# OS Introduction *

Jens Lechtenbörger

IT Systems, Summer Term 2025

This presentation is an introduction to Operating Systems.

## 1 Introduction

- Partially based on Chapter 1 of (Hailperin 2019)
  - Book available in Learnweb, LaTeX sources on GitHub
  - Tasks/quizzes and code examples in OS part may use bibliographic keys "(Hailperin 2019)" or "[Hai19]" to refer to this book (or "[Hai17]" for an earlier edition)

For Operating Systems, the textbook cited here is the main source. Note its bibliographic keys, which may serve as pointers to the book in various places.
As usual, references in presentations may indicate other sources.

### 1.1 Learning Objectives

- Explain notion of Operating System and typical services
  - Explain notion of kernel with system call API, user mode and kernel mode
    - (More details in next presentation)
- Explain notions and relationships of program, process, thread, multitasking
- Use the Bash command line: Navigate in directories, view lines of files, search for patterns, use redirection and pipelines

Recall that learning objectives specify what you should be able to do after working through a presentation and its tasks.
Take some time to think about the learning objectives specified here.

### 1.2 Recall: Big Picture of IT Systems

- Explore **abstractions bottom-up**

  In IT Systems, we explore abstractions bottom-up.

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

– **Computer Architecture**: Build computer from logic gates


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

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

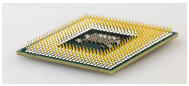

Figure 2: "CPU" under CC0 1.0; cropped and converted from Pixabay

> We already know how to build a von Neumann computer starting from Nand gates.

– **Experiment with OS concepts**
  * Explain core OS **management** concepts, e.g., processes, threads, virtual memory
  * Use GNU/Linux command line and explore system

Figure 3: "Tux, the Linux mascot" under CC0 1.0; from Wikimedia Commons

– **Experiment with containerization** for cloud infrastructures
  * Explain core concepts
  * Build images, run Docker **containers** and Kubernetes cluster

Figure 4: "Kubernetes logo" under Kubernetes Branding Guidelines; from GitHub

Figure 5: "Docker logo" under Docker Brand Guidelines; from Docker

In the next weeks, we look at major operating system concepts for the management of computer hardware.

In the final part of the course, we then explore concepts related to virtualization and cloud computing.

### 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. Some drawings come with longer explanations while others are meant to speak for themselves as additional perspective on class topics (or even beyond class topics).

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".

## Agenda

The agenda for the remainder of this presentation is as follows.

After this introduction, we look at sample operating systems, their definition, and general concepts. Afterwards, we introduce multitasking as central ability of an OS to execute multiple tasks at the same time, which may happen with processes or threads of execution.

Conclusions end the presentation.

# 2 Operating Systems

Let us look at 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 elsewhere, 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 (Hailperin 2019): **Software**
  - that **uses hardware** resources of a computer system
  - to provide support for the **execution of other software**.
- Towards these goals, OS provides API with services

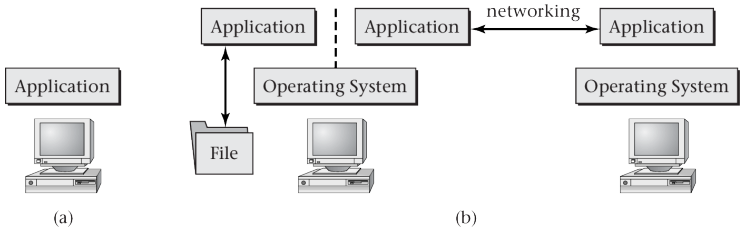


Figure 6: "Figure 1.1 of (Hailperin 2019)" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

According to our textbook, an OS is software that uses hardware resources of a computer system to provide support for the execution of other software.

Towards these goals, an OS provides an API with services for other software.

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 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.
- OS kernel provides system call interface for its services

Regarding wording, an API is an interface for application programmers. E.g., the language Java comes with an API.
In OSs, the kernel provides an API for its services in terms of system calls.

### 2.2.2 OS Services

- OS services/features/functionality defined by its API
  - Functionalities include:
    Different OSs provide different services.
    - Support for **multiple concurrent** computations
      - Run **applications**, divide hardware, manage state

      Nowadays we expect support for multiple concurrent computations, e.g., in the form of multiple running applications, each of which needs CPU time, memory, and potentially other hardware resources.
    - **Control interactions** between concurrent computations
      - E.g., locking, private memory

      If applications run concurrently, we may want to control their interactions, for example with private memory or with locking mechanism for shared data.
    - Files for **persistent** storage and interaction
      OSs usally offer file system functionality, allowing users and applications to manage, and possibly to share, data persistently on secondary storage.
    - Typically, also **networking** support
      With networking support, applications on different machines can communicate with each other.

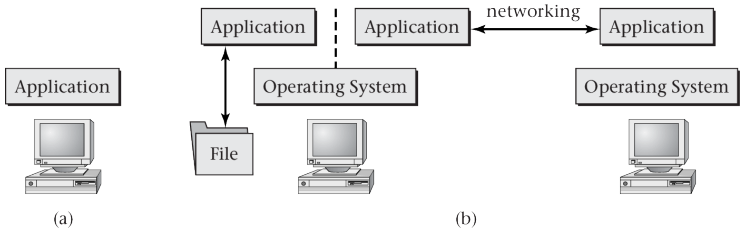


Figure 7: "Figure 1.1 of (Hailperin 2019)" by Max Hailperin under CC BY-SA 3.0; converted from GitHub

## 2.3 OS, Kernel, User Interface

- Boundary between OS and applications is fuzzy

  The boundary between OSs and applications turns out to be a fuzzy one. For example, is the web browser an integral part of the OS or not? A virus scanner? A tool to format disk drives?

- **Kernel** is fundamental, core part of OS

  - Kernel defines **API** and services via **system call** interface
  - More details on next slides and in next presentation

  The core part of an OS that defines its elementary services and system calls is called kernel. The kernel offers as an API with system calls for applications that require OS services.

- 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

  The user interface, in case it exists at all, is not part of the kernel. In contrast, the OS starts processes that make use of kernal functionality to provide an interface for users. Through that interface, users can then interact with the computer, e.g., to start and interact with applications.

  In this course, you will learn about a particular command line interface called Bash. Beyond that, we do not care much about different types of user interfaces, some of which are listed here.

### 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 introduced on the next slide.

### 2.3.2 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

A system call looks and feels like any other function call, only that the function provides an OS service. E.g., if you start an application, under the hood a system call creates a new process. If the application needs access to I/O devices, for example to access files on disk, system calls are executed.

Not surprisingly, different vendors implement their system call APIs differently. Details are not important for us.

### 2.3.3 User Space and Kernel Space

- CPU has privilege levels/rings/modes
  - Machine instruction set restricted depending on level **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.)
    * E.g., **4 rings** since Intel 80286
    * Ring 3: **User mode** for programs
      · I/O and memory access restricted
    * Ring 2, 1: Usually unused
      · Originally for system services and device drivers
    * Ring 0: **Kernel mode** for OS
      · Traditionally, most privileged
      · (Recall negative ring numbers)

This drawing explores a distinction between *user space* and *kernel space*.

Briefly, as you know from the OS Motivation, modern CPUs have different privilege levels, which may also be called rings, levels, or modes.

In these levels, different subsets of the entire set of machine instructions are available. E.g., in user mode, access to hardware is restricted, which implies that applications needing hardware access must invoke system calls into the kernel for hardware access. As the kernel runs at a more privileged level, it can then perform hardware access on behalf of the application. For example, the visualization here is about a write operation on a file, which requires hardware access via the `write` system call.

Based on these privilege levels, we distinguish user space from kernel space, where the former refers to "normal" applications, while the latter refers to the OS.

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.

## 2.4 OS Architecture and Kernel Variants

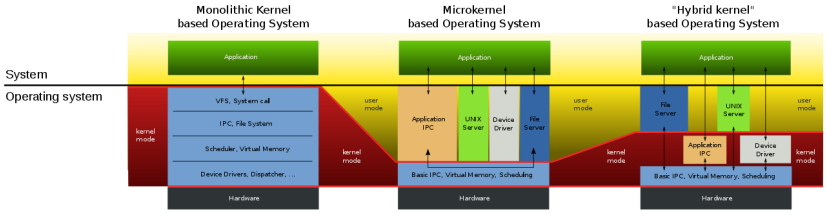


Figure 8: "Monolith-, Micro- and a "hybrid" kernel" under CC0 1.0; from Wikimedia Commons

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 a horizontal line separates applications from the OS. The variants differ in the amount of OS functionality that is marked with a red background labeled "kernel mode".

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 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, 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 can be built as trade-off between both extreme approaches. Research and practice continue to explore different variants.

### 2.4.1 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, (Klein et al. 2009)
  - **Formal verification** of the OS kernel seL4
    - Mathematical **proof** of correctness
      - Updates/patches are a **thing of the past**
    - More recent description in (Klein et al. 2014)

This slide contains some details about the highly influential 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 almost 1 MB in 1994 (which has grown to about 100 MB in the next 25 years for Linux 4.x in 2019). Thus, 12 KB is really small for software. This software contains necessary kernel functionality for the creation of threads, for memory management, and for 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. Indeed, a break-through result is cited here, where the correctness of an L4 variant 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 at least 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.

- 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

On a side note, L4 variants are actually deployed in billions of devices.

# 3 Multitasking

Multitasking refers to the ability of an OS to execute multiple tasks at the same time.

Technically, the system switches rapidly between different tasks, giving the illusion that the tasks are progressing simultaneously.

## 3.1 Multitasking Terminology

- 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

    Multitasking is a fundamental OS service. Multitasking is the management of multiple tasks that seem to be active at the same time. E.g., a student might surf on the Web, while a larger Java project compiles, and some application plays music.

    Clearly, with a single CPU core, only one machine instruction belonging to one of these tasks is executed at any point in time.

- OS supports multitasking via **scheduling**
    - Decide what computation to execute when on what CPU core
        * Recall: Frequently per second, **time-sliced**, beyond human perception

    The OS supports multitasking by scheduling the execution of multiple tasks or computations on available CPU cores. This scheduling process involves deciding which computation to execute at what specific CPU core and sharing CPU cores through time-slicing.

    A time slice is just a short period of time. With multitasking, each task takes a turn, running for a time slice on the CPU. Once that time slice ends, the OS schedules another task for execution, again for a time slice. This scheduling happens frequently per second, beyond human perception, leading to the seemingly simultaneous execution of multiple tasks.

- Multitasking introduces **concurrency**
    - (Details and challenges in upcoming material)
    - Recall: Even with single CPU core, illusion of "simultaneous" or "parallel" computations

    - (Later presentation: Advantages include improved responsiveness and improved resource usage)

Multitasking based on time-slicing introduces concurrency, where even with a single CPU core the illusion of simultaneous or parallel computations arises. We will investigate challenges and advantages in upcoming presentations, and a later drawing offers a preview.

## 3.2 Computations

- Various technical terms for "computations": Jobs, tasks, processes, threads, ...
  - We use only **thread** and **process**

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. Typical modern OSs support multiple processes, each of which can contain multiple threads.

  - **Process**
    - Created upon start of program and by programs (child processes)
    - Container for related threads and their resources
    - Unit of management and protection (threads from different processes are isolated from another)

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, as a single program can ask the OS (via system calls) to create lots of processes. If one process P creates another process C, then P is also called parent process of C, while C is a child process of P. That way, a hierarchy of parent-child processes arises over time.

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 challenge this goal of isolation.)

  - **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

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 system calls in the Java runtime.

The OS keeps track of all existing threads and schedules them for execution on CPU cores. This topic will be explored 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.)

As illustrated here, a process may contain lots of threads, all of which share memory.

Each thread can execute its own code.

Sharing of resources may lead to race conditions, which are programming bugs, where different threads may access or overwrite intermediate results of other threads. You know such problems as update anomalies in database systems. We will revisit race conditions and counter measures in later presentations.

Importantly, with multiple cores, multiple threads can execute in parallel, speeding up computations.

### 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.)

This drawing visualizes several aspects related to processes.

### 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!)

As a first approximation, you may think of a process as a running program. The OS is able to manage multiple processes, which leads to multitasking.

To be more precise, a process is whatever your OS manages as process. While details depend on the OS, the previous drawing offers some ideas, and the next slide provides examples.

### 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
      - Enter `about:processes` into address bar

  The view of one running program as one process is only an approximation as a single program is free to create multiple processes. (In fact, the code of the program then contains multiple system calls that ask the OS to create new processes.)

  As examples, web servers and web browsers often start multiple processes. E.g., with Firefox, open some web pages in different tabs, and enter `about:processes` into the address bar to see the amount of isolated processes.

- Many-to-many relationship between "programs" and processes

- E.g., GNU Emacs provides lots of "programs"
  - 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

In fact, the boundaries between programs may not be immediately visible, which can be understood as many-to-many relationship between programs and processes. E.g., your instructor's work environment, Emacs, is a program running as process which runs other programs with their processes for specific tasks.

Neal Stephenson has written these beautiful words about Emacs: "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."

## 3.3 Processes vs Threads

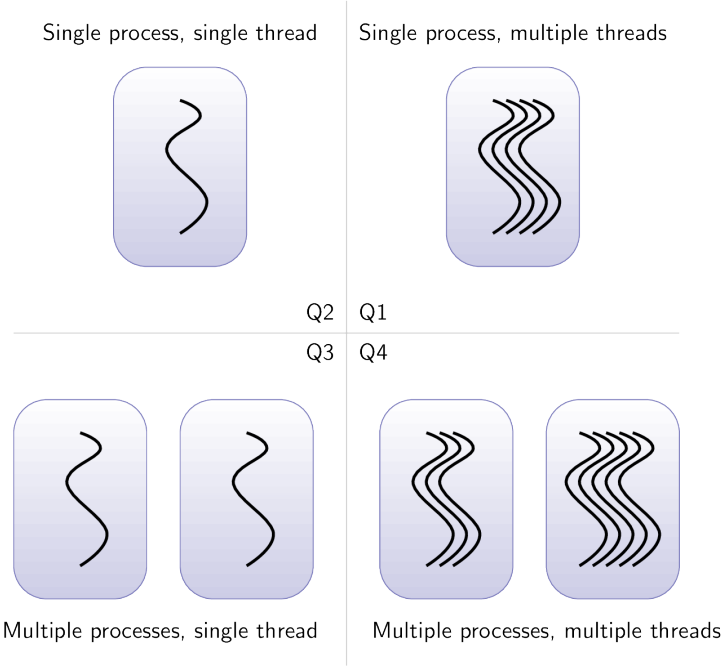


Figure 9: Classification of Processes and Threads from Anderson et al. (1997)

This figure shows a classification of platforms or execution environments for processes and threads. 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 Computer Architecture part of our course 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 subsequently.

# 4 Conclusions

Let us conclude.

## 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
  - runs in kernel mode of CPU,
  - provides interface for applications, and
  - manages resources.
- Micro kernels allow for correctness proofs.

An 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, and threads are grouped into processes, which represent running programs.

The code of the OS kernel runs with high privileges on the CPU. The kernel provides an interface with services for applications and manages hardware resources. Different kernel variants exist, among which micro kernels may come with correctness proofs.

## 4.2 Exercises and Self-Study Tasks

Please take a break for some self-study tasks.

### 4.2.1 Processes and threads

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 ;)

Sort the OSs mentioned here into the quadrants of a previous slide.
This task is available for self-study in Learnweb.

### 4.2.2 Study Work: Bash Command Line

- Investigate The Command Line Murders
  - 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)

As part of upcoming study work, investigate the Command Line Murders as instructed here.

### 4.2.3 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)

Investigating the Command Line Murders requires that you use a command line called Bash. This slide provides pointers to get you started.

## Bibliography

Hailperin, Max. 2019. *Operating Systems and Middleware – Supporting Controlled Interaction*. revised edition 1.3.1. https://github.com/Max-Hailperin/Operating-Systems-and-Middleware--Supporting-Controlled-Interaction.

Klein, Gerwin, June Andronick, Kevin Elphinstone, Toby Murray, Thomas Sewell, Rafal Kolanski, and Gernot Heiser. 2014. "Comprehensive Formal Verification of an Os Microkernel." *Acm Trans. Comput. Syst.* 32 (1): 2:1–2:70. https://doi.org/10.1145/2560537.

Klein, Gerwin, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, et al. 2009. "seL4: Formal Verification of an OS Kernel." In *Proceedings of the Acm Sigops 22nd Symposium on Operating Systems Principles*, 207–20. Sosp '09. Big Sky, Montana, USA: ACM. https://doi.org/10.1145/1629575.1629596.

The bibliography contains references used in this presentation.

# License Information