Exploring Rust for Unikernel Development

Stefan Lankes
slankes@eonerc.rwth-aachen.de
Institute for Automation of Complex Power Systems
RWTH Aachen University
Aachen, Germany

Jens Breitbart
jens.breitbart@de.bosch.com
Bosch Chassis Systems Control,
Robert Bosch GmbH
Abstatt, Germany

Simon Pickartz
pickartz@par-tec.com
ParTec Cluster Competence Center GmbH
Munich, Germany

Abstract

System-level development has been dominated by programming languages like C / C++ for decades. These languages are inherently unsafe, error-prone, and a major reason for vulnerabilities. High-level programming languages with a secure memory model and strong type system are able to improve the quality of the system software. In this paper, we explore the programming language Rust for kernel development and present RustyHermit, which is a unikernel completely written in Rust without any C / C++. We show that the support for RustyHermit can be transparently integratable in the Rust toolchain and common Rust applications are build-able on top of RustyHermit. Previously, we developed the C-based unikernel HermitCore with a similar design to RustyHermit and we are able to compare both kernels. We show that the performance of both kernels is similar and only ~3.27 % of RustyHermit relies on unsafe code, that cannot be checked by the compiler in detail.

1 Introduction

C was invented by Denies Ritchie in 1972 to reduce the usage of assembly in the original UNIX kernel to a minimum. Consequently, it was mainly developed for system software and is still the prevalent programming language for Operating System (OS) kernels. Today, common operating systems are written in C to a great extent. This is mainly motivated by its ability to provide high performance and to allow direct unchecked memory accesses which are required for a small fraction of the kernel and are typically thought of as a requirement for high performance programming languages. However, C is error-prone and is difficult to use in large scale projects as even senior developers can hardly avoid an incorrect usage of C. Dangling pointers and missing boundary checks are other typical reasons for issues within kernel code. As reported by Microsoft Security Response Center [6], about 70 % of Microsoft's vulnerabilities are memory safety issues.

C++ as the follow-up programming language tries to solve these problems. For instance, the C++ standard being introduced in 2011 provides owning and sharing smart pointers to avoid the problem of dangling pointers. However, C++ still relies on developers *to do the right thing* which is in large projects almost impossible to enforce. In contrast to C / C++, modern high-level programming languages with a secure memory model and strong type system are able to avoid most of these issues.

Overall, this is not a new observation and in principle an old discussion. As described in [9], the Pilot kernel [41] and the Lisp machine [15] are early examples of the usage of a high-level language (Mesa and Lisp, respectively) for OS development. However, the approach has not yet gained acceptance and is hardly used. This is because memory safety of high-level languages commonly relies on garbage collection introducing runtime overhead that is proscribed in the area of OSs. Various research projects have improved the performance of garbage collection. However, it is still not negligible. To this day, performance is often considered more important than safety.

Furthermore, operation systems saw a fundamental requirement change over the last years. The basic infrastructure within OSs were established in the seventies, when the hardware was expensive and resource sharing was the focus of OSs. The virtualization of hardware resources has been established to simplify resource sharing such as sharing a processor in round-robin manner. However, in the era of cloud computing, complete machines are virtualized to support server consolidations. This is typically solved by just adding another software layer that allows the modern virtualization techniques, but leaves the old software unchanged. As a result, virtualization adds another layer to an already highly layered software stack, which includes now the support for old physical protocols (e. g., floppy disks), irrelevant optimizations (e. g., disk elevator algorithms on SSDs) and backward-compatible interfaces (e. g., POSIX). Anil Madhavapeddy et. al. discuss these issues in [25, 26] and present

PLOS'19, October 27, 2019, Huntsville, ON, Canada

ACM ISBN 978-1-4503-7017-2/19/10…$15.00
https://doi.org/10.1145/3365137.3365395

unikernels, i. e., specialized library OSs, as a solution. Unikernels are built by compiling high-level languages directly into specialized single-address-space machine images. In doing so, unused code is removed by static code analysis and system calls are replaced by common function calls promising a faster resource handling. Unikernels are able to run directly on a hypervisor or bare metal on the hardware. They provide a smaller footprint compared to traditional OS kernels and have more prospect to optimize the applications. For instance, the application and the kernel can be optimized by means of Link-time Optimization (LTO).

Current Unikernels relinquish backward compatibility, often rely on uncommon programming interfaces, and barely support multi-processor systems. In this paper, we present a rewrite of HermitCore [20] in Rust called RustyHermit and demonstrate that the performance of the Rust implementation is on-par with the original C implementation. RustyHermit is integrated into the standard runtime of Rust, and its compiler infrastructure. It is almost trivial to port an existing Rust application to RustyHermit, as it just requires a configuration change. Furthermore, existing C / C++ and Fortran applications can be linked with RustyHermit and generate a bootable image. Finally, we describe our experience with using Rust for OS development and conclude that it provides various benefits compared to C. We show that the safety-critical area of RustyHermit is only ~3.27 % of the total kernel size.

The rest of this paper is structured as follows: We start with a discussion of the related work in the area of unikernels and the usage of high-level programming languages for kernel development. In Section 3, we give a short introduction to Rust, followed by the Section 4 on kernel development using Rust. In the Sections 6 and 7 we compare the design and performance of our kernels. Section 8 concludes the paper.

2 Related work

High-level programming languages provide type-/memory-safety and convenient abstractions of concurrent programming reducing the susceptibility to errors. However, kernel developers are often skeptical to use new languages because they expect them to introduce additional overhead compared to C [45] and require a redevelopment of kernel components. Yet, many research projects use high-level programming languages to benefit from the new features such as a safe memory handling. New system programming languages, e. g., D [10], Nim [39], Go [14], and Rust [32], have emerged in the last decade. For nearly every language there exists an OS project such as Powernex [40] for D and nimkernel [37] for Nim. From the scientific point of view one of the most interesting projects is Biscuit which is written in Go and analyzed in [9]. Biscuit is able to run bare-metal in contrast to other Go kernels such as Clive [3]. Go uses a garbage collection for the implementation of safe memory handling introducing a certain runtime overhead as discussed before.

In Rust, the compiler is able to determine when memory must be freed avoiding the need for according runtime checks. This results in far less runtime overhead compared to other high-level programming languages, but introduces unique memory handling at language level. Levy et al. [22, 23] show that Rust is attractive for kernel development because it promises memory-safety while providing good performance. In addition, Balasubramanian et al. [2] show that Rust offers software fault isolation (SFI) with lower overhead and Narayanan et al. in [36] steps to realize a Rust-based verified firmware. Currently, Microsoft [7] is also analyzing Rust as a system programming language. Projects such as Redox [43], Tock [44] or teaching kernels like our eduOS-rs [12] show that Rust is usable for OS development, but all these Rust kernels were not written with the goal to compare with C.

Both HermitCore and RustyHermit belong to the class of unikernels or library OSs. Typical representatives of these types *MirageOS* [25], *IncludeOS* [4], *rumprun kernels* [17] and *OSv* [18]. The fundamental drawback of unikernels is the porting effort that is required to adapt existing applications to the underlying minimalistic OS. This often requires both expert work and a considerable amount of time. One objective of the Unikraft [46] project is to build unikernels targeted at specific applications, without requiring the time-consuming, expert work. Unikraft is written in C, uses newlib [42] as the C library, and LwIP [11] as the network stack. However, the compatibility to common OSs (e. g., Linux) is currently still limited. HermiTux [38] has similar objectives and realizes compatibility to Linux by rewriting system calls and using a modified C library. However, the compatibility of HermiTux is limited as not all Linux system calls have been re-implemented.

3 Introduction to Rust

Rust is a new programming language originally designed by Graydon Hoare as a replacement for C / C++. Its goal is to provide the same level of performance, but to allow for more comprehensive safety checks at compile time and by default enabled runtime checks when the compile time checks are not sufficient (e. g., array access with indices not known at compile time). We discuss only the features relevant to understand this paper, a detailed overview on Rust can be found in [1].

Rust relies on *ownerships* to provides safe memory handling without runtime overhead. Each value in Rust has a variable that is called its owner. There is exactly one owner and whenever this owner goes out of scope, the value will be dropped and the memory freed. Ownership can forward the ownership to another variable invalidating the original owner, or the owner can borrow the value to another variable.

The value can be borrowed multiple times if it is borrowed immutable, i. e., the value cannot be changed via the lender, or it can only be borrowed once in case a mutable borrow is required. In general, these rules prevent the dangling pointer problem and prevent pointer aliasing for mutable access. For most tasks it is possible to develop code that these rules are satisfied at compile time, however it is also possible to use `std::cell::RefCell` to bypass compile time checks, but enforce runtime checks.

Similarly, to these checks, Rust also provides compile time checks to ensure that concurrent or parallel code works well. Data that is shared between threads must implement the so-called sync trait or must be wrapped in a mutex that provides this trait. This rule prevents data races, as long as the synchronization mechanism (e. g., the mutex) is implemented correctly. Furthermore, the Rust compiler checks the lifetime of values shared by threads and will not compile code in which a value is not guaranteed to outlive the threads borrowing a value.

All checks named before can be circumvented by using the **`unsafe`** keyword. Unsafe Rust provides the same level of control as C and for example, provides *raw pointers* that allow direct unchecked memory accesses and even allows the usage of inline assembly. Code in unsafe regions should be reviewed more carefully than code that checked by the compiler and as a result are typically frown upon by the Rust community.

4 Kernel development with Rust

A discussed before, software developers should avoid unsafe code, however this is not possible in some areas of the kernel. For instance, the Advanced Programmable Interrupt Controller (APIC) of an x86 processor can be programmed in one of two ways, yet both require the use of unsafe code. The APIC is mapped at a fix physical address, which is not freeable and consequently not manageable like a common memory region. This memory area can directly be accessed by raw pointer. The other way to program the APIC is via the Machine Specific Registers (MSR), which requires assembly as the instructions used to program the MSR are not emitted by the Rust compiler.

One of the most interesting parts for kernel development is splitting the runtime into an OS independent library and an OS dependent library. By implementing Rust's global memory allocator, the *alloc* library [33]—which provides smart pointers and basic data structures such as linked lists, binary heap, ring buffer, and maps—are available and usable in kernel space. To compile these libraries only a target specification file [34] is required, which specifies for instance the processor type and pointer width. Consequently, kernel developers are able to reuse existing, well tested code from the Rust community, which simplifies the development and increases the robustness of the kernel.

```rust
// Describes the CPU state when an
// interrupt arrived
#[repr(C)]
pub struct ExceptionStackFrame {
    pub instruction_pointer: u64,
    pub code_segment: u64,
    pub cpu_flags: u64,
    pub stack_pointer: u64,
    pub stack_segment: u64,
}

extern "x86-interrupt" fn handler(
  stack_frame: &ExceptionStackFrame) {
    /* handle interupt */
}
```

Listing 1. A basic interrupt handler written in Rust.

In contrast to most other high-level programming languages, Rust provides extensions to support low-level programming. A typical example is the support of interrupt handlers for x86 processors. In contrast to common function calls, an interrupt handler has also to restore the privilege level of the interrupted task. Consequently, an interrupt on x86 processors stores automatically the segment selectors of the code and stack segment on the stack, which are used by the interrupt task. In addition, the interrupt handler has to leave with the instruction `iret`[1], which has also to restore the privilege level and to clean up the stack.

A basic interrupt handler written is Rust is shown in Listing 16. In Rust, the keyword *x86-interrupt* [35] is used to mark a function as interrupt handler. The argument `stack_frame` allows the access to the data, which is automatically stored by the hardware. Such marked functions can be directly registered in the interrupt descriptor table. Consequently, it is possible to write an interrupt handler without any usage of unsafe code.

There are various scenarios in kernel development, for which full control over the function stack layout is necessary. A typical example is the implementation of a context switch or the startup code, which initialize for instance the stack. Typically, this is addressed by building the required code with an assembler and linking it to the rest of the OS. However, this breaks the development workflow and the Rust compiler cannot check these parts. Rust supports so called *naked functions* [30], which can be used instead. For such functions, the compiler will not expect a valid stack pointer and does not create a function prologue / epilogue.The usage of naked functions reduces the number of lines of unverifiable code.

The major disadvantage of using Rust for kernel development is the dependency to currently unstable features such

[1] Abbreviation for the *return from interrupt.*

Module	**Reasons for *unsafe* code**
Startup code	Assembly to use control registers
Scheduler	Access to a static task table
Dispatcher	Assembly to store the context
Memory management	Assembly to manage the MMU Access to static data Raw pointer manipulation
Synchronization	Access to the protected data
I / O	Assembly to use machine specific registers and I / O access ports

Table 1. Reasons for *unsafe* code regions.

as naked functions and inline assembly. It is not clear, if and how these features will be supported by the stable version of the Rust compiler.

5 Unsafe code in RustyHermit

A main objective of RustyHermit was to use unsafe code region as little as possible. Table 1 lists different parts of the kernel and reasons for the usage of unsafe code regions. The most commons reason for unsafe code is the direct or the implicit usage of inline assembly. To enable processor features such as AVX, direct access to the control registers is required. To program these registers, RustyHermit reuses the *x86* [27] crate. This crate is also used by other projects and promise a stable interface to the hardware. However, the methods to program the control registers are marked as unsafe to signalize the users that inline assembly is used.

A second frequently reason is currently the usage of static mutable data. For instance, the task table and the synchronization primitive to protect the table is created at startup. Consequently, the global static reference to it is initialized at boot time. To avoid such unsafe code regions, the reference could be protected by a mutex, which provides safe access to static data. However, the data is only initialized at boot time and afterwards only a read access is provided. To avoid synchronization overhead, RustyHermituses currently unsafe code regions to access this data.

6 From a C-based to a Rust-based libOS

As said before, RustyHermit is mostly a rewrite of our 64 bit kernel HermitCore [20, 21], which provides basic OS functionalities, e. g., memory management and priority-based round-robin scheduling. The kernel is completely written in C and supports the Intel 64 Architecture and comes with support for SSE4, AVX2, and AVX512. Although no more than a single process is executed at a time, the kernel still provides a scheduler. Only thus more threads than available cores can be supported. This is important for features of managed programming languages, e. g., garbage collection, or performance monitoring tools. Currently, the scheduler does not support load balancing because explicit thread placement is favored over automatic strategies. The scheduling overhead is reduced to a minimum by the employment of a dynamic timer, i. e., the kernel does not interrupt computation threads which run exclusively on certain cores and do not use any timer.

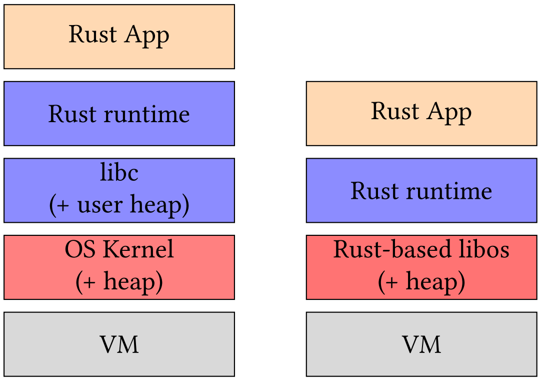

Figure 1. Comparing to build a Rust application on top of a common OS or RustyHermit.

As described in [20], applications can be built by using a cross toolchain which is based on the *GNU Compiler Collection*. Therefore, the kernel supports all programming languages which are supported by gcc. The kernel supports Symmetric Multiprocessing (SMP), uses newlib [42] as C library and LwIP [11] as IP stack. RustyHermit has nearly all features of the C-based kernel, but cannot be used a multikernel. Instead of using LwIP, RustyHermit uses *smoltcp* [24] as user-level IP stack.

Nearly all runtimes of high-level programming languages depend on a C library, which is expected to support POSIX [16] at least on UNIX-like kernel. For instance, the Go runtime of *GNU Compiler Collection* use it to allocate memory, to create pipes and to get access to the IP stack. By using a C library on a POSIX-based system, it is simple to port a runtime to a new OSWe used this approach to support Go on our C-based library OS. However, by using an interface, which is defined for the multi-tasking, multi-user OS Unix, the kernel and the user-space maintain their own heaps. In a library operating is this gratuitous because there is no separation between kernel and user-space.

The left side of Fig. 1, shows the used technique to support the high-level programming language on top of a common operating system or C-based library operating system. As shown on the right-hand side of Fig. 1, in RustyHermit we are able to directly call Rust kernel methods without the indirection via the C library or the POSIX API. In addition, there is no need to switch between safe (Rust) and unsafe code (C). It is also possible to use Rust's ABI between the kernel and Rust's standard runtime. Hereby, the same error handling can be system-wide used and a fall back to error handling by integer number is not required.

The disadvantages of this approach are the required changes to Rust's standard library, which are required to support our kernel. However, the interface between the standard library and the host OSs is relatively small and should easily to maintain. The implementation for every OSs is located in a special directory of Rust's source tree[2] and consist of ~25 files for each operating system. In case of our library OS, ~2500 lines of code (without comments) are required to build the interface to the kernel.

Cargo [31] is Rust package manager and coordinates the build process of Rust binaries. The difference to the typical build process of C / C++ is that the package manager does not install binaries, headers, static or shared libraries. It downloads the source code, compiles it with the same compiler flags and links it directly to the executable. The Rust community calls such kind of packages *crate*. However, for RustyHermit we still build a classical static library, install it as system library, and link it to the binaries. This is required because the library handles all interrupts and the FPU state. Consequently, the kernel should not use the FPU and the red zone of x86_64 ABI [29]. This seems to be easy to realize because a common OS kernel does not rely on any kind of floating-point operations. However, AVX and SSE is part of the FPU handling and is today not longer limited to floating-point operations. Consequently, the Rust compiler use these instructions to optimize the kernel code. By splitting the application from the kernel, we are able to use different CPU features / compiler flags and forbid the usage of AVX / SSE and a red zone in our kernel.

By the full integration of our kernel into the Rust toolchain, cargo can be used to define the dependencies for the application. In principle, every published crate e. g. at crate.io can be used for the executable based on our library OS. The only requirement is that the crate should not directly call the host OS and bypass the Rust's standard runtime.

7 Evaluation

All benchmarks were performed on an Intel Xeon Gold 6132 with 14 physical cores, clocked at 2.6 GHz, equipped with 376 GiB DDR4 RAM and 19.25 MiB L3 cache. Processor features like SpeedStep Technology, TurboMode, and Hyper-threading are deactivated to avoid side effects. We used a 3.10.0 Linux kernel on CentOS 7 installation. All benchmarks are compiled with optimization level 3. Programs, which are written in C and run natively on CentOS7, are compiled with the gcc (Red Hat 4.8.5-36).The C version of HermitCore also all its programs are compiled with gcc 6.3.0, which belong to the HermitCore toolchain. Our Rust toolchain is integrated into Rust's nightly compiler 1.37.0-dev.

Crate	LOC	LOUC	Share in %
RustyHermit	8,834	255	2.89
bitflags	746	0	0.00
log	686	3	0.44
spin	760	24	3.16
x86_64	4,515	314	6.95
cpuid	4,264	36	0.84
multiboot	159	20	12.58
Total	19,964	652	3.27

Table 2. A comparison of the LOC and the LOUC for RustyHermit related crates.

All HermitCore benchmarks run within the lightweight hypervisor *uhyve* [21], Linux benchmarks run natively on CentOS 7.

7.1 Lines of Unsafe Code

In case of Rust, the Lines of Unsafe Code (LOUC) are countable. Consequently, it can be used as a metric to quantify code quality. The smaller the amount of unsafe code, the more the compiler is able to check the code. However, Lines of Code (LOC) metrics are inherently difficult to interpret as not every line is identical. Furthermore, they ignore the developer experiences. However, lines of code serves as an intuitive metric for measuring the size and the complexity of software and in our case the size of unsafe code.

Table 2 shows the number of lines of (safe) code without comments and blank lines and compare it with the number of unsafe lines. The first line of the table shows the results of our kernel, while the rest shows the results of crates we reuse from other projects. These crates are used to simplify the usage of bitflags, to reuse established and often used logging mechanism and spinlocks. The other crates are forked from existing projects and are slightly modified to handle multiboot information [13], to identify the feature set of the x86 processors, and to handle control registers.

The numbers are counted by the cargo extension *cargo-count* [5] and shows that the number of unsafe code is of the kernel is rather small. The largest number of lines of unsafe code has the crate *x86_64*, which has to handle the control registers.

The kernel related part of HermitCore consists of 20.354 lines of C / assembly code and C headers (without comments). RustyHermit only consist of 8,504 LOC. This is mostly achieved by reusing of existing crates.

7.2 OS Micro-Benchmarks

In this section we present benchmarks regarding system call overhead and scheduling. During out benchmark we call `getpid` and `sched_yield` 10 000 000 times and measure the number of cycles the call took. `getpid` is the system call with

[2] https://github.com/rust-lang/rust/tree/master/src/libstd/sys

System activity	Linux	HermitCore	RustyHermit
`getpid()`	1,962	13	36
`sched_yield()`	2,428	108	233
Thread creation	9,782	8,785	7,790
page fault handling	6,969	7,718	9,061

Table 3. Comparison of the CPU cycles required for basic system services by Linux, HermitCore, and RustyHermit respectively.

smallest runtime and closely represents the overhead of a system call. The system call `sched_yield` checks if another task is ready and switches to them. In our case, the system is idle and consequently the system call returns directly after the check of the ready queues. Table 3 summarizes the results as average number of CPU cycles for Linux, HermitCore and RustyHermit. The overhead of HermitCore is clearly smaller because in a library OS the system calls are mapped to common functions. Furthermore, the difference between `getpid` and `sched_yield` on HermitCore is smaller, which proves the small overhead of HermitCore's scheduler. RustyHermit is slightly slower than HermitCore. The schedule time on RustyHermit is ~125 cycles slower.

We also measure the time between calling the function to create a new thread and the first instructions executed within that thread. This is shown in the column *thread creation*. The results show that performance of Rust is comparable and, in this case, also faster.

All tested operating systems are able to bind memory on demand. The first access to a page triggers a page fault and its handler will allocate a page frame and map into the virtual address space. To benchmark the overhead of a page fault, we measure the cycles it takes to write just 1 Byte to an unmapped page. Huge pages are disable for this benchmark. In case of a hypervisor, we typically have one page fault on the host side and one on the guest side. To measure only the overhead within the guest, the hypervisor initializes the memory before starting the guest.

Both unikernels use free lists to maintain the virtual and physical address space. Rust seems to slightly increase the overhead, yet we consider ~1300 cycles acceptable to avoid memory issues.

7.3 Data parallelism with Rayon

Rayon [28] is a data-parallelism library for Rust. It is comparable with *Threading Building Blocks* [8] for C++.

To evaluate the performance of Rayon on top of our HermitCore, we use a parallel version of the matrix multiplication based on the Strassen algorithm. The implementation is part of Rayon's demo suite.

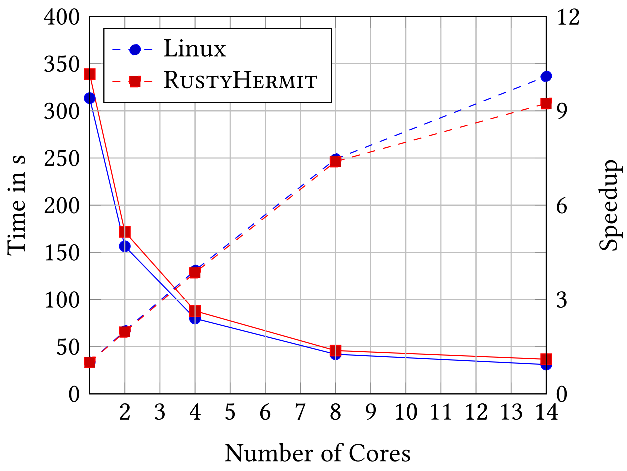

Figure 2. The scalability of the matmul benchmark as part of Rayon's demo suite showing the absolute runtime (left axis; solid) and the speedup (right axis; dashed) for different core counts respectively.

Figure 2 shows the performance of the matrix multiplication for matrices of dimension of 8192*x*8192. The performance of RustyHermit is similar to that of with the native Linux version. The runtime difference could be caused by using a Virtual Machine (VM) and must be investigated further.

To create the application for RustyHermit, only one function for determining the number of existing processors had to be replaced. Otherwise, all additional crates were used unmodified. We plan to merge these changes to the crate *num_cpus*, which Rayon used to determine the number of CPUs. Rayon itself does not require any modifications to be work on RustyHermit.

8 Conclusion

In this paper, we present RustyHermit, which is completely written in Rust and does not use C / C++. RustyHermit is published on GitHub [19] and is completely integrated into Rust's toolchain. Consequently, common Rust applications, which do not bypass the Rust runtime and directly use OS services are able to run on RustyHermit without modification.

We compare RustyHermit with the C-based kernel HermitCore and Linux and showed that the performance of RustyHermit is similar to the other solutions. The major advantage of Rust is that unsafe code must be marked explicitly and that only ~3.27 % of RustyHermit are unsafe.

Acknowledgments

This research and development was supported by the German Federal Ministry of Education and Research under Grant 01IH16010C (Project ENVELOPE).

References

[1] S. Klabnik andn C. Nichols. 2018. *The Rust Programming Language (Manga Guide)*. No Starch Press, San Francisco, CA, USA.

[2] Abhiram Balasubramanian, Marek S. Baranowski, Anton Burtsev, Aurojit Panda, Zvonimir Rakamari, and Leonid Ryzhyk. 2017. System Programming in Rust: Beyond Safety. *SIGOPS Oper. Syst. Rev.* 51, 1 (Sept. 2017), 94–99. https://doi.org/10.1145/3139645.3139660

[3] Ballesteros, Francisco J. 2015. The Clive Operating System. (March 2015), 1–15. http://lsub.org/ls/clive.html.

[4] A. Bratterud, A. Walla, H. Haugerud, P. E. Engelstad, and K. Begnum. 2015. IncludeOS: A Resource Efficient Unikernel for Cloud Services. In *Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*.

[5] Cargo-Count. 2019 (accessed August 1, 2019). *A cargo subcommand for counting lines of code in Rust projects*. https://github.com/kbknapp/cargo-count.

[6] Microsoft Security Responce Center. 2019 (accessed August 1, 2019). *We need a safer systems programming language*. https://msrc-blog.microsoft.com/2019/07/18/we-need-a-safer-systems-programming-language/.

[7] Microsoft Security Responce Center. 2019 (accessed August 1, 2019). *Why Rust for safe systems programming*. https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming/.

[8] Intel Corporation. 2019 (accessed August 1, 2019). *Threading Building Blocks*. https://www.threadingbuildingblocks.org.

[9] Cody Cutler, M Frans Kaashoek, and Robert T. Morris. 2018. The benefits and costs of writing a POSIX kernel in a high-level language. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 1–19.

[10] D Language Foundation. 2019 (accessed March 4, 2019). *The D Programming Language*. https://dlang.org/.

[11] A. Dunkels. 2001. *Design and Implementation of the LwIP TCP/IP Stack*. Swedish Institute of Computer Science.

[12] eduOS-rs. 2019 (accessed February 13, 2019). *A teaching operating system written in Rust*. https://rwth-os.github.io/eduOS-rs/.

[13] Inc. Free Software Foundation. 2019 (accessed August 1, 2019). *Multiboot Specification version 0.6.96*. https://www.gnu.org/software/grub/manual/multiboot/multiboot.html.

[14] Google. 2019 (accessed March 4, 2019). *The Go Programming Language*. https://golang.org.

[15] Richard D Greenblatt, Thomas F Knight, John T Holloway, and David A Moon. 1980. A LISP machine. *ACM SIGIR Forum* 15, 2 (April 1980), 137–138.

[16] IEEE and The Open Group. 2019 (accessed August 1, 2019). *The System Interfaces volume of POSIX.1-2017*. http://pubs.opengroup.org/onlinepubs/9699919799/.

[17] A. Kantee. 2012. *Flexible Operating System Internals – The Design and Implementation of the Anykernel and Rump Kernels*. Ph.D. Dissertation. Department of Computer Science and Engineering, Aalto University, Aalto, Finland.

[18] Avi Kivity, Dor Laor, Glauber Costa, Pekka Enberg, Nadav Har'El, Don Marti, and Vlad Zolotarov. 2014. OSv - Optimizing the Operating System for Virtual Machines. *USENIX Annual Technical Conference* (2014).

[19] Stefan Lankes, Jens Breitbart, and Simon Pickartz. 2019 (accessed October 3, 2019). *RustyHermit – A Rust-based, lightweight unikernel*. https://github.com/hermitcore/libhermit-rs.

[20] S. Lankes, S. Pickartz, and J. Breitbart. 2016. HermitCore: A Unikernel for Extreme Scale Computing. In *Proc. of the 6th International Workshop on Runtime and Operating Systems for Supercomputers (ROSS '16)*. ACM, New York, NY, USA, Article 4, 8 pages.

[21] S. Lankes, S. Pickartz, and J. Breitbart. 2017. A Low Noise Unikernel for Extrem-Scale Systems. In *30th International Conference on Architecture of Computing Systems (ARCS 2017), Vienna, Austria, April 3–6, 2017*. Springer International Publishing, 73–84. https://doi.org/10.1007/978-3-319-54999-6_6

[22] Amit Levy, Michael P Andersen, Bradford Campbell, David Culler, Prabal Dutta, Branden Ghena, Philip Levis, and Pat Pannuto. 2015. *Ownership is theft: experiences building an embedded OS in Rust*. ACM, New York, New York, USA.

[23] Amit Levy, Bradford Campbell, Branden Ghena, Pat Pannuto, Prabal Dutta, and Philip Levis. 2017. The Case for Writing a Kernel in Rust. *Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys 2017)* (2017), 1–7.

[24] M-Labs. 2019 (accessed August 1, 2019). *uhyve - A minimal hypervisor for RustyHermit*. https://github.com/m-labs/smoltcp.

[25] A. Madhavapeddy, R. Mortier, C. Rotsos, D. Scott, B. Singh, T. Gazagnaire, S: Smith, S. Hand, and J. Crowcroft. 2013. Unikernels: Library Operating Systems for the Cloud. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 461–472. https://doi.org/10.1145/2451116.2451167

[26] Anil Madhavapeddy and David J Scott. 2013. Unikernels: Rise of the Virtual Library Operating System. *ACM Queue* 11, 11 (Nov. 2013), 30.

[27] B. Martin, C. Boylan, D. Schatzberg, E. Kidd, G. Zellweger, J. Erricson, N. Edigaryev, P. Oppermann, and R. Lunae. 2019 (accessed September 29, 2019). *Rust library to use x86 (amd64) specific functionality and registers*. https://github.com/gz/rust-x86.

[28] Niko Matsakis and Josh Stone. 2019 (accessed Azgust 1, 2019). *Rayon — A data parallelism library for Rust*. https://github.com/rayon-rs/rayon.

[29] M. Matz1, J. Hubicka, Andreas Jaeger, and M. Mitchell. 2014 (accessed August 1, 2019). *System V Application Binary Interface – AMD64 Architecture Processor Supplement*. https://www.uclibc.org/docs/psABI-x86_64.pdf.

[30] Mozilla. 2019 (accessed March 11, 2019). *Add support for generating naked functiosn*. https://github.com/nox/rust-rfcs/blob/master/text/1201-naked-fns.md.

[31] Mozilla. 2019 (accessed March 4, 2019). *Cargo – A Rust package manager*. https://doc.rust-lang.org/cargo/.

[32] Mozilla. 2019 (accessed March 4, 2019). *The Rust Programming Language*. https://www.rust-lang.org.

[33] Mozilla. 2019 (accessed March 7, 2019). *The Rust core allocation and collections library*. https://doc.rust-lang.org/alloc/index.html.

[34] Mozilla. 2019 (accessed March 7, 2019). *Target specification files*. https://github.com/japaric/rust-cross#target-specification-files.

[35] Mozilla. 2019 (accessed March 9, 2019). *Add support for the x86-interrupt calling convention*. https://github.com/rust-lang/rust/pull/39832.

[36] Vikram Narayanan, Marek S. Baranowski, Leonid Ryzhyk, Zvonimir Rakamarić, and Anton Burtsev. 2019. RedLeaf: Towards An Operating System for Safe and Verified Firmware. (2019), 37–44. https://doi.org/10.1145/3317550.3321449

[37] nimkernel. 2019 (accessed March 4, 2019). *An operating system written in Nim*. https://github.com/dom96/nimkernel.

[38] P. Olivier, D. Chiba, S. Lankes, C. Min, and B. Ravindran. 2019. A Binary-Compatible Unikernel. In *15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'19)*. Accepted for publication.

[39] D. Picheta. 2019 (accessed March 4, 2019). *Nim in action*. http://nim-lang.org/.

[40] PowerNex. 2019 (accessed March 4, 2019). *An operating system written in D*. https://github.com/PowerNex/PowerNex.

[41] David D Redell, Yogen K Dalal, Thomas R Horsley, Hugh C Lauer, William C Lynch, Paul R McJones, Hal G Murray, and Stephen C Purcell. 1980. Pilot – An Operating System for a Personal Computer. *Commun. ACM* 23, 2 (1980), 81–92.

[42] RedHat. 2019 (accessed February 13, 2019). *Newlib – A C library for embedded systems*. https://sourceware.org/newlib/.

[43] Redox. 2019 (accessed February 13, 2019). *A Unix-like Operating System written in Rust*. https://www.redox-os.org.

[44] Tock. 2019 (accessed March 4, 2019). *A secure embedded operating system for Cortex-M based microcontrollers*. https://www.tockos.org.

[45] L. Torvalds. 2004. . http://harmful.cat-v.org/software/c++/linus.

[46] Unikraft. 2019 (accessed March 4, 2019). *An easy way of crafting Unikernels*. http://unikraft.neclab.eu.