David Bednárek
Computing in virtual environments

virtual

- virtual
  - Merriam-Webster dictionary
    - very close to being something without actually being it
    - existing or occurring on computers or on the Internet
    - from Latin *virtus* - strength, virtue
      - from *vir* - man

Virtual – examples outside computing (2009)

Virtual elements in computing

Virtual elements in computing infrastructure

- Virtual memory
  - 1962; in daily use since 1970s (IBM S/370 and many others)
  - Always implemented in hardware, controlled by OS
- Virtual machines
  - 1972 (IBM S/370), abandoned before 1990
  - Revived in 1999 (VMWare at Intel/AMD x86)
  - Originally implemented purely in software
    - But co-developed with hardware in IBM S/370
    - Specific hardware support in Intel/AMD CPUs since 2005
- Virtual disks
  - 1974 (Unix)
  - Originally implemented as block-device drivers (RAM-disks etc.)
  - High-performance versions implemented in dedicated HW (RAID controllers)
- Virtual NICs, VLANs, VPNs, ...

- Virtual execution environment
  - An environment in which a piece of software runs
  - Different from the native environment for which the software was designed
    - Even if the software developers know that they are developing for a virtual environment, they want to ignore the complexity of the target environment, pretending that they develop for the plain old physical world
  - Built upon some or all of the previously existing virtual technologies:
    - Virtual memory (always)
    - Virtual machines (sometimes; always in clouds) and/or containers
    - Virtual disks or virtual file systems
    - Virtual NICs (always)
    - VLANs, VPNs (in large installations and clouds)

Motivation for virtualization

Multi-tenant environments

- Tenant – a person/corporation using a set of services
  - Different from the owner of the hardware
    - A completely different (legal) person (a customer), or
    - An organizational unit using services supplied by an IT department, etc.
- Multi-tenant environments
  - Hardware resources shared among multiple tenants
  - Tenants are not able to share resources voluntarily
    - They usually do not know each other
    - They don't want to negotiate on resources
    - Their software cannot be sufficiently customized to share resources
- Granularity of multi-tenant sharing
  - A physical computer is often too big
    - Load balancing may require fragments of the power of a physical computer
  - It is too difficult to reassign a physical computer to a different tenant
    - Even if automated, such a reassignment may take hours

Dependency hell

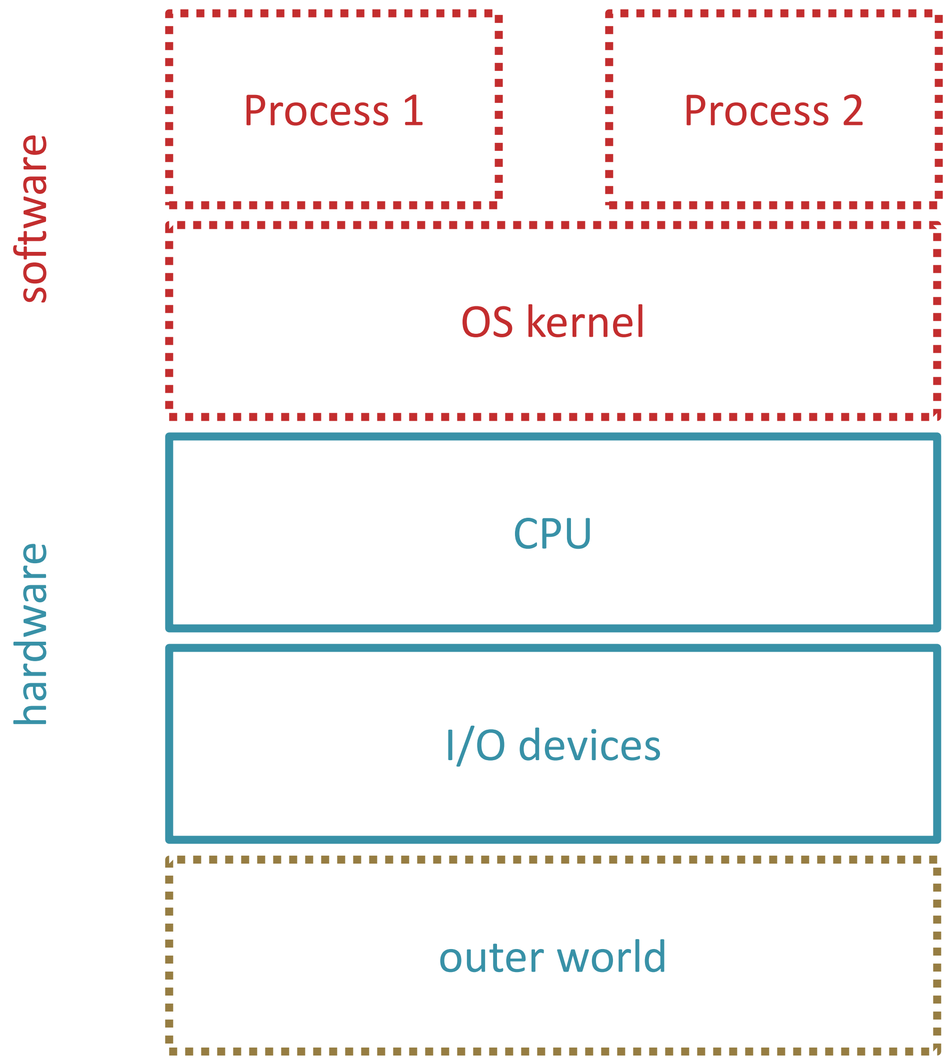
- A piece of software is not a single file or folder
  - Executables are linked to dynamically-loaded libraries
    - Referenced by a short name like “libcrt.so”
  - An application is often divided into communicating processes
    - Often because some parts of code cannot coexist inside the same executable
    - Linked by named pipes or IP sockets, identified by file names, port numbers
  - There are resources, configurations, data, multimedia, ...
    - Stored as files somewhere, identified by relative/absolute file names
    - Different systems have conflicting conventions
  - All the constituents must have the same or compatible version
- Coexistence of two versions of the same software
  - Needed if software A and B require different versions of software C
  - A and B shall be configured so that they find different versions of C under the same name
    - Preparing such configurations is difficult
    - Such configurations would deviate from system conventions (like /etc/*)
    - Complex configurations may degrade performance (copying of large environments)
    - There is often no configuration option at all

Motivation for virtualization

- Problems
  - Multi-tenancy
    - Different tenants cannot share the same machine
  - Dependency hell
    - Often, different software of the same tenant cannot share the same machine
- At the same time, load-balancing requires sharing the same machine between different tenants and/or software
- Solution: Virtualization
  - Disconnect the notion of machine from the physical hardware
    - A hardware machine may host multiple virtual machines
    - Virtual machines may migrate across hardware machines
    - Virtual machines may be easily stopped, created, destroyed, ...

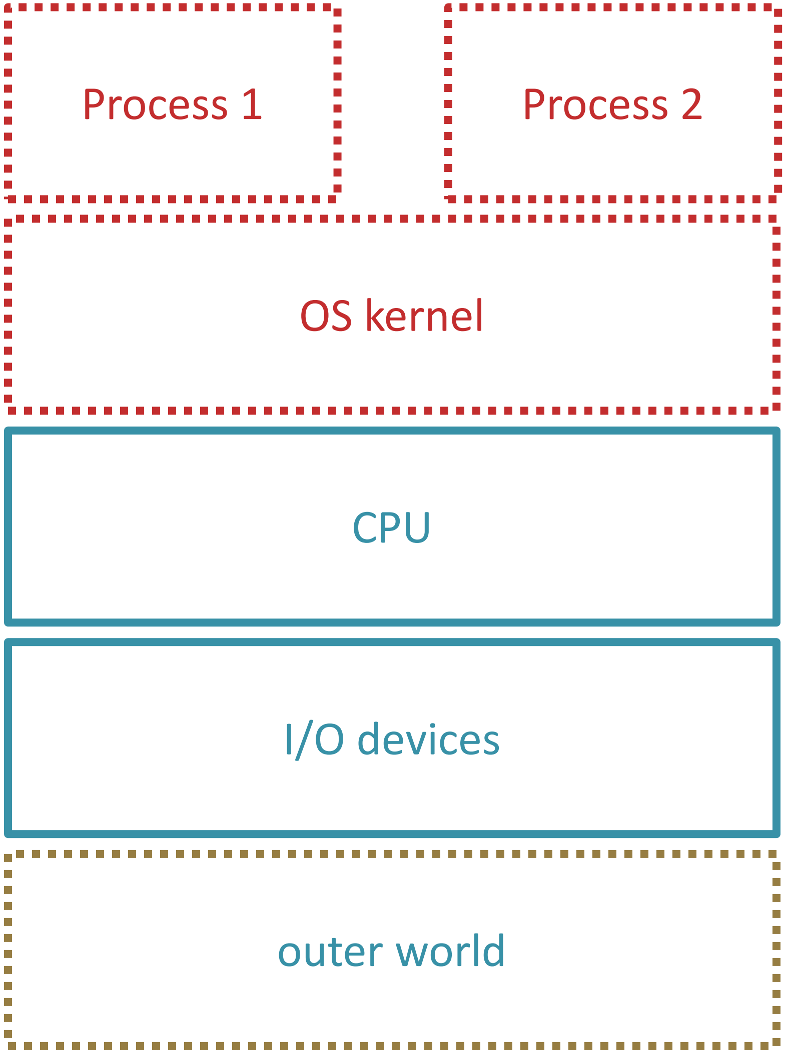
- In the plain non-virtualized world, people think about machines (physical computers)
  - "I want to log into computer X"
  - "I want to install software Y at computer X"
  - The naming, addressing, configuration is mostly machine-centric
    - machine:port addressing in TCP/UDP
    - /usr/bin or "c:\Program Files" installations of software
    - /etc/* or HKEY_LOCAL_MACHINE registry configurations of software
    - machine-wide scope of "ps", /proc/*, ...
  - This could have been done differently, but it was not
    - Nobody is going to modify all the software built in the machine-centric era
    - The people will not change either
- Result: we want to virtualize machines
  - Creating an illusion of a complete computer

Plain Old Execution Environment



- Naïve picture
- In reality
  - Processes directly interact with CPU and memory
  - I/O devices may directly interact with memory
  - There may be more than one CPU in the system

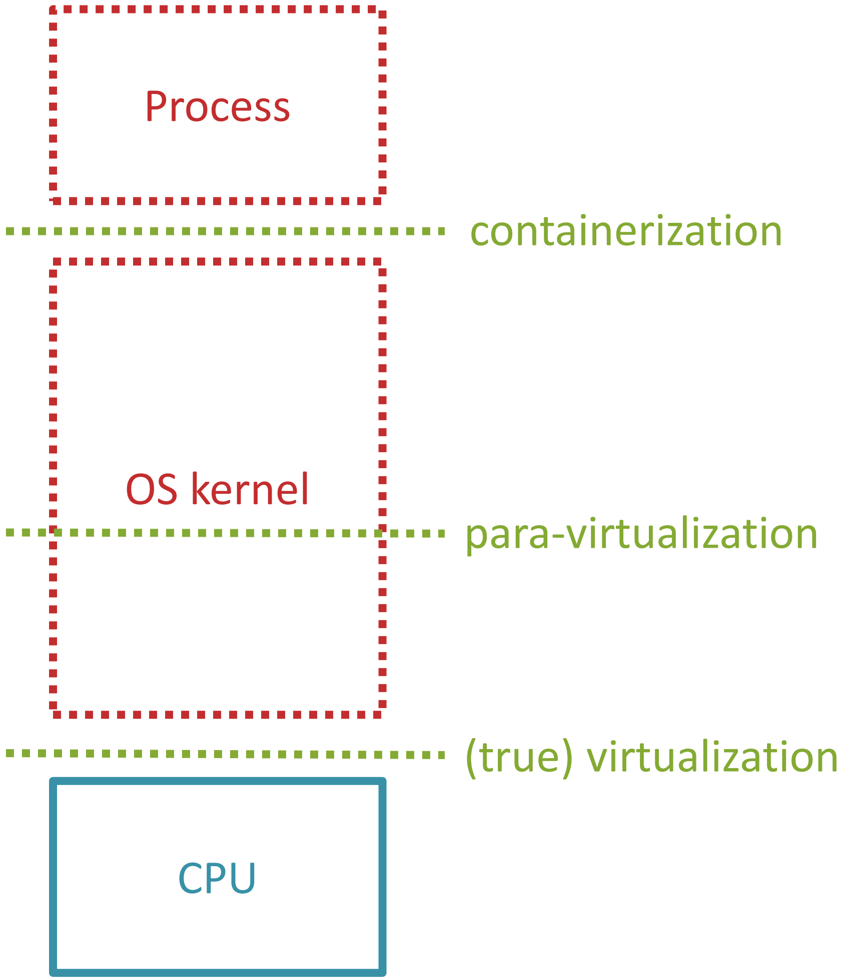
Plain Old Execution Environment



- Without virtualization, the separation between processes is deemed insufficient
  - Operating systems (since Unix) are built to facilitate inter-process communication
  - Processes on the same machine compete for resources (memory, CPUs)
  - Processes share global name spaces (file names, port numbers, UIDs, …)
- In theory, communication, competition and access are limited by priority, environment, and access-rights mechanisms
  - Nobody believes that these old mechanisms are sufficient against modern risks
  - Access rights cannot solve naming conflicts
    - Cannot have two web servers on port 80
    - Cannot have two gcc versions with the same /usr/include

Flavors of virtualization

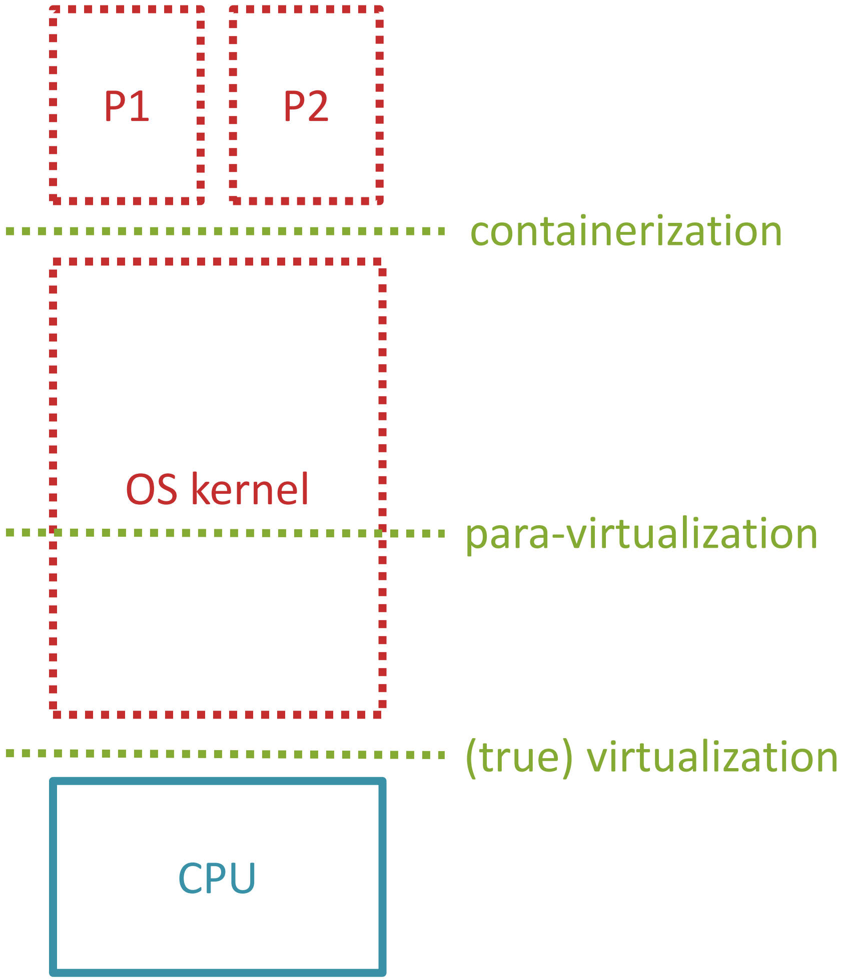
Virtualization at different layers



- Containerization
  - OS kernel improved so that it now offers different views (via the same interface) for different processes
- Para-virtualization
  - Lower layers of OS kernel modified so that multiple kernels may coexist on the same CPU
- (True) virtualization
  - Hardware support in CPU and/or emulation by software enables coexistence of multiple unmodified OS kernels on the same CPU

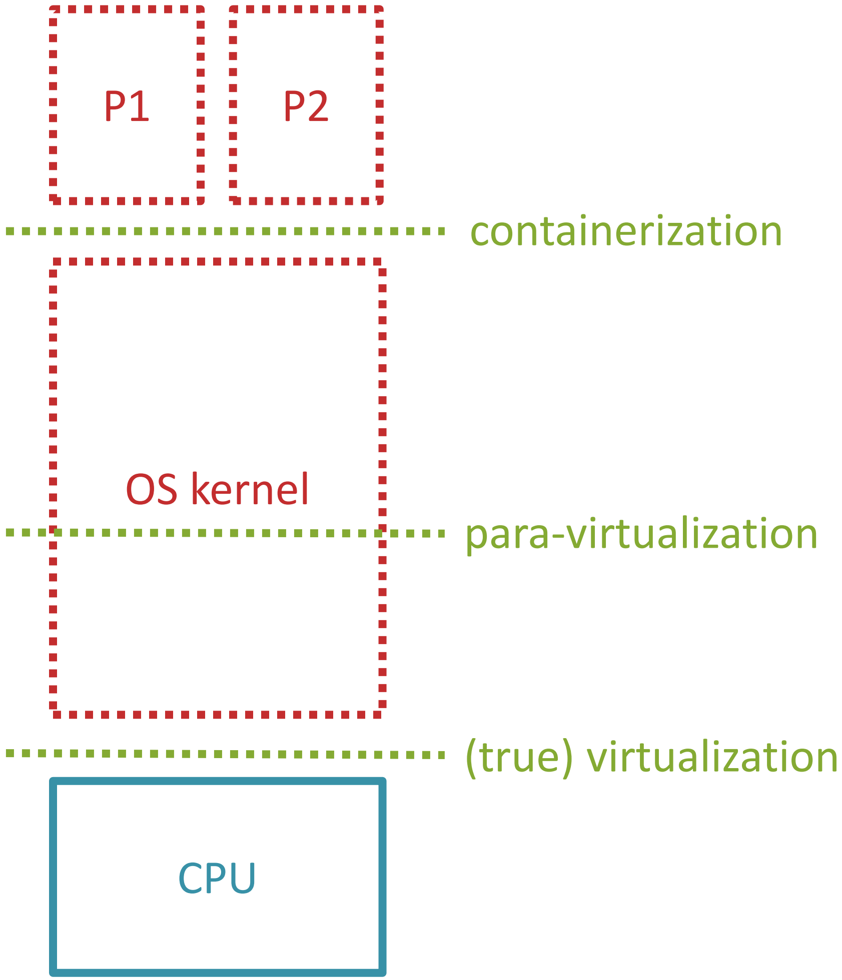
- Originally, these were three independent approaches
- Today, the three approaches may share some underlying hardware and/or software technology
- They may coexist on the same machine

Virtualization at different layers



- Outcome of virtualization
  - A set of processes lives in an illusion that they are alone at a hardware machine
  - In containerization, this illusion is created by the OS kernel
    - The same kernel may be shared by several such sets of processes
  - In para- and true virtualization, also the OS kernel lives in this illusion
    - OS kernels always need to feel alone
    - In para-virtualization, this applies only to the upper, unmodified majority of the kernel
    - Each such set of processes has its own kernel
- For software developers, the outcome is almost identical for the three approaches
- For system maintenance, there is huge difference between containerization and virtualization
  - Think about updates to the kernel(s)

Virtualization at different layers

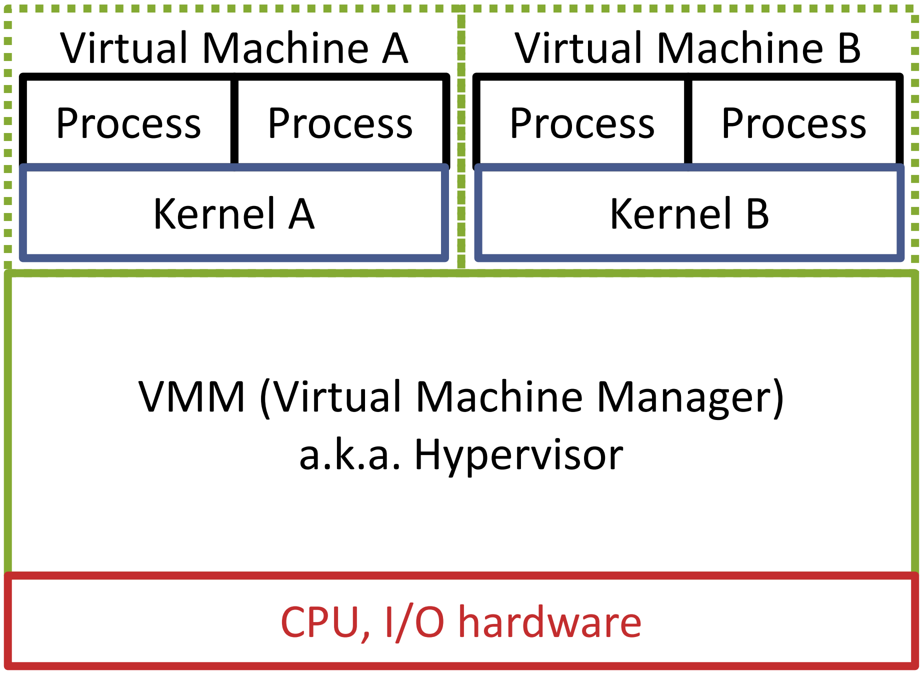


- Containers vs. virtual machines
  - Originally, containerization and virtualization were completely independent techniques
  - Now, they often share parts of the underlying technology
    - Some container systems use hardware-based isolation developed for virtual machines
    - Some virtual machine systems use software tricks developed for containers
    - There are interfaces/libraries/apps capable of controlling both containers and virtual machines
- There is still a fundamental difference:
  - Containers
    - Only one instance of OS kernel per hw machine
      - Shared among all containers
  - Virtual machines
    - Each virtual machine has its own instance of OS kernel
      - More memory required
    - In addition, there may be a *host* OS kernel

Types of Virtual Machine Systems

Type 1 (Bare Metal) Hypervisor

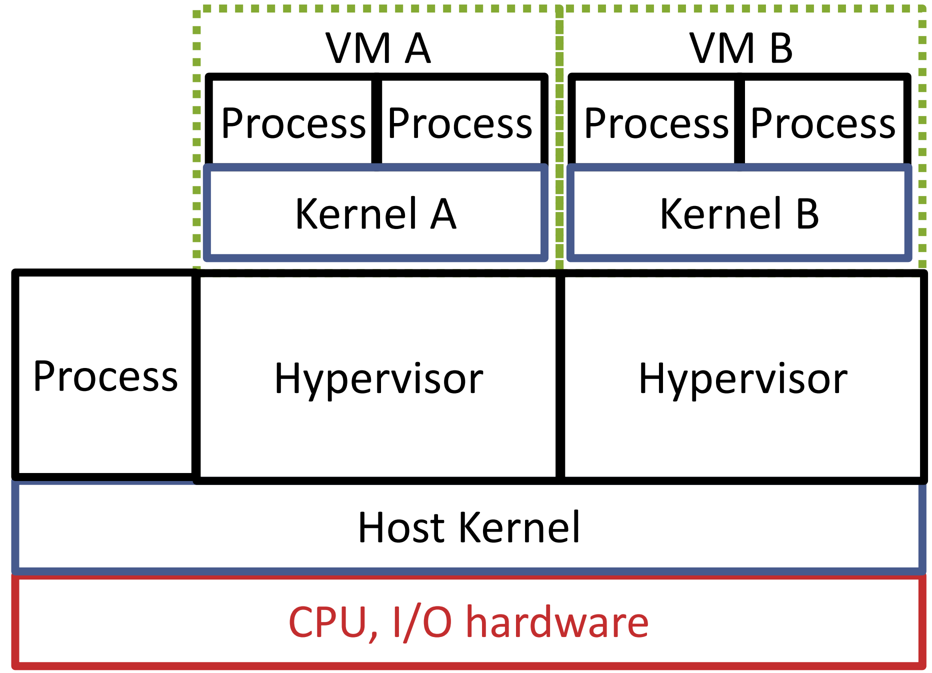
Example: VMWare ESXi



- Hypervisor on bare metal
  - Hypervisor directly performs all hardware access (CPU configuration, I/O)
    - Requires device drivers
    - Complex but fast

Type 2 (Hosted) Hypervisor

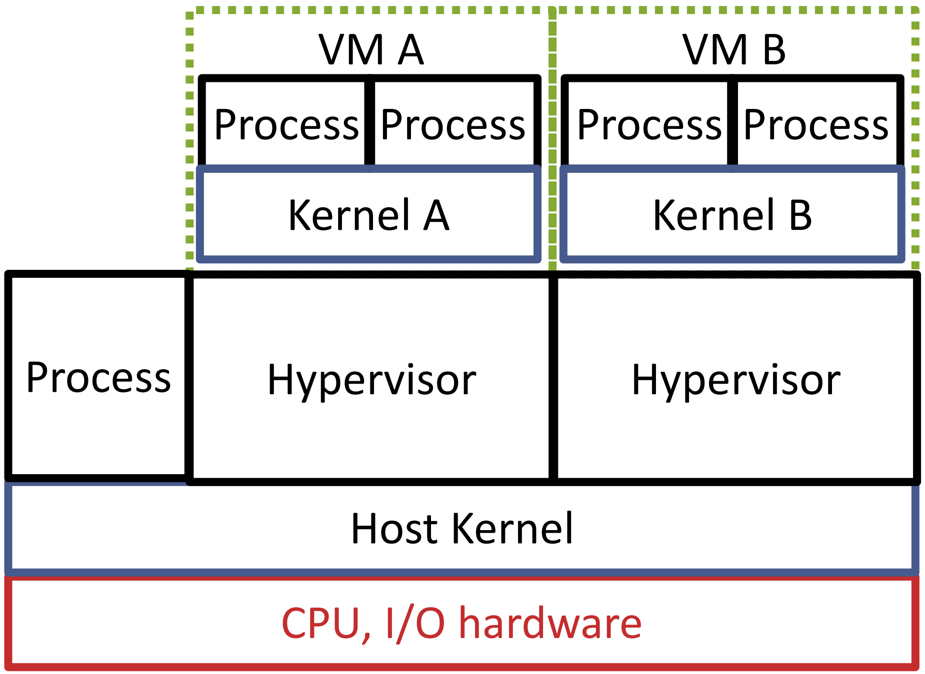
Example: VMWare Workstation Player



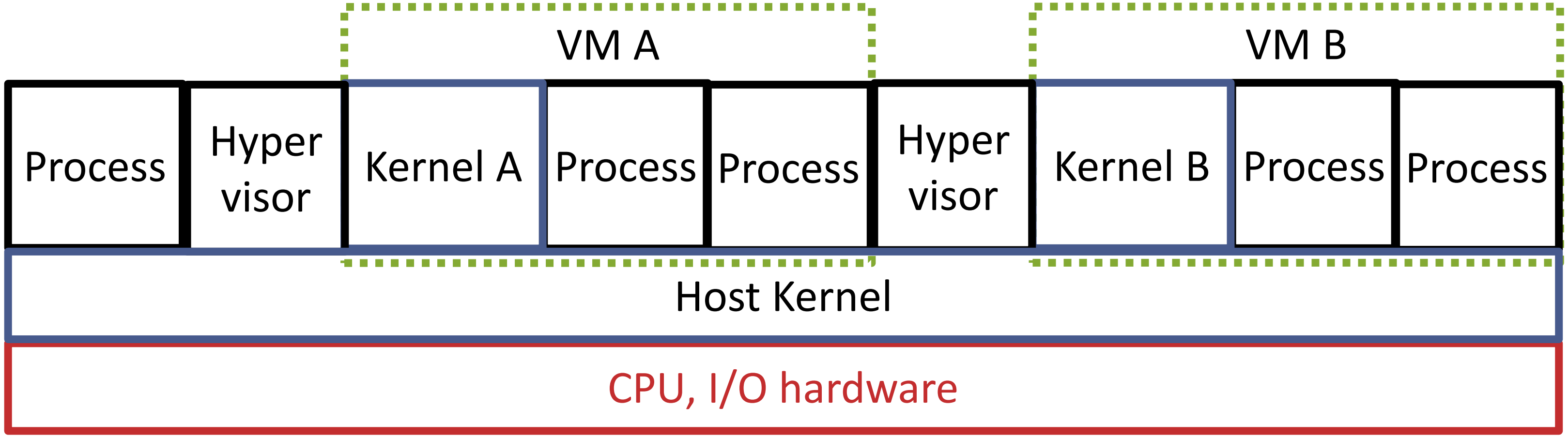
- Hypervisor above an host OS
  - Hypervisor is a (privileged) process
    - Often one per VM
    - I/O access performed by host kernel
    - CPU control requires support from the host kernel (debugging services)

Beware

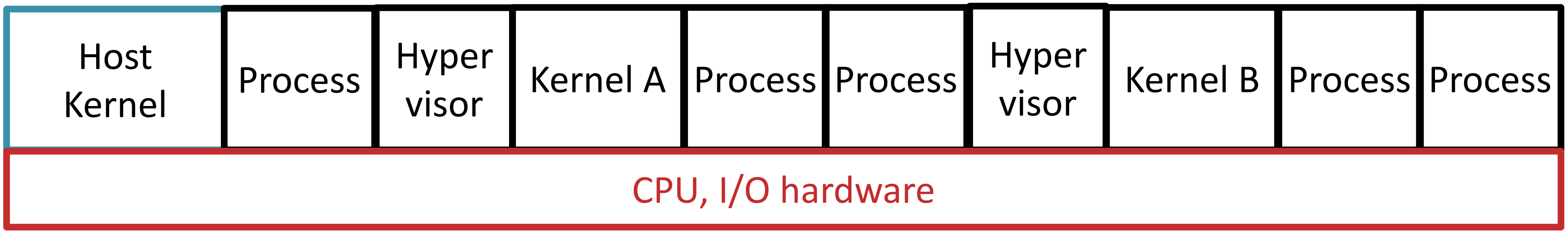
- Pictures like this are misleading



- The host kernel actually sees this:



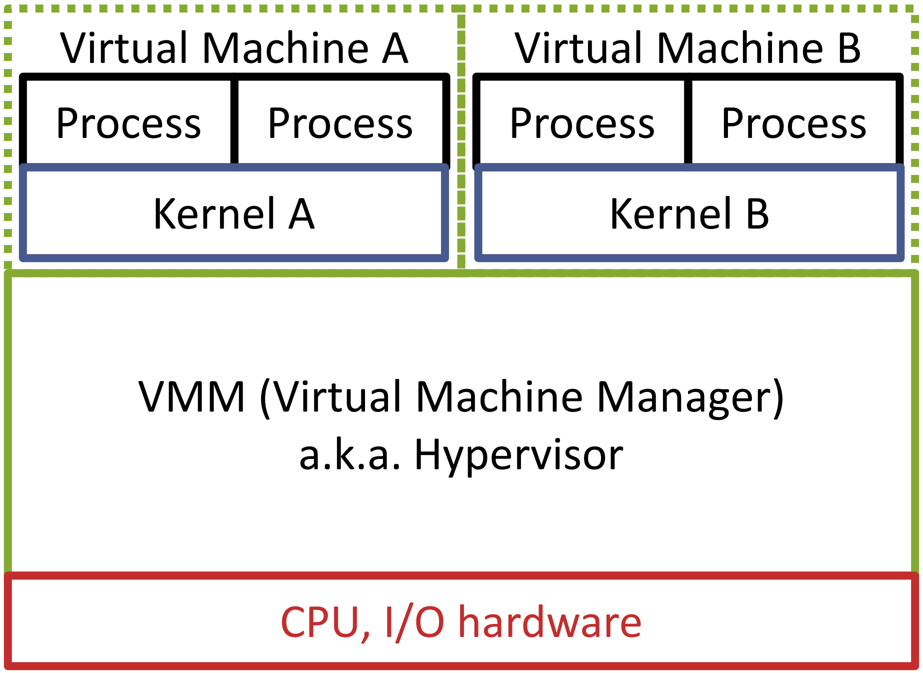
- The CPU sees this:



Flavors of Type 1 Hypervisors

Traditional

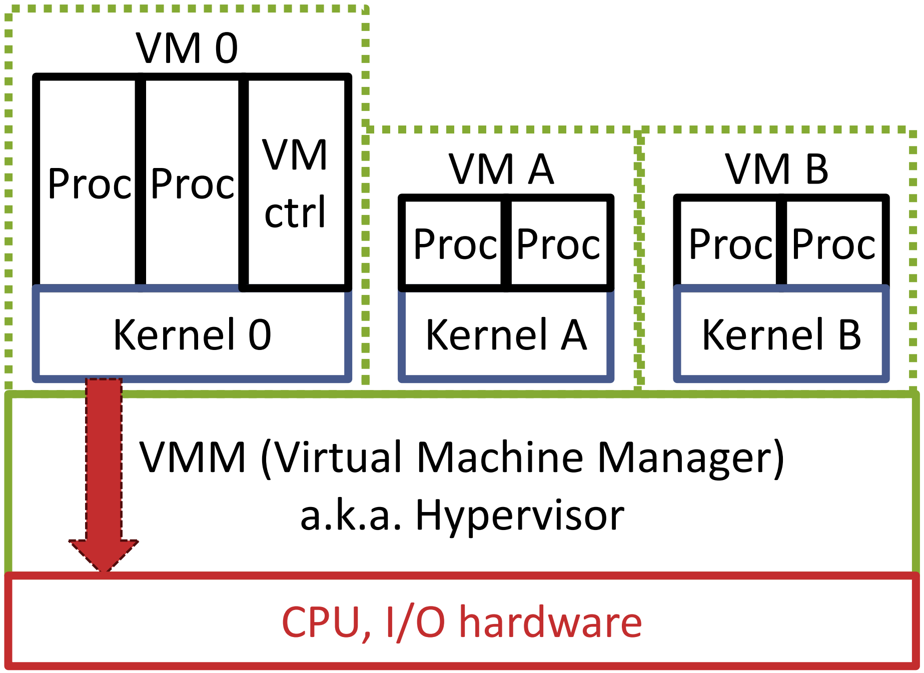
Example: VMWare ESXi



- Hypervisor performs I/O
  - Requires device drivers tailored for the hypervisor
  - Too costly development

With root partition (Microsoft terminology)

Example: Microsoft Windows + Hyper-V

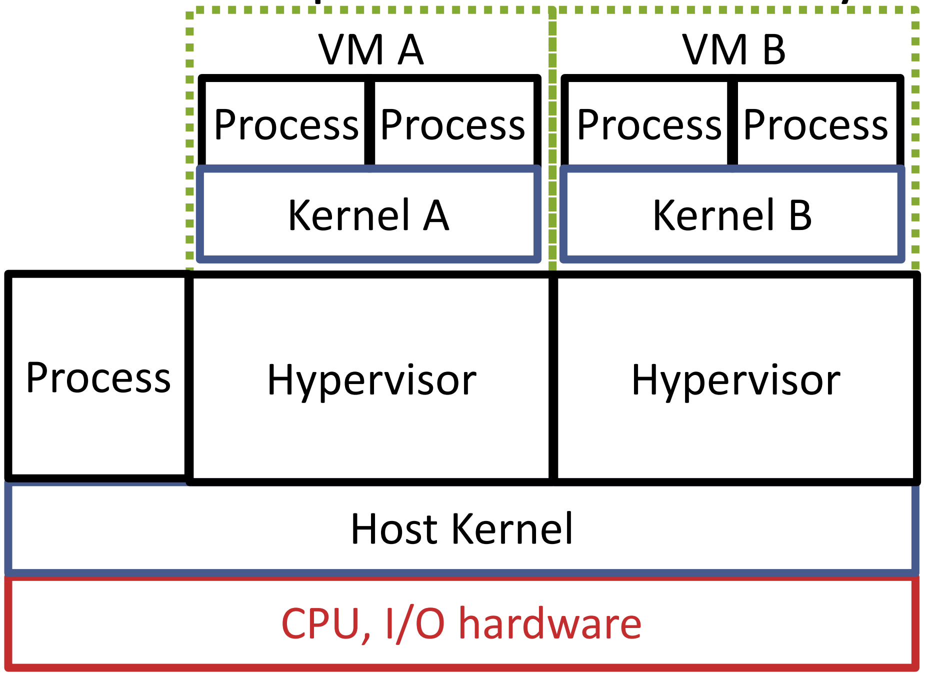


- Hypervisor only controls CPU
  - VM 0 aka Root partition
    - Allowed to directly access I/O hardware
    - Standard OS with device drivers
  - Hypervisor forwards I/O requests

Flavors of Type 2 Hypervisors

Implemented in user-space

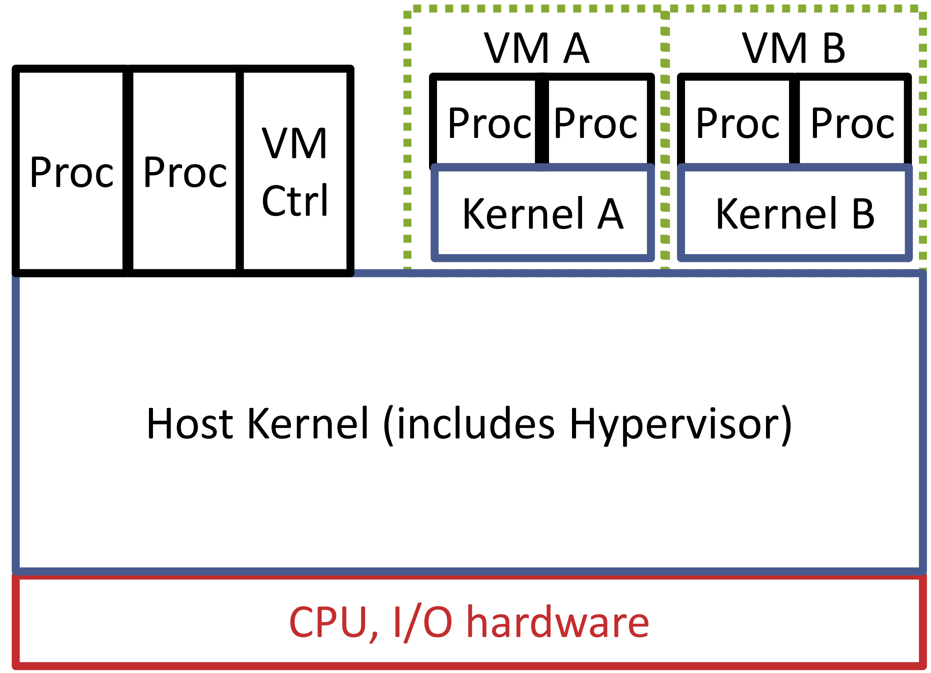
Example: VMWare Workstation Player



- Hypervisor above an host OS
  - Hypervisor is a (privileged) process
    - Often one per VM
    - I/O access performed by host kernel
    - CPU control requires support from the host kernel (debugging services)

Implemented in a kernel

Example: Linux KVM

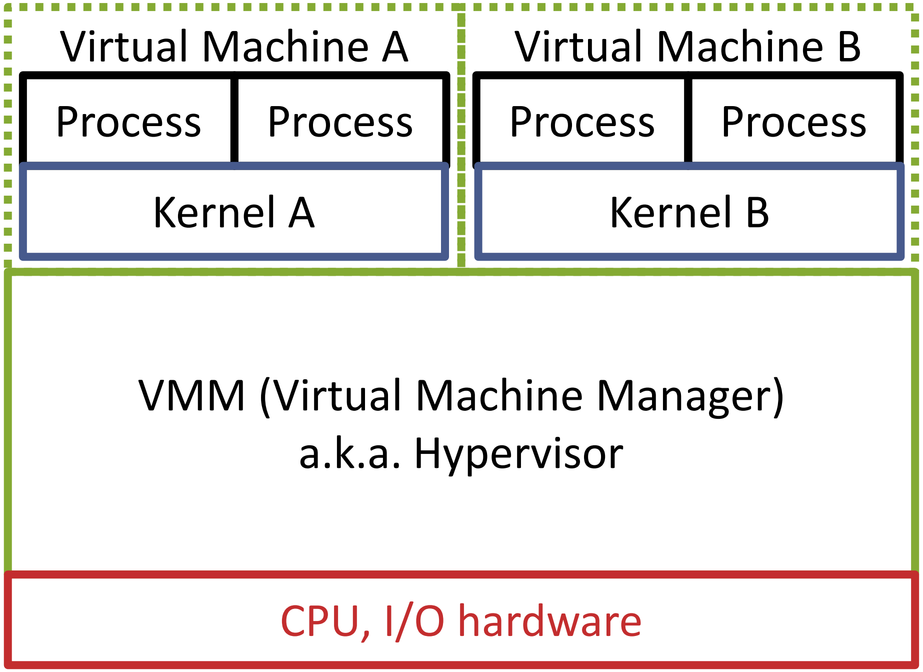


- Hypervisor integrated in kernel
  - Fast
    - No need to indirect CPU control via kernel service
  - Complex and dangerous
    - Kernels were not designed for this

Where is the difference? Only in the history.

Traditional type 1 hypervisor

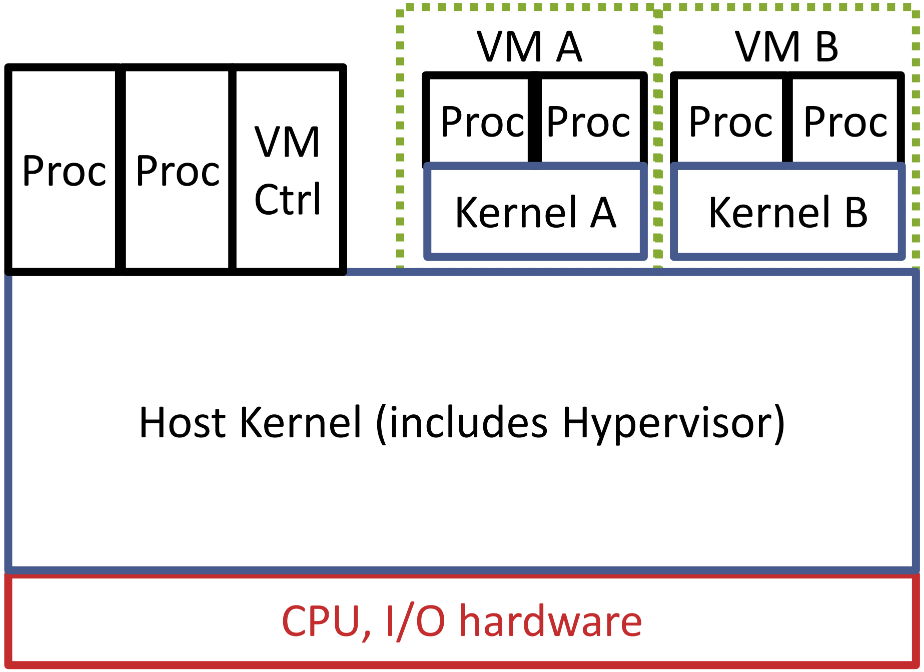
Example: VMWare ESXi



- Hypervisor does everything
  - CPU control, time sharing, and I/O in the same project
  - Complex and dangerous

Type 2 implemented in a kernel

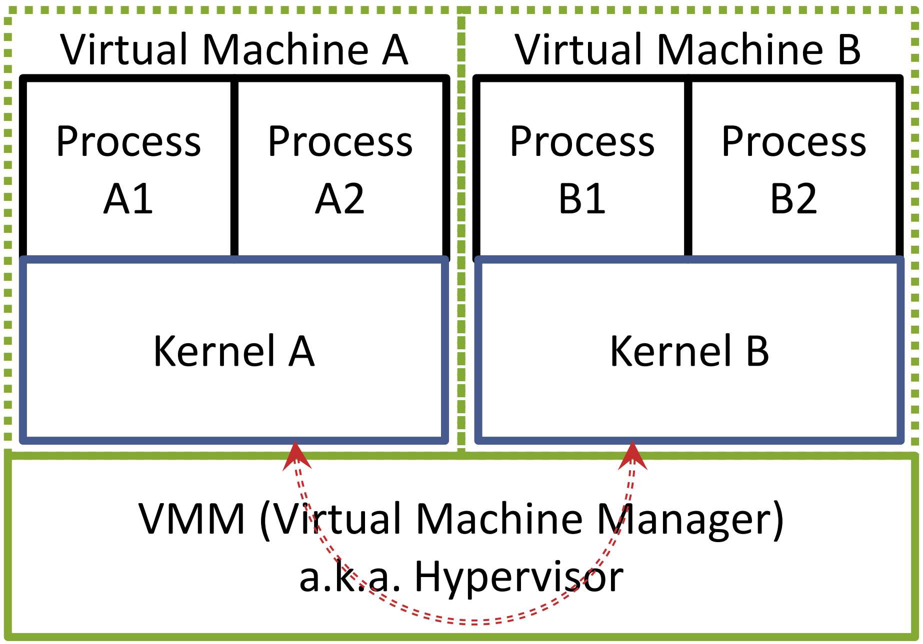
Example: Linux KVM



- Hypervisor implanted in kernel
  - CPU control, time sharing, and I/O in the same project
  - Complex and dangerous

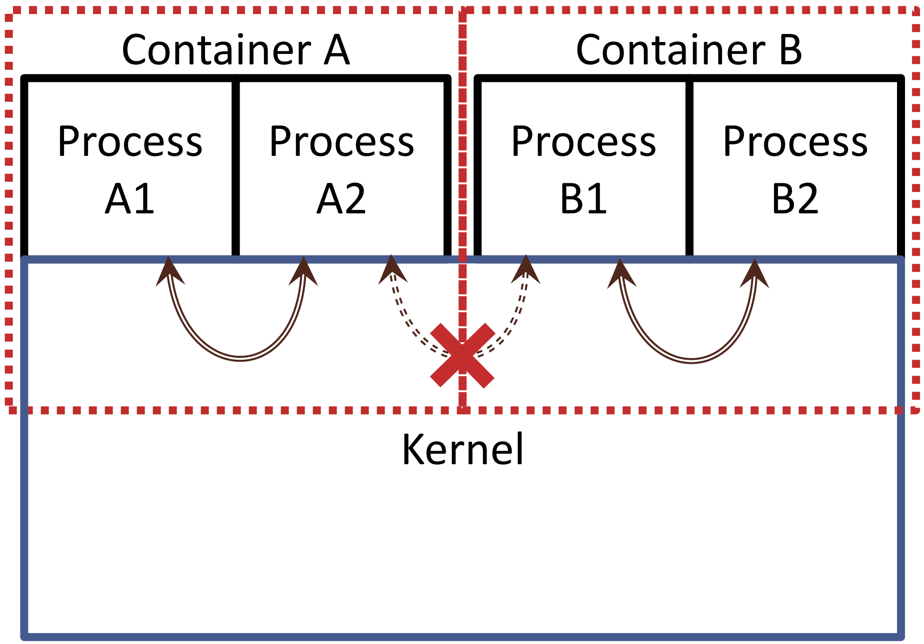
Virtual Machines vs. Containers

Virtual Machines



- Inherent safety
  - Kernel-HW interface was not designed for Kernel-Kernel communication
  - VMM adds well-controled holes into a natural barrier

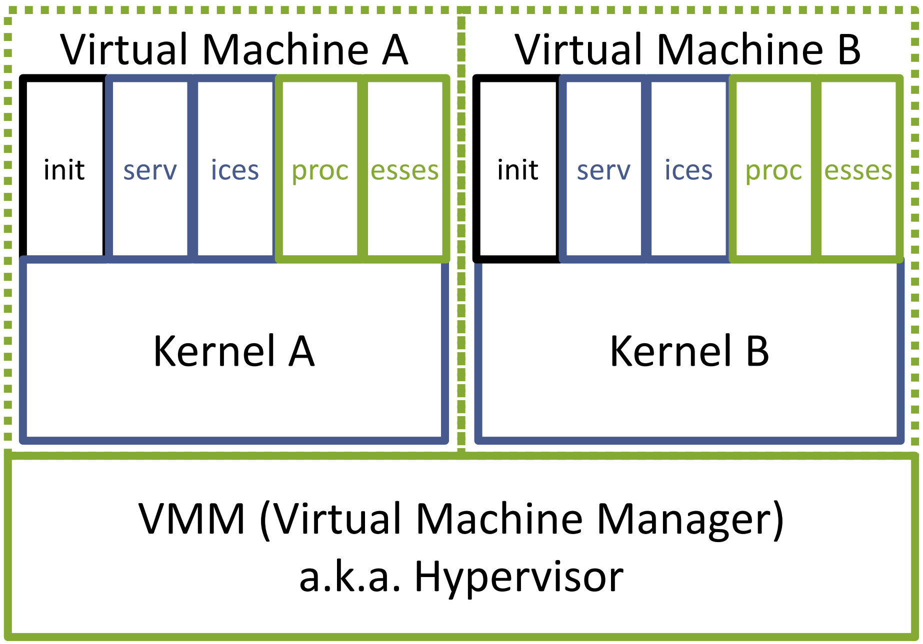
Containers



- Limited safety
  - Process-Kernel interface was designed for Process-Process communication
  - Containerization requires blocking existing communication channels

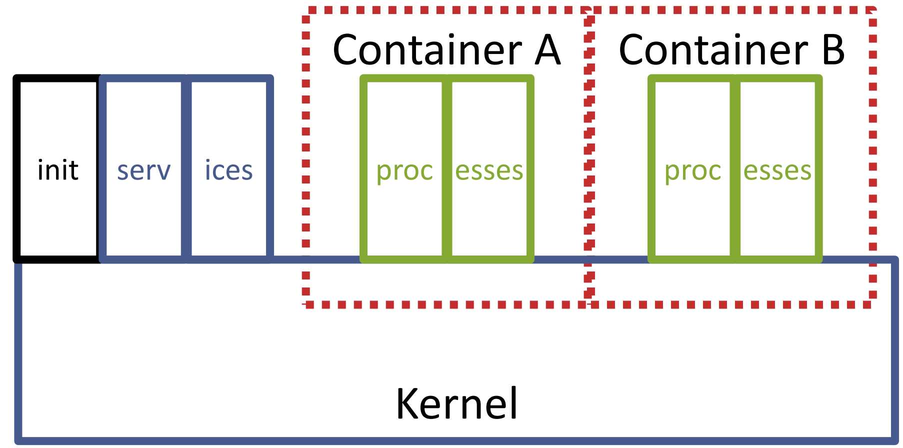
Virtual Machines vs. Containers

Virtual Machines



- Each VM is a complete OS
  - Each VM runs its services in specific settings
  - User (admin) processes (e.g. install scripts) can control services (edit /etc/..., run systemctl, ...)

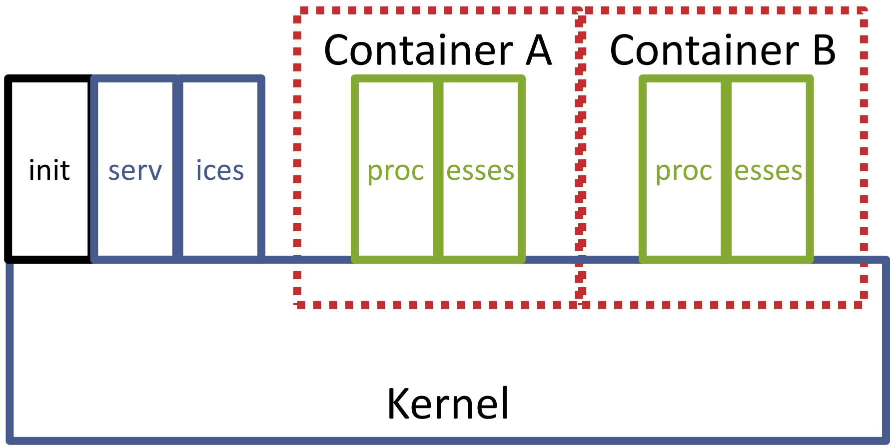
Containers



- Container is not a complete OS
  - Services shared among containers
    - Dependency hell still present
    - Processes inside containers usually cannot control services outside containers - their install scripts cannot run inside containers

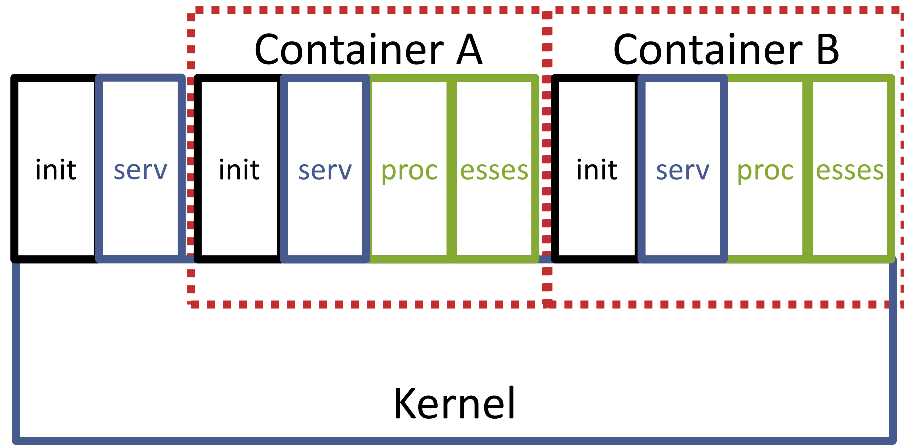
Plain vs. System Containers

Plain Containers



- Container is not a complete OS
  - Services shared among containers
    - Dependency hell still present
    - Processes inside containers usually cannot control services outside containers - install scripts cannot run inside containers

System Containers



- System container resembles a complete OS
  - Each container contains its service manager (init)
    - Install scripts work inside containers
  - The illusion is not yet complete
    - Certain privileges/capabilities/roles are hardwired in Linux kernel and denied for containers