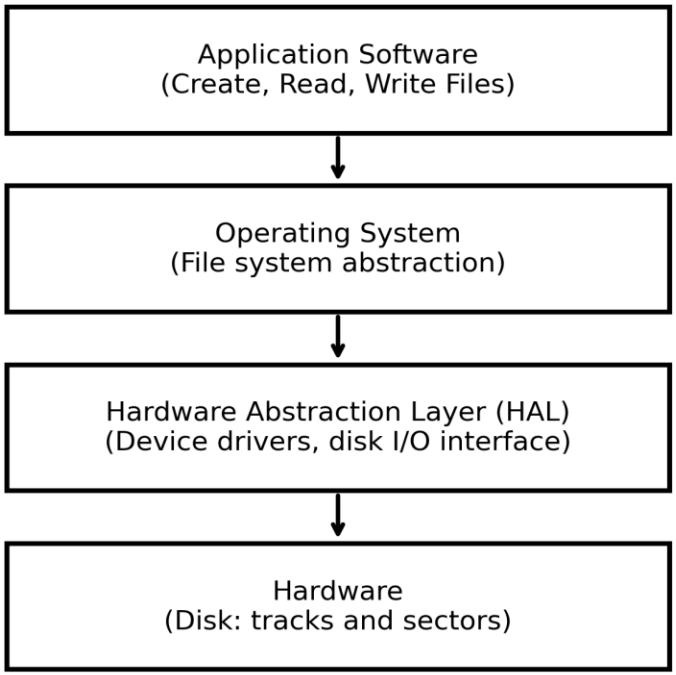
Detailed Introduction to Virtual Machines
Chapter 1 of Smith and Ravi

Abstraction and Complexity Management

- Modern computer systems are highly complex and require structured design to remain manageable.
- The technique is the division of the system into multiple levels of abstraction.
- These are separated by well-defined interfaces that regulate interaction between components.
- Abstraction allows designers to ignore or simplify lower-level implementation details, thereby simplifying higher-level system design.

Abstraction Hierarchy and Disk Example

- Hierarchical structure.
- Hardware-level components have properties.
- Software-level components are logical.
- Application programmers can create, read, and write files without knowledge of the physical disk organization.



Virtual Machines

- Virtualization provides a way of relaxing the foregoing constraints, increasing flexibility and makes a real system appear to be a set of virtual systems
- **One-to-many virtualization**
  - E.g. one physical machine may appear as multiple virtual machines
  - one physical disk may look like multiple virtual disk
  - one physical network may look like multiple virtual networks

Virtual Machines

- **Many-to-one virtualization**
  - Many physical machines/disks/networks may appear to look like one virtual machine/disk/network etc.
- **Many-to-many virtualization**
  - Extend the above statements.

Virtual Machines

- Virtual Machines are:
- Logical/Emulated representations of full computing system environment
- CPU + memory + I/O
- Implemented by adding layers of software to the real machine to support the desired VM architecture.

Virtual Machines

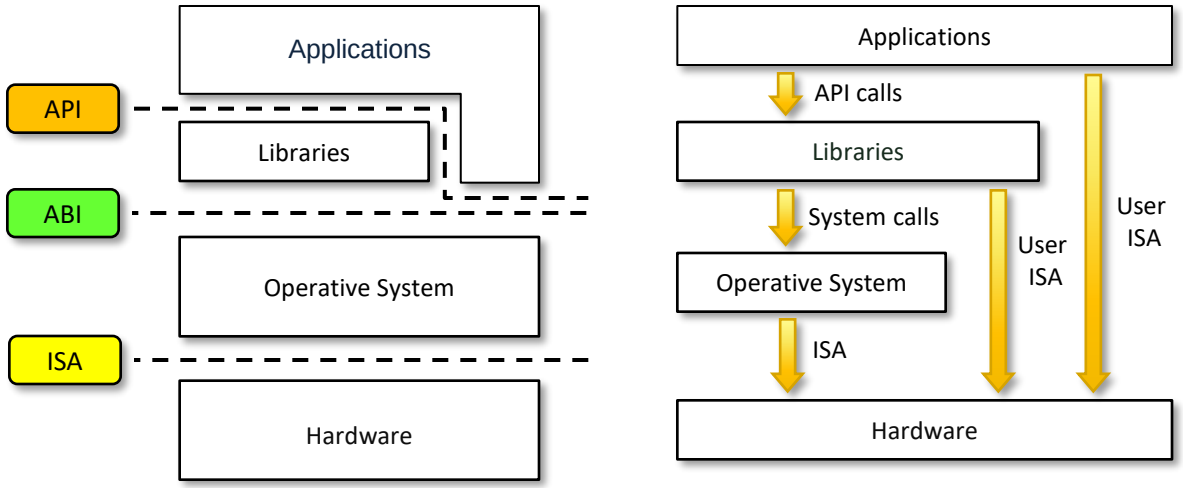
- Uses:
  - Multiple OSes on one machine, including legacy OSes
  - Isolation
  - Enhanced security
  - Live migration of servers
  - Virtual environment for testing and development
  - Platform emulation
  - On-the-fly optimization
  - Realizing ISAs not found in physical machines

Execution Virtualization

- It includes all those techniques whose aim is to *emulate* an execution environment that is separate from the one which is hosting the virtualization layer.
- It can be implemented directly
  - on top of the hardware,
  - by the operating system,
  - an application,
  - libraries dynamically
  - statically linked against an application image

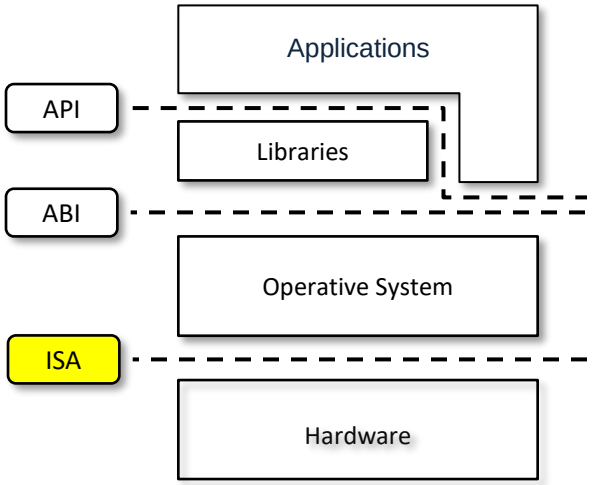
Machine Reference Model

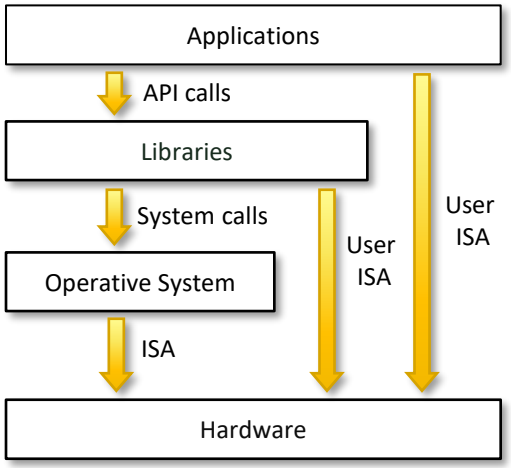
- Reference model defines the interfaces between the levels of abstractions, which hide implementation details.



ISA

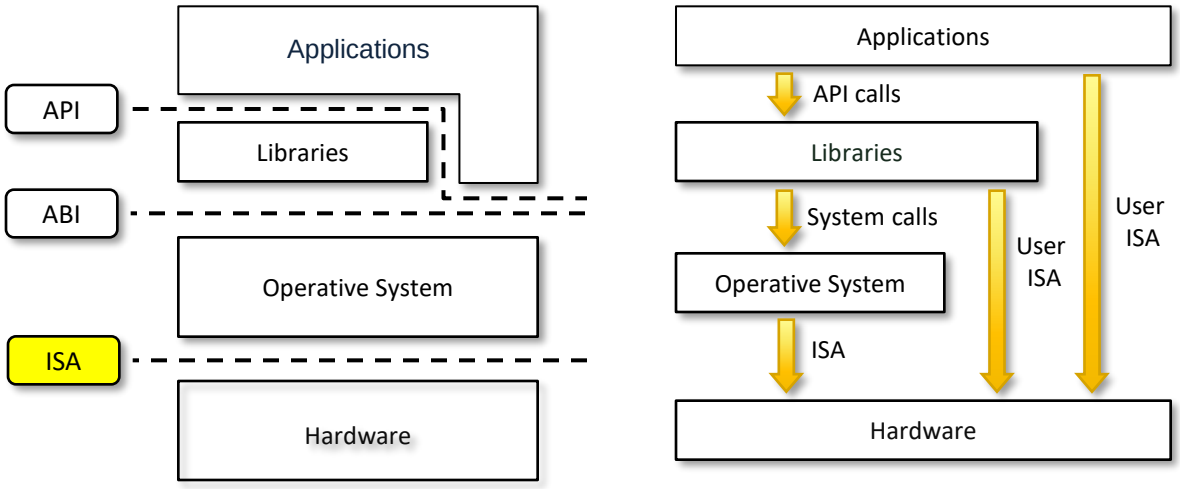
- At the bottom layer, the model for the hardware is expressed in terms of the *Instruction Set Architecture (ISA)*, which defines the instruction set for the processor, registers, memory, and interrupts management.





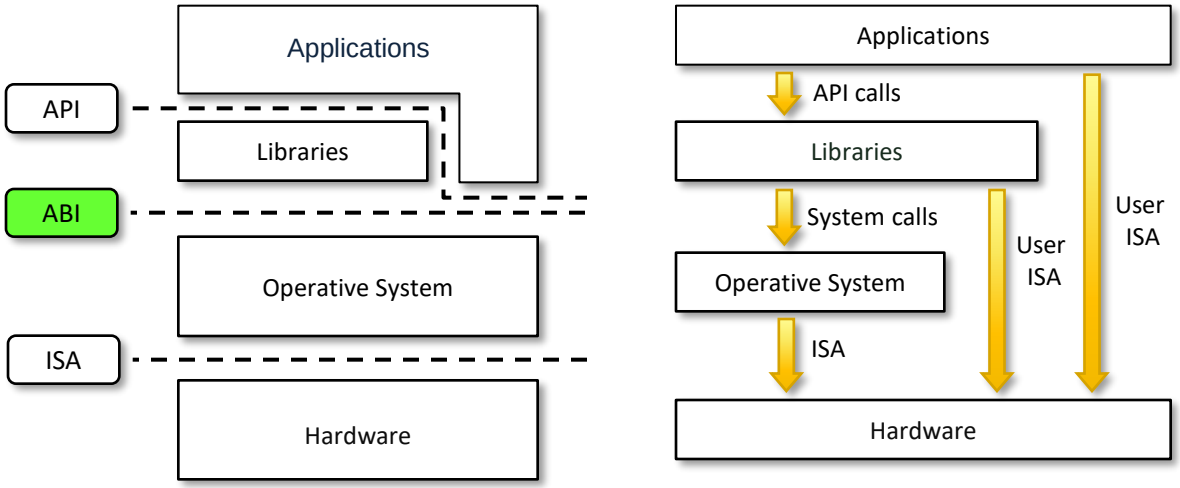
ISA Conti..

- ISA is the interface between HW/SW and it is important for the OS developer (using *System ISA*), and applications developers to directly manage the underlying hardware (using *User ISA*).



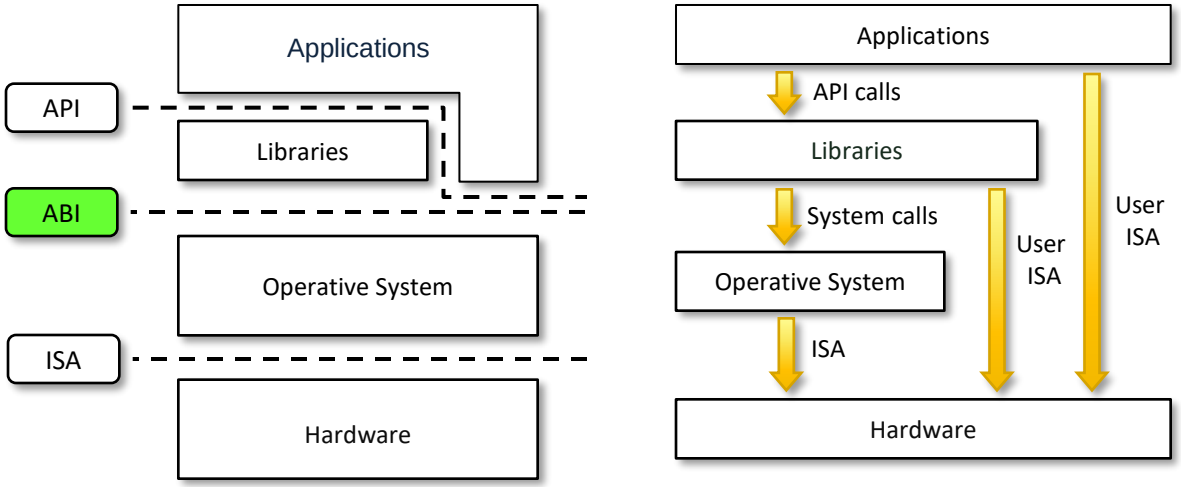
ABI

- The *Application Binary Interface (ABI)* separates the OS layer from the applications and libraries, which are managed by the OS.



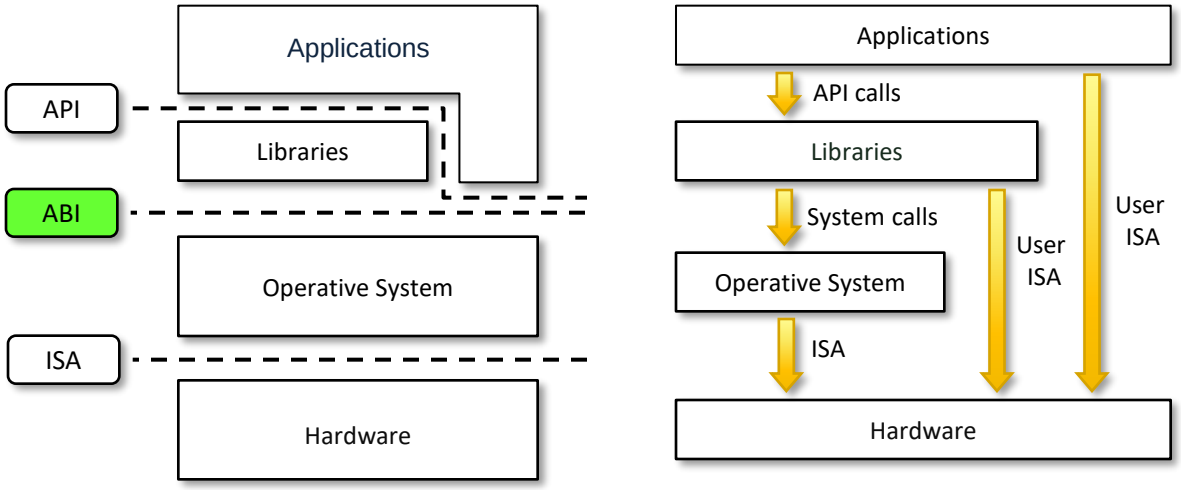
ABI Conti..

- ABI covers details such as low-level data types, alignment, and call conventions and defines a format for executable programs. System calls are defined at this level.



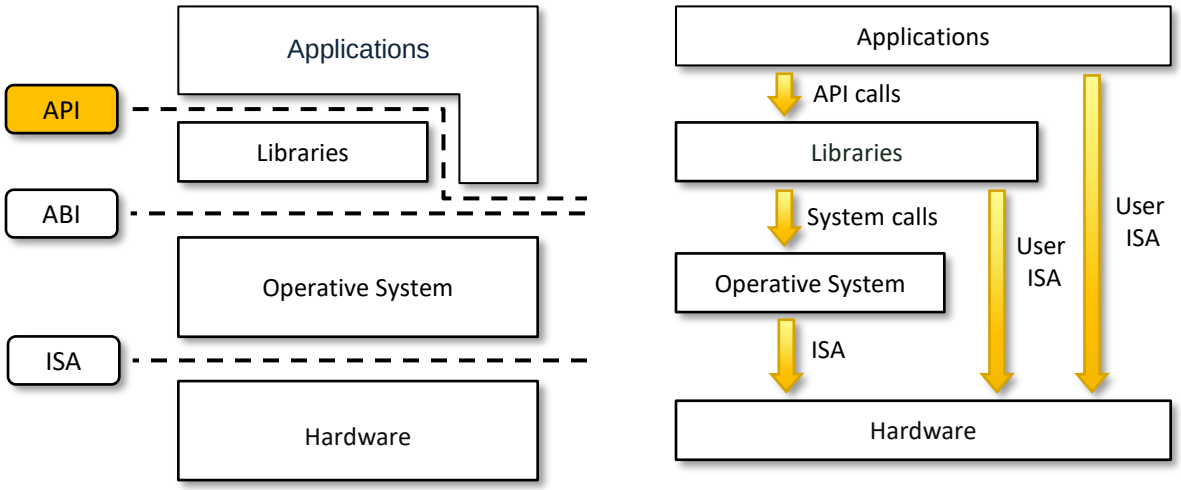
ABI Conti..

- This interface allows portability of applications and libraries across operating systems that implement the same ABI.

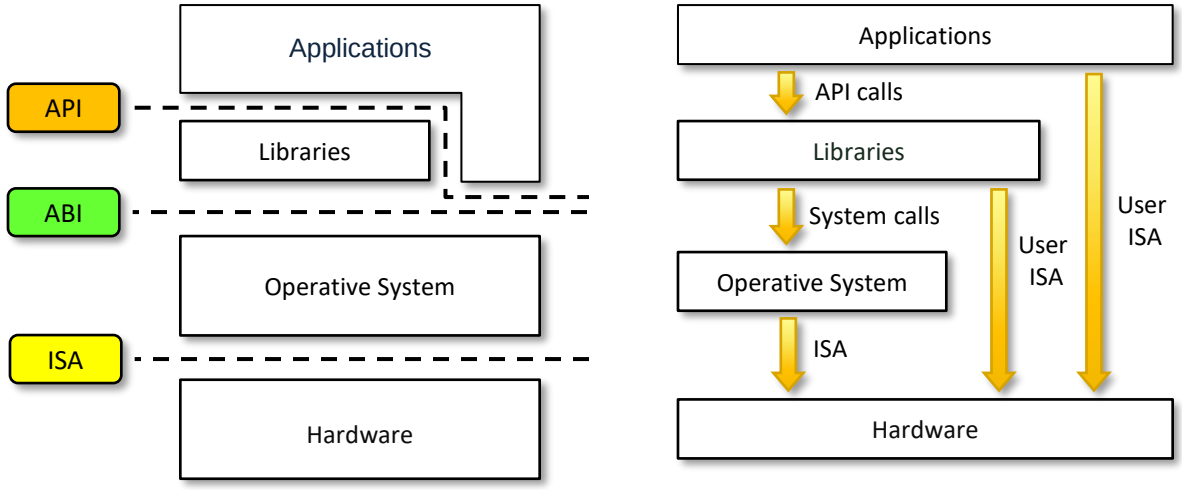


API

- The highest level of abstraction is represented by the *Application Programming Interface (API)*, which interfaces applications to libraries and/or the underlying operating system.



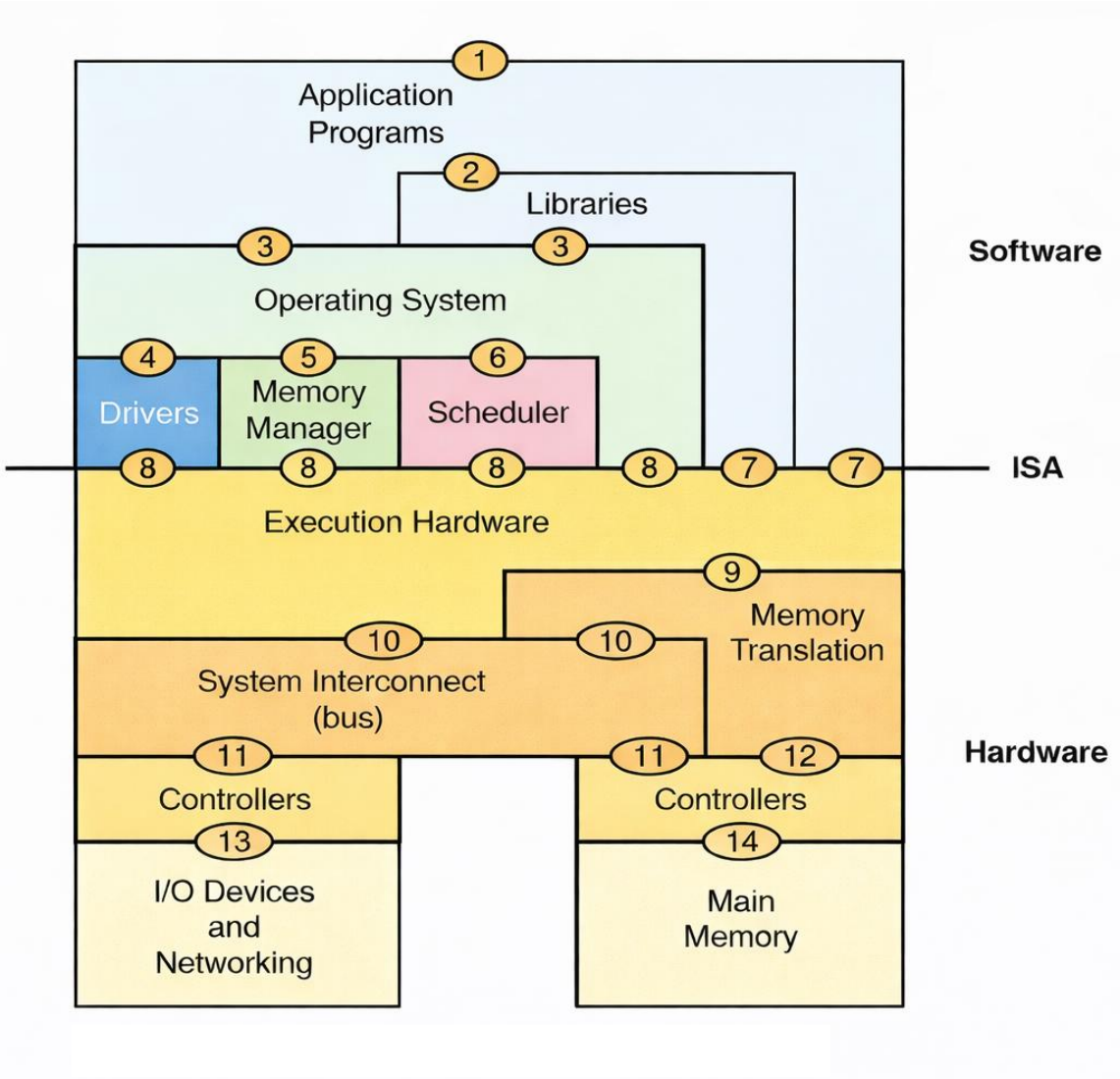
- Such reference model requires limited knowledge of the entire computing stack, and also provides ways for implementing a minimal security model for managing and accessing shared resources.



Architecture and implementation

- Analogously, the term *architecture*, when applied to computers, refers to the functionality and appearance of a computer system or subsystem but not the details of its implementation.
- The architecture is often formally described through a specification of an interface and the logical behavior of resources manipulated via the interface.
- Any architecture can have several implementations, each one having distinct characteristics, e.g., a high-performance implementation or a low-power implementation

Interfaces and implementation layers



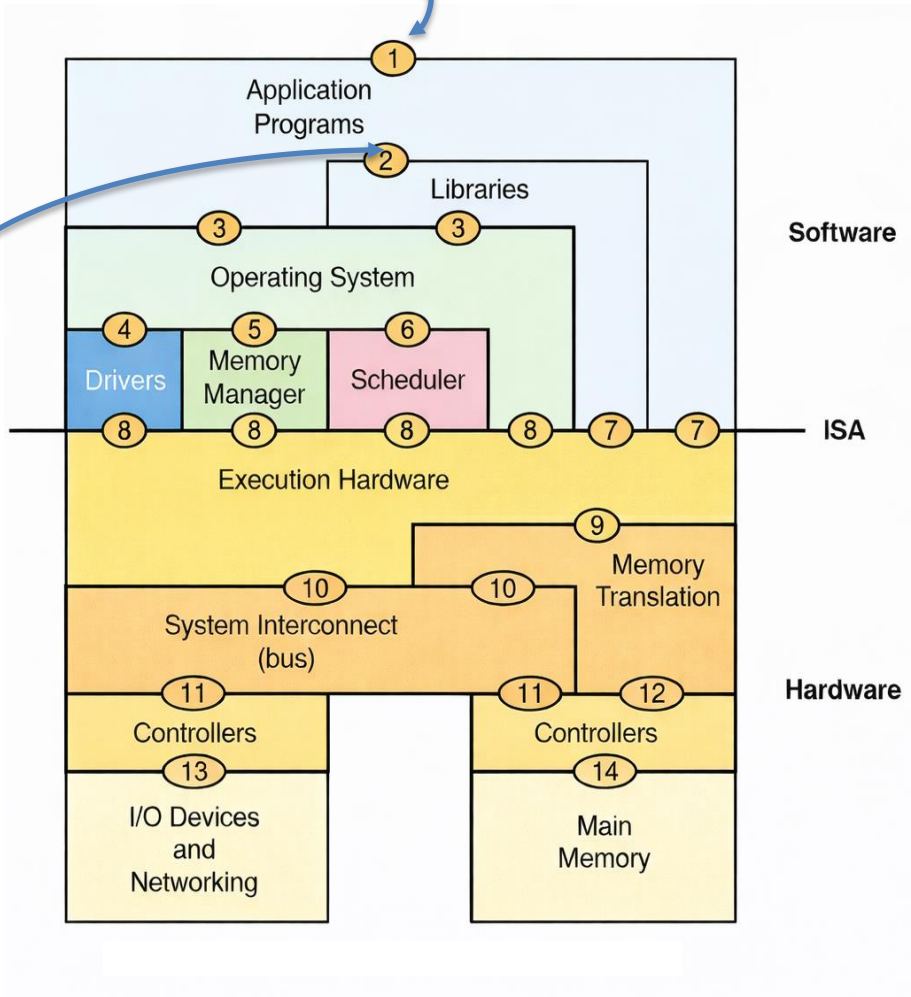
Interfaces and implementation layers

1. Application Programs

- End-user programs (browser, editor, DB, etc.)
- Run in **user mode**
- Cannot access hardware directly

2. Libraries

- Provide reusable functions for applications
- Hide OS and hardware complexity
- Use system calls to request OS services



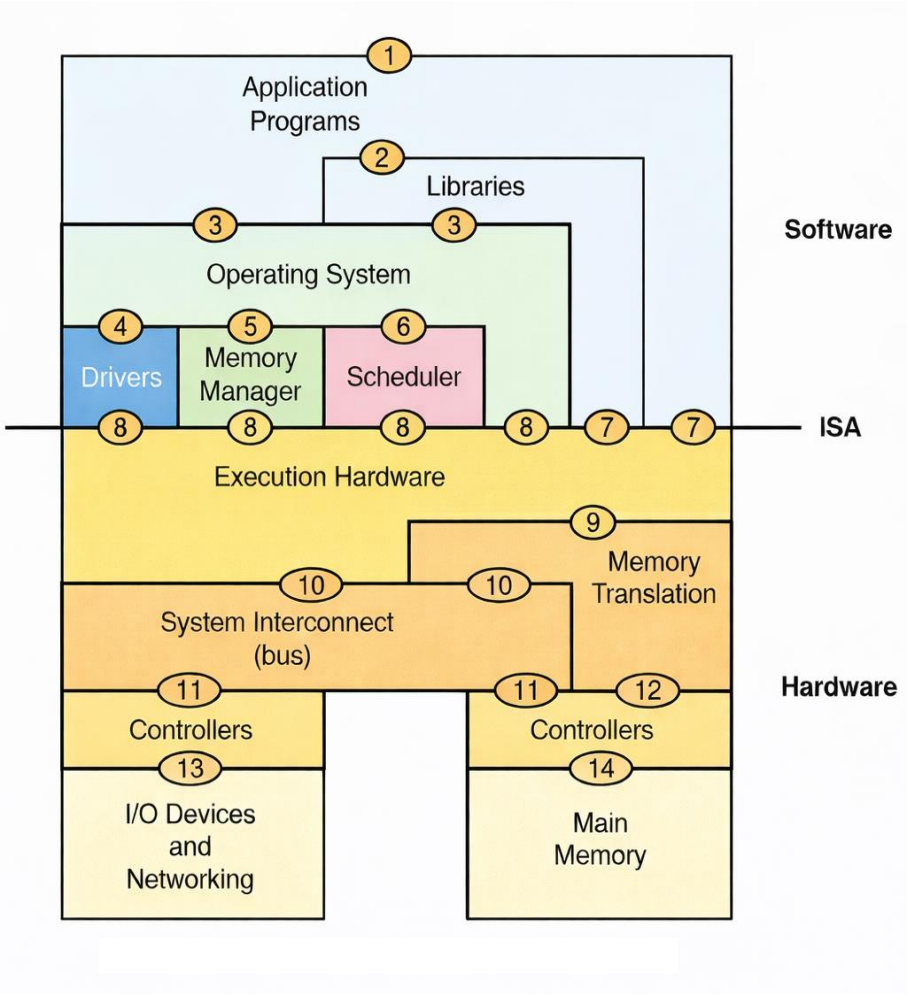
Interfaces and implementation layers

3. **Operating System Interface (System Calls)**

- Boundary between user space and kernel space
- Applications/libraries request OS services here
- Examples: process creation, file I/O, memory allocation

4. **Device Drivers**

- OS modules that control hardware devices
- Convert OS requests into device-specific commands
- Example: disk driver, NIC driver



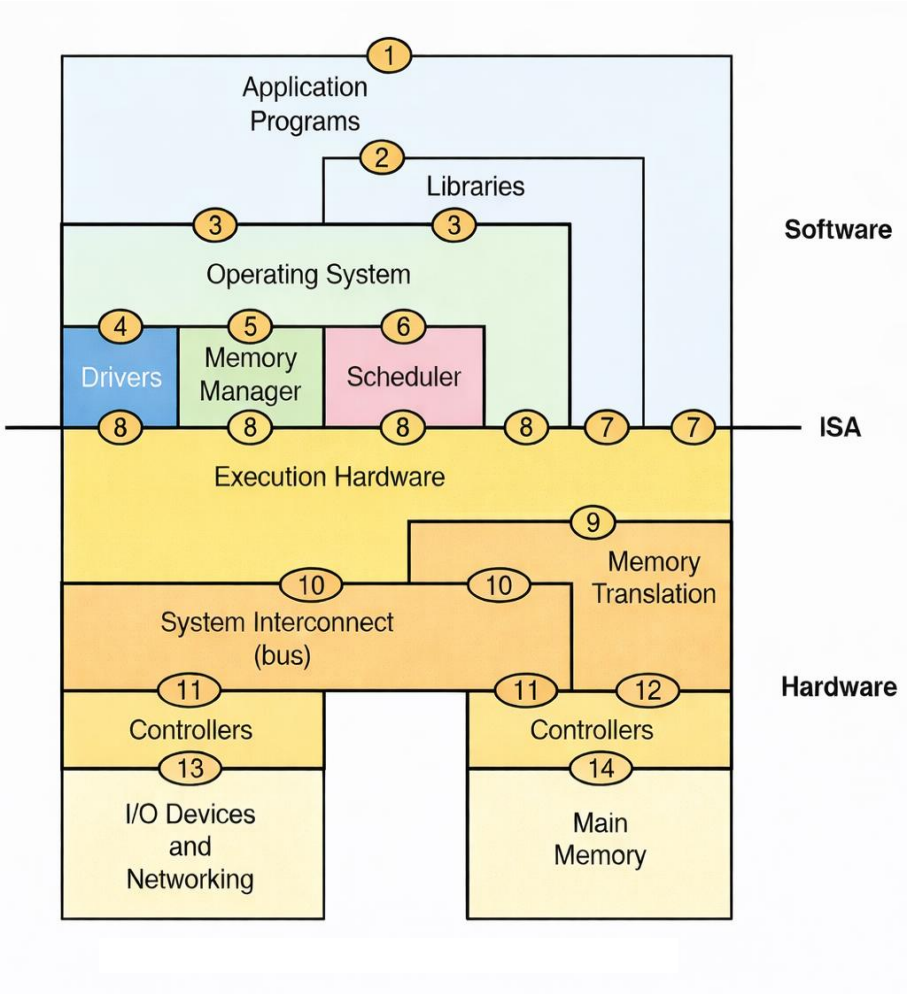
Interfaces and implementation layers

5. Memory Manager

- Manages physical and virtual memory
- Handles:
  - Paging / segmentation
  - Address space isolation
- Ensures protection between processes

6. Scheduler

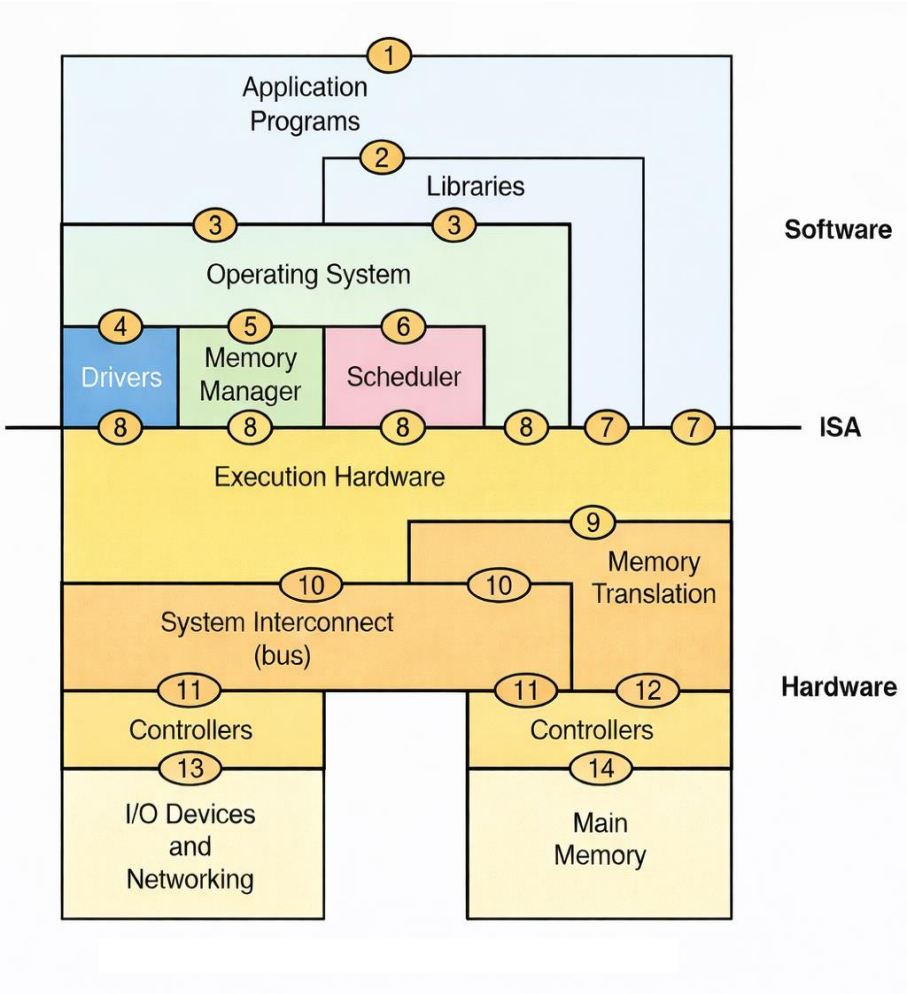
- Controls CPU allocation
- Decides:
  - Which process runs
  - For how long
- Performs context switching



Interfaces and implementation layers

7. User ISA (User-Level Instruction Set Architecture)

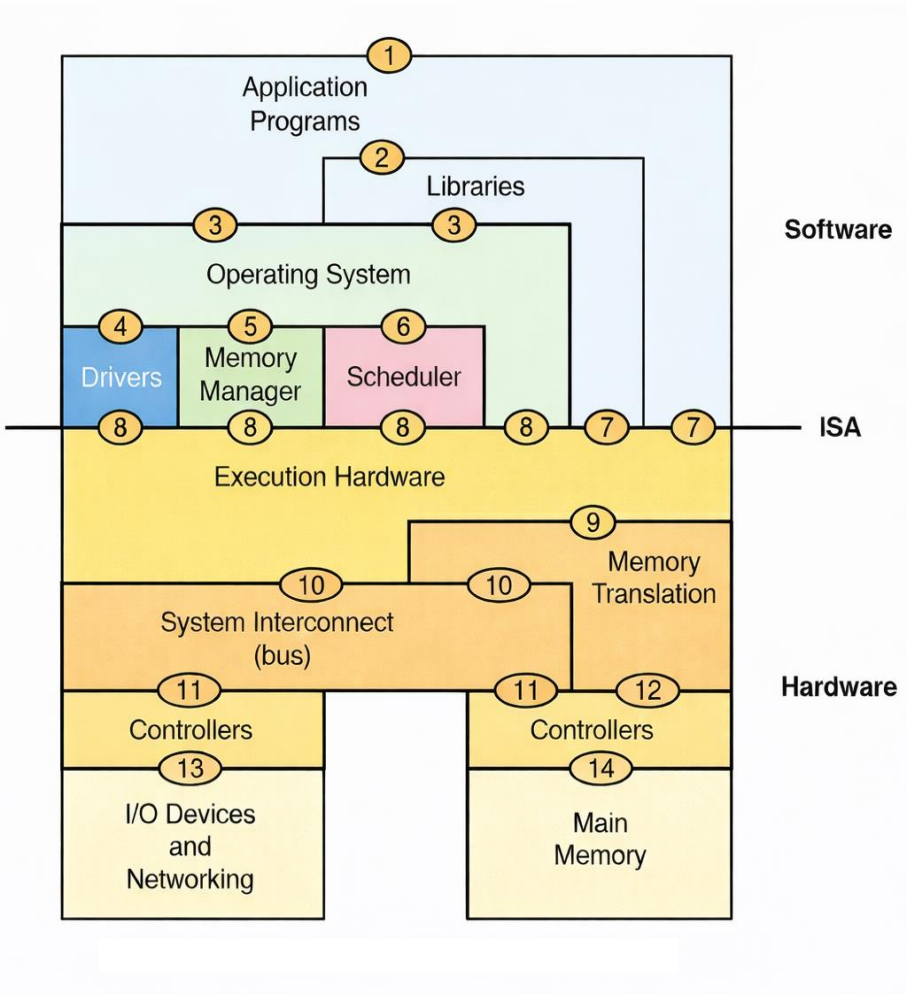
- ISA instructions available to applications
- Includes:
  - Arithmetic
  - Logical operations
  - User registers
- Safe instructions only



Interfaces and implementation layers

8. Privileged ISA (Supervisor Instructions)

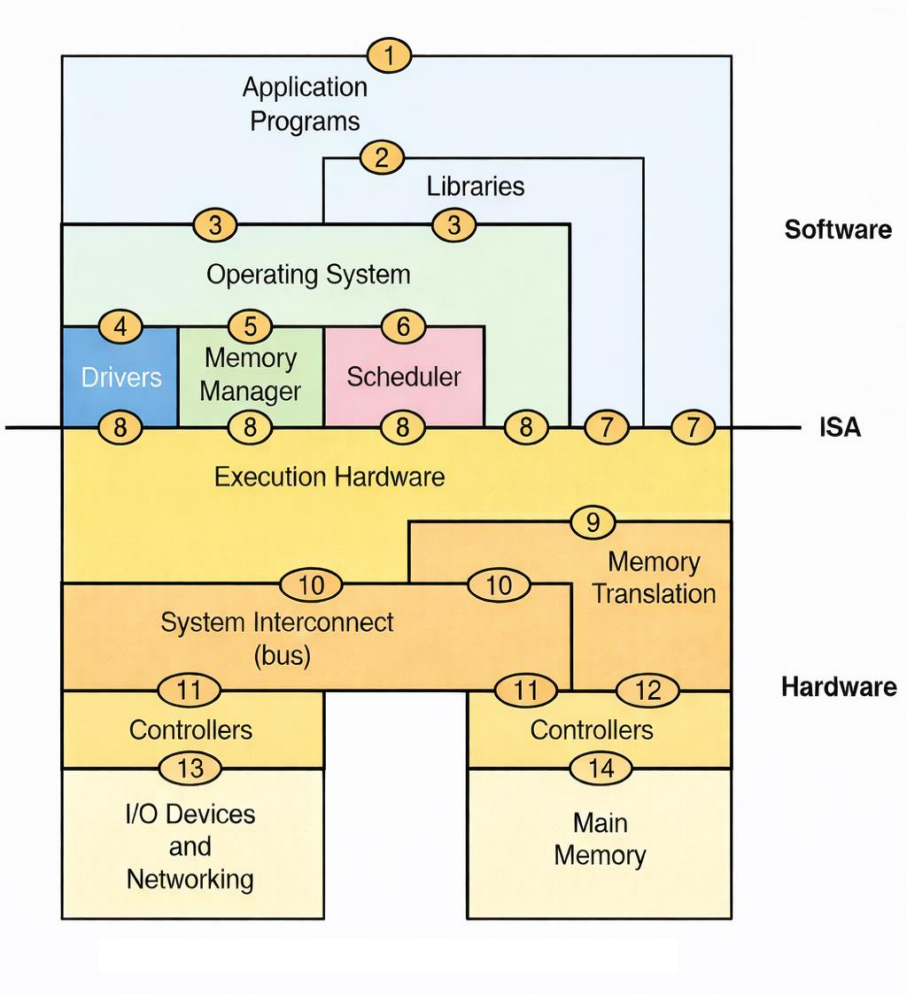
- ISA instructions available only to OS kernel
- Used by:
  - Scheduler
  - Memory manager
  - Drivers
- Examples:
  - Interrupt control
  - Page table updates
  - I/O control
- Hardware enforces this separation



Interfaces and implementation layers

9. Memory Translation Hardware

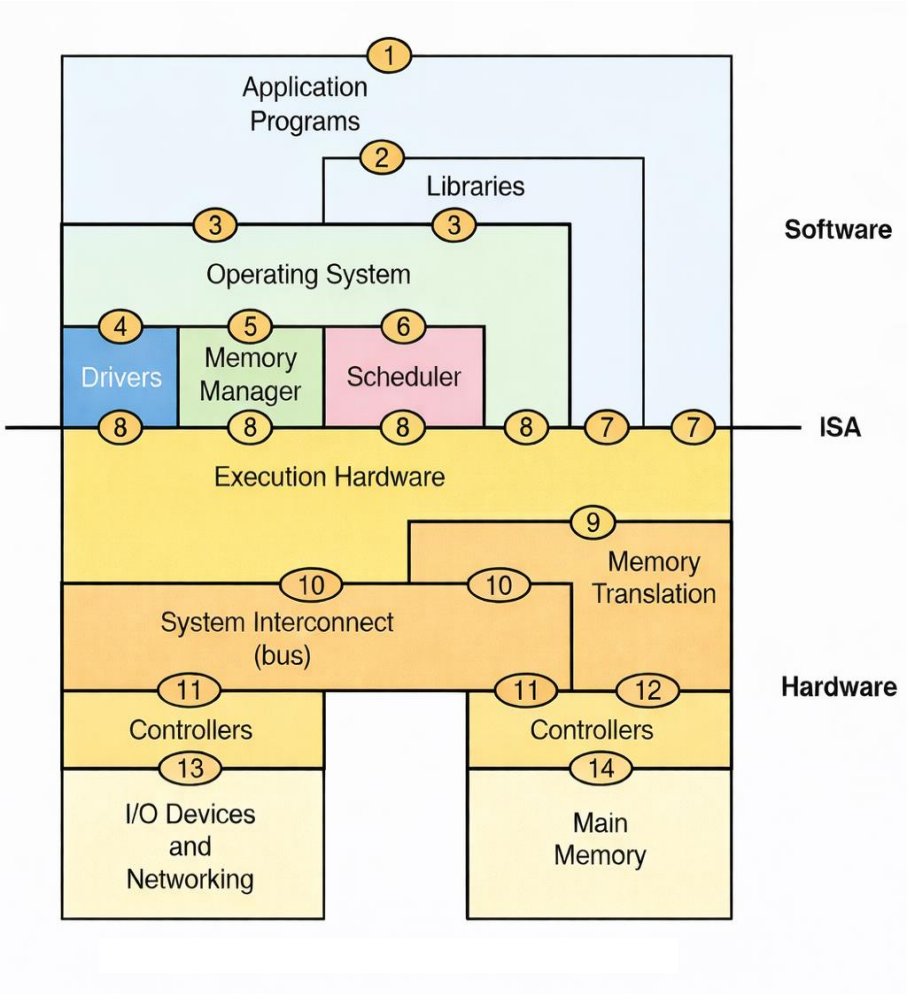
- Translates **virtual addresses → physical addresses**
- Implemented using:
  - MMU (Memory Management Unit)
  - TLB (Translation Lookaside Buffer)
- Critical for:
  - Virtual memory
  - Process isolation
- Examples: PCIe, system bus



Interfaces and implementation layers

10. System Interconnect (Bus)

- Communication path between components
- Connects:
  - CPU
  - Memory
  - I/O devices
- Examples: PCIe, system bus



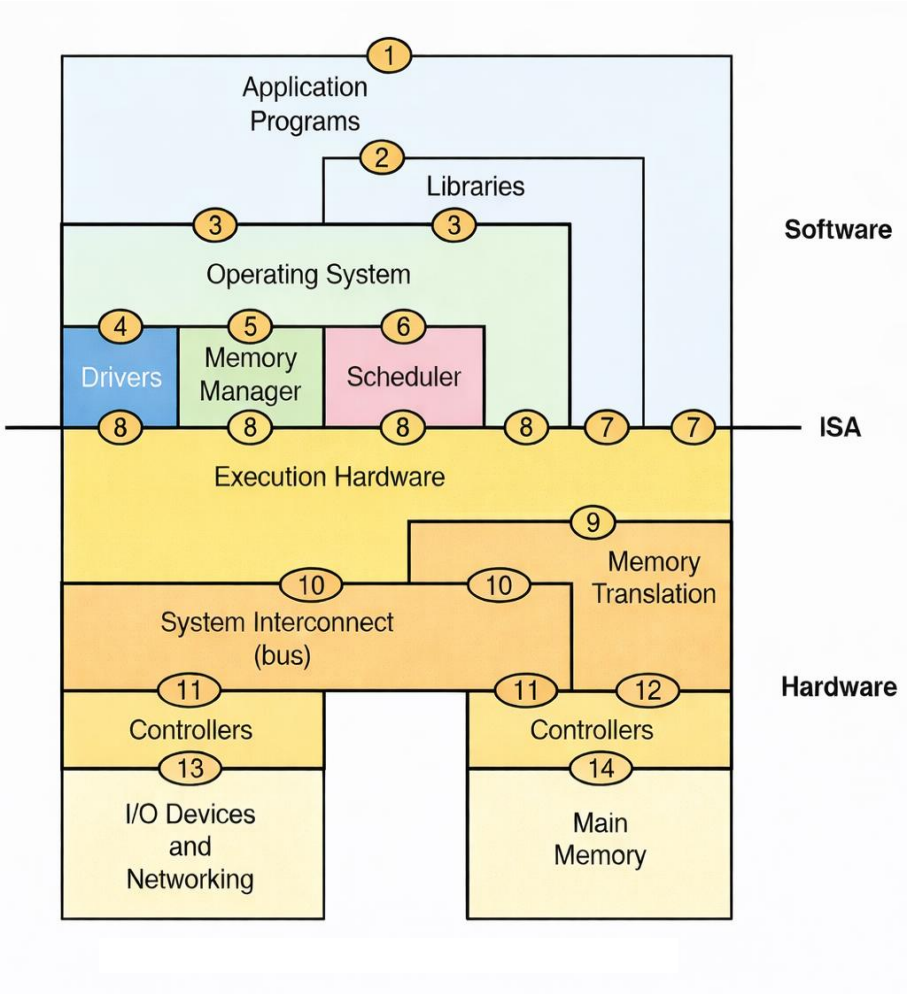
Interfaces and implementation layers

11. Controllers

- Hardware control units for devices
- Manage data transfer between devices and CPU
- Examples:
  - Disk controller
  - Network controller
  - Memory controller

12. Controllers (Memory Side)

- Specialized controllers for **main memory**
- Handle:
  - Memory access timing
  - Data transfers
- Work closely with MMU



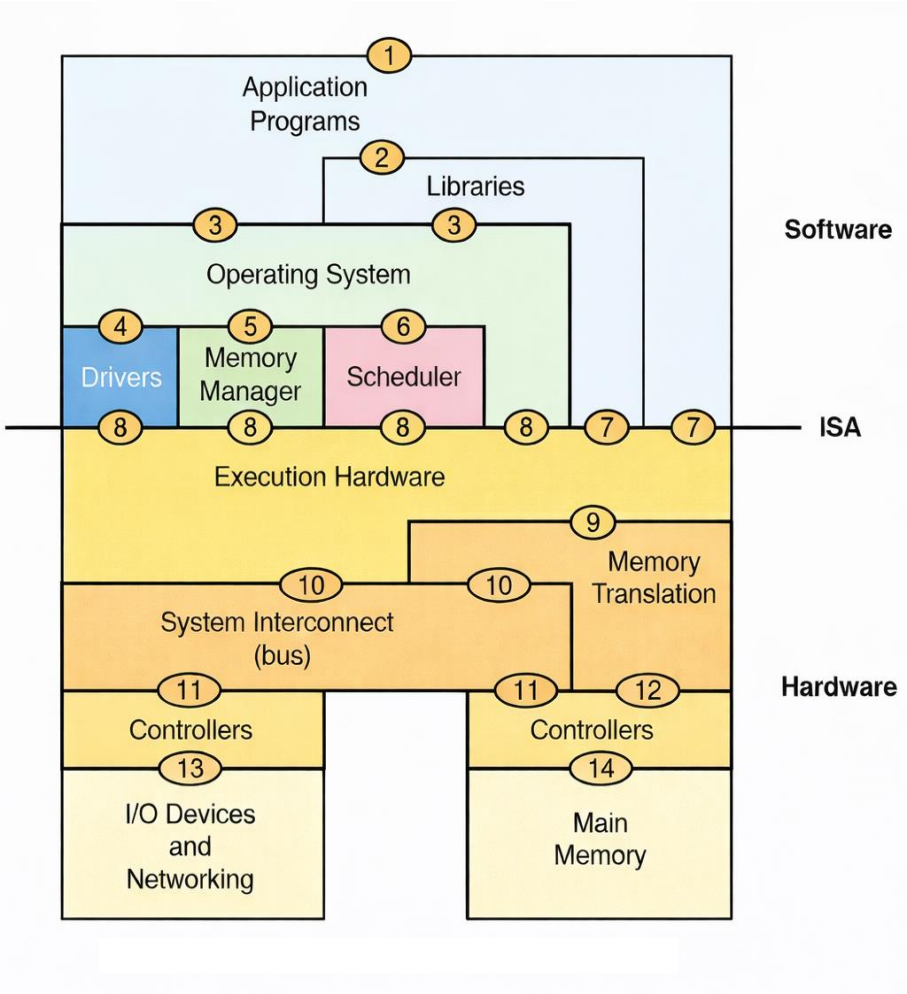
Interfaces and implementation layers

13. I/O Devices and Networking

- External devices:
  - Disk
  - Keyboard
  - Network cards
- Interact with OS via **device drivers**

14. Main Memory (RAM)

- Stores:
  - Running programs
  - OS kernel
  - Data
- Directly accessed by CPU via memory controller



ISA

- The **Instruction Set Architecture (ISA)** has two important parts for virtualization: the **user ISA** and the **system ISA**.
- The **user ISA** includes instructions visible to application programs, while the **system ISA** includes privileged instructions visible to the operating system for hardware management.
- The operating system uses **both user ISA and system ISA**, whereas applications can use **only the user ISA** (as shown by interfaces 7 and 8 in Figure).

Define Virtualization!

- Virtualization refers to the creation of a virtual resource such as a server, desktop, operating system, file, storage or network.
- *Virtualization is a level of indirection between hardware and software*
- Virtual Machine abstraction
  - Run all software written for physical machines

What is Emulation?

- **Emulation** refers to the ability of a computer program in an electronic device to **imitate** another program or device.
- VM can emulate complete hardware, means and unmodified guest OS for one PC can be run
  - Bochs, BIRD, QEMU

Virtualization vs Emulation

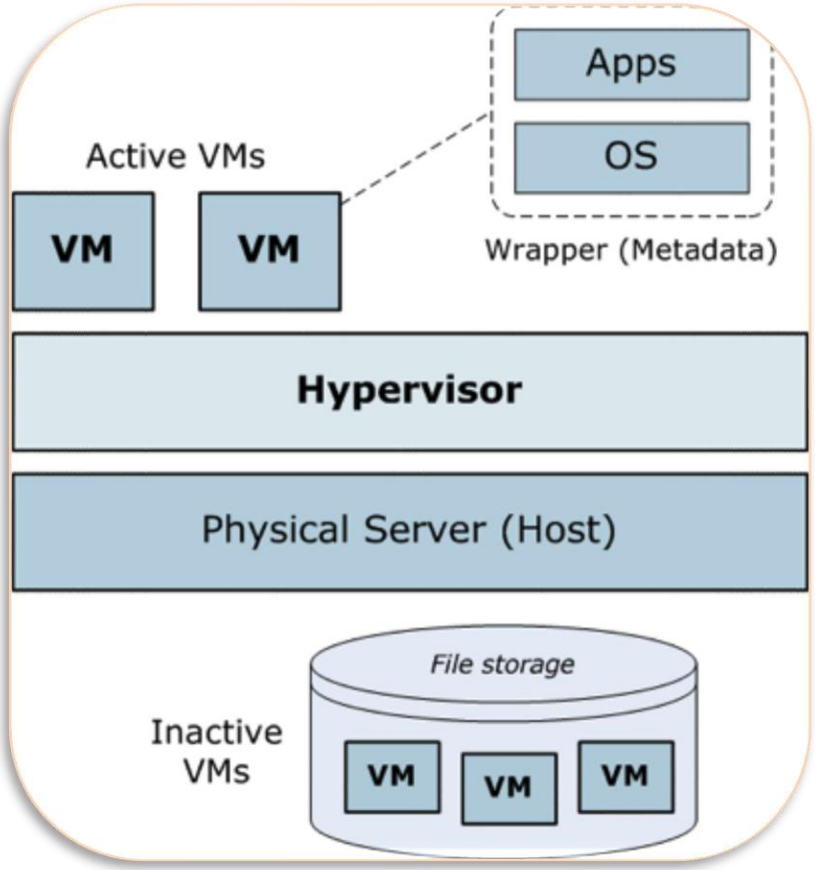
- Virtualization involves simulating parts of a computer's hardware – **just enough** for a guest operating system to run unmodified - but most operations still occur on the real hardware for efficiency reasons.

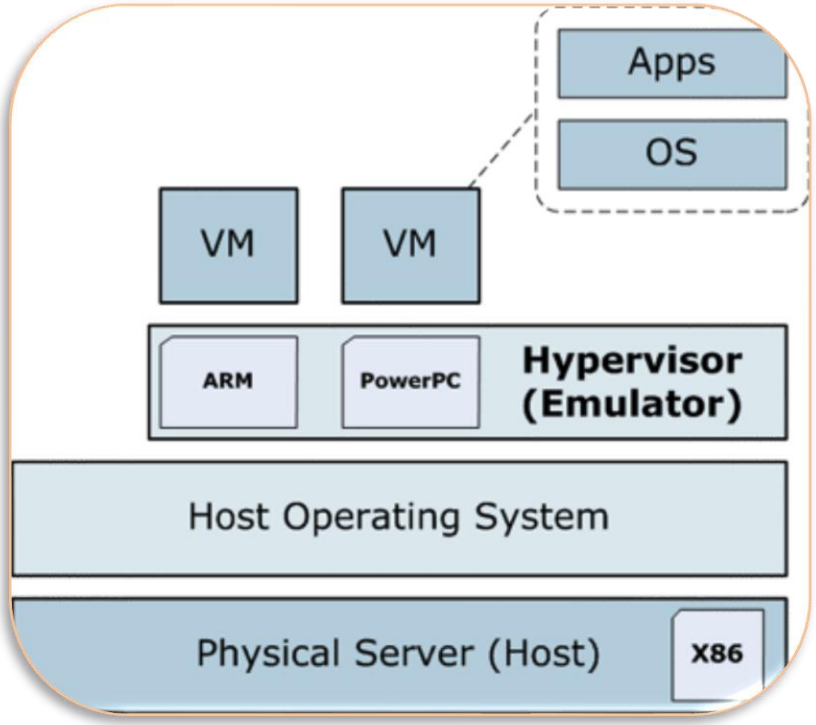
- Ex. VMWare can provide a virtual environment for running a virtual WindowsXP machine. However VMWare cannot work on any real hardware other than a real x86 PC.

Virtualization vs Emulation

- In emulation the virtual machine simulates the complete hardware in software.
- This allows an operating system for one computer architecture to be run on the architecture that the emulator is written for.

Difference between Virtualization and Emulation

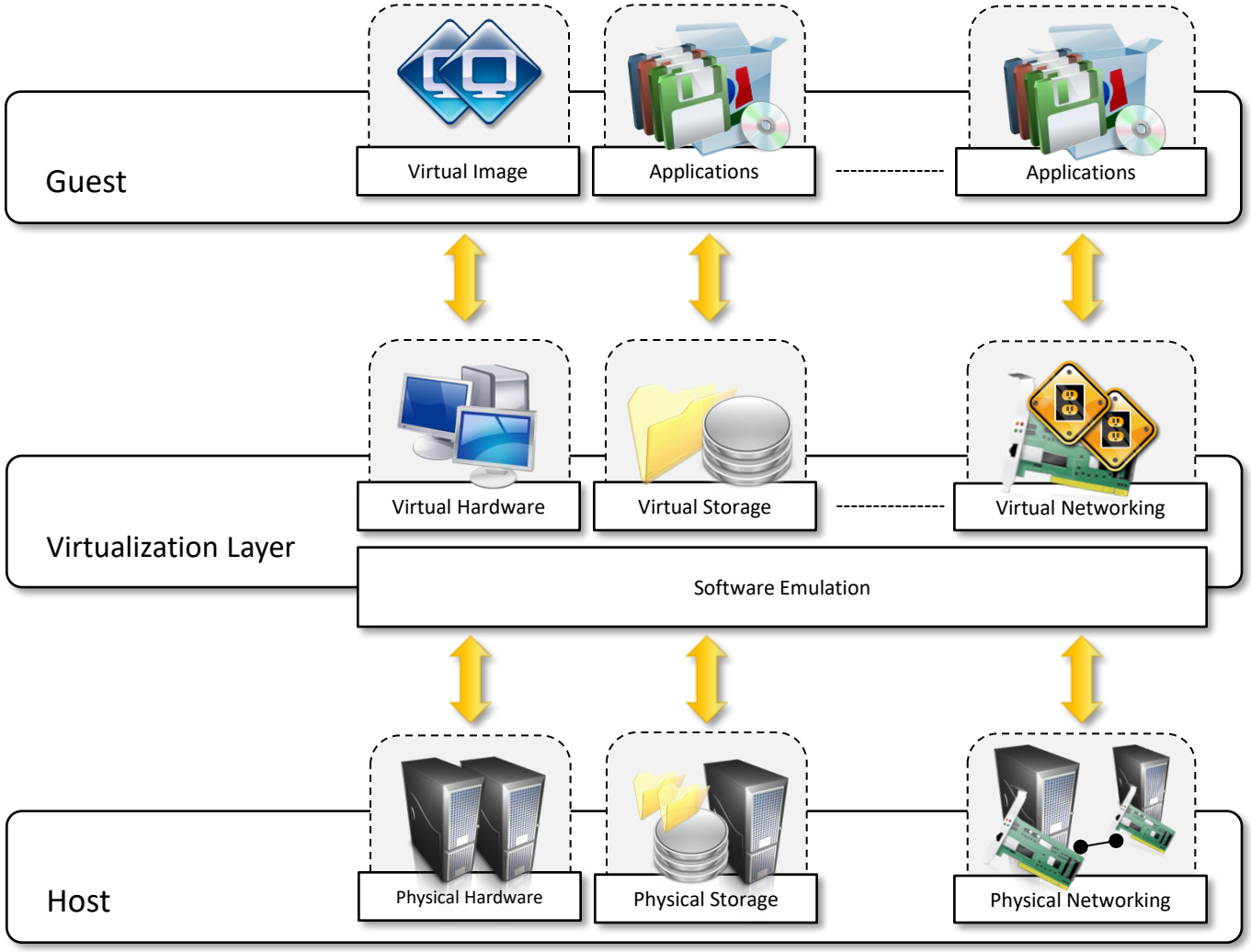




Ref.: PowerPC: Performance Optimization With Enhanced RISC-Performance Computing, (1991), Apple-IBM-Motorola

Components of Virtualized Environments

- In a virtualized environment there are three major components: *guest*, *host*, and *virtualization layer*.
  - The *guest* represents the system component that interacts with the virtualization layer rather than with the host as it would normally happen.
  - The *host* represents the original environment where the guest is supposed to be managed.
  - The *virtualization layer* is responsible for recreating the same or a different environment where the guest will operate.



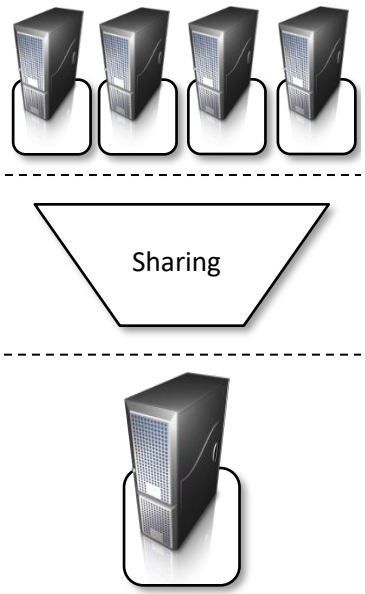
The main common characteristic: virtual environment is created by means of a *software program*.

Managed Execution

- Virtualization of the **execution environment** does not only allow the increased security but a wider range of features can be implemented.
  - *Sharing*,
  - *aggregation*,
  - *emulation*, and
  - *isolation*

  are the most relevant.

Managed Execution Conti..

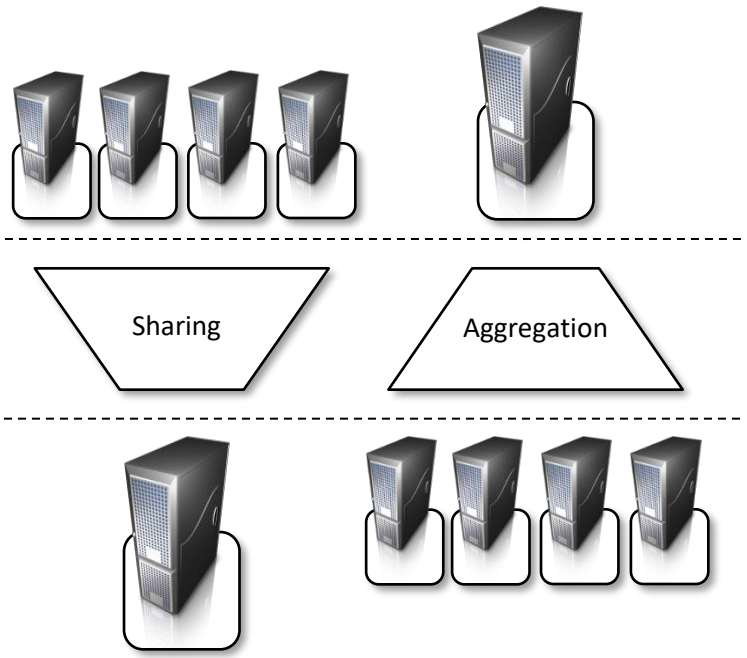


Virtual Resources

Virtualization

Physical Resources

Managed Execution Conti..

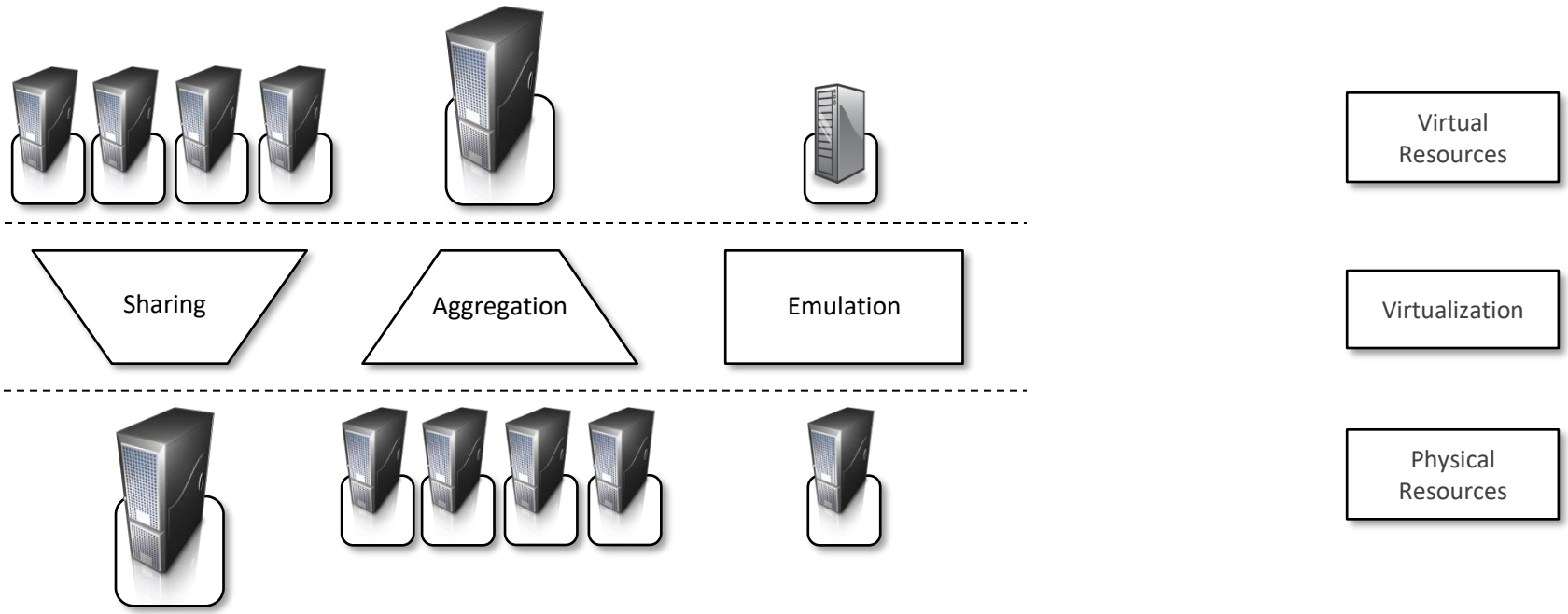


Virtual Resources

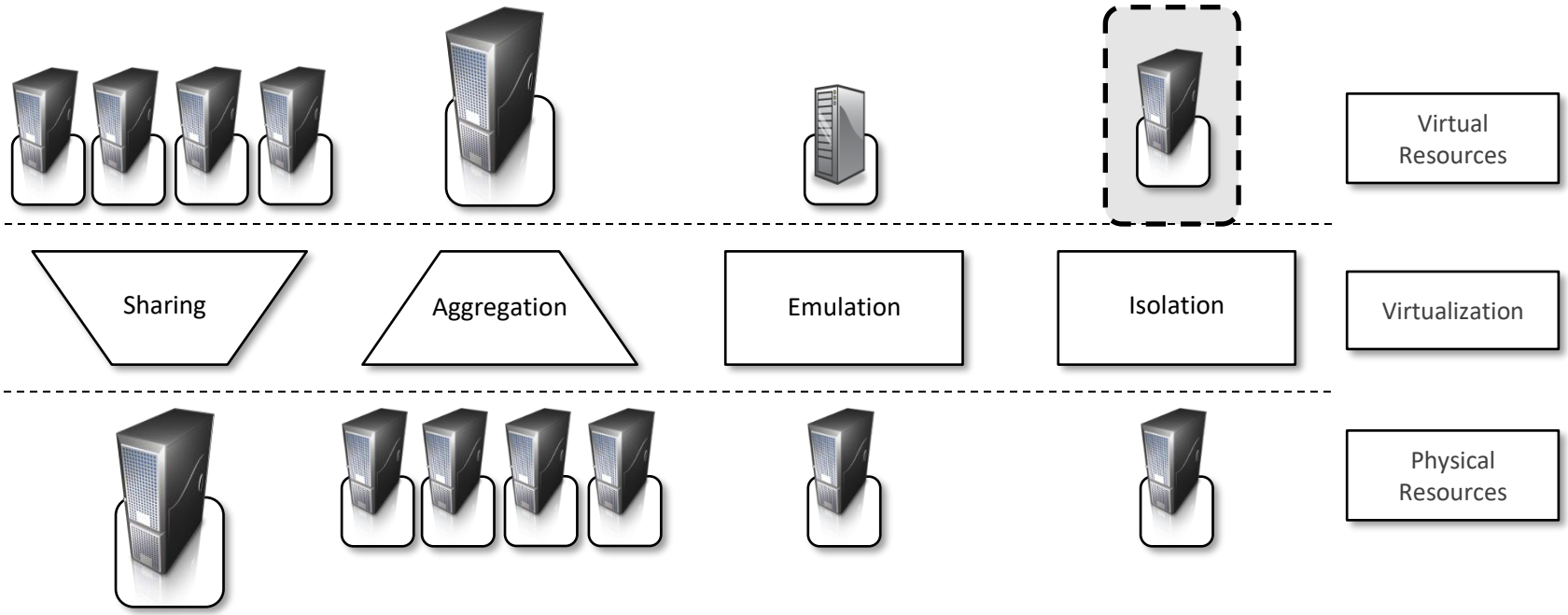
Virtualization

Physical Resources

Managed Execution Conti..



Managed Execution Conti..



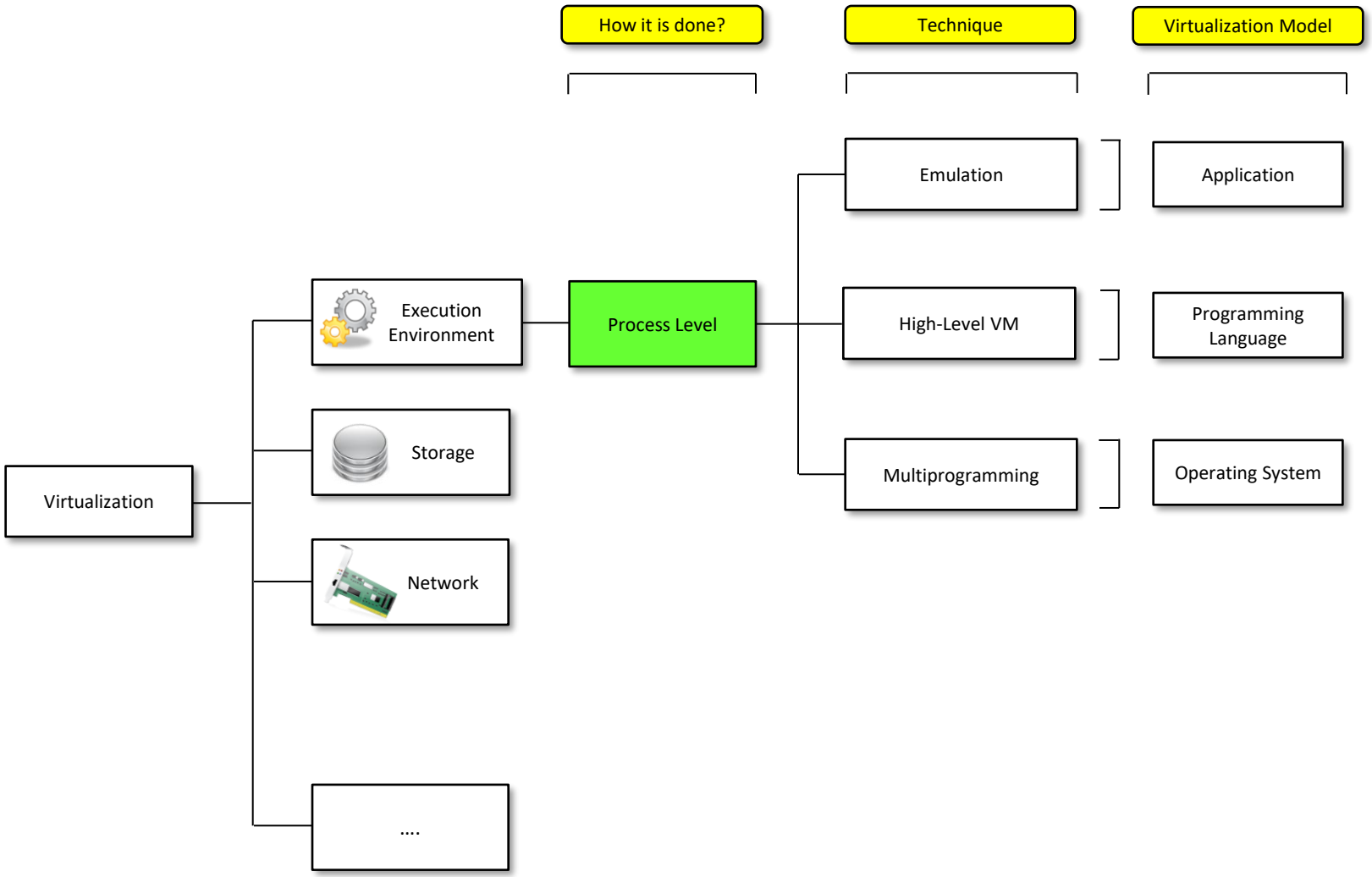
Managed Execution Conti..

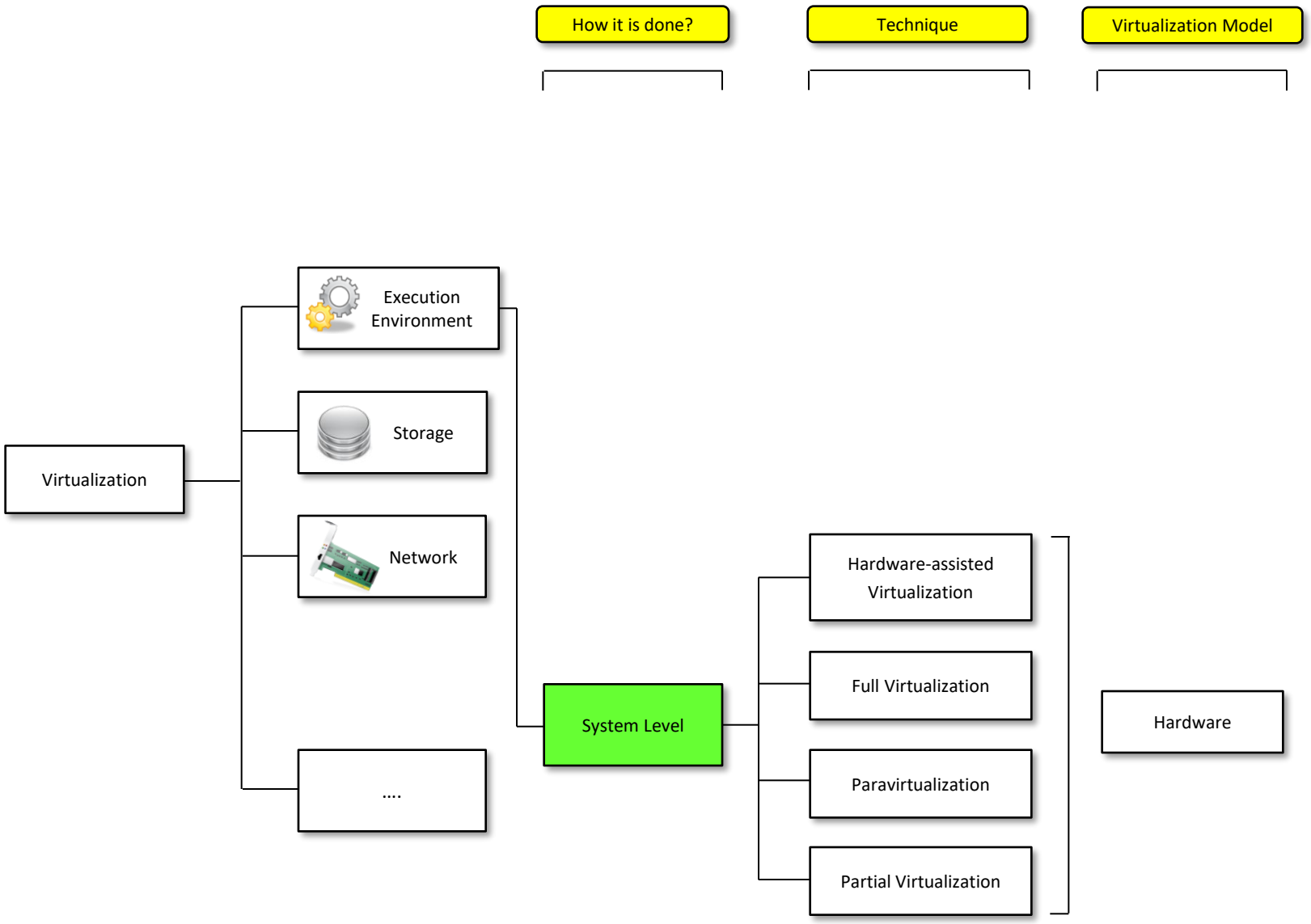
- *Sharing.* Virtualization allows the creation of a separate computing environment within the same host.
- *Aggregation.* Virtualization also allows the aggregation resources, which is the opposite process.
- *Emulation.* Guests are executed within an environment that is controlled by the virtualization layer, which ultimately is a program. *Emulation* allows for controlling and tuning the environment that is exposed to guests.

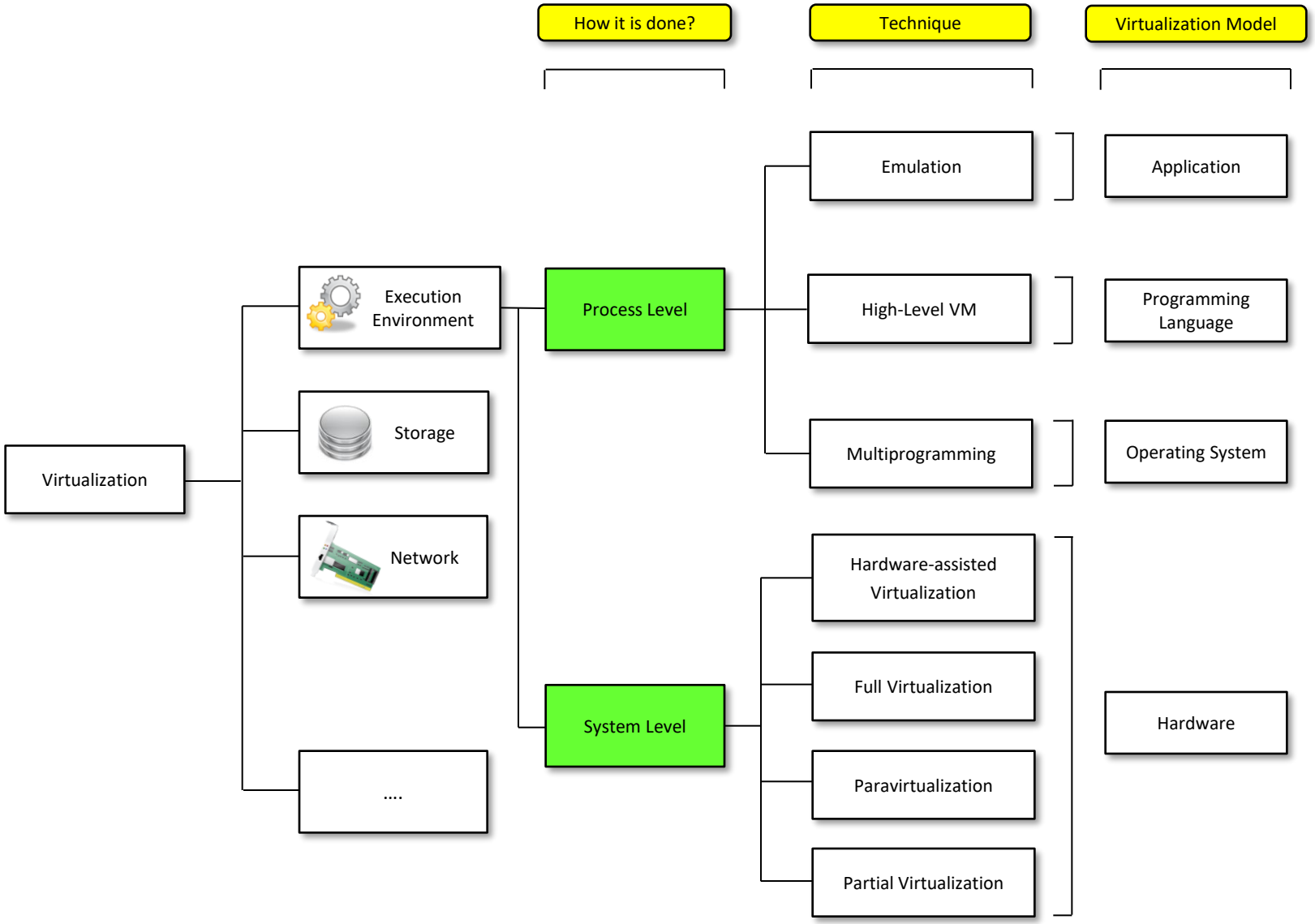
Managed Execution Conti..

- *Isolation.* Virtualization allows providing guests—whether they are operating systems, applications, or other entities—with a complete separate environment, in which they are executed.
- *performance tuning.* Control the performance of the *guest* by finely tuning the properties of the resources exposed to effectively implement a QoS infrastructure that more easily fulfill the SLA established for the guest.

Taxonomy of Virtualization Techniques

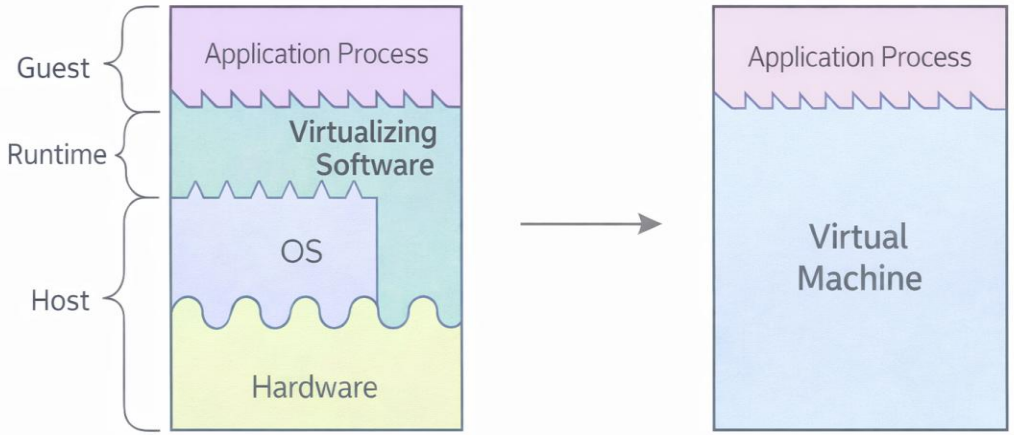

How it is done?
Technique
Virtualization Model
Virtualization
Execution Environment
Storage
Network
....
Process Level
Emulation
High-Level VM
Multiprogramming
Application
Programming Language
Operating System


How it is done?
Technique
Virtualization Model
Virtualization
Execution Environment
Storage
Network
....
System Level
Hardware-assisted Virtualization
Full Virtualization
Paravirtualization
Partial Virtualization
Hardware


How it is done?
Technique
Virtualization Model
Virtualization
Execution Environment
Storage
Network
....
Process Level
System Level
Emulation
High-Level VM
Multiprogramming
Hardware-assisted Virtualization
Full Virtualization
Paravirtualization
Partial Virtualization
Application
Programming Language
Operating System
Hardware

Virtual Machines- Process VM

- Process VM
- Virtualizes the ABI
- Virtualization software = Runtime
  - Runs in non-privileged mode (user space)
  - Performs binary translation.
- Terminates when guest process terminates.



Example- Process VM

- Java Virtual Machine (JVM)
- Type: Process VM
- Virtualizes: ABI (plus a virtual ISA at a higher level)
- Runtime: JVM
- How?
  - Runs in user space
  - Executes Java bytecode using:
    - Interpretation (initially)
    - JIT (dynamic binary translation of frequently executed bytecode into native machine code)
  - Terminates when the Java process exits

Example Process Virtualization

```
class Hello {
    public static void main(String[] args) {
        System.out.println("Hello JVM");
    }
}
```

- This is a **single process**, executed by the **JVM** so this is **Process Virtualization**.

What does "Process Virtualization" mean here?

- The JVM creates a **virtual execution environment for one program**, hiding:
  - real CPU
  - real memory
  - real OS differences
- The program thinks:

  I am running on my own machine
- But actually:

  JVM is *translating and managing everything*.

Complete Execution Flow

1. **Java Source Code**

```
System.out.println("Hello JVM");
```

- Written once
- Platform independent
- Cannot run directly on hardware

2. **Compilation to Bytecode (Process VM ISA)**

Command: `javac Hello.java`

Output: `Hello.class`

Complete Execution Flow

- What is inside `Hello.class`?
- **Bytecode**
- This is the **JVM's ISA**
- Not x86, not ARM
- **This is the key point: JVM bytecode = virtual ISA**
- This confirms **Process VM (Different ISA)**

Complete Execution Flow

3. JVM Starts (Virtual Process Created)

```
Command: java Hello
```

- Now the JVM: Creates a **virtual process**
- Allocates:
  - Heap
  - Stack
  - Method Area
- Sets up **security and isolation**
- OS sees **one process**: java
- JVM sees **one virtual machine process**

Complete Execution Flow

4. Class Loader (Virtual Memory View)

- JVM loads:
  - Hello.class
  - System.class
  - PrintStream.class
- Class Loader ensures:
  - No illegal access
  - Proper namespace isolation
- Still **process-level virtualization**

```
Virtual Memory of JVM
-----------------------------
Method Area  ← Classes loaded here
Heap         ← Objects
Stack        ← Threads
```

Complete Execution Flow

5. Bytecode Interpreter Starts: Example bytecode (simplified):

```
getstatic java/lang/System.out
ldc "Hello JVM"
invokevirtual java/io/PrintStream.println
```

- The JVM:
  - Reads **bytecode**
  - Executes instruction by instruction
- Two choices:
  - Interpret (slow)
  - JIT compile (fast)

Complete Execution Flow

6. JIT Compilation (Virtual to Native)

- Hot code is detected: `println()`
- JIT does:
  - Bytecode to **native machine code**
  - x86 OR ARM OR RISC-V
  - Caches compiled code
- Important:
  - JVM decides **when**, **what**, and **how**
- This is dynamic binary translation which is core process VM behavior.

Complete Execution Flow

7. Execution on Real Hardware:

- Now: Native instructions execute on CPU
- JVM still controls:
  - Memory
  - Threads
  - Garbage Collection
- The Java program:
  - never sees real CPU
  - never sees real memory
  - never sees real OS details
- **Perfect process virtualization**

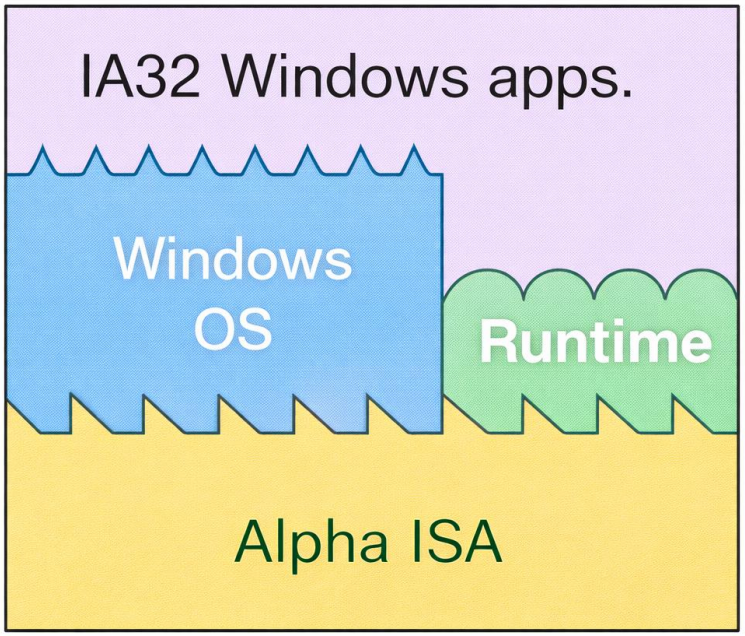
Complete Execution Flow

8. Output: `Hello JVM`

Program ends then JVM cleans up virtual process.

Other Process VMs

- Emulators
  - Support one ISA on hardware designed for another ISA
  - Interpreter:
    - Fetches, decodes and emulates individual instructions. Slow.
  - Dynamic Binary Translator:
    - Blocks of source instructions converted to target instructions.
    - Translated blocks cached to exploit locality.



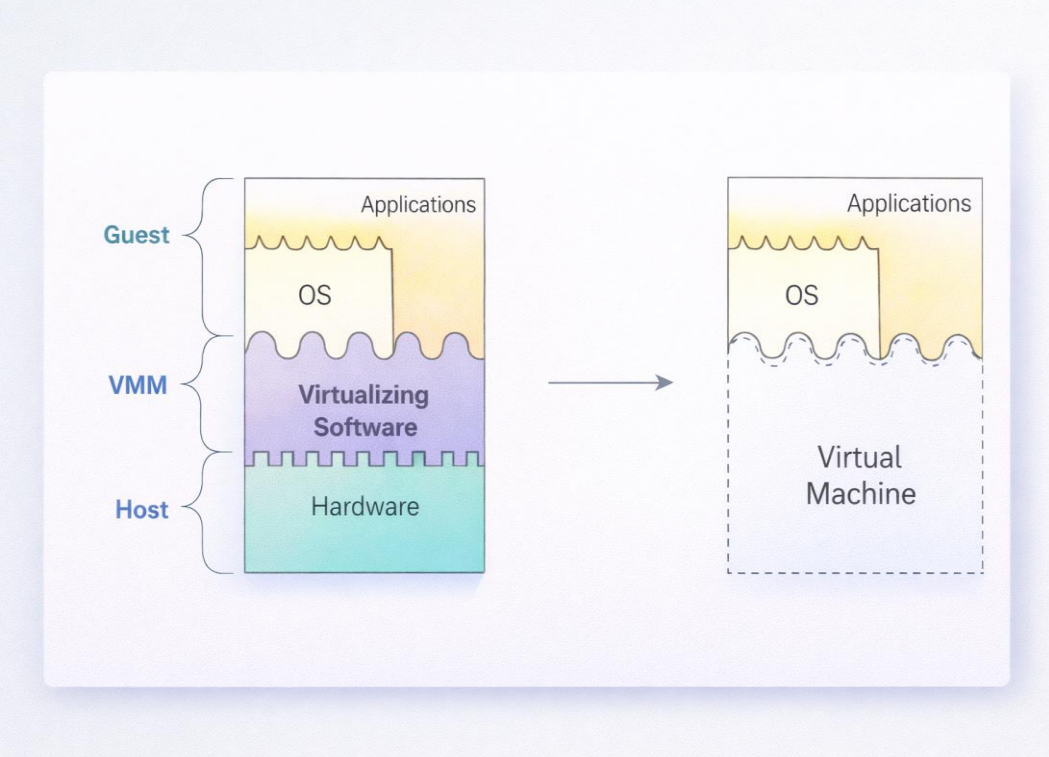
Digital FX!32 Emulator

Other Process VMs

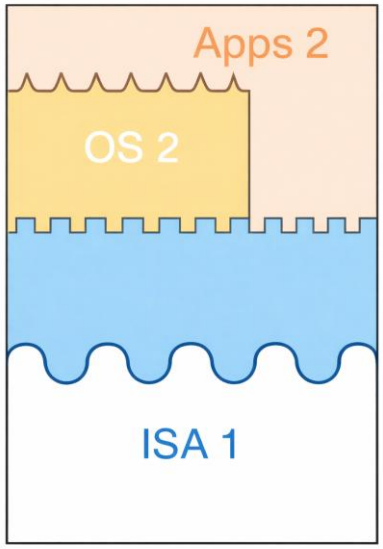
- Same ISA Binary Optimizers
  - Optimize code on the fly
  - Same as emulators except source and target ISAs are the same.
- Process in a multiprogramming OS
  - Standard OS syscall interface + instruction set
  - Multiple processes, each with its own address space and virtual machine view.

Virtual Machines: System VM

- System VM
- Virtualizes the ISA
- Virtualization software = Hypervisor
  - Runs in privileged mode
  - Traps and emulates privileged instructions

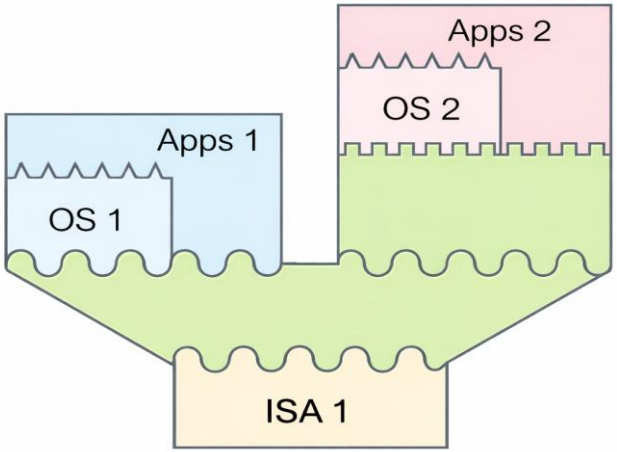


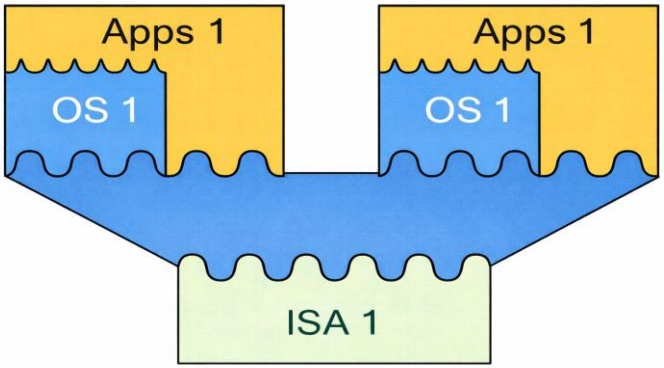
Virtual Machines Applications



(a) Emulating one instruction set with another

(b) replicating a virtual machine so that multiple operating systems can be supported simultaneously

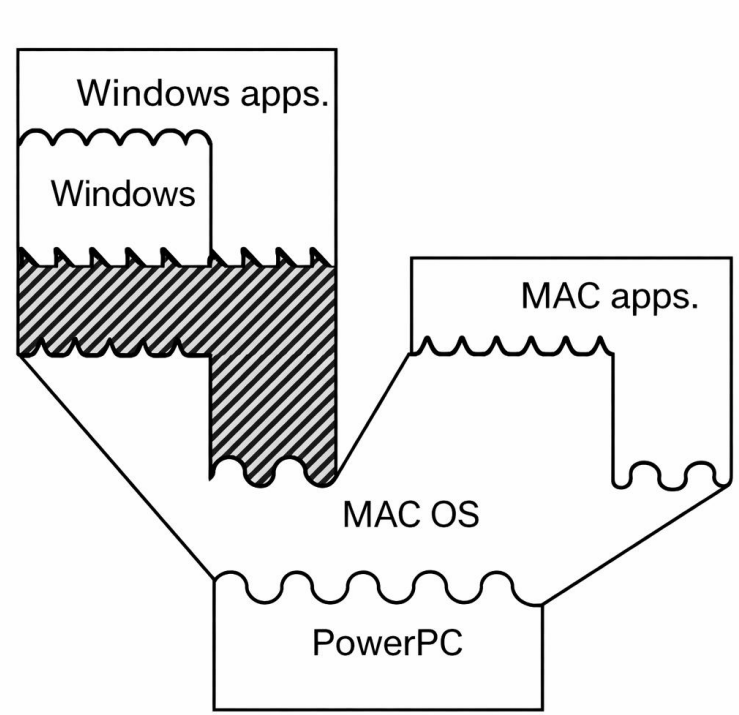




(c) composing virtual machine software to form a more complex, flexible system.

Whole System VMs: Emulation

- Host and Guest ISA are different
- So emulation is required
- Hosted VM + emulation
- E.g. Virtual PC (Windows on MAC)



Co-designed VMs

- The hypervisor is designed closely with (and possibly built into) a specific type of hardware ISA (or native ISA).
- Goal: Performance improvement of existing ISA (or guest ISA) during runtime.
- Hypervisor performs Emulation from Guest ISA to Native ISA.
  - E.g. Transmeta Crusoe
  - Native ISA based on VLIW
  - Guest ISA = x86
  - Goal power savings

Virtualizing individual resources in System VMs

CPU Virtualization for VMs

- Each VM sees a set of "virtual CPUs"
- Hypervisors must emulate privileged instructions issued by guest OS.
- Modern ISAs provide special interfaces for Hypervisors to run VMs
  - Intel provides the VTx interface
  - AMD provides the AMD-v interface
- These special ISA interfaces allow the Hypervisors to efficiently emulate privileged instructions executed by the guest OS.

CPU Virtualization for VMs

- When guest OS executes a privileged instruction
- Hardware traps the instruction to the hypervisor
- Hypervisor checks whether instruction must be emulated.
- If so, Hypervisor reproduces the effect of privileged operation.