Chapter 7: What's Next?

Introduction

And now, after writing a simple operating system's kernel and learning the basics of creating kernels, the question is "What's Next?". Obviously, there is a lot to do after creating 539kernel and the most straightforward answers for our question are the basic well-known answers, such as: implementing virtual memory, enabling user-space environment, providing graphical user interface or porting the kernel to another architecture (e.g. ARM architecture). This list is a short list of what you can do next with your kernel.

Previously, I've introduced the term *kernelist* [1] in which I mean the person who works on designing operating system kernels with modern innovative solutions to solve real-world problem. You can continue with your hobby kernel and implement the well-known concepts of traditional operating systems that we have just mentioned a little of them, but if you want to create something that can be more useful and special than a traditional kernel, then I think you should consider playing the role of a kernelist.

If you take a quick look on current hobby or even production operating system kernels through GitHub for example, you will find most of them are traditional, that is, they focus on implementing the traditional ideas that are well-known in operating systems world, some of those kernels go further and try to emulate another previous operating system, for example, many of them are Unix-like kernel, that is, they try to emulate Unix. Another examples are ReactOS[2] which tries to emulate Microsoft Windows and Haiku[3] which tries to emulate BeOS which is a discontinued proprietary operating system. Trying to emulate another operating systems is good and has advantages of course, but what I'm trying to say that there are a lot of projects that focus on this line of operating systems development, that is, the traditionalists line and I think the line of kernelists needs to be focused on in order to produce more innovate operating systems.

I've already said that the kernelist doesn't need to propose her own solutions for the problems that she would like to solve. Instead of using the old well-known solutions, a kernelist searches for other better solutions for the given problem and designs an operating system kernel that uses these solutions. Scientific papers (papers for short) are the best place to find novel and innovative ideas that solve real-world problem, most probably, these ideas haven't been implemented or adopted by others yet[4].

[1] In chapter where the distinction between a kernelist and a traditionalist has been established.

[2] https://reactos.org/

[3] https://www.haiku-os.org/

[4] Scientific papers can be searched for through a dedicated search engine, for example, Google Scholar.

In this chapter, I've chosen a bunch of scientific papers that propose new solutions for real-world problem and I'll show you a high-level overview of these solutions and my goal is to encourage interested people to start looking to the scientific papers and implement their solutions to be used in the real-world. Also, I would like to show how the researches on operating systems field (or simply the kernelists!) innovate clever solutions and get over the challenges, this could help an interested person in learning how to overcome his own challenges and propose innovate solutions for the problem that he faces.

Of course, the ideas on the papers that we are going to discuss (or even the other operating system's papers) may need more than a simple kernel such as 539kernel to be implemented. For example, some ideas may need a networking stack being available in the kernel, which is not available in 539kernel, so, there will be two options in this case, either you implement the networking stack in your kernel or you can simply focus on the problem and solution that the paper present and use an already exist operating system kernel which has the required feature to develop the solution upon this chosen kernel, of course, there are many open source options and one of them is HelenOS[5] microkernel[6].

A small note should be mentioned, this chapter only shows an overview of each paper which means if you are really interesting on the problem and the solution that a given paper represents, then it's better to read it. It is easy to get a copy of any mentioned paper in this chapter, you just need to search for its title in Google Scholar (https://scholar.google.com/) and a link to a PDF will show for you. However, before getting started in discussing the chosen papers, I would like in the next subsection to discuss a topic that I've deferred till this point, this topic is related to the architecture design of a kernel.

The Design of Kernel's Architecture: Monolithic vs. Microkernel

The architecture of an operating system, as in any other software, can be designed in many different ways and the most commonly known kernel's architecture are *monolithic* and *microkernel*. When the monolithic architecture is used in a kernel, the whole code of the kernel runs in the kernel-mode, that is, all the code of the kernel (hence, all its different modules) has the same privileges to perform any operation and to change anything in the system, even the device drivers. A notable example of monolithic kernels is Linux, also, the modern BSD family (FreeBSD, NetBSD and OpenBSD) uses the monolithic architecture, the original Unix itself used a monolithic kernel. As you may notice, 539kernel is a monolithic kernel.

The other well-known design is microkernel, where not every component of an operating system kernel is run as a privileged code (in kernel-mode), instead, only the code that really needs to perform privileged instructions. The other

[5] http://www.helenos.org/

[6] The concept of *microkernel* will be explained in this chapter.

parts of the kernel that doesn't need to perform privileged instructions are separated from the microkernel and run in the user-mode as any other user application and they are known as *servers* in microkernel architecture. The goal of those servers is providing an interface that represents the kernel services for the user applications, and when a privileged instructions needed to be performed, the server communicates with the microkernel which runs in kernel-mode to do so. For example, when a user process needs to create new process, it needs to communicate with *processes server* which runs in user-mode and request from it to create a new process. The processes server maintains the processes list and their process control block since these data structures don't need to be in the kernel-mode or kernel's address space, so, when a process creation request arrives, the new entry for the new process will be the responsibility of the processes server with no need to run any privileged code, once a privileged code is needed, the microkernel will be called by the server.

Let's take process management module of 539kernel as example. This module provides one function which is `process_create` in `process.c`, if you read the code of this module you will see there is no any part of it needs to run in kernel-mode. That means, if 539kernel was a microkernel, this whole module can run as a user-space server instead of being in the kernel itself. Another example from 539kernel is the scheduler, you can see for example in the function `scheduler` in `scheduler.c` that there is no need to run it in the kernel-mode, so, if 539kernel was a microkernel, this module can be run as a separated user-space server instead. If you review the code of `scheduler` carefully, you can see that is needs to reach the processes list and also needs to modify the processes control block, that's mean the scheduler server needs to communicate with processes server to perform these operations. In microkernels this can be done through message passing, for example, the scheduler server can send a message to the processes server to get the ready processes list or to change some attribute in a process control block and so on, the same is applicable between the other servers. The function `run_next_process` in `scheduler.c` is an example from 539kernel's scheduler module that needs to run as privileged code, so, if 539kernel was a microkernel this function should reside in the kernel itself and not in the scheduler server. Another example of 539kernel that should be in the kernel itself instead of the server is the interrupt handler `isr_32` in `idt.asm`.

The goal of microkernel design is keeping the code that needs to run as privileged code as small as possible and move all the other code to the user-space. This can make the kernel itself more secure, reliable and easier to debug. Microkernels have a long history of research to improve its performance and make it better, there are many microkernels available nowadays, for example, L4, Mach which has been used in NeXTSTEP operating system that the current macOS based on [7], Minix, HelenOS and Zircon which is the kernel of Fuchsia operating system and maybe one of the famous microkernel's related stuff is a debate known as *Tanenbaum–Torvalds debate* between Andrew S. Tanenbaum (the creator of Minix

[7] Though, macOS' kernel is considered as a hybrid kernel and not a microkernel.

and the author of the book "Operating Systems: Design and Implementation") and Linus Torvalds (the creator of Linux) in 1992 after few months Linux kernel release[8].

In-Process Isolation

In current operating systems, any part of a process can read from and write to any place of the same process' memory. Consider a web browser which like any other application consists of a number of different modules (in the perspective of programmers) and each one of them handles different functionality, rendering engine is one example of web browser's module which is responsible for parsing HTML and drawing the components of the page in front of the user. When an application is represented as a process, there will be no such distinction in the kernel's perspective, all application's modules are considered as one code that each part of it has the permission to do anything that any other code of the same process can do.

For example, in web browser, the module that stores the list of web pages that you are visiting right now is able to access the data that is stored by the module which handles your credit card number when you issue an online payment. As you can see, the first module is much less critical than the second one and unfortunately if an attacker can somehow hack the first module through an exploitable security bug, she will be able to read the data of the second module, that is, your credit card information and nothing is going to stop her.

This happens due to the lack of *in-process isolation* in the current operating systems, that is, both sensitive and insensitive data of the same process are stored in the same address space and any part of the process code is permitted to access all these data, so, there is no difference in your web browser's process between the memory region which stores that titles of the pages and the region which stores you credit card information. A severe security bug known as *HeartBleed vulnerability* showed up due to the lack of in-process isolation. Next, two of the solutions for the problem of data isolation that has been proposed by kernelists will be discussed.

Lord of x86 Rings

A paper named "Lord of the x86 Rings: A Portable User Mode Privilege Separation Architecture on x86" [9] proposes an architecture (named LOTRx86 for short) which provides an in-process isolation, the paper uses the term *user-mode privilege separation* which has the same meaning. LOTRx86 doesn't use

[8] The title of the post which started the debate was "LINUX is obsolete" by Andrew Tanenbaum. The text of the debate is available online here: https://www.oreilly.com/openbook/opensources/book/appa.html

[9] Authored by Hojoon Lee, Chihyun Song and Brent Byunghoon Kang. Published on 2018.

the new features of the modern processors to implement the in-process isolation, Intel's Software Guard Extensions (SGX) is an example of these features. The reason of not using such modern feature in LOTRx86 is portability, while SGX is supported in Intel's processors, it is not in AMD's processors[10] which means that employing this feature will make our kernel only works on Intel's processor and not AMD's. Beside that, SGX is a relatively new technology[11] which means even older Intel's processors don't support it and that makes our kernel less portable and can only run on specific types of Intel's processors. So, if we would like to provide in-process isolation in our kernel, but at the same time, we want it to work on both Intel's and AMD's processors, that is, portable [12], what should we do? According to LOTRx86, we use privilege levels to do that.

Throughout this book, we have encountered x86 privilege levels and we know from our previous discussions that modern operating systems only use the most privileged level `0` as kernel-mode and the least privileged level `3` as user-mode. In LOTRx86 a new area in each process called *PrivUser* is introduced, this area keeps the sensitive data of the process and it's only accessible through special code that runs on the privilege level `2`, so, in a kernel which employs LOTRx86 a process may run in privilege level `3` (user-mode), as in modern operating systems, and may run in privilege level `2` (PrivUser). Most of the normal work of a process will be done in level `3`, but when the code is related to sensitive data, such as storing, accessing or processing them, the process will run on level `2`. Of course, the sensitive data cannot be accessed by process' normal code since the latter runs on level `3` and the former needs a code that runs on privilege level `2` to be accessed. If an attacker exploit a vulnerability that allows him to read the memory of the process, he will not be able to read the secret data if this vulnerability is on the normal code of the process.

A kernel with LOTRx86 should provide a way for the programmers to use the feature provided by LOTRx86, so, the authors of the paper propose a programming interface named *privcall* which works like Linux kernel's system calls. Through this interface an application programmer can write functions (routines) that process the secret data, these functions will run on privilege level `2` and will be stored in PrivUser, we will call these functions as *secret functions* in our coming discussion. When the normal code of the process need to do something with some secret data that is stored in PrivUser a specific secret function can be called through `privcall` interface, once this call is issued, the current privilege level will be changed from `3` (user-mode) to `2` (PrivUser[13]) by using x86 call gates that we have discussed earlier in this book . Note that this

[10] Beside Intel, also AMD provides processors that use x86 architecture.

[11] Intel's SGX is deprecated in Intel Core but still available on Intel Xeon.

[12] In LOTRx86 when the term *portable* is used to describe something it means that this thing is able to work on any modern x86 processor. The same term has another boarder meaning, for example, if we use the boarder meaning to say "Linux kernel is *portable*" we mean that it works on multiple processors architecture such as x86, ARM and a lot more and not only on Intel's or AMD's x86.

[13] In the paper, the name PrivUser means two things, the execution mode and the secret memory area.

solution **mitigates** vulnerabilities like HeartBleed but doesn't **prevent** them necessarily.

To implement this architecture, two requirements should be satisfied in order to reach the goal. The first requirement is called `M-SR1` in the paper and it states that the PrivUser area should be protected from the normal user mode which most of the application's code run on. The second requirement is called `M-SR2` in the paper and it states that the kernel should be protected from PrivUser code.

To satisfy the first requirement, the pages of PrivUser are marked as privileged pages in their page entry [14], that is, the code that run on privilege level `3` cannot access them while the code that runs on levels `0`, `1` and `2` can. To satisfy the second requirement, the authors propose to use segmentation, `LDT` table is employed to divided each process into segments and a special segment for the secret functions and data, that is, PrivUser is defined and the definition of this segment indicates that the secret functions can only access the secret data under privilege level `2` in order to protect the kernel's data which reside in privilege level `0`.

This was the high-level description of LOTRx86 solution, there are some challenges that have been faced by the authors and the details of them and how they overcame them can be found in the paper, so, if you are interested on implementing LOTRx86 in your kernel, I encourage you to read the original paper which also discusses how the authors managed to implement their solution in Linux kernel as kernel modules, also, the paper shows the performance evaluation of their implementation. There is something to note, the authors assume that the solution is implemented in `64-bit` environment instead of `32-bit` and due to that they faced some challenges that the may not be faced in `32-bit` environment.

Of course LOTRx86 is not the only proposed solution for our problem, there are a bunch more and some of them are mentioned on the same paper that we are discussing. What makes LOTRx86 differs from them is the focus on a solution that has a better performance and portable as we have examined in the beginning of this sub-section .

As you saw in this solution how the authors played the role of a kernelist, they proposed a solution for real-world problem, they used some hardware feature that is usually used in a different way in the traditional operating systems (privilege level `2`) and they proposed a different and useful idea for operating system kernels.

Endokernel

The proposed solution In LOTRx86 paper isolates the memory within the process but what about the other system resources (e.g. files)? For example, what if a

[14] We have discussed this bit in a page entry in Chapter .

critical module in the process needs to read and write on a secret file while the other modules of the same process should not reach this file at all. The only system resource that LOTRx86 protects is the memory while the other resources of the system are accessible by any module within the process.

The paper "The Endokernel: Fast, Secure, and Programmable Subprocess Virtualization" [15] proposes a solution to handle this case by modifying the traditional process model which used by most modern operating systems. In Endokernel Architecture a monitor is attached within each process. This monitor, which is called endokernel, isolates itself from the untrusted parts of the process and also provides a lightweight virtual machine, called endoprocess, to the rest of the process and through defined polices the isolation can be enforced, for example, some processor's instructions can be permitted to be executed by the untrusted parts of the process without monitoring but some other can be defined that the should be monitored. Also, the filesystem's operations that are allowed to be used can be defined by the policies and the endokernel is going to ensure that these policies are enforced.

Nested Kernel

In monolithic design, the kernel is considered as one entity and each component of the kernel is able to read/modify the data and maybe the code of another component since the whole of the kernel's code works on kernel mode. Beside the standard components of the kernel (e.g. process management and memory management), usually, the device drivers are considered as a part of the monolithic kernel and they run on the kernel mode, these device drivers are, most probably, written by a third party entity which makes them considered as an untrusted code, also, they may be buggy if they are compared to the standard code of the kernel. Any exploitable bug in any part of a monolithic kernel (either in a device driver or not) will give the attacker the access to the whole kernel. This problem reminds us with the problem which has been presented earlier in section but this time the kernel is the one which suffers from it.

Microkernel design solves this problem by separating the most components of the kernel as user-space servers, but what if we would like to keep the monolithic design and have this kind of separation? This is what a paper titled "Nested Kernel: An Operating System Architecture for Intra-Kernel Privilege Separation" [16] is trying to do by proposing a new kernel's design called *nested kernel*.

Memory is the root of all evil, that's what I feel this paper is trying to tell us. In nested kernel design, the operating system kernel is divided into two parts, the first one is nested kernel and the second part is *outer kernel*. The nested

[15] Authored by: Bumjin Im, Fangfei Yang, Chia-Che Tsai, Michael LeMay, Anjo Vahldiek-Oberwagner and Nathan Dautenhahn. Published on 2021.

[16] Authored by Nathan Dautenhahn, Theodoros Kasampalis, Will Dietz, John Criswell and Vikram Adve. Published on 2015.

kernel is isolated from the outer kernel and both parts run on kernel mode. The job of nested kernel is to isolate the memory management unit (MMU) from the outer kernel. To make the outer kernel able to use the functionality that MMU provides, the nested kernel exposes and controls an interface of the MMU, this interface is called *virtual MMU* (*vMMU*) in the paper, so, if any part of the outer kernel needs to manipulate the state of MMU then vMMU interface can be used. The nested kernel part has small and trusted code while the outer kernel contains all other code that cannot be trusted (e.g. device drivers) or may be buggy. When we say isolating MMU we mean that the data structures and registers that build the state of MMU are isolated, so, in x86 for example, isolating MMU means isolating page directory, page tables and the control registers that are related to paging.

The memory regions which a kernel implementer would like to protect from being modified by the outer kernel (protected memory) are marked as read-only region in nested kernel design and only the nested kernel has the permission to modify them. For example, say that you have decided to protect the memory that contains the code of the kernel which checks if the current user has the permissions to read or modify a specific file, this region can by marked as read-only and can be protected by the nested kernel all the time from being modified by any part of the outer kernel. Now, assume that an attacker found an exploitable security bug in one of the device drivers, and his goal is to modify that code of permission checking in order to let him to read some critical file, this cannot be done since the memory region is protected and read-only, the paper discusses how in details how to ensure that the outer kernel doesn't violate the protection of nested kernel in x86 architecture.

That's not the whole story. Making the nested kernel the only way to modify the protected memory by the outer kernel means that the nested kernel can be a mediator which will be called before any modification performed. This will let the kernel's implementer to define security policies and enforce them while the system is running. For example, the authors propose *no write policy* which doesn't let the outer kernel to write on a specific memory region at all (e.g. the example of checking permissions code). Another proposed policy is *write-once policy* which lets the outer kernel to write to a region of memory just one time, this policy will be useful with the memory region that contains the `IDT` table for example, so, the attacker cannot modify the interrupt service routines after setting them up by the trusted code of outer kernel. More policies were presented in the paper. You can see here how the kernelists proposed a new kernel design other than the popular ones (microkernel and monolithic) in order to solve a specific real-world problem.

Multikernel

The paper "The Multikernel: A new OS architecture for scalable multicore systems" [17] shows a good example of kernelists who get rid of the old designs completely in order to provide a modern one which is more suitable for current days. In the paper, the authors have observed the new trends in the modern hardware, these trends motivated them to propose a new kernel architecture named *multikernel*. One of these observations is the diversity of the new systems, according to the authors, the operating systems in the new systems need to work with machines that may have cores with different instruction set architectures , that is, they have heterogeneous cores, either in term of instruction set architecture or performance. Another observation is that the message passing is now easier in the modern hardware and can be used instead of shared memory in order to share information between two processes for example, the idea of multikernel aims to handle these observations and provide an architecture of a kernel that is suitable for the modern multicore systems.

In multikernel architecture, a multicore system is handled as a network of cores, as if each core is a separate processor, and the communications between the cores are performed through message passing, it is not necessary that the cores belong to the same machine. When the cores are handled as a network of machines, the algorithms and techniques of distributed systems can be used.

The design of multikernel depends on three principles. First, all communications between the cores in the kernel should be explicit through message passing and no implicit communications (e.g. through shared memory) is allowed, one of the benefits of this principle is the ability to use well-known networking optimizations in order to make the communications between cores more efficient. Also, making the communication explicit can help in reasoning about the correctness of the kernel's code. The second principle is separating the structure of the operating system as much as possible from the hardware, that is, the structure should be hardware neutral. The benefits of this principle are obvious and one of them is making the adaptation of processor's specific optimization easier [18]. The last principle is dealing with the state of the operating system (e.g. processes table) as replicated instead of shared, that is, when a core need to deal with a global data structure, a copy of this data structure is sent to this core instead of using just one copy by all the cores in the system. Based on these design principles, the authors built an implementation called Barrelfish, according to the authors, this implementation is an example of multikernel but not the only way to build one. The paper discusses in details how they designed Barrelfish to realize multikernel's design principles and how they overcame the challenges that the have faced.

[17] Authored By: Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach and Akhilesh Singhania. Published in 2009.

[18] The paper mentioned that applying one of optimizations on Windows 7 caused changes in 6,000 lines of code through 58 files.

Dynamic Reconfiguration

Changing a specific module while the system is running can be an important aspect in some systems, for most desktop users, when some module of a system is changed, due to updating the system for example, it will be fine to reboot the system to get the new changes applied, but what about a server that needs to run all the time with no downtime, rebooting it is not an option. Current operating systems still require a reboot when an update to specific parts is performed, for example, updating Linux kernel in a running system requires a reboot to this system to be able to use the new version of the kernel.

Dynamic reconfiguration is the process of changing a specific module of the system while keeping it running without the need of rebooting it, that's how a paper titled "Building reconfigurable component-based OS with THINK" [19] defines this term. According to the paper, dynamic reconfiguration consists of the followings steps: First, the part that we would like to reconfigure (the reconfiguration target) should be identified and separated from other parts, to do that, THINK framework uses a component model called Fractal [20] in order to identify each part of the system as a separated component, after that, the process of reconfiguration is going to deal with these components, for example, in 539kernel the process management part, the scheduler, the memory management part, the allocator and the filesystem can be defined as separated components, as you can see each of these part has its own functionality and can be encapsulated, by using dynamic reconfiguration we can for example change the current scheduler with another one while the system is running.

The second step is to make sure that the reconfiguration target is on the safe state, that is, there is no other part that is using the target right now, thread counting is one technique that has been proposed in the paper to detect safe state, when employing this technique any call to a component causes the thread counter to increase by `1` and when this call finishes the thread counter decreases by `1`, a component is on the safe state when the thread counter is `0`, that is, no thread (or process) is currently using the target component. After the target component reaches the safe state it can be changed to the new component, the context of the target should be moved to the new component and after that the execution of the component can be resumed.

Unikernel

Virtualization is widely used today and cloud computing is an obvious example of employing virtualization technologies. Nowadays, you can easily start and stop virtual machines that run a commodity operating system (e.g. Linux or Windows) and via this virtual machine you can run whatever software you wish

[19] Authored by: Juraj Polakovic, Ali Erdem Özcan and Jean-Bernard Stefani. Published on 2006

[20] https://fractal.ow2.io/

as if this virtual machine is a real one. There are many cases where a virtual machine is used to provide just one thing, for example, a virtual machine that runs a web server solely. To do that, of course an operating system is needed to be installed in the virtual machine, say for example Linux, and of course a web server should be installed on top this operating system, say Apache. Linux (and modern general purpose operating systems) is a multiprocess and multiuser kernel which contains a lot of code that handle the protection of the processes, the users and the kernel itself (via the separation of kernel-mode and user-mode as we have discussed through this book), beside that, there are a lot of services that are provided in general purpose operating systems so they can be suitable for all users.

In our example of the virtual machine which only runs a web server all of these services are not needed, they can be omitted and only the services that are needed by the web server are kept, this is what *unikernels* do. In this model of kernels design, everything that is not needed is omitted, even the separation between the kernel and the user application (in our example the web server) and let both of them to run in a single address space. All of these changes on the kernel's architecture provide us with many benefits, the size of the kernel and the final binary will be smaller, it will have a better performance [21], it will boot faster and the attack surface will be smaller.

I think unikernel is a good path to start your journey as a kernelist, especially that this topic is gaining a momentum these days. The idea behind a unikernel is simple, a skeleton of an operating system's kernel which targets a specific architecture (e.g. x86) is provided to the user with specific services (e.g. functions and so on) to make it easy for the user to write his own application, in this stage, the combination of the kernel and those provided services is known as a *library operating system*, after writing the application, say a web server, both the application and the kernel are built as one entity which is the unikernel that is going to run on a virtual machine and provide a specific service.

There are many library operating systems available, for example: IncludeOS [22] which its design is presented in a paper titled "IncludeOS: A minimal, resource efficient unikernel for cloud services" [23], Unikraft [24] which its design is presented in a paper titled "Unikraft: Fast, Specialized Unikernels the Easy Way" [25], OSv [26] and MirageOS [27]. Also, there are many new scientific papers that try to find

[21] In the website of a unikernel called Unikraft the following is stated: "On Unikraft, NGINX is 166% faster than on Linux and 182% faster than on Docker".

[22] https://www.includeos.org/

[23] Authored by Alfred Bratterud, Alf-Andre Walla, Harek Haugerud, Paal E. Engelstad and Kyrre Begnum. Published on 2015.

[24] https://unikraft.org/

[25] Authored by Simon Kuenzer, Vlad-Andrei Bădoiu, Hugo Lefeuvre, Sharan Santhanam, Alexander Jung, Gaulthier Gain, Cyril Soldani, Costin Lupu, Stefan Teodorescu, Costi Răducanu, Cristian Banu, Laurent Mathy, Răzvan Deaconescu, Costin Raiciu and Felipe Huici. Published on 2021.

[26] https://osv.io/

[27] https://mirage.io/

solutions for unikernel problems and advance the area. For example, the paper "Towards a Practical Ecosystem of Specialized OS Kernels" [28] proposes a way to build an ecosystem for library operating systems which helps the user to find a kernel that fits his needs and helps in building the last result of the user's application. Another paper is titled "A Binary-Compatible Unikernel" [29] which proposes a unikernel named HermiTux [30] that provides binary compatibility with Linux applications, that is, instead of writing a wholly new application to be used as a unikernel, with binary compatibility one of mature applications that already exists for Linux can be used instead, for example, running Apache web server a unikernel instead of writing a wholly new web server is most probably better idea.

Of course, there are a lot more papers either about unikernels or any other operating system topics, just search for them and you will find a lot. I hope you a happy kernelist/traditionalist journey and thanks for reading this book!

[28] Authored by Conghao Liu and Kyle C. Hale. Published on 2019.

[29] Authored by Pierre Olivier, Daniel Chiba, Stefan Lankes, Changwoo Min and Binoy Ravindran. Published on 2019

[30] https://ssrg-vt.github.io/hermitux/