OS01: OS Introduction *

Partially based on Chapter 1 of [Hai19]

Jens Lechtenbörger

Computer Structures and Operating Systems 2023

Contents

1 Introduction

1.1 Learning Objectives

- Explain notion of Operating System and its goals
  - Explain notion of kernel with system call API
    - (More details in next presentation)
- Explain notions and relationships of process, thread, multitasking

Learning objectives specify what you should be able to do after having worked through a presentation. Thus, they offer guidance for your learning.

Each learning objective consists of two major components, namely an *action* verb and a topic. Action verbs specify what actions you should be able to perform concerning the topic, and they indicate the target level of skill (in Bloom's Taxonomy or its revised version as sketched under the hyperlink above).

You may want to think of learning objectives as sample exam tasks.

1.2 Recall: Big Picture of CSOS

- Computer Structures and Operating Systems (CSOS)

*This PDF document is an inferior version of an OER HTML page; free/libre Org mode source repository.

– CS: How to build a computer from logic gates?

Figure 1: "NAND" under CC0 1.0; from GitLab

  * Von Neumann architecture
  * CPU (ALU), RAM, I/O

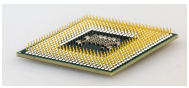

Figure 2: "CPU" under CC0 1.0; cropped and converted from Pixabay

– OS: What **abstractions** do Operating Systems provide for applications?

  * Processes and threads with scheduling and concurrency, virtual memory

Figure 3: "Pong in TECS VM" under GPLv2; screenshot of VM of TECS software suite

  * What is currently executing why and where, using what resources how?

1.2.1 OS Responsibilities

Warning! External figure **not** included: "What does your OS even do?" © 2016 Julia Evans, all rights reserved from julia's drawings. Displayed here with personal permission.
(See HTML presentation instead.)

Several OS presentations contain awesome drawings by Julia Evans such as this one. In general, these drawings are meant to speak for themselves as additional perspective on class topics (or even beyond class topics), and they come without any explanation.

Except for this additional context, this drawing would not be accompanied by a note. It shows typical services provided by OSs and to be used by programs. The interface between programs and OS will be revisited as API of so-called "system calls".

2 Operating Systems

2.1 Sample Modern Operating Systems

- Different systems for different scenarios
  - Mainframes
    - BS2000/OSD, GCOS, z/OS
  - PCs
    - MS-DOS, GNU/Linux, MacOS, Redox, Windows
  - Mobile devices
    - Variants of other OSs
    - Separate developments, e.g., BlackBerry (BlackBerry 10 based on QNX, abandoned), Google Fuchsia, Symbian (Nokia, most popular smartphone OS until 2010, now replaced)
  - Gaming devices
  - Real-time OS
    - Embedded systems
    - L4 variants, FreeRTOS, QNX, VxWorks

There is a vast variety of OSs for different devices and usage scenarios, of which this slide shows a selection.

The goal of the OS sessions is not to turn you into an expert for any specific OS, but to teach you major concepts and techniques that are shared by most modern OSs. As explained previously my hope is that you can apply your knowledge on the one hand when designing, analyzing, or implementing information systems and on the other when taking control of your own devices.

Based on my personal beliefs, I will not teach you anything about non-free OSs (except maybe first steps to get away from them). In particular, examples shown in presentations and in class will be based on the free OS GNU/Linux. As GNU/Linux is free, you can experiment with it at any level of detail yourself.

2.2 Definition of Operating System

- Definition from [Hai19]: **Software**
  - that **uses hardware** resources of a computer system
  - to provide support for the **execution of other software**.

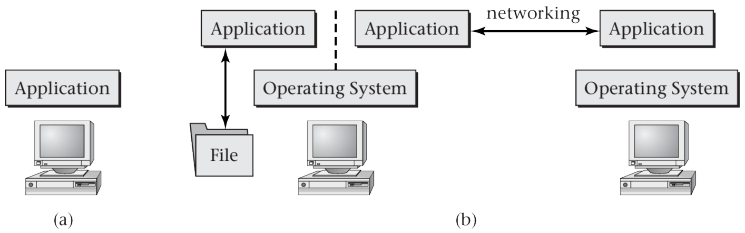

Figure 4: "Figure 1.1 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

Part (a) of the figure shows the situation of a computer without an OS. Here, applications (and programmers) need to interact with hardware directly at a low level of abstraction. This is what you did on Hack. E.g., you needed to know a specific memory location to access the keyboard.

Part (b) illustrates typical services provided by an OS to shield applications (and programmers) from hardware-specific details. E.g., multiple applications may run concurrently, interact as parts of distributed systems with networking functionality, or share persistent storage at the abstraction of file systems (without needing to worry about, say, specifics of particular keyboards, disks, or network cards and their interfaces).

What you see here is a typical example of layering to hide lower-layer details with the abstractions of an interface in software engineering: The OS provides an API (see next slide) of functions that application programmers can invoke to access OS services, in particular to access underlying hardware. As explained later, that API is provided by a core part of the OS, which is called kernel and whose functions are called system calls.

2.2.1 Aside: API

- API = Application Programming Interface
  - Set of functions or interfaces or protocols defining how to use some system (as programmer)
  - E.g., Java 18 API
    - Packages with classes, interfaces, methods, etc.

2.2.2 OS Services

- OS services/features/functionality defined by its API
  - Functionality includes
  - Support for **multiple concurrent** computations
    - Run programs, divide hardware, manage state
  - **Control interactions** between concurrent computations
    - E.g., locking, private memory
  - Typically, also **networking** support

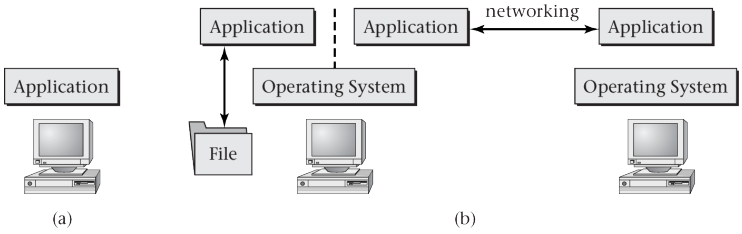

Figure 5: "Figure 1.1 of [Hai17]" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

2.3 OS, Kernel, User Interface

- Boundary between OS and applications is fuzzy
- **Kernel** is fundamental, core part of OS
  - Kernel defines API and services via **system call** interface
  - (More details on next and later slide)
- User interface (UI; not part of kernel)
  - UI = process(es) **using** kernel functionality to handle user input, start programs, produce output, ...
    - User input: Voice, touch, keyboard, mouse, etc.
    - Typical UIs: Command line, explorer for Windows, various desktop environments for GNU/Linux, virtual assistants
  - Note: OSs for embedded systems may not have UI at all

2.3.1 How to Talk to OSs

Warning! External figure **not** included: "How to talk to your operating system" © 2016 Julia Evans, all rights reserved from julia's drawings. Displayed here with personal permission.
(See HTML presentation instead.)

System calls are an important concept as they define the services provided by an OS kernel in terms of an API. Here, you see names of sample system calls, which are not important to remember but which might help to shape your understanding, before system calls are revisited subsequently.

2.3.2 User Space and Kernel Space

Warning! External figure **not** included: "User space vs. kernel space" © 2016 Julia Evans, all rights reserved from julia's drawings. Displayed here with personal permission.
(See HTML presentation instead.)

This drawing introduces a distinction between *user space* and *kernel space*, which is revisited on a later slide and in the next presentation. Briefly, in kernel space the OS has full control over the underlying hardware, while applications running in user space need to invoke system calls to ask the OS to perform more privileged operations (e.g., to receive input from hardware devices or to write to them as illustrated for a sample file access here).

System calls lead to so-called *context switches* between different execution contexts, here between user space and kernel space (and back), which will be revisited in later presentations when discussing interrupt handling and thread switching.

2.3.3 OS Size

- From [TB15]
  - Size of source code of the heart Windows or GNU/Linux is about **5 million lines of code**
    - Think of book with 50 lines per page, 1000 pages
    - Need 100 books or an entire bookcase
  - Windows with essential shared libraries is about **70 million lines of code**

    - 10 to 20 bookcases
- How to **understand** or maintain that?
  - → Abstraction, layering, modularization

2.4 OS Architecture and Kernel Variants

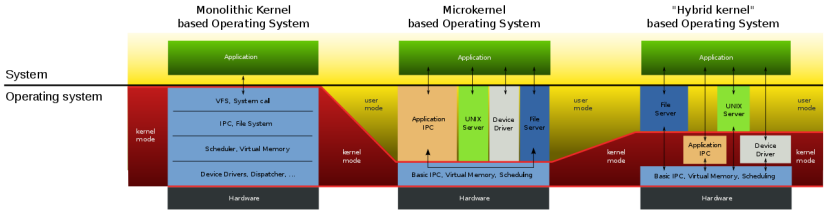

Figure 6: “Monolith-, Micro- and a "hybrid" kernel” under CC0 1.0; from Wikimedia Commons

This map of the Linux kernel provides a real-life monolithic example

This figure shows different approaches towards layering and modularization in the context of OS kernels. First of all, note the common layers, namely applications at the top and hardware at the bottom.

In between are different layers related to what we think of as OS functionality. Note that this OS functionality is marked with a red (left) and yellow (middle and right) background labeled “kernel mode” and “user mode”, respectively. These modes refer to different CPU privilege levels, which will be discussed in the next presentation; for now it is sufficient to know that code running in kernel mode has full control over the underlying hardware, while code running in user mode is restricted and needs to invoke lower layers (that run in kernel mode) for certain functionality.

At one extreme, shown in the middle here, are so-called *micro kernels*, which just provide the minimal functionality and services as foundation for full-fledged OSs. Typical functionality that we expect from OSs, such as file services or hardware independent network access, is then *not* implemented in the kernel but in user mode processes or servers. The L4 family mentioned later on as well as Fuchsia provide examples for micro kernels.

The other extreme is made up of so-called *monolithic kernels*, which provide (almost) everything that we expect from OSs. For modularization, such kernels may be structured in a sequence of layers, where the top layer provides the system call API to be explained on subsequent slides, while the bottom layer implements device driver abstractions to hide hardware peculiarities. Intermediate layers offer levels of abstraction on the way from hardware to application facing functionality. GNU/Linux and Windows come with monolithic kernels.

Finally, hybrid kernels (e.g., Windows NT) can be built as trade-off between both extreme approaches.

2.4.1 OS Kernel

- OS runs as code on CPU
  - Just as any other program
- **Kernel** contains central part of OS
  - Provides **API** for OS services via system calls (next slide)
  - Code and data of kernel typically main memory resident
  - Kernel functionality runs in kernel mode of CPU, reacts to system calls and interrupts

        * Details in next presentation
    - Variants (previous slide)
        * Monolithic ("large," all OS related services)
        * Micro kernel ("small," only necessary services)
        * "Best" design subject to research
            · Provable security only with micro kernels (seL4)

2.4.2 System Calls

- System call = function = part of kernel API
    - Implementation of OS service
        * E.g., process execution, main memory allocation, hardware resource access (e.g., keyboard, network, file and disk, graphics card)
- Different OSs offer different system calls (i.e., offer incompatible APIs)
    - With different implementations
    - With different calling conventions

2.4.3 Sample Microkernel: L4

- L4, developed by Jochen Liedtke, late 1990s
    - Liedtke's 4th system (after Algol interpreter, Eumel, and L3)
    - Now with family of L4 based kernels
    - Notable properties
        * **12 KB** source code
            · (Vs 918 KB for (heavily compressed) source code of Linux 1.0 in 1994)
        * **7** system calls
        * Abstractions: Address space, Threads, Inter-Process-Communication (IPC)
- Breakthrough result in 2009, [Kle+09]: **Formal verification** of the OS kernel seL4
    - Mathematical **proof** of correctness
        * Updates/patches are a **thing of the past**
    - More recent description in [Kle+14]

This slide contains some details about the micro kernel L4. First of all, note its size of 12 KB. In contrast, the (heavily compressed) source code of Linux 1.0 had a size of 918 KB in 1994 (which has grown to 113 MB for Linux 5.12 in 2021). Thus, 12 KB is really small for software, which in this case contains necessary kernel functionality regarding the creation of threads for multitasking as well as their memory use via address spaces and their communication.

The question of what constitutes a minimal OS kernel is not just an academic one. In fact, for smaller pieces of software we can hope to perform mathematical correctness proofs for their functionality. Indeed, a break-through result was achieved in 2009, when the correctness

of the L4 variant seL4 was formally verified. Please take a moment to think about this fact. Such software will **never** need to be patched to fix bugs. Bugs do not exist.

What I would like you to remember is that formally verified software exists, and it exists up to the complexity of micro kernels. Thus, if you should ever find yourself in a position where you are responsible for the correctness of software, say for autonomous devices or critical infrastructures, you should remember that the state-of-the-art makes it hard to find an excuse for buggy software and resulting system failures.

As stated on the next slide, L4 variants are actually deployed in billions of devices.

- L4 variants today
  - OKL4, deployed in over 2 billion devices
    - OS for baseband processor (modem, management of radio functions)
      - Starting with Qualcomm
    - Embedded, mobile, IoT, automotive, defense, medical, industrial, and enterprise applications
  - Another variant in Apple's Secure Enclave coprocessor (see PDF on this page)
    - A7 processor (iPhone 5S, iPad mini 3) and later
  - Airbus 350, Merkelphone
  - Project Sparrow based on seL4

3 Multitasking

3.1 Multitasking

- Fundamental OS service: **Multitasking**
  - Manage **multiple computations** going on **at the same time**
  - E.g., surf on Web while Java project is built and music plays
- OS supports multitasking via **scheduling**
  - Decide what computation to execute when on what CPU core
    - Recall: Frequently per second, time-sliced, beyond human perception
- Multitasking introduces **concurrency**
  - (Details and challenges in upcoming sessions)
  - Recall: Even with single CPU core, illusion of "simultaneous" or "parallel" computations
    - (Later presentation: Advantages include improved responsiveness and improved resource usage)

3.2 Computations

- Various technical terms for "computations": Jobs, tasks, processes, threads, ...
  - We use only **thread** and **process**
  - **Process**
    - Container for related threads and their resources
    - Created upon start of program and by programs (child processes)
    - Unit of management and protection (threads from different processes are isolated from another)
  - **Thread**
    - Sequence of instructions (to be executed on CPU core)
    - Single process may contain just one or several threads, e.g.:
      - Online game: different threads with different code for game AI, GUI events, network handling
      - Web server handling requests from different clients in different threads sharing same code
    - Unit of scheduling and concurrency

(Audio for this slide is split into several audio files, one for each step of the animation. In contrast, these notes contain a transcript of all animation steps.)

Among the various technical terms that can be used for the computations going on in our machines, we are only interested in **process** and **thread** as explained here and on subsequent slides. The specifics of processes and threads vary from OS to OS, and, in fact, some OSs may not know either of both notions. We only consider OSs that support multiple processes, each of which can contain multiple threads.

Roughly, when you execute a program, e.g., a Java program, your OS creates a process to manage computations and resources associated with that program. (As revisited later, the situation is more complex, though, as a single program can ask the OS (via system calls) to create lots of processes.)

Importantly, the OS isolates different processes from each other so that they are protected from malicious and accidental actions of other processes. (In theory, the crash of one process should not affect any other process; in practice, security issues usually violate this goal of isolation.)

In any case, when you start a program, the OS creates a process for that program, and it also creates a thread to execute the program's instructions. The programmer is free (to ask the OS via system calls) to create more threads that execute in the context of the same process and, thus, can share resources and data structures of their process. A later presentation will address how to create threads in Java, where you invoke functions of the Java API to create threads, which in turn are implemented with systems calls in the Java runtime.

The OS keeps track of all existing threads and schedules them for execution on CPU cores. Recall that scheduling usually involves time slicing, which leads to the illusion of a *parallel* execution of all threads (even on a single-core system) and which will be revisited in the presentation on scheduling.

3.2.1 Threads!

Warning! External figure **not** included: "Threads!" © 2016 Julia Evans, all rights reserved from julia's drawings. Displayed here with personal permission. (See HTML presentation instead.)

3.2.2 Process Aspects (1/3)

Warning! External figure **not** included: "What's in a process?" © 2016 Julia Evans, all rights reserved from julia's drawings. Displayed here with personal permission.
(See HTML presentation instead.)

3.2.3 Process Aspects (2/3)

- Approximately, process ≈ **running program**
  - E.g., text editor, game, audio player
  - OS manages lots of them simultaneously
- Really, process = "whatever your OS manages as such"
  - OS specific tools to inspect processes (research on your own!)

3.2.4 Process Aspects (3/3)

- Single program may create **multiple** processes, e.g.:
  - Apache Web server with "process per request" (MPM prefork)
  - Web browsers with "process per tab" or separation of UI and web content
    - E.g., Firefox with projects Electrolysis and Project Fission
- Many-to-many relationship between "applications" and processes
  - E.g., GNU Emacs provides lots of "applications"
    - Core process includes: Text editor, chat/mail/news/RSS clients, Web browser, calendar
      - Neal Stephenson, 1999: "emacs outshines all other editing software in approximately the same way that the noonday sun does the stars. It is not just bigger and brighter; it simply makes everything else vanish."
    - On-demand child processes: Spell checker, compilers, PDF viewer

3.3 Processes vs Threads

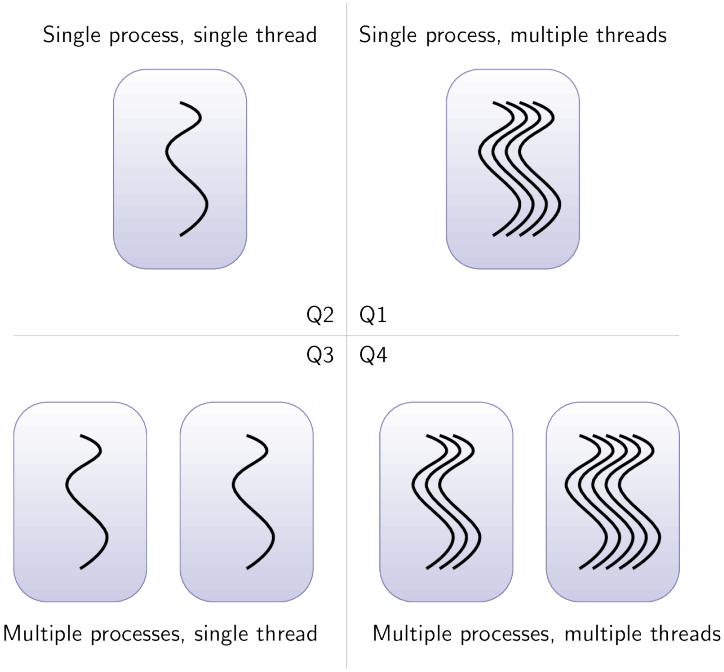

Figure 7: Classification of Processes and Threads from Anderson et al. (1997)

This figure shows a classification of platforms or execution environments for processes and threads from [And+97]. Note that although all threads are represented using the same curved line for graphical simplicity, each thread shown in the figure can actually execute its own instructions, independently from all other threads. Furthermore, although multiple threads are shown in parallel, *no* assumptions are made whether their instructions are really *executed* in parallel; clearly, parallel execution requires hardware support, e.g., in the form of multiple CPU cores, as well as OS support.

As shown in quadrant Q2, a platform may be characterized as supporting just a single process with a single thread, which effectively means that it has no notion of process or thread at all but just happily executes whatever instructions are there in one undifferentiated context. Thus, multitasking is *not* supported. Actually, the CS part of CSOS introduced one such platform ...

Q1 indicates multiple threads executing inside a single process, which may appear strange at first sight, but you actually also know one such execution environment quite well. You should not think about platforms consisting of hardware with OS here but about execution environments that can be started inside OSs ...

Q3 captures platforms with multiple single-threaded processes. Again, if everything is single-threaded, then the platform actually does not support threads, but just schedules processes for execution. This is mostly the case for older OSs.

Finally, Q4 contains multiple processes which in turn can host multiple threads. This is what we take for granted in upcoming OS sessions.

3.4 Exercises and Self-Study Tasks

3.4.1 Processes and threads

This task is available for self-study in Learnweb.

Sort sample OSs into the quadrants of Anderson et al.

- Hack, MS-DOS, Java Virtual Machine, Windows 10, GNU/Linux, GNU/Linux prior kernel 1.3.56, GNU/Linux starting with kernel 1.3.56
  - It is no problem if you do not know those environments and guess for this task
  - MS-DOS dates back to the 1980s, the GNU/Linux kernel 1.3.56 to 1996
    - Use educated guessing there ;)

3.4.2 Exercise Outlook: Bash Command Line

- Investigate The Command Line Murders among first OS exercises
  - Game, which teaches use of the Bash command line
  - Command line = shell = text-mode user interface for OS
    - Create processes for programs or scripts
  - Different shells come with incompatible features
    - Game supposes Bash in combination with typical GNU/Unix tools (e.g., `grep`, `head`, `tail`)
    - See next slide for some options
- Task
  - Access files for game
    - Download or clone with `git clone https://github.com/veltman/clmystery.git`
  - Start playing game according to its README
    - See next slide for hints
  - While investigating the case, you need to search files for clues, learning essential commands and patterns along the way
  - We will ask you to **submit** some command(s)
- (Command line examples show up throughout this course; details of file handling to be revisited in presentation on processes)

3.4.3 Using Bash as Command Line

- Where/how to start Bash as command line
  - Built-in with GNU/Linux; use own (virtual) machine
  - Alternatively, students reported success with Windows Subsystem for Linux/Ubuntu on Windows
  - Alternatives without Linux kernel (no or incomplete `/proc` for later presentations)
    - Maybe use Cygwin according to hints in game's cheatsheet, but note that more **students report problems** with Cygwin than with Windows Subsystem for Linux/Ubuntu mentioned above
    - Shell coming with Git for Windows
    - Terminal of macOS

- Basic hints for The Command Line Murders
  - Game's cheatsheet is misnamed; it contains **essential** information to get you started
    - Open in editor
  - Once on command line, maybe try this first:
    - `mount` to show filesystems, e.g., with Cygwin, the location of `C:` may be shown as `/cygdrive/c`
    - `ls` (short for "list") to view contents of current directory
    - `ls /cygdrive/c` to view contents of given directory (if it exists)
    - **Beware!** Avoid spaces in names of files and directories: Space character separates arguments (need to escape spaces with backslash or use quotation marks around name)
    - `pwd` (short for "print working directory") to print name of current directory
    - `cd replace-this-with-name-of-directory-of-mystery` (short for "change directory") to change directory to chosen location, e.g., location of mystery's files
    - `man name-of-command` shows manual page for `name-of-command`
    - Try `man man` first, then `man ls`
  - Afterwards, follow game's README
    - (Which supposes that you changed to the directory with the game's files already)

3.4.4 Feedback

- This slide serves as reminder that I am happy to obtain and provide feedback for course topics and organization. If **contents** of presentations are confusing, you could describe your current understanding (which might allow us to identify misunderstandings), ask questions that allow us to help you, or suggest improvements (maybe on GitLab). Please use the session's shared document or MoodleOverflow. Most questions turn out to be of general interest; please do not hesitate to ask and answer where others can benefit. If you created additional original content that might help others (e.g., a new exercise, an experiment, explanations concerning relationships with different courses, ...), please share.

4 Conclusions

4.1 Summary

- OS is software
  - that **uses hardware** resources of a computer system
  - to provide support for the **execution of other software**.
    - Computations are performed by threads.
    - Threads are grouped into processes.

- OS kernel
  - provides interface for applications and
  - manages resources.

Bibliography

[And+97] Thomas E. Anderson et al. "Thread Management for Shared-Memory Multiprocessors". In: *The Computer Science and Engineering Handbook*. Ed. by Allen B. Tucker. CRC Press, 1997. URL: https://homes.cs.washington.edu/~tom/pubs/threads.pdf.

[Hai17] Max Hailperin. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3, 2017. URL: https://gustavus.edu/mcs/max/os-book/.

[Hai19] Max Hailperin. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3.1, 2019. URL: https://gustavus.edu/mcs/max/os-book/.

[Kle+09] Gerwin Klein et al. "seL4: Formal Verification of an OS Kernel". In: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*. SOSP '09. Big Sky, Montana, USA: ACM, 2009, pp. 207–220. ISBN: 978-1-60558-752-3. DOI: 10.1145/1629575.1629596. URL: https://dl.acm.org/citation.cfm?doid=1629575.1629596.

[Kle+14] Gerwin Klein et al. "Comprehensive Formal Verification of an OS Microkernel". In: *ACM Trans. Comput. Syst.* 32.1 (Feb. 2014), 2:1–2:70. ISSN: 0734-2071. DOI: 10.1145/2560537. URL: https://dl.acm.org/citation.cfm?doid=2560537.

[TB15] Andrew S. Tanenbaum and Herbert Bos. *Modern Operating Systems*. 4th ed. Pearson, 2015.

License Information