- Goals and topics

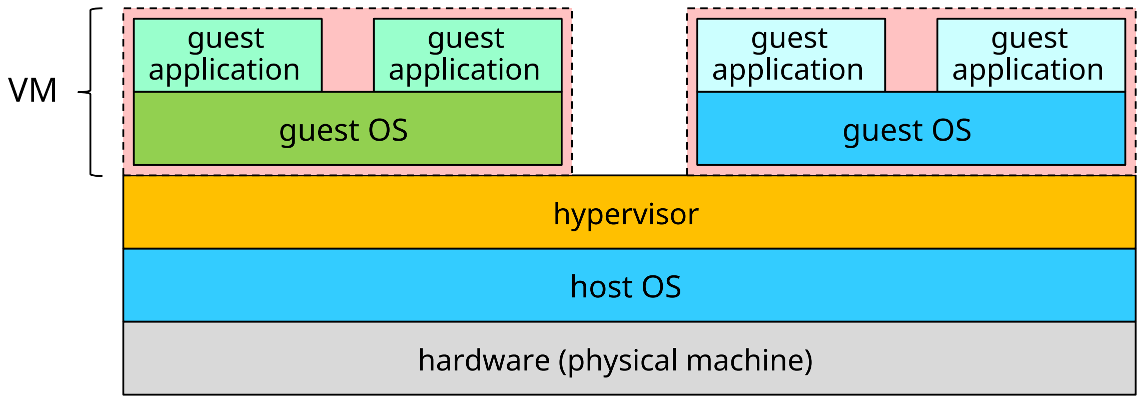
- Goals
  - introduce containers and container-based virtualization
  - compare containers and virtual machines
  - discuss containers as an option for software release
- Subjects
  - introduction
  - recalls of preliminary notions
  - container-based virtualization
  - container
  - techniques for container-based virtualization
  - introduction to Docker containers
  - containers and virtual machines compared
  - container and software release
  - discussion

* Introduction

- Two ways to introduce containers and container-based virtualization
  - as a lightweight variant of VMs and system virtualization
  - in reference to the use that can be made of it
- Let us consider both points of view
  - the most relevant to the software architecture is the use of containers
  - we also do a comparison between VMs and containers, and discuss the use of containers in the context of the software release

* Recalls of preliminary notions

- System virtualization, based on a hypervisor, offers virtual machine abstraction
  - a VM is a virtual computer - what is virtualized is the hardware of a computer
  - in each VM it is then possible to install a complete OS and run applications and services

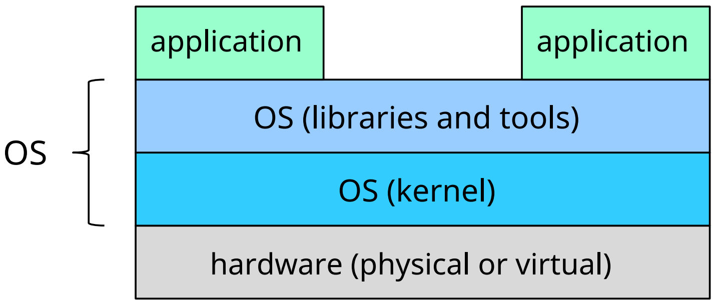


Preliminary notions

- System virtualization
  - offers several benefits
    - has numerous applications, provides operational flexibility, supports isolation between VMs, each with their own services and applications
  - however, it can introduce a high overhead
    - e.g., in I / O management
    - furthermore, a host hosting N virtual machines must take care of the management and execution of N (instances of) operating systems

Preliminary notions

- A *operating system* (*OS*) is made up of several software elements
  - eg. scheduling, memory manag.
  - the *kernel* - which handles some critical responsibilities of the OS
  - a set of *libraries* And *tools* and other utility programs - which operate on top of the kernel es GUI

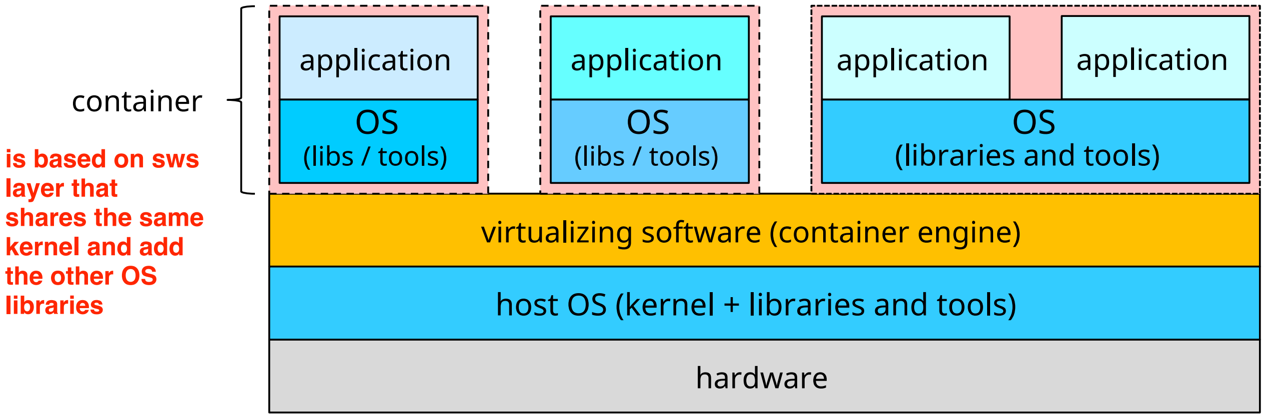


=> idea: create a virtual environment but sharing the host kernel

* Container-based virtualization

- There **virtualization based on container** (*container-based virtualization*)
  - provides the abstraction of **container** ("containers ") - also called *lightweight container*
  - gives an abstraction of:
  - a container is a "virtual entity" that comprises the hardware of a computer together with the kernel of an OS
    - in practice, the kernel of each container corresponds to the kernel of the host OS
  - in each container it is then possible to install an OS (libraries and tools) and run applications and services

Container-based virtualization



Implementation

- Some considerations on container-based virtualization
  - it is widespread especially in the Unix / Linux world
  - is a form of lightweight virtualization, also called *OSlevel virtualization* - because it is supported directly by the host OS kernel
    - it does not require a hypervisor on the host
    - it does not use hardware emulation techniques and does not require hardware virtualization support
  - container virtualization software (*container engine*) allows you to define a container as a collection of processes and other resources
    - e.g. a container running a Java service essentially consists of a JVM process (running on the host)

Implementation

- Some considerations on container-based virtualization
  - Only one shared kernel runs on the host system
    - this kernel manages both the resources of the host system and those of the running containers
  - the kernel is shared by the containers
    - but each container runs its own OS and has its own "virtual" resources - e.g., its own complete file system and its own network stack, with its own IP address

can i have containers on multi OS environment? NO eg. i have a hw and an Windows OS => i can not make a container linux based (because Windows kernel is not compatible)

But i can make a linux container on a MAC OS machine

Can i make a container that has his own GUI? NO cause the GUI has access on physical layers => containers don't virtualize the hw (VM's do)

Implementation

- Some considerations on container-based virtualization
  - the shared kernel manages all processes and all other resources (eg files), both of the host and of the containers
    - processes of a container are managed as processes of the host

can i have a multiprocess program in a container? NO, only in the host
the solution is to program a multithread program or create a VM and than multiprocess

    - a container's file system is managed as a subtree of the host's file system
      - eg, starting with
        / var / lib / docker / containers / *containerid* / filesystem /
    - accessing a file in a container is realized as accessing a file in the host's file system

Implementation

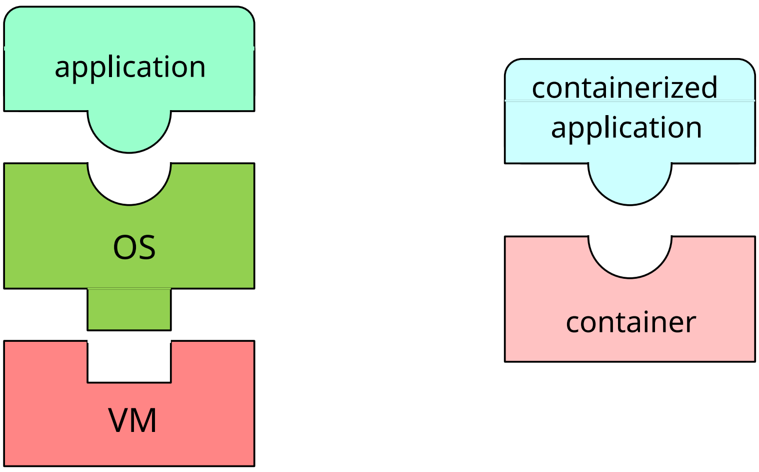
oss: you keep the advantages of virtualizations

- Some considerations on container-based virtualization
  - container virtualization software ensures isolation between different containers - and between containers and the host
    - e.g., a container cannot interfere with another container's processes or access its file system - nor can it directly access the host's resources
    - however, the containers may not be completely isolated from the host

-flexibility on os versions and libraries

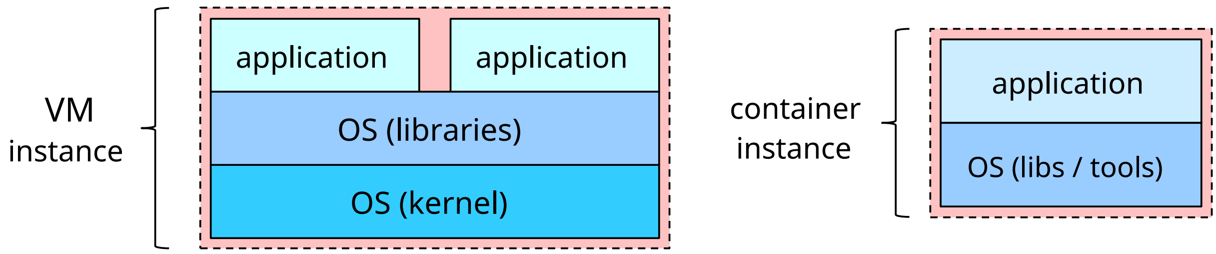
Container and interface

- In terms of interface, comparing VMs and containers
  - the interface exposed by a VM is that of the hardware of a computer
  - the interface exposed by a container is that of an OS's kernel - the system call interface of the kernel



Containers and instances

- A' *VM instance* it also includes the OS and the applications that are installed on it
  - similarly, a '*container instance* it also includes the libraries and tools of the OS and the applications that are installed on it



  - the term container is also often used to mean a container instance

Containers and images

- A' *VM image* includes the contents of a VM's disks - to facilitate the creation of one or more VM instances from that image
  - similarly, a '*container image* consists of the file system image of a container
    - includes one or more applications to run in the container - along with the libraries and tools of an OS and all the software needed to run those applications
    - so that you can easily create one or more container instances from that image

Discussion

- A preliminary comparison between container and VM
  - containers introduce less overhead than VMs (they are "lighter")
    - performance is almost native - you don't need to use processor virtualization, I / O virtualization, or a hypervisor
  - However, containers offer less operational flexibility and isolation than VMs
    - a container's OS must be compatible with the kernel running on the host

* Container

- Let's now consider containers from the point of view of their use

good practice: a container for each application

- Each container (container instance) is typically used to run a specific software service or application (although sometimes more than one / one)
  - the container is therefore used to create the virtual execution environment required by that specific software service
    - the container "contains" everything needed to run this software service (which is "contained")

Container

from the developer side:

- A **container** is a standardized software unit, which packages one or more software applications, along with their configurations and dependencies - such that these applications can run quickly and reliably in a suitable container execution environment

Container

- A container is a standardized unit of software, which packages one or more software applications, along with their configurations and dependencies
  - the container has the purpose of providing the application software of interest with a complete and autonomous execution environment, with all the necessary dependencies - without any dependence on the host
    - these dependencies include the OS libraries and tools, the runtime libraries required by the languages used, and the middleware
    - the specific dependencies of the application software of interest are installed and configured in the container (instead of the host)

Container

- A container is a standardized unit of software, which packages one or more software applications, along with their configurations and dependencies
  - the container is also a standardized execution environment for its application (or applications)
    - each application (in its own container) can thus be released and run consistently across a variety of platforms, both on the cloud and on premises
    - There are several standard container formats - the most popular today is Docker

Types of containers

- A classification of containers, in relation to their use
  - a *OS container* is a container designed to be used as a lightweight VM - with its own OS, in which to run multiple applications or services
  - a *application container* is a container designed to hold and run a single application or service
  - we mainly focus on application containers

Application container

- Application containers allow you to focus each container on a single application or service
  - the container must contain only the dependencies for that specific software service
    - this reduces the risk of inconsistencies in the software stack
  - individual containers are lighter - they can be created and started faster, and at runtime they only use the resources required for their specific service
    - this supports availability, scalability and modifiability
  - the containers are isolated from each other
    - this supports reliability and safety
  - from the point of view of an application or service running in a container, it is as if the application or service were running on its own node - with its own IP address and its own file system

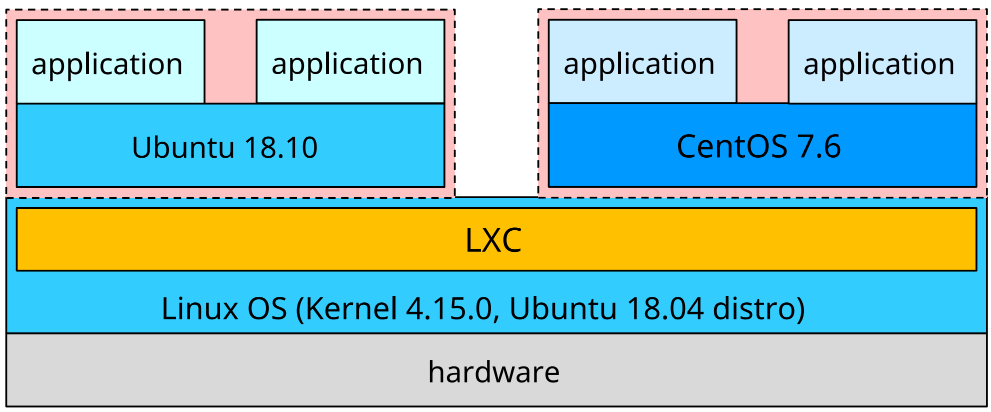
* Techniques for container-based virtualization

- There are several technologies for containers, in the context of UNIX and Linux operating systems
  - e.g., LXC, OpenVZ for Linux, Solaris Containers for Solaris, FreeBSD jail for FreeBSD and Docker
  - we describe some container virtualization techniques

- Linux container support

- Some insights into containers *LXC* (*Linux Containers*, 2008) - the first full container implementation for Linux
  - in the Linux kernel, each process can spawn other processes, in a hierarchical fashion
  - a container is a subtree of the system's process tree - associated with resources (such as CPU, memory and disk) and which is kept isolated from other containers (with their resources)
  - a container's kernel is the host's - but a container's OS can be different from the host's OS

LXC container



LXC container

- Practically, *LXC* (*Linux Containers*) is a set of simple tools - but based on a powerful API - to create and manage containers on a Linux host (real or virtual)
  - LXC allows you to create and run one or more containers - with their applications
    - the containers are isolated from each other and from the host OS, and behave as independent machines
  - containers are similar in functionality to VMs
    - but they are controlled directly by the host OS kernel, without the need for a hypervisor

LXC container

- LXC combines some features of the Linux kernel to provide container abstraction - in particular
  - control groups - to control the use of resources
  - namespaces - to control the visibility of resources

Control group

- A *control group* (*cgroup*) And
  - a group of processes - rooted in a certain process and includes all its current and future children (and descendants)
  - associated with parameters and / or limits in the use of resources (such as CPU, memory, network, file system, ...)
  - cgroups allow you to isolate, limit and measure the use of resources assigned to a group of processes

Namespace

- A *namespace* represents a self-contained collection of resources that are given virtual names, which are then mapped onto real resources
  - resource examples are process and user ids, network resources, host name and its ports, files in the file system
  - namespaces allow you to decouple a group of processes from the real resources that will be assigned to it - to control the visibility of resources and to avoid name conflicts and inconsistencies in references
  - in practice, namespaces allow you to separate the resources of different process groups

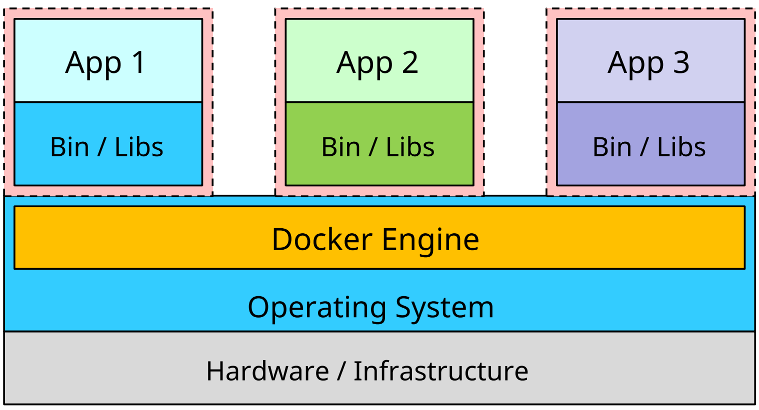
Discussion

- An LXC container offers an execution environment similar to a standard Linux distribution, with some level of control and resource isolation
  - they are typically used as "OS containers" - to run multiple applications or services as well
- The use of LXC takes place through container management tools
  - which are based on a powerful API - but which is not easy to use

* Introduction to Docker containers

- *Docker* is a container platform
  - to build, release and run distributed applications - simply, quickly, scalable and portable

each app has its BIn/ligs OS libraries



Docker container

- A *Docker container* is a standardized unit of software, which packages a software service, along with its configurations and dependencies
  - a container contains everything needed to run that software service - executable code, configurations, libraries, and system tools
  - Docker containers are lightweight, standardized and open and secure

Functionality and use

- Here are the main features offered by the Docker platform
  - create a container (a container instance) from a container image
  - start, monitor, inspect, stop and destroy containers
  - create and manage container images
  - manage related groups of containers - in which to run multi-container distributed applications

Discussion

- Docker (since 2013) has been an instant hit and is used in production by many companies - few technologies have seen a similar adoption rate
  - the main benefits of Docker are lightness, efficiency, simplicity, provisioning speed, openness, possibility of release on a variety of platforms
  - one of the main features of Docker is portability

create a container for each application -> microservices

  - Docker containers are optimized for the release of individual applications or services - they are "application containers"
  - the ecosystem of tools for Docker is very interesting
    - in particular, it supports the composition and orchestration of containers (discussed in subsequent lecture notes)
  - see also the handout on Docker

* Containers and virtual machines compared

- Containers and virtual machines have characteristics that are complementary to each other
  - VMs are very flexible
    - each VM has its own complete OS and its own applications
  - containers offer less flexibility
    - a container's OS must be compatible with the host's OS (which is usually Unix or Linux)
  - isolation between VMs is complete
  - the isolation offered by containers is not complete
  - however, the flexibility and isolation provided by system virtualization are not always required
    - containers offer execution environments that are adequate for many applications

Containers and virtual machines in comparison

- Containers and virtual machines have characteristics that are complementary to each other
  - Furthermore, the flexibility offered by VMs comes at a cost - one VM
    - requires more resources on the host system
    - it can introduce greater execution overhead
    - it takes longer to start up
  - containers are instead "lighter" than VMs
    - the performance is almost native
    - they require fewer resources
      - eg, a minimal Linux installation takes approx 1.1MB - Ubuntu Server libraries require around 180MB
    - a higher density of containers per host is possible
    - containers can be created and started faster

* Container and software release

- Containers are another release option for distributed software systems
  - each container (“application container”) encapsulates a software service, along with the software stack needed for that service

Benefits

- Benefits of using containers for software release
  - each container encapsulates a single software service - releasing an instance of that service can be handled as easily and reliably as creating a container
  - fault isolation and security - each container (with its service) runs quite isolated
  - Containers are lightweight - you can allocate resources to fine-grained containers (and related services)
  - creating and starting a container typically takes from a split second to seconds - less than a VM
  - containers can be released both in the cloud and on premises, in their own private data center
  - specifically, containers can be released into a container orchestration platform (e.g., Kubernetes)

Drawbacks

- Drawbacks to using containers for software release
  - isolation between containers is not complete
  - overhead in administering and updating container images
  - overhead in administering the container execution infrastructure - unless the containers are running in a cloud-hosted solution (such as AWS ECS or Google GCE)

* Discussion

- Both containers and virtual machines have their own advantages and drawbacks
  - VMs offer greater isolation and generality - but at the price of greater overhead
  - containers offer better performance and better resource utilization - but with less isolation and flexibility
  - therefore, containers and VMs have complementary characteristics
    - each technology offers advantages that may be useful in specific situations

Discussion

- It's limiting to think of containers as just a light form of virtualization
  - Containers are significantly changing the way distributed software systems are released and run - and how they are designed and developed
    - the adoption of containers requires a change in software architecture
  - Container adoption is so rapid that regular use of containers in many software systems is expected within a few years
    - "from Gmail to YouTube via Search, all Google products and services run in containers ... every week we run over several billion containers "
    - "80% of all containers in the cloud run on AWS "