RustyHermit: A Scalable, Rust-based Virtual Execution Environment

Stefan Lankes[1], Jonathan Klimt[1], Jens Breitbart[2], and Simon Pickartz[3]

[1] Institute for Automation of Complex Power Systems
RWTH Aachen University {slankes,jonathan.klimt}@eonerc.rwth-aachen.de
[2] Bosch Chassis Systems Control, Robert Bosch GmbH
jens.breitbart@de.bosch.com
[3] ParTec Cluster Competence Center GmbH pickartz@par-tec.com

Abstract. System-level development has been dominated by programming languages such as 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. This paper explores the programming language Rust for development of a scalable, virtual execution environment and presents the integration of a Rust-based IP stack into RustyHermit. RustyHermit is part of the standard Rust toolchain and common Rust applications are able to build on top of RustyHermit.

1 Introduction

The C programming language is still dominating system-level software development as it was designed for this exact case and is known to provide high performance. However, C is also known to be error-prone and 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. This is not a new observation. As described in [6], the Pilot kernel [29] and the Lisp machine [11] are early examples of the usage of a high-level language (Mesa and Lisp, respectively) for Operating System (OS) development. However, the approach has not gained acceptance and is hardly used because memory safety of high-level languages often induces runtime overhead (e. g., due to garbage collection).

Furthermore, the OS requirements changed fundamentally over the last years. The basic infrastructure within OSs was established in the seventies when hardware was expensive and resource sharing was the focus. The virtualization of hardware resources has been established for a simplified resource sharing, e. g., sharing a processor in round-robin manner. However, in the era of cloud computing, complete machines are virtualized supporting server consolidations. Virtualization is implemented as another software abstraction layer in an already highly layered software stack. Typical modern OSs still include 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 [22,21] and present 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, e. g., 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 [16], we present a rewrite of HermitCore [14] in Rust called RustyHermit and demonstrate that the performance of the Rust implementation is on a par with the original C implementation. RustyHermit is integrated into the standard runtime of Rust and its compiler infrastructure. It is trivial to port pure Rust application to RustyHermit, as it just requires a configuration change. Only applications, which bypass the Rust runtime and call directly a C library, have to port also the C library to the new systen. Furthermore, existing C / C++ and Fortran applications can be linked with RustyHermit and generate a bootable image. In this paper, we focus on the integration of a Rust-based IP stack enabling the building and deployment of secure and efficient cloud applications.

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 and the integration of the IP stack. In the Section 5 we compare the performance of our kernel with Linux. Finally, Section 6 summarizes the paper and give a short outlook.

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 [34] and require a redevelopment of kernel components. Yet, many research projects use high-level programming languages to benefit from new features such as a safe memory handling. New system programming languages, e. g., D [7], Nim [28], Go [10], and Rust [24], have emerged in the last decade. For nearly every language there exists an OS project. From the scientific point of view one of the most interesting projects is Biscuit which is written in Go and analyzed in [6]. Biscuit is able to run bare-metal in contrast to other Go kernels such as Clive [2]. 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 the language level. Levy et al. [18,19] show that Rust is attractive for kernel development because it promises memory-safety while providing good performance. In addition, Balasubramanian et al. [1] show that Rust offers software fault isolation (SFI) with lower overhead and Narayanan et al. in [25] steps to realize a Rust-based verified firmware. Currently, Microsoft [5] is also analyzing Rust as a system programming language. Projects such as Redox [31], Tock [33] or teaching kernels such as our eduOS-rs [9] show that Rust is usable for OS development, but all these Rust kernels were not designed for cloud environments.

Both HermitCore and RustyHermit belong to the class of unikernels or library OSs. *MirageOS* [21], *IncludeOS* [3], *rumprun kernels* [12], and *OSv* [13] are typical representatives. 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 [35] project is to build unikernels targeted at specific applications, without requiring the time-consuming, expert work. Unikraft is written in C, uses newlib [30] as the C library, and LwIP [8] as the network stack. However, the compatibility to common OSs (e. g., Linux) is currently still limited. HermiTux [27] 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. RustyHermit is also not compatible to common OSs, but it offers the possibility to write portable Rust applications. Changes to the source code are not required to run the application on Linux or other OSs.

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 [4].

Rust relies on *ownership* to provide safe memory handling without runtime overhead. Each resource (e. g., memory) in Rust has a variable that is called its owner. There is exactly one owner at a time and whenever this owner goes out of scope, the resource will be dropped and the memory freed. Ownership can be forwarded to another variable invalidating the original owner, or the owner can borrow the resource to another variable. Read only access can be provided to multiple variables at a time via immutable borrows, as long as no mutable borrow is happening at the same time. In general, these rules prevent data races, the dangling pointer problem, and 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 provides compile time checks as well ensuring the correct execution of concurrent or parallel code. Data that is shared between threads must implement the so-called `sync` trait (the rust term for an interface) or must be wrapped into a mutex providing 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 code provides the same level of control as C, e. g., it provides *raw pointers* enabling direct, unchecked memory accesses and even supports the usage of inline assembly. Code in unsafe regions should be reviewed more carefully than code that is checked by the compiler and as a result it is typically frown upon by the Rust community to use this feature. Currently, it is not possible to write a complete kernel without the usage of unsafe code. For instance, inline assembly is important to restore the context of the FPU. However, the presented library operating system only requires 1170 lines of unsafe code corresponding to only 1.71 % of total code size.

The Rust standard library is divided into an OS-independent and an OS-dependent part. The library known as *core* library is the major part of the OS-independent library and already implements basic error / panic handling, string operations, and atomic operations. Furthermore, Rust offers the possibility to redefine the global memory allocator. This allocator is used by all other Rust codes unless explicitly circumvented. In contrast, the part known as *std* condenses the OS-dependent libraries and extends them with various data structures, console output, and thread handling. It is easily possible to create a project that does not use *std* by adding `#![no_std]` to the main file.

4 A unikernel written in Rust

RustyHermit is a rewrite of our 64 bit unikernel HermitCore [14,15] which was written in C. RustyHermit is completely written in Rust, supports the Intel 64 Architecture and comes with support for SSE4, AVX2, and AVX512. It has multi-core and single-core multiprocessing support by the means of multithreading and multiprocessing. The Kernel supports the execution of more threads than available cores. This is an important feature for dealing with concurrent applications or to integrate performance monitoring tools. Currently, the scheduler does not support load balancing as explicit thread placement is favored over automatic strategies. 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. To improve the security behavior, RustyHermit provides a stack guard and is completely position-independent. Consequently, the loader is able to randomize the memory layout.

4.1 Integration of RustyHermit into libstd

One major goal of RustyHermit was a complete integration into the Rust toolchain to simplify the application development. Any common Rust application should be buildable with RustyHermit. To achieve this goal, the kernel provides the required interfaces to the Standard Library (*libstd*) whilst being based only on the *core* library. The operating system abstraction layer of the Rust toolchain is relative small, so only around 26 files within a total of ˜3000 lines of code are required to integrate RustyHermit into the standard library of Rust.

Most operating systems are written in C and use a common C library as interface to the kernel. These functions are typically provided by a helper crate[4] in Rust realizing an interface to the C functions. For instance, the C interface for Rust is published in the crate *libc*[5]. All functions within this helper crate are marked as *unsafe* and as a consequence the interface is not checkable by the Rust compiler.

In case of RustyHermit, the complete kernel is written in Rust and theoretically, it could be directly integrated into the Rust standard library. However, the kernel uses a set of external crates to detect processor features, programming of the interrupt controller, or log messages. As the Rust community wants to reduce the dependencies of the basic runtime libraries to external crates, we cannot integrate RustyHermit into *libstd* directly. Instead, we create two helper crates *hermit-abi*[6] and *hermit-sys*[7]. The former describes only the interface to the library operating system for linkage and is included in *libstd*s dependencies, just like the *libc* crate does for the Linux interface of the Standard Library. The latter is a helper crate, with the main purpose of building the kernel as static library from source and linking it to the application.

Separating the kernel and *libstd* into separate compilation units also allows the use of different compiler settings for each of them. Hereby, we are able to disable the FPU and AVX / SSE support for the kernel and to enable it for the rest of the application. This is necessary because AVX and SSE is not longer limited to floating-point operations and the compiler would use these instructions to optimize the kernel code. The usage of AVX and SSE within the kernel could trigger interrupts to save the FPU context—something which should be certainly avoided.

[4] Crate is a tree of modules that produces a library or executables. Much like a package in other programming languages.

[5] `https://crates.io/crates/libc`.

[6] `https://crates.io/crates/hermit-abi`.

[7] `https://crates.io/crates/hermit-sys`.

Listing 1.1: Extension of Cargo.toml to integrate RustyHermit

```
[target.'cfg(target_os = "hermit")'.dependencies]
hermit-sys = "0.1.*"
```

To make a Rust application a RustyHermit application, it is sufficient to include the *hermit-sys* crate by adding it to the applications dependencies as shown in Listing 1.1 and declaring it as an `external crate` in the applications source. Rust's package manager Cargo [23] will then download the kernel's sources, compile it, and link it to the application.

4.2 Network Support

The library operating system only provides basic features such as interrupt handling, device drivers, memory management, and scheduling. One solution to integrate network support, is the use of real hardware drivers. The hypervisor emulates these devices by trapping every request to the device and emulating the behavior of the real hardware (*trap and emulate*). This approach comes with an important overhead.

An alternative to this is *para-virtualization* where the hypervisor provides a simpler and faster interface for the I / O devices to the guest, who is aware of running on a hypervisor. Today, *virtio* is the standard abstraction layer [26] for these para-virtualized I / O devices on KVM-accelerated hypervisors. The driver is split into two parts: the frontend and the backend. The former is provided by the guest kernel while the backend is provided by the host. This abstraction layer can be used for para-virtualization of any I / O device. In case of a network interface, there exist at least two buffers. One buffer is handling all incoming packets, while the second buffer is handling all outgoing packets. The original version of virtio [32] was developed by Rusty Russell for the support of his own virtualization solution. RustyHermit provides a frontend driver within the kernel which is used to realize file system access by *virtio-fs*[8] and network support.

As shown in Fig. 1, RustyHermit uses *smoltcp* [20] as a dual IPv4 / IPv6 stack and is provided by *hermit-sys* to the Rust runtime. *smoltcp* is an event-driven TCP/IP stack being completely realized in Rust and designed for bare-metal, real-time systems. In principle, *hermit-sys* creates a thread, which handles all incoming packets including ARP and ICMP packets with the help of *smoltcp*. Putting the IP stack into *hermit-sys* and not directly into the kernel offers the option to use the memory allocator of the Rust runtime and to enable hardware dependent optimizations (e. g., AVX support) as explained in Section 4.1. To provide TCP streams to a common Rust application, an interface between *smoltcp* and Rust's standard library is required. In this case, the data or at least the reference to the data for a certain stream has to be forwarded from the thread

[8] `https://virtio-fs.gitlab.io`

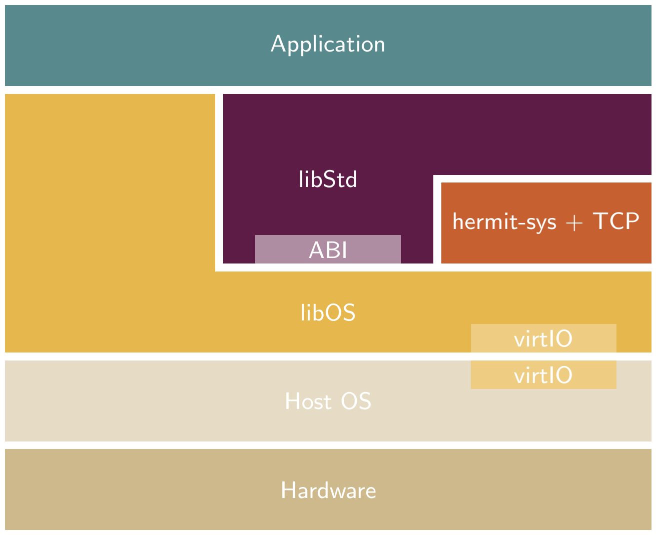

Fig. 1: Architecture overview of RustyHermit.

handling the IP packets to the thread waiting for the data or producing the data. Rust's standard library provides a synchronization channel based on a multi-producer, single-consumer FIFO queue and is used to realize the communication between these threads.

5 Evaluation

All benchmarks were performed on a NUMA system possessing two sockets each with 12 physical cores, exposing 24 cores in total. The CPUs are Intel Skylake CPUs (Xeon Gold 6128) clocked at 3.4 GHz, equipped with 256 GiB DDR4 RAM and 19.25 MiB L3 cache. Processor features such as SpeedStep Technology, TurboMode, and Hyperthreading are deactivated to avoid side effects. We used a 4.18.0 Linux kernel with CentOS 8. All benchmarks are compiled with optimization level 3.

As said before, unikernels are designed to run within a hypervisor. For the evaluation, Qemu 2.12.0 is used and accelerated by KVM. All benchmarks run within virtual machines with the same setup. The network interface and the storage is integrated by virtio to reduce the overhead. The only difference is that for Linux guests the virtual machine is configured to provide 4 GB of main memory, while RustyHermit is configured with 512 MByte main memory.

5.1 OS Micro-Benchmarks

In this section we present benchmarks regarding system call overhead and scheduling. The `getpid` system call is the one with the smallest runtime and closely represents the overhead of a system call. The function `yield_now` of the Rust runtime triggers the scheduler to check if another task is ready and switches to them. In our case, the system is idle and consequently the function returns directly after the check of the ready queues. For benchmarking the system call performance, we call `getpid` and `yield_now` 1 000 000 times and measure the number of cycles the call took. Table 1 summarizes the results as average number of CPU cycles for Linux and RustyHermit. The overhead of RustyHermit is clearly smaller because in a library OS the system calls are mapped to common functions and the runtime system is cleary smaller in comparsion to the Linux software stack. A performance improvements by using Rust's LTO support is in these micro benchmarks not measureable.

Table 1: Comparison of basic system services by Linux and RustyHermit.

System activity	**Linux**	**RustyHermit**
Time to boot	≤15 s	≤1.0 s
Reserved memory	748 MByte	55 MByte
Boot image size	1.8 GByte	1.6 MByte
`yield_now()`	1439 cycles	68 cycles (70 cycles wo LTO)
`getpid()`	1147 cycles	43 cycles (43 cycles wo LTO)

Table 1 shows also memory consumption of a minimal CentOS 8 configuration, where only a secure shell server is running and compares it with the memory consumption of the smallest possible RustyHermit application. To determine these numbers, the memory consumption of the hypervisor on the host system is evaluated. The numbers show the physically allocated memory. The reserved memory in the logical address space is clearly larger because the virtual machines are configured to use up to 4 GByte memory for the Linux guest and 500 MByte for RustyHermit as guest. Both virtual machines are not fully utilized. The low memory consumption and the small image size for RustyHermit promise a better utilization of system in data centers.

To evaluate the boot time, the time between the start of the virtual machine and the first response of a ICMP-based ping request is measure. To avoid side effects from the storage device, the boot image is stored in *tmpfs*. The last step before entering the main function of the Rust application in RustyHermit is the initialization of network stack. Therefore, the results show the minimal time to start the unikernel application within a hypervisor. While it is possible to start applications in Linux before the network services have started, this is a rather unlikely scenario and it is more likely that other services are started between the network service start and the application start. As expected for a unikernel, RustyHermit is clearly faster in comparison to Linux which is beneficial for services requiring low latencies.

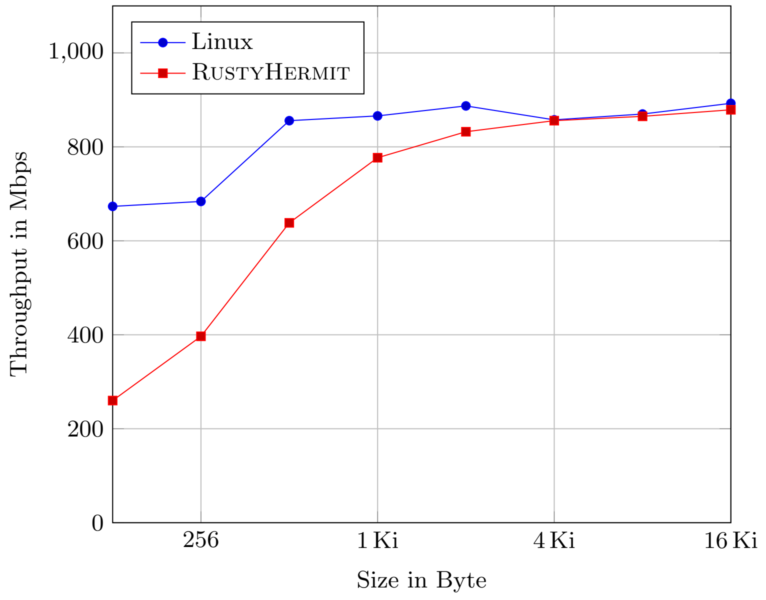

Fig. 2: Comparsion of the network throughput between RustyHermit and Linux

5.2 Network performance

To determine the network performance, a benchmark is used transferring data with Rust's standard TCP stream interface. Both the server and the client are running on two different nodes and are connected through a standard Ethernet interface with theoretical bandwidth of 1 Gbps. The receiver is in both cases a Linux process running natively on the hardware while the senders are running within a virtual machine. In case of the senders, the checksums of the IP packets are built within the guest machine. All interfaces use an MTU of 1500 Bytes and the Nagle algorithm is disabled.

Figure 2 compares the performance between RustyHermit and Linux. The latter provides higher bandwidth for small messages. We expect this to be a result of a misconfigured packet transmission interrupt, however this requires a deeper analysis[9]. The peak performance is equivalent between RustyHermit and Linux.

6 Conclusion

In this paper, we present RustyHermit a unikernel completely written in Rust. We integrate a Rust-based IP stack not depending on C / C++. RustyHermit is published on GitHub [17] and is completely integrated into Rust's toolchain.

[9] We intend to either fix this issue or provide a detailed explanation in case the paper is accepted for publication.

Consequently, common Rust applications, which do not bypass the Rust runtime and directly use OS services are able to run on RustyHermit without modifications.

We show that RustyHermit provides excellent performance in micro benchmarks and provides a small memory footprint compared to a minimal CentOS 8 virtual machine image. A deeper analysis is required to optimize the IP stack for small messages. However, in combination with the low memory footprint is RustyHermit already suitable for the development of micro services.

References

1. Balasubramanian, A., Baranowski, M.S., Burtsev, A., Panda, A., Rakamari, Z., Ryzhyk, L.: System programming in rust: Beyond safety. SIGOPS Oper. Syst. Rev. **51**(1), 94–99 (Sep 2017). https://doi.org/10.1145/3139645.3139660, `http://doi.acm.org/10.1145/3139645.3139660`
2. Ballesteros, Francisco J.: The Clive Operating System pp. 1–15 (Mar 2015), `http://lsub.org/ls/clive.html`
3. Bratterud, A., Walla, A., Haugerud, H., Engelstad, P.E., Begnum, K.: IncludeOS: A Resource Efficient Unikernel for Cloud Services. In: Proceedings of the 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom) (2015)
4. andn C. Nichols, S.K.: The Rust Programming Language (Manga Guide). No Starch Press, San Francisco, CA, USA (2018)
5. Center, M.S.R.: Why rust for safe systems programming (2019 (accessed August 1, 2019)), `https://msrc-blog.microsoft.com/2019/07/22/why-rust-for-safe-systems-programming/`
6. Cutler, C., Kaashoek, M.F., Morris, R.T.: 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). pp. 1–19 (Sep 2018)
7. D Language Foundation: The D Programming Language (2019 (accessed March 4, 2019)), `https://dlang.org/`
8. Dunkels, A.: Design and Implementation of the LwIP TCP/IP Stack. Swedish Institute of Computer Science (2001)
9. eduOS-rs: A teaching operating system written in rust (2019 (accessed February 13, 2019)), `https://rwth-os.github.io/eduOS-rs/`
10. Google: The Go Programming Language (2019 (accessed March 4, 2019)), `https://golang.org`
11. Greenblatt, R.D., Knight, T.F., Holloway, J.T., Moon, D.A.: A LISP machine. ACM SIGIR Forum **15**(2), 137–138 (Apr 1980)
12. Kantee, A.: Flexible Operating System Internals – The Design and Implementation of the Anykernel and Rump Kernels. Ph.D. thesis, Department of Computer Science and Engineering, Aalto University, Aalto, Finland (2012)
13. Kivity, A., Laor, D., Costa, G., Enberg, P., Har'El, N., Marti, D., Zolotarov, V.: OSv - Optimizing the Operating System for Virtual Machines. USENIX Annual Technical Conference (2014)
14. Lankes, S., Pickartz, S., Breitbart, J.: HermitCore: A Unikernel for Extreme Scale Computing. In: Proc. of the 6th International Workshop on Runtime and Operating Systems for Supercomputers. pp. 4:1–4:8. ROSS '16, ACM, New York, NY, USA (2016)

15. Lankes, S., Pickartz, S., Breitbart, J.: 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. pp. 73–84. Springer International Publishing (2017)
16. Lankes, S., Breitbart, J., Pickartz, S.: Exploring rust for unikernel development. In: Proceedings of the 10th Workshop on Programming Languages and Operating Systems. pp. 8–15. PLOS'19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3365137.3365395
17. Lankes, S., Breitbart, J., Pickartz, S.: Rustyhermit – a rust-based, lightweight unikernel (2019 (accessed October 3, 2019)), https://github.com/hermitcore/libhermit-rs
18. Levy, A., Andersen, M.P., Campbell, B., Culler, D., Dutta, P., Ghena, B., Levis, P., Pannuto, P.: Ownership is theft: experiences building an embedded OS in Rust. experiences building an embedded OS in rust, ACM, New York, New York, USA (Oct 2015)
19. Levy, A., Campbell, B., Ghena, B., Pannuto, P., Dutta, P., Levis, P.: The Case for Writing a Kernel in Rust. Proceedings of the 8th Asia-Pacific Workshop on Systems (APSys 2017) pp. 1–7 (2017)
20. M-Labs: uhyve - a minimal hypervisor for rustyhermit (2019 (accessed August 1, 2019)), https://github.com/m-labs/smoltcp
21. Madhavapeddy, A., Mortier, R., Rotsos, C., Scott, D., Singh, B., Gazagnaire, T., Smith, S., Hand, S., Crowcroft, J.: Unikernels: Library Operating Systems for the Cloud. In: Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems. pp. 461–472. ASPLOS '13, ACM, New York, NY, USA (2013). https://doi.org/10.1145/2451116.2451167
22. Madhavapeddy, A., Scott, D.J.: Unikernels: Rise of the Virtual Library Operating System. ACM Queue **11**(11), 30 (Nov 2013)
23. Mozilla: Cargo – a rust package manager (2019 (accessed March 4, 2019)), https://doc.rust-lang.org/cargo/
24. Mozilla: The Rust Programming Language (2019 (accessed March 4, 2019)), https://www.rust-lang.org
25. Narayanan, V., Baranowski, M.S., Ryzhyk, L., Rakamarić, Z., Burtsev, A.: Redleaf: Towards an operating system for safe and verified firmware pp. 37–44 (2019). https://doi.org/10.1145/3317550.3321449, http://doi.acm.org/10.1145/3317550.3321449
26. OASIS Virtual I/O Device (VIRTIO) TC: Virtual I/O Device (VIRTIO) Version 1.1 (2018), https://docs.oasis-open.org/virtio/virtio/v1.1/csprd01/virtio-v1.1-csprd01.html
27. Olivier, P., Chiba, D., Lankes, S., Min, C., Ravindran, B.: A binary-compatible unikernel. In: 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'19) (2019), accepted for publication
28. Picheta, D.: Nim in action (2019 (accessed March 4, 2019)), http://nim-lang.org/
29. Redell, D.D., Dalal, Y.K., Horsley, T.R., Lauer, H.C., Lynch, W.C., McJones, P.R., Murray, H.G., Purcell, S.C.: Pilot – An Operating System for a Personal Computer. Commun. ACM **23**(2), 81–92 (1980)
30. RedHat: Newlib – a c library for embedded systems (2019 (accessed February 13, 2019)), https://sourceware.org/newlib/
31. Redox: A unix-like operating system written in rust (2019 (accessed February 13, 2019)), https://www.redox-os.org

32. Russell, R.: Virtio: Towards a de-facto standard for virtual i/o devices. SIGOPS Oper. Syst. Rev. **42**(5), 95–103 (Jul 2008). https://doi.org/10.1145/1400097.1400108, `https://doi.org/10.1145/1400097.1400108`
33. Tock: A secure embedded operating system for cortex-m based microcontrollers (2019 (accessed March 4, 2019)), `https://www.tockos.org`
34. Torvalds, L.: (Jan 2004), `http://harmful.cat-v.org/software/c++/linus`
35. Unikraft: An easy way of crafting unikernels (2019 (accessed March 4, 2019)), `http://unikraft.neclab.eu`