5-24-2019

Wayless: a Capability-Based Microkernel

Waylon Cude
Portland State University

Wayless: A Capability-Based Microkernel

by
Waylon Cude

An undergraduate honors thesis submitted in partial fulfillment of the
requirements for the degree of
Bachelor of Science
in
University Honors
and
Math

Thesis Adviser
Bart Massey

Portland State University
2019

Abstract

Microkernels are a family of operating system kernels characterized by their small codebase. Due to their small size microkernels provide extra security and reliability as compared to larger, monolithic kernels. When combined with a capability-based architecture these benefits are enhanced, allowing for even greater security through the principle of least privilege. The seL4 microkernel, written in C and formally verified, takes advantage of this capability-based architecture.

Formal verification is an incredibly powerful tool—however, its cost is significant. Rust is a strongly-typed systems programming language that provides numerous safety advantages over C and C++, completely eliminating large classes of bugs. This makes the Rust programming language a good candidate for operating system development.

This thesis describes Wayless, a capability-based microkernel heavily inspired by seL4. The kernel aims to provide an seL4-compatible application binary interface (ABI), letting seL4 programs run without modification on Wayless. Wayless provides some of the same safety benefits as seL4 through use of the Rust programming language, without going through the process of formal verification. Development on Wayless is still in its early stages but progress has been promising: Wayless already implements most of seL4's system calls and a portion of seL4's capability system.

Acknowledgements

I'd like to thank all the people who read through drafts of this paper and provided feedback, especially family members and readers from outside my field. I'm grateful to Mark Jones for sharing his knowledge of seL4 and capability-based security. Mark Jones put me on the track of the research around seL4 and helped me understand some of the inner workings of the seL4 kernel. I'd also like to thank my university, Portland State University, and especially the Honors college. They made the creation of this thesis possible. Most of all I'd like to thank my thesis advisor, Bart Massey, for guiding me through the thesis process and providing timely feedback on every one of my drafts.

Contents

List of Figures

List of Tables

1 Introduction

Computers are an ever-growing part of our daily lives. This not only includes desktop and laptop computers, but phones and smart devices. Smart homes and internet-of-things devices are becoming commonplace. We are increasingly relying on the cloud and the servers that back it.

The security of every one of these systems is paramount. We can improve and ensure the security of computing devices by improving the common foundation these systems share—that of the operating system.

1.1 Operating Systems and Kernels

An operating system is the system that controls all of the basic functionality of the computer. This includes managing hardware and giving a usable interface to the computer. An operating system kernel is the core part of the operating system. The kernel contains the most essential functionality of the operating system; it has the highest level of privilege, able to control all parts of the system.

Microkernels are a family of kernels that contain the smallest possible amount of code in the kernel, greatly increasing the security and reliability of the operating system. Limiting the amount of code limits the number of bugs present in the kernel: estimates put the number of bugs introduced at around 5 or 6 bugs per thousand lines of code [6]. In monolithic kernels large parts of the operating system run inside the kernel, giving processes like drivers kernel-level privileges. Counter to this approach, microkernels run large parts of the system with user-level privileges, limiting the effects of buggy or malicious drivers. To accomplish this, drivers and user processes communicate with each other by sending messages back and forth using inter-process communication (IPC). This causes microkernels to perform significantly more system calls than other kernels: each message passed requires a system call to send the message and another to receive it. As a result early microkernels were significantly slower than monolithic kernels [13].

A large amount of work has been devoted to making microkernels fast: L4 is a prime example of a microkernel that is fast in spite of its heavy use of IPC [11]. seL4 is a microkernel in the L4 family, inheriting L4's fast IPC but expanding on the security mechanisms in the kernel [8]. seL4 is primarily targeted at embedded systems, but supports a wide array of hardware including both 32-bit and 64-bit x86 architectures. The seL4 microkernel is notable for its formal verification and proof of correctness of the kernel [4], and that all system calls are guaranteed to complete within

a certain amount of time making seL4 a real-time kernel. The real-time properties of seL4 prevent it from being delayed by nonessential work when a pilot is making emergency maneuvers or when a driverless car is braking for a pedestrian.

1.2 Capabilities

One of the largest improvements seL4 has made is building capabilities into the kernel from the ground up. Capabilities are an access control mechanism, designed to restrict and control access to various resources such as memory and IO devices [1]. Capabilities allow processes to have the least privilege necessary to operate, limiting the amount of damage malicious or faulty components can cause. The principle of least privilege states that processes should only have the minimum system access necessary to operate, as any unnecessary access can let the process interfere with or compromise other parts of the system [21]. Capabilities are a powerful mechanism to limit access to system resources. Capabilities not only greatly enhance the security given by a microkernel, but allow for even more separation between processes and enable fine-grained resource allocation.

seL4 is a formally verified microkernel targeted at embedded devices [3]. Formal verification means that seL4 has been mathematically verified to follow a specification: seL4 has been proven to implement the specification without any bugs. This specification, which provides a mathematical model of the kernel, provides for the ability to prove additional properties of the kernel. Formal verification is especially important in the realm of security—seL4 is guaranteed to follow a specification so a formal security model can be reliably implemented on top of this specification. seL4 makes use of the take-grant security model [4]. Unverified kernels using this model do not get the same level of assurance. If there are defects in the implementation then the whole model falls apart.

1.3 Rust

Rust is a strongly-typed systems programming language that aims to achieve the dual goals of memory safety and speed [20]. This is accomplished through compile-time checks, meaning that all of the cost of this safety is paid before the software is even run. Rust is a programming language whose speed is similar to that of C or C++. Rust enforces memory-safety, which means there is no possible way to overflow a buffer, create a dangling pointer, or cause a memory leak. This is important because it protects

against large classes of bugs. Rust has a borrow checker that ensures that references, an abstraction over pointers, are well behaved and ensures that creating a dangling reference is impossible. In addition to this, the type system makes code even more reliable as many invalid programs that would compile in C would fail to pass the Rust type checker.

Memory-safety is a useful tool for writing correct code. However, sometimes you need to work directly with memory or dereference a raw pointer. This is especially important when interfacing with C code, as it lets you work with pointers returned from C. Rust has a mode for achieving this, called unsafe. When in an unsafe block of Rust code most of the same protection is available—the type checker will still make sure your program is well-typed, and the borrow checker will make sure you do not borrow any reference you wouldn't otherwise be able to. However, you are free to modify memory at will and to dereference and write to any pointer. This can lead to the same classes of memory bugs as C: as such unsafe code is discouraged. However, unsafe code still gets most of the protections of safe code and provides a useful mechanism for interacting directly with memory and for interfacing with foreign C code.

1.4 Related Operating Systems

There are a number of notable Rust and microkernel-based operating systems. Redox OS is a Unix-like operating system built entirely in Rust [19]. Redox is a microkernel-based operating system, but does not have capabilities and uses a different method of message passing than L4 kernels. Redox is very far along in its development—there is a fully usable graphical interface. The system in its current state demonstrates the feasibility of microkernel-based architectures.

Genode is a microkernel framework built in C++ [5]. Genode is capability-based, and primarily supports static configuration. This is especially useful on embedded devices, where the resources available to processes are static and unchanging. The capability-based architecture lets processes run with the least privilege possible, allowing for greater reliability and security. While Genode is not written in Rust, it does allow for the ability to write components of the operating system in Rust and shows the potential of capability-based security.

Phil Opperman's Blog OS is another notable Rust operating system, primarily aimed at teaching [16]. Blog OS solely targets the x86-64 architecture and is a monolithic kernel. Though not a full-fledged operating system, Blog OS shows that Rust is an effective programming language for introducing

and teaching people about operating system development.

1.5 Wayless

Microkernels and capability-based security are a promising way to increase the security of operating systems and computing in general. Though progress has been made in formal methods and formal verification, Rust gives some of the same benefits without most of the development costs associated with these methods.

Wayless is a capability-based microkernel, written in Rust. Through Wayless I hope to not only show that Rust is capable of developing useful operating systems, but that it provides significant benefits over C and is significantly less expensive in terms of time and effort as compared to formal verification. I hope to show that Rust is a good candidate for the development of secure operating systems.

2 Design

Wayless is a capability-based microkernel. This kernel targets desktop computers and servers on the x86-64 architecture, with the possibility of future ports to additional architectures. Wayless draws heavy inspiration from seL4, the formally verified microkernel that is pushing the bounds of security through the use of capabilities. Wayless has been created with the aim of binary compatibility with seL4.

2.1 Development Language

While Wayless does not make use of formal methods, the use of a strongly-typed language like Rust provides some of the same guarantees. In addition properties of the kernel can be checked with tools like `proptest`. While this method does not prove that a property of the kernel is guaranteed to hold, it gives reasonable assurance of properties and is much more powerful than unit testing. Rust also provides protection against large classes of errors. This includes buffer overflows and use-after-free bugs, major causes of bugs in software written in C. For example, Heartbleed [7], the famous bug in OpenSSL, resulted from a buffer over-read that would not have been possible in safe Rust code. While this does mean Rust is a better choice of programming language than C, formal verification gives exactly the same benefits and more. However, the costs of formal verification are high, at around $200-400/line of code [17]. Some parts of Wayless are written in

x86 assembly, but this was kept to a minimum as assembly has no safety guarantees.

2.2 ABI

The application binary interface to Wayless is designed to replicate the seL4 ABI. This allows for compatibility with seL4 and the existing tools and programs built for that kernel. The long-term goal is binary compatibility, the ability to run programs written for seL4, which will let Wayless take advantage of all of the existing drivers and libraries written for seL4. Compatibility will significantly shortening the time required to develop useful programs that run on the Wayless kernel, and will enable contributions to the greater seL4 ecosystem.

2.3 Targeted Platforms

While seL4 is targeted at embedded systems that need real-time operating systems, Wayless is targeted at x86-64 desktops and servers that often do not care whether the operating system is real-time. Servers are not used in the same kind of situations as embedded devices, and they typically will not require an instant response. The kernel can potentially gain speedups over seL4's average time because of this, though Wayless is not at the point where a useful benchmark can happen. Because there is only a single platform targeted there is the possibility of making platform-specific optimizations. The single target additionally makes programming and bootstrapping significantly easier: targeting multiple architectures makes bootstrapping more complex and potentially requires separate code for each architecture.

3 Implementation

Wayless supports much of the functionality of seL4. However, implementation differs markedly from that of seL4. There are features here that might not make it into the final version of Wayless as they are outside the scope of a microkernel. The major components of Wayless are covered in this section, as well as a brief overview of their implementation.

3.1 Development

Wayless was developed using the QEMU emulator [18]. This emulator shortens the development cycle of Wayless: it allows Wayless to be executed with-

out going through the process of transferring the kernel to physical hardware and physically booting up the system. Wayless takes advantage of QEMU's ability to connect the serial port to the standard input and standard output of a Linux terminal, letting Wayless easily show the user debugging information. Additionally, QEMU supports remote debugging, allowing for easy debugging from the development environment.

3.2 Threads

As Wayless targets the x86-64 architecture it supports the usual x86 virtual addressing mechanisms. Each thread is represented by a thread control block (TCB) that resides in the kernel, which includes an address space, capability tree, and various other information needed to run a thread. In the future threads will be able to manage this control object through a capability.

Threads are scheduled using a round-robin scheduler. They can optionally yield to allow the scheduling of another thread or, if the thread runs for long enough, it will be preempted by the scheduler.

3.3 Memory

Wayless needs to be able to manage physical memory to bootstrap the computer and to start a userspace thread. Physical memory is allocated using a buddy allocator [10] with stacks of 4KB pages, 64KB blocks, 1MB blocks, and 16MB blocks. Wayless is a Multiboot2 kernel [15], so on startup memory information is read from the Multiboot2 boot information and added to these stacks. Wayless's current allocator places a hard cap on the amount of physical memory available: up to 4GB of physical memory can be used.

In the future this allocator will be removed. Capabilities representing unused sections of memory will serve a similar purpose, letting threads use memory at will from these regions. This type of capability is called an Untyped capability as portions of it can be used and retyped to other kinds of capabilities. The control of allocation through capabilities will allow the application of a formal security model to memory allocation, ensuring that processes are only allowed to use memory granted to them [2]. Reworking the allocator will move it out of the kernel and into userspace, increasing security and giving processes a greater ability to control their own memory use.

3.4 Interrupts

Interrupts are given to provide useful debugging information when an exception occurs. Interrupts are static and threads have no control over them; in the future these will be configurable as part of the TCB through the associated capability. To handle bootstrapping there is an interrupt that will map a page under any faulting address. This allows for Wayless to easily bootstrap a userspace thread and to increase the kernel's memory space. However, this lets any thread use memory without restraint, and provides no way of freeing memory. Once Untyped capabilities are implemented this ability to use memory without limit can be removed.

3.5 Capabilities

Capabilities are tokens that represent and restrict access to kernel objects and hardware devices, as described by Dennis and Van Horn [1]. Kernel objects in Wayless have associated capabilites. If a userspace process possesses a capability to a kernel object it can call methods on the object, letting the process interact with and modify the object given that it has the capability. This enforces the principle of least privilege—processes are only provided with capabilities that are absolutely essential to their functionality. Every process, even drivers, can have access to only a small potion of the hardware or kernel objects in the system. For example, the audio driver only needs a capability to an audio device: it should not be allowed to access your graphics card or any other hardware on the system.

In addition to providing access to kernel objects and hardware, capabilities grant permission to perform message passing. This is an essential component of microkernels. Only a small portion of the operating system resides in the kernel: system services, such as drivers, run as individual processes. This minimality requires that there is some form of message passing between processes, to allow userspace processes to communicate with and use drivers and other services. To send a message to another process a message is sent to a capability that represents a messaging endpoint. This messaging endpoint is linked to another endpoint capability in another process. Messages can be sent and received between these endpoints.

Capabilities live inside of a directed graph. This structure is functionally identical to the capability space implemented in seL4 [4] and implements the same model of security, based off the classical take-grant model described by Lipton and Snyder [14]. In this take-grant model the directed graph represents what processes have access to what resources. If a path in the

graph can be traced from a process to a resource, then the process has permission to use that resource. Processes with the right capabilities can modify the graph, changing the permissions given to other processes.

This directed graph of capabilities works much like a directory tree in a filesystem, and will henceforth be referred to as a capability tree. This capability tree is a implemented as a guarded page table, as described by Liedtke [12]. Guards allow for the compact representation of large address spaces using only a small number of nodes. A guard represents empty entries in the page table. For example, if the only accessible entries in a guarded page table are addresses of the form `0x400XXXXX`, a guard of `0x400` can be used. With this guard accesses to empty areas of memory, like `0x50000000`, will immediately fail and will not cause additional lookups. In addition the effective address space is now only 20 bits instead of the original 32. This requires less backing page tables and thus less memory.

CNodes, capabilities that represent the nodes of the tree, can contain many children while all other types of capabilities are at leaves of the tree. Every thread has a root CNode, letting each process have its own addressable tree of capabilities. Each CNode has a *size*, specified when it is created. The *size* of the node determines the number of bits used to address its children, so thus the maximum number of children the CNode can contain is determined by 2^{size}. In addition to this *size*, CNodes possess a *guard*. As explained above, the guard allows for the creating of compact, efficient address spaces. The size of this *guard*, in bits, is specified by the *guard_size*.

3.6 Capability Addressing

Capabilities are accessed with capability addresses. A capability address is a 64-bit number, similar to a memory address. An address resolves to a specific capability, starting resolution from the root CNode of the calling thread. This is a recursive process: first, the initial *guard_size* bits of the address are checked against the guard of the root CNode. If the guard does not match then a LookupFailed error is returned. Otherwise, if the guard matches, then the next *size* bits of the address are used to select a child node. If this selected capability is anything but a CNode, or if we've resolved all bits of the address, the process ends. If the process has not ended it repeats over again. Eventually this recursive process will exit: a capability has been resolved.

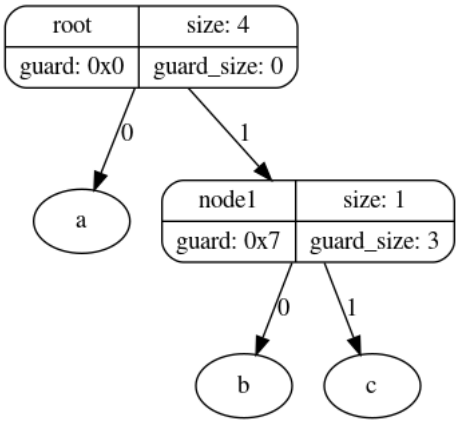

Figure 1: A possible capability tree

A possible capability tree is shown above. This tree has two CNodes, *root* and *node1*. In addition there are three other capabilities in this tree, *a*, *b*, and *c*. *root* has no guard, while *node1* has a 3-bit guard of `0x7`. Say that we wanted to access *a*. There's no guard on *root*. This means the first *size* bits of the address should be 0, *a*'s index in *root*. Because of this the first four bits of the address should be `0x0`, which means the address `0x0000000000000000` will resolve to *a*. However, because address resolution stops after the first 4 bits, any address of the form `0x0XXXXXXXXXXXXXXX` will resolve to *a*.

Now we want to access *c*. First, we need to get to *node1*. This means the first bits of the address should be *node1*'s index in *root*, meaning the first 4 bits should be `0x1`. Now, we want to access *c*. However, *node1* has a guard, requiring the next 3 bits of the address to be `0x7`. After passing the guard we want to access *c*, which is at index 1. Combining these two values gives us the 4-bit number `0xF`. Finally, combining the two 4-bit numbers we get `0x1F` and the address `0x1F00000000000000`, which will correctly resolve to *c*. Similarly, any address of the form `0x1FXXXXXXXXXXXXXX` will successfully resolve to *c*. To then address *b*, we just change the index of *node1* to 0, giving the address `0x1EXXXXXXXXXXXXXX`.

Description	Address	Examined Bits	Location
Resolution starts from the root CNode with the given address	`0x1F00000000000000`		*root*
The first 0 bits of the address are checked against *root*'s guard	`0x1F00000000000000`	None	*root*
The next 4 bits of the address are used to select a child of *root*	`0x1F00000000000000`	`0x1`	*root*
Index `0x1` of *root* was selected, moving us to *node1*	`0xF00000000000000`		*node1*
The first 3 bits of the address are checked against *node1*'s guard	`0xF00000000000000`	`0x7`	*node1*
The guard `0x7` matches, so the next bit is used to determine the next index to move to	`0x100000000000000`	`0x1`	*node1*
Index 0x1 was selected and we've reached *c*. Because we're at a capability resolution stops and the remaining bits of the address are discarded	`0x00000000000000`		*c*

Table 1: Resolving the address `0x1F00000000000000`

4 Results

Wayless is a functioning kernel with a handful of different types of capabilities. Though a long way off from supporting the full seL4 ABI, the initial progress is promising: large parts of the kernel have been implemented.

4.1 Performance

Wayless contains 1.5 thousand lines of Rust code, including tests, and 500 lines of assembly. Once compiled the kernel contains 92 kilobytes of code.

The complete binary is slightly larger, at a size of 208 kilobytes. Some initial optimizations have been performed to reduce the size of the kernel, but future work may be possible in this area.

A preliminary benchmark of Wayless system call performance was performed. This benchmark took place in the QEMU emulator without KVM acceleration enabled. In the benchmark the Send system call was called 1000000 times, repeatedly calling a dummy capability. This benchmark tests both the system call overhead as well as the overhead associated with resolving capabilities. The benchmark yielded an average system call time of $13\mu s$. It is not yet possible to do a ping-pong benchmark, a benchmark that tests the speed of inter-process communication, as message-passing capabilities are not yet implemented. Once these are a part of Wayless it should be possible to get an authoritative benchmark.

4.2 Rust

Rust proved to be well suited to the task of writing a microkernel. Rust's unsafe mode proved useful when working directly with memory and some of the many Rust libraries usable without the standard library were used to decrease development time. The usage of unsafe Rust was kept to a minimum. However, wherever inline assembly is used an unsafe block is required. There are additionally points where unsafe blocks are used to directly modify memory, which is required when setting up paging and virtual memory, and when reading in Multiboot2 boot information. Once this information is read in it is immediately converted to a safe datastructure, minimizing the usage of unsafe Rust.

The testing functionality in Rust was useful and helped catch quite a few bugs. Of course there were some trickier bugs: `gdb` proved useful for finding and debugging these as `gdb` has very good Rust support. Rust's built-in unit testing was used to test important parts of the kernel, and `proptest` was used to verify properties of kernel datastructures. Though this is not the same as formal verification, it both shows that properties of my kernel hold and tests critical paths of execution. Rust thus gave Wayless a large number of benefits over C, in testing and in safety, and carried none of the cost associated with formal methods.

4.3 seL4 Compatibility

The seL4 ABI had to be reverse engineered to make Wayless ABI compatible. This was done by reading the seL4 manual, reading the proofs and spec

associated with seL4, and finally by reading the C source code. This process went surprisingly well: though the seL4 documentation does not give a lot of information the combination of everything being both in C code and in Isabelle proofs made the process easier.

There has been no test of the Wayless ABI as it compares to seL4: there is no guarantee that the seL4 ABI was correctly reverse engineered. Once Wayless is more complete it will be possible to test seL4 programs on Wayless to verify that the ABI was correctly implemented.

No clean-room approach was taken when reverse-engineering seL4, and as such Wayless is licensed under the GPLv2 to ensure license compatibility.

5 Conclusion

Wayless is an initial implementation of a Rust microkernel. While only a subset of seL4's ABI was implemented, Wayless serves as a useful proof-of-concept Rust operating system. This subset of seL4's ABI did prove enough to be able to run userspace programs, and both the system calls that were implemented and the ability to address and call capabilities were tested. Rust proved to be a suitable choice of programming language. The memory safety and testing features included with Rust proved advantageous. Rust's ability to use unsafe code was required for the implementation of portions of the kernel, though this use of unsafe code was kept to a minimum.

It might be possible to formally verify Rust in the near future, as preliminary research has been done in proving the correctness of the standard library [9]. This would enable the formal verification of Wayless, ensuring that the kernel holds to the seL4 specification, and is a potential area of further development and research.

Continuing work on Wayless is planned with the eventual goal of feature-parity with seL4. It may even be possible to build a fully-featured operating system based on this kernel. I hope that Wayless shows not only that operating system development in Rust is feasible, but that, when it comes to the development of operating systems, Rust might even be superior to C.

References

[1] J. B. Dennis and E. C. Van Horn. "Programming Semantics for Multiprogrammed Computations". In: *Commun. ACM* 9.3 (Mar. 1966), pp. 143–155. ISSN: 0001-0782. DOI: 10.1145/365230.365252. URL: http://doi.acm.org/10.1145/365230.365252 (visited on 11/12/2018).

[2] D. Elkaduwe. "A principled approach to kernel memory management". In: (2010).

[3] D. Elkaduwe, P. Derrin, and K. Elphinstone. "Kernel Design for Isolation and Assurance of Physical Memory". In: *Proceedings of the 1st Workshop on Isolation and Integration in Embedded Systems*. IIES '08. New York, NY, USA: ACM, 2008, pp. 35–40. ISBN: 978-1-60558-126-2. DOI: 10.1145/1435458.1435465. URL: http://doi.acm.org/10.1145/1435458.1435465 (visited on 10/30/2018).

[4] D. Elkaduwe, G. Klein, and K. Elphinstone. "Verified protection model of the seL4 microkernel". In: *Working Conference on Verified Software: Theories, Tools, and Experiments*. Springer. 2008, pp. 99–114.

[5] *Genode - Genode Operating System Framework*. [Online; accessed 11. May 2019]. Mar. 2019. URL: https://genode.org.

[6] L. Hatton. "Reexamining the fault density component size connection". In: *IEEE software* 14.2 (1997), pp. 89–97.

[7] *Heartbleed - CVE-2014-0160*. [Online; accessed 12. May 2019]. May 2019. URL: https://nvd.nist.gov/vuln/detail/CVE-2014-0160.

[8] G. Heiser and K. Elphinstone. "L4 Microkernels: The Lessons from 20 Years of Research and Deployment". In: *ACM Transactions on Computer Systems* 34.1 (Apr. 6, 2016), pp. 1–29. ISSN: 07342071. DOI: 10.1145/2893177. URL: http://dl.acm.org/citation.cfm?doid=2912578.2893177 (visited on 11/01/2018).

[9] R. Jung et al. "RustBelt: Securing the foundations of the Rust programming language". In: *Proceedings of the ACM on Programming Languages* 2.POPL (2017), p. 66.

[10] K. C. Knowlton. "A Fast Storage Allocator". In: *Commun. ACM* 8.10 (Oct. 1965), pp. 623–624. ISSN: 0001-0782. DOI: 10.1145/365628.365655. URL: http://doi.acm.org/10.1145/365628.365655.

[11] J. Liedtke. “On Micro-kernel Construction”. In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP ’95. New York, NY, USA: ACM, 1995, pp. 237–250. ISBN: 978-0-89791-715-5. DOI: 10.1145/224056.224075. URL: http://doi.acm.org/10.1145/224056.224075 (visited on 11/13/2018).

[12] J. Liedtke. “Address space sparsity and fine granularity”. In: *ACM SIGOPS Operating Systems Review* 29.1 (1995), pp. 87–90.

[13] J. Liedtke. “Toward real microkernels”. In: *Communications of the ACM* 39.9 (1996), pp. 70–78.

[14] R. J. Lipton and L. Snyder. *A linear time algorithm for deciding subject security*. Tech. rep. YALE UNIV NEW HAVEN CT DEPT OF COMPUTER SCIENCE, 1976.

[15] *Multiboot2 Specification version 2.0*. [Online; accessed 20. May 2019]. May 2019. URL: https://www.gnu.org/software/grub/manual/multiboot2/multiboot.html.

[16] P. Oppermann. *Writing an OS in Rust*. [Online; accessed 11. May 2019]. May 2019. URL: https://os.phil-opp.com.

[17] D. Potts et al. “Mathematically verified software kernels: Raising the bar for high assurance implementations”. In: (2014).

[18] *QEMU*. [Online; accessed 23. May 2019]. May 2019. URL: https://www.qemu.org.

[19] *Redox - Your Next(Gen) OS - Redox - Your Next(Gen) OS*. [Online; accessed 11. May 2019]. Mar. 2019. URL: https://www.redox-os.org.

[20] *Rust programming language*. [Online; accessed 16. May 2019]. May 2019. URL: https://www.rust-lang.org.

[21] J. H. Saltzer and M. D. Schroeder. “The protection of information in computer systems”. In: *Proceedings of the IEEE* 63.9 (1975), pp. 1278–1308.

Appendix A
Wayless Developer's Manual

1 Building and Running Wayless

Wayless can be built with the Make build system. The source code is available on Gitlab at `https://gitlab.com/waylon531/wayless/`, under the GPLv2 license. The default build target, called by running `make`, will compile the kernel and the included dependencies. This builds a copy of seL4 to generate bindings and to get the right system call numbers, ensuring that Wayless provides the same interface as seL4.

There are a few extra provided targets. `make run` will compile Wayless, then start it up in QEMU. Similarly, `make with-kvm` will start Wayless in QEMU but with kvm enabled.

There are both `make clean` and `make distclean` targets. `clean` cleans up the bindings, object files, and the final Wayless binary, while `distclean` cleans up everything including the downloaded copy of seL4. Finally, there is a `make test` target that will run the tests provided with Wayless.

1.1 Dependencies

There are some necessary programs and libraries required to build and run Wayless. These are listed here.

- `xargo`
- `grub`
- `mtools`
- `xorriso`
- `qemu` (may be called `qemu-system` on your machine)
- `nasm`
- `clang`
- `ninja` (`ninja-build` on ubuntu and debian)
- `cmake`

There are also a handful of python libraries required to compile seL4. Wayless depends upon header files generated by seL4 to ensure binary compatibility.

- `setuptools`
- `sel4-deps`

1.2 Bootstrapping

Wayless is able to bootstrap and run a simple user program. This happens through a macro that includes `userspace/start`, which gets compiled from `userspace/start.asm`. You can create your own userspace program by modifying the Makefile in `userspace/Makefile` to compile the program of your choice: this compiled userspace space program must be located at `userspace/start`.

This program is given an initial capability space. This space consists of a root CNode with two slots, and an IOPort capability in the first slot at index 0. This IOPort capability is for all IO ports on the system, and can be used to print and read from the serial port. This space, though not enough to write featured programs, is enough to print to the screen and to show that capabilities and system calls are working.

2 Structures

2.1 msgInfo

msgInfo is a single 64-bit number, subdivided into 4 sections. The first section, starting from the least significant bit and consisting of the first 7 bits of the number, represents the length of the message. The length is measured in 64-bit words, the initial words get placed into machine registers and the rest will go into the IPC buffer. The next 2 bits represent the number of capabilities sent alongside the message, while the following 3 bits represent which capabilities were unwrapped by the receiver. Finally, the last 52 bits of the number are a label. The label is used to specify what method to call on capabilities, but is unmodified by the kernel and can be used for any purpose when doing IPC. The correspondence between label numbers and capability methods is described in Section 4.

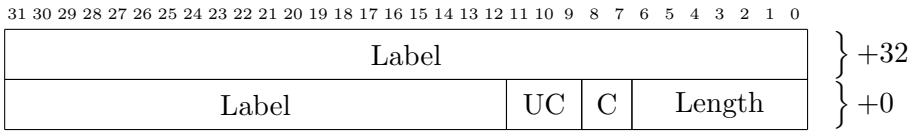

Figure 2: msgInfo

3 System Calls

Wayless has the same system calls as seL4. These are documented here. To call a system call make sure to save `rsp`, as well as `rcx` and `r11`, as these registers will be clobbered during the process. The system call number should go in `rdx`, while `rdi` is the pointer to the capability you want to operate on and `rsi` is the msgInfo. There are also four message registers, `r10`, `r8`, `r9`, and `r15`. All system calls happen using the `syscall` instruction, as only 64-bit mode is supported.

3.1 Send

Send has a systemcall number of -3, and is used to send a message to a capability. The first words of the message are passed in the four message registers, while the rest of the message has to be passed in an IPC buffer. This system call will block until the message is received.

3.2 NBSend

NBSend has a systemcall number of -8. NBSend is a nonblocking version of send.

3.3 Recv

Recv has a systemcall number of -5, and will get a message from an endpoint. The message will be returned into the message registers and IPC buffer of the current thread, the sender's badge will be returned in `rdi`, and the received msgInfo will be returned in `rsi`. This is a blocking call and will wait until a message is received.

3.4 NBRecv

NBRecv has a systemcall number of -8. NBRecv works the same as receive but will not block.

3.5 Call

Call has a systemcall number of -1. Call is used to make calls to capabilities, being functionally identical to a method call. This is the same as doing a Send and then a Recv. In seL4 Call will additionally give the caller a Reply capability, which can be used to send a reply to the callee, but this has not been implemented yet in Wayless. Similar to Recv the response will be passed as an IPC message, filling up first the message registers and then the IPC buffer.

3.6 Reply

Reply has a systemcall number of -6. Reply is unimplemented, but once implemented will be used to send messages via Reply capabilities.

3.7 ReplyRecv

ReplyRecv has a systemcall number of -2. ReplyRecv does a Reply and then a Recv. This is unimplemented.

3.8 Yield

Yield has a systemcall number of -7. Yield pauses the current thread, letting the next process start. It takes no arguments.

4 Capabilities

Methods on capabilities can be called by using the Call or Send system calls. Using Call will let you get values back from the method, while Send will throw away any return values. Arguments to these methods are passed as an IPC message; the first four arguments are passed in the message registers, `r10`, `r8`, `r9`, and `r15`, and any following arguments are passed in the IPC buffer. Methods return values in the same way, in message registers or in the IPC buffer if the result is too long.

4.1 IOPort

IOPort capabilities are used for sending and receiving data over system IO ports. This is used in the provided userspace program to print to the serial terminal.

4.1.1 X86IOPortOut8

X86IOPortOut8 takes an io port as the first argument, and an 8-bit byte to output to the serial port as the second argument. The label for this method is 46.

4.1.2 X86IOPortOut16

X86IOPortOut16 takes an io port as the first argument, and a 16-bit word to output to the serial port as the second argument. The label for this method is 47.

4.1.3 X86IOPortOut32

X86IOPortOut32 takes an io port as the first argument, and a 32-bit double word to output to the serial port as the second argument. The label for this method is 48.

4.1.4 X86IOPortIn8

X86IOPortIn8 takes an io port as the first argument. This will read a 8-bit byte from the specified IO port, which will be returned in the first message register. The label for this method is 43.

4.1.5 X86IOPortIn16

X86IOPortIn16 takes an io port as the first argument. This will read a 16-bit word from the specified IO port, which will be returned in the first message register. The label for this method is 44.

4.1.6 X86IOPortIn32

X86IOPortIn32 takes an io port as the first argument. This will read a 32-bit double word from the specified IO port, which will be returned in the first message register. The label for this method is 45.

4.2 CNode

CNodes hold other nodes and work like nodes in the capability tree. Currently no methods are supported on CNodes.