Introduction – Part 1: Operating systems & Virtualization – Some key concepts

Renaud Lachaize

Univ. Grenoble Alpes
M2 MoSIG & Phelma-SEOC
September 2025

What is an operating system? (1/3)

- An operating system (hereafter “OS”) is a piece of software (often made of different components/layers) that sits between the hardware and the applications.
- An OS is typically made of several software components/layers. The most critical component is called the **OS kernel**.
- An OS plays two main roles: **virtualization** and **resource management.**
- **Isolation** properties are a key requirement to support these two roles.

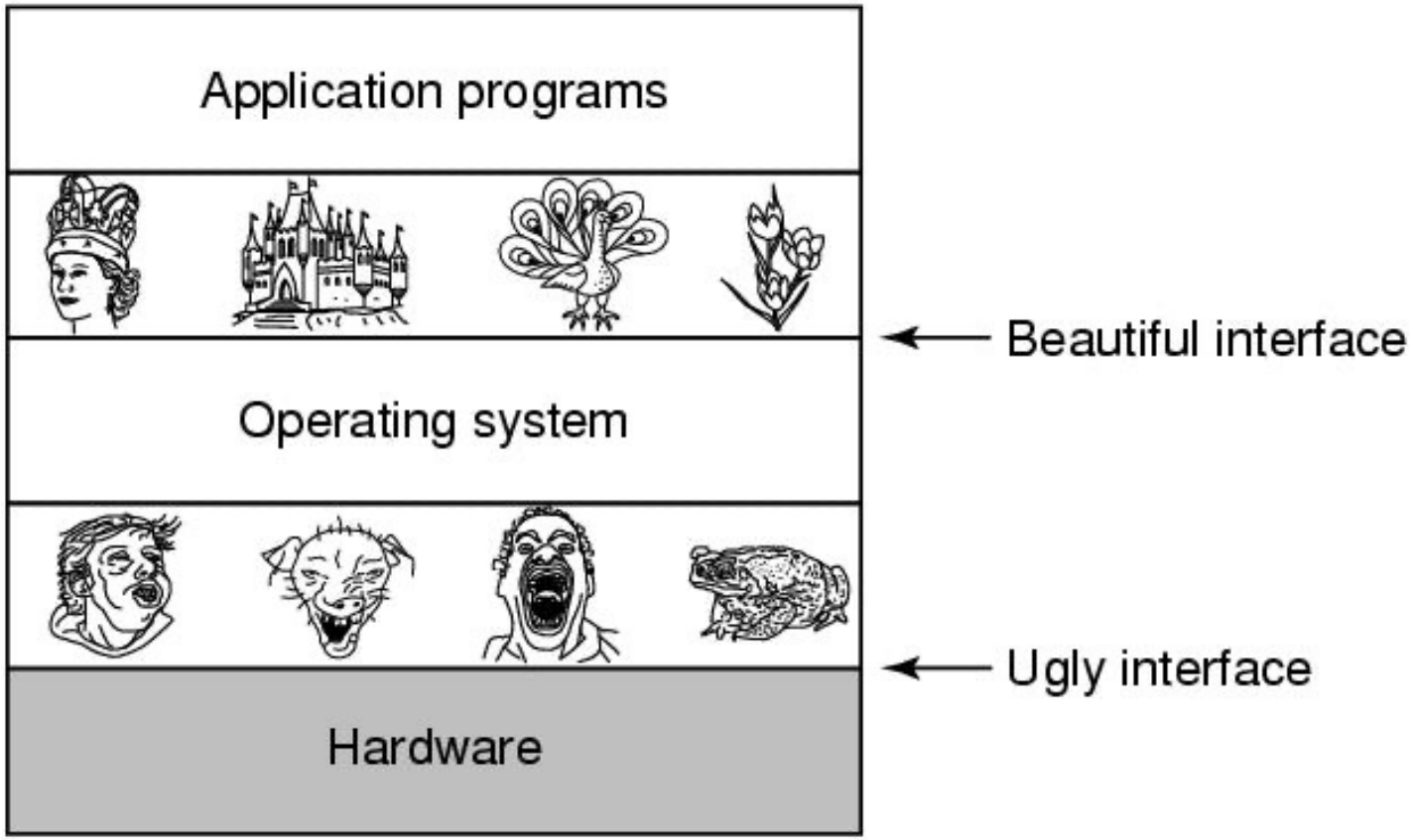
- **Warning:** Expressions such as **“virtualization”**, **“virtual machines”**, and **“isolation”** are used for many purposes in computer science/engineering. Arguably, there is no standard/unique/homogeneous definition for these concepts, so the precise intended meaning is very dependent on the context.

What is an operating system? (2/3)

OS role #1: Virtualization

The OS makes it easier to write/run/manage applications on a machine, by **hiding complexity via abstractions**. This includes:

- **Hiding** the **physical resource limitations** of a machine
- **Hiding** the **differences between machines**
  - Regarding the amount of resources (e.g., number of CPU cores, RAM capacity, …)
  - Regarding the diversity of devices (e.g., disk models and low-level interfaces)
- **Hiding** the **sharing of hardware resources** between concurrent applications/users
- **Hiding** the **low-level interface of the hardware**; replace it with higher-level concepts



Source: Andrew Tanenbaum. Modern operating systems. Pearson education.

What is an operating system? (3/3)

OS role #2: Resource management

The OS is in charge of managing the resources of a computer system, so that applications can **run safely, securely, efficiently, fairly [...]** despite the fact that they run concurrently.

- Encompasses **diverse types of resources**
  - **Physical resources**: CPUs, main memory, devices …
  - **Logical resources**: programs/code, data, communications, execution flows, …
- Encompasses **several aspects**
  - **Allocation**, **sharing**, **protection**
  - Concurrent execution implies **competition** and also possibly **cooperation**, which require **synchronization**
- Internally, an OS is mainly structured as a set of **mechanisms** and **policies**

The Hardware/Software interface

Key elements:

- **CPU Instruction Set Architecture (ISA)**, including also:
  - The definition of privileged and unprivileged execution modes
  - The support for hardware **exception/trap** handling (e.g., syscalls, page faults, …) & **hardware interrupt** handling
- Memory management unit (**MMU**) – a per-CPU hardware unit supporting memory virtualization
- Memory controller
- **Devices**
  - Timer, disks, network interfaces, accelerators (e.g., GPUs), peripherals (e.g., keyboard), …
- **I/O controllers**, allowing interactions between CPUs/memory and devices, via:
  - Memory-mapped registers ("MMIO": memory-mapped I/O)
  - DMA (Direct Memory access) engines
  - Interrupts
- **Interrupt controllers** (ICs)
  - Local (per-CPU) controllers + global (I/O) controller
  - Manage device interrupts and also inter-processor interrupts (IPIs)

The OS interface (1/3)

- An operating system typically provides **two kinds of interfaces**:
  - A **command/user interface** aimed at humans
  - **[Our focus here]** **A programmatic interface** aimed at application code (including the programs implementing the user/command interface).

- The OS programming interface is composed of a set of procedures/functions
  - Including **libraries** and **system calls**
  - Defined at two levels:
    - **Source code**: Application Programming Interface (**API**)
    - **Machine/binary code**: Application Binary Interface (**ABI**)

The OS interface (2/3)

Case study: Linux

- Several key libraries (**libc**, libpthread, libm, librt, …) + many other ones
- **System calls**: **~400** (as of 2025)
  - https://man7.org/linux/man-pages/man2/syscalls.2.html
  - https://syscalls.mebeim.net/
- **API**: In C, with bindings for various other languages
  - https://man7.org/linux/man-pages/index.html
  - Main libraries: see Man Pages – Section 3
  - Syscalls: see Man Pages – Section 2
- **ABI** – Calling conventions (among many other aspects in the ABI)
  - Intel/AMD x86-64:
    - https://waynestalk.com/en/x86-64-calling-conventions-en/
    - https://en.wikibooks.org/wiki/X86_Assembly/Interfacing_with_Linux
  - Arm64:
    - https://johannst.github.io/notes/arch/arm64.html
    - https://peterdn.com/post/2020/08/22/hello-world-in-arm64-assembly/
    - https://www-inf.telecom-sudparis.eu/COURS/CSC4508/public/C4/C4-book.html

The OS interface (3/3)

Case study: Linux (continued)

For **full specifications/standards**, see also:

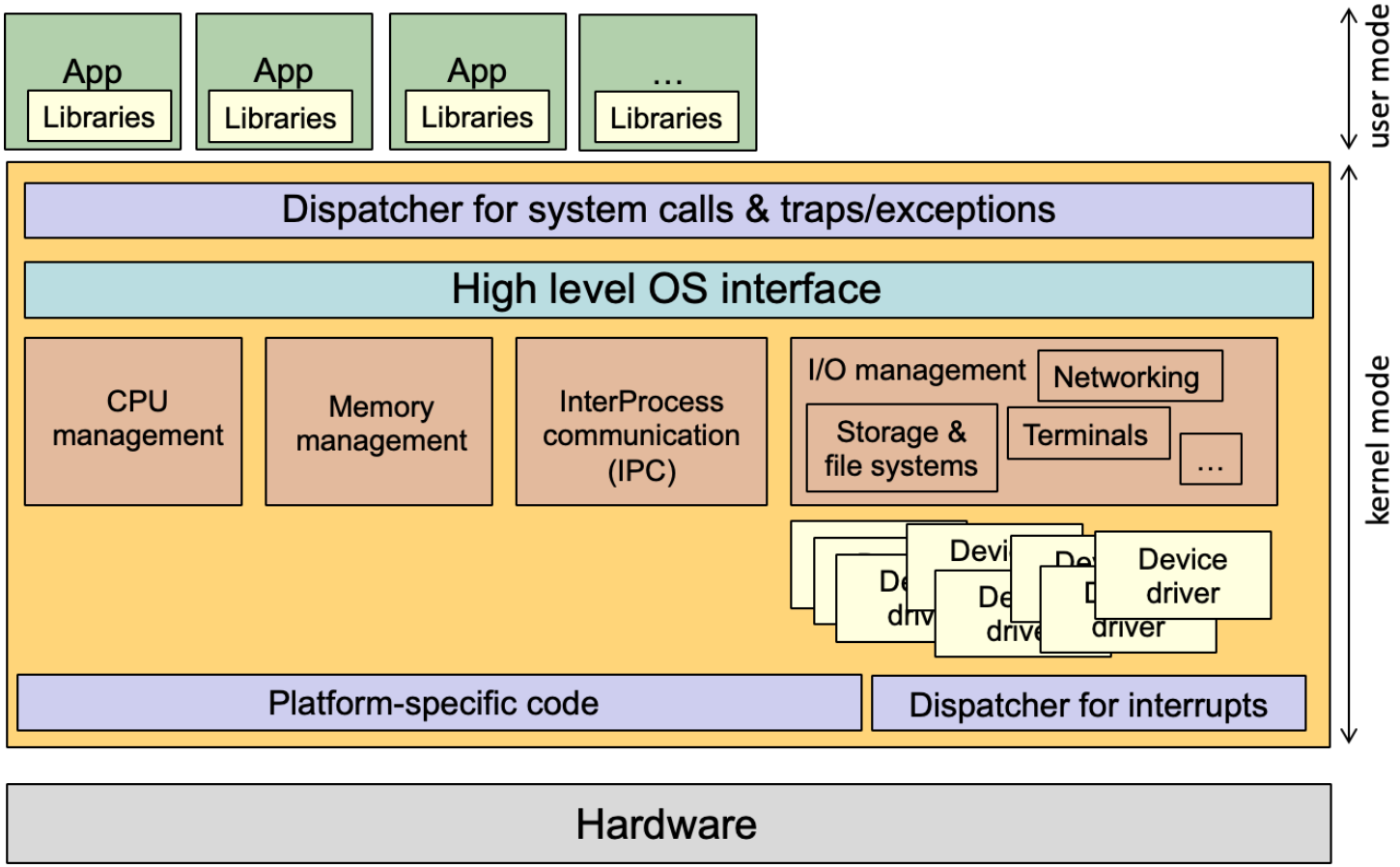
- https://refspecs.linuxfoundation.org
- Base/generic ABI:
  - Initial document: https://refspecs.linuxfoundation.org/elf/gabi41.pdf
  - Partial updates: https://www.sco.com/developers/gabi/latest/contents.html
- Intel/AMD x86-64 ABI
  - https://gitlab.com/x86-psABIs/x86-64-ABI
- Arm ABI
  - https://github.com/ARM-software/abi-aa

OS kernel designs

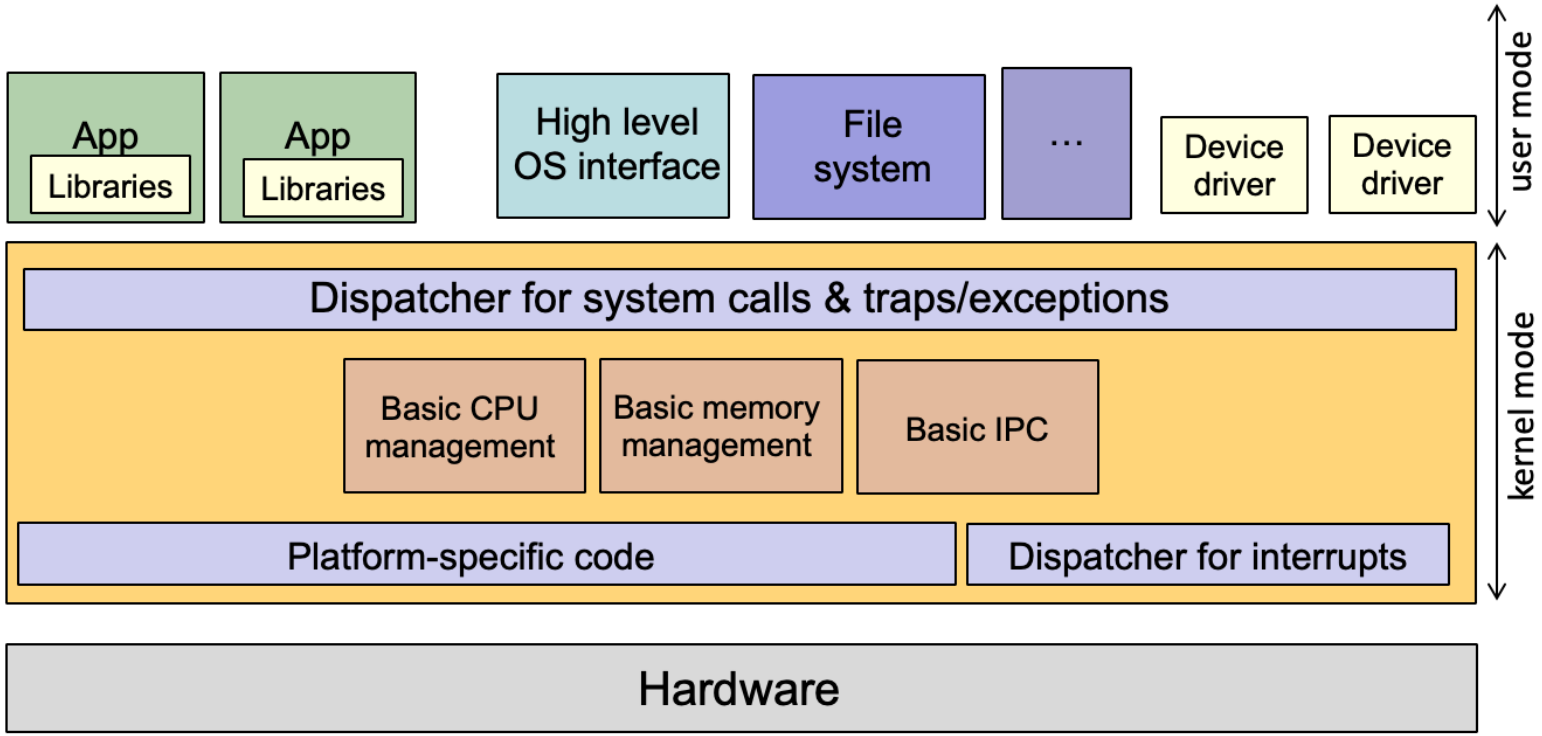
In terms of software architecture, there are **two main types of designs** for OS kernels:

- **Monolithic kernels**
  - Examples: Linux, Windows
  - Rich feature set
  - Optimized for fast interactions between OS subsystems
  - Large code base (~Millions of lines of source code)
  - Extensible via (dynamically) loadable kernel modules (e.g., device drivers)
  - Prone and vulnerable to bugs/attacks
  - (The Linux kernel also supports extensions via safe/restricted programs executed by the BPF interpreter.)
- **Microkernels**
  - Examples: L4 family (e.g., seL4), Zircon
  - Minimalist feature set – Most features externalized as user-level services
  - Optimized for reliability
  - Small code base (down to ~10,000 lines) – amenable to formal verification

Monolithic OS kernel design (e.g., Linux)



Microkernel design (e.g., L4)



Isolation

- To fulfill its key roles (virtualization and resource management), an OS must define and enforce a set of **isolation guarantees**.
- This requirement is typically managed in a hierarchical way, involving facilities (mechanisms and policies) at different levels of the hardware/software stack
  - ... And possibly also/beyond the OS (i.e., including middleware and application layers)
- At large, isolation guarantees encompass a **wide and diverse spectrum of concerns** (see details on next slides):
  - Software configuration and dependency management
  - Performance and Quality of Service (QoS)
  - Safety/security/confidentiality of physical and logical resources
- The **above concerns are not always orthogonal**. Examples:
  - Denial of Service (DoS)
  - Side-channel attacks

Isolation aspect #1: Software configuration

- **Main idea:** **Enable standalone (self-sufficient) packaging of an application with all its dependencies** ("software appliance"), including:
  - Libraries & utility programs
  - Configuration files
  - Configuration parameters (e.g., TCP port numbers)

- **Goal:** **Abstract away potential compatibility issues** w.r.t. the host platform, including:
  - Missing dependencies
  - Clashes w.r.t. software versions/configuration needed by host platform and/or other applications
  - Clashes w.r.t. system resources needed by host platform and/or other applications (examples: TCP server ports, file/directory names)
  - Lack of sufficient privileges to modify host configuration

Isolation aspect #2: Performance

- Many server applications must provide guarantees regarding their expected **quality of service (QoS)**, defined by **service level objectives (SLOs)** based on important metrics (for example, request throughput & tail latencies).

- **Concurrent applications** (or concurrent tasks within the same application) compete for hardware/software resources and may **interfere** with each other.

- **Performance isolation facilities** allow system administrators to define and implement policies (at various abstraction levels) to:
  - Separate applications/users into different classes (with distinct priorities, goals, quotas, …)
  - Limit interference between tasks
  - Enforce resource allocation limits (upper and lower bounds)

Isolation aspect #3: Safety & security

- **Goal:** **Define and enforce access control boundaries/rules** to **prevent unauthorized (accidental or malicious) accesses/operations** to physical and logical resources.

- These boundaries can be placed **at multiple levels of the hardware/software stack**, and target **various kinds of resources and operations**:
  - Memory accesses
  - Low-level resources such as physical devices (e.g., disks, network interfaces)
  - Including higher-level resources such as data, metadata, code
  - Flows such as control/execution flows (e.g., tasks/threads/processes), communication flows (e.g., via facilities like pipes and sockets), operations (e.g., service requests, event notifications)

Isolation – Multiple scopes

- Between the **"host" hardware/software infrastructure** and the **"guest" code**
  - In general, the host infrastructure is fully trusted.
  - However, recent infrastructures also support some scenarios in which the guest code does not fully trust the host infrastructures (e.g., concepts such as "security enclaves" and "confidential computing").
- Between applications/**processes** (or process groups)
- Between distinct users/**tenants**
- Between **compartments** within a given application

- Isolation (esp. for performance, safety and security) is typically **enforced at multiple places** (and using different mechanisms) in the hardware/software stack. This **multi-level approach** typically improves reliability through better **fault containment**.

Trusted Computing Base (TCB)

In general, the **vulnerability/fragility** of a hardware/software computer system regarding bugs and security issues is correlated with the size/complexity of its TCB.

Some definitions (sources: NIST, Wikipedia):

- "The trusted computing base of a computer system is the **set of all hardware, firmware, and/or software components that are critical to its security**, in the sense that bugs or vulnerabilities occurring inside the TCB might jeopardize the security properties of the entire system."
- "By contrast, parts of a computer system that lie outside the TCB must not be able to misbehave in a way that would leak any more privileges than are granted to them in accordance to the system's security policy."
- "The careful design and implementation of a system's TCB is paramount to its overall security. **Modern systems strive to reduce the size of the TCB** so that an exhaustive examination of its code base (by means of manual or computer-assisted software audit or program verification) becomes feasible."

Virtualization – Some definitions (1/3)

- As mentioned before, “virtualization” is a somewhat vague expression, with a diverse set of **context-dependent definitions** (introducing major and more subtle nuances, sometimes conflicting).
- Nonetheless, in many contexts, virtualization is achieved through a small set of common principles.

- In modern computer systems, virtualization is **typically performed in several layers of the hardware/software stack**:
  - **Hardware**: CPU, controllers, I/O devices
  - **Software**: Operating system (low-level & high-level layers), middleware, application libraries

Virtualization – Some definitions (2/3)

“Virtualizing a resource means **providing an abstract interface for it**, hiding its actual implementation, managing the allocation of the underlying (physical) elements.”

Sacha Krakowiak (with some modifications)

“A program that virtualizes a physical object **simulates the interface of the physical object using techniques such as multiplexing, aggregation and emulation**. For the user of the simulated object, it provides the same behavior as a physical instance. A primary goal of virtualization is to **preserve an existing interface**.”

Jerome H. Saltzer & M. Frans Kaashoek (with some modifications)

Virtualization – Some definitions (3/3)

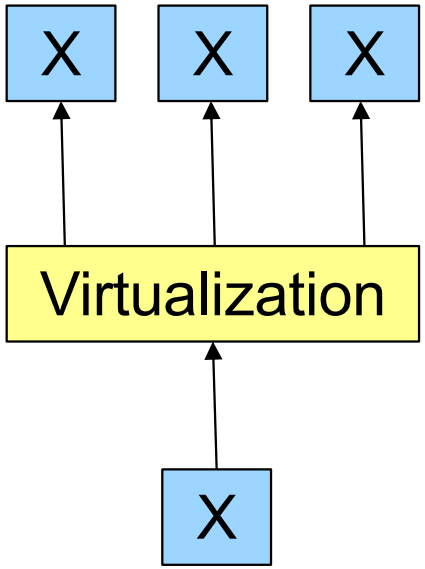
"Virtualization is the **application of the layering principle through enforced modularity**, whereby the exposed virtual resource is identical to the underlying physical resource being virtualized."

"**Layering** is the **presentation of a single abstraction**, realized by **adding a level of indirection**, when (i) the indirection relies on a single lower layer and (ii) uses a **well-defined namespace to expose the abstraction**."

"**Enforced modularity** additionally **guarantees that the clients of the layer cannot bypass the abstraction layer**, for example to access the physical resource directly or have visibility into the usage of the underlying physical namespace."

Edouard Bugnion, Jason Nieh, Dan Tsafrir

Virtualization – Some key principles (1/5)

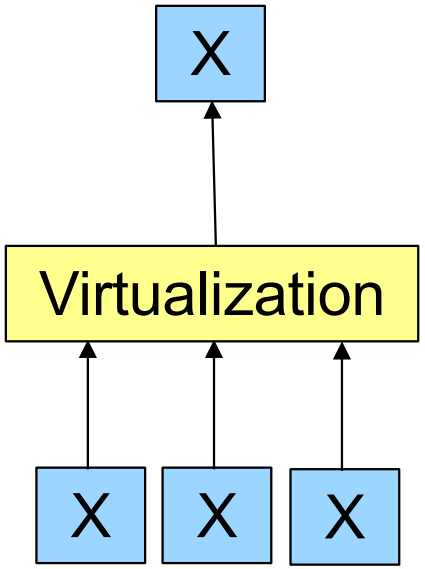


Multiplexing

- **Definition:**
  - Multiplexing exposes a resource among several virtual entities.
  - **Two types** of multiplexing: **in time** (i.e., time sharing) and/or **in space** (i.e., partitioning).
- **Examples:**
  - Resources managed by an OS:
    - CPU(s), physical memory, disk, network interface
  - Self-virtualizing hardware devices
    - E.g., network interface with SR-IOV

Inspired by: Saltzer & Kaashoek, Bugnion & Nieh & Tsafrir

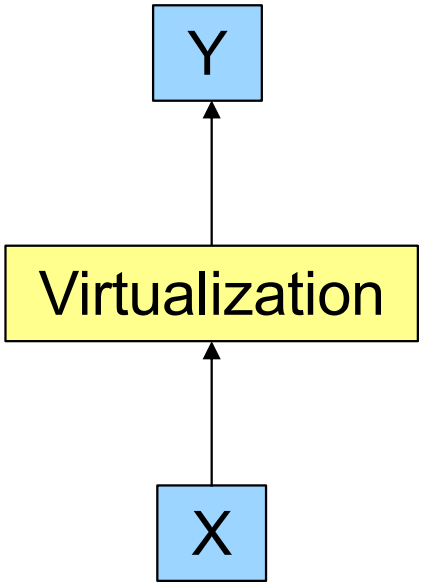
Virtualization – Some key principles (2/5)



Aggregation

- **Definition:**
  - Aggregation combines multiple resources and makes them appear as a single abstraction.
  - In essence, this is **the opposite of multiplexing**.
  - Resources can be aggregated **in space (often)** and/or possibly **in time**.
- **Examples:**
  - Hardware: memory controller combining multiple DIMMs into a single physical memory address space.
  - Hardware/Software: RAID storage controller.
  - OS migrating a task from high-performance to low-power CPU core.

Virtualization – Some key principles (3/5)



Emulation

- **Definition:**
  - Emulation allows exposing a virtual resource (or device) that is not present in the underlying system, or that does not have the same interface as the underlying system's resources.
  - There are **various degrees of heterogeneity** between the exposed resources (Y) and the underlying ones (X).
- **Examples:**
  - An OS can use main memory to emulate a disk. And vice versa.
  - An emulator (software-based or less often hardware-based) enables backward compatibility for code requiring a specific CPU ISA and/or specific I/O devices.
  - A traditional OS typically exports higher-level abstractions built above low-level hardware interfaces (e.g., a file system on top of a block device).

Virtualization – Some key principles (4/5)

The high-level principles previously discussed (multiplexing, aggregation, emulation) themselves rely on **some key lower-level paradigms**:

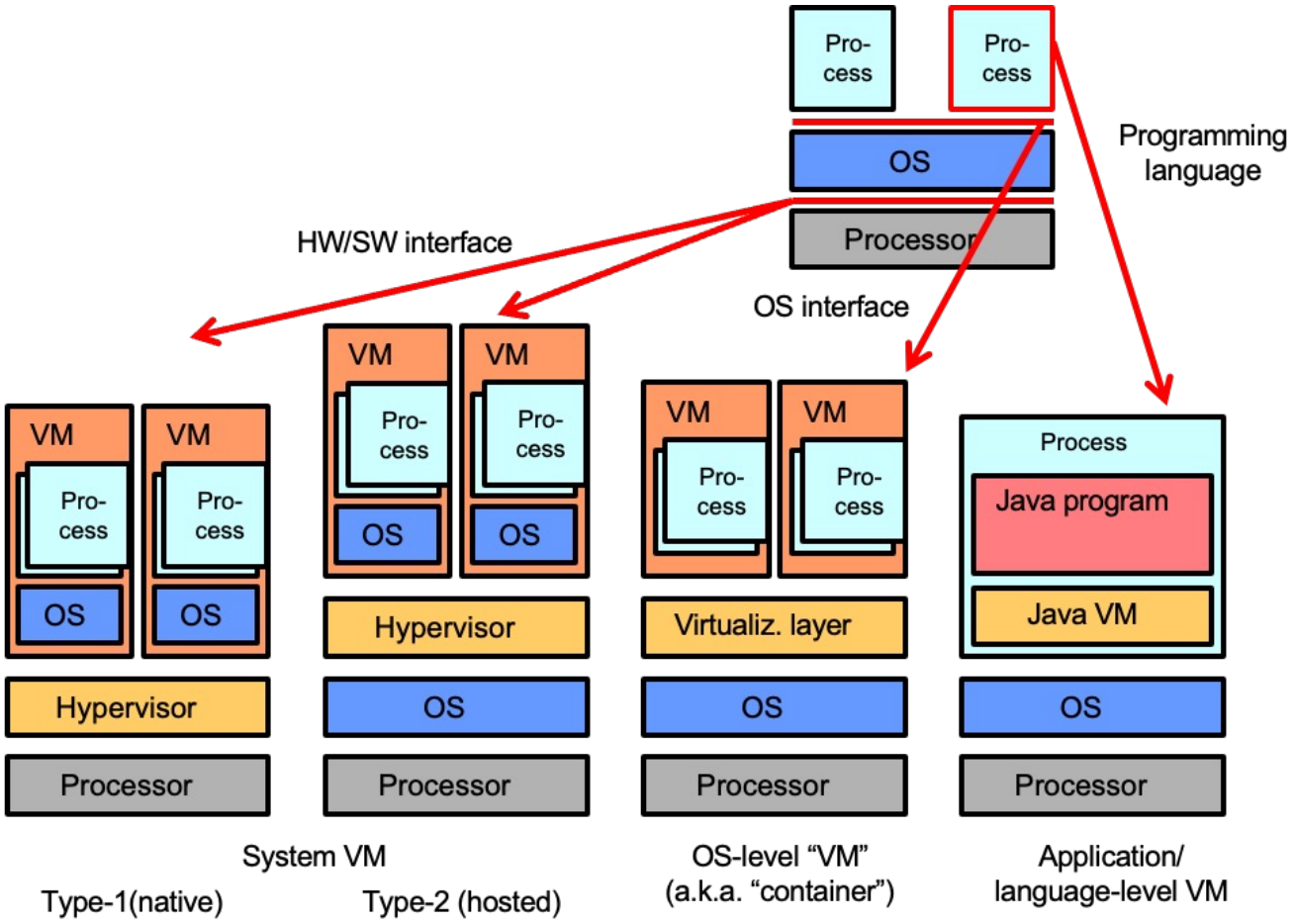
- **Naming**: Mapping a name (from a given namespace) to a thing (an "object").
- **Binding**: From a name (in a specific context), giving access to an object. When a binding exists, the name is said to be "bound".
  - **A binding may occur at any time**, up to when the name is resolved (late/dynamic binding) and may be reconfigured over time.
  - Names are also objects. So resolving a high-level name may involve a chain of bindings.
  - More generally, "binding" means choosing a low-level implementation/value for a high-level feature.

Virtualization – Some key principles (5/5)

The high-level principles previously discussed (multiplexing, aggregation, emulation) themselves rely on **some key lower-level paradigms**:
[CONTINUED]

- **Indirection**: Decoupling a connection between an object A and another object B by using an “abstract” name. Through the binding of that name, it is possible to delay (and/or later change) the choice of the target object B.
- **Interposition**: Leveraging indirection allows inserting an additional layer P (proxy) between two existing layers A and B. This allows P to intercept and modify the flow of operations between A and B.
- **Namespace isolation**: Indirection also allows using specific (distinct / contextual) namespaces for name resolution. In other words, resolving a given name X may yield a different result depending on which entity is using this name.

Types of virtualization



(Figure courtesy of Gernot Heiser, UNSW Sydney)

System VMs & containers – Timeline

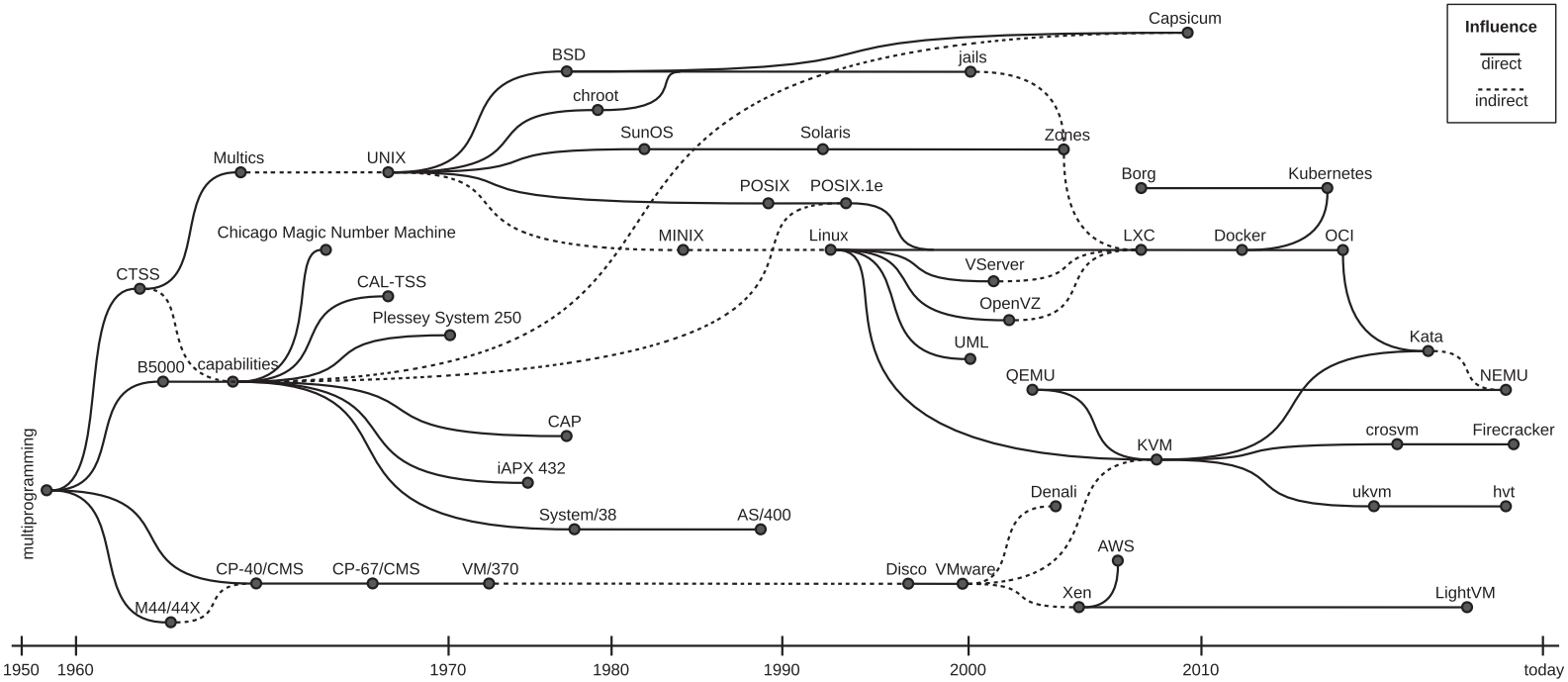


Fig. 1. The evolution of virtual machines and containers.

Source: Allison Randal. The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers. ACM Computing Surveys. 2020.

Virtualization – Main use cases

- **Main context: Cloud/Edge computing**
  - **Server/workload consolidation** → Isolation requirements (see previous discussion)
  - Flexible, reproducible and fast application deployments
  - Elasticity (horizontal & vertical scaling)
  - Live migration
  - State capture (suspend/resume/rollback)

- **Also in other domains**, at least for isolation purposes:
  - Software development and testing (DevOps, CI/CD, debugging, fuzzing, …)
  - Legacy software management: backwards compatibility and archival
  - Mobile/embedded computing (smartphones, cars, …)

System-level virtual machines (1/4)

- *A (system-level) virtual machine is an **efficient & isolated duplicate** of a real (physical) machine*
- Often abbreviated as **"VMs"** or **"Guest VMs"** or **"Guests"**
- Machine **resources** include CPU(s), main memory (RAM), I/O devices (disks, NICs, peripherals …)
- Goals:
  - **"Duplicate":** code running in a VM cannot distinguish between real or virtual hardware
  - **"Isolated":** Several VMs execute concurrently on the same machine without interfering with each other (at least w.r.t. safety and security considerations)
  - **"Efficient":** VMs should execute at a speed close to that of real hardware

System-level virtual machines (2/4)

The resources exported by a virtual machine may or may not correspond to the ones of the underlying physical hardware.

Typically, in practice, on a server/desktop machine:

- **Regarding the functional interface:**
  - The VM exports the same CPU model as the one of the physical machine (*same ISA: Instruction Set Architecture – such as Intel/AMD x86-64*).
  - Some ISA features (available for the physical machine) may be disabled for the VM.
  - The VM may or may not export the same types of I/O devices as the one of the physical machine.
- **Regarding the amount of available resources:**
  - A VM typically exports fewer resources than the total of physical resources.
  - Two main reasons:
    - Virtualization overhead
    - Concurrent execution of multiple VMs (with decent performance)

System-level virtual machines (3/4)

- The software layer in charge of supporting (system-level) virtual machines is called a **"Hypervisor"** or a **"Virtual Machine Monitor"**.
  - **Warning:** These two expressions are often use interchangeably. However, in some designs/documents, they correspond to different parts of the virtualization system.

- In order to achieve better virtualization performance (i.e., lower overhead), nowadays **most hypervisors partially rely on support from the physical hardware**.
  - For CPU and main memory virtualization
  - Also possibly for some types of I/O devices (e.g., high-speed network interfaces)

System-level virtual machines (4/4)

Warning: Beware of confusions between the concept of “hypervisor” and the other types of software layers listed below.

- **Language-level virtual machines:**
  - Examples: Java Virtual Machine (JVM), Python VM
  - Provide higher-level abstractions
  - Generally rely on a standard OS interface
- **Emulators:**
  - Conceptually closer to hypervisors
  - But with significant differences between the physical and virtual hardware (e.g., different ISA)
  - Mostly used for backwards compatibility or cross-development/testing
- **Hardware simulators:**
  - Aimed at precisely modelling the internal behavior of computer hardware (CPUs and/or devices)
  - Very different trade-off in terms of speed vs. accuracy
  - In contrast, hypervisors are only focused on externally-visible behavior of the hardware resources.

Strong security?? (1/3)

One of the main motivations for leveraging virtualization technologies is to obtain strong security guarantees while sharing hardware resources between multiple applications/tenants.

However, current hardware/software systems also exhibit **significant weaknesses** in their designs and implementations:

- Large code bases & TCB
- Vulnerability to side channel attacks

Strong security?? (2/3)

- **Large code bases & TCB**
  - Up to 100,000s or even millions of lines of code!
  - Often combining hardware/software components from multiple vendors
  - Large attack surface at various levels (Hw/Sw interface, OS ABI)
  - Many security-critical bugs
  - Software supply chain security is becoming a major concern

- **Vulnerability to side channel attacks**
  - Such attacks hijack features of modern CPUs to bypass security isolation mechanisms, thus allowing a malicious guest to access private/sensitive information from the host or other tenants (e.g., passwords, cryptographic keys, disk/network buffers …)
  - Example: “Variants of L1TF are especially troublesome for virtual machines, because they allow an unprivileged process in the user space of a guest to access any memory on the physical machine, including memory allocated to other guests, the host operating system, and host kernel.”
  - New attack vectors/variants are appearing on a frequent basis

Strong security?? (3/3)

How to deal with these issues?

Unfortunately, no simple/perfect/universal solutions … but some guidelines:

- Clearly specify and (re)assess the **trust/threat models**
- Carefully manage the **software supply chain & configuration**
- Understand **fundamental trade-offs**:
  - Rich feature set vs attack surface
  - Streamlined performance vs multi-level containment
  - Consolidation density vs blast radius
- Regarding **side channel attacks**:
  - Temporary case-by-case mitigations
  - More comprehensive solutions will require deeper hardware/software changes

General references about operating systems

- Remzi Arpaci-Dusseau & Andrea Arpaci-Dusseau. **Operating Systems: Three easy Pieces**. 2023 (Version 1.0 or above). Freely available from https://ostep.org

- Andrew Tanenbaum & Herbert Bos. **Modern Operating Systems**. Preferably 4th edition (2014) or above. Pearson Education.
  - **Also features a chapter about "virtualization and the Cloud"**, including system virtual machines (+ OS containers in 5th edition).

References about virtualization

- **A summary/overview about the history of containers and virtual machines:**
  - Allison Randal. **The Ideal Versus the Real: Revisiting the History of Virtual Machines and Containers**. ACM Computing Surveys. February 2020. https://dl.acm.org/doi/10.1145/3365199

- **Main textbook about system virtual machines:**
  - Edouard Bugnion, Jason Nieh, Dan Tsafrir. **Hardware and software support for virtualization.** Morgan & Claypool. 2017.

References about side channel attacks

- Note: This is an interesting topic but beyond the scope of this course. The references below are only for the curious readers.

- Jiliang Zhang, Congcong Chen, Jinhua Cui, Keqin Li. **Timing Side-channel Attacks and Countermeasures in CPU Microarchitectures.** ACM Computing Surveys. April 2024.

- Gernot Heiser. **For safety's sake: we need a new hardware-software contract!** IEEE Design and Test, Volume 35, Issue 2, pp. 27-30, March 2018. https://trustworthy.systems/publications/full_text/Heiser_18.pdf