Gabriele **Proietti Mattia**

Advanced Operating Systems and Virtualization

[12] Virtualization

Department of Computer, Control and Management Engineering "A. Ruberti", Sapienza University of Rome

gpm.name · proiettimattia@diag.uniroma1.it

A.Y. 2020/2021 · v1

Outline

12.1

12. Virtualization

Introduction

System Virtualization

Virtualization allows to show different resources from the physical ones. More operating systems can be run on the same hardware.

A Virtual Machine is a mixture of software- and hardware-based facilities. The software component that is in charge of managing the Virtual Machine is called the Hypervisor or VMM (Virtual Machine Monitor).

The main advantages of the virtualization are:

- isolation of different execution environments (on the same hardware)
- reduction of hardware and administration costs

Host & Guest

We can distinguish between:

- **host system**: the real system where (software implemented) virtual machines run
- **guest system**: the system that runs on top of a (software implemented) virtual machine

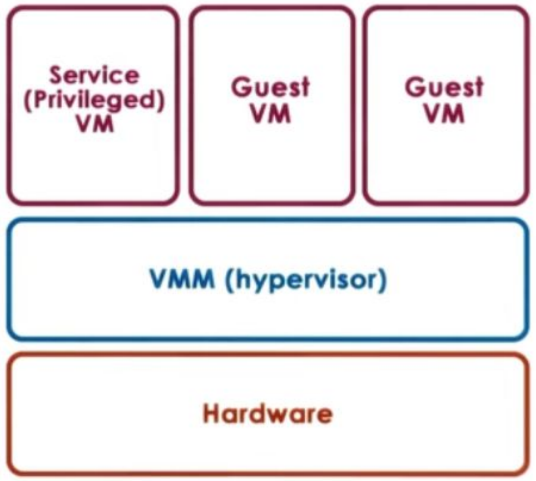
The roles of the Hypervisor are:

- managing hardware resources provided by the host system
- making virtualized resources available to the guest system in a correct and secure way

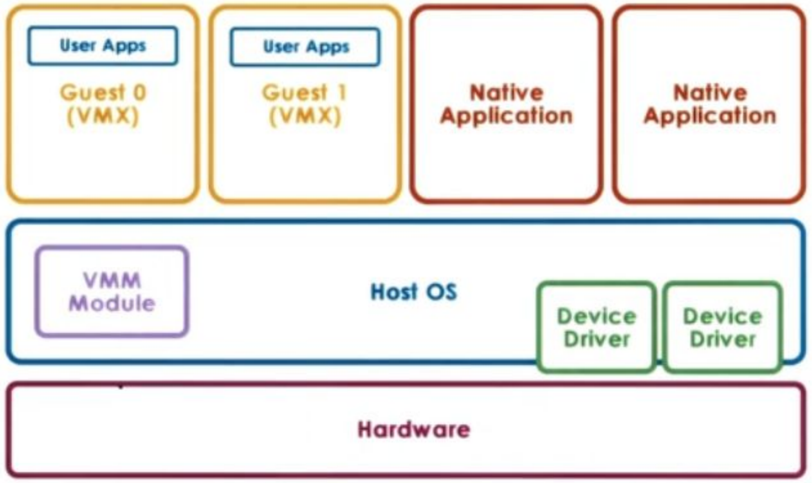
There are two kinds of hypervisors:

- **Native Hypervisor**: runs with full capabilities on bare metal. It resembles a lightweight virtualization kernel operating on top of the hardware.
- **Hosted Hypervisor**: it runs as an application, which accesses host services via system calls

Native vs Hosted Hypervisor



Native Hypervisor



Hosted Hypervisor

https://applied-programming.github.io/Operating-Systems-Notes/9-Virtualization/

Kinds of virtualization

We can have two kinds of virtualizations:

- ***software-based***: the guest application code runs **directly** on the processor, while the guest privileged code is **translated** and the translated code runs on the processor.

  The translated code is slightly larger and usually **runs more slowly** than the native version. As a result, guest applications, which have a small privileged code component, run with speeds very close to native. Applications with a significant privileged code component, such as system calls, traps, or page table updates can run slower in the virtualized environment.

- ***hardware-assisted***: certain processors provide hardware assistance for CPU virtualization. When using this assistance, the guest can use a **separate** mode of execution called **guest mode**. The guest code, whether application code or privileged code, runs in the guest mode. On certain events, the processor exits out of guest mode and enters root mode. The hypervisor executes in the root mode, determines the reason for the exit, takes any required actions, and restarts the guest in guest mode.

  When you use hardware assistance for virtualization, there is <u>**no need to translate the code**</u>. As a result, system calls or trap-intensive workloads run very close to native speed. Some workloads, such as those involving updates to page tables, lead to a large number of exits from guest mode to root mode. Depending on the number of such exits and total time spent in exits, hardware-assisted CPU virtualization can speed up execution significantly.

https://docs.vmware.com/en/VMware-vSphere/7.0/com.vmware.vsphere.resmgmt.doc/GUID-B14C8267-C2A4-4BF8-B680-70C2B350B325.html

12.2

12. Virtualization

Software-based Virtualization

Software-based Virtualization

The instruction of the virtual machine are executed by the native physical CPU in the host platform, but a subset of the instruction set must be emulated.

No particular hardware component plays a role in virtualization, but there are different problems:

- what if ring `0` is required for guest activities? (Ring Aliasing)
- there could be the risk to bypass the VMM resource management policy in case of actual ring 0 access

The solution: ***ring de-privileging***. Which is implemented and assisted by:

- **binary translation**: for rewriting some Ring `0` instructions
- **shadowing** of some data structures, e.g. page table
- **I/O emulation** of not supported devices

Ring De-Privileging

A technique to let the guest kernel run at a privilege level that “simulates” 0

Two main strategies:

- **0 / 1 / 3 Model**:
  - VMM runs at ring 0
  - Kernel guest runs at ring 1 (not typically used by native kernels)
  - Applications still run at ring 3
  - This is the most used approach
- **0 / 3 / 3 Model**:
  - VMM runs at ring 0.
  - Kernel guest and applications run at ring 3.
  - too close to emulation, too high costs

0/1/3 Model

Applications (running at **ring 3**) cannot alter the state of the guest operating system (running at **ring 1**). The guest operating system cannot access privileged instructions and data structures of the host operating system we guarantee the isolation of guest systems. Any exception must be trapped by the VMM (at **ring 0**) and must be properly handled (e.g. by reflecting it into **ring 1** tasks).

Issues to cope with:

- Ring aliasing
- Virtualization of the interrupts
- Frequent access to privileged resources

Ring Aliasing

An OS kernel is designed to run at ring `0`, while it is actually being run at ring 1 for guest systems. This can create problems for instructions that was specifically designed for running in Ring `0` and they cannot work anymore:

- **Privileged instructions**: generate an exception if not run at CPL `0` (e.g. `hlt`, `lidt`, `lgdt`, `invd`, `mov %crx`)
- **I/O sensitive instructions**: they generate a trap if executed when CPL > IOPL (I/O Privilege Level) (e.g. `cli`, `sti`)
- **Other instructions**: they are simply skipped if not in Ring `0`, e.g. `popf` in the x86 architecture

The generated trap (general protection fault) must be handled by the VMM, so as to finally determine how to handle it (emulation vs interpretation). If they do not generate a trap, then they must be translated (binary translation) or hardware-assisted virtualization need to be used.

12.2.1

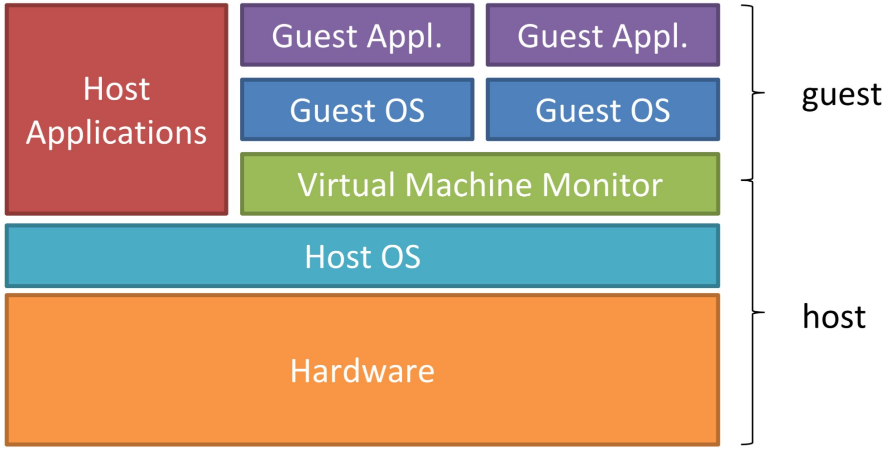
12. Virtualization

2. Software-based Virtualization

VirtualBox

The Virtual Box Example

Based on **hosted** hypervisor with ad-hoc kernel facilities, via classical special devices (**0/1/3 model**).



Pure software virtualization is supported for x86:

- Fast Binary Translation (code patching) the kernel code is analysed and modified before being executed
- Privileged instructions replaced with semantically equivalent blocks of code

Execution Modes and Context

Virtualbox lives in two contexts:

- **Guest context** (GC): execution context for the guest system. It is based on two modes:
    - *Raw mode*: native guest code runs at ring 3 or 1
    - *Hypervisor mode*: VirtualBox runs at ring `0`
- **Host context** (HC): execution context for userspace portions of VirtualBox (ring 3)
    - the running thread implementing the VM lives in this context upon a mode change
    - critical/privileged instructions are emulated in this context upon a GPF (General Protection Fault, e.g. due to ring aliasing)

VBOXGDT

Description	Offset	DPL	Base
Entry 0	$(0000)_H$	-	null
…	. . .	. . .	…
Kernel Code Segment	$(0060)_H$	1	
Kernel Data Segment	$(0068)_H$	1	
…	. . .	. . .	…
Virtualbox TSSD	$(FFE0)_H$	0	
…	. . . .	. . .	…
Hypervisor Data Segment	$(FFF0)_H$	0	
Hypervisor Code Segment	$(FFF8)_H$	0	

Kernel Code and Data Segments are registered with DPL 1, they are accessible with CPL=1

New TSSD which keeps information about stack at Ring 0 and 1

2 new segments for the Hypervisor are added with DPL=0

Original TSS

. . .	. . .
esp0	. . .
ss0	$(0068)_H$
esp1	unused
ss1	unused
. . .	. . .

VBOX TSS

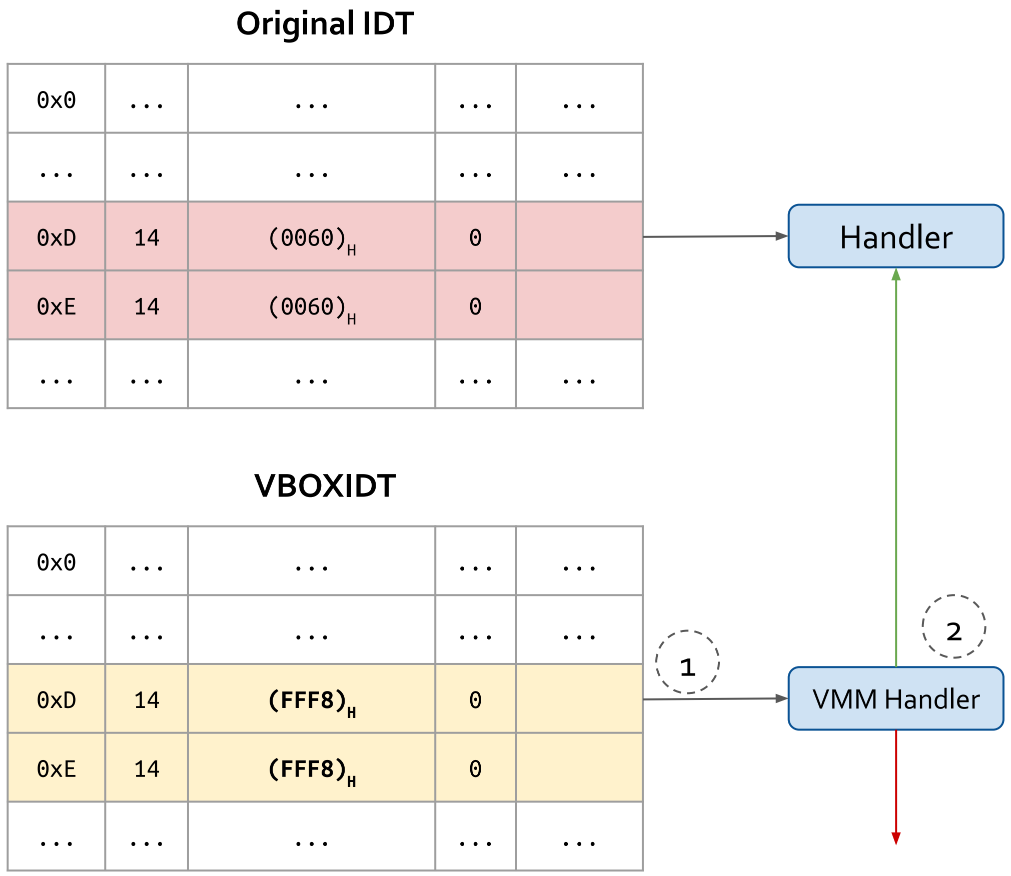
. . .	. . .
esp0	$(FE557000)_H$
ss0	$(FFF0)_H$
esp1	$(F70D3FF9)_H$
ss1	$(0069)_H$
. . .	. . .

VBOXIDT

Interrupt gate

Interrupt must be managed by the VMM. To this end, a wrapper for the IDT is generated.

Proper handlers are instantiated, which get executed by the Hypervisor upon traps. VMM can take control thanks to the ad-hoc segment selector (1) (at the GDT offset for the hypervisor code segment). In case of a "**genuine**" trap, the control goes to the native kernel, otherwise the **virtual** handler is executed (2).



Original IDT

0x0	...	...	...	...
...	...	...	...	...
0xD	14	$(0060)_H$	0	
0xE	14	$(0060)_H$	0	
...	...	...	...	...

VBOXIDT

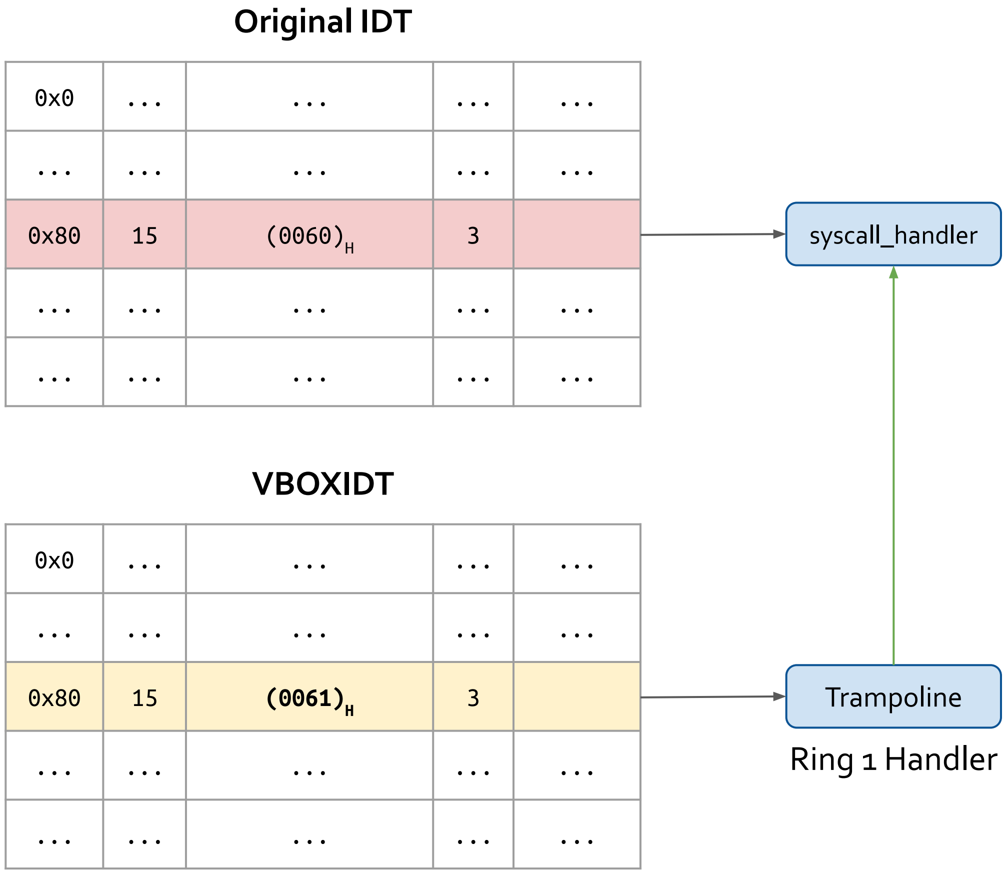
0x0	...	...	...	...
...	...	...	...	...
0xD	14	$(FFF8)_H$	0	
0xE	14	$(FFF8)_H$	0	
...	...	...	...	...

VBOXIDT

Gate `0x80`

INT 0x80 has an ad-hoc management. The syscall gate is modified so as to provide a segment selector with RPL = 1 and it indicates the GDT offset for the code segment (at ring 1).

Hence calling a system call **does not require interaction with the Hypervisor**. The *trampoline* handler is then used to launch the actual syscall handler



12.3

12. Virtualization

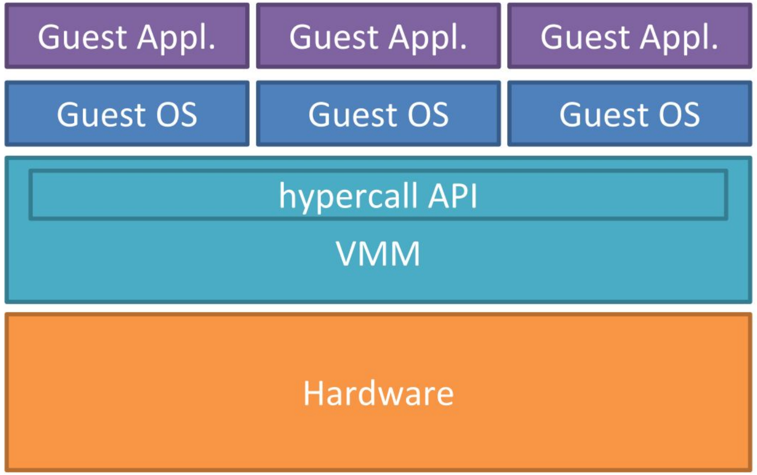
Paravirtualization

Paravirtualization

Paravirtualization is not that different from Binary Translation. BT changes "critical" or "dangerous" code into harmless code on the fly; paravirtualization does the same thing, but in the source code. Of course, changing the source code allows a bit more flexibility than changing everything on the fly, which has to happen quickly. One advantage is that paravirtualization eliminates a lot of unnecessary "traps" by the VMM (or Hypervisor), even more than BT.

The VMM offers a virtual interface (hypercall API) used by guest OS to access resources. To run privileged instructions, hypercalls are executed.

An example: Xen.



12.4

12. Virtualization

Hardware-assisted Virtualization

VT-x

Intel Vanderpool Technology, referred to as VT-x, represents Intel's virtualization technology on the x86 platform. Its goal is simplify VMM software by closing virtualization holes by design.

- ring compression (lack of OS/Applications separations if only 2 rings are used)
- non-trapping instructions (some instructions at ring 1 are not trapped, for example popf)
- excessive trapping

Eliminate need for software virtualization (i.e paravirtualization, binary translation). The CPU flag for VT-x capability is "vmx". "VMX" stands for **Virtual Machine Extensions**, which adds 13 new instructions: `VMPTRLD`, `VMPTRST`, `VMCLEAR`, `VMREAD`, `VMWRITE`, `VMCALL`, `VMLAUNCH`, `VMRESUME`, `VMXOFF`, `VMXON`, `INVEPT`, `INVVPID`, and `VMFUNC`. These instructions permit entering and exiting a virtual execution mode where the guest OS perceives itself as running with full privilege (ring `0`), but the host OS remains protected.

In the Linux kernel the module that is in charge of enabling the support for VT-x (or AMD-V) is **KVM**, that makes the kernel able to work as an hypervisor.

Virtual-Machine Extension (VMX)

Virtual Machine Extensions define CPU support for VMs on x86 by a new form of operation called **VMX operation**. There are two kinds of VMX operation:

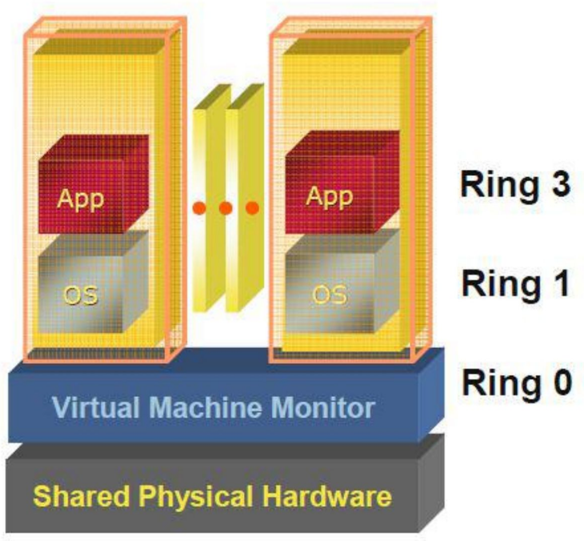
- **root**: VMM runs in VMX root operation
- **non-root**: Guest runs in VMX non-root operation

This eliminates ring de-privileging for guest OS. The VMX Transitions between VMX root operation and VMX non-root operation are called:

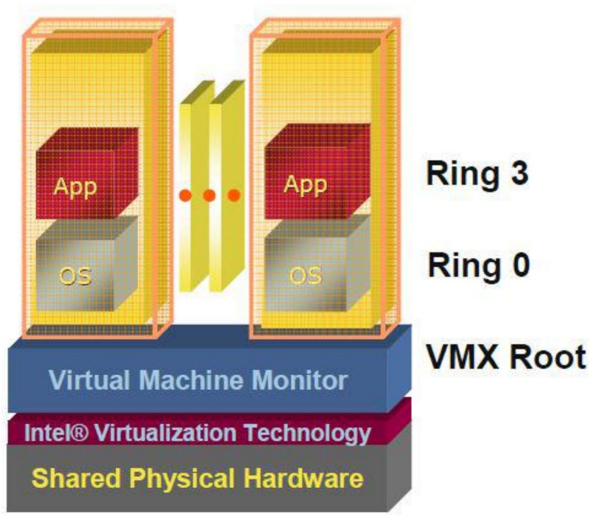
- VM Entry: Transitions into VMX non-root operation.
- VM Exit: Transitions from VMX non-root operation to VMX root operation.

Registers and address space are swapped in one atomic operation. In general VM Entry and Exit are **very heavy operations** at least in the first implementation, they were such heavy that paravirtualization was faster (in terms of CPU cycles).

Virtual-Machine Extension (VMX)



Pre VT-x



Post VT-x

Pre VT-x	Post VT-x
VMM ring deprivileging of guest OS	VMM executes in VMX root-mode
Guest OS aware it's not at Ring 0	Guest OS de-privileging eliminated
	Guest OS runs directly on hardware

VMCS: VM Control Structure

The virtual-machine control data structure (**VMCS**) is defined for VMX operation. A VMCS manages transitions in and out of VMX non-root operation (VM entries and VM exits) as well as processor behavior in VMX non-root operation.

The VMCS consists of six logical groups:

- **Guest-state area**: processor state saved into the guest-state area on VM exits and loaded on VM entries.
- **Host-state area**: processor state loaded from the host-state area on VM exits.
- **VM-execution control fields**: fields controlling processor operation in VMX non- root operation.
- **VM-exit control fields**: fields that control VM exits.
- **VM-entry control fields**: fields that control VM entries.
- **VM-exit information fields**: read-only fields to receive information on VM exits describing the cause and the nature of the VM exit.

MMU Virtualization with VT-x: VPIDs

First generation VT-x forces TLB flush on each VMX transition, but this obviously has as a consequence:

- performance loss on all VM exits
- performance loss on most VM entries, since the guest page tables are not always modified always

Better VMM software control of TLB flushes is beneficial and thus is was introduced the VPID:

- 16-bit **V**irtual-**P**rocessor-**ID** field in the VMCS
- cached linear translations tagged with VPID value
- no flush of TLBs on VM entry or VM exit if VPID active
- TLB entries of different virtual machines can all co-exist in the TLB

12.4.1

12. Virtualization

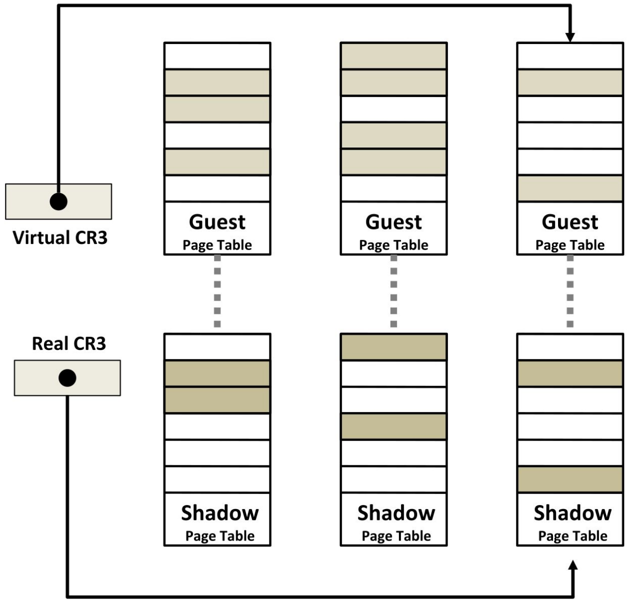
Virtualization of Memory

Shadow Page Tables

Granting the guest OS direct access to the MMU would mean **loss of control** by the virtualization manager, some of the work of the x86 MMU needs to be duplicated in software for the guest OS using a technique known as **shadow page tables**. This involves denying the guest OS any access to the actual page table entries by trapping access attempts and emulating them instead in software.

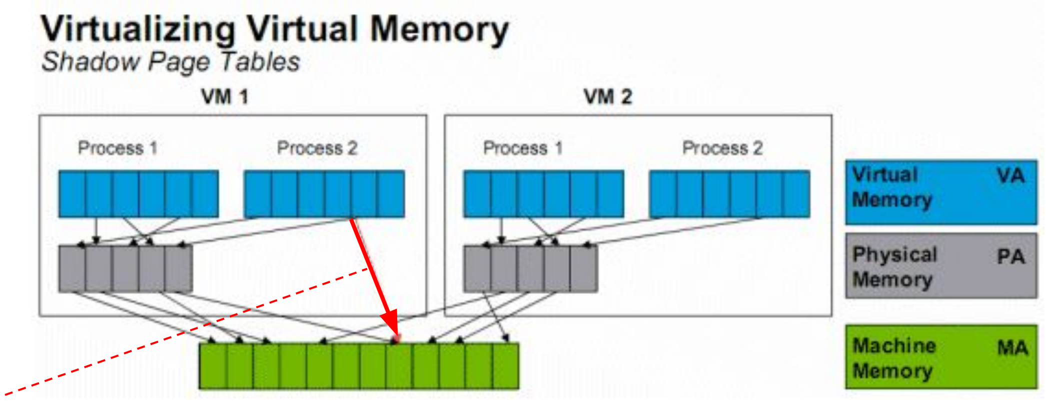
The x86 architecture uses hidden state to store segment descriptors in the processor, so once the segment descriptors have been loaded into the processor, the memory from which they have been loaded may be overwritten and there is no way to get the descriptors back from the processor. *Shadow descriptor* tables must therefore be used to track changes made to the descriptor tables by the guest OS.

Shadow Page Tables



However, this has different **drawbacks**:

- maintaining consistency between guest page tables and shadow page tables leads to an overhead: VMM traps
- loss of performance due to TLB flush on every "world-switch"
- memory overhead due to shadow copying of guest page tables



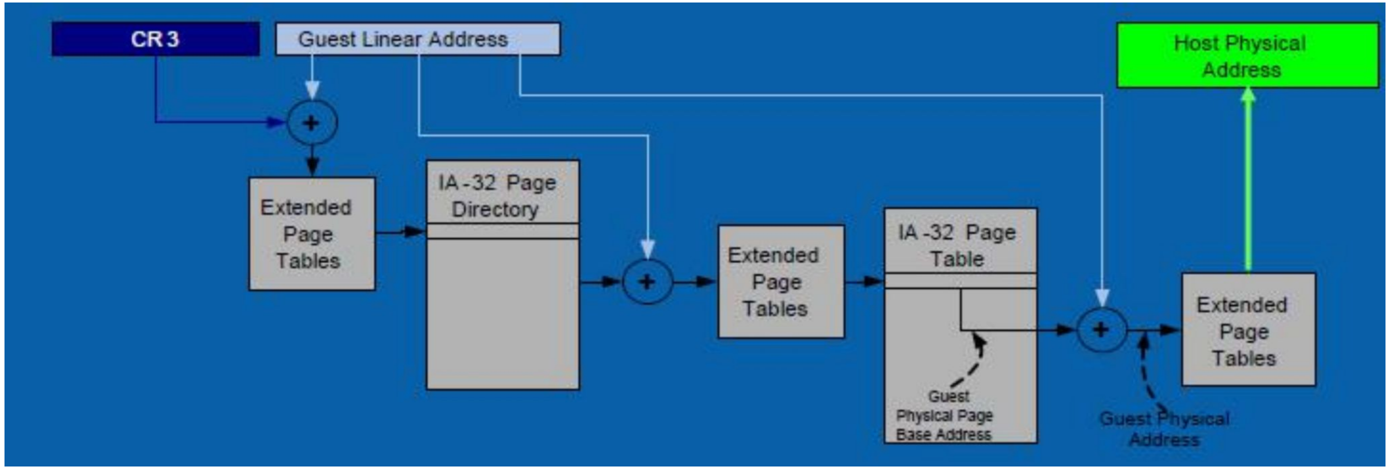
Thanks to shadowing we can go directly from blue to green, without passing to grey

https://userpages.umbc.edu/~dgorin1/451/virtualization/HWvirtualization.htm

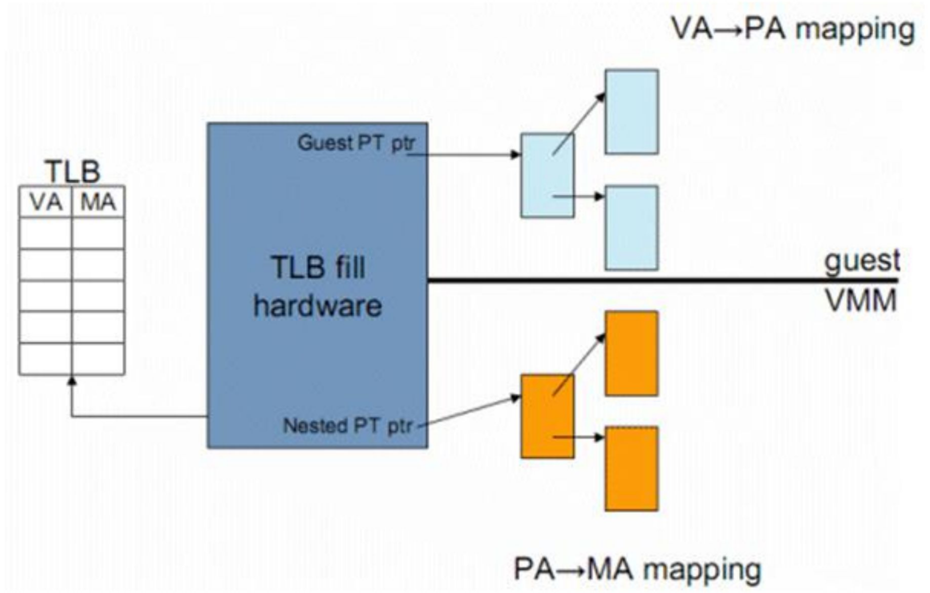
Nested / Extended Page Tables (EPT)

The Extended Page-Table mechanism (EPT) is used to support the virtualization of physical memory. It translates the guest-physical addresses used in VMX non-root operation.

Guest-physical addresses are translated by traversing a set of EPT paging structures to produce physical addresses that are used to access memory.



Nested / Extended Page Tables (EPT)



As you can see in the picture above, a CPU with hardware support for nested paging caches both the Virtual memory (Guest OS) to Physical memory (Guest OS) as the Physical Memory (Guest OS) to real physical memory transition in the TLB. The TLB has a new VM specific tag, called the **Address Space IDentifier** (ASID). This allows the TLB to keep track of which TLB entry belongs to which VM. The result is that a VM switch does not flush the TLB. The TLB entries of the different virtual machines all coexist peacefully in the TLB… provided the TLB is big enough of course!

https://userpages.umbc.edu/~dgorin1/451/virtualization/HWvirtualization.htm

Nested / Extended Page Tables (EPT)

Pros / Cons

Advantages:

- Simplified VMM design
- Guest page table modifications need not to be trapped, hence VM exits reduced
- Reduced memory footprint compared to shadow page table algorithms

Disadvantages:

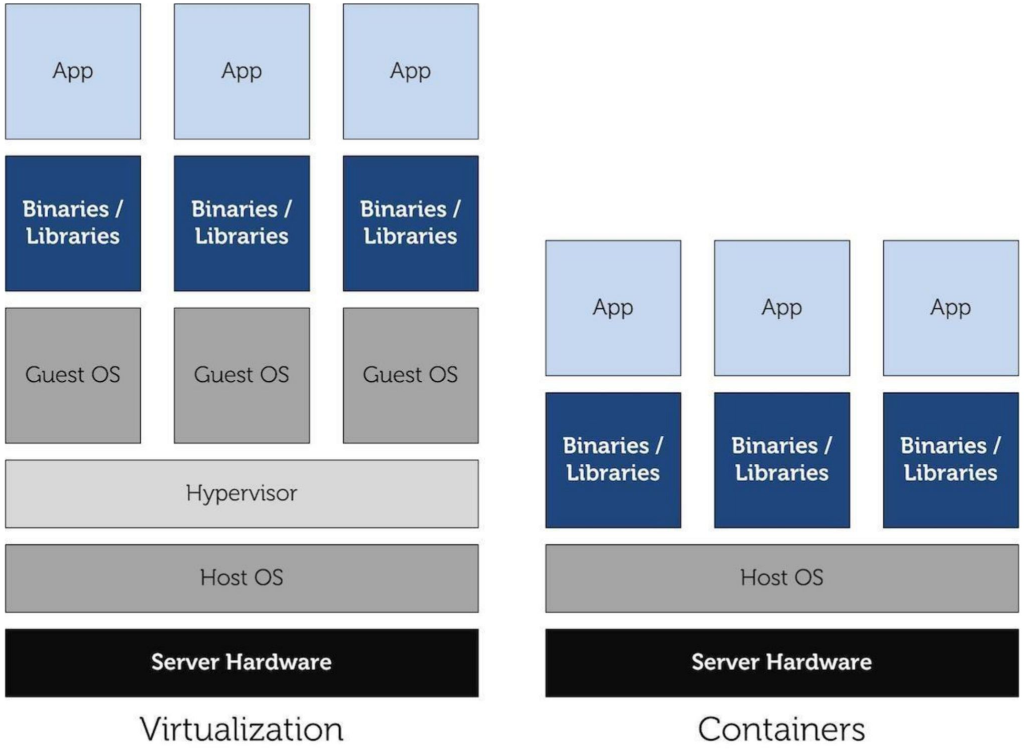
- TLB miss is very costly since guest-physical address to machine address needs an extra EPT walk for each stage of guest-virtual address translation

12.5

12. Virtualization

Linux Containers

Linux Containers



Underlying Kernel Mechanisms

The kernel mechanisms used for implementing containers are:

- `cgroups`: manage resources for groups of processes
- `namespaces`: per-process resource isolation
- `seccomp`: limit the possible syscalls to be executed to exit(), sigreturn(), read() and write(), the last two only to already-opened file descriptors
- capabilities: privileges and permissions available to processes

12.5.1

12. Virtualization

5. Linux Containers

cgroups

cgroups

Overview

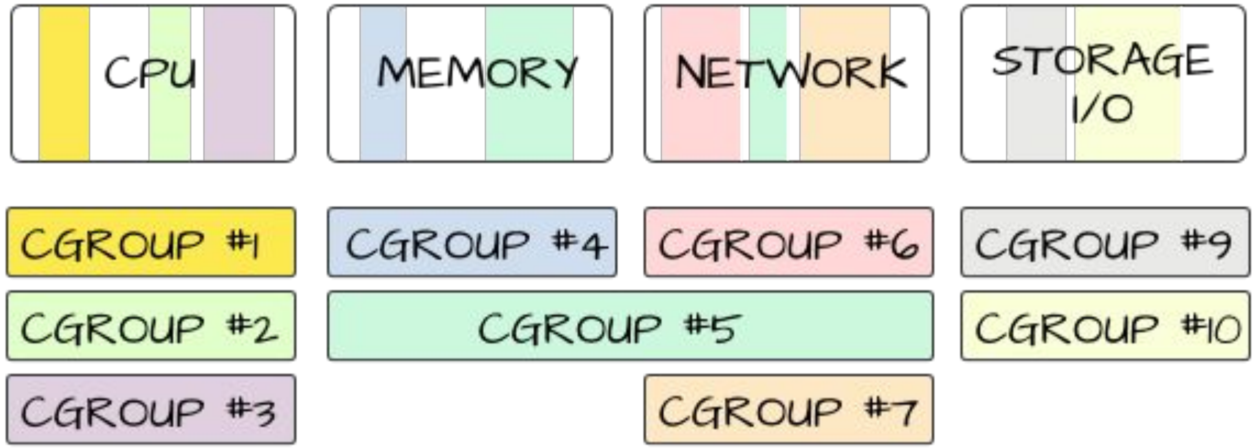
Control Groups provide a mechanism for **aggregating/partitioning sets of tasks**, and all their future children, into hierarchical groups with specialized behaviour.

Definitions

- A ***cgroup*** associates a set of tasks with a set of parameters for one or more subsystems.
- A ***subsystem*** is a module that makes use of the task grouping facilities provided by cgroups to treat groups of tasks in particular ways. A subsystem is typically a "resource controller" that schedules a resource or applies per-cgroup limits, but it may be anything that wants to act on a group of processes, e.g. a virtualization subsystem.
- A ***hierarchy*** is a set of cgroups arranged in a tree, such that every task in the system is in exactly one of the cgroups in the hierarchy, and a set of subsystems; each subsystem has system-specific state attached to each cgroup in the hierarchy. Each hierarchy has an instance of the cgroup virtual filesystem associated with it.

cgroups

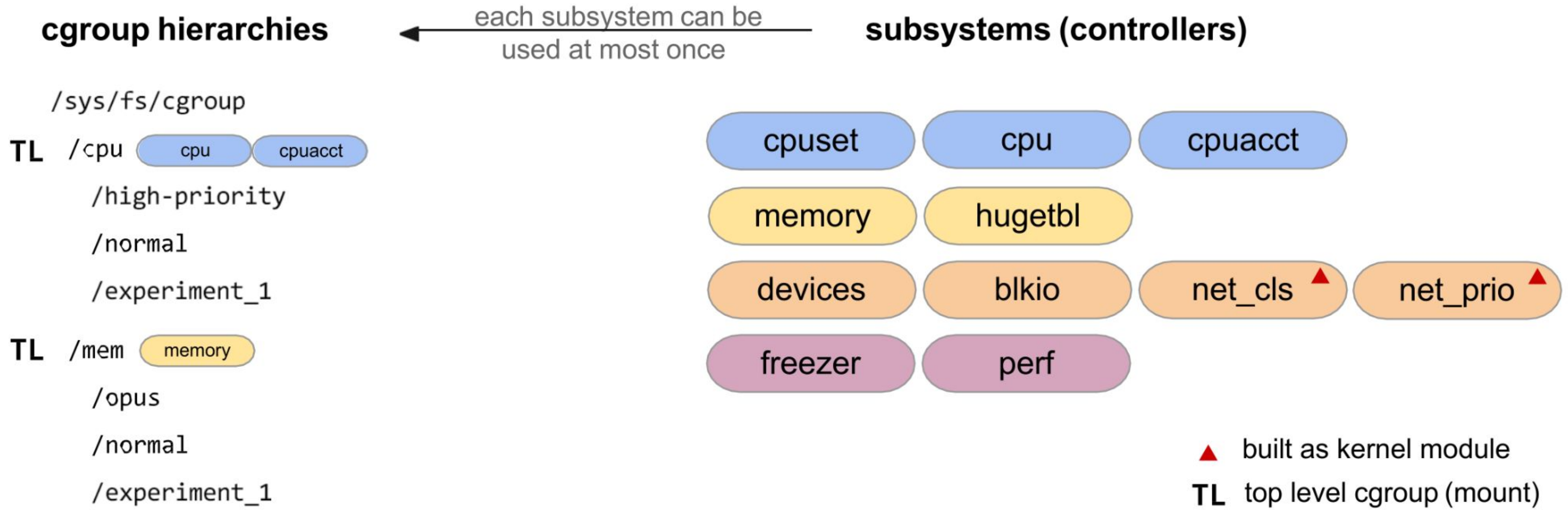
Overview



cgroups

As seen from userspace

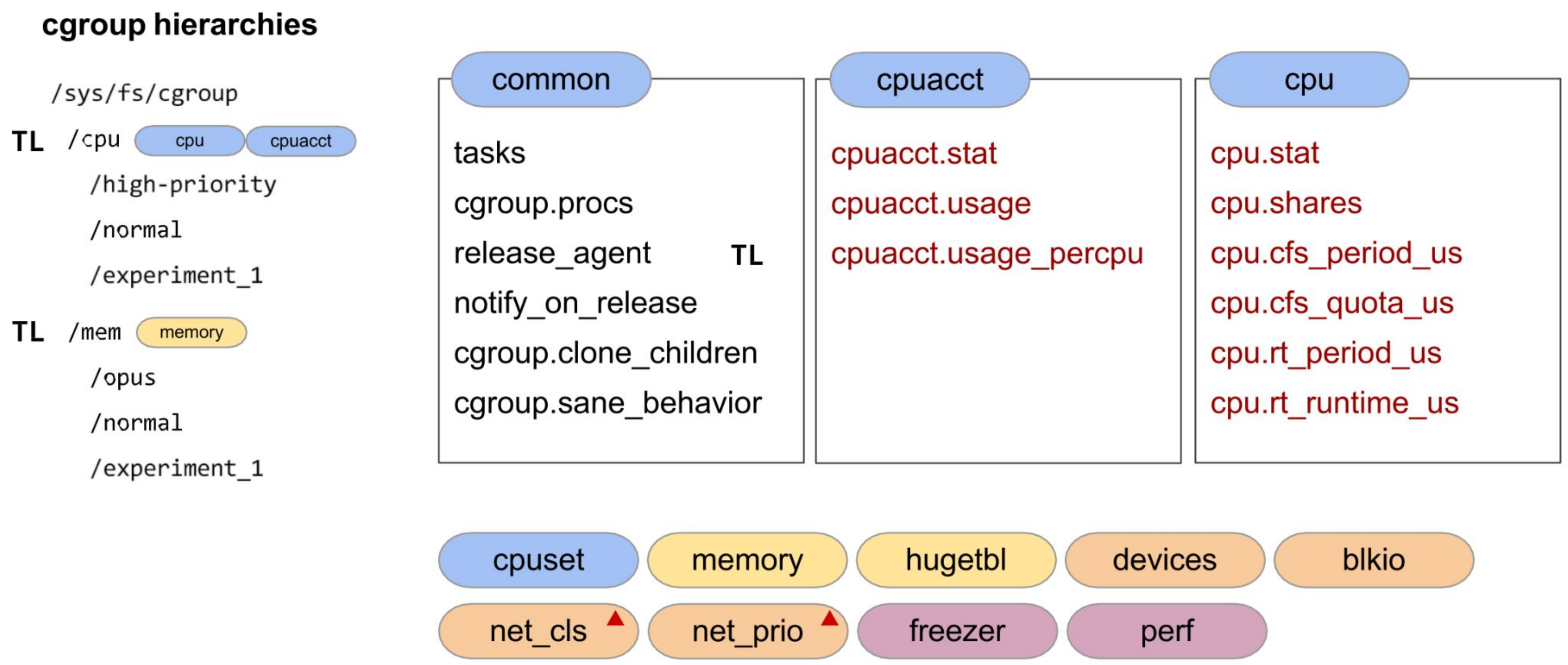
From the user space side the cgroups subsystem can be controlled and displayed by `sysfs`, but it is mounted as a pseudo filesystem, for this reason it is under /fs/, in particular in the folder `/sys/fs/cgroup`.



https://access.redhat.com/documentation/en-us/red_hat_enterprise_linux/6/html/resource_management_guide/ch01

cgroups

As seen from userspace



cgroups

As managed by the kernel

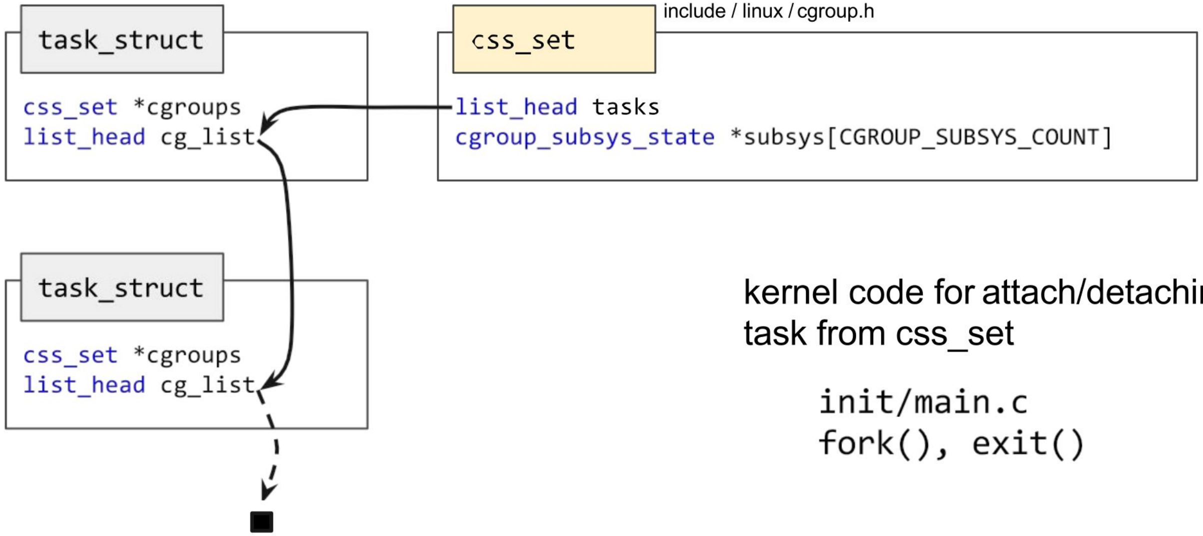
Control Groups extends the kernel as follows:

- each task in the system has a reference-counted pointer to a `css_set`.
- a `css_set` contains a set of reference-counted **pointers** to `cgroup_subsys_state` objects, one for each cgroup **subsystem** registered in the system. There is no direct link from a task to the cgroup of which it's a member in each hierarchy, but this can be determined by following pointers through the `cgroup_subsys_state` objects. This is because accessing the subsystem state is something that's expected to happen frequently and in performance-critical code, whereas operations that require a task's actual cgroup assignments (in particular, moving between cgroups) are less common. A linked list runs through the `cg_list` field of each `task_struct` using the `css_set`, anchored at `css_set->tasks`.
- a cgroup hierarchy filesystem can be mounted for browsing and manipulation from user space.

You can list all the tasks (by PID) attached to any cgroup.

cgroups

As managed by the kernel



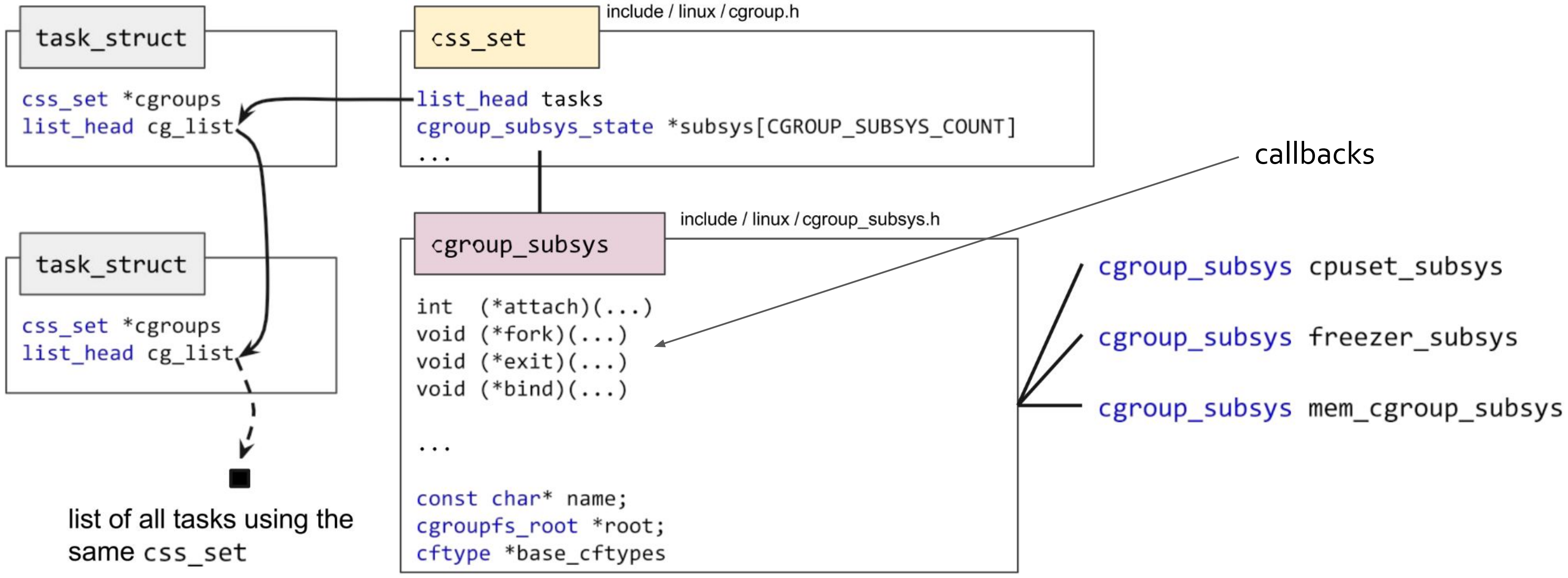
list of all tasks using the same `css_set`

kernel code for attach/detaching task from css_set

```
init/main.c
fork(), exit()
```

cgroups

As managed by the kernel



12.5.2

12. Virtualization

5. Linux Containers

namespaces

namespaces

As seen from userspace

namespaces limit the scope of kernel-level names and data structures at process granularity.

There are 8 kinds of namespaces:

- **mnt** (mount points, file systems), this namespace virtually partitions the file system and consequently processes running in separate mount namespaces cannot access files outside of their mount point
- **pid** (processes), the first processes spawn as children of PID 1, which forms the root of the process tree. The process namespace cuts off a branch of the PID tree, and doesn't allow access further up the branch. Processes in child namespaces will actually have multiple PIDs - the first one representing the global PID used by the main system, and the second PID representing the PID within the child process tree, which will restart from 1
- **net** (network stack) this namespace manages which network devices a process can see.

https://www.cloudsavvyit.com/742/what-are-linux-namespaces-and-what-are-they-used-for/

namespaces

As seen from userspace

- **`ipc`** (System V IPC) this namespace controls whether or not processes can talk directly to one another
- **`uts`** (unix timesharing) this namespace controls hostname and domain information, and allows processes to think they're running on differently named servers
- **`user`** (UIDs) this namespace allows process to have "virtual root" inside their own namespace, without having actual root access to the parent system
- **`cgroup`** controls which cgroups a process can see, and does not assign it to a specific cgroup
- **`time`** The time namespace allows processes to see different system times in a way similar to the UTS namespace

By default, any process you run uses the global namespaces, and most process on your system do as well unless otherwise specified.

https://www.cloudsavvyit.com/742/what-are-linux-namespaces-and-what-are-they-used-for/

namespaces

As seen from userspace

There are three system calls for management:

- `clone()`: create new process, new namespace, attach to namespace
- `unshare()`: create new namespace, attach current process to it
- `setns(int fd, int nstype)`: join an existing namespace

Each namespace is identified by a unique inode symlinked from in `/proc/<pid>/ns`

namespaces

As managed by the kernel

For each namespace type, a default namespace exists (the global namespace). `struct nsproxy` is shared by all tasks with the same set of namespaces.

```
task_struct

struct nsproxy *nsproxy
struct cred *cred
```

include / linux / nsproxy.h

```
nsproxy

atomic_t count
struct uts_namespace *uts_ns
struct ipc_namespace *ipc_ns
struct mnt_namespace *mnt_ns
struct pid_namespace *pid_ns_for_children
struct net           *net_ns
```

include / linux / cred.h

```
cred

...
struct user_namespace *user_ns
```

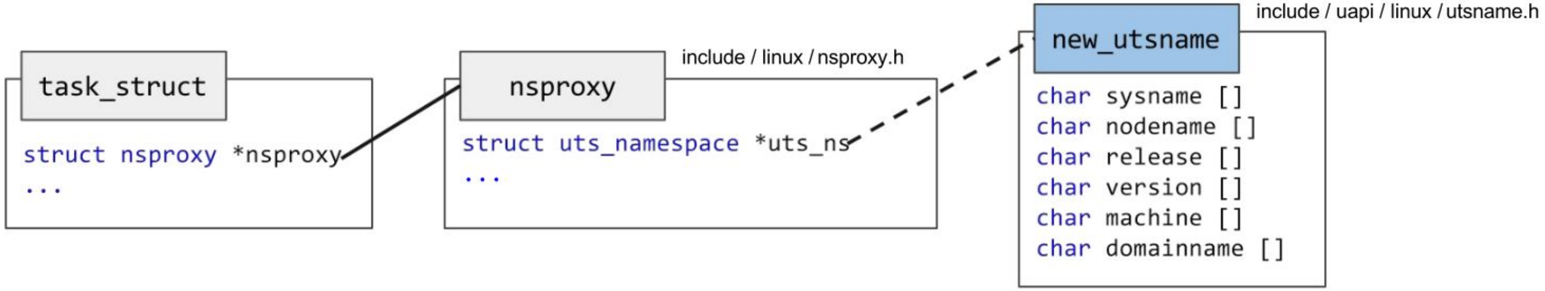
include / linux / nsproxy.h

```
nsproxy* task_nsproxy(struct task_struct *tsk)
```

namespaces

As managed by the kernel

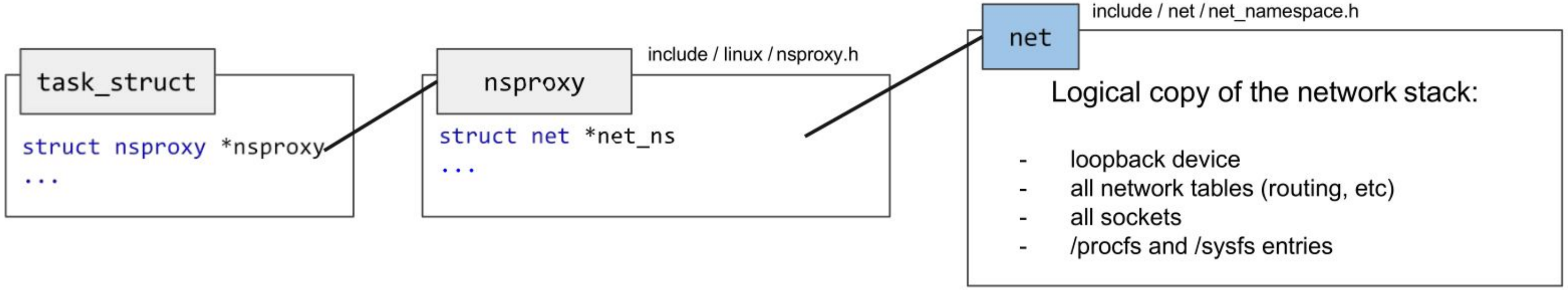
Example for the UTS namespace



namespaces

As managed by the kernel

Example for the **net** namespace


task_struct
struct nsproxy *nsproxy
...
nsproxy
include / linux / nsproxy.h
struct net *net_ns
...
net
include / net / net_namespace.h
Logical copy of the network stack:
- loopback device
- all network tables (routing, etc)
- all sockets
- /procfs and /sysfs entries

A network device belongs to exactly one namespace. A socket belongs to exactly one namespace. A new namespace only includes the loopback device. Communication between namespaces are handled via veth or unix sockets.

namespaces vs cgroups

Differently to cgroups, namespaces implements information isolation by changing what a process can **see** (fs, time, hostnames and not the resources like CPU, RAM) and not how much resources a process can use.

12.5.3

12. Virtualization

5. Linux Containers

Container Runtimes and Docker

DIAG

Containers

A light form of resource virtualization based on kernel mechanisms. Differently from VMs:

- a container is a user-space construct
- multiple containers run on top of the same kernel
- illusion that they are the only one using resources (cpu, memory, disk, network)

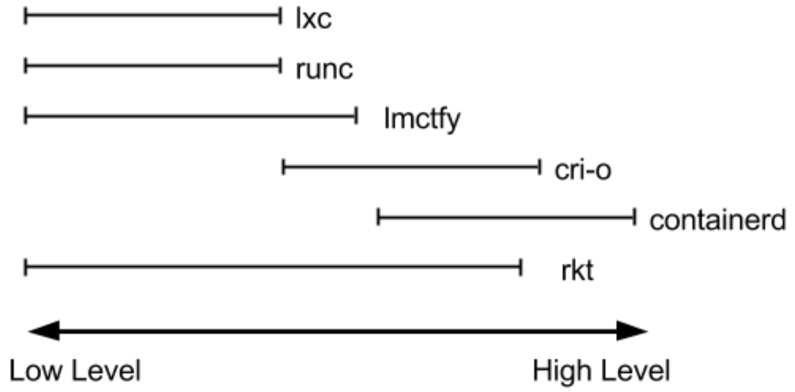
Come implementations offer support for:

- container templates
- deployment / migration
- union filesystems

Container Runtimes

We already discussed all the facilities that the kernel offers for implementing isolation between processes. A container runtime is a set of tools and binaries that implements containers by using the underlying kernel facilities.

There are many container runtimes available and they can be classified according to their capabilities. High level runtimes allows to create and manage container images for example, instead low level ones only allows for basic container facilities.



LXC

LXC is a container runtime which allows to create a container. An LXC container is a userspace process created with the clone() syscall:

- with its own pid namespace
- with its own mnt namespace
- net namespace is configurable

Container templates can be found in /usr/share/lxc/templates

For creating a container by using a template you can use the following:

```
lxc-create -t ubuntu -n containerName
```

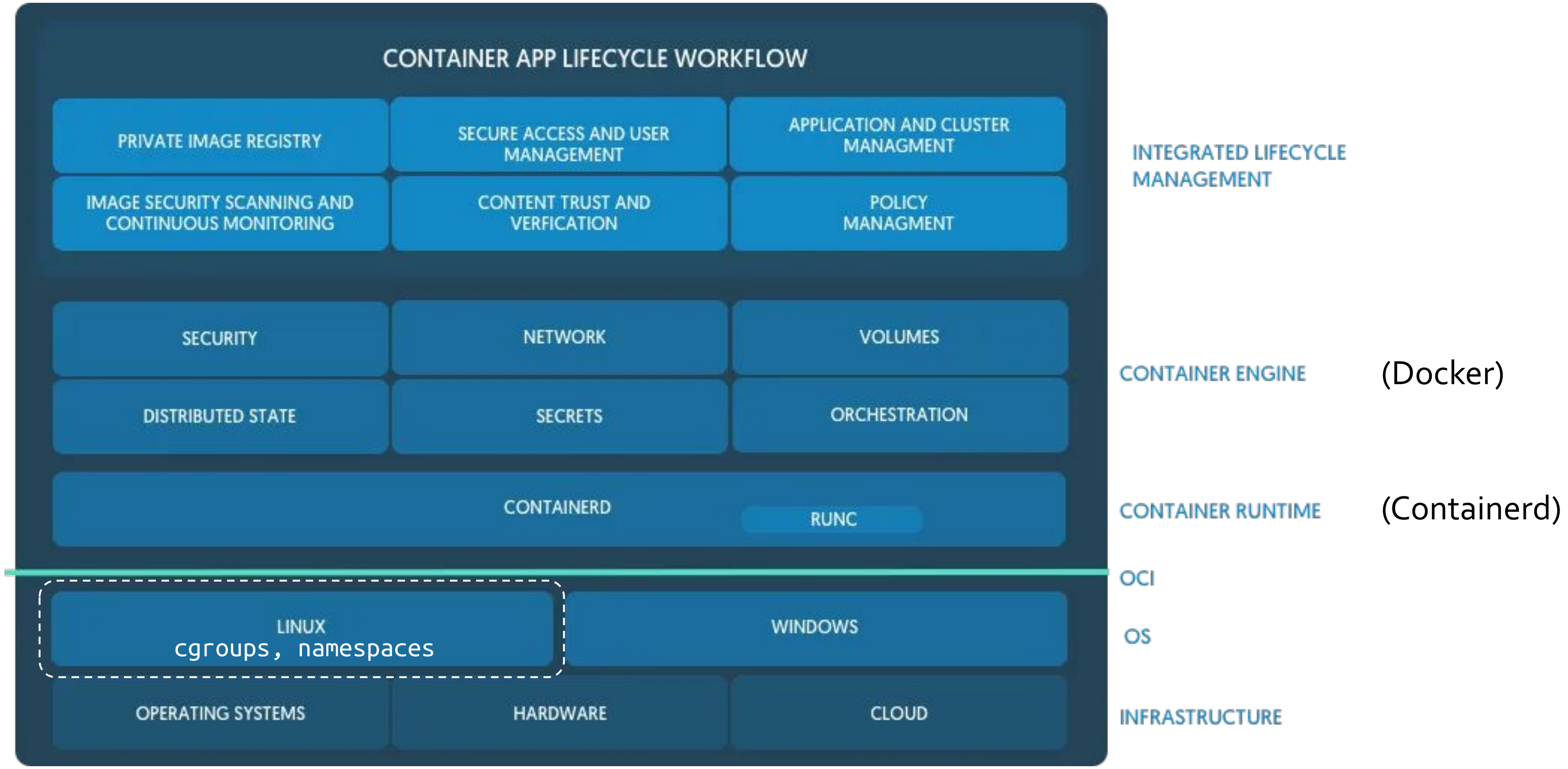
Docker

The well-known Docker is a Container Engine, not a container runtime. It allows to manage:

- Docker **Containers** that are started from images
- Docker **Images** which are read-only images which contains the filesystem of a container. Images are based on layers for storage efficiency, if multiple images shares the same layers they are only downloaded once. When containers are started they write on a new layer and then it can be merged.
- Docker **Registries** which are repositories of Docker Images

Under the hood, the modern versions of Docker relies on containerd as container runtime.

Docker



CONTAINER APP LIFECYCLE WORKFLOW
PRIVATE IMAGE REGISTRY
SECURE ACCESS AND USER MANAGEMENT
APPLICATION AND CLUSTER MANAGMENT
IMAGE SECURITY SCANNING AND CONTINUOUS MONITORING
CONTENT TRUST AND VERFICATION
POLICY MANAGMENT
INTEGRATED LIFECYCLE MANAGEMENT
SECURITY
NETWORK
VOLUMES
DISTRIBUTED STATE
SECRETS
ORCHESTRATION
CONTAINER ENGINE
(Docker)
CONTAINERD
RUNC
CONTAINER RUNTIME
(Containerd)
OCI
LINUX
cgroups, namespaces
WINDOWS
OS
OPERATING SYSTEMS
HARDWARE
CLOUD
INFRASTRUCTURE

Advanced Operating Systems and Virtualization

[12] Virtualization

LECTURER

Gabriele **Proietti Mattia**

BASED ON WORK BY

http://www.ce.uniroma2.it/~pellegrini/

gpm.name · proiettimattia@diag.uniroma1.it

DIAG