263-3855-00: Cloud Computing Architecture

Section 02: Virtualization

Swiss Federal Institute of Technology Zurich (ETH Zürich)

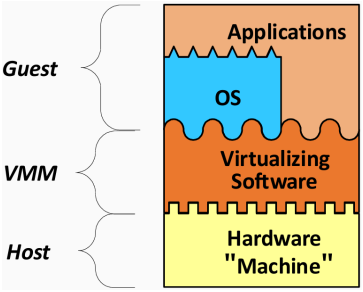
Last Edit Date: 05/31/2025

I. Virtual machines

Systems are built on levels of abstraction. Higher levels hide details at lower levels. For example, files are an abstraction of a disk. Virtualization creates a virtual representation of a resource on physical machines. Provides a level of indirection between abstract and concrete. Does not necessarily hide low-level details.

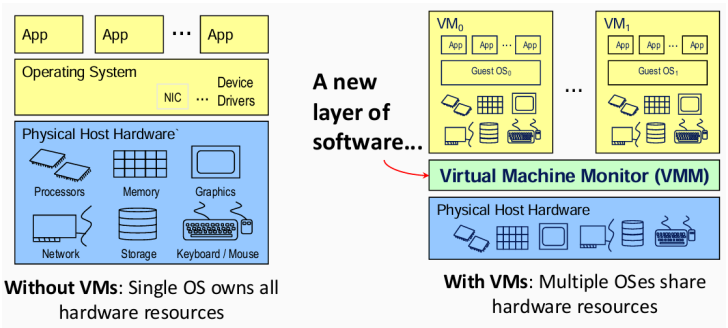
1. VM Terminology

- **Host**: the physical platform (hardware, sometimes also host OS)
- **Guest**: the additional platforms that run on the VM (OS, app, etc.)
- **Virtual Machine Monitor (VMM) or hypervisor**: thin layer of software that supports virtualization



2. System VM

A virtual machine monitor (VMM) honors existing hardware interfaces to create virtual copies of a complete hardware system.



3. Properties of Virtualization

Partitioning: resource sharing and isolation (security)

Encapsulation: checkpoint / restore, migrate, and execution replay

Why use virtualization in the cloud?

- Share hardware efficiently and securely between multiple users
- It enables server consolidation to improve resource utilization, load balancing, and datacenter scaledown

Virtualization implementation

- Three main requirements
    - Safety: isolation between guests, isolation between guests and VMM
    - Equivalency: fidelity of results with and without VMM
    - Efficiency: good performance (minimal overhead)
- Several different approaches

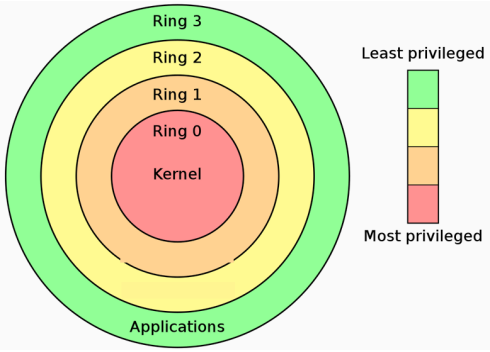
    - Hosted interpretation
    - Direct execution with trap-and-emulate, dynamic binary translation, hardware-assisted virtualization
    - Paravirtualization

Operating systems terminology

- **Trap**: any kind of transfer of control to the operating system
- **System** call: synchronous (i.e., planned), program-to-kernel transfer
- **Exception**: synchronous program-to-kernel transfer caused by exceptional events
- **Interrupt**: asynchronous device-initiated transfer

Hardware-enforced privileged rings

Code in a more privileged ring can read and write memory in a lower privilege ring. Function calls between rings can only happen through hardware-enforced mechanisms.

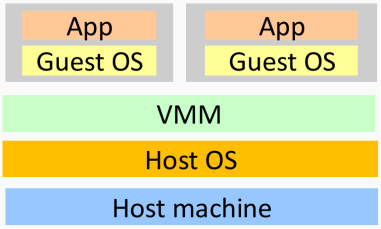


Examples of privileged instructions

- Update memory address mapping
- Flush or invalidate data cache
- Read or write system registers
- Change the voltage and frequency of processor
- Reset a processor
- Perform I/O operations
- Context switch, change form kernel mode to user mode

4. Virtualization approach 1 - hosted interpretation

- Run the VMM as a regular user application on top of a host OS
- VMM maintains a software-level representation of physical hardware
- VMM steps through instructions in the VM code, updating virtual hardware as necessary (e.g., register values)



Advantages:

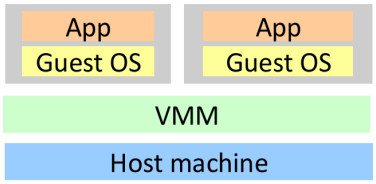
- Complete isolation, no guest instruction is directly executed on host hardware
- Easy to handle privileged instructions

Disadvantages:

- Emulating a modern processor is difficult
- Interpretation is very slow (e.g., 100x slower than direct execution on hardware)

5. Virtualization approach 2 - direct execution with trap-and-emulate

Run VMM directly on host machine hardware. Whenever the guest OS executes a privileged instruction, it results in a trap (i.e., transfer of control to VMM). The VMM uses a policy to handle the trap, e.g., execute the instruction on behalf of the guest OS, kill the VM, etc.

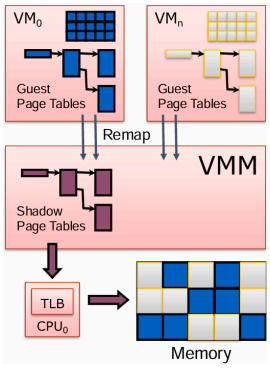


- Due to hardware-enforced ring protections
  - Guest apps cannot tamper with the guest OS
  - Guest apps and guest OS cannot tamper with the VMM
- When the guest OS executes a privileged instruction, it will trap into the VMM
- When a guest app generates a system call, the app will trap into the VMM

Example:

Page-table shadowing

- VMM intercepts paging operations, constructs copy of page tables ("shadow" tables)
- Guest page tables map - guest virtual address to guest physical address
- Shadow page tables map - guest virtual address to host / machine physical address



Overheads

- Trap to VMM adds to execution time

- Shadow page tables consume significantly memory

Advantages:

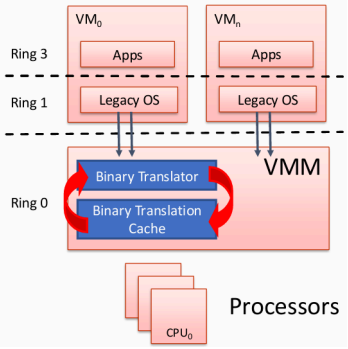
- Faster than approach 1 (hosted interpretation)

Disadvantages:

- Still slow because of emulation
- Does not always work

6. Virtualization approach 3 - direct execution with binary translation

The VMM dynamically rewrites non-virtualizable instructions.



- VMM scans the guest instruction stream and identifies sensitive instructions.
- VMM dynamically rewrites the binary code to add instructions that will trap.

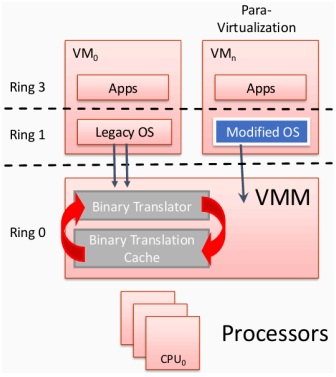
Advantages:

- Can run unmodified guest OSes and apps
- Most instructions run at bare-metal speed

Disadvantages:

- Implementing the VMM is difficult and translation impacts performance

7. Virtualization approach 4 - para-virtualization



- Direct execution with binary translation is tricky, so modify the guest OS to remove sensitive-but-unprivileged instructions.
- Example: Xen hypervisor
    - Guest OS is modified to inform VMM of changes to page table mappings
    - Guest OS modified to install “fast” syscall handlers
- Guest applications are unmodified

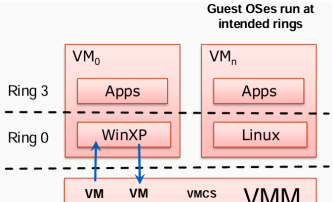
Advantages:

- Faster than direct execution with translation
    - Fewer context switches
    - Less bookkeeping

Disadvantages:

- Requires substantial modifications to the OS, which can be even harder to implement than binary translation logic.

8. Virtualization approach 5 - direct execution with hardware support

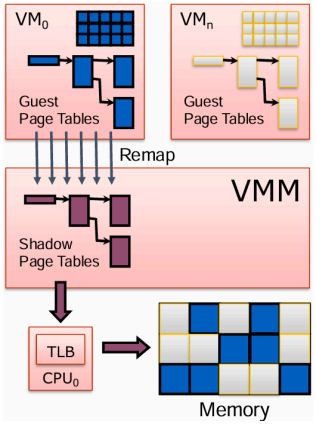


- Direct execution with binary translation is tricky, so add hardware support for virtualization to the CPU, e.g., Intel VT-x, AMD-V.
- Add new privilege mode "VT root mode"
    - Allow direct execution of VM on the processor (vmentry) until a privileged instruction is executed (the vmexit)
- Add Virtual Machine Control Structure (VMCS)
    - Can be configured to control which instructions trigger vmexit, e.g., interrupts, memory faults, IO, etc.
    - Can only be accessed via privileged VMREAD and VMWRITE instructions

Advantages:

- Fast and supported on most CPUs today.
- This approach is most commonly used today.

9. Virtualizing memory

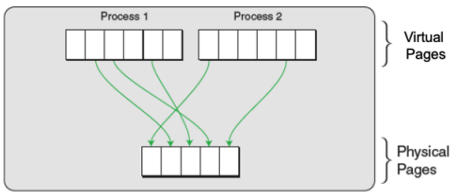


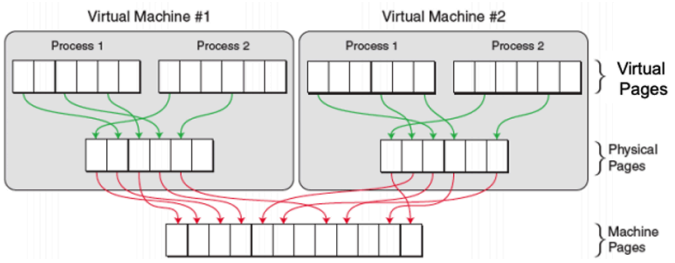
Challenges:

- Address translation
    - Guest OS expects contiguous, zero-based physical memory
    - VMM must provide this illusion
- Page-table shadowning
    - VMM intercepts paging operations
    - Constructs copy of page tables
- Overheads
    - VM exits add to execution time
    - Shadow page tables consume significantly memory

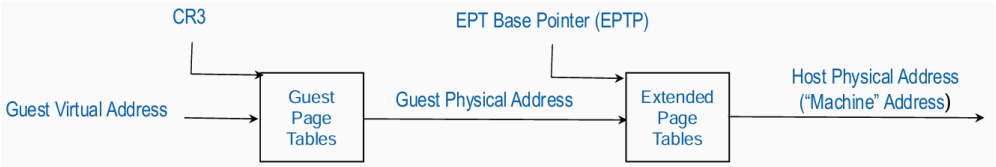
Hardware support for memory virtualization - extended page tables

- Regular page tables
    - Map guest virtual to guest physical





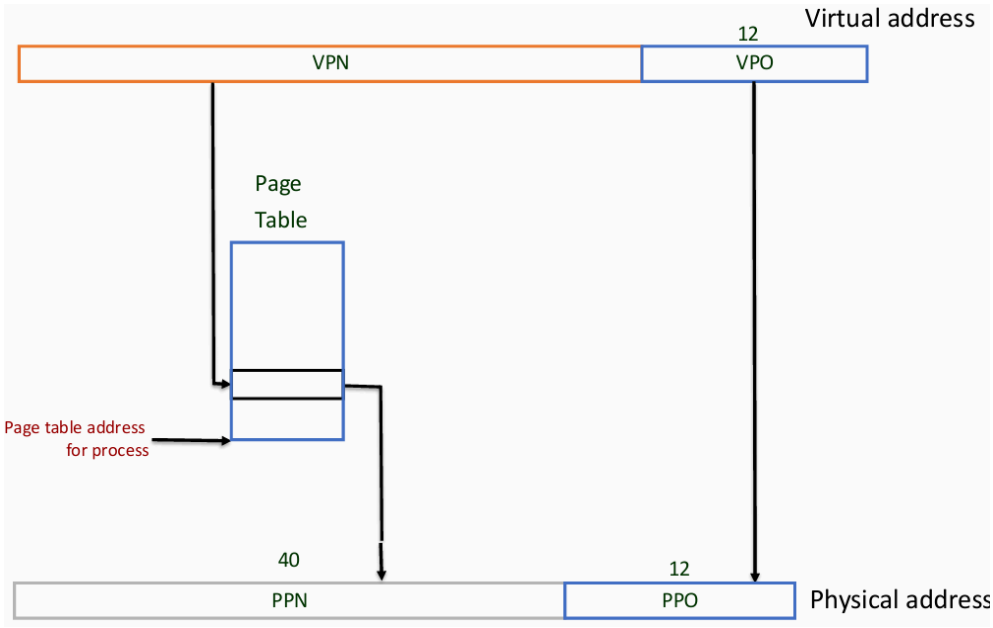
- Extended pages tables (EPT)
    - Map guest physical to host physical (or "machine") address



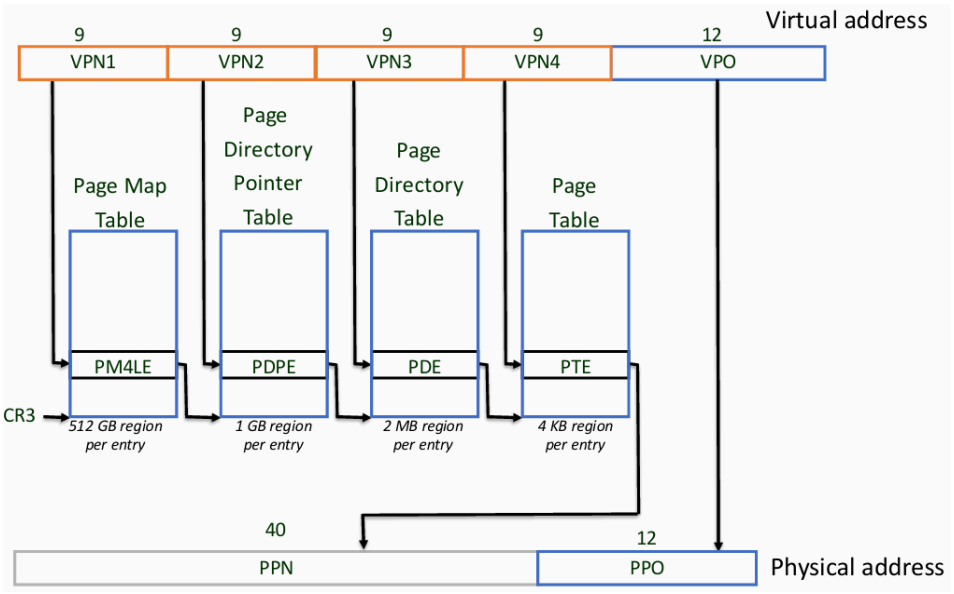
- Performance benefits
    - Guest OS can modify its own page tables freely
    - Avoid VM exit due to page fault
- Memory savings
    - Without EPT, would require a shadow page table for each guest user process
    - A single EPT supports entire VM

10. Address translation

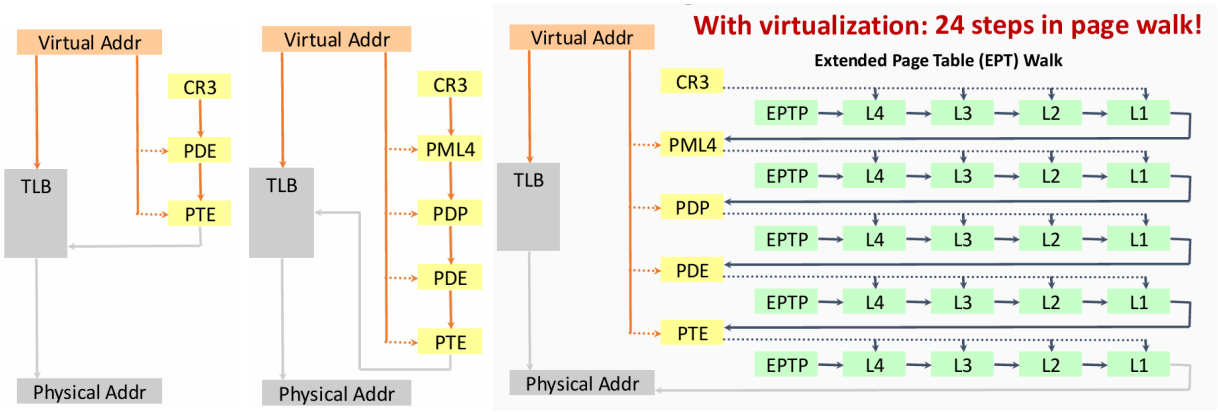
Address translation with page table



x86-64 paging: multi-level translation



Translation lookaside buffer (TLB)



32-bit: 2-level 64-bit 4-level

11. Virtualization IO

Virtual device interface

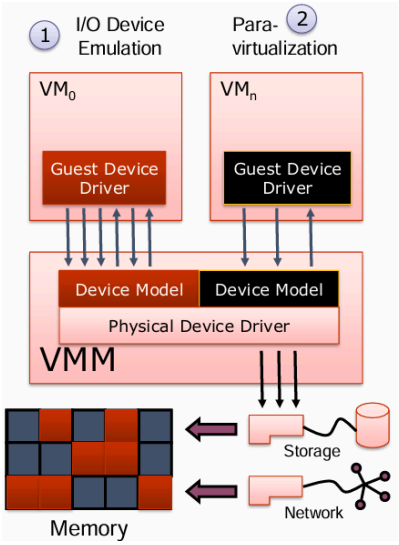
- Traps device commands
- Translate DMA operations
- Injects virtual interrupts

Software methods

- I/O device emulation
- Paravirtualization device interface

Challenges

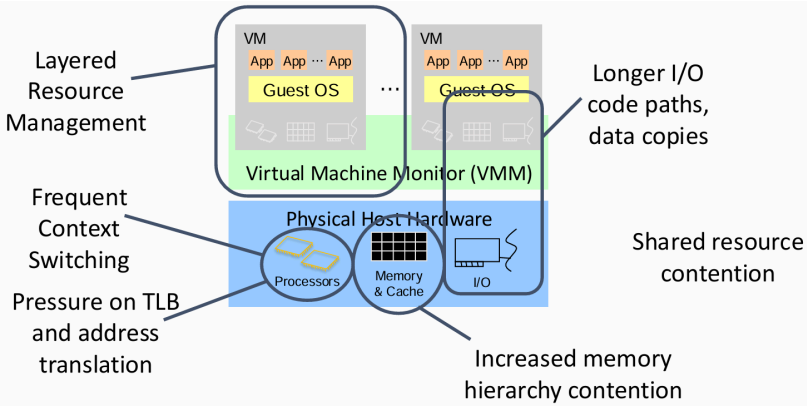
- Overheads of copying I/O buffers
- Controlling DMA and interrupts



Single-Root I/O virtualization

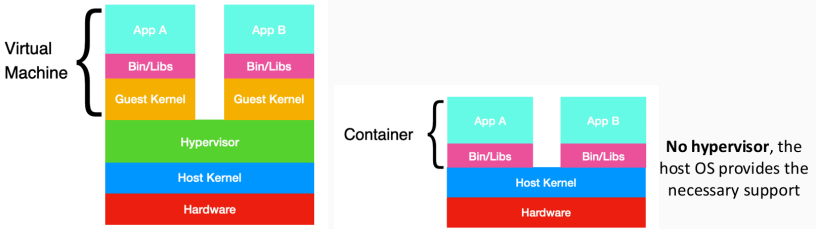
- Extension of PCIe specification
- Allows a PCIe device to appear as multiple separate physical PCIe devices
    - Physical Function (PF): original device
    - Virtual Function (VF): extra “device,” limited functionality, which can be created / destroyed dynamically

Performance implication of virtualization



II. Containers

A container provides lightweight operating system level virtualization, which can achieve higher density sharing of resources on a machine, faster startup and shutdown time, and bare-metal like performance since no VMM traps or binary translation. Containers share the host OS, but have their own system binaries, libraries, and dependencies, having less secure isolation than VMs.



There are two notions of containers:

Docker

- Goal is packaging
- Standard way to package app and its dependencies so can move easily between environments
- Production matches test environment

Linux containers

- Goal is performance isolation
- Not security isolation; uses VMs for that

- Resource isolation with namespaces
- Manage CPU cores, memory, bandwidth limits with cgroups

1. Linux containers (LXC)

Uses three key isolation mechanisms:

- **namespaces**: abstract and limit which global resources a process can see
- **cgroups**: monitor and limit the amount of resources a process can use
- **seccomp-bpf**: limit system calls that a process can call

Linux kernel namespaces

- A namespace abstract a global system resource (e.g., network interface)
- Goal: restrict what a container can see
    - Process-level isolation of global resources
    - Processes have the illusion they are only processes in the system
- Changes to the global resources are visible to other processes in the same namespace, but not to others processes
- A linux system starts with a single default namespace of each type, used for all processes. Processes can create new namespaces and join them.
- **Examples**:
    - MNT: what file systems and mount points are visible?
    - PID: what other processes are visible?
    - NET: which network devices are visible and how are the routing tables configured?
    - Users: what user IDs are visible?
    - IPC: which inter-process communication channels are available?

Linux control groups (cgroups)

- Goal: limit the resources that a container has access to
    - Group processes based on resource limit
    - Enable resource accounting / monitoring for the group
- Implementation
    - cgroups are created, deleted, and modified by altering the structure of a virtual filesystem called cgroupfs
    - Support nested groups; child processes inherit attributes of parent
    - Each cgroup has several subsystems whose limits can be set (e.g., CPU, memory, devices, etc.)

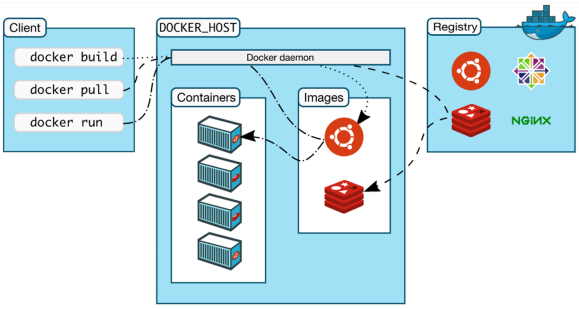
Linux seccomp-bpf

- Goal: limit the system calls the container can call

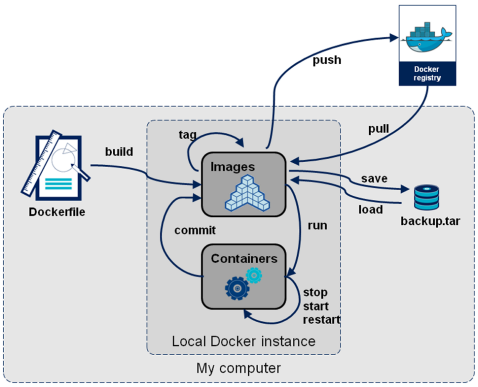
    - Reduce the “attack surface” of the kernel, making it more secure
- seccomp-bpf stands for Secure Computing with Berkeley Packet Filters
- Implementation:
    - Users specify filters for incoming system calls and / or their parameters using the Berkeley Packet Filter interface (originally used to enable user-space programs to specify network packet filters)

2. Docker

Docker builds on top of Linux containers, making features like cgroups and namespace easy for application developers to use. Standard platform for building and sharing containerized applications.



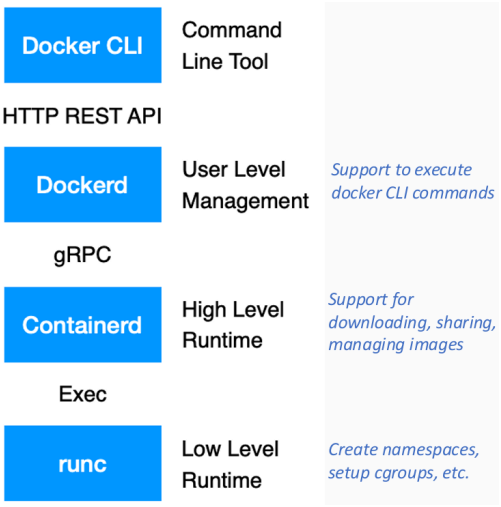
Docker basic concepts



- **Image**: Read-only template for creating a container. An image is often based on another image, e.g., starting with an Ubuntu image and adding a web server on top.
- **Container**: Runnable instance of an image that you can create, stop, move, or delete
- **Registry**: A shared storage service for Docker images
- **Engine**: Open source software that creates and runs containers, consists of daemon process dockerd.
- **Client**: Communicates with dockerd by executing docker client commands, e.g., docker build

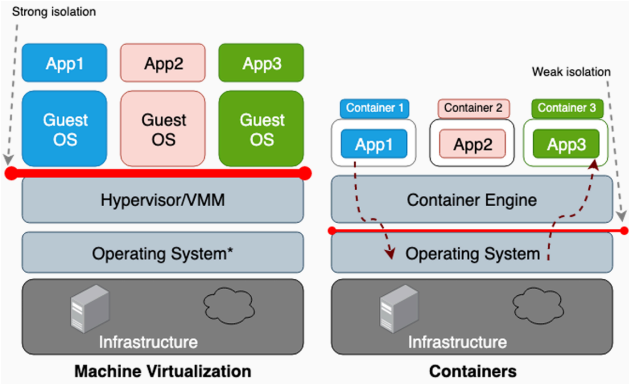
Container Images

- Images are divided into a sequence of layers. Each command creates a new layer on top of the previous layers. Defined by a file called Dockerfile - a script for setting up containers.
- Container runtime represents library responsible for starting and managing container
    - Takes as input the root file system and a configuration file for the container
    - Unshares the namespace
    - Creates cgroup and sets resources limits
    - Execute container command in cgroup



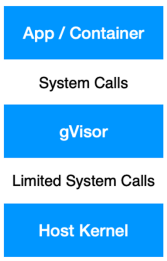
Why / when NOT to use containers?

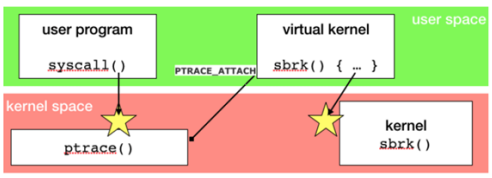
- Containers provide less secure isolation than virtual machines.
- Containers "escape" (i.e., getting root access on the machine) is possible with a single kernel vulnerability.
- The kernel is typically a much larger code base than the VMM / hypervisor, and is hence more likely to have vulnerabilities.



3. gVisor - making container more secure

- A new kind of low level container runtime to create containers and sandboxes it.
    - Intercepts syscalls and performs them in a secure user space kernel instead of the actual kernel.
- Good for running untrusted applications, e.g., serverless.

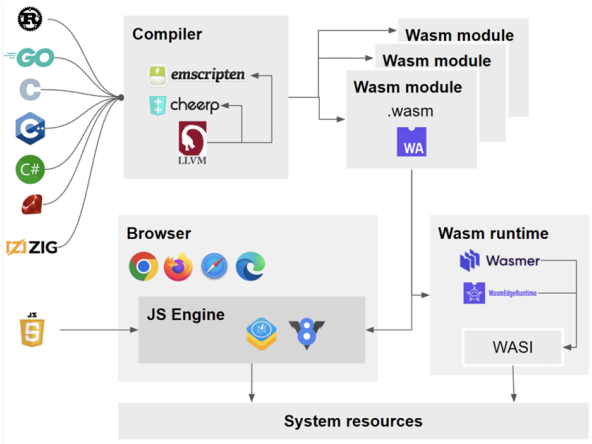




III. WebAssembly

1. WebAssembly Basics

- Different programs:
    - Run with only the assistance of the operating system (compile C, C++, Go, and Rust) into a binary executable format that can directly run on the OS
    - Written using scripting languages, source code is read and executed in rapid succession (JavaScript, Python, Ruby, PHP, and Perl)
    - Compiled to a bytecode format but then require a special program (a runtime or virtual machine) to run them (Java and C#)
- Binary instruction format for secure, fast, and portable execution
- Originally for the web, now used in cloud and edge computing



2. Key features

- Strong security
- Small binary sizes
- Fast loading and running
- Support for many operating systems and architectures
- Interoperability with the cloud service

IV. WebAssembly v.s. VM v.s. Containers

	VM	**Container**	**WebAssembly**
Isolation	Strong	Moderate	Strong
Performance	Slower (emulation	Fast (lightweight,	Near-native (no

	overhead)	shared kernel)	emulation overhead)
Startup time	Slow (seconds to minutes)	Fast (milliseconds to seconds)	Instant (milliseconds)
Footprint	Heavy (GBs)	Moderate (MBs)	Light (KBs)

IV. Firecracker

Firecracker is an open-source VMM developed at Amazon that uses the Linux Kernel-based Virtual Machines (KVM). Firecracker is designed to support lightweight MicroVMs, optimized for security, speed, and efficiency. Amazon uses Firecracker to run Amazon Lambda serverless functions. Securely run functions from different customers on the same machine, without sacrificing performance.

How does Firecracker achieve the best of both containers and VMs?

- Key insight: to support serverless functions, only need a limited set of features in the hypervisor
- For example, do not need support for BIOS, CPU instruction, VM migration or emulation of devices such as USB, PCI, sound, video, etc.

Firecracker uses kernel virtual machine (KVM)

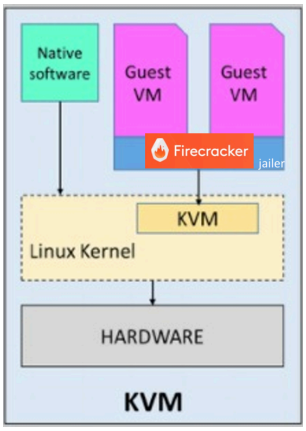
- KVM is a virtualization module that lets you use Linux as a hypervisor. Kernel module and processor-specific module, which can be loaded on Linux x86 machines.
- Provides a full virtualization solution for Linux on hardware with virtualization extensions.

QEMU is a common way to run guest VMs on KVM

- QEMU (Quick EMUlator) is a generic machine emulator and virtualizer.
- System-mode QEMU: emulates a machine's processor (including guest OS and devices) through dynamic binary translation. Often used with KVM.

Firecracker virtualization stack

- Uses KVM, but entirely replaces QEMU with a new VMM device model, implemented in Rust
- One Firecracker process runs per MicroVM
- There is also a jailer wrapper around Firecracker
- With the minimal Linux guest kernel configuration, Firecracker enables:
    - Memory overhead < 5 MB per MicroVM
    - MicroVM boot time < 125 ms
    - Up to 150 MicroVM creations per second per host



V. Related readings

1. (Optional) Operating Systems: Three Easy Pieces
2. (Optional) Bolt: I Know What You Did Last Summer… In the Cloud