Virtualization Technologies

Paravirtualization and Operating System Level Virtualization

Reference:

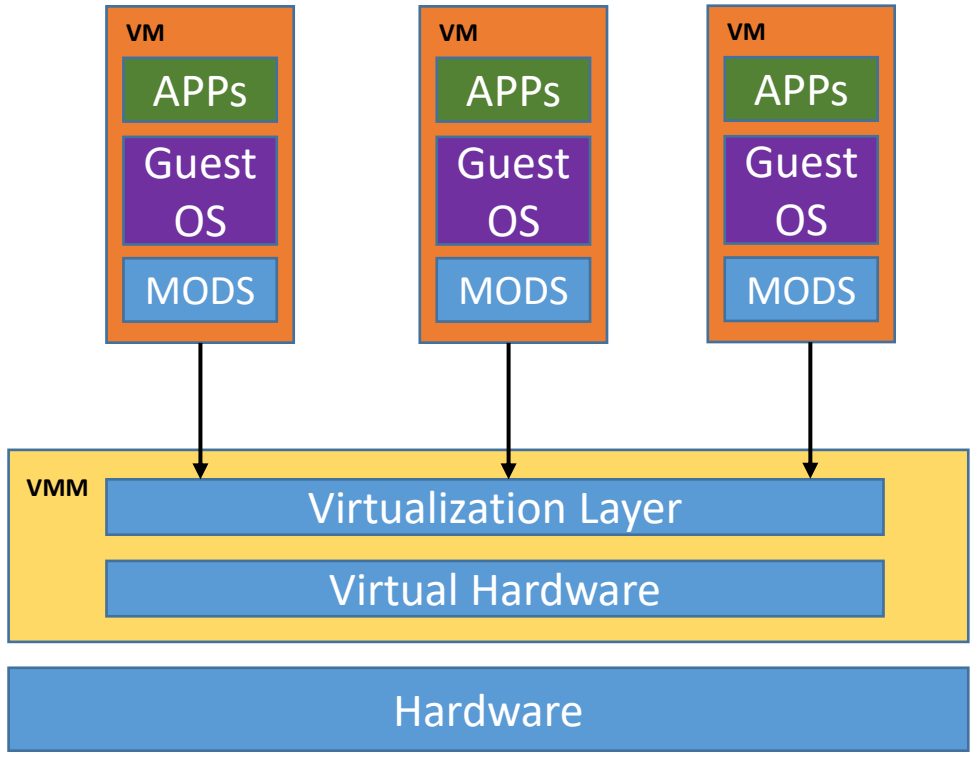
- Material available on the course website

Paravirtualization

- Full virtualization required the virtualization of the whole architecture
- The virtual hardware is exposed to the VM, in which an OS runs **unchanged**
- Guest OS instructions that can not be directly executed, are emulated by the VMM
- Without hardware support, virtualization results in a large number of context switch (between the VM and the VMM)
- In addition, emulation is performed frequently, as many instructions must be emulated as they cannot be executed directly by the guest OS
- As an alternative approach (before hardware support was introduced) **paravirtualization** was introduced
- In paravirtualized system, we still have a VMM, but the hypervisor it is implemented starting from the assumption that the guest OS can be modified and can be made aware of the fact that is running inside a VM
- Modifications to the guest OS are usually limited to the kernel (applications running inside the VM are still unaware)

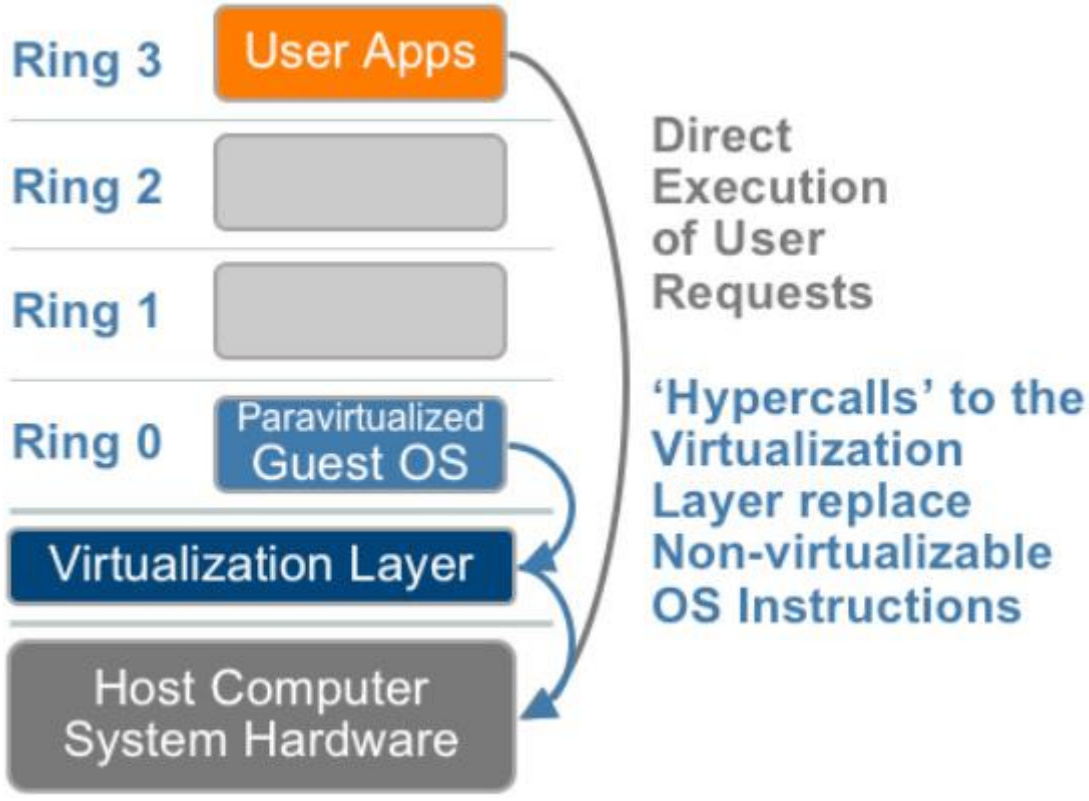
Paravirtualization – Architecture

- The goal is to introduce modifications to the guest OS that prevent the OS to execute instructions that cannot be executed directly (e.g. an I/O instruction) or implement some functionalities in a manner that does not require emulation (e.g. memory management)
- Such instructions or functions are substituted with the call to specific APIs exposed by the VMM through an interface named Virtualization Layer
- The VMM is responsible for implementing the required instructions/functions and create a virtual representation of the hardware and eventually managing the real hardware



Paravirtualization – Implementation

- Emulation is avoided by modifying the guest OS and by invoking '**Hypercalls**', the functions exposed by the VMM that replace non-virtualizable instructions/functions
- Hypercalls code and the VMM are executed in system mode, so they can execute privileged instructions and interact directly with the hardware



Paravirtualization – Pros/Cons

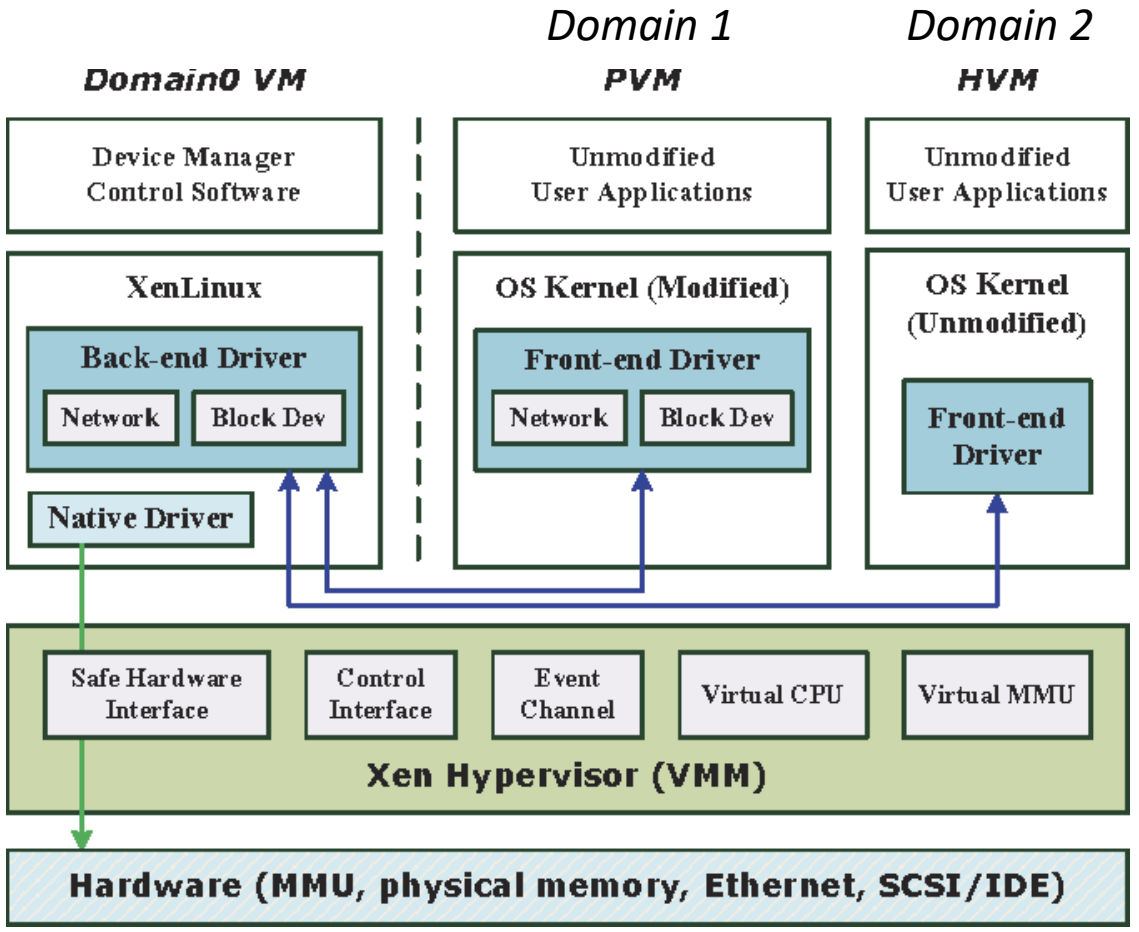
- Pros
  - It improves significantly virtualization performance, as emulation is avoided
  - It does not require specific hardware support
- Cons
  - It supports only modified version of the OS
  - Some closed source OSs cannot be supported
- Paravirtualization systems, e.g. XEN, were very popular at the beginning for the considerable performance increase
- With the advent of hardware support for virtualization they no longer have a significant advantage over full virtualization, consequently they are not popular
- Paravirtualized systems have been converted to full virtualization systems

Xen

- Xen is the most popular hypervisor based on paravirtualization
- Released in 2003, it was build before the introduction of hardware virtualization support, to reduce the overhead of full virtualization, as in normal x86 systems were not easy to virtualize by a standard trap-and-emulate hypervisor
- It tried to find a compromise between rewriting the kernel of the OS and extensive emulation
- Xen is based on a set of modifications of the Linux kernel so you can run a linux OS in an efficient manner in a paravirtualized manner
- The advent of hardware support for virtualization made paravirtualization less convenient, so Xen now support both paravirtualization and full virtualization via hardware support
- Using the latter method any OS (not only the ones modified) can run on Xen

Xen Architecture

- Xen is a very small kernel that isolated first on the machine and gains direct control to the hardware
- On top of Xen, at lower privilege level, we have so-called domains
- There may be as many domains as needed, and inside each domain we may run an entire OS with its own applications
- Domains are isolated from each other, and are used to implement the virtual machines
- One special domain, Dom 0, has access to the Xen API to create and destroy other domains. Inside Dom 0 any Linux OS can be installed, custom tools can be developed to manage the other domains
- Domains can be given direct access to some I/O devices, or they can use fully virtualized devices, or they can use paravirtual devices



Virtualization Overhead

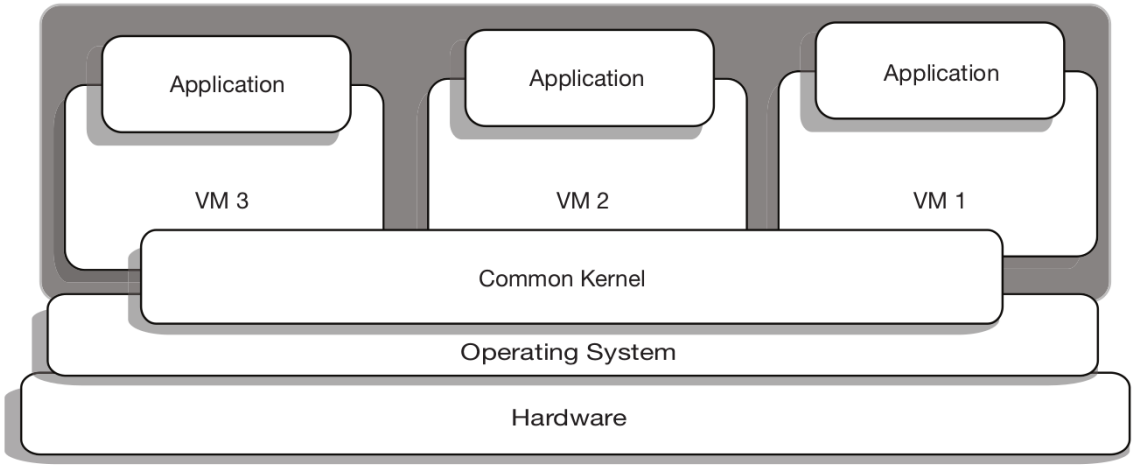
- One use-case for virtualization is the safe sharing of hardware resources among different applications running on the same system
- Application **isolation** should be provided by the OS itself, however, recently OSs become less and less effective in ensuring isolation
- As consequence, the typical configuration to ensure isolation is to run applications on different virtual machines
- *The virtualization overhead is not negligible, both in terms of memory and processing time*:
  - the VMM
  - the guest OS in which the application runs
- It can often happen that different VMs share the same guest OS
- As result, we can have multiple instances of the same OS running in the same hardware

Lightweight Virtualization

- Paravirtualization is still characterized by the overhead introduced by the VMM
- In addition to this, in case multiple instances of the same OS are deployed inside VMs on the same hardware multiple instances of the kernel, OS processes and libraries will run on the machine
- Recently new *lightweight virtualization* technologies have been introduced
- Goal: create a **lightweight execution environment** for services and applications that *do not require the virtualization of an entire system* and still guarantees the same advantages of virtualization:
  - Isolation
  - dynamic instantiation
  - self contained environment

Operating System Virtualization

- A different lightweight virtualization approach has been proposed recently: *Operating System Virtualization or Shared Kernel Approaches*
- Its goal is to have a lightweight approach that minimizes the overhead of running multiple VMs with the same OS on the same system
- With OS virtualization, the hypervisor is removed: *virtual servers are enabled by the kernel of the operating system of the physical machine*
- *The kernel of the OS is shared among all the virtual servers running over it*
- *Since the kernel is shared all the VMs share the same OS, which has to implement logically distinct user space instances*
- FreeBSD Jails or ***Linux Containers*** are Examples



Pros/Cons

Pros

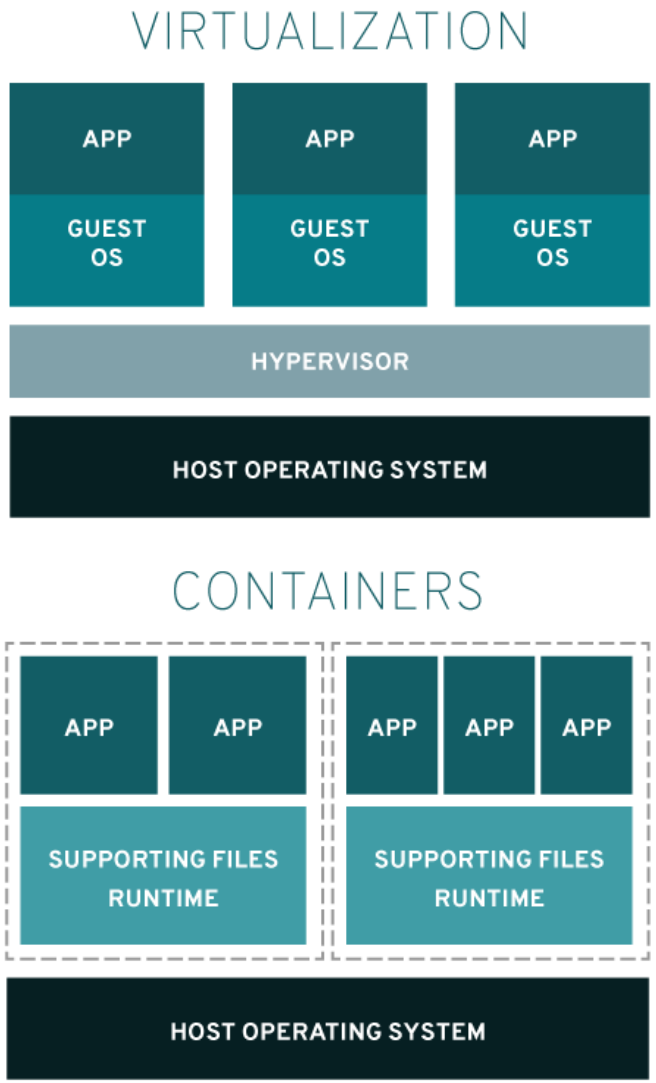
- OS virtualization is lighter in overhead since all the virtual servers share the same instance of the kernel
- A single system with the same amount of resources can support more VMs than full virtualization

Cons

- All VMs must share the same kernel
- Not all the OSs offers OS virtualization solutions

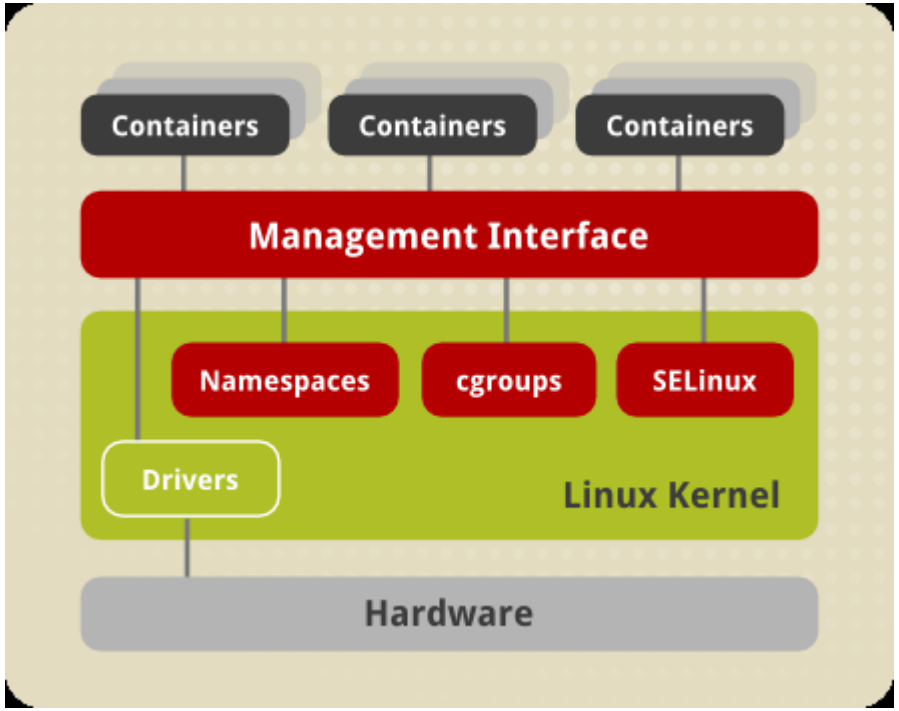
Containers

- *Containers are a way to isolate a set of processes and make them think that they are the only ones running on the machine*
- The machine they see may feature only a subset of the resources actually available on the physical hardware (e.g., less memory, less disk space, less CPUs, less network bandwidth)
- Containers are not virtual machines: *processes running inside a container are normal processes running on the host kernel*. Consequently there is no guest kernel running inside the container
- *You cannot run an arbitrary operating system in a container*
- Since the kernel is shared with the host, it must be the same of the host OS
- The most important advantage with respect to virtual machines is performance: **there is no performance penalty in running an application inside a container compared to running it on the host**



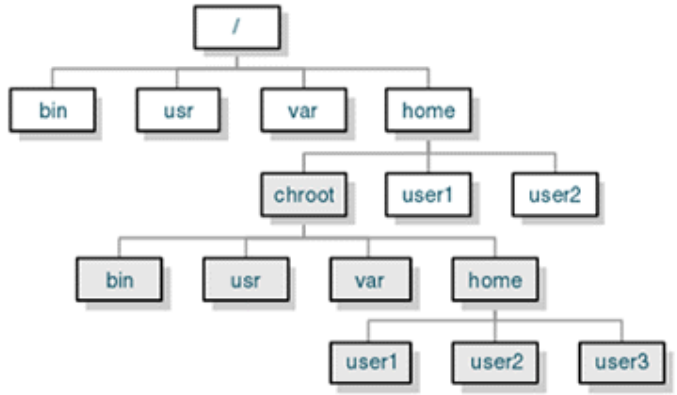
Linux Containers

- **Linux containers** are the most famous and most widely adopted implementation of OS virtualization
- They are implemented using two distinct kernel features: **namespaces** and **control groups**
- Those features ensure that each container represents an isolated running space for applications that can access only the resources (file system, devices, other processes, etc.) they are allowed to



Namespaces

- *Namespaces* provide a means to segregate system resources so that they can be hidden from selected processes
- They are the extension of an old Unix function: *chroot(),* a system call that allows for a process to specify a portion of the file system, to which the application is confined. Using chroot a root user select a subdirectory (a subtree of the file system) that is make as the root file system of the process
- It was defined to confined untrusted process to a portion of the file system so they can access only the files they need
- This function worked for file system, namespaces have been introduced to hide or create other copies of other resources of the system
- Different namespaces are defined, e.g. network namespaces to hide network interfaces or to create virtual interfaces, pid namespaces to hide processes



Control groups

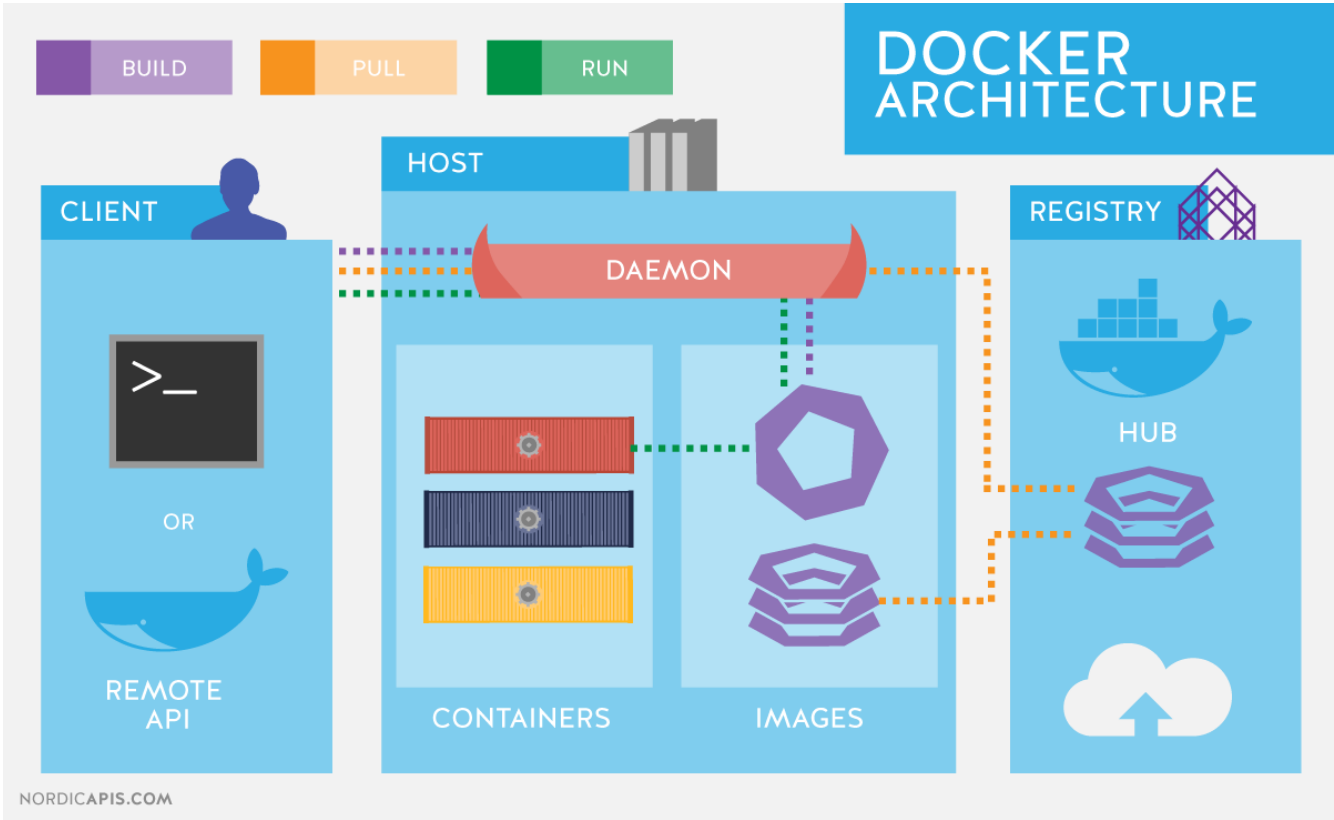
- Namespaces are not sufficient to isolate a set of processes so they can not interfere with others
- Even if processes cannot interact with others, they can abuse of system resources, e.g. by allocating too much memory, by using too much CPU time or network bandwidth
- *Control groups are groups of processes, for which the resource usage is controller and enforced*
- *They are created by root user, each process must belong to a group to which it cannot escape*. When a process creates a child process the child inherits the control group of the parent
- Each group can be linked to one or more subsystems to limit access to system resources, for instance: memory, to limit the amount of RAM accessible to each group; cpu, to limit the maximum fraction of CPU that each group can use

Docker

- In addition of providing lightweight virtualization containers are today exploited for **software distribution**
- An example of that is Docker, a popular system that exploits OS level virtualization to deliver software packages into containers
- Containers are perfect for software delivery as they provide an isolated bundle in which software, libraries and configuration files can be included
- A Docker package after installation is ready to run, as it does not require the installation of additional libraries or dependencies

Docker architecture

- Docker allows to instantiate containers on a system based on an image downloaded from a centralized repository, the *Docker registry*
- A host runs a Docker engine that can download and run a container downloaded from the registry
- Software bundles can be uploaded by developers to the registry, while the local Docker engine provides an interface to users to select and install software packages



Serverless Computing

- Containers enabled a new type of cloud computing model: serverless computing or microservices
- Cloud consumers do not create full VMs to deploy their services (or software) on the cloud, but they create and deploy containers with their software running into an environment with software dependencies and libraries pre-installed and ready to use
- Cloud providers are selling services based on this model, new cloud platforms are created to support it (we will see that in detail)