared` pointers in the eduOS-rs scheduler have been replaced by safe reference-counted ones using the `Rc` container. This decision has highly reduced the amount of required `unsafe` blocks in the scheduler.

3.8 Features Not Covered

Several additional features have already been implemented in the Rust version of HermitCore, however detailed explanations of them would exceed the scope of this thesis.

To prepare for Multi-Kernel mode, the Lightweight IP (lwIP) TCP/IP stack has been integrated along with a driver for the Intel E1000 Ethernet adapter. As no adequate network stack written in Rust is currently available, this step required designing a Rust interface around the lwIP C interface. Adapting lwIP to work with Rust also improved the design of HermitCore in general, because synchronization primitives that used to be implemented in included header files are now properly exported as system calls by the operating system. The network components are compiled with every build of HermitCore, but currently remain disabled due to a lack of testing.

Network support also requires enumerating the PCI bus. This is currently accomplished by probing for all possible 32 devices per bus on the first 32 PCI buses. Multi-function adapters and bridges are not considered, however these are rarely required for PCI network adapters. The results of the enumeration are output on the console, with vendor and device IDs resolved to names. For this feature, a copy of the PCI ID database from [44] has been added to the HermitCore source tree and is preprocessed into a series of Rust arrays by a prebuild step.

Additionally, the Rust version of HermitCore provides 4 synchronization primitives: Spinlocks, interrupt-safe spinlocks, semaphores, and recursive mutexes. These have been written from scratch and not imported from an external crate like `spin`, because their implementations need to account for the specifics of HermitCore. For example, semaphores and recursive mutexes call into the scheduler to block a task when they cannot acquire a lock directly. Other than that, the implemented versions follow the publicly documented behaviors for such synchronization primitives, so a more detailed explanation would be of little value. For a general explanation of these synchronization primitives, the reader is referred to [45].

Finally, HermitCore is booted in Unikernel mode by a custom boot loader as shown in Figure 3.4. It initializes console output and basic memory management

in order to move the HermitCore image to the next 2 MiB boundary. After that, it jumps to the entry point of the moved HermitCore kernel, which can then map the entire HermitCore image in 2 MiB pages for a better TLB usage. Within this thesis, the boot loader has also been rewritten in Rust. By making use of the same *bitflags* and *x86* crates as well as the HermitCore paging and serial port modules, the actual boot loader code is kept small and easily maintainable. However, the current Rust boot loader offers no additional features compared to its counterpart written in C, so it is not presented in more detail.

4 Evaluation

In this chapter, the implemented Rust version of HermitCore is evaluated in terms of performance, maintainability, and hardware compatibility.

4.1 Test Systems

While most of the testing during development has been conducted inside QEMU virtual machines, accurate performance measurements require running HermitCore on real hardware. Additionally, actual hardware often differs in details from the specification and the QEMU implementation, so testing on real hardware is inevitable for ensuring compatibility. To guarantee a maximum level of hardware compatibility, a broad variety of systems has been used as listed in Table 4.1.

	System 1	**System 2**	**System 3**
Processors:	2x Quad-Core Intel Xeon L5630, 2.13 GHz, with SMT	2x Dual-Core AMD Opteron 270, 2.00 GHz	Dual-Core Intel Core 2 Duo T7300, 2.00 GHz
Mainboard:	Supermicro X8DTH-6	Rioworks HDAMA	Lenovo ThinkPad X61
Chipset:	Intel 5520	AMD-8111/8131	Intel GM965
RAM:	32 GiB	4 GiB	3 GiB
APIC:	xAPIC	xAPIC	xAPIC
Serial port:	Yes	Yes	Yes
MP 1.4:	Yes	Yes	Yes

	System 4	**System 5**	**System 6**
Processors:	Dual-Core Intel Core i7-640LM, 2.13 GHz, with SMT	2x 10-Core Intel Xeon E5-2650 v3, 2.30 GHz, with SMT	Dual-Core AMD Turion 64 X2 TL-56, 1.60 GHz
Mainboard:	Lenovo ThinkPad X201s	Supermicro X10DAi	Dell Latitude D531
Chipset:	Intel QM57	Intel C612	AMD M690T
RAM:	8 GiB	64 GiB	2 GiB
APIC:	xAPIC	x2APIC	xAPIC
Serial port:	No	No	Yes
MP 1.4:	Yes	Yes	No

Table 4.1: Systems used for testing HermitCore

System 1 is a 2010-era server system featuring 2 processors, 4 cores per processor, and Simultaneous Multithreading (SMT) to expose each core as 2 (also known as *Intel HyperThreading*). This totals to 16 processors that should be seen by the operating system. Additionally, the system features a serial port, so all console messages can be redirected to a terminal that saves a log file. This is required for running the benchmarks and makes System 1 the preferred one for benchmarking.

System 2 features a 2003-era server mainboard with 2005-era processors. With that configuration, it comes close to the very first x86-64 systems that have ever been released. Testing HermitCore on such an early system validates the hypotheses about all x86-64 systems from Section 3.5.1. It is also the only MultiProcessor Specification 1.4-compliant test system based on AMD processors.

System 3 is a 2007-era business laptop with a serial port. Running HermitCore on this system confirms the hypothesis that also modern non-server systems still provide MultiProcessor Specification 1.4-compliant MultiProcessor Tables.

System 4 from 2010 validates the same hypothesis, however its usefulness for testing HermitCore is limited by its lack of a serial port. It can still output console messages on the screen though, so basic debugging is possible.

System 5 is the 2014-era server used for benchmarking HermitCore in [1]. Due to its lack of a serial port, it could not be used for logging benchmark results in the Rust version of HermitCore. However, among the test systems, it is the only one providing x2APIC mode.

Finally, System 6 is another 2007-era business laptop with serial port, but based on an AMD processor. However, testing has revealed that this machine is not MultiProcessor Specification 1.4-compliant.

4.2 Hardware Compatibility

The broad variety of available systems allowed for extensive testing of the hardware support of the HermitCore Rust version.

By featuring multiple processors, multiple cores per processor, and multiple threads per core, as well as a serial port and MultiProcessor Specification 1.4 compliance, System 1 has been the candidate of choice for HermitCore testing and running further benchmarks. Booting HermitCore on this machine has worked successfully from the beginning without any hardware-specific fixes required. However, the operating system can only see and initialize 8 instead of 16 processors. This has turned out to be a limitation of the MultiProcessor Specification 1.4: Processors making use of SMT are only exposed as a single processor in the MultiProcessor Tables [46]. Detecting all 16 processors would require implementing a parser for ACPI tables, which exceeds the scope of this thesis.

When testing the AMD-based System 2, an early revision of the HermitCore Rust implementation failed during the boot process. The reason was a spurious PIC interrupt right after enabling interrupts for the first time. The same problem could be reproduced with the other AMD-based System 6. While the root cause

of the interrupt could not be determined, implementing a graceful handling of spurious PIC interrupts fixed the problem. This allowed HermitCore to continue booting and no further spurious interrupts occurred, which could have hampered the predictable runtimes of the operating system. Apart from this early problem, HermitCore works flawlessly on such an old x86-64 server system. Compared to the first x86-64 processor, the installed AMD Opteron processor only supports SSE3 in addition. As long as SSE3 is not used inside HermitCore code, this test validates the hypotheses from Section 3.5.1. In particular, it proves that the operating system does not depend on any modern processor features and is universally usable on any x86-64 platform.

System 3 and 4 have been tested after the previous fixes and exposed no additional problems. Like System 1, the MultiProcessor Tables of the SMT-enabled System 4 also lack information about the logical processors.

As System 5 has been used for running the benchmarks in [1], it would have been a preferred candidate for benchmarking the HermitCore Rust version. However, its lack of a serial port only allowed examining a few console messages on the screen instead of logging all of them to a text file. Therefore, System 1 has been used for benchmarking instead. Nevertheless, running the HermitCore Rust version on System 5 has triggered a bug in the x2APIC implementation, which has been fixed in the meantime.

Due to the lack of MultiProcessor Tables, System 6 failed to boot HermitCore at the Local APIC initialization stage. Anyway, this test proved that the MultiProcessor Specification 1.4 detection code works properly and fails with a descriptive error message when no tables are detected.

4.3 Benchmarks

The performance of the C version of the HermitCore operating system has been evaluated with several micro-benchmarks in [1]. These benchmarks are available as regular HermitCore applications in the HermitCore source tree, so it has been straightforward to run them for the Rust version as well.

Due to the aforementioned reasons, System 1 has been used to run these benchmarks. To make the results comparable, this required rerunning the C version of HermitCore in Unikernel mode on the test system.

A bug in the current C version only allows it to boot on computers with consecutive Local APIC IDs. However, System 1 does not address its processors with consecutive IDs. Therefore, the Local APIC IDs for System 1 have been determined and entered into a fixed array in the C version. The `for` loop for initializing application processors has then been modified to take the IDs from this array. This allowed fixing the C implementation for benchmarking System 1 without changing a lot of code.

Furthermore, extra care has been taken that network components are disabled in both versions in order to not incur any noise from a TCP/IP background task.

4.3.1 Basic Micro-Benchmarks

Table 4.2 shows the results of some benchmarks run on the Rust and C versions of HermitCore in Unikernel mode. Each benchmark warms up the caches and then measures the average latency of a specific system operation.

Most of these benchmarks are run by the `basic` application shipped with the HermitCore source code. Additionally, a `taskswitch` benchmark has been developed within this thesis to measure the latency of a switch between two tasks running on the same processor. Furthermore, a single test of the `basic` application had to be adapted to consider the 2 MiB pages used in the HermitCore Rust version.

Within this thesis, these benchmarks have not been rerun on Linux, so the Linux testing results in Table 4.2 are taken from [1].

System operation	HermitCore Rust	HermitCore C	Linux*
`getpid()`	17	17	143
`sched_yield()`	218	100	370
`malloc()`	764	6080	6575
first write access to a page	27 (4 KiB), 925 (2 MiB)	1407	4007
task switch	5170	934	

Table 4.2: Results of the basic micro-benchmarks on System 1 under HermitCore and System 5 under Linux (in processor cycles)

The `getpid()` call is considered the shortest system call, so it serves as a measure for the general overhead of a system call. Both the Rust and the C version of HermitCore are on par in this benchmark. Even though the Linux benchmark has been run on the newer and better performing System 5, the overhead of a system call is more than 8 times higher on it. This can be explained due to the required context switch in Linux and other general-purpose operating systems.

The `sched_yield()` call needs more than twice the processor cycles in the Rust version compared to the C version. This happens, because the Rust version only implements a single `scheduler` function for calling into the scheduler to check for new or higher prioritized tasks and switch to them. This function performs cleanup operations, locks the scheduler, and handles all possible cases. On the other hand, the C implementation of HermitCore implements `scheduler` and the lightweight `check_scheduling` function. The latter one is called during the `sched_yield()` test and finishes quicker when no new task is available. However, it does not perform any locking, so it may be prone to race conditions. Additionally, the maintenance of the Rust scheduler should be easier, because only a single function needs to be maintained. In any case, both implementations still incur less overhead than the Linux benchmark on System 5.

The two memory micro-benchmarks are drastically faster in the HermitCore Rust version. This is the result of using 2 MiB pages and a single large Virtual Memory

region for application memory. In contrast to the C implementation, no lists need to be modified when an application resizes its heap through the `sbrk` system call.

By default, the *first write access to a page* benchmark always assumes 4 KiB pages. This makes it unsuitable for measuring the first write access to 2 MiB pages and hence the required cycles are much lower in the HermitCore Rust implementation. However, these results are still viable if they are reinterpreted as *first write access to 4 KiB distant data*.

For the sake of completeness, this micro-benchmark has also been adapted for 2 MiB pages and run on the Rust version of HermitCore. As expected, the number of processor cycles is much higher compared to the test assuming 4 KiB pages. However, the Rust version still outperforms the C version in this regard. This proves a positive performance impact of the simpler application memory management in combination with the recursive Paging algorithm presented in Section 3.4.1.

Finally, the *task switch* benchmark again shows the better performance of the C scheduler compared to the Rust version. The current Rust scheduler needs more than 5x as many cycles as the C implementation for switching between tasks. The additional cycles probably stem from the borrow-checking and reference-counting happening at runtime in the Rust implementation.

4.3.2 Hourglass Benchmark

The *Hourglass* benchmark has been presented in [47] to measure the operating system noise. It continuously reads the processor's TSC in a loop and stores the difference to the last TSC read. In a completely noiseless system with only the benchmark running, this difference should remain constant. However, if larger gaps occur, a background process is stealing processor time from the application.

	HermitCore Rust	**HermitCore C**	**Linux***
Minimum	24	24	40
Average	30.14	30.15	69.46
Maximum	2551744	5372052	51840

Table 4.3: Results of the Hourglass benchmark on System 1 under HermitCore and System 5 under Linux (in processor cycles)

The results of the Hourglass benchmark are shown in Table 4.3, with the Linux results again taken from [1]. They prove that the Rust implementation of HermitCore created within this thesis is no more noisier than the existing C implementation. In fact, the average number of required processor cycles over almost 283 million iterations is close to the minimum number and several magnitudes smaller than the maximum. Linux is more than two times as noisy as HermitCore on average, even though the tested processors have been isolated with the *isolcpu* option and periodic ticks have been disabled with *nohz*.

However, there are no simple explanations for the high maximum values of System 1. One reason may be the System Management Mode (SMM) triggered periodically on an x86-64 system. It is responsible for handling hardware monitoring, power management, and vendor-specific functions in the background. Disabling these periodic interrupts requires a customized BIOS, because the relevant configuration registers are already locked when the operating system is booted [48]. This theory is supported by the lower maximum values of the Linux results of System 5. It is possible that this server has periodic SMM interrupts disabled or at least a better performing implementation.

4.4 Memory and Storage Usage

The HermitCore C version reserves a kernel stack of 8 KiB for each processor and this stack size is sufficient. In contrast, the compiled Rust code uses more stack memory and requires a much larger kernel stack to successfully boot up and launch the HermitCore application. Currently, a per-processor stack of 32 KiB is allocated for the kernel written in Rust. As this is no problem for an HPC operating system, the reasons behind the higher stack usage have not been researched further. However, this has also been observed independently during the development of another Rust operating system in [49].

Due to the lack of the Buddy System, the current kernel heap memory usage is also expected to be higher in the Rust version compared to the C version. However, this has neither been a problem on the tested computer systems. If this becomes a concern in the future, it can be easily improved by the introduction of the Buddy System now that the Rust memory management components have stabilized.

The HermitCore C images of benchmark applications are 7 MiB in size on average. Compared to that, the HermitCore Rust images for the same applications are only around 3 MiB in size. Furthermore, around 800 KiB of that is taken by the PCI ID database, which is not compiled into HermitCore C images. One reason for this huge difference are the additional features of the C version, such as the network stack (whose lwIP library can already consume up to 500 KiB in compiled code). On the other hand, the Rust version relies on less statically allocated buffers. A notable preallocated buffer of the C version is the 8 KiB kernel stack for 256 processors, which sums up to 2 MiB in image size.

4.5 Code Maintainability

In order to get a metric for the required maintenance efforts of both HermitCore versions, their Number of Files and Lines of Code have been compared using the `cloc`[1] tool.

[1] `http://cloc.sourceforge.net`

For a fair comparison, all applications, external libraries, and tools have been removed from both source trees in advance. This also applies to all implemented network drivers. Otherwise, the comparison would be highly in favor of the HermitCore Rust version, because the C version currently implements more such drivers. In the end, only source files written as part of the HermitCore project and required for Unikernel mode are left.

Depending on the preferred style, a programmer may write curly braces at the end of a line or always on a separate line. Because the first style is more dominant in the Rust version while the latter one has often been used in the C implementation, the `cloc` language definition file has been modified to exclude lines with only curly braces from counting.

The results are shown in Table 4.4.

	HermitCore Rust		**HermitCore C**	
	Files	Lines of Code	Files	Lines of Code
Rust	57	4157	0	0
C Source	8	667	37	5781
C Header	22	866	70	4987
Assembly	6	579	4	932
Sum	**93**	**6269**	**111**	**11700**

Table 4.4: Number of Files and Lines of Code of the HermitCore Rust and C versions

The total number of code lines of the current HermitCore Rust version is almost half the number of the HermitCore C version. However, it must be taken into account that the compared parts of the C version still implement features like Multi-Kernel mode, Buddy System, and signals, which are not yet available in the Rust version. Anyway, these numbers prove the hypothesis that the Rust rewrite resulted in a smaller codebase. The advantages are an easier maintenance and less possible bugs.

One of the reasons is that C code often requires the same function to be defined in a source and a header file. In addition to that, the programmer also has to consider the order of declarations in a header file. On the other hand, each function only needs to be defined and implemented once at an arbitrary location in a Rust source file. Additionally, the implicit deallocation of memory and release of synchronization objects when a variable goes out of scope reduces the number of required code lines to implement the same logic in Rust. At the same time, this feature protects against memory leaks and deadlocks. Finally, Rust's design guards against common programming mistakes as outlined in Section 2.2. All these aspects generally result in less code and a possibly increased productivity when developing in Rust.

Apart from this, the lines of assembly code could be highly reduced in the HermitCore Rust version due to a simpler implementation of task switching and interrupt handlers. However, a notable number of C header files remain in the Rust tree. These are still required to make HermitCore interfaces available to C applications.

It should be noted that these numbers are only suitable to roughly compare the custom parts of both HermitCore codebases. From a security standpoint, all external libraries, compiler-supplied functions, as well as the compiler itself also need to be verified. This is expected to increase the total number of code lines significantly.

Finally, a fair comparison of the maintainability of both HermitCore versions also has to consider the state of the underlying programming languages.

C has been the dominant language for writing operating system code over the last two decades. It is internationally standardized since 1990 and new versions of the standard emphasize backward compatibility to the original. Furthermore, the standard is supported by many independent compiler platforms such as GCC, LLVM, and Visual Studio. This guarantees a high quality and maturity of both the standard's documents and the widespread compilers. Additionally, every operating system developer can be expected to know about the C language, so finding a developer to extend an existing C operating system is relatively easy.

On the other hand, Rust is a new programming language that is constantly evolving. While substantial changes require documentation and undergo a review process[2], there is no single document describing Rust detailed enough for implementing an independent compiler. Therefore, all Rust applications currently depend on the single Rust compiler and the community lead by Mozilla Research. Furthermore, new versions of Rust only promise backward compatibility to stable features of the Rust toolchain, but Rust operating systems usually require features of nightly toolchain builds. Interfaces to these features are unstable and subject to change. While Rust is currently introducing the concept of *editions* to let a project stick to a specific language level, this also implies that using the latest features may require syntactical code changes in the future [50]. Consequently, at the time of writing, an operating system written in Rust is expected to require more maintenance work when moving to a new compiler version compared to an operating system in C. Additionally, Rust developers are currently scarce, so finding a developer to extend a Rust operating system is considerably harder.

4.6 The Rust Toolchain

Version `1.22.0-nightly (17f56c549 2017-09-21)` of the Rust toolchain that has been used within this thesis has proven itself to be stable and reliable. During the entire development, no Internal Compiler Errors (ICEs) have occurred and no incorrectly compiled code has been determined. This applies to both unoptimized debug builds as well as highly optimized release builds. It is also a consequence of basing the Rust toolchain on the LLVM compiler framework, which is mature enough to serve as the code generator for many popular operating systems (such as Apple macOS or FreeBSD).

[2] `https://github.com/rust-lang/rfcs`

The integration into the popular GNU Debugger (GDB) also makes source-level debugging of Rust code available to a variety of existing development environments. However, in the beginning, GDB itself was incapable of debugging an x86-64 QEMU virtual machine. For several years, this required a patch [18] and therefore a patched GDB had to be used within this thesis. By the end of the thesis work, this problem has finally been fixed officially[3].

Nevertheless, the high dependence on features only available in nightly builds of the compiler currently complicates operating system development in Rust. An update to compiler version `1.26.0-nightly` already requires some customizations to the HermitCore Rust code to make it compile again.

[3] `https://sourceware.org/git/gitweb.cgi?p=binutils-gdb.git;a=commit;h=5cd63fda035d4ba949e6478406162c4673b3c9ef`

5 Conclusion

The topic of this thesis was the evaluation of the Rust programming language for operating system development and porting key components of the HermitCore Unikernel. Within the 6 months of thesis work, it has been possible to port the entire HermitCore operating system in Unikernel mode to Rust and document the implementation. A new Memory Manager has been written, which leverages a generic Free List structure and uniformly supports all page sizes of the x86-64 architecture. It has been stress-tested to ensure stability in all memory allocation cases. Furthermore, the new hardware initialization code removes redundant checks and improves several algorithms, like booting application processors or handling multiple APIC modes. Implementing interrupt and exception handlers in Rust reduces their overhead and also the amount of required assembly code. Finally, the task scheduler written in Rust fixes concurrency bugs of the original, features a cleaner design and requires less code for the same functionality.

All in all, the Rust language has proven to be a viable language for operating system development and the resulting code is shorter, faster in some benchmarks, and easier to maintain (cf. Chapter 4). The Rust codebase is also expected to be less prone to bugs due to Rust's design advantages outlined in Section 2.2.

In a future work, the Rust implementation of HermitCore should be extended by the features currently missing compared to the original. This includes support for the Multi-Kernel mode, Buddy System, signals, as well as an integration with the lightweight *uhyve* hypervisor developed at the ACS. The same applies to the ARM AArch64 port currently being developed.

Additionally, HermitCore could highly benefit from a basic parser for the ACPI tables. This would enable it to use all logical processors of an SMT processor and also make it compatible with computer systems lacking MultiProcessor Specification 1.4 compliance.

Regarding the code, the HermitCore Rust version has to use many `unsafe` blocks for global variables. These variables are only set once or only accessed by the single kernel thread on the boot processor. Such accesses are conceptually safe and should not require an `unsafe` block, however the current Rust compiler cannot detect the safety of these situations. This either needs to be addressed by the Rust compiler developers or it can be solved by enclosing these variables in a dummy synchronization primitive, which asserts that all accesses come from the boot processor. The latter solution may however introduce an overhead that is measurable in the micro-benchmarks of Section 4.3.

In any case, the Rust language needs improved support for classical arrays, which are prevalent in low-level system development. It is currently impossible to write code, which is generic over the size of an array. This limitation also affects the Rust core library, because many of its standard traits are currently only implemented for array sizes from 1 through 32.

Finally, a logical next step for a Rust version of HermitCore is adding support for HermitCore applications written in Rust. This should be implemented without relying on the *newlib* library written in C, as all other HermitCore applications currently do.

Appendix

A Source Code

This appendix presents additional source code that has not been included in the main chapters for the sake of clarity.

A.1 Universal APIC Register Access

The APIC initialization in Section 3.5.5 makes use of read and write functions to access the APIC registers. These functions have been implemented in a universal way to support both xAPIC and x2APIC mode. The corresponding Rust code is presented in the following:

```
/// Translate the x2APIC MSR into an xAPIC memory address.
#[inline]
fn translate_x2apic_msr_to_xapic_address(x2apic_msr: u32)
-> usize {
    unsafe {
        LOCAL_APIC_ADDRESS +
        ((x2apic_msr as usize & 0xFF) << 4)
    }
}

fn local_apic_read(x2apic_msr: u32) -> u32 {
    if processor::supports_x2apic() {
        // x2APIC is simple, we can just read from the
        // given MSR.
        unsafe { rdmsr(x2apic_msr) as u32 }
    } else {
        unsafe { *(
            translate_x2apic_msr_to_xapic_address(x2apic_msr)
            as *const u32
        ) }
    }
}
```

```
fn local_apic_write(x2apic_msr: u32, value: u64) {
    if processor::supports_x2apic() {
        // x2APIC is simple, we can just write the given
        // value to the given MSR.
        unsafe { wrmsr(x2apic_msr, value); }
    } else {
        if x2apic_msr == IA32_X2APIC_ICR {
            // Instead of a single 64-bit ICR register, xAPIC
            // has two 32-bit registers (ICR1 and ICR2).
            // There is a gap between them and the destination
            // field in ICR2 is also 8 bits instead of
            // 32 bits.
            let destination = ((value >> 8) & 0xFF00_0000)
                as u32;
            let icr2 = unsafe { &mut *(
                (LOCAL_APIC_ADDRESS + APIC_ICR2) as *mut u32
            ) };
            *icr2 = destination;

            // The remaining data without the destination will
            // now be written into ICR1.
        }

        // Write the value.
        let value_ref = unsafe { &mut *(
            translate_x2apic_msr_to_xapic_address(x2apic_msr)
            as *mut u32
        ) };
        *value_ref = value as u32;

        if x2apic_msr == IA32_X2APIC_ICR {
            // The ICR1 register in xAPIC mode also has a
            // Delivery Status bit that must be checked.
            // Wait until the CPU clears it.
            // This bit does not exist in x2APIC mode
            // (cf. Intel Vol. 3A, 10.12.9).
            while (unsafe {
                ptr::read_volatile(value_ref)
            } & APIC_ICR_DELIVERY_STATUS_PENDING) > 0 {
                hint_core_should_pause();
            }
        }
    }
}
```

Listing A.1: Universal implementations for reading and writing APIC registers in both APIC modes

B Sample Console Log

Sections 3.2, 3.5.1, and 3.8 have detailed the console output produced by the HermitCore Rust version and emphasized its importance for associating a log to the used computer system. In the following, a sample log from System 1 (cf. Section 4.1) running the `basic` benchmark (cf. Section 4.3.1) with the message level set to `INFO` is shown:

```
[LOADER] Started
[LOADER] Found Multiboot information at 0x2000
[LOADER] Found an ELF module at 0x114000
[LOADER] This is a HermitCore Application
[LOADER] File Size: 1171456 Bytes
[LOADER] Mem Size:  1173049 Bytes
[LOADER] Jumping to HermitCore Application Entry Point at 0x800000
[0][INFO] Welcome to HermitCore 0.2.2 (bdf60d5c28aa0ee453b78a5eefa18ff04cbc1fd6)
[0][INFO]
[0][INFO] ====================== PHYSICAL MEMORY FREE LIST ======================
[0][INFO] 0x00000000A00000 - 0x0000007F780000
[0][INFO] 0x00000100000000 - 0x00000880000000
[0][INFO] ========================================================================
[0][INFO]
[0][INFO]
[0][INFO] ================== KERNEL VIRTUAL MEMORY FREE LIST ===================
[0][INFO] 0x00000000A00000 - 0x00000100000000
[0][INFO] ========================================================================
[0][INFO]
[0][INFO]
[0][INFO] ========================== CPU INFORMATION ===========================
[0][INFO] Model:                   Intel(R) Xeon(R) CPU           L5630  @ 2.13GHz
[0][INFO] Frequency:               2130 MHz (from CPUID Brand String)
[0][INFO] SpeedStep Technology:    Available, enabled with maximum P-State 18 (
    Turbo Mode), disabled Performance/Energy Bias
[0][INFO] Features:                MMX SSE SSE2 SSE3 SSSE3 SSE4.1 SSE4.2 EIST AESNI
     MCE FXSR VMX RDTSCP MWAIT CLFLUSH DCA
[0][INFO] Physical Address Width:  40 bits
[0][INFO] Linear Address Width:    48 bits
[0][INFO] Supports 1GiB Pages:     Yes
[0][INFO] ========================================================================
[0][INFO]
[0][INFO]
[0][INFO] ======================== PCI BUS INFORMATION ========================
[0][INFO] 00:00 Host bridge [0600]: Intel Corporation 5520 I/O Hub to ESI Port
    [8086:3406]
[0][INFO] 00:01 PCI bridge [0604]: Intel Corporation 5520/5500/X58 I/O Hub PCI
    Express Root Port 1 [8086:3408]
[0][INFO] 00:03 PCI bridge [0604]: Intel Corporation 5520/5500/X58 I/O Hub PCI
    Express Root Port 3 [8086:340A]
[0][INFO] 00:05 PCI bridge [0604]: Intel Corporation 5520/X58 I/O Hub PCI Express
    Root Port 5 [8086:340C]
[0][INFO] 00:07 PCI bridge [0604]: Intel Corporation 5520/5500/X58 I/O Hub PCI
    Express Root Port 7 [8086:340E]
[0][INFO] 00:09 PCI bridge [0604]: Intel Corporation 7500/5520/5500/X58 I/O Hub PCI
     Express Root Port 9 [8086:3410]
```

```
[0][INFO] 00:13 PIC [0800]: Intel Corporation 7500/5520/5500/X58 I/O Hub I/OxAPIC
    Interrupt Controller [8086:342D]
[0][INFO] 00:14 PIC [0800]: Intel Corporation 7500/5520/5500/X58 I/O Hub System
    Management Registers [8086:342E]
[0][INFO] 00:16 System peripheral [0880]: Intel Corporation 5520/5500/X58 Chipset
    QuickData Technology Device [8086:3430], IRQ 10
[0][INFO] 00:1A USB controller [0C03]: Intel Corporation 82801JI (ICH10 Family) USB
     UHCI Controller #4 [8086:3A37], IRQ 10
[0][INFO] 00:1D USB controller [0C03]: Intel Corporation 82801JI (ICH10 Family) USB
     UHCI Controller #1 [8086:3A34], IRQ 6
[0][INFO] 00:1E PCI bridge [0604]: Intel Corporation 82801 PCI Bridge [8086:244E]
[0][INFO] 00:1F ISA bridge [0601]: Intel Corporation 82801JIR (ICH10R) LPC
    Interface Controller [8086:3A16]
[0][INFO] 01:00 Ethernet controller [0200]: Intel Corporation 82576 Gigabit Network
     Connection [8086:10C9], IRQ 10
[0][INFO] 03:00 USB controller [0C03]: VIA Technologies, Inc. VL805 USB 3.0 Host
    Controller [1106:3483], IRQ 10
[0][INFO] 04:00 PCI bridge [0604]: PLX Technology, Inc. PEX8112 x1 Lane PCI Express
    -to-PCI Bridge [10B5:8112], IRQ 10
[0][INFO] 06:00 VGA compatible controller [0300]: NVIDIA Corporation G98 [GeForce
    9300 GE] [10DE:06E0], IRQ 10
[0][INFO] 08:04 VGA compatible controller [0300]: Matrox Electronics Systems Ltd.
    MGA G200eW WPCM450 [102B:0532], IRQ 10
[0][INFO] ======================================================================
[0][INFO]
[0][INFO]
[0][INFO] ==================== MULTIPROCESSOR INFORMATION ====================
[0][INFO] APIC in use:             xAPIC
[0][INFO] Initialized CPUs:        8
[0][INFO] ======================================================================
[0][INFO]
[0][INFO] Creating task 8
Determine systems performance
============================
Average time for getpid: 17 cycles, pid 8
Average time for sched_yield: 218 cycles
Average time for malloc: 764 cycles
Average time for the first page access: 27 cycles
[0][INFO] Finishing task 8 with exit code 0
[0][INFO] Cleaning up task 8
[0][INFO] Shutting down system
```

Listing B.1: Console Log of System 1 running the `basic` benchmark

List of Figures

List of Tables

List of Listings

List of Abbreviations

ABI	Application Binary Interface
ACPI	Advanced Configuration and Power Interface
ACS	Institute for Automation of Complex Power Systems
AMD	Advanced Micro Devices
AML	ACPI Machine Language
APIC	Advanced Programmable Interrupt Controller
BIOS	Basic Input/Output System
CISC	Complex Instruction Set Computer
CR	Control Register
DMA	Direct Memory Access
EIST	Enhanced Intel SpeedStep Technology
FFI	Foreign Function Interface
FPU	Floating-Point Unit
GCC	GNU Compiler Collection
GDB	GNU Debugger
GDT	Global Descriptor Table
HPC	High-Performance Computing
ICE	Internal Compiler Error
IDT	Interrupt Descriptor Table
IPI	Inter-Processor Interrupt
IR	Intermediate Representation
LfBS	Lehrstuhl für Betriebssysteme
lwIP	Lightweight IP
MMU	Memory Management Unit
MPI	Message Passing Interface
MSB	Most Significant Bit
MSR	Machine-Specific Register
NUCA	Non-Uniform Cache Architecture
NUMA	Non-Uniform Memory Access
NX	No-eXecute
PCI	Peripheral Component Interconnect
PD	Page Directory
PDPT	Page Directory Pointer Table
PIC	Programmable Interrupt Controller
PIT	Programmable Interval Timer
PML4	Page Map Level 4

POSIX	Portable Operating System Interface
PT	Page Table
PTE	POSIX Threads for Embedded systems
RAII	Resource Acquisition Is Initialization
RAM	Random Access Memory
RTOS	Real-Time Operating System
SIMD	Single Instruction, Multiple Data
SMM	System Management Mode
SMP	Symmetric Multiprocessing
SMT	Simultaneous Multithreading
TLB	Translation Lookaside Buffer
TMP	Template Metaprogramming
TSC	Time Stamp Counter
TSS	Task State Segment

Bibliography

[1] Stefan Lankes, Simon Pickartz, and Jens Breitbart. "A Low Noise Unikernel for Extrem-Scale Systems". In: *30th International Conference on Architecture of Computing Systems – ARCS 2017* (2017), pp. 73–84.

[2] Dan Tsafrir et al. "System Noise, OS Clock Ticks, and Fine-Grained Parallel Applications". In: *ICS '05 Proceedings of the 19th annual international conference on Supercomputing* (2005), pp. 303–312.

[3] Anil Madhavapeddy et al. "Unikernels: Library Operating Systems for the Cloud". In: *ASPLOS '13 Proceedings of the 18th international conference on Architectural support for programming languages and operating systems* (2013), pp. 461–472.

[4] Graydon Hoare. *Project Servo - Technology from the past come to save the future from itself*. 2010. URL: `http://venge.net/graydon/talks/intro-talk-2.pdf`.

[5] Rust project developers. *Frequently Asked Questions - The Rust Programming Language*. 2018. URL: `https://www.rust-lang.org/en-US/faq.html`.

[6] Rust project developers. *Data Races and Race Conditions - The Rustonomicon*. 2018. URL: `https://doc.rust-lang.org/nomicon/races.html`.

[7] Yves Younan. *25 Years of Vulnerabilities: 1988-2012*. 2012. URL: `https://courses.cs.washington.edu/courses/cse484/14au/reading/25-years-vulnerabilities.pdf`.

[8] Abel Avram and Graydon Hoare. *Interview on Rust, a Systems Programming Language Developed by Mozilla*. 2012. URL: `https://www.infoq.com/news/2012/08/Interview-Rust`.

[9] Rust project developers. *Appendix: Influences - The Rust Reference*. 2018. URL: `https://doc.rust-lang.org/reference/influences.html`.

[10] TOP500.org. *Operating system Family / Linux - TOP500 Supercomputer Sites*. 2018. URL: `https://www.top500.org/statistics/details/osfam/1`.

[11] TOP500.org. *Highlights - November 2017 - TOP500 Supercomputer Sites*. 2017. URL: `https://www.top500.org/lists/2017/11/highlights`.

[12] Kevin McGrath and Dave Christie. "The AMD x86-64 Architecture - Extending the x86 to 64 bits". In: *Hot Chips 14* (2002).

[13] P.K. Nizar. "Advanced Programmable Interrupt Controller (APIC) for MP and 32-bit Operating Systems". In: *Hot Chips 04* (1992).

[14] Hans-Peter Messmer and Klaus Dembowski. *PC Hardwarebuch - Aufbau, Funktionsweise, Programmierung*. Addison-Wesley, 2003. ISBN: 9783827320148.

[15] Intel Corporation. *MultiProcessor Specification 1.4*. 1997. URL: https://web.archive.org/web/20121002210153/http://download.intel.com/design/archives/processors/pro/docs/24201606.pdf.

[16] Unified EFI Forum, Inc. *Advanced Configuration and Power Interface (ACPI) Specification - Version 6.2 Errata A*. 2017. URL: http://www.uefi.org/sites/default/files/resources/ACPI%206_2_A_Sept29.pdf.

[17] Intel Corporation. *5-Level Paging and 5-Level EPT White Paper*. 2017. URL: https://software.intel.com/sites/default/files/managed/2b/80/5-level_paging_white_paper.pdf.

[18] Steffen Vogel. "Eine generische Speicherverwaltung mit Hilfe von Seitentabellen für ein minimalistisches Betriebssystem". Bachelor's Thesis. RWTH Aachen University, 2014.

[19] Kenneth C. Knowlton. "A fast storage allocator". In: *Communications of the ACM* 8.10 (1965), pp. 623–624.

[20] Peter Bright. *Better on the inside: under the hood of Windows 8*. 2012. URL: https://arstechnica.com/information-technology/2012/10/better-on-the-inside-under-the-hood-of-windows-8/2/.

[21] Jonathan Corbet. *(Nearly) full tickless operation in 3.10*. 2013. URL: https://lwn.net/Articles/549580/.

[22] TIOBE software BV. *TIOBE Index - February 2018*. 2018. URL: https://www.tiobe.com/tiobe-index/.

[23] Dennis M. Ritchie. "The development of the C language". In: *ACM Sigplan Notices* 28.3 (1993), pp. 201–208.

[24] Bjarne Stroustrup. "A History of C++: 1979-1991". In: *ACM Sigplan Notices* 28.3 (1993), pp. 271–297.

[25] Peter Seibel. *Coders at Work: Reflections on the Craft of Programming*. Apress, 2009. ISBN: 9781430219484.

[26] Graydon Hoare. *[rust-dev] stage1/rustc builds*. 2011. URL: https://mail.mozilla.org/pipermail/rust-dev/2011-April/000330.html.

[27] Michael Woerister. *rust-gdb. rust-lldb.* 2015. URL: https://michaelwoerister.github.io/2015/03/27/rust-xxdb.html.

[28] David A. Wheeler. *The Apple goto fail vulnerability: lessons learned*. 2017. URL: https://www.dwheeler.com/essays/apple-goto-fail.html.

[29] Ed Felten. *The Linux Backdoor Attempt of 2003*. 2013. URL: https://freedom-to-tinker.com/2013/10/09/the-linux-backdoor-attempt-of-2003.

[30] Rust project developers. *Functions - The Rust Programming Language*. 2018. URL: https://doc.rust-lang.org/book/first-edition/functions.html.

[31] Rust project developers. *Rust RFC 1210 - Specialization*. 2018. URL: https://github.com/rust-lang/rfcs/blob/master/text/1210-impl-specialization.md.

[32] Paul Ducklin. *Anatomy of a security hole – the break that broke sudo*. 2012. URL: https://nakedsecurity.sophos.com/2012/05/21/anatomy-of-a-security-hole-the-break-that-broke-sudo.

[33] Rust project developers. *Patterns - The Rust Programming Language*. 2018. URL: https://doc.rust-lang.org/1.6.0/book/patterns.html.

[34] Rust project developers. *What is Ownership? - The Rust Programming Language, Second Edition*. 2018. URL: https://doc.rust-lang.org/book/second-edition/ch04-01-what-is-ownership.html.

[35] Pete Isensee. *C++ Optimizations You Can Do "As You Go"*. 1998. URL: http://www.tantalon.com/pete/cppopt/asyougo.htm.

[36] Rust project developers. *References and Borrowing - The Rust Programming Language, Second Edition*. 2018. URL: https://doc.rust-lang.org/book/second-edition/ch04-02-references-and-borrowing.html.

[37] Rust project developers. *Foreign Function Interface - The Rustonomicon*. 2018. URL: https://doc.rust-lang.org/nomicon/ffi.html.

[38] Philipp Oppermann. *Page Tables - Writing an OS in Rust*. 2015. URL: https://os.phil-opp.com/page-tables.

[39] Intel Corporation. *Intel® 64 and IA-32 architectures software developer's manual combined volumes: 1, 2A, 2B, 2C, 2D, 3A, 3B, 3C, 3D, and 4*. 2018. URL: https://software.intel.com/sites/default/files/managed/39/c5/325462-sdm-vol-1-2abcd-3abcd.pdf.

[40] OSDev.org Authors. *8259 PIC*. 2018. URL: https://wiki.osdev.org/8259_PIC.

[41] InstLatX64 Authors. *x86, x64 Instruction Latency, Memory Latency and CPUID dumps*. 2018. URL: http://instlatx64.atw.hu.

[42] OSDev.org Authors. *Symmetric Multiprocessing*. 2018. URL: https://wiki.osdev.org/Symmetric_Multiprocessing.

[43] Embedded System Software Group of Technische Universität Dortmund. *Klasse CPU*. 2018. URL: https://ess.cs.tu-dortmund.de/DE/Teaching/WS2017/BSB/Aufgaben/aufgabe4/klassen/cpu.html.

[44] United Computer Wizards. *The PCI ID Repository*. 2018. URL: http://pci-ids.ucw.cz.

[45] Anthony Williams. *Locks, Mutexes, and Semaphores: Types of Synchronization Objects*. 2014. URL: https://www.justsoftwaresolutions.co.uk/threading/locks-mutexes-semaphores.html.

[46] Timo Richter. *Hyper-Threading oder Simultaneous Multithreading*. 2002. URL: http://www.weblearn.hs-bremen.de/risse/RST/WS02/hyperthreading.pdf.

[47] John Regehr. "Inferring Scheduling Behavior with Hourglass". In: *Proceedings of the USENIX 2002 Annual Technical Conference, Freenix Track* (2002), pp. 143–156.

[48] Insyde Software. *BIOS Customizations for Optimized RTOS Performance*. 2013. URL: https://de.slideshare.net/insydesoftware/bios-customizations-for-optimized-rtos-performance.

[49] Philipp Oppermann. *Handling Exceptions - Writing an OS in Rust*. 2017. URL: https://os.phil-opp.com/handling-exceptions.

[50] Rust project developers. *Rust's 2018 roadmap*. 2018. URL: https://blog.rust-lang.org/2018/03/12/roadmap.html.