DCS216 Operating Systems

Lecture 28
Virtualization

Jun 19th, 2024

Instructor: Xiaoxi Zhang

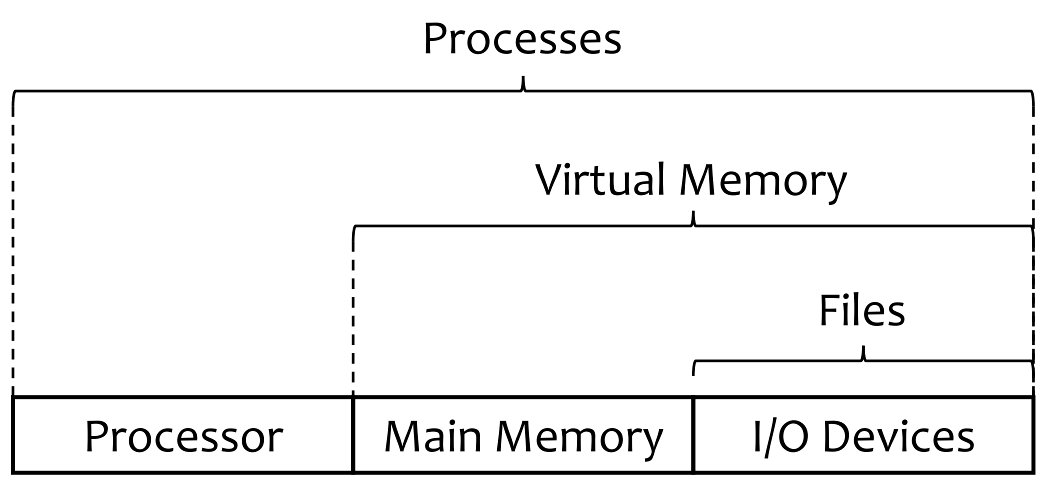
Sun Yat-sen University

Content

Content

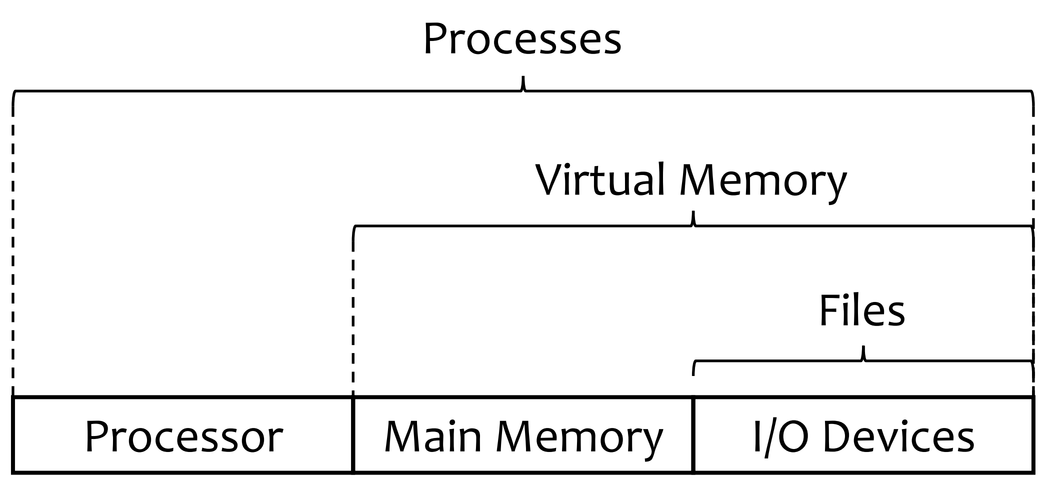
Overview

- The **Operating System** is all about **Abstractions**.
  - **Files** is an abstraction of I/O devices.
  - **Virtual Memory** is an abstraction of main memory and I/O devices.
  - **Processes** is an abstraction of CPU + Memory + I/O devices.



Overview

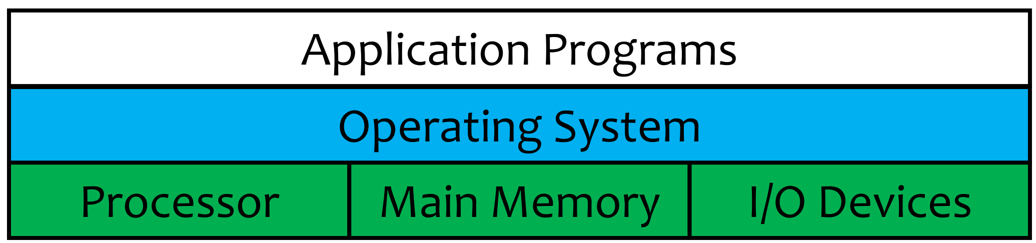
- The **Operating System** is all about **Abstractions** and **Virtualization**.
  - **Files** is an abstraction of I/O devices.
    - by virtualizing raw blocks into logical blocks (with File Systems, Device Drivers, etc.)
  - **Virtual Memory** is an abstraction of main memory and I/O devices.
    - by virtualizing physical addresses into logical addresses (via Page Tables, TLB, DMA, etc.)
  - **Processes** is an abstraction of CPU + Memory + I/O devices.
    - by virtualizing CPU states and related resources using PCBs.

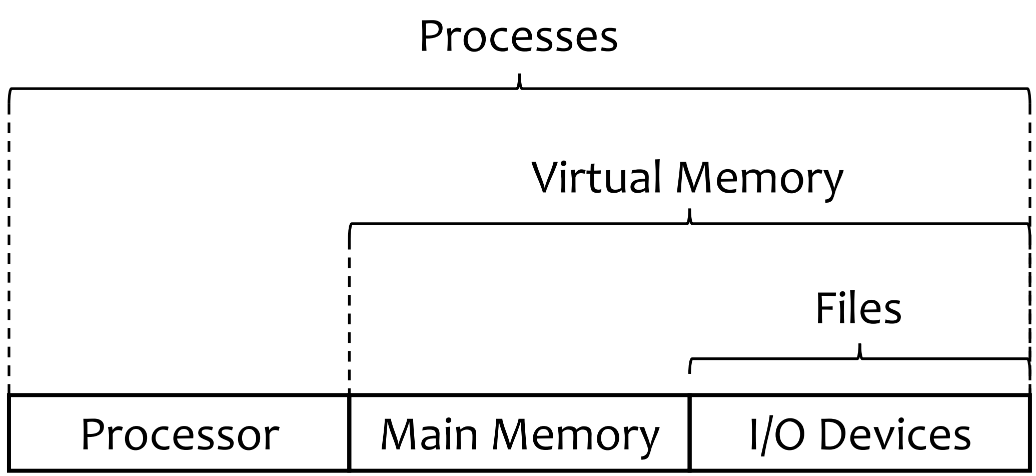


Overview

Overview

- The **Operating System** is the collection of **software** that **manages** the underlying ***hardware*** and **provides services** (system call APIs) to ***application programs***.





Overview

- The **Operating System** is the collection of **software** that **manages** the underlying ***hardware*** and **provides services** (system call APIs) to ***application programs***.

Application Programs		
Operating System		
Processor	Main Memory	I/O Devices

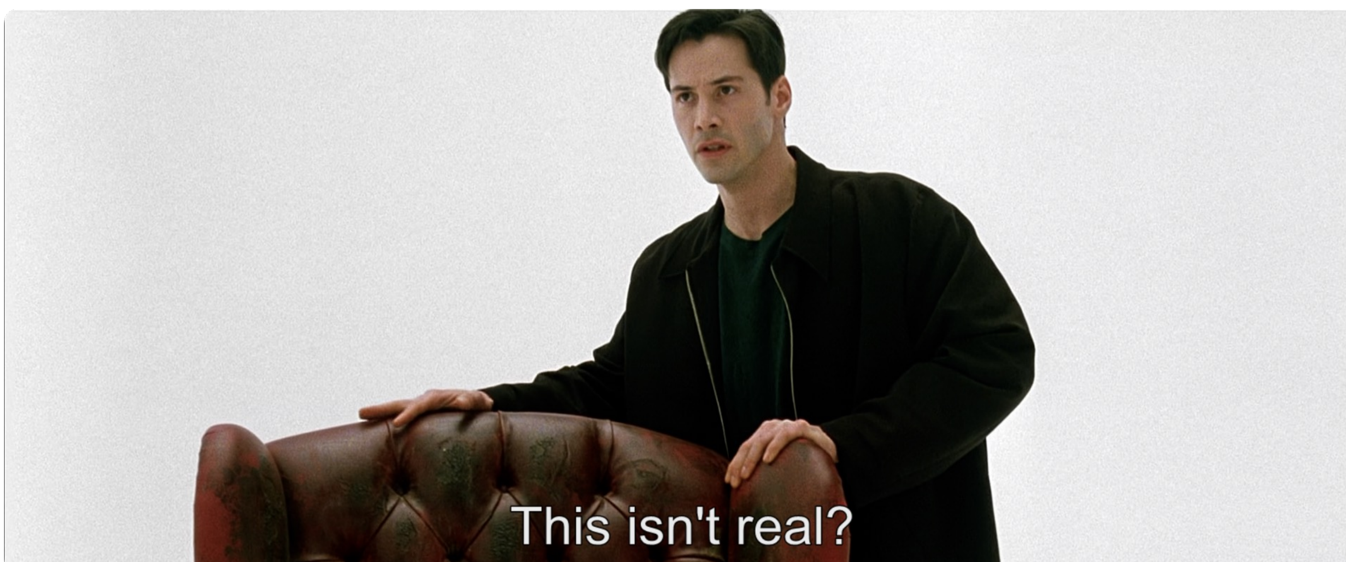
The "***Hardware***" may not necessarily be **REAL** (*in terms of physical form*).

Overview

Overview

- The **Operating System** is the collection of **software** that **manages** the underlying ***hardware*** and **provides services** (system call APIs) to ***application programs***.

Application Programs		
Operating System		
Processor	Main Memory	I/O Devices



Overview

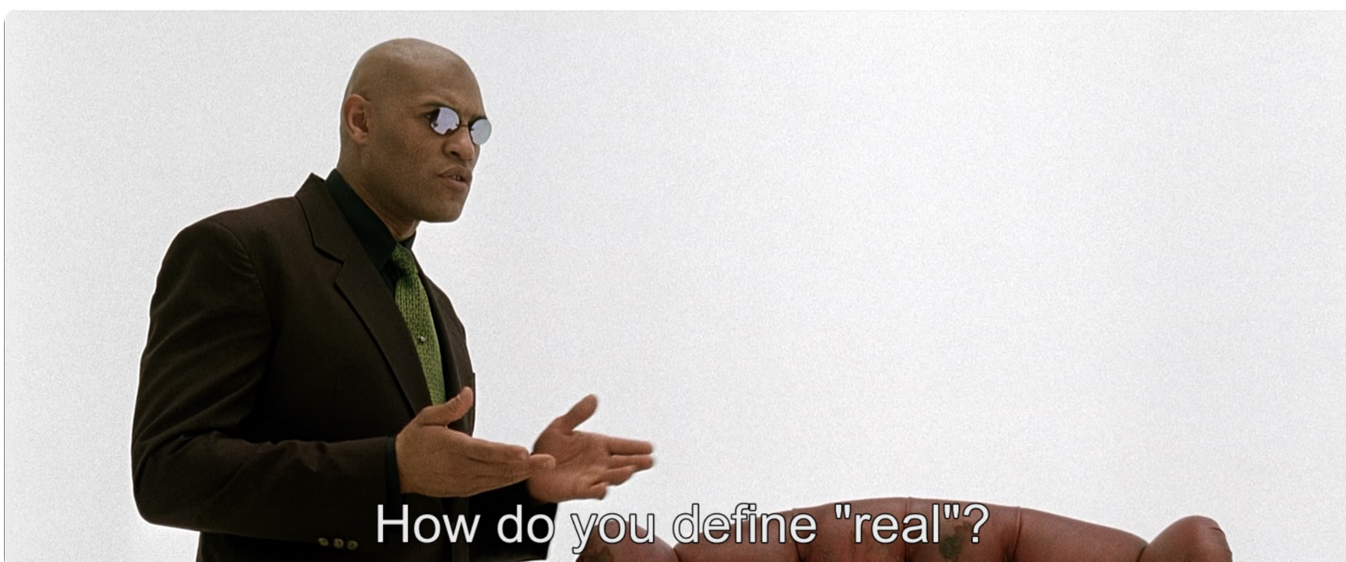
- The **Operating System** is the collection of **software** that **manages** the underlying ***hardware*** and **provides services** (system call APIs) to ***application programs***.

Application Programs		
Operating System		
Processor	Main Memory	I/O Devices

Overview

- The **Operating System** is the collection of **software** that **manages** the underlying ***hardware*** and **provides services** (system call APIs) to ***application programs***.

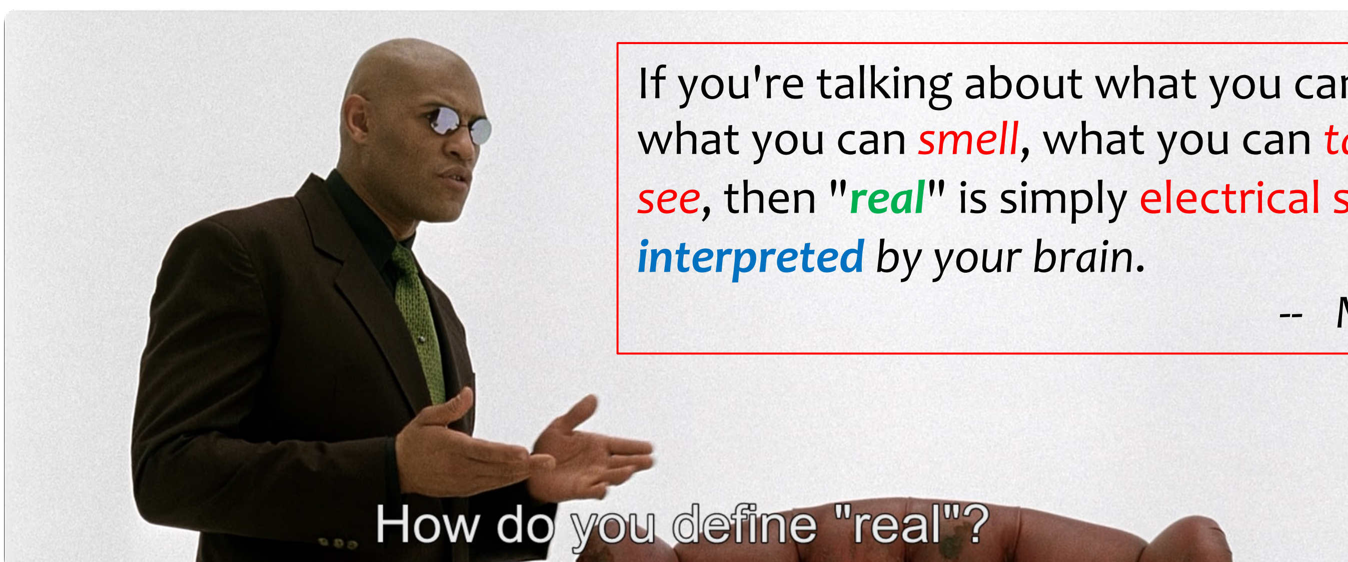
Application Programs		
Operating System		
Processor	Main Memory	I/O Devices



Overview

- The **Operating System** is the collection of **software** that **manages** the underlying ***hardware*** and **provides services** (system call APIs) to ***application programs***.

Application Programs		
Operating System		
Processor	Main Memory	I/O Devices



If you're talking about what you can *feel*, what you can *smell*, what you can *taste* and *see*, then "***real***" is simply electrical signals ***interpreted*** *by your brain.*

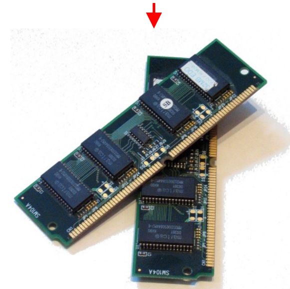
-- Morpheus

Overview

- The **Operating System** is the collection of **software** that **manages** the underlying ***hardware*** and **provides services** (system call APIs) to ***application programs***.

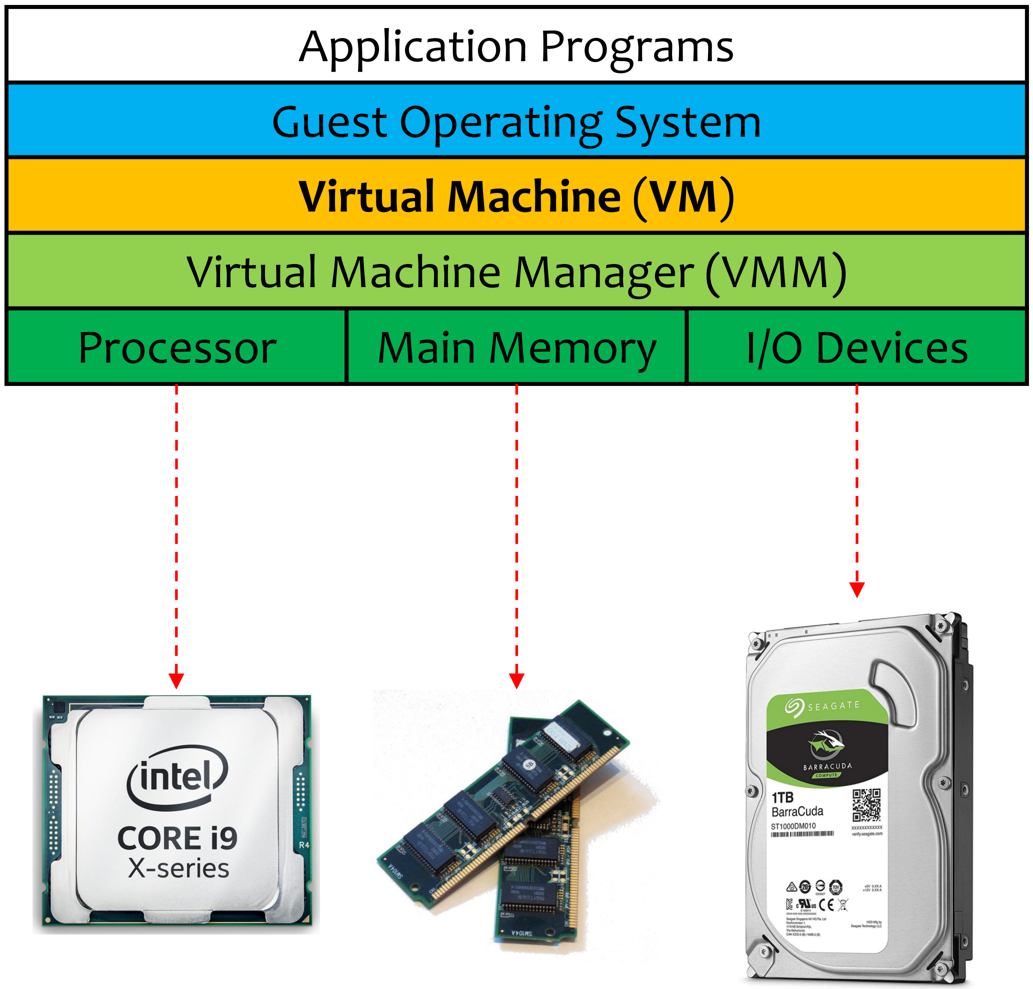
Application Programs		
Operating System		
Processor	Main Memory	I/O Devices

Similarly, "***hardware***" is simply electrical signals (0s and 1s) ***interpreted*** by the **OS**.



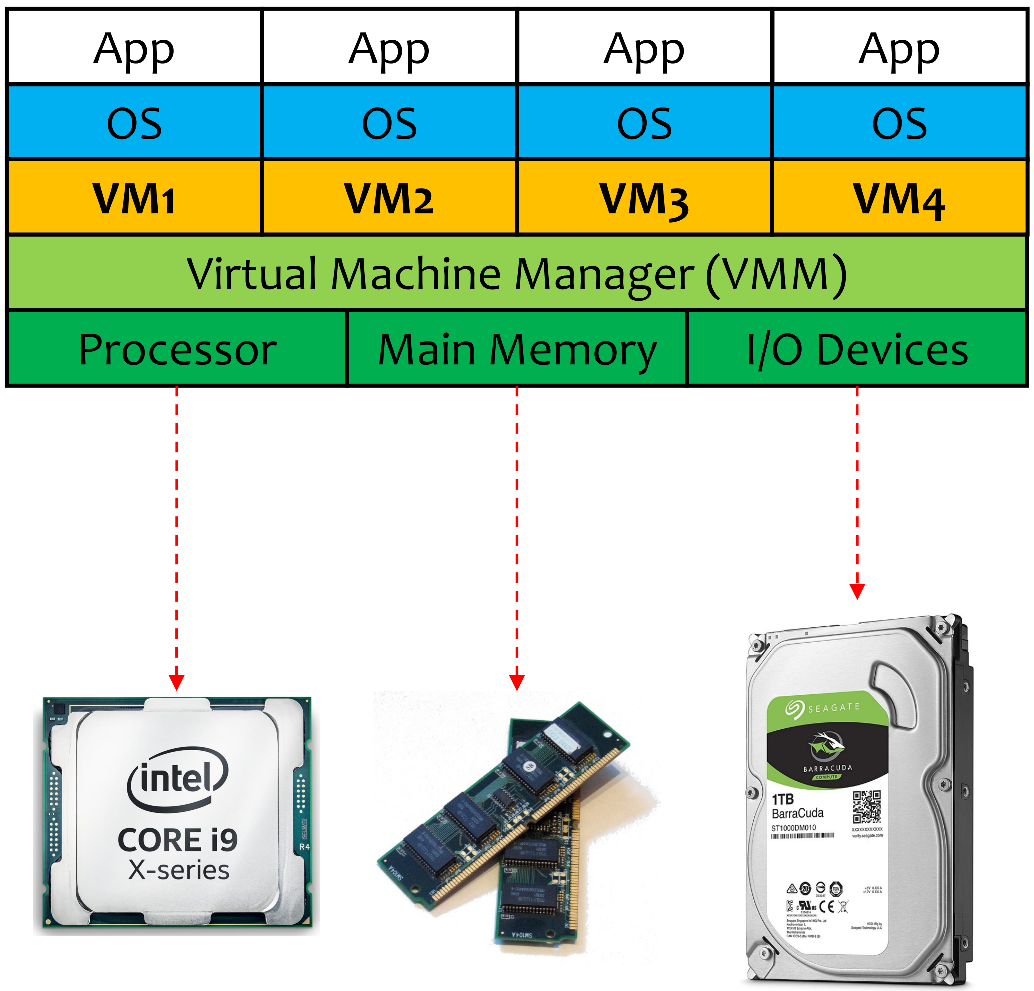
Overview

- A **Virtual Machine** (**VM**) is a software emulation of a physical computer. *It runs an OS just like a physical machine does.*



Overview

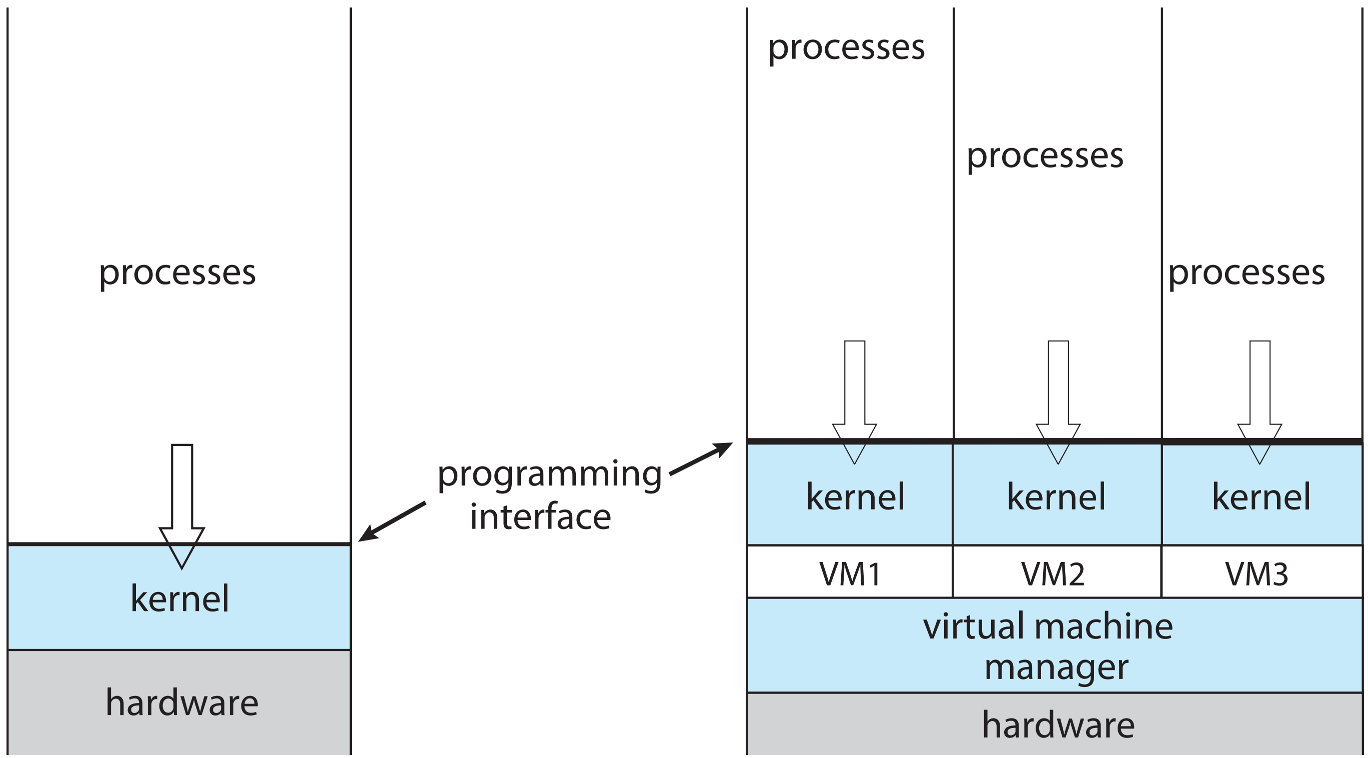
- A **Virtual Machine Manager (VMM)** multiplexes physical resources among the OSes in the VMs.



Overview

- **Overview**
  - Fundamental idea – abstract hardware of a single computer into several different execution environments
    - Similar to layered approach in OS architecture
    - But layer creates virtual system (virtual machine) on which OSes can run
  - Several components:
    - **Host**: underlying hardware(?) system.
    - **Virtual Machine Manager** (**VMM**) or Hypervisor: creates and runs VMs by providing an **interface** that is identical to the **host.**
    - **Guest**: Each guest process is provided with a *virtual copy* of the **host.** Usually, the guest process is in fact an Operating System.
  - Single physical machine can run multiple OSes concurrently, each in its own virtual machine.

System Models



Physical Machine

Virtual Machine

History

- First appeared in IBM mainframes (IBM大型机) in 1972.
  - IBM **VM/370** divided (simple hardware-level multiplexing) a mainframe into multiple virtual machines, each running its own OS.
  - The hard part is **virtualizing** disks – **minidisks**.
- Formal definition of virtualization helped to establish system requirements and a target for functionality:
  - **Fidelity**. A VMM provides an environment for programs that is essentially identical to the original machine.
  - **Performance**: Programs running within that environment show only minor performance decreases.
  - **Safety**: The VMM is in complete control of system resources.
- In late 1990s, Intel CPUs fast enough for researchers to try virtualizing on general purpose PCs
  - **Xen** and **VMware** created virtualization technologies (still used today)
  - **Virtualization** has expanded into many OSes, CPUs, VMMs.

- **Benefits and Features**
  - Host system **protected** from VMs, VMs **protected** from each other
    - A virus that infected one of the VMs is not likely to spread to other VMs, nor the host system.
    - Sharing is provided via shared file system volume, or networking.
  - Freeze, suspend, running **VM**s
    - They can move or copy somewhere else and resume.
    - Snapshot of a given state, able to restore back to that state
      - Some hypervisors allow multiple snapshots per VM
    - Clone by creating copies and running both original and the copies.
  - Great for OS research, better system development efficiency
  - Run multiple, different OSes on a single machine
    - Consolidation, better utilization, etc.

- **Benefits and Features**
  - **Templating (虚拟机模版)** – create an OS + application VM, provide it to customers
    - use it to create multiple instances of that combination quickly.
  - **Live Migration (热迁移)** – move a running VM from one host to another
    - No interruption of services during migration
  - All those features taken together → **Cloud Computing**
    - Use APIs to instruct cloud infrastructure (e.g., AWS, Azure) to create new VMs or web applications.

Types of VMs

- **Implementation of VMMs**
  - **Type 0** hypervisors
    - hardware-based, e.g.,
  - **Type 1** hypervisors
    - bare-metal, OS-level, e.g., KVM, Hyper-V, VMware ESXi, etc.
  - **Type 2** hypervisors
    - application-level, process-based, e.g., VirtualBox, Parallels Desktop
  - **Paravirtualization**
    - E.g., Xen
  - **Programming Environment Virtualization**
    - E.g., **JVM** (Java Virtual Machine)
  - **Emulators**
    - E.g., QEMU, Bochs
  - **Application Containment** (**Containers**)
    - E.g., LXC, Docker, Kubernetes
  - ...

Building Blocks

- Although the concept of **VM** is useful, it is difficult to implement.
  - Much work is required to provide an ***exact*** duplicate of the underlying (*host*) machine *to make the guest believe it is running on hardware*.
  - For example, on a **dual-mode** system where the CPU has only *user-mode* and *kernel-mode*, how can you **virtualize** the "***kernel-mode***" for the Guest OS?
- Most VMMs implement **virtual CPU** (**vCPU**) to represent state of CPU of the Guest.
  - The **vCPU** does not execute code.
  - It represents the state of the CPU as Guest machine *believes it to be.*
  - When the Guest is context-switched onto a CPU by the VMM, information from the **vCPU** is used to load the corresponding context.
    - much like the **PCB** in a general purpose OS.

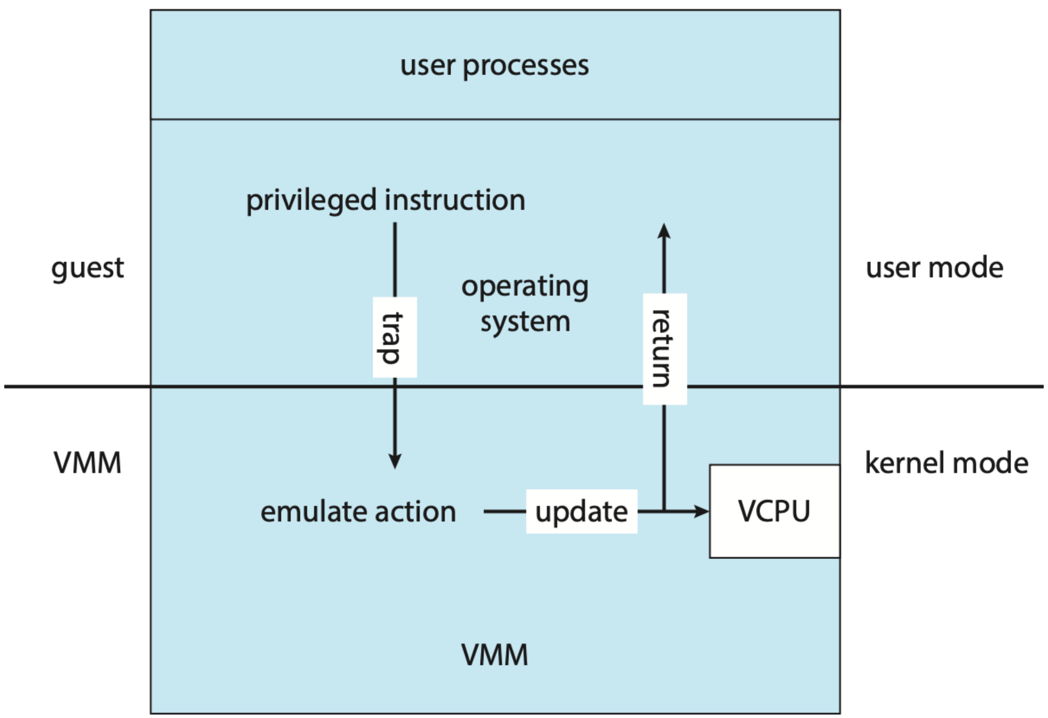
Building Blocks

- **Trap-and-Emulate**
  - Dual-mode CPU means Guest **only** executes in physical *user-mode*.
  - The kernel of the Host OS runs in physical *kernel-mode*.
    - Not safe to let Guest kernel run in physical *kernel-mode*, too. **Why?**
    - No way for us to keep control of the physical machine!
    - No way to keep the Guest away from physical devices!
    - Once we let the Guest kernel run in physical kernel-mode, we can never get back control, *e.g., Guest runs **I/O** instructions to overwrite the VMM*.

- **Trap-and-Emulate**
  - Dual-mode CPU means Guest **only** executes in physical *user-mode*.
  - The kernel of the Host OS runs in physical *kernel-mode*.
    - Not safe to let Guest kernel run in physical *kernel-mode*, too.
  - Consequently, we must invent **two modes** for the **Virtual Machine**:
    - virtual *user-mode* and virtual *kernel-mode*.
      - both of which run in physical *user-mode*.
  - Those actions that cause a **transfer** from *user-mode* to *kernel-mode* on a physical machine (e.g., syscalls, interrupts, or an attempt to execute a privileged instruction) must also cause a **transfer** (in the Virtual Machine) from virtual *user-mode* to virtual *kernel-mode*.
  - How can such a **transfer** be accomplished?

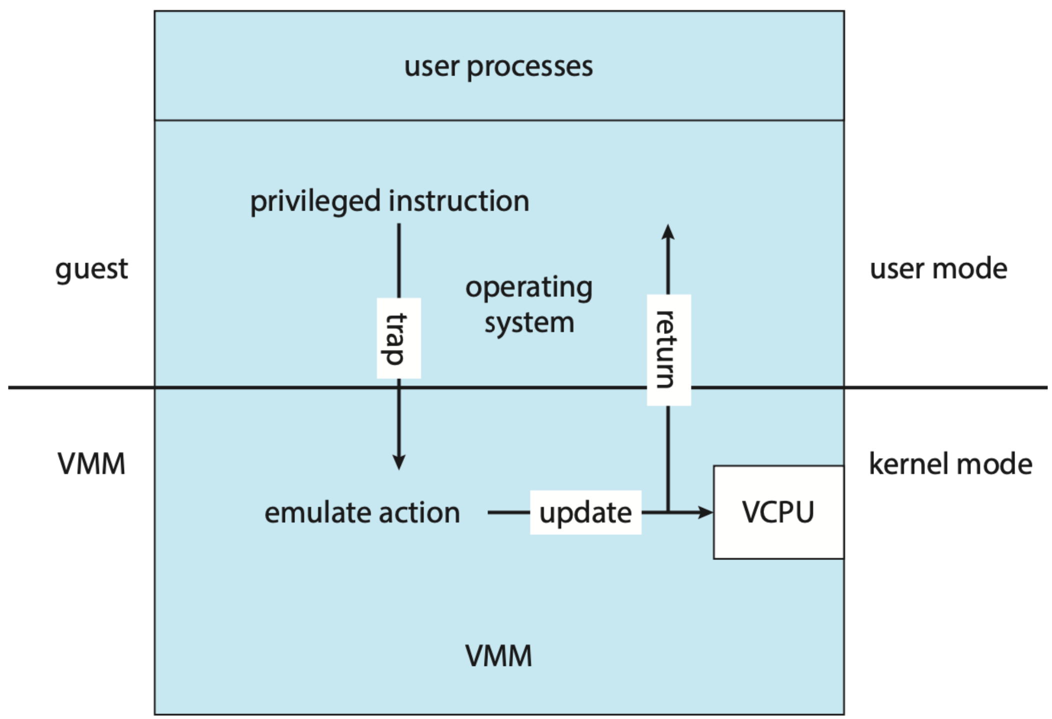
Trap-and-Emulate

- How can such a **transfer** be accomplished? (**Trap-and-Emulate**)
  - When the **Guest** attempts a privileged instruction in user-mode, it would cause an error → traps into the VMM in Host machine.
  - VMM gains control, analyzes error, executes operations as attempted by the **Guest**
  - Returns control to **Guest** in user-mode.



Trap-and-Emulate

- How can such a **transfer** be accomplished? (**Trap-and-Emulate**)
  - For example, suppose the Guest issues a privileged instruction to modify a certain CPU flag → traps into the VMM
  - The VMM handles such a trap by emulating the privileged instruction on the **vCPU** that corresponds to the specific Guest, rather than executing it on *physical CPU*: **Maintain the illusion**.



Binary Translation

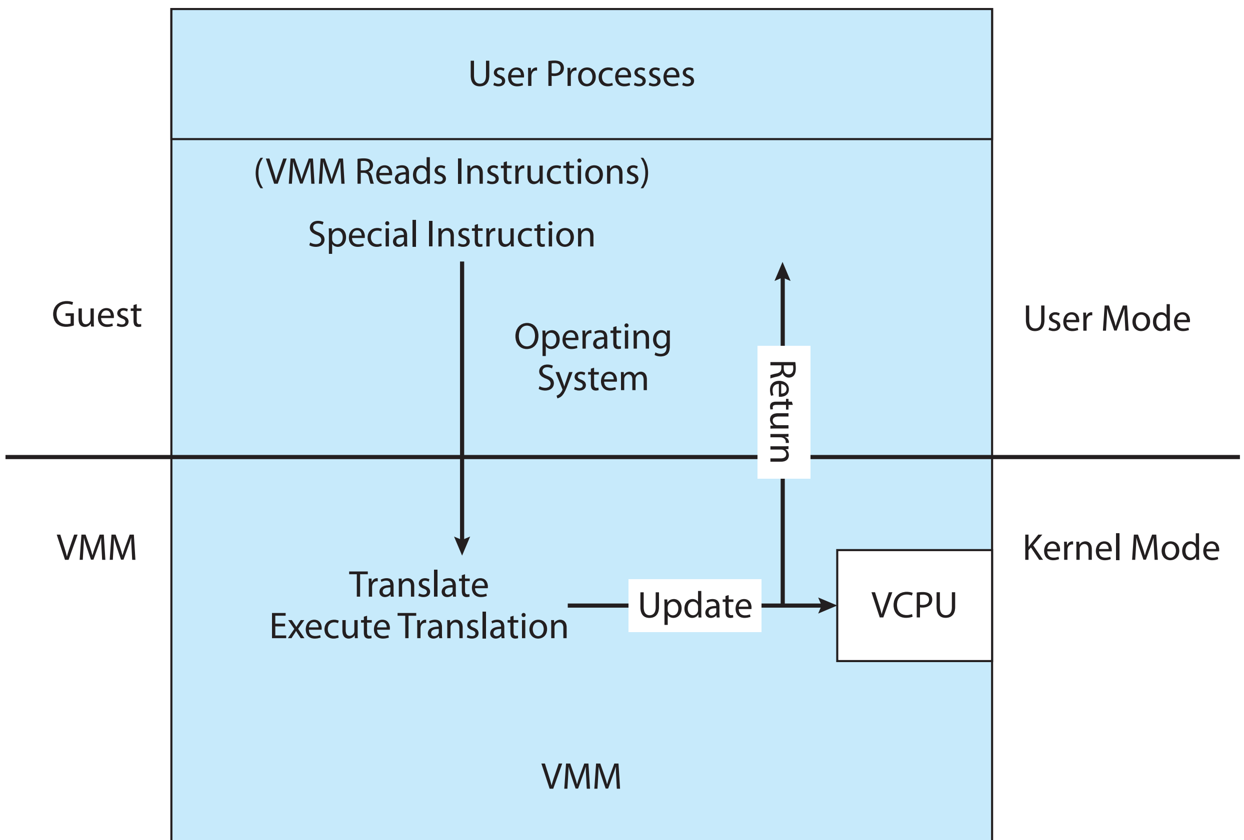
- Some CPUs don't have a clear separation between privileged and non-privileged instructions.
  - Earlier Intel x86 CPUs are among them
    - In fact, the earliest Intel CPU was designed for a calculator.
  - Intel CPUs are backward compatible → lack of separation persists
  - Consider the `popf` instruction in Intel x86:
    - Loads CPU flags register from the contents of the stack.
    - If CPU is in privileged mode → all flags replaced
    - If CPU is in user mode → only some flags replaced
      - No trap is generated. Therefore the aforementioned **trap-and-emulate** procedure is useless.
- Some other x86 instructions cause similar problems, we call this set of instructions ***special instructions***.

Building Blocks

- **Binary Translation**
  - Some other x86 instructions cause similar problems, we call this set of instructions ***special instructions***.
    - caused **trap-and-emulate** method considered impossible until 1998.
  - Binary translation solves this problem:
    - Concept is simple, but implementation very complex
    - If Guest vCPU is in user-mode, Guest can run instructions natively
    - If Guest vCPU is in kernel-mode (Guest believes it is in kernel-mode)
      - VMM examines every instruction that the Guest is about to execute by reading a few instructions ahead of program counter.
      - Non-special instructions run natively
      - ***Special instructions*** translated into **new set of instructions** that perform equivalent tasks (for example, changing the flags in the **vCPU**)

Binary Translation

- Binary translation is implemented by **translation code** within the VMM. The code reads native binary instructions dynamically from the Guest, on demand, and generates native binary code that executes in place of the original code.



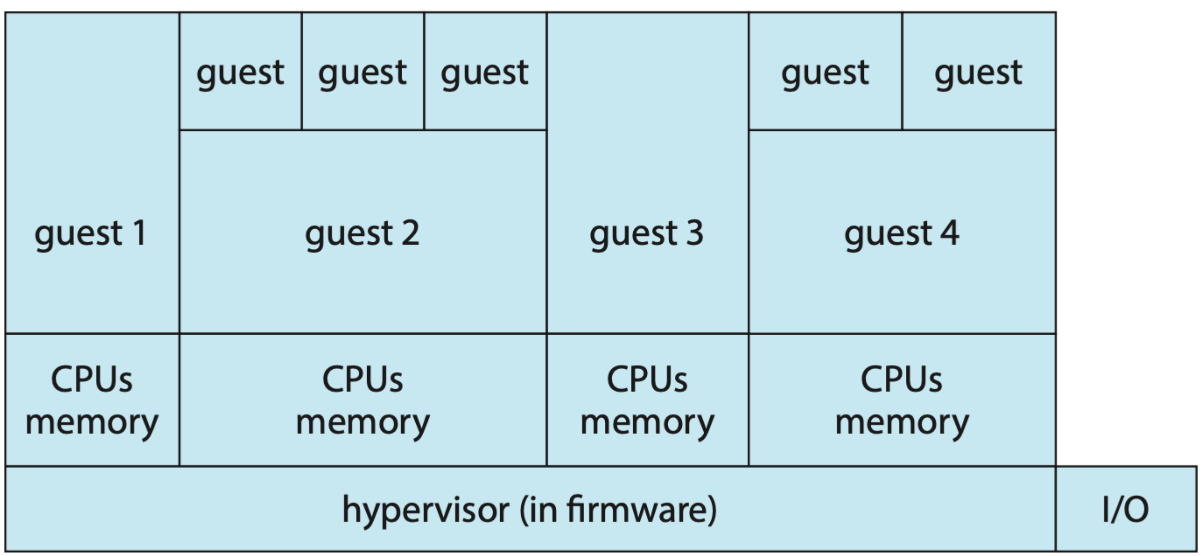
Hardware Assistance

- All virtualization needs some hardware support
  - More support $\Rightarrow$ more feature rich, stable, better performance
- Intel added new **VT-x** instructions in 2005, AMD added the **AMD-V** instructions in 2006
  - CPUs with these instructions ***remove the need for binary translation.***
  - Generally define more CPU modes: "**guest**" and "**host**"
    - from dual-mode to four-mode:
      - Host Mode; Host User Mode
      - Guest Kernel Mode; Guest User Mode
  - VMM can enable **host** mode, define characteristics of each guest VM, switch to **guest** mode, passing control to a Guest OS.
  - In **guest** mode, the Guest OS thinks it is running natively
    - Access to virtualized device, priv instructions cause trap to VMM
    - CPU maintains vCPU, context switches as needed
- Hardware support for Nested Page Tables, DMA, interrupts as well.

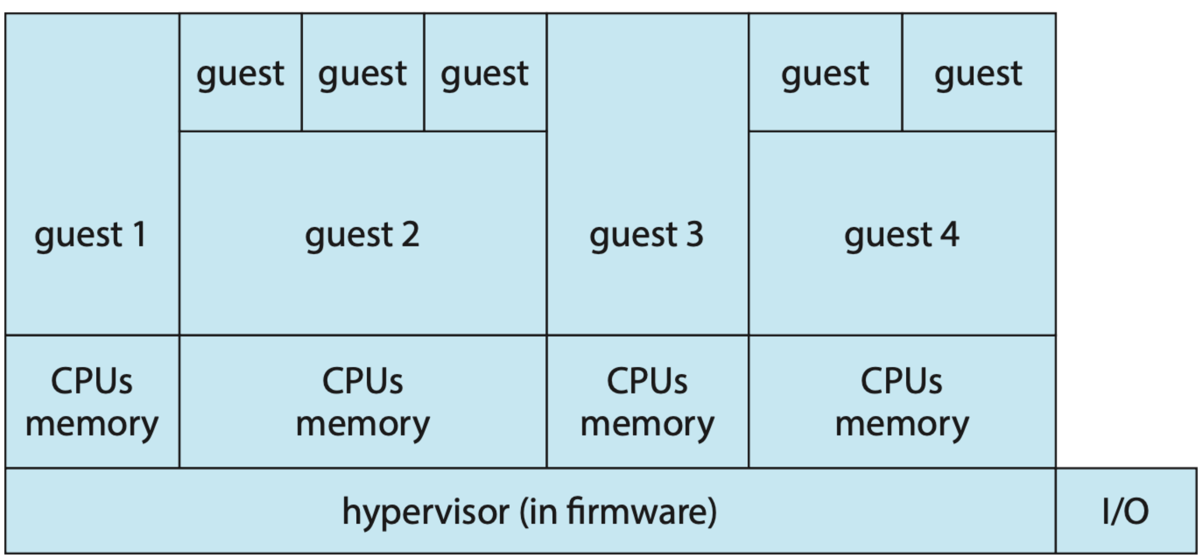
Virtual Machine Life Cycle

- Whatever the hypervisor type, a typical VM has a life cycle:
  - Created by VMM (hypervisor)
  - Resources assigned to VM
    - # of CPU or vCPU cores
    - # of memory
    - # of disk space
    - networking details (# of NICs, bandwidth requirements, etc.)
    - ...and so on
  - Resources are usually **shared** (multiplexed, or provisioned) among VMs
    - except for type 0 hypervisor, where resources are usually dedicated
  - When the VM is no longer needed, it can be deleted or suspended.

- **Type 0 Hypervisor**
  - Historically also called "*partitions*", "*domains*".
  - A Hardware feature implemented by **firmware**.
    - The **VMM** itself is encoded in the **firmware** rather than in OS software.
    - At boot time, the **VMM** loads the guest images to run in each "partition"
    - Provides much less features compared with other types of hypervisors.
    - Each guest *believes* it has **dedicated** hardware (*because it really does*), simplifying many implementation details. ⇒ much less flexible.
  - Each guest is basically running on raw hardware.



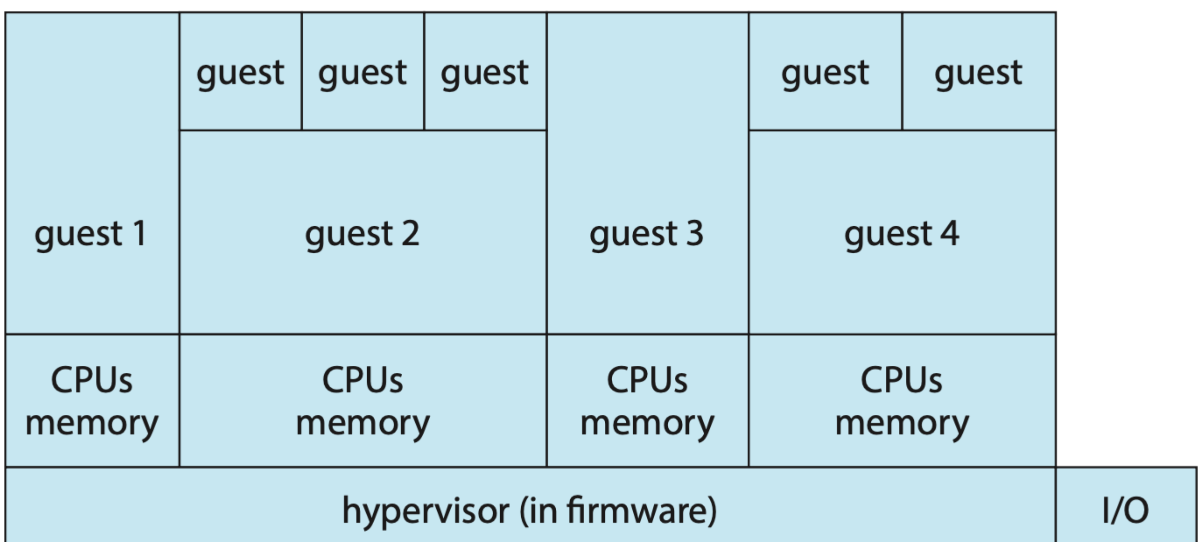
- **Type 0 Hypervisor**
  - Each guest is basically running on raw hardware.
    - Close or identical to the underlying hardware in performance.
      - Each guest itself can even be a VMM that supports more VMs
    - **I/O** is a huge challenge:
      - must have enough devices, controllers to dedicate to each guest.
      - For example, the underlying hardware must have at least 3 CPU cores in order to support 3 running guests, or at least 2 Ethernet ports to dedicate to two separate guests.



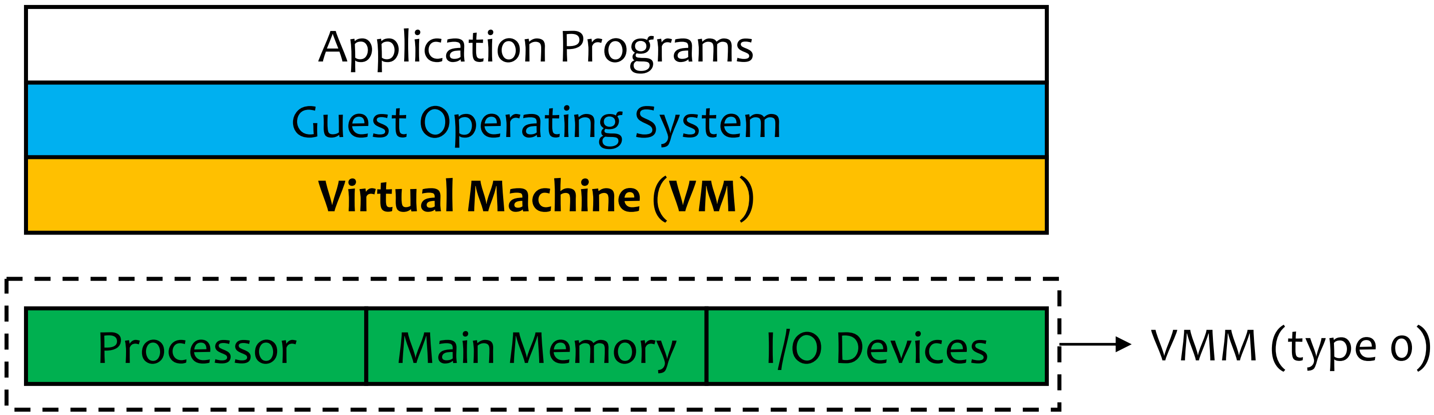
Types of VMs

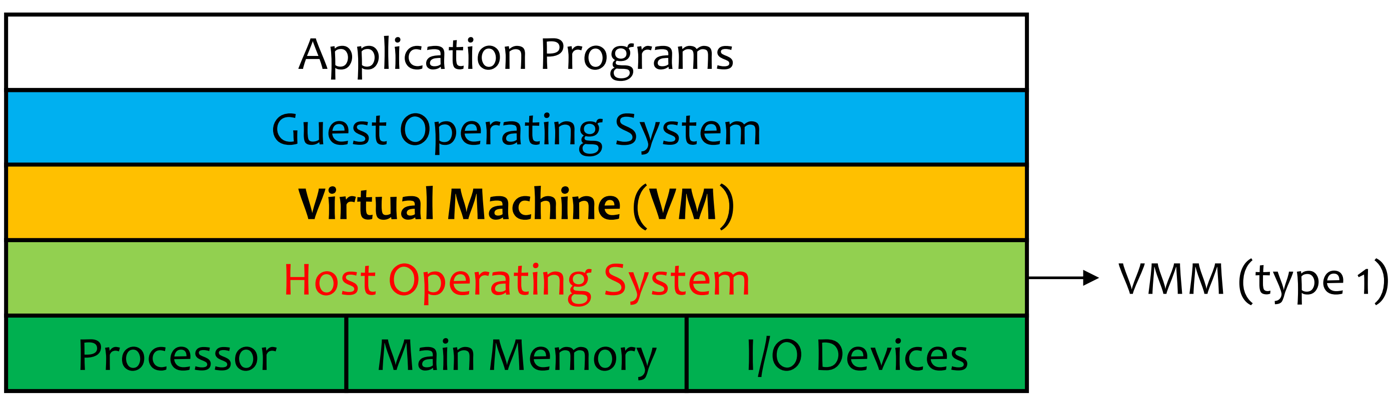
Type 0 Hypervisor

- It is rare to find a *pure* "**Type-0**" hypervisor nowadays.
  - Implemented in hardware (firmware) ⇒ Too inflexible.
  - Hardware resources dedicated to each guest ⇒ Low overall utilization.
  - Whatever a "**Type-0**" hypervisor can do, a "**Type-1**" can do it, too.
    - Outside the textbook, the term "**Type-0**" hypervisor is not widely recognized in the standard classification of hypervisors in the computing industry, i.e., they would instead be categorized as "**Type-1**".



- **Type 1 Hypervisor**
  - **Type 1** Hypervisor shifts the **VMM** functionalities from *hardware* (**firmware**) to *software* (**OS**).

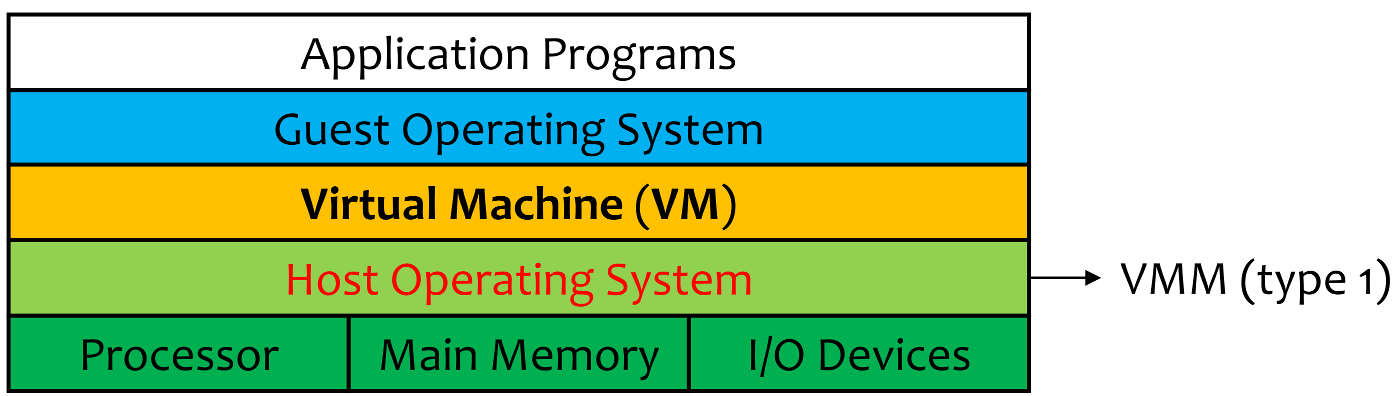




Types of VMs

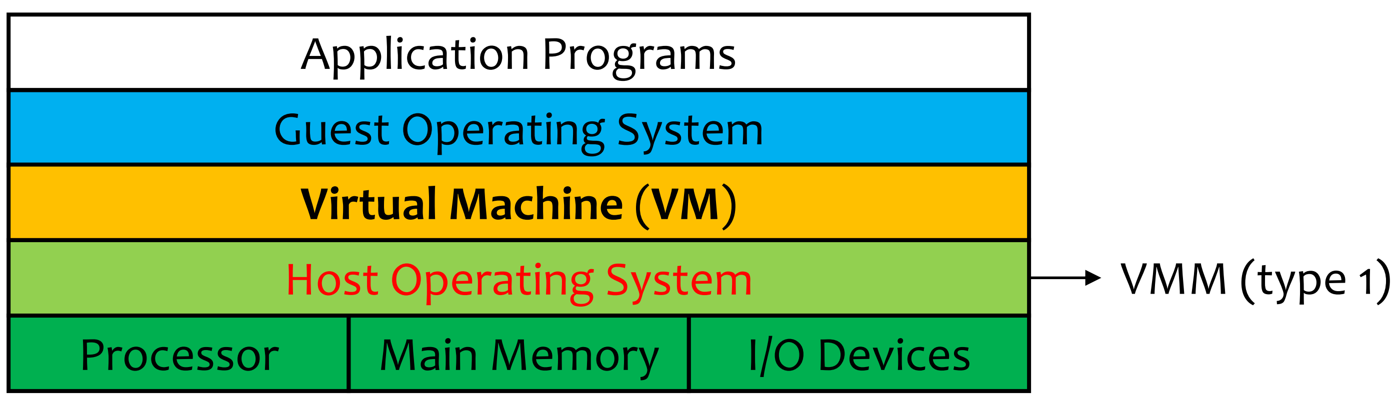
Type 1 Hypervisor

- **Type 1** Hypervisor shifts the **VMM** functionalities from *hardware* (**firmware**) to *software* (**OS**).
  - Special purpose OS that runs natively on hardware ⇒ in kernel mode.
    - Rather than providing syscall APIs for application programs, it exports **APIs** for VM management (creation, deletion, etc.)
  - Can also be general purpose OS that ***integrates*** **VMM** functionalities.
    - E.g., a typical Linux OS *with* ***KVM*** *module enabled* can be used as a **Type-1** hypervisor ⇒ the most popular Open Source hypervisor solution available today.
  - In many ways, the **guest OS** is treated as just another **process**.



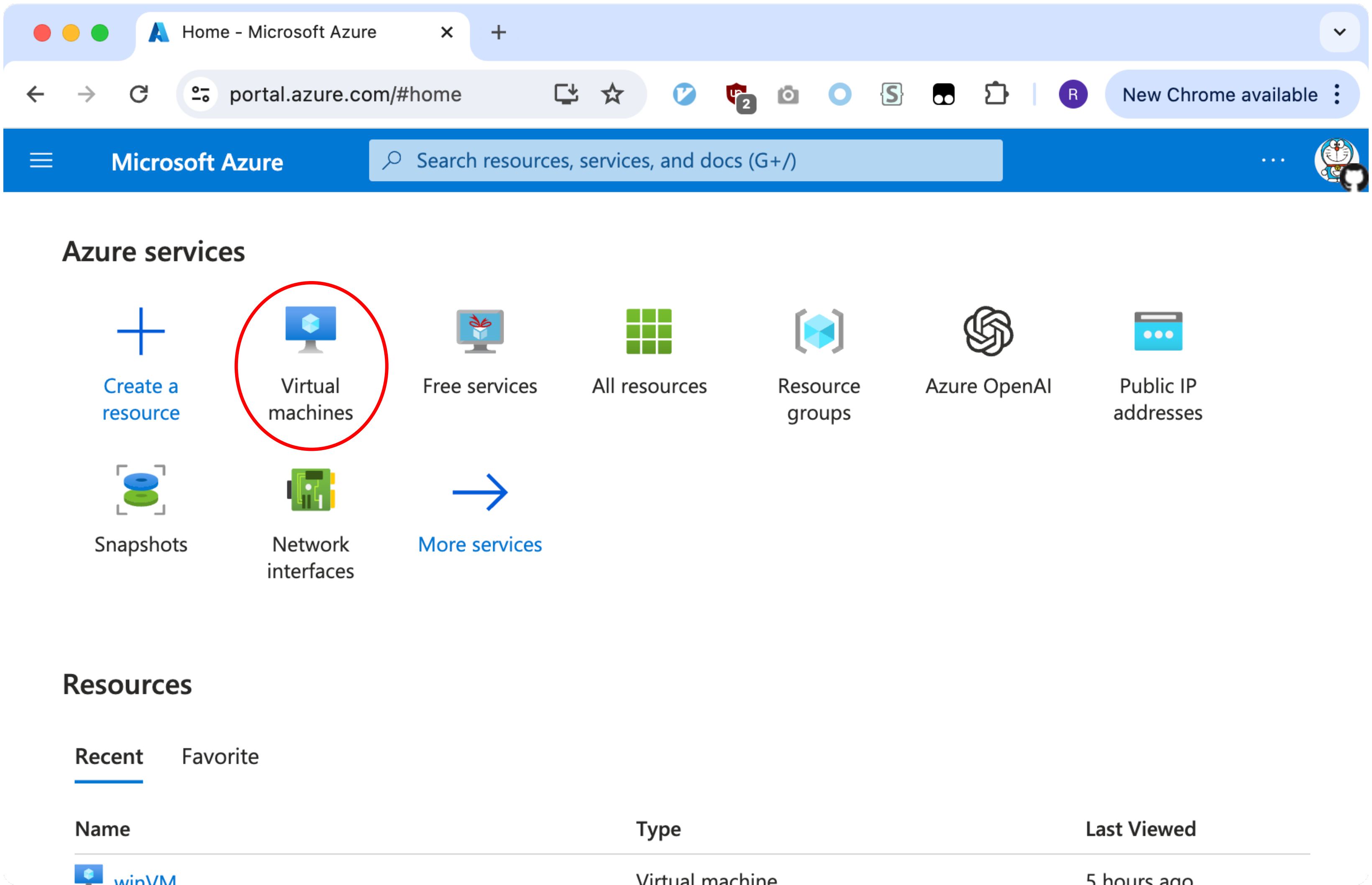
Types of VMs

- **Type 1 Hypervisor**
  - **Type 1** Hypervisor also called "the Data-center Operating System".
    - Data-center managers control and manage OSes in new, sophisticated ways by controlling the Type-1 hypervisor.
    - Consolidation of multiple OSes and apps onto less hardware.
    - Move guests between systems to balance performance
    - Snapshots and cloning
    - The cornerstone of **IaaS** Cloud Computing.



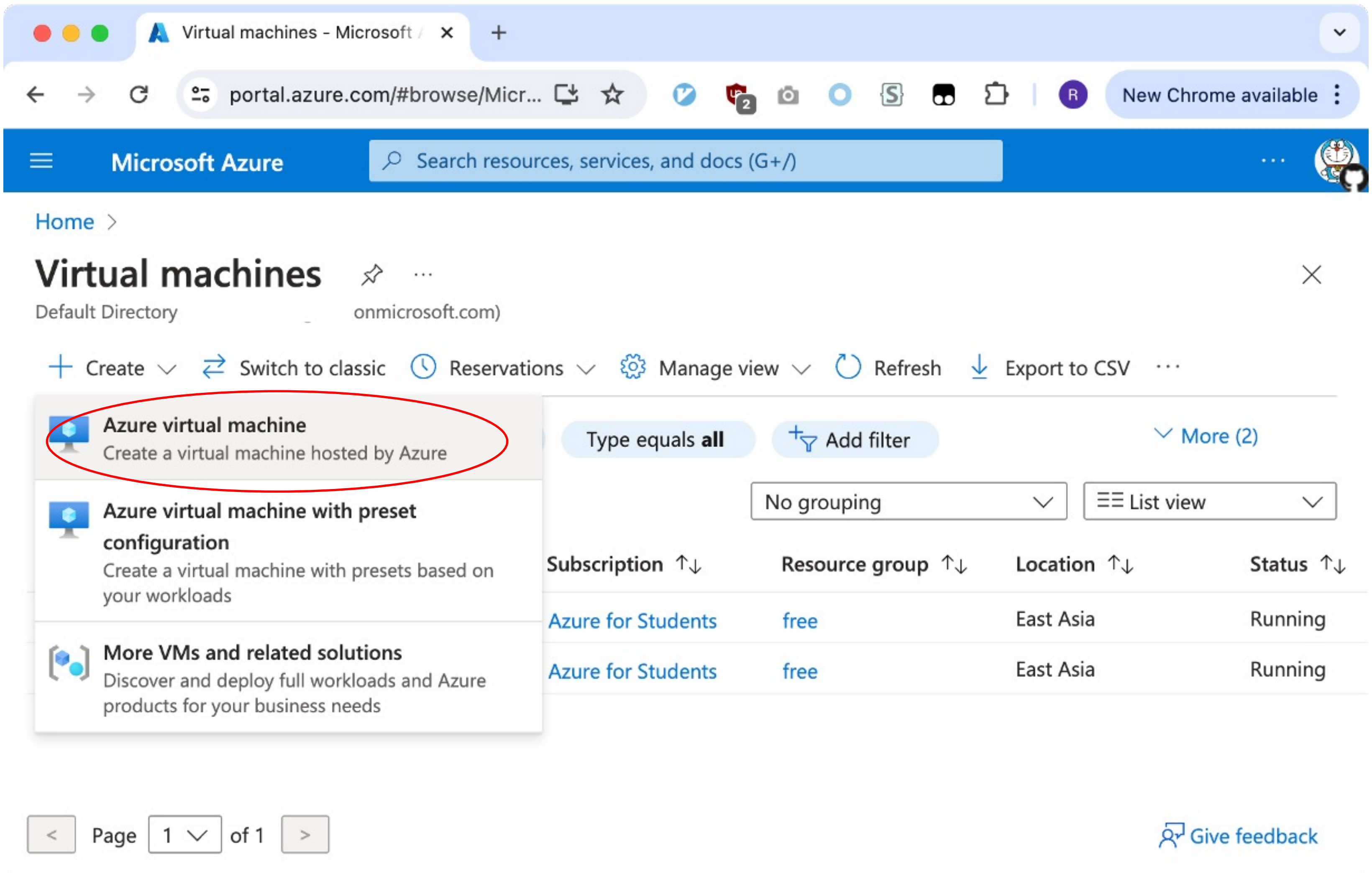
Types of VMs

- **Type 1 Hypervisor**
  - The cornerstone of **IaaS** Cloud Computing.



Types of VMs

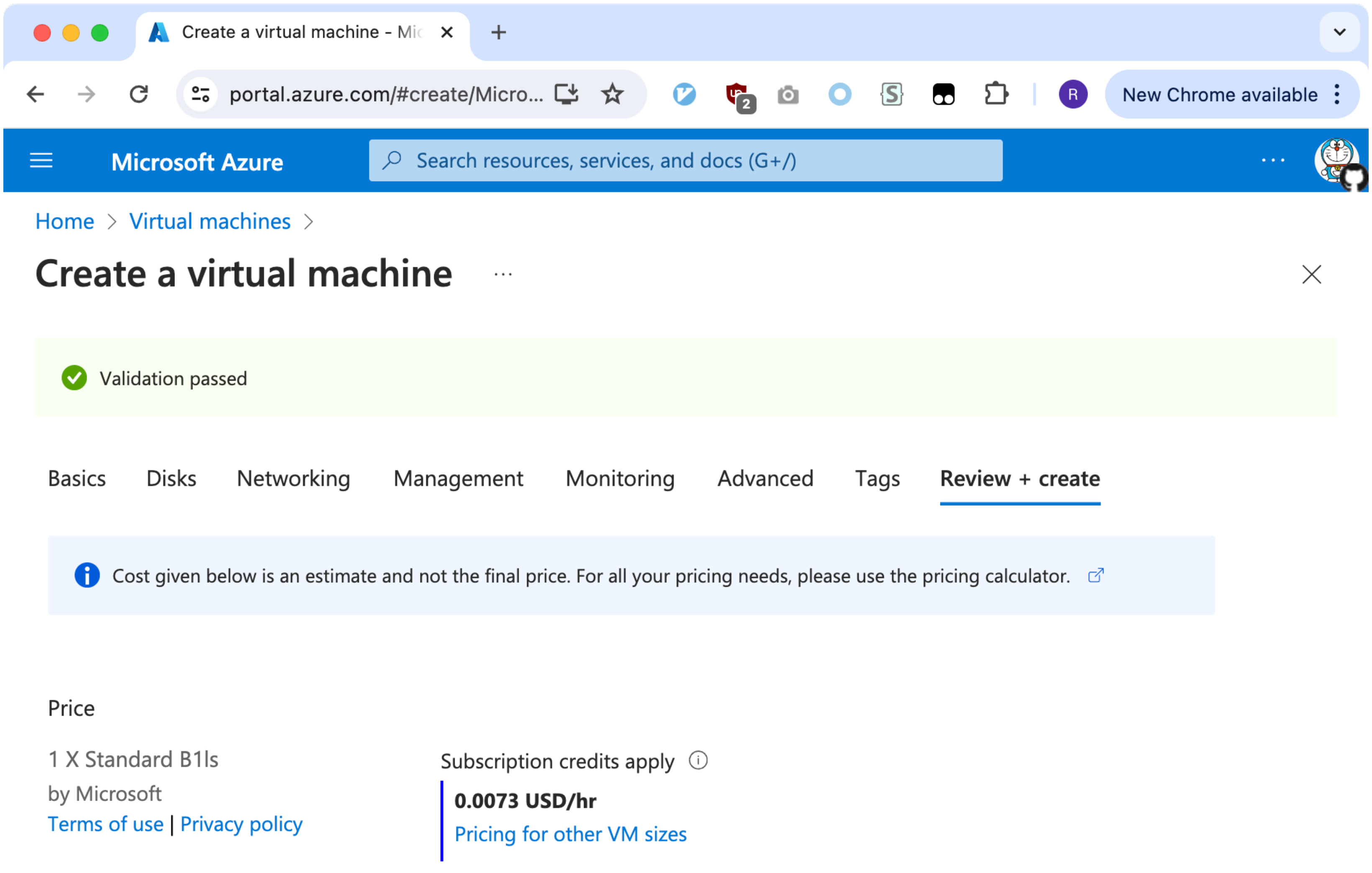
- **Type 1 Hypervisor**
  - The cornerstone of **IaaS** Cloud Computing.



Types of VMs

Type 1 Hypervisor

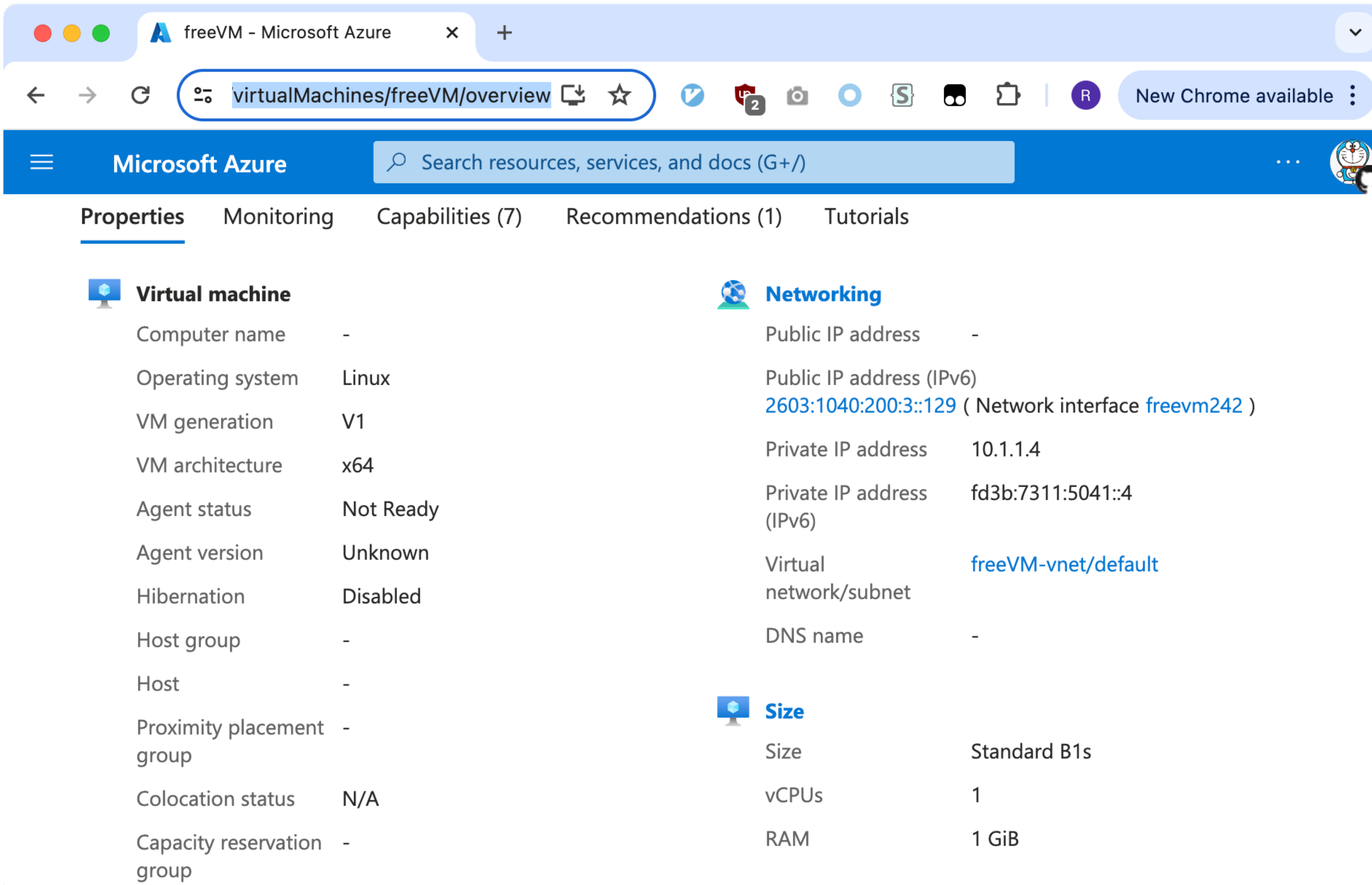
- The cornerstone of **IaaS** Cloud Computing.



Types of VMs

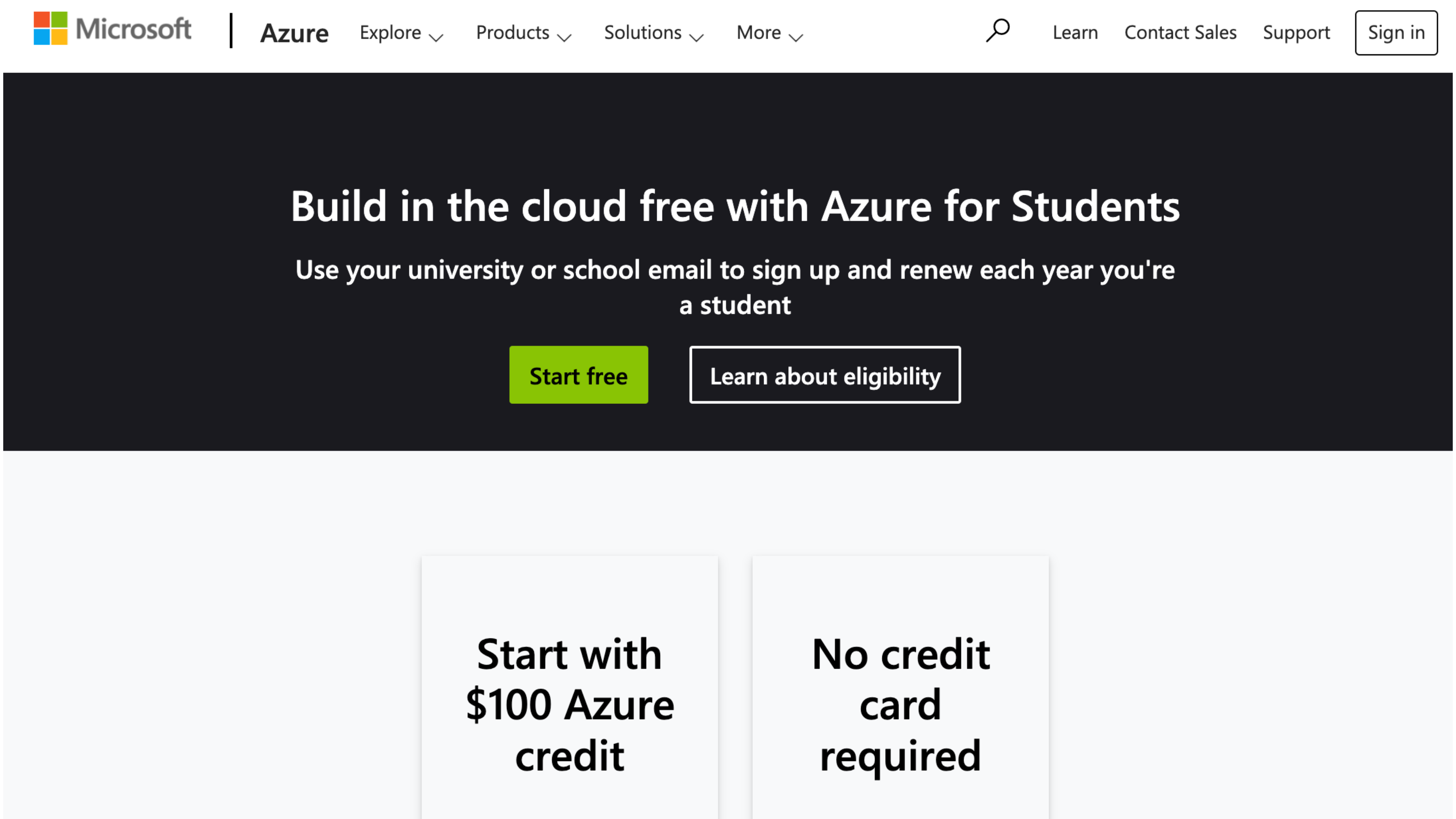
Type 1 Hypervisor

- The cornerstone of **IaaS** Cloud Computing.



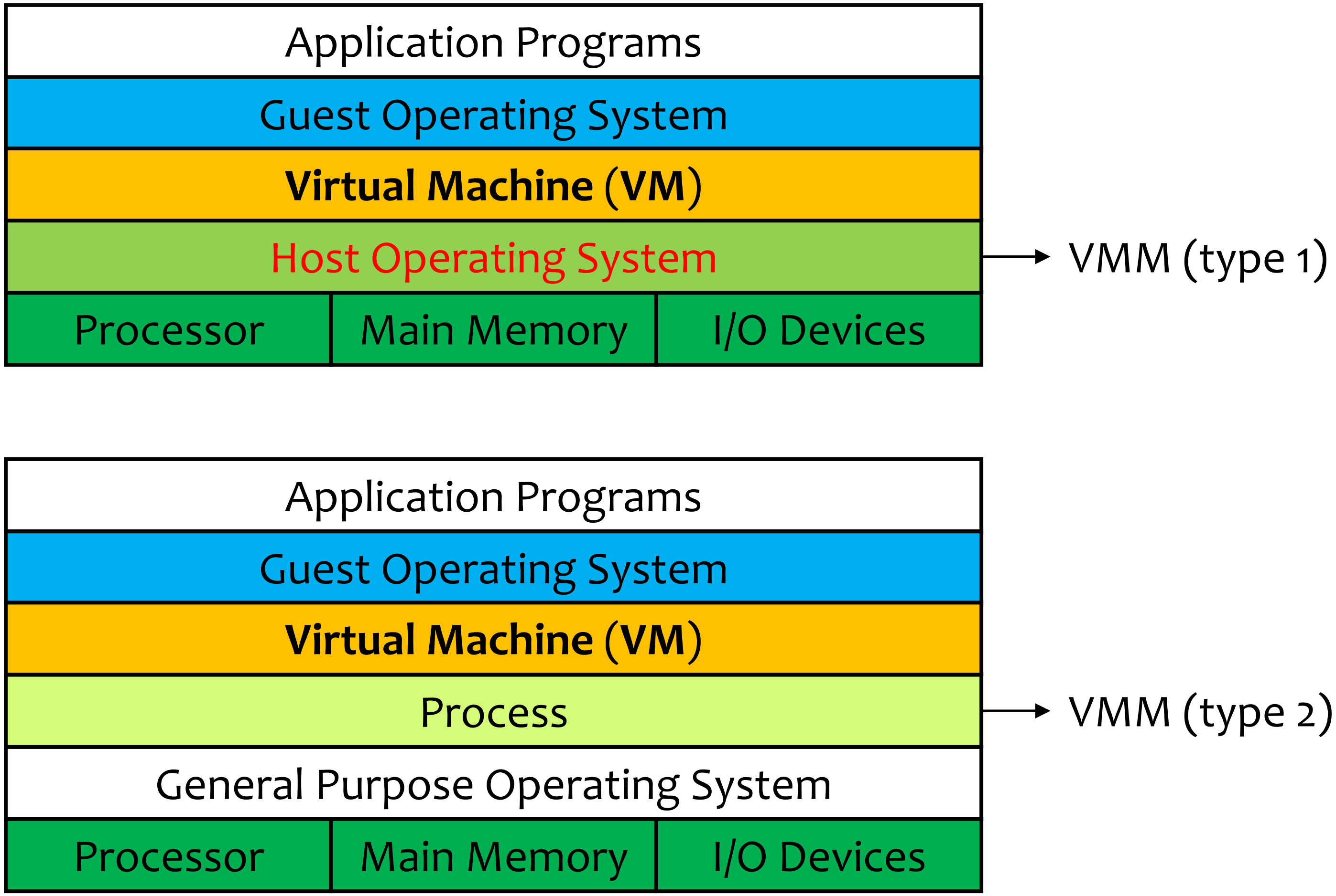
Types of VMs

- **Type 1 Hypervisor**
  - **Recommended:** Create an Azure Student account
    - https://azure.microsoft.com/en-us/free/students



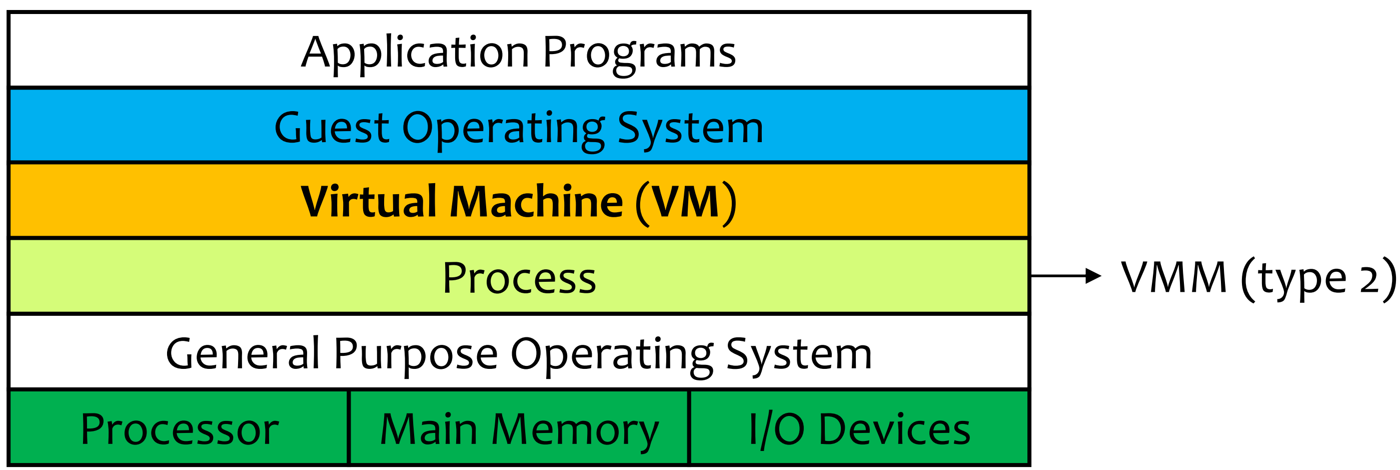
Type 2 Hypervisor

- **Type 2** Hypervisor shifts the **VMM** functionalities from *kernel-mode* **OS** to *user-mode* **Processes**.

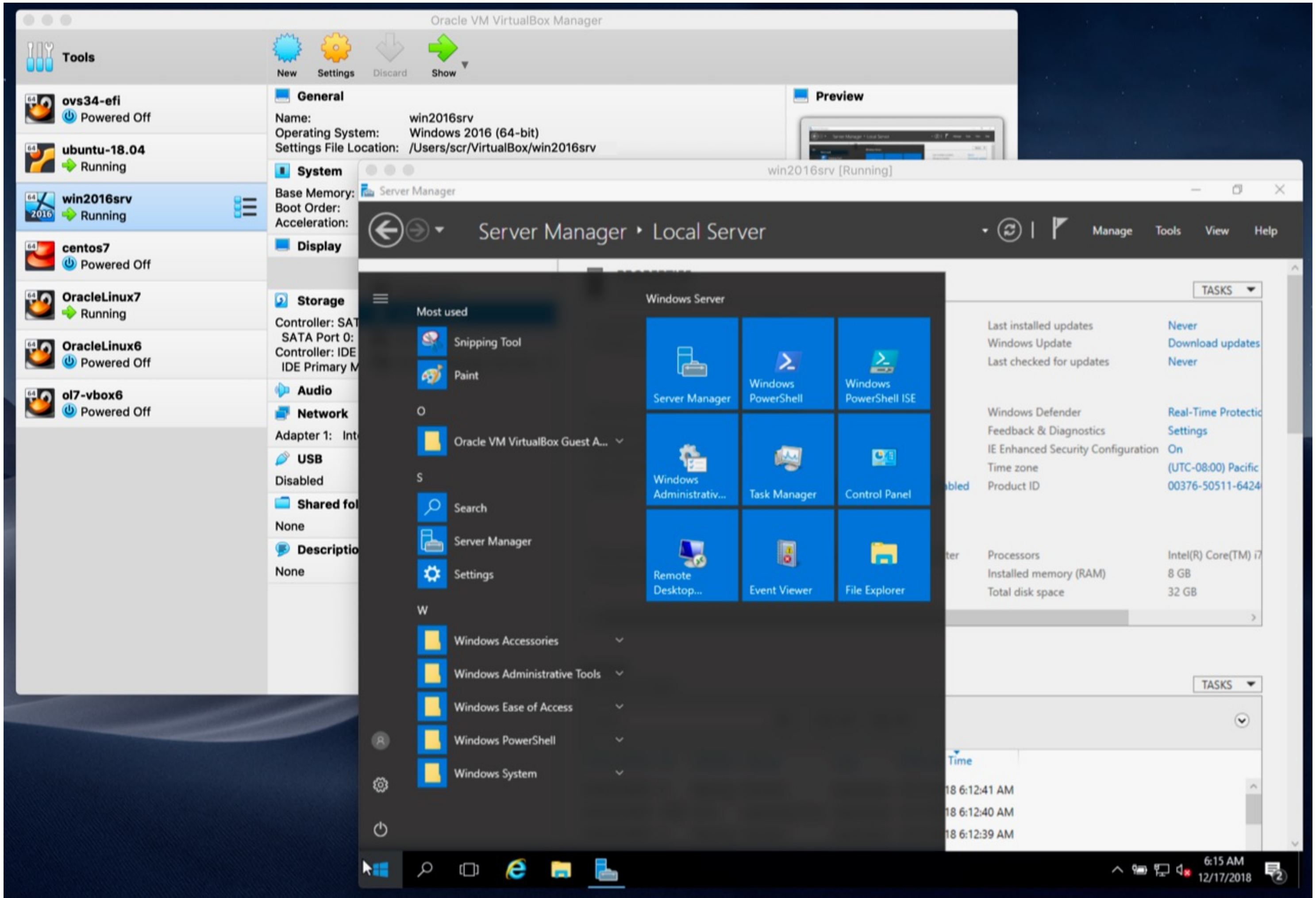


Types of VMs

- **Type 2 Hypervisor**
  - **Type 2** Hypervisor shifts the **VMM** functionalities from *kernel-mode* **OS** to *user-mode* **Processes**.
    - Very little OS involvement in virtualization.
    - **VMM** is simply another process, run and managed by host OS.
      - Even the host doesn't know it is a hypervisor that manages guests.
    - **Disadvantage:** poor performance (can't take advantage of HW features)
    - **Advantages:** **Flexible** (a normal user can install a **Type-2** hypervisor software, e.g., VirtualBox, Parallels Desktop, on any supported OS).



- **Type 2 Hypervisor**
  - A normal user can install a Type-2 hypervisor software, e.g., VirtualBox, Parallels Desktop, on any supported OS.



- **Type 2 Hypervisor**
  - A normal user can install a Type-2 hypervisor software, e.g., VirtualBox, Parallels Desktop, on any supported OS.

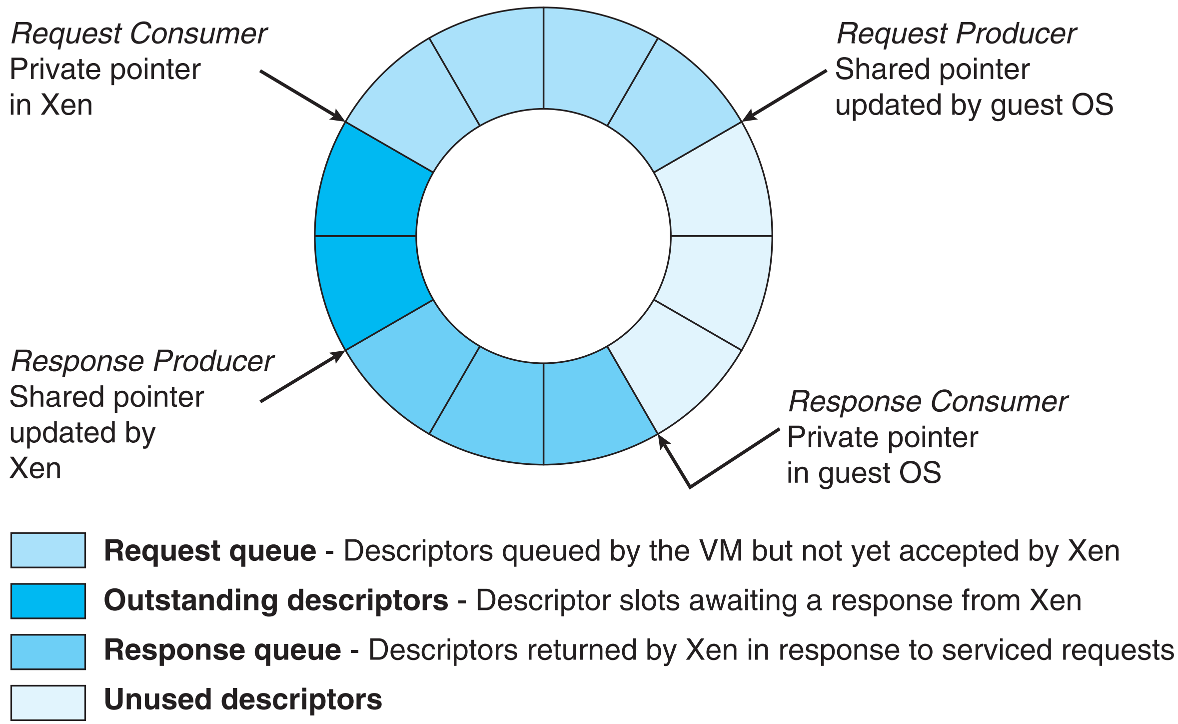
Types of VMs

- **Type 3 Hypervisor**
  - **Type 3** Hypervisor shifts the **VMM** functionalities from *user-mode* **Processes** to ???

- **Type 3 Hypervisor**
  - ~~**Type 3** Hypervisor shifts the **VMM** functionalities from *user-mode* **Processes** to ???~~
  - No, there is no such thing as a **Type-3** Hypervisor, since there is no more higher-level abstractions than **Processes** that can *fully* sustain a Virtual Machine.

- **Paravirtualization**
  - A **hypervisor** uses a technique called "**Full Virtualization**" (**全虚拟化**) when it comes to managing Virtual Machines, which means that the Virtual Machine OSes need not be altered or modified.
    - In other words, in "Full Virtualization", the Guest OS does not know whether it is running on real hardware or a VMM.
  - Apparently, this is not the only way. **Paravirtualization** (**半虚拟化**) works differently by providing the Guest OS with specialized APIs.
    - The Guest **knows** it is not running on hardware.
    - The Guest OS must be **modified** to accommodate the specialized APIs.
    - The specialized APIs would normally enable more **efficient use of resources** and **a smaller virtualization layer**.

- **Paravirtualization**
  - The Xen VMM was the leader in paravirtualization by implementing several techniques to optimize the performance of **guests** and **host**. For example, clean and simple device abstractions:
    - More **Efficient** I/O
    - Better **communication** between guest and VMM about device I/O
    - Each device has circular buffer shared by guest and VMM (*shared memory*).



Types of VMs

- **Paravirtualization**
  - The Xen VMM was the leader in paravirtualization by implementing several techniques to optimize the performance of guests and host. For example, clean and simple device abstractions:
    - **Memory Management** does not include **Nested Page Tables.**
      - Each guest has its own read-only tables
      - Guest uses hypercall (call to hypervisor) when page-table changes needed
  - Paravirtualization allowed virtualization of older x86 CPUs (and others) without binary translation.
  - But on modern CPUs, Xen no longer required guest modification
    - Hence, no longer paravirtualization.

Types of VMs

- **Programming Environment Virtualization**
  - Not really virtualization, but using same techniques, providing similar features.
  - Programming language is designed to run within custom-built virtualized environment
    - For example, Java has many features that depend on running in **Java Virtual Machine** (**JVM**).
  - In this case, virtualization is defined as providing APIs that define a set of features made available to a language and programs written in that language to provide an improved execution environment.
  - JVM compiled to run on many systems.
  - Programs written in Java run in the JVM
    - compiled into Java Bytecode
    - can run on anywhere with JVM (no matter the underlying system).
    - "Write Once, Run Anywhere".

Programming Environment Virtualization

Java Source Code

```
outer:
for (int i = 2; i < 1000; i++) {
    for (int j = 2; j < i; j++) {
        if (i % j == 0)
            continue outer;
    }
    System.out.println (i);
}
```

Java Bytecode

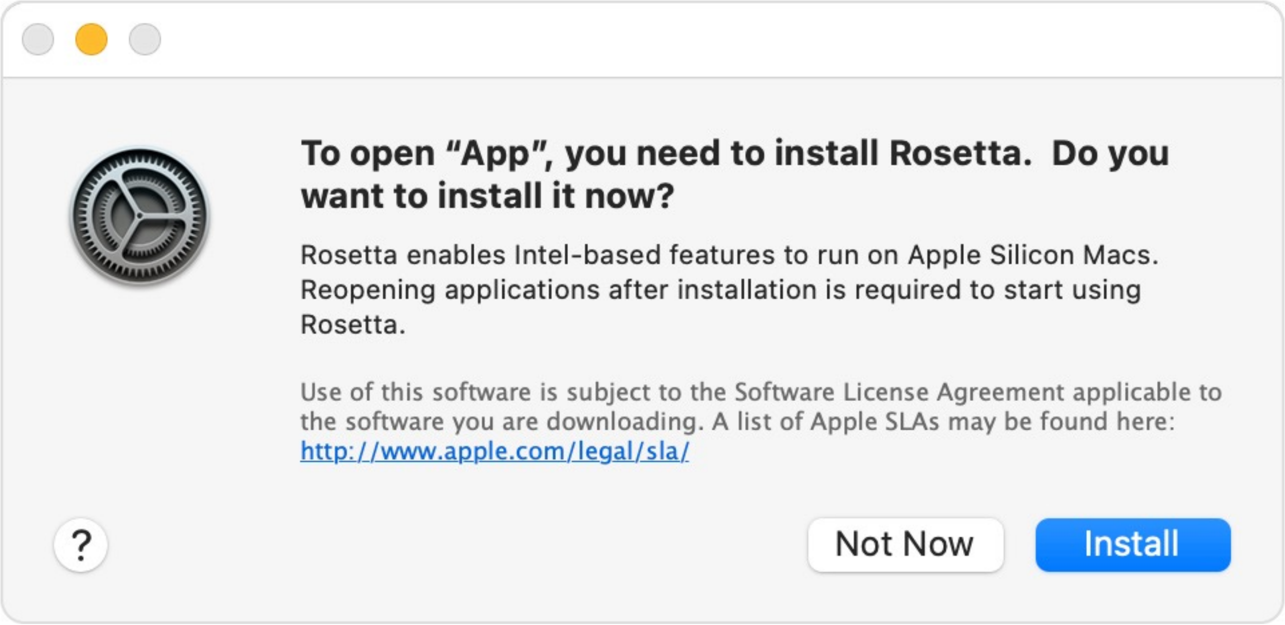
```
0: iconst_2
1: istore_1
2: iload_1
3: sipush 1000
6: if_icmpge 44
9: iconst_2
10: istore_2
11: iload_2
12: iload_1
13: if_icmpge 31
16: iload_1
17: iload_2
18: irem
19: ifne 25
22: goto 38
25: iinc 2, 1
28: goto 11
31: getstatic #84
34: iload_1
35: invokevirtual #85
38: iinc 1, 1
41: goto 2
44: return
```

Types of VMs

- **Emulation**
  - Another (older) way for running one OS on a different OS
    - Virtualization requires underlying CPU (or more precisely, the Instruction Set Architecture, ISA) to be the same as guest was compiled for
    - Emulation allows guest to run on different CPU
  - Need to translate all guest instructions from Guest CPU to native CPU
    - **Emulation**, not **Virtualization**
  - Useful when host system has one architecture, guest compiled for other architecture
    - Company replacing outdated servers with new servers containing different CPU architecture, but still want to run old applications
  - Performance challenge: order of magnitude slower than native code
    - New machines faster than older machines so can reduce slowdown
    - Bochs (pure software emulator) is way slower than QEMU (hardware-assisted emulator)

Emulation

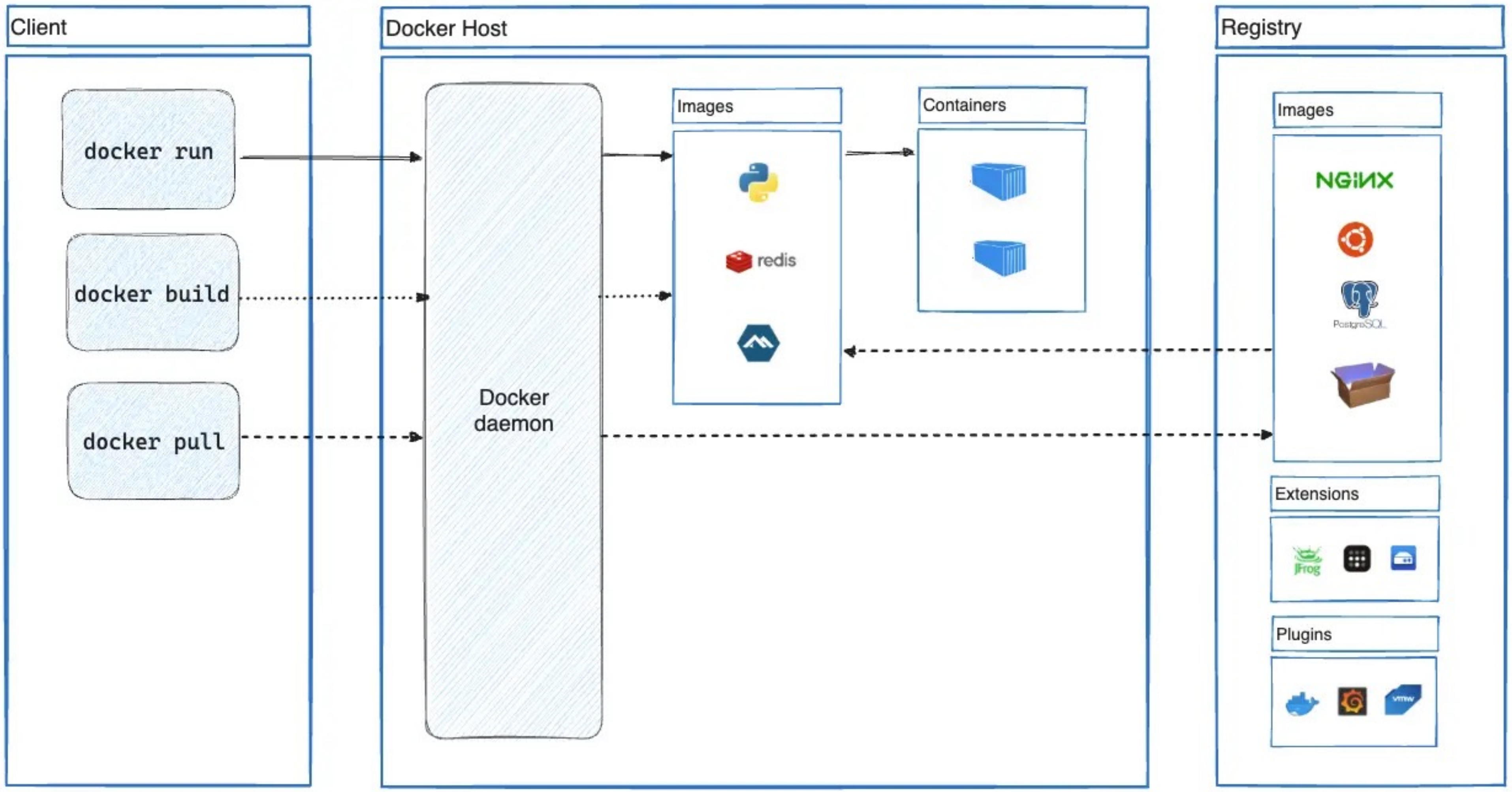
- Not all emulators are poor in performance. For example, Rosetta 2, developed by Apple Inc for the transition from ***Intel processors*** to ***Apple Silicon*** chips, can have even better performance when running x86 programs on M1 macOS than on native Intel CPUs!



Types of VMs

- **Application Containment**
  - Some goals of **virtualization** are segregation of apps, performance and resource management, easy start, stop, move, and management.
  - Can do those things without full-fledged virtualization
    - If applications compiled for the host operating system, don't need full virtualization to meet these goals.
  - Containers create virtual layer between OS and apps.
    - Only one kernel running – host OS
    - OS and devices are virtualized, providing resources within zone with impression that they are only processes on system.
    - Each zone has its own applications; networking stack, addresses and ports; user accounts, etc.
    - CPU and memory resources divided between zones
      - Zone can have its own scheduler to use those resources.

- **Docker**
  - A successful example of Application Containment.
    - much more **lightweight** than traditional Virtual Machines.



Thank you!