

BACHELOR INFORMATICA



UNIVERSITY OF AMSTERDAM

Technologies and Basic Blocks for a Modern Educational Op- erating System

David Julian Veenstra

August 30, 2016

Supervisor(s): dr. R.C. Poss

Signed: Signees

Abstract

Operating System concepts should be learned by doing. Yet the world of computer science has changed. Multi-core processors are now commonplace, and there are new system languages available. This creates an opportunity for a new educational kernel for OS courses. This thesis searches for the appropriate technologies and building blocks. We find that a kernel written in Rust that targets the Xen hypervisor is desirable. A message passing kernel with little to no shared state is proposed. Furthermore, implementation and assignment guidelines are presented.

Contents

1	Introduction	7
2	Background and objectives	9
2.1	Related Work	9
2.2	Educational Design Objectives	11
3	Technical Choices	13
3.1	Language Choice	13
3.2	Virtualization Technology	20
4	Design	23
4.1	Interprocess Communication	24
4.2	Memory management and protection	24
4.3	Scheduling	25
5	Implementation Guidelines	27
5.1	Minimal kernel	27
5.2	Paging and Heap allocation	28
5.3	Event channels	30
5.4	Console and xenstore driver	30
5.5	Block device driver	30
5.6	Single Core Scheduling and Message passing	31
5.7	Multicore Scheduling and Message Passing	32
6	Assignment guidelines	33
6.1	Hello World and Frame Allocator	33
6.2	Paging and Heap allocator	33
6.3	Event Channels	34
6.4	Console driver	34
6.5	Block device driver	34
6.6	Single vcpu kernel	34
6.7	Multi vcpu kernel	34
6.8	Available example code	34
7	Conclusion and final remarks	35
7.1	Epilogue on the choice to use Rust	35
7.2	Future work	36

Introduction

The purpose of this project is to find and describe the technology and the basic blocks that are needed to develop an educational OS that is able to illustrate the different aspects of a modern OS.

Operating System concepts are something that should be learned by doing, not by reading or listening [2]. For example, the Minix OS was developed [19] with this philosophy. The OS was stripped of non-essential features, to highlight different aspects of the design of an OS. However, Minix was developed in the 80's. The field of Computer Science has made numerous advances since then.

Firstly, Moore's law seems to be at an end. In recent years, the increase of computational power of a processor has primarily been increased by adding more cores, and by increasing the efficiency of the use of multiple cores. These aspects were not an integral part of the design of Minix.

Secondly, the internet has connected the computers across the world. This made computers a target of malicious users. The experience gained resulted in many advances in the field of computer security.

Thirdly, the programming languages of today offer more safety and more tools to abstract. Historically, C has been *the* system programming language. This is still the case today, but modern languages offer desirable features that C does not have. For example, Haskell and Rust have an advanced type system, which is able to catch many programming errors at compile time. This greatly reduces the occurrence of common vulnerabilities such as buffer overflows. Moreover, the type-system is even able to encode properties and invariants of a problem in the type system itself.

Modern operating systems that are used today, do have mature support for multi-core processors. However, their core design was also developed before the advances mentioned before. And as such these advances were not a fundamental part of the design.

Modern virtualization also offer benefits for an educational OS. Writing a device driver is a time consuming endeavor. Technologies like Xen and QEMU-KVM offer simplified devices, that generalize over the peculiarities of real hardware. Moreover, virtualization makes it easier to restart the kernel, has extra debugging features, and hides more complexity from the developer. For these reasons, a virtualization platform will be targeted.

The research question of this project will be: *What are the appropriate technologies and building block to implement a modern OS?* This question will be answered by researching the following sub-questions:

- What language is suitable for a secure and modern OS?
- What virtualization technologies are suitable for an educational OS?
- What is an appropriate design for a modern education OS
- How can this design be implemented?

This thesis is structured as follows. Chapter 2 highlights previous work in the area. Chapter 3 analyses the different technology choices. In section 3.1 it is analyzed which desirable properties an OS implementation language should have. Subsequently, a rating system is proposed based on these discovered properties. Consequently, 5 different languages are compared, and rated, leading to the choice of Rust as implementation language. In section 3.2 Xen and QEMU-KVM are compared on features, and a virtualization technology is chosen. In Chapter 4 the disadvantages of current OS designs are listed, and an alternate design is proposed. Subsequently in Chapter 5, implementation guidelines are presented. Lastly, in Chapter 6 assignment guidelines are presented. Chapter 7 concludes and gives a brief evaluation of Rust.

Background and objectives

2.1 Related Work

One of the first educational OS, and one of the most well known is the Minix OS [19]. This OS was developed in the late 80's. The main motivations for its creation was that conventional OSs were too large and complex for students to understand in a relative short time. In addition, Unix was not freely available. Subsequently, Minix 2 and Minix 3 were developed. The latter built upon Minix 2 and expanded its goals to also be a OS suitable for research [12]. In particular, it focuses on reliability and self-healing of faults. Minix uses a microkernel design. This reduces the number of lines of code to 4000. This aided the simplicity of the kernel, and reduces the occurrence of bugs. Support for multicore architectures was only added in recent years, arguably as an afterthought.

Nachos [7] is an OS that was build for similar reason as the first version of Minix, and developed roughly in the early 90's. In addition, they found that educational kernels available at that time, did not reflect the changed landscape of OS and architecture design.

But an educational was still desirable, as they believed that OS concepts should be taught by experimentation, and examination.

Specifically, they missed emphasis on threading and distributed programming, but also did not illustrate the new cost-performance trade-off between CPU speed, memory and secondary storage.

The result of the Nachos project was a working kernel of roughly 2500 lines of code, half of which is documentation. This kernel had to be run on their deterministic MIPS emulator, which was designed to ease developed by determinism. The components of the kernel had the most simple implementation the authors could conceive of. It was up to the students to implement something better, which would be tested against micro-benchmarks.

Nachos wants to cover a number of OS topics. The most important being:

1. concurrency and synchronization,
2. caching and locality,
3. trade of between simplicity and performance,
4. building reliable components from unreliable components,
5. dynamic scheduling,
6. distributed computing.

The Pintos [17] OS was developed at Stanford University, and was meant to replace Nachos. It was different in two important ways. First, it could be run on both real hardware and in simulated and emulated environments. The authors believed that its ability to run on actual hardware increased student engagement. Second, they designed analytical tools that can detect memory corruption and race conditions, and they provided test suite to aid development.

Pintos at a glance:

- developed to be a replacement of Nachos;
- small enough so that entire code can be read by students;
- 100 percent reproducible;
- it can be run directly on the hardware, for them an important reason as they perceive that to be more engaging;
- for debugging purposes it can run in an emulator such as QEMU;
- explicit choice to switch from C++ to C;
- killer features is analytical tools that is able to detect memory corruption; and race conditions
- simplicity is again an important descition.

The cornerstones of their design:

- Read before you code (skill to read code, and provide examples on the expected coding style)
- Maximize creative freedom
- Practice Test-driver Development
- Work in a Team
- Justify your design
- Provide a Reproducible, Manageable environment
- Include Analysis tools
- Provide extensive and structured Documentation

The authors mention that multicore support might be added in the future, but it is unclear whether this is the case.

In similar vein to Pintos, GeekOs [14] is a kernel that is designed to run on real hardware. But that can also be executed in emulated environment to ease development. The authors mention three reasons why it is desirable to have a bare metal kernel. First, it gives the instructor control of the complexity that is exposed to the student. The second reason is realism. Third, it gives students hand-on experience with the actual architecture. They surveyed the students of the first class that used the GeekOs. 18 out of 25 students found it interesting that it could be run on real hardware.

GeekOS provides:

- microkernel;
- kernel threads with a simple handler;
- interrupts handlers;
- allocation of physical memory pages;
- allocation of memory pages from a kernel heap;
- device drivers;
- no support for multicore.

Another interesting educational kernel is the TOS kernel [17]. Their main motivation is that real kernels are too complicated, and in their view distributed programming is an important aspect that should be highlighted. To this end, a microkernel is implemented in Java. Like the other kernels they mention that minimalism and simplicity is an important design goal.

Finally, the xv6 kernel [8] is worth mentioning. This kernel was developed fairly recently (2006) at MIT. The lectures were based on the V6 unix kernel, which ran on the PDP-11. The assignment made use of the JOS kernel, that ran on x86. It was hard to justify teaching a kernel that was more than 30 years old, and only ran on outdated hardware. In addition, students struggled to understand both the x86 and PDP-11 architecture. For these reasons they developed the xv6 kernel which is a modern version of the V6 kernel with support for multicores.

In summary, all the educational kernels that have been examined mention simplicity and minimalist kernel as one of its design goals.

2.1.1 Unikernels

In this subsection a number of unikernels (kernel that runs a single application such as a http server) will be mentioned.

As explained later in this thesis, we opt to run on either QEMU-KVM or Xen hypervisor. For QEMU-KVM an extensive protocol manual is available. Nonetheless, example implementations are useful for a better understanding of the working of the virtio devices (virtual devices defined by the hypervisor to provide access to external I/O devices). The documentation of Xen is fairly scarce. Information about Xen is scattered over blogs, source documentation, the Xen website, and Xen book. Xen does have an example implementation of a minimal kernel. However, the readability of the code leaves much to be desired. A few examples can thus help better understand what can be reached with KVM and Xen.

Solo5¹ is a kernel written in C by IBM. It is minimal and the code is of high quality. It has excellent examples to initialize and use the virtio devices. Similarly, includeos also uses virtio devices, but is written in C++. It has an example of an OOP based api built around the virtio devices.

For Xen Examples the HalVM unikernel² is very useful. The HalVM itself has an example of start up code. It also has high level Haskell examples on how to use the different facilities that Xen provides. The runtime of their customized GHC (Haskell Compiler) has example code on event channels, xenstore, memory management for pv (para-virtualisation) mode, and examples on startup other cores in pv mode. They also provide a Fedora repository to install Xen compiled with debugging mode. The ling erlang kernel, also has high quality examples on the different Xen facilities.

Finally, there is also the rump kernel³. The rumprun repository contains mini-os code that has been cleaned up.

2.2 Educational Design Objectives

This project has a number of design objectives. First, the design of the kernel should have a focus on simplicity above performance. Large parts of the target audience are first year computer science students. These students still have difficulty with programming, and have not been trained in the use of concurrent and parallel programming. Furthermore, in a course of Operating Systems it is the high-level ideas that are important, complex details only detract from this. Second, the kernel should be small so that it is possible for the students to understand the kernel in its entirety, and it also supports the first objective. Third, the more the students can implement themselves the better. If the students can implement most of the kernel themselves they would likely feel like it is their kernel. This increases their engagement, and it also makes the kernel easier to understand. And finally the kernel should be able to illustrate the challenges within the field of OS design. Especially, it should highlight the challenges of scaling the OS across cores.

¹<https://developer.ibm.com/open/solo5-unikernel/>

²<https://github.com/GaloisInc/HaLVM/>

³<http://rumpkernel.org/>

Technical Choices

3.1 Language Choice

One important aspect of an educational kernel is the implementation language. This decision is important as it affects the difficulty of implementation, and the ability to focus on concepts from the OS.

This section is structured as follows. First, an overview is given of the language choice of other projects. Second, the important aspects will be distilled and supplemented by other aspects that were not mentioned. Subsequently, the relative importance of each aspect will be decided. After that a number of languages will be evaluated based on the weighting. Finally, the evaluations are summarized and a language will be chosen.

In the past, educational OSes have been written in numerous languages such as C++, Java and C. Especially, C has been popular as an implementation language. However, many projects do not specify the reason for choosing C. One of the exceptions is Pintos [17]. The primary reason they give is that C is the predominant language within the field of system and kernel programming. Practice and C debugging experience is one of the educational goals of the project. Secondly, they want to show that low-level operations are not incompatible with high-level tools. Here at the UvA the Operating System course is also taught in C, for the primary reason stated above. It is suspected that this is also the primary reason that other projects have chosen C as the language of implementation.

Other projects that have chosen another language do substantiate their decision. For example, the first Version of Nachos was implemented in a subset of C++ [7]. They made this decision because Object Oriented Programming was popular at that time, and it provided them with a natural idiom to stress the importance of modularity in both the OS design and implementation.

In later years Nachos was also ported to Java at Berkeley [13]. Many undergraduate courses offered at that time at Berkeley were given in Java. Hence, Java was something that students were familiar with. In addition, Java is a type-safe language with less dangers of undefined behavior. The latter is mentioned as a major problem of the old Nachos, as students struggled with bugs that had little to do with concepts from operating systems. There were also a few practical reasons that were relevant at that time. For example, the compiled bytecode was smaller, which reduced the amount of data that had to be transferred to the network disks.

TOS is another OS that was implemented in Java [15]. The authors argued that Java provided a stable base to write robust programs. Moreover, the fact that Java compiles to bytecode, makes it easier to port it to other architectures. Having a working bytecode interpreter, also provides a base implementation for threads, exceptions, and automatic garbage collection. The latter is mentioned as preventing some of the most frequent and expensive bugs in C or C++, due to the misuse of pointers and pointer arithmetic.

These were the arguments in favor of a language that were in the literature.

For TOS and the Java Version of Nachos, type and memory safety is an important reason to choose something other than C. This prevents bugs that are difficult to debug at compile time, which leaves more time to focus on OS concepts. From a security standpoint the choice of C

is also difficult to justify. Due to the unsafety of C, a slight programming error could result in a vulnerability such a buffer overflow. Studies have shown that every 1000 lines of code, on average, contains 6-16 bugs [12]. Thus, from both a security and educational perspective a type and memory safe language is preferable.

There are other educational aspects that are relevant. Firstly, languages vary in their difficulty and learning-curve. Implementing an OS is not a trivial tasks. It cannot be expected from the students to spend much time learning a new programming language or style. Therefore, a language that the student can pick up easily, is preferred.

Another relevant aspect is that the language itself might be worthwhile to learn. As mentioned before, gaining experience in in C is worthwhile on its own. Similarly, a language that students might use in the future is preferred. Nonetheless, even if students will likely never use the language again, it can still be instructive as it might offer an interesting programming model that is not covered elsewhere in the curriculum. These are two aspects that should also be taken into consideration.

Finally, the last educational aspect is the productivity of the language. It is desirable to have a productive language, as it leaves more time to highlight the concepts, makes it easier to implement and understand the kernel. However, the productivity of a language is fairly subjective. The languages that are considered will be selected based on the availability on modern construct that fit the language program style, such as OOP, iterators, higher-order functions, abstract data types, generics, etc.

Besides educational aspects, there are also technical aspects that should be taken into consideration. Higher-level programming languages require a run-time library (called simply “run-time”), and for current languages often relies heavily on the services that are provided by a separate OS. Since it is desirable to have a small kernel, it is desirable to use a runtime that requires few services.

Below we estimate the size of the runtime by compiling and executing two simple programs. The first program is a program that immediately returns 1. The second program is a simple program that sums the number one to ten, using a dynamic data structure. By counting the number of needed system calls, an estimate is made of the amount of services the runtime requires. In addition, the size of the compiled program is measured.

Secondly, some of these services might have to be implemented in another, possibly less safe, language. In the name of simplicity, it is preferred to only require one language. Furthermore, the benefits of a safe language diminish, if large parts of the OS have to be implemented in a less safe language.

In summary, these are the aspects of a language that are relevant here. here is a list of the arguments in favor for a given language.

- the language might be used by the student in the future;
- the language offer a programming model that is instructional;
- the language has a slow learning-curve;
- the language is type-safe;
- the language is memory-safe;
- the language is productive and does not detract from concept from OS;
- the runtime is small;
- only a small part of the kernel cannot be written in the language of choice.

Now that the relevant aspects are known, a weighing has be to decided. Reviewing properties that the operating should have, aspects that makes the OS easy to understand should have a higher weight. For the same reason, aspects that allows more focus on OS concepts, instead of things like obscure bugs, should be rated high. However, type and memory safety have both strong education and technical arguments, therefore these will have the strongest weight.

In the following subsection, a rough estimate of the real-world usability is presented. In the subsequent sections an overview will be given of the languages, and a rating will be given for every aspect.

Aspect	Weight
real-world usability	1
instructional programming model	1
steepness of learning curve	2
type safety	3
memory safety	3
size of runtime	2
need for other language	2
Total	14

Table 3.1: Weighing of the different aspects

3.1.1 Real World Usability

An estimate for the “Real World Usability” is the popularity of a language. The more popular it is, the more professionals use it. There are is little research data available on the popularity of languages, but there are some groups who calculate a popularity ranking. These ranking are a very rough estimate, but it should at least serve as an indication of the popularity.

Table 3.2 displays the language popularity for TIOBE [11], Redmonk [16], and TLPI [9]. All give the absolute ranking, with the exception of Redmonk, which gives a rating from 0 to 100, with 100 being the most popular. The numerical values given below for Redmonk were estimated from a picture in the original publication. In addition, the work of Bissyandé et al [4], and the PYPL [5] rating have also been consulted, but only one of the language appeared (Haskell with a ranking of 10-20). Based on this data, the ordering seems to roughly be: $nim < OCaml < Rust < D < Haskell$

Language	TIOBE	Redmonk	TLPI
Rust	47	60	67
Nim	-	10	-
D	20	60	21
OCaml	50+	60	81
Haskell	39	80	25

Table 3.2: Popularity ranking per language and per index

3.1.2 Haskell

Haskell is a lazy and purely functional programming language. It has a type system that is based on the Hindley-Milner type system. It has many abstractions and language extensions to encode properties and invariants in the type system. Haskell also offers type inference, higher order functions, type classes, abstract data types, pattern matching, interpreter, and more.

Students usually have already seen the functional programming style early in their curriculum. However, Haskell offers relatively uncommon features that are interesting. In particular, type classes offer many interesting abstraction such as monads, monoids, and many different flavors of functors. In addition, type classes provide a decent solution for generic programming, and as such is interesting to study.

The Haskell language has existed for a while, but has never come to prominence. It is still a fairly niche language that is popular among academics and the functional programmers. Therefore, it is not likely that a student will use Haskell again after their studies.

Yet, by using Haskell some of these abstractions will be needed immediately, as the OS needs to do many I/O operations. One of these is the I/O monad, as the OS performs many I/O operations. Hence, the student will have to work with monads from the start. It takes considerable amount of time to understand this abstraction. Getting familiar enough with this

abstraction and the basics of Haskell to be productive will takes weeks, not days. Moreover, it is likely that more advanced abstractions will be needed. For these reasons the learning curve will be rated to be steep.

The final aspect is the size of the runtime. The documentation of GHC describes the Haskell runtime as a beast consisting out of more than 50,000 lines of code. Figure 3.3 shows the metrics of the two small programs. Even these small programs require many system calls, and produces large binaries. In addition, Haskell is a garbage collected language. Before Haskell can be run on bare metal, the garbage collectors and other facilities, such as lightweight threads, locks, memory management, will have to be provided.

	Simple	Sum
total number of system calls	133	133
total different types of system calls	23	23
size of executable	1128	1144

Table 3.3: Metrics of Haskell Programs

As an example, the HalVM, a haskell unikernel library, can be inspected. The HalVM-GHC runtime requires roughly 4000 lines of C code to support their OS based on the Xen hypervisor. Hence, it is a daunting task to support the Haskell runtime. To summarize, the size of the runtime is very large. Table 3.4 gives a summary of the rating for the suitable of Haskell.

Aspect	Rating
real-world usability	2
instructional programming model	4
learning-curve	1
type safety	5
memory safety	4
size of runtime	1
need for other language	1
Weighted Total	2.79

Table 3.4: Rating of Haskell

3.1.3 OCaml

OCaml is another functional programming language with a strong typing system, in spirit of ML. It offers many constructs that are often found in functional programming languages, such as higher-order functions, pattern matching, abstract data types and type inference.

OCaml uses a strict type system that is also based on the Hindley-Milner type system. However, it does not have support for type classes. Compared to Haskell, the type system is a bit less strict, as it allows side-effects, without special abstraction such as the monads. This eases the learning process, and allows students to always use the familiar print statements to debug their code.

Similar to Haskell, OCaml is a garbage collected language. But unlike Haskell, there are some options that reduce the difficulty of running OCaml without an OS. The first option is the *libc-ocaml* library that implements a minimal runtime, with only a few dependencies. It needs a function to halt the processor, and a function to enlarge the heap. The *ocaml-freestanding* project is another option. This project is designed to make it easy to compile OCaml against a unikernel. Table 3.5 corroborate that OCaml has a small runtime.

While the above mentioned options make it easier to get OCaml running, it does require a large part of memory management and allocation to be written in another language. And this is not a trivial part.

The programming model that OCaml offers is not new to the students. Nonetheless, first year students still have difficulties with the functional programming model. Hence, if OCaml does not offer something substantially different, it is still worth while to use.

	Simple	Sum
total number of system calls	107	108
total different types of system calls	19	20
size of executable	20	20

Table 3.5: Metrics of OCaml Programs

Aspect	Rating
real-world usability	1
instructional programming model	3
learning-curve	3
type safety	4
memory safety	4
size of runtime	5
need for other language	1
Weighted Total	3.29

Table 3.6: Rating of OCaml

3.1.4 D

D is an imperative language, that originated as a re-engineering of C++. It's design goal is to be expressive as dynamic languages such as Python, and Ruby, while still offering the safety and speed of a compiled languages. It offers features such as inline assembly, higher-order functions, OOP, garbage collection, templates, generics (with templates), and more.

Memory safety is mainly provided through the use of garbage collection. There is also a subset of the language that guarantees memory safety by disabling some features, such as cast that circumvent the type system, and modification of pointer values. The garbage collection can be disabled, but this also disables all memory safety features.

Programs written in D can also be compiled without support for garbage collection. This does make it feasible to write the garbage collection in D. But this would have to be written without any memory safety.

As far as the type system goes, it is not as strict or powerful as the one found in Haskell, OCaml, or Rust. Templates do provide plenty of compile-time power. However, running code at compile time is not as robust as a strong type checker. It is more likely that heavy use of templates will only make it more difficult for the students.

The language has a syntax and semantics that are very similar to that of C, C++ and Java. In essence, it is a modern interpretation of an imperative language with templates. Most students are comfortable with one or more of the languages mentioned above, and hence should feel comfortable with D. However, this does mean that D does not offer a programming model that is substantially different.

3.1.5 Nim

Nim is also an imperative language. It has similar design goals as D, but has a different philosophy. It offers iterators, higher order functions, compile time evaluation of functions, variants, generics, a REPL (read-eval-print loop), compilation to C++, an official C to Nim converter and more.

	Simple	Sum
total number of system calls	101	101
total different types of system calls	22	22
size of executable	540	544

Table 3.7: Metrics of D Programs

Aspect	Rating
real-world usability	2
instructional programming model	1
learning-curve	5
type safety	3
memory safety	3
size of runtime	3
need for other language	5
Weighted Total	3.36

Table 3.8: Rating of D

Memory safety is provided by the garbage collector. Other than that no, memory safety features are provided. To support GC, the runtime has to be modified to support the new kernel.

Nim can be compiled without GC. Hence, it should be possible to write the code that is needed for the GC in Nim. But this is again a non-trivial part of the kernel that would have to be written in memory unsafe variant of Nim.

Nim's syntax is similar to that of python, and the language itself is similar to a pythonesque C++. Hence, it provides a known programming model, but with a different flavor. Furthermore, the language is still very unknown, With only few packages, and a few resources that are available. From all the languages that are considered, this is the language that the students are the least likely to need in the future.

	Simple	Sum
total number of system calls	40	44
total different types of system calls	12	14
size of executable	164	164

Table 3.9: Metrics of nim Programs

Aspect	Rating
real-world usability	1
instructional programming model	1
learning-curve	4
type safety	3
memory safety	3
size of runtime	4
need for other language	5
Weighted Total	3.29

Table 3.10: Rating of Nim

3.1.6 Rust

Rust is more difficult to categorize. At first glance, it looks like an imperative language in similar fashion as C. However, many ideas have been borrowed from other styles of programming. Instead of using templates for generics, it uses something that is similar to type classes. Moreover, just like OCaml and Haskell, it uses a type system that is based on Hindley-Milner. The type system is augmented with phantom types and associated types to facilitate encoding of properties into the type system. Other worthwhile features include type inference, iterations, pattern matching, inline assembly.

One of the biggest features of Rust is memory safety at compile time. In the literature the technique used is called affine types; Rust calls it the ownership system. Rust's type system places a number of restriction on the use of pointers, such as no null pointers, no pointer aliasing to mutable data, and this way the compiler is able to guarantee memory safety even without garbage collection. The compiler is not smart enough to recognize all code that is memory safe, and in some cases there is no other option but to violate these restrictions. For such conditions, a block of code can be allowed to be unsafe, but it has to be explicitly marked so in the source code.

While Rust is usable without heap allocation, it is convenient and sometimes necessary to use heap allocation. The ownership system also helps here, as it knows when the heap allocated memory is no longer needed, and insert the necessary de-allocations in the compiled code. Adding support for heap allocation is fairly easy. Rust has special language constructs to enable or exchange the heap allocator. This allocator can also be written in Rust, and upon activation the collection crate (Rust's term for a library) can be used. This provides an implementation of useful data structures like linked list, heap, hash table, dynamic strings and more.

The combination of the different features borrowed from other languages, and the ownership system provides an unique, but also challenging programming language. The type system forces the student to think about the type information that the compiler needs, how to encode this into the type system, and it also forces the student to think about the lifetime of data.

Without all the advanced features that Rust has to offer, it is just like any other imperative language like C. Despite the challenging programming model, the students can be eased into the language, and features can be introduced along the way. Hence, the learning curve is not steep but gradual.

	Simple	Sum
total number of system calls	89	90
total different types of system calls	19	21
size of executable	616	632

Table 3.11: Metrics of rust Programs

Aspect	Rating
real-world usability	1
instructional programming model	5
learning-curve	3
type safety	4
memory safety	5
size of runtime	3
need for other language	5
Weighted Total	3.93

Table 3.12: Rating of Rust

3.1.7 Final Language Choice

Table 3.13 gives a summary of the weighted rating of the languages that were considered. Rust scored well in all aspects, and scored considerably better than other languages. It offers an interesting programming model, it has strong type and memory safety, and it does not have a steep learning curve. From all the languages that have been considered, Rust scored significantly higher, hence Rust is chosen here as the implementation language.

Language	Weighted Rating
Haskell	2.79
OCaml	3.29
D	3.36
Nim	3.29
Rust	3.93

Table 3.13: Summary of ratings

3.2 Virtualization Technology

In the introduction it was argued that having an educational OS that is able to run on actual hardware is desirable because it is more engaging to the students.

An OS that runs on specialized emulated platforms does have its advantages, as it often has special debugging features, restarting the kernel take very little time, and it has simplified components which make the implementation easier.

Here, an intermediate solution will be used. The kernel will be developed for a virtualization platform. This has similar benefits as an emulated platform, including the simplified I/O devices, which makes it feasible to design drivers that the students are also able to implement themselves.

Despite the virtualization it is still close to the bare metal, as it runs directly on the hardware, with only parts of the hardware interface provided by a hypervisor. Moreover, the current popularity of “infrastructure as a service” in Cloud providers has made running an OS on a virtualized platform commonplace. Two virtualization technologies will be considered here, namely QEMU-KVM and Xen.

QEMU uses full virtualization. This allows one to pick and choose the desired architecture and hardware components. Linux provides some kernel models called KVM, and this allows QEMU to use hardware extensions for more efficient use of virtual address translation and protection, and efficient system calls. The extensions are called Intel VT-X, and AMD-V, for Intel and AMD processors respectively. Not all processors have these extensions, but all new and recent models do. Hence, this should not be a problem in the future.

Besides access to the virtualization extensions, KVM also provides virtio [1] devices. Each virtio device is a simplified version of a class of devices, such as a block device, and removes any of the hardware peculiarities that a real device might have. Whenever a request is made to the device, the request is transferred to the hypervisor kernel, which uses the real drivers to process the requests. All of the virtio devices have a similar interface, and are present to the guest kernel as a normal device. For example, under x86 the virtio devices are presented as a PCI devices. It is also worth mentioning that it is relatively easy to attach a patched gdb (to support mode switch to 64-bit long mode) to the kernel.

Xen uses a hypervisor that directly runs on the hardware [3]. Recent versions of Xen only support the x86-64 and ARM architectures. There are a number of features that are available to the kernel developer. Guests in Xen can run in different modes, and the features that are available depend on the virtualization mode.

To use different features of Xen, a mechanisms very similar to system calls is provided. This mechanisms is called hypercalls. Whenever a hypercall is executed, the system traps to the Xen hypervisor, which then processes the calls. Due to the context switch to the hypervisor, it is

expensive to use the hypercalls. In each mode different part of the hardware are not directly accessible and have to be configured with hypercalls.

There are many convenient features provided through the hypercalls. Not all of the features are available on all modes. To give a few examples: stack switching, configuration of clock interrupts, bringing up/down a processor core, printing to special debugging console, and more.

The x86-64 architecture uses the Advanced Programmable Interrupt Controller (APIC) to configure interrupts (such as device interrupts), and route them to the appropriate processor. To notify other process of an interrupt, Interprocess Interrupts (IPI) can be send. IPI's also make use of the APIC. To facilitate the programming of these facilities, Xen offers a higher-level mechanisms called event channels. Through hypercalls the different interrupts can be routed to the appropriate core; the IPI can also be configured and used through the event channels.

Xen also offers simplified devices [6]. Similar to virtio devices, the drivers exists out of two parts, one part that exists on the host OS and another part on the guest OS that has to drive the simplified devices. Xen's high level constructs have to be used to implement the driver. Within Xen the Xenstore, which is somewhat similar to a key-value store, is used to communicate with other domains (virtual machines). This is also used to configure the devices. When available, the event channels are used for interrupts. The data between the two parts of the drivers is shared through a shared ring buffer in main memory.

All of the high level constructs that are available in Xen are not immediately available upon booting the kernel. The facilities such as the Xenstore, ringbuffer, and event channel do need an API built around the hypercalls that are needed to configure them. Thus Xen does need more work upfront, but eases the amount of work that has to be done later.

As mentioned before, the guests can be started in a number of different modes. The first mode that Xen offers is the pv mode, which uses para-virtualization. All of the hardware components have to be accessed and configured with hypercalls. The same is true for the Memory Management unit (MMU). This makes memory management considerably more difficult. It also adds another layer of indirection in the address translation. Virtual addresses translate to pseudo-physical addresses, and these translate to machine addresses which are the actual physical addresses. To the kernel the pseudo physical addresses appear linear and continuous, but are not due the extra layer of indirection. Most hypercalls operate on machine address.

On the x86-64 architecture, the processor starts in 16-bit real mode with paging disabled, due to legacy reasons. There is quite a lot of assembly code needed to jump to 64-bit long mode with paging enabling. To the majority of the students this code will look arcane and be hard to understand. It is not worth to spend much time on code that exists mostly for legacy reasons.

The pv mode hides this complexity, and starts a guest in 64-bit long mode with paging enabled. In addition, the guests starts with the Interrupt Description Table (IDT) containing interrupts handlers that log plenty of useful debugging information. The performance of this mode is good, as it runs mostly on the hardware with no interference. Modification of the page table, and system call are the exceptions, as these trap to the hypervisor, and hence are expensive operations.

There are three additional different modes that uses QEMU-KVM to emulate actual hardware components instead of providing abstract interfaces like the modes above. All of these additional modes emulate motherboard components and start in 16-bit real-mode. And all of the modes use virtualization hardware extensions [21]. The extensions gives the guests direct access to its own page tables, and are able to do efficient system calls without having to trap to the hypervisor. The three modes are called: hvm, hvm + pv drivers, pvhvm. Going from the left to right, more components are para-virtualized. Hvm is the least performing of the three, and the pvhvm the highest performing mode.

Hvm mode is full virtualization with almost none of the Xen features. The hvm + pvdrivers mode is very similar to QEMU-KVM with virtio devices. The simplified Xen devices are accessible as pci devices, but event channels are not available. The pvhvm mode para-virtualize almost every component, with the exception of the motherboard. This mode is able to use event channels.

Finally, there is the pvh mode, that was added in 2010. This mode combines the best of both types of modes [18]. Similar to the pv mode, the pvh mode starts in 64-bit long mode with paging enabled. Unlike the pv mode, it does have to configure the IDT and GDT as one would

with a bare metal kernel. And similar to the hvm modes it uses the virtualization hardware extensions for efficient system calls and direct access to the page tables. Furthermore, this mode also uses event channels, and is able to start up other processor cores in protected 64-bit mode with page enabled. This mode is the easiest to develop for, and hides much of the complexity. It might appear that this is the highest performing mode, but it is also the youngest mode. The Xen documentation does not yet claim that it is the fastest mode, but recent benchmark are in favor of pvh mode¹.

In summary, QEMU-KVM provided a fully virtualized system, with simplified devices. Xen's most suitable mode, the hvm mode, hides complexity with higher level constructs, such as the event channels, and the hypercalls that are able to start processor cores up. In addition, it also hides more complexity by starting the kernel in 64-bit long mode with paging enabled. Thus Xen offers more and hides more complexity. This increases the amount of code that the students can implement themselves, and reduces the difficulty. For these reasons, it is chosen to use Xen's pvh mode as the target virtualization platform.

¹http://events.linuxfoundation.org/sites/events/files/slides/PVH_Oracle_Slides_LinuxCon_final_v2_0.pdf

Design

A multicore kernel needs many components: memory management, protection, device drivers, initialization of the processors, scheduling and a means of interprocess communication. The device drivers do not belong to the essential components that are needed to run a kernel, but from an educational perspective these components are essential to cover topics such as filesystems, device drivers and command shells.

Besides the goal of simplicity and minimalism, scalability is also desired. There are three big problems with the conventional story of multicore support, whereby kernel and user code run on the same processors.

The first problem is memory locality. Since both kernel and user code are running on the same core, both will have to share the cache. Memory usage of one core might push out cache entries of the other. Secondly, every line of the kernel could be executed. This means that after every context switch a wildly different system call might be executed. This could result in intra-kernel cache trashing [20].

Another problem is that a kernel commonly uses shared memory to maintain global state. Whenever the shared state is written, the cache coherency protocol has to update all relevant caches. This can be quite costly as the latency increases linearly with the number of cores. At some point adding more cores will be detrimental to the performance [10].

Moreover, the interconnect becomes more complex as the number of cores on the architecture increases. As a result, the cache coherency protocol becomes more complex as well, and as a consequence more expensive [10].

The third problem is that the shared memory has to be locked before it can be used. To support many cores the lock must be fine grained [10]. Programming the necessary many locks is difficult and error prone. And is certainly too difficult for a first year student. Secondly, if a finer granularity is needed it might not be enough to add more locks; the entire algorithm/-datastructure might need to be redesigned.

To solve these problems the available vcpus (virtualized processors) will be partitioned into two classes: normal and privileged. The normal vcpus will run user programs, while the privileged vcpus will run the kernel code. It might appear inefficient to reserve vcpus for the kernel. But the majority of the modern processors, such as x86-64 architecture, support a form of hardware multi-threading. With multi-threading a part of the circuits of a processor core is duplicated to run multiple threads at the same time. Each of these threads is presented to the OS as a normal processor (called vcpu henceforth). Even if the privileged vcpu might not have work to do, other vcpus can still utilize the same physical core.

Now the question is how the partition the kernel over the available privileged vcpus. There are multiple approaches. The first is to split the components of the kernel into independent parts that run on different privileged vcpu. Every part of the component is responsible for a portion of the state. Then the global state of the component is negotiated with IPC.

Another possible approach is to split the kernel into independent components. Each component is then allocated to run on a given processor. Here there is no need for a protocol to synchronize the state, which simplifies the implementation of the components. In this thesis, we

choose for independent components, as it is the simplest of the two solutions.

4.1 Interprocess Communication

The services that are provided by the OS will be divided over the different vcpu. Hence, if one vcpu needs a particular service, it might need to communicate with another vcpu. At the least a rudimentary communication mechanism is needed. In traditional kernels shared memory is often used as the important mechanism for communication. But as mentioned before, shared memory doesn't scale well with number of cores, it is error prone, and makes it difficult to get performance right.

Instead, a rudimentary message passing protocol will be proposed. Sadly, the x86-64 architecture does not expose facilities to optimize message passing to programmers, even though the lower level interconnect does use message passing to implement Inter-Processor Interrupts (IPIs) and its cache coherency protocol. Thus, shared memory will be used in this first implementation, although other mechanisms can be explained to students. The scaling problems of shared memory will be avoided by only sharing memory between two vcpu.

There are some desirable properties that the message protocol must have. First it must be as simple as possible. Secondly, since protection on shared memory has the granularity of a whole page, the usage of shared pages must not explode as the number of cores increases. Third, a process should not be able to interfere with the contents of its own or others message. Forth, other processes should be able to run, when a process is waiting for a message. And lastly, the use of synchronization mechanism should be use as least as possible, as it complicated the implementation and synchronization mechanism can be difficult to implement.

To achieve these properties, a blocking message passing protocol is proposed. It uses immutable and fixed sized messages that fit into a cache line.

Between every two vcpu a "connection" is constructed. For each of these "connection" a configurable number of shared pages will be reserved to send from the first vcpu to the second, and another number of shared pages will be reserved for the second vcpu to send to the first. Every page is divided in a number of cache line slots. By limiting the number of shared pages per "connection", the number of shared pages is limited.

In addition every process has a "mailbox", existing out of a shared page, to send and receive messages. Since the protocol is blocking it only needs to contain space for one message and one response.

The scheduler will also be involved with the protocol. Whenever a vcpu sends a message, the vcpu switches the context to the scheduler. If there is an unused slot in one of the pages that is shared with the destination, the message is copied to the slot, and the process is put in the waiting queue. When there is no slot available, the message is copied to a queue, and the process is moved to the waiting queue. Since the kernel runs in kernel mode, the user space process cannot compromise the message.

On the receiving side, it becomes apparent that a message is waiting, when the scheduler is run, and the receiving slots are inspected for messages. It then copies the message to the awaiting process, and moves the process from the waiting queue to the ready queue. Now, no component needs the copy of the message that is found in the slot. And the receiving vcpu can clear the slot, so that it can be reused, without any need for locking.

Due to the blocking nature of the message passing protocol, it will appear to the user as any other function call. This would also make it safe to pass a reference to a data structure for it to be read/written by the receiving process.

4.2 Memory management and protection

Most modern kernels use virtual addressing. The memory is accessed by virtual addresses. A special data structure contains information to translate this address to a physical address. In addition, the special data structure contains access permissions.

The x86-64 architecture uses paging. Its memory is divided into chunks of 4KiB. These chunks are called pages. The translation and protection information is contained in a special data structure that is called the page table.

This is a tree-like structure that functions similar to a hashtable. Without any special options, the tree has a depth of 4. The root node, or the page map level 4 map (PML4), has 512 children. Every internal node also has 512 children. The internal nodes also have information if any of the children is present and other options. The leaf nodes, called the page tables, have 512 entries, and contain access permissions of pages, and the physical address of a page.

The page table structure is stored in memory and can be modified. Whenever a read or write occurs, the hardware traverses the page table to find the associated entry. If the action is permitted the action is performed on the physical address that is found.

From a security perspective, it is important to isolate the memory of each process. In most modern kernels, memory isolation is achieved by given each process its own page table. The virtual address space of each process has a similar layout, with similar virtual addresses, but it maps to a different physical addresses. The processes that run in user space are not able to directly modify its own or other page table, and hence memory isolation is achieved.

Here another approach will be taken. Shared memory is still an important aspect of the kernel. It is needed for the message passing mechanism and for inter-vcpu communication. For this reason, it is chosen to share the same address space between all processes. Every process no longer has its own page table. Instead, every vcpu will have its own page table to prevent the need for locks on the page table.

On the x86-64 architectures there is 256 TiB of address space available. Even if 16 GiB of address space is reserved for every process, a maximum of 16K processes could be created. For our purposes this is more than enough.

Ideally, the page table entries would also contain the ID of the process that owns the page, to prevent other processes from accessing. This feature is not available in the x86-64 architecture. For this reason, another per process data structure per process is needed, to describe which pages the processes owns.

Taking the above and implementation simplicity in mind, the following scheme will be used. Every process is allocated a part of the address space that is at least as large as the amount of memory available. And is aligned to and round up to 2 MiB boundaries. The processor page ownership can now be described by 2MiB increments, instead of 4 KiB. Setting/clearing the present bit in the internal nodes found at depth 3 (called the page directory), marks 2 MiB of address space as present/not present.

The memory usage of a process is usually in the order of a few MiB. Hence, on a context switch only a few entries in page directory has to be set/cleared. Luckily, the TLB on x86-64, which is a cache for the address translation, do contain a processor ID. Hence, no entries have to be flushed on a context switch.

4.3 Scheduling

The scheduler is another important part of the kernel. There are at least three things that have to be scheduled. The newly made user processes have to be scheduled on some vcpu. The kernel components have to be scheduled among the privileged vcpu. And eventually load balancing is needed for the normal and privileged vcpu.

Ideally, it would be possible for the privileged vcpu to change the state (registers and etc.) of another vcpu. Sadly, this is not possible on the x86-64 architecture. This complicates the story considerably. The scheduler for the normal vcpu cannot run on the privileged vcpu, it has to run on the same normal vcpu, and every vcpu will need a scheduler. The privileged vcpu can no longer freely edit the Process Control Block (PCB), as the scheduler will also need to modify that information. To prevent any contention, the OS services, such as sleep, that need to modify the PCB will have to be delegated to the scheduler.

Since the scheduler has to provide some of the services, the scheduler also needs to be able to receive and send messages. It would be a problem if the scheduler would block with message passing. But the scheduler has access to the message queue and “connections” and is able to

send and receive without blocking.

The scheduling itself can be fairly straightforward. The scheduler needs to maintain a ready queue for the processes that are runnable, and a waiting queue for process that are waiting for an event. Scheduling can be done with a simple Round Robin scheme. Room could be left to add some form of priority scheduling at a later stage.

Implementation Guidelines

Table 5 displays the steps that have to be taken to implement the kernel on the Xen hypervisor.

This chapter will highlight each step and list some of the difficulties that might arise. The first ten steps are based on first hand experience, the other steps are based on accumulated experience and knowledge gained during the project. Hence, the last steps will be less substantial.

5.1 Minimal kernel

The first step is to have a bootable 64-bit kernel that starts in hvm-mode. This includes configuring the Rust compiler, configuring Xen, installing a rudimentary IDT and some assembly code for event channels.

Before Rust can be used in a free standing environment, some additional steps must be taken. At the time of this writing, only Rust’s “nightly” compiler has the features enabled that are needed, such as a “no stdlib” mode and allocator options. Moreover, some features such as stack unwinding have to disabled as this needs OS services to function.

For Rust in a freestanding environment, there is the core crate. This library provides the most elementary types, such as arrays, integers and booleans, and other intrinsic and primitive building blocks. The crate does have some linker dependencies. First, it is assumed that the external functions *memset*, *memcpy*, *memcmp* are present. These can be installed with the *rlibc* Rust crate. There is also the issue that freestanding compilation results in undefined references to floating point functions, even when no FP operation are used. The Rust community is aware of this problem and is currently developing a solution. For now, the easiest solution is to add placeholder functions.

There are some features that have to be disabled. The first is stack unwinding on a panic. Even when disabled, it will still results in undefined references to *_Unwind_Resume*. Another placeholder functions has to added accordingly. The second feature that has to be disabled is the usage of the “red zone.” The red zone is an area reserved on the stack that serve as scratchpad. However when a interrupts arrives, it might write into the red zone, and as as result the stack would get corrupted. Another thing that has to be disabled, are the floating point registers. Besides for floating point operation, Rust also uses these registers for optimizations. Hence, an interrupts might corrupts the state of the previous function.

Now that Rust is configured, Xen has to be configured. The kernel can configure Xen by adding notes in the ELF binary. The following features are required for pvh mode

- *writable_descriptor_tables*
- *auto_translated_physmap*
- *supervisor_mode_kernel*
- *hvm_callback_vector*

Action description	Student	Instructor
1 configuring Rust 1.2 configuring xen		x
1.3 event channel setup		x
1.4 porting of hypercalls and Xen headers		x
1.4 rudimentary IDT		x
2 Hello World	x	
3 frame allocator	x	
4 Paging	x	
4.1 rudimentary heap	x	x
5 event channels	x	x
6 console device driver	x	
7 xenstore		x
8 Grant table		x
8.2 Shared Ring		x
8.3 Block Device	x	
9 Moving kernel to PCB	x	
9.2 scheduling for multiple processes	x	
10 Basic message passing on 1 vcpu	x	
11 Starting other vpcu's	x	
11.2 Multi core scheduling and message passing	x	

Table 5.1: Step by step implementation plan. The middle column indicate if the student has to implement a part of it. The rightmost column indicates that a part should be provided by the supervisor.

In short, these features enable native system calls, page tables and the IDT. The rest of the configuration can be based on the HalVM boot code. This includes the assembly that handles nested virtual interrupts, which is most likely the most difficult part of the event channels. From the host OS it also needs to be specified to boot the kernel in hvm-mode. This can be done by adding `\pvh=1` to the guest configuration file. Now it should be able to compile and boot a minimal kernel.

The next step would be porting the hypercalls to Rust. With `pvh-mode`, there is a special hyper page. By calling this page with a given offset, hypercalls can be executed. The porting is quite laboursome, and it is easy to make a mistake. There are some translation tools, such as *corrode* or *crust*, that can be used to reduce the work. But this does not always function properly, or gives an idiomatic translation, as these tools are a work in progress. With this completed, a hello world kernel can be made with the `HYPERVISOR_console.io` hypercall.

Since `pvh` has a native IDT, some rudimentary interrupt handling has to be added. Particularly, the page fault handler is important. A simple interrupt handler has to be installed to display useful debugging information, and is sorely needed before paging can be implemented. It should be noted that the interrupts handlers cannot be normal functions. When an interrupts occurs, some exception specific information is pushed on the stack, subsequently the stack frame of the handler is created. Thus, the return address is not present before the stack frame, and any return instructions will jump to a random address. Hence, a divergent function (a rust function that never returns) has to be used. There is also the problem that the rust function prologue pushes data on the stack, which makes it more difficult to access the exception information on the stack. Thankfully, Rust also offers so called “naked functions,” which do not have a prologue or epilogue. A naked function can be used to retrieve the stack pointer and pass this to a normal rust function.

5.2 Paging and Heap allocation

Before paging can be implemented a simple frame allocator is needed. A bitmap can be used to hold the information about the free/used frames. To speed up the retrieval of unused frames, a stack can be used to hold recently freed frames. This step does not need to be programmed in

freestanding Rust, and is fairly easy to program.

Now that there is a frame allocator, print functionality, and a debugging page fault handler, paging can be implemented. Here the strong type system of Rust can be of use. Listing 5.1 gives an example. There are four types created to indicate the current depth in the tree. These types use up zero memory, and are called zero sized types (ZST). These ZST, in conjunction with Phantom Data, can be used to annotate, at the type level, the depth of the page table. Now a trait can be created to encode the fact that it is only possible to traverse down the tree. This is the nextable trait. The function nextTable takes as argument a table and a memory address, and returns the next table. To encode that we can traverse from L1 to L2, associated types are used. This can be seen at line 18 and 27. The oddly looking 'a indicates that nextTable returns a table that has the same memory lifetime as self. The mut indicate that the tables can be mutated.

Listing 5.1: Example of Rust type system

```
1 use ::core::marker::PhantomData;
2
3 pub struct Level4; pub struct Level3;
4 pub struct Level2; pub struct Level1;
5
6 pub trait TableDepth {}
7 impl TableDepth for Level4 {}
8 impl TableDepth for Level3 {}
9 impl TableDepth for Level2 {}
10 impl TableDepth for Level1 {}
11
12 // repr makes sure that PageTable has the same
13 // memory layout as a c array with 512 integers.
14 #[repr(C)]
15 pub struct PageTable<T> {
16     entries: [u64; 512],
17     level: T
18 }
19
20 trait Nextable {
21     type NextTable;
22     fn next_table<'a>(&'a mut self, addr: u64)
23         -> &'a mut Self::NextTable;
24 }
25
26 pub type L4 = PageTable<Level4>; pub type L3 = PageTable<Level3>;
27 pub type L2 = PageTable<Level2>; pub type L1 = PageTable<Level1>;
28
29 impl Nextable for L1 {
30     type NextTable = L2;
31     fn next_table<'a>(&'a mut self, addr: u64)
32         -> &'a mut L2 { ... }
33 }
```

There is still the problem that the page tables cannot be read/written if they are not mapped. The easiest solution is to have some pages reserved to temporary map the page tables.

With a paging API, the heap can be set up. A heap allows the collection crate to be imported, which contains many useful data structures. It also simplifies the implementation of the xenstore and grant table. Setting up the allocator, is a bit cumbersome. It has to be in its own crate for it to work. At this stage, it is enough to have a simple allocator that is able to allocate but not free memory. A more advanced allocator can be implemented at a later stage.

5.3 Event channels

Next up, the event channels have to be implemented. The most confusing part should already be included in the boot code. The event channel itself works with two bitmaps and a bit mask per vpcu. The smaller bitmap indexes into the larger bitmap, and indicates which words of the other bitmap has to be inspected for pending interrupts. The second bitmap specify which channels have pending interrupts. The bit mask has the same size as the second bitmap, and determine if the interrupts are blocked on a given channel. These bitmaps are updated by the hypervisor concurrently. Hence, atomic operations are needed to read and write the data. This should be the first time that students needs synchronization mechanisms.

Around the event channels mechanism an API has to be built. First a virtual interrupts handler (one per vpcu) has to be registered through a hypercall. This function should determine which channel has a pending interrupt, clear the associated bits, and dispatch it to the appropriate handler. Furthermore, there are functions needed to manage the channels and to bind a handler with a given channel. The difficulty lies mostly with the details of the working of the event channels. It should not be difficult for student to implement, if there is a good explanation about the event channels and atomic operations.

5.4 Console and xenstore driver

With an API for the event channels implemented, the console and xenstore driver can now be written. The console is able to read from they keyboard and display text in a console. The xenstore, as mentioned before, is similar to a key value store, and is needed at a later stage for the block device driver. Both use two simple, lockless ring buffer to share data with the hypervisor. One to send data and one to receive data. Each ring has two heads, one for the producer and one for the consumer. Intelligent use of unsigned underflow and overflow is used to prevent having to wrap the heads with modulo operations. Rust standard integer arithmetic panics when underflow or overflow occurs. Hence, when the rings are ported to Rust, `wrapping_sub` and `wrapping_add` should be used instead.

When text is typed into the console, a virtual interrupt is generated. The associated event handler must look into the ring buffer to retrieve the data. The process to retrieve the data is similar. When the consumer header is updated, a memory barrier must be used for the back end to see the update.

Writing to the console is very similar. The only difference is that the console should be notified with an event, to process some requests until the ringbuffer has enough space for new requests. A full memory barrier is also needed to ensure that rust sees the updates to the ring. After the data is written to the ring, the console have to be notified of new request.

The xenstore is very similar. The basic procedures to send a raw request or response are almost identical to that of the console. The functionality of the xenstore can be build with these basic procedures. The responses from the xenstore are strings of unknown size. Dynamic string are here useful to make the api more user-friendly.

5.5 Block device driver

The next element that is needed for the block device, is the grant table. This table is used to share pages with other domains, such as the host domain, which holds the second part of the device drivers. To allocate the grant table a hyper call is need. This call returns an array of frames, that contain the grant table. Subsequently, the frames have to mapped into the address space.

Each entry of the grant table contain information about permission flags, and the machine frame that is to be shared. When the flags are not set to invalid, the grant become active. Hence, the permission flags should be the last field to be set. Implementing the grant table is not difficult. But it is very Xen specified, and hence not very educational.

The final element that is needed to implement the block device is the shared ring. Porting these rings is considerable more difficult, than porting the rings that were used for the xenstore

or console. This ring has many more operations, requests and responses are stored on the same ring, and there is another layer of indirection. This extra indirection is to increase performance by adding ability to batch multiple changes to the ring. Due to the increased complexity it is difficult to get the memory barriers right. Which results in code that is hard to debug.

To initialize the disk, the following steps have to be taken:

1. initialize the shared ring,
2. use the grant table to share the ring with the host,
3. use the grant table to share buffers with the host,
4. allocate a channel to signal/be signaled by the device,
5. use xenstore to negotiate the features,
6. bind an handler on the allocated channel,
7. allocate structures to maintain information that connects requests with responses.

The disk devices are operated as follows. A request, containing a request ID, operation specific options, and references to the shared buffers, is pushed on the shared ring. Subsequently, the disk device is notified of new work. Any information that is needed upon completion must be stored separately, such as the shared buffers that will be used to store the data read from disk. When the device has processed the request, the kernel will be notified that a new response was pushed on the ring. The response contains the request ID and a boolean indicating success. Based on the request ID, and the information that was stored the request can be completed.

Most of the difficulty lies within the shared ring and getting the memory barriers right. Given an user-friendly implementation of the ring and xenstore, it should be possible for the students to implement the driver. But it will not be the easiest exercise.

5.6 Single Core Scheduling and Message passing

The very first step to creating a scheduler is to design the process control block (PCB), and to continue the kernel with an attached PCB. The PCB at the least need space to hold:

- an processor id,
- information about owned pages,
- location of stack and heap,
- location of text and data,
- space to store the processor state,
- status of process.

Subsequently, a hardware timer has to be configured. With a special event channel hypercall, a channel for the timer can be allocated. Now an handler can be bound to the timer. This will be the entry point of the scheduler.

The scheduler should have a queue to hold the runnable processes, and a queue to hold waiting processes. Another queue is needed to hold messages that have to be send. It is possible that an interrupt occurs during the scheduler. This might happen when a message is being added to the message queue. This becomes a problem when the interrupt handler also has to send a message. A possible solution is to temporarily disable interrupts when accessing the message queue. Another solution is to use separate lockless ring buffer for the messages of the interrupts.

When the scheduler is activated, it should add messages to queue, if any. Move the current process to end of the waiting queue, and send any deliverable messages to the correct process, and add them to the ready queue. Other features that are necessary: an idle process to run if there is no other process, functionality to create a process and add it to the ready queue.

Most of the scheduler will use high level constructs, and thus be fairly easy to implement. There is some discipline required with the messages. The payload should be first written, and then it should be marked as present/active. A compiler barrier should be used to be sure that compiler doesn't reorder the writes.

5.7 Multicore Scheduling and Message Passing

Before other vcpus can be started, a PCB has to be created for the scheduler, the page table has to be copied, and the shared pages for the message passing have to be created. As was mentioned before, Xen does have hypercalls to start another vcpu. The begin state of the processor has to be passed as an argument, and subsequently the vcpu is started in long-mode with paging enabled. The existence of the hypercall does reduce the complexity considerably. However, the begin state has a lot of fields. The students should be able to start other vcpus, but there needs to be a thorough explanation of all the fields.

Next up is the message passing. The scheduler should now also inspect the shared pages for message, and write any messages to the correct shared page. The problem is that the scheduler does not know to which vcpu the messages have to be send. Since, there is no load balancer, all processes will always run on the same vcpu as where they started. The process ID could be augmented with the identifier of the vcpu where they started.

Assignment guidelines

This chapter presents some assignment guidelines. In each section one possible assignment is discussed, It argues what part should be given to the student and what part the student can implement themselves.

Comparing the number of sections and steps in Table 5, it can be seen that some steps are excluded. The xenstore is one of them. For the students it is interesting to implement the console driver, as it is component through which a lot interaction happens. The xenstore, on the other hand, will only be used for feature negotiation with the block device. Furthermore, the implementation of the xenstore is very similar to the console driver. Consequently, the implementation holds little educational value, and hence should be provided to the student. It is also recommended to supply the students with an implementation of the grant table. This table is very specific to Xen, and has little payoff for the students. Another element that should be provided to the student is the shared ring used in the block device. As mentioned before, porting the ring is fairly complicated, and the memory barriers are hard to get right. Furthermore, the memory barriers also makes it hard to test the ring within the kernel.

There is also the question what should be given to the student at the start of the project. It is desirable to give the student a stable framework. This makes the work of the student-assistants easier, as it makes it more predictable what problems the students might face. This framework should include things that are not interesting, or that have potential to introduce obscure bugs. Xen and Rust configuration both fall in this category. Rust translation of the needed Xen structs and hypercalls should also be included, as these are also error prone, and time consuming to translate. In addition, a rudimentary IDT should also be provided. A page fault handler is needed early on for debugging. Lastly, there is assembly code needed to handle nest virtual interrupts. This should also be provided.

6.1 Hello World and Frame Allocator

The first assignment is fairly light. Print hello world from the kernel, and make a Frame Allocator. Installing Xen can be a bit difficult, and hence, a lighter exercise is needed. The Frame Allocator can also be programmed in normal Rust as long as nothing out of the standard library is used.

6.2 Paging and Heap allocator

Paging is good opportunity to get more familiar with the type system that Rust has to offer. Understanding how the page table works and translating it to Rust should be a fair challenge for the students. For the heap allocator, some boilerplate for the allocator crate could be provided. This saves the students some trial and error. Heap allocation enables a lot of interesting data structures, and thus has a high payoff for the students.

6.3 Event Channels

The most confusing part of the event channels is already provided by the framework. But there should be a good explanation about what that code does, and why. Understanding how to prevent interrupts from corrupting the processor state is an important aspect of interrupt handling code. The implementation itself should not be that difficult, given that a good explanation is given about the working of the event channels and the atomic bit operations. There are a number of instructions that are useful to operate on bitmaps. Such as *btsq* instruction. This instruction takes as input a memory address and a bit offset, and will set the bit at the given offset starting from the address given. In Rust these instructions can be used with inline assembly. However, Rust uses LLVM inline assembly, although care must be taken since LLVM supports less assembly instructions than the underlying hardware. These bit operations could be provided as a precompiled object, or examples could be given that use LLVM inline assembly.

6.4 Console driver

With a good understanding of memory barriers, it should be fairly easy to implement. Moreover, there are only a few ways that it can be implemented. The assignment on its own is probably too short. This assignment could be joined with the event handling assignment, or added along side an assignment to improve the heap allocator or interrupt handler.

6.5 Block device driver

Here the students will use the provided xenstore, grant table, and shared ring. This should reduce the complexity considerably. But there are still a lot of steps that have to be taken. Students do have a lot of freedom on how they associate a response with a request that was made in the past. As a bonus, students could implement a simple file system that stores the data in a linked-list like structure.

6.6 Single vcpu kernel

This is a more challenging exercise as both the scheduling and the message passing have to be implemented. As mentioned before, it is important that the payload is written first, before the message is marked as present/active. Otherwise it is possible that a corrupted message will be sent.

6.7 Multi vcpu kernel

Starting the other vcpu's and scheduler is the last part. Starting up the vcpu should be the most difficult part of the assignment. The begin state that the vcpu needs should be properly explained.

6.8 Available example code

There is a considerable amount done to build an example implementation. This code is still a work in progress, but can still serve as a starting point for a full implementation. There is code available for step 1 up to step 10.3, from table 5. Step 4 is the exception, only some code is available for paging in pv mode. The code is available in a private repository.

Conclusion and final remarks

In this thesis appropriate technology and language were searched to design an educational kernel. Xen was chosen as the most suitable virtualization, as it hides much complexity, has many convenient features, and is able to start up other vcpus with a hypercall. In the language department, Rust was the winning candidate, with its strong type and memory safety, convenient features, and interesting programming model. In practice however, the language still has some issues, as detailed further below. With simplicity and scalability in mind, a software OS architecture was devised that is similar to the OS architecture of Mach. Shared state is avoided by splitting kernel in services that run in their own process. Shared memory between more than two vcpu is avoided, and instead a message passing scheme was devised. Ideally, the architecture would have hardware assisted message passing, and ability to change the state of other vcpu with messages. Sadly, these features are not available on x86-64, and complicated the design considerably. Instead, we provide framework components to abstract the complexity of the underlying architecture and suggest guidelines to design the accompanying educational activities.

7.1 Epilogue on the choice to use Rust

Rust was chosen due to the high rating that was given to Rust. Even so, this is one of the few large projects that uses Rust in an freestanding environment. This section highlights some of the problems of using freestanding Rust.

First, Rust does not offer anything to generalize over the size of an array, and this something that is needed quite often. A work around is to wrap the array in a struct, and implement every trait that is needed. This requires a lot of boilerplate. A similar problem is that there is no compile-time calculation of size, which means that in some context, like the initialization of a static variable, size calculations cannot be used.

The core crate contains useful functions, yet a lot of useful modules are not available. For example, the C types, and the general types for reading and writing data are only available in “std,” a non-core crate. There are also many functions and modules that are marked as unstable. The intrinsics module contain many useful atomic operations, but the documentation makes it clear that it is likely that it always be unstable, and that the intrinsics from the non-core standard library should be used. Another problem is the conversion of C structs that makes use of unions. In Rust there is no equivalent. A workaround is to replace the union with a struct that wraps an array that is as large as the union. For every union entry, there should be a method defined on the struct that converts the array to the appropriate data type. Care should be taken that an array has a different memory layout than the union, and hence can result in incorrect amounts of padding.

There is also some inconvenience with the use of large arrays. Rust has the Debug trait to convert data to a string. This is used in string interpolation and print macros. Every array of a given size has its own type, hence every array of given size needs its own implementation of the Debug trait. An implementation is provided for arrays up to 32 elements. This makes its

impossible to derive an implementation of Debug for a new data type that uses large arrays. But (large) arrays, with a size that is a power of two, occur very frequently. Moreover, it is not possible to add additional implementations of Debug, as the array type and the Debug trait is not defined in your own module. The only option left is to create a new implementation for your new data type, which requires much boiler plate.

7.2 Future work

There is still much work left to be done. In the current proposed framework, the processes always remain on the same vpcu where they started. This could mean that all the processes are running on the same vpcu. A load balancer is needed to solve this problem. Secondly, no mechanism was discussed to determine to which vcpu a message must be send, when a load balancer is active. Thirdly, there are no facilities to measure performance. In OS research and teaching this is an important feature.

Bibliography

- [1] Committee Specification Draft 01. *Virtual I/O Device (VIRTIO) Version 1.0*. <http://docs.oasis-open.org/virtio/virtio/v1.0/csprd01/virtio-v1.0-csprd01.html>. December 2013.
- [2] Guillermo Aguirre et al. “Experiencing Minix as a didactical aid for operating systems courses”. In: *ACM SIGOPS Operating Systems Review* 25.3 (1991), pp. 32–39.
- [3] Paul Barham et al. “Xen and the art of virtualization”. In: *ACM SIGOPS Operating Systems Review*. Vol. 37. 5. ACM. 2003, pp. 164–177.
- [4] Tegawendé F Bissyandé et al. “Popularity, interoperability, and impact of programming languages in 100,000 open source projects”. In: *Computer Software and Applications Conference (COMPSAC), 2013 IEEE 37th Annual*. IEEE. 2013, pp. 303–312.
- [5] Pierre Carbonnelle. *PopularitY of Programming Language*. May, 2016.
- [6] David Chisnall. *The definitive guide to the xen hypervisor*. Pearson Education, 2008.
- [7] Wayne A Christopher, Steven J Procter, and Thomas E Anderson. “The Nachos instructional operating system”. In: *Proceedings of the USENIX Winter 1993 Conference Proceedings on USENIX Winter 1993 Conference Proceedings*. USENIX Association. 1993, pp. 4–4.
- [8] Russ Cox, M Frans Kaashoek, and Robert Morris. *Xv6, a simple Unix-like teaching operating system*. 2011.
- [9] Gautier De Montmollin. *Transparent Language Popularity Index*. Juli, 2013.
- [10] Mehmet Gönen and Ethem Alpaydın. “Multiple kernel learning algorithms”. In: *Journal of Machine Learning Research* 12.Jul (2011), pp. 2211–2268.
- [11] TIOBE Group et al. *TIOBE Index for ranking the popularity of Programming languages*. May, 2016.
- [12] Jorrit N Herder et al. “MINIX 3: A highly reliable, self-repairing operating system”. In: *ACM SIGOPS Operating Systems Review* 40.3 (2006), pp. 80–89.
- [13] Dan Hettena and Rick Cox. *A Guide to Nachos 5.0 j*.
- [14] David Hovemeyer, Jeffrey K Hollingsworth, and Bobby Bhattacharjee. “Running on the bare metal with GeekOS”. In: *ACM SIGCSE Bulletin*. Vol. 36. 1. ACM. 2004, pp. 315–319.
- [15] Tyrone Nicholas and Jerzy A Barchanski. “TOS: an educational distributed operating system in Java”. In: *ACM SIGCSE Bulletin*. Vol. 33. 1. ACM. 2001, pp. 312–316.
- [16] Stephen OGrady. “The RedMonk Programming Language Rankings: May 2016”. In: *Tecosystems blog* (2016).
- [17] Ben Pfaff, Anthony Romano, and Godmar Back. “The pintos instructional operating system kernel”. In: *ACM SIGCSE Bulletin*. Vol. 41. 1. ACM. 2009, pp. 453–457.
- [18] *PVH specification*. URL: <https://xenbits.xen.org/docs/4.5-testing/misc/pvh.html> (visited on 08/15/2016).
- [19] Andrew S Tanenbaum and Albert S Woodhull. *Operating systems: design and implementation*. Vol. 2. Prentice-Hall Englewood Cliffs, NJ, 1987.

- [20] David Wentzlaff and Anant Agarwal. “Factored operating systems (fos): the case for a scalable operating system for multicores”. In: *ACM SIGOPS Operating Systems Review* 43.2 (2009), pp. 76–85.
- [21] *Xen Project Overview*. URL: https://wiki.xen.org/wiki/Xen_Project_Software_Overview (visited on 08/15/2016).