

# Virtualization Support

Advanced Operating Systems and Virtualization

Alessandro Pellegrini

A.Y. 2018/2019



SAPIENZA  
UNIVERSITÀ DI ROMA

# System Virtualization

- Virtualization allows to show resources different 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 is the Hypervisor or VMM (Virtual Machine Monitor).
- Advantages:
  - Isolation of different execution environments (on the same hardware)
  - Reduction of hardware and administration costs



# Hypervisor

- *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
- Hypervisor:
  - It manages hardware resources provided by the *host system*
  - It makes virtualized resources available to the *guest system* in a correct and secure way
  - *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



# Software-based Virtualization

- Instructions are executed by the native physical CPU in the host platform
- A subset of the instruction set must be emulated
- No particular hardware component plays a role in virtualization
- *The main problems:*
  - What if ring 0 is required for guest activities?
  - Risk to bypass the VMM resource management policy in case of actual ring 0 access
- *The solution:* ring deprivileging



# Ring Deprivileging

- A technique to let the guest kernel run at a privilege level that “simulates” 0
- Two main strategies:
  1. 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
  2. 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



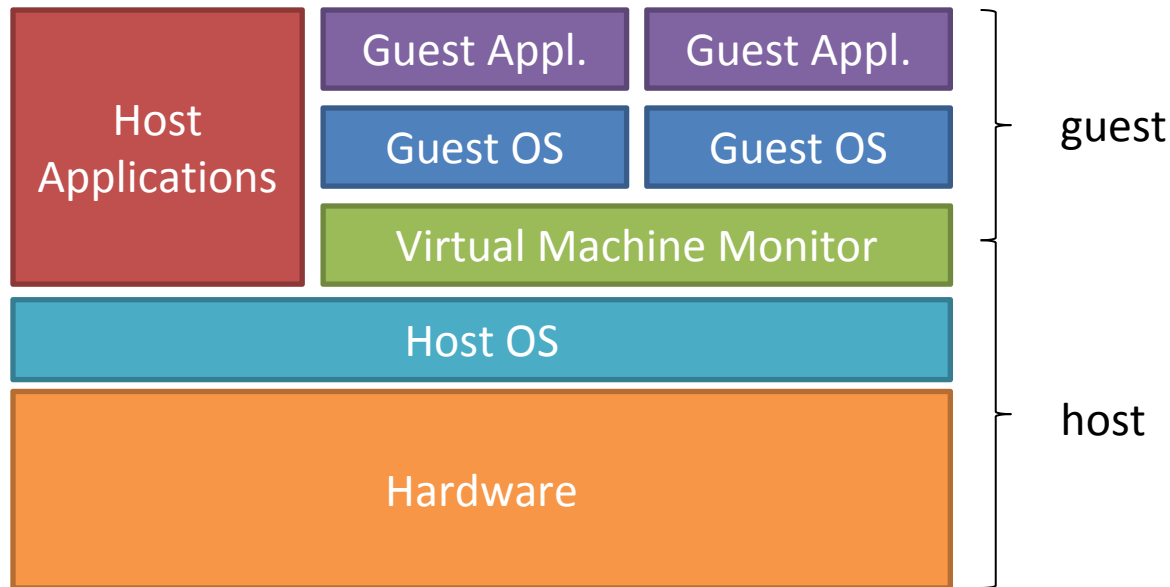
# Ring Aliasing

- An OS kernel is designed to run at ring 0, while it is actually being run at ring 1 for guest systems
- Privileged instructions generate an exception if not run at CPL 0:
  - Some examples: `hlt`, `lidt`, `lgdt`, `invd`, `mov %crx`
- *I/O sensitive instructions*: they generate a trap if executed when  $CPL > IOPL$  (I/O Privilege Level). Classical examples are:
  - `cli`, `sti`
- The generated trap (*general protection fault*) must be handled by the VMM, so as to finally determine how to handle it (emulation vs interpretation)



# The VirtualBox 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

- 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: 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 upon a GPF



# VBOXGDT

- Introduction of gate descriptors for kernel code/data segments with DPL=1. These segments are accessible with CPL=1
- New TSSD pointing to the TSS wrapper which keeps info on stack positioning at ring 1 (ss1,esp1) and ring 0 (ss0,esp0).
- 2 new segments for the Hypervisor are added with DPL=0

DESCRIPTION	OFFSET	DPL	BASE
Entry 0	(0000) <sub>H</sub>	-	null
...	...	...	...
KERNEL CODE SEGMENT	(0060) <sub>H</sub>	1	
KERNEL DATA SEGMENT	(0068) <sub>H</sub>	1	
...	...	...	...
VIRTUALBOX TSSD	(FFE0) <sub>H</sub>	0	
...	...	...	...
HYPERVISOR DATA SEGMENT	(FFF0) <sub>H</sub>	0	
HYPERVISOR CODE SEGMENT	(FFF8) <sub>H</sub>	0	

ORIGINAL TSS

...	...
esp0	...
ss0	(0068) <sub>H</sub>
esp1	unused
ss1	unused
...	...

VBOXTSS

...	...
esp0	(FE557000) <sub>H</sub>
Ss0	(FFF0) <sub>H</sub>
esp1	(F70D3FF8) <sub>H</sub>
ss1	(0069) <sub>H</sub>
...	...

ss1=ss0 | 1

CPL = Current Privilege Level  
DPL = Descriptor Privilege Level



# 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 (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

ORIGINAL IDT

0xD	14	(0060) <sub>H</sub>	0	
0xE	14	(0060) <sub>H</sub>	0	
...				

VBOXIDT

0x0				
...				
0xD	14	(FFF8) <sub>H</sub>	0	
0xE	14	(FFF8) <sub>H</sub>	0	
...				

Genuine trap

VMM handler

Virtual trap



# 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
- 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

ORIGINAL IDT

...				
0x80	15	(0060) <sub>H</sub>	3	
...				

VBOXIDT

...				
0x80	15	(0061) <sub>H</sub>	3	
...				

system\_call  
handler

Ring 1 handler

Handler  
trampoline

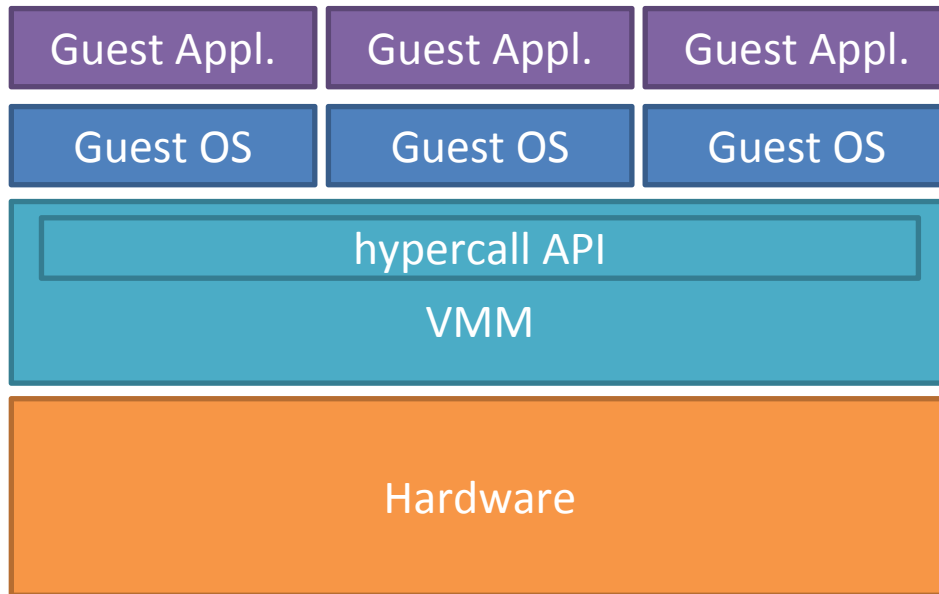


# Paravirtualization

- The VMM offers a virtual interface (*hypercall API*) used by guest OS to access resources
  - To run privileged instructions, hypercalls are executed
  - There is a need to modify the code of the guest OS
  - VMM is simplified: no need to account for traps generated by virtualized OS
- An example: Xen



# Paravirtualization



# Hardware-Assisted Virtualization: VT-x

- Intel *Vanderpool Technology*, referred to as VT-x, represents Intel's virtualization technology on the x86 platform.
- Its goal: 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 3 are not trapped, for example `popf`)
  - Excessive trapping
- Eliminate need for software virtualization (i.e paravirtualization, binary translation).



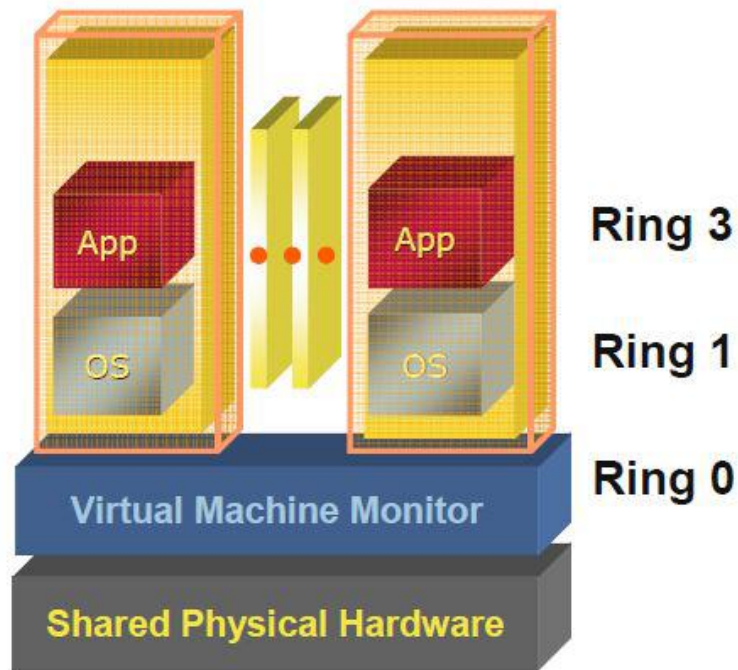
# Virtual Machine Extension (VMX)

- Virtual Machine Extensions define CPU support for VMs on x86 by a new form of operation called *VMX operation*
- Kinds of VMX operation:
  - **root:** VMM runs in VMX root operation
  - **non-root:** Guest runs in VMX non-root operation
- Eliminate ring deprivileging for guest OS
- VMX Transitions between VMX root operation and VMX non-root operation:
  - **VM Entry:** Transitions into VMX non-root operation.
  - **VM Exit:** Transitions from VMX non-root operation to VMX root operation.
  - Registers and address space swapped in one atomic operation.

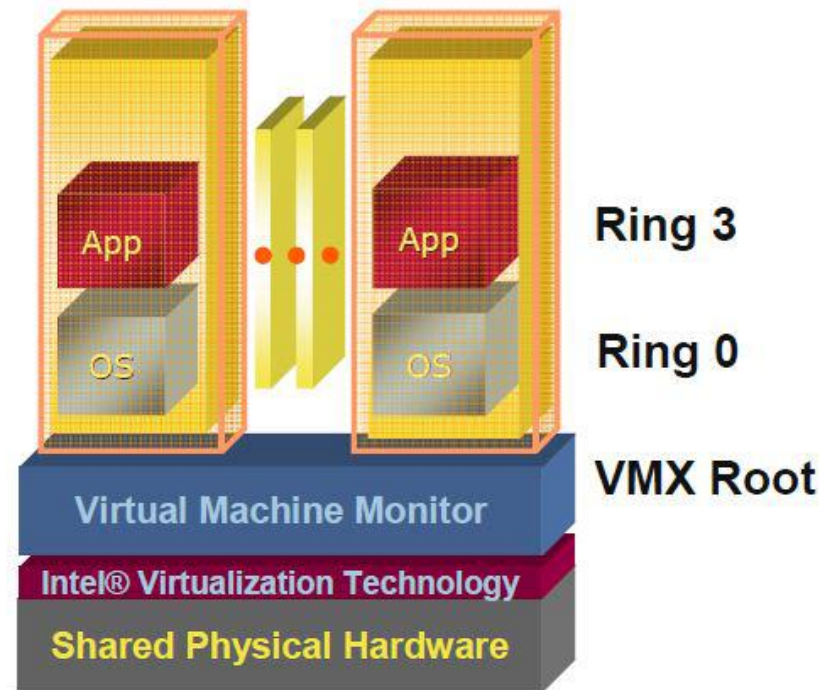




# Virtual Machine Extension (VMX)



**Pre VT-x**



**Post VT-x**

VMM ring deprivileging of guest OS	VMM executes in VMX root-mode
Guest OS aware its not at Ring 0	Guest OS deprivileging eliminated
	Guest OS runs directly on hardware



# VMCS: VM Control Structure

- Data structure to manage VMX non-root operation and VMX transitions
  - Specifies guest OS state
  - Configured by VMM
  - Controls when VM exits occur

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.



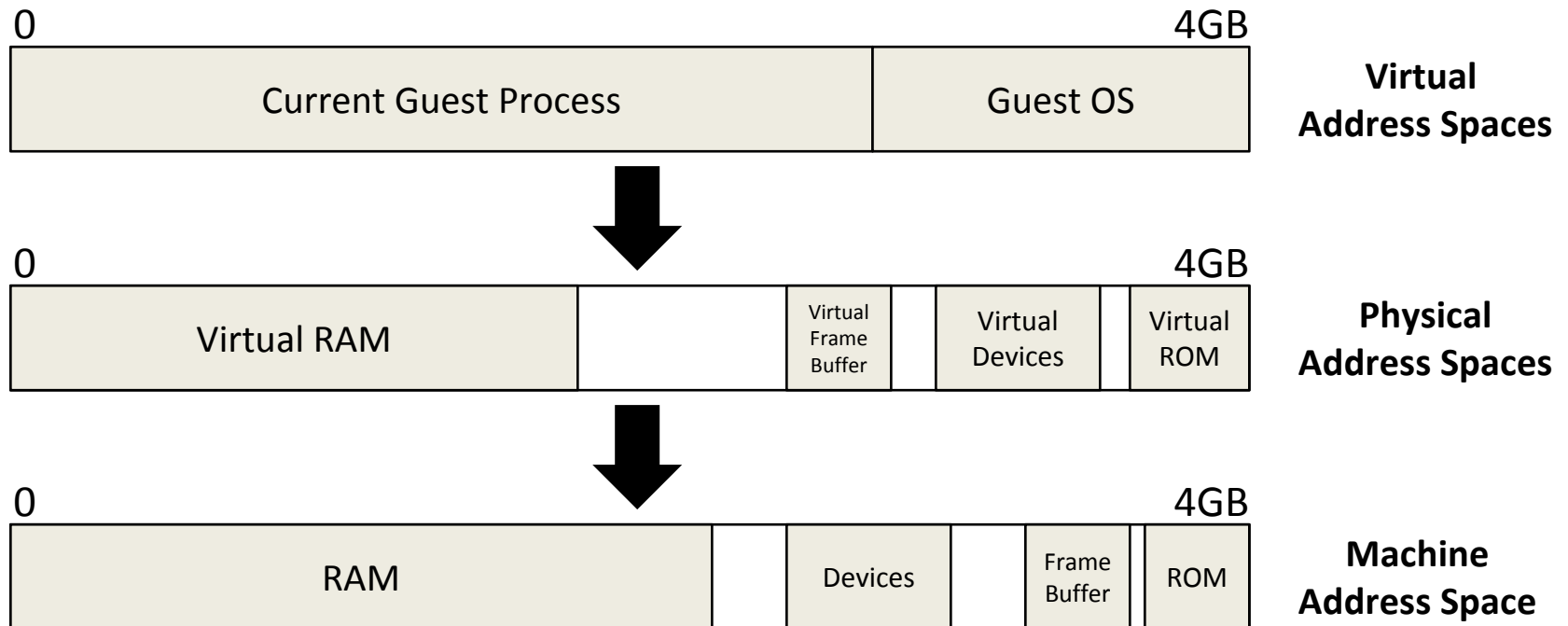
# MMU Virtualization with VT-x: VPIDs

- First generation VT-x forces TLB flush on each VMX transition
- Performance loss on all VM exits
- Performance loss on most VM entries
  - Guest page tables not modified always
- Better VMM software control of TLB flushes is beneficial
- VPID:
  - 16-bit virtual-processor-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



# Virtualizing Memory in Software

- Three abstractions of memory:

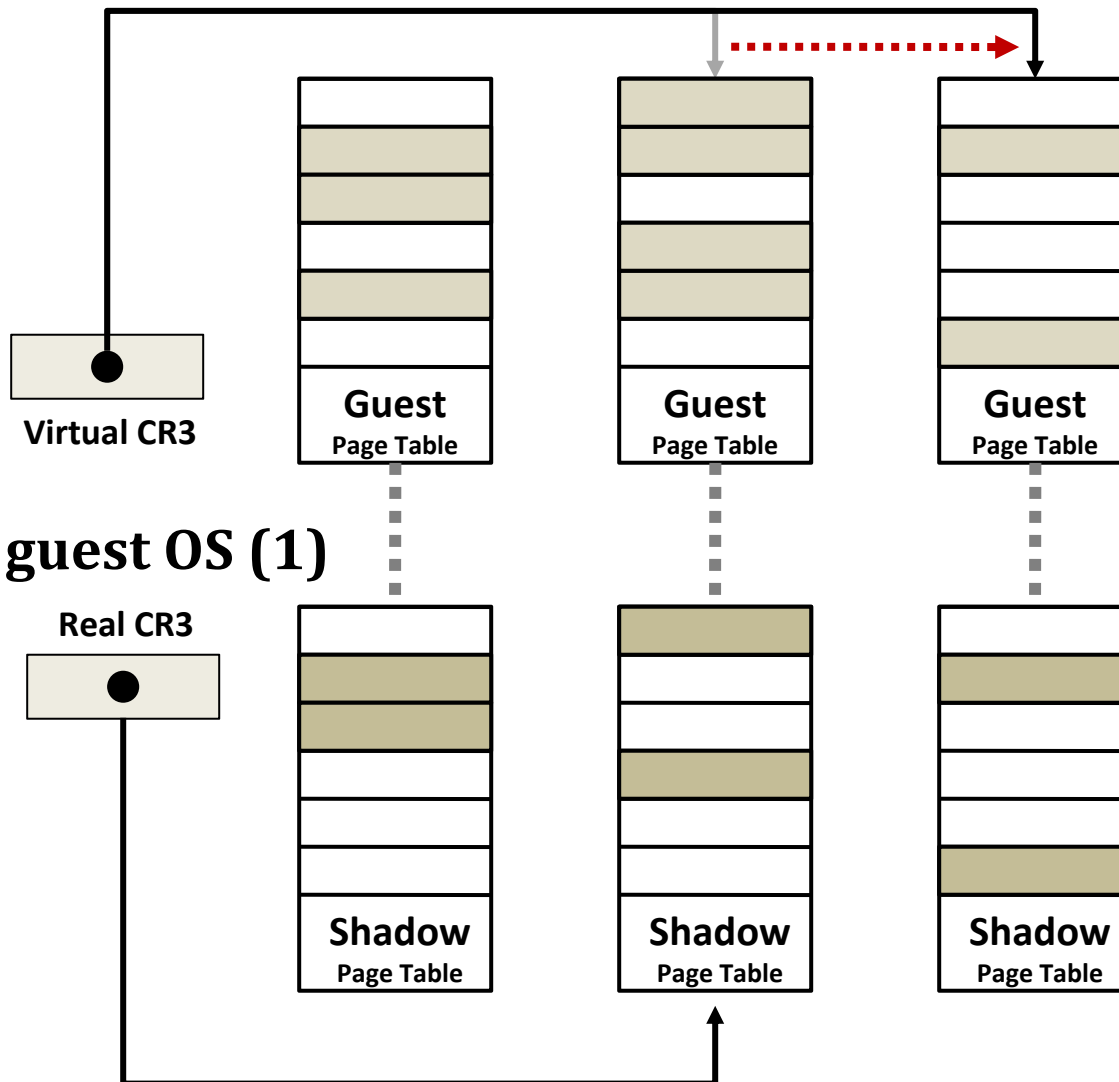


# Shadow Page Tables

- VMM maintains shadow page tables that map guest-virtual pages directly to machine pages
- Guest modifications to  $V \rightarrow P$  tables synced to VMM  $V \rightarrow M$  shadow page tables
  - Guest OS page tables marked as read-only
  - Modifications of page tables by guest OS  $\rightarrow$  trapped to VMM
  - Shadow page tables synced to the guest OS tables



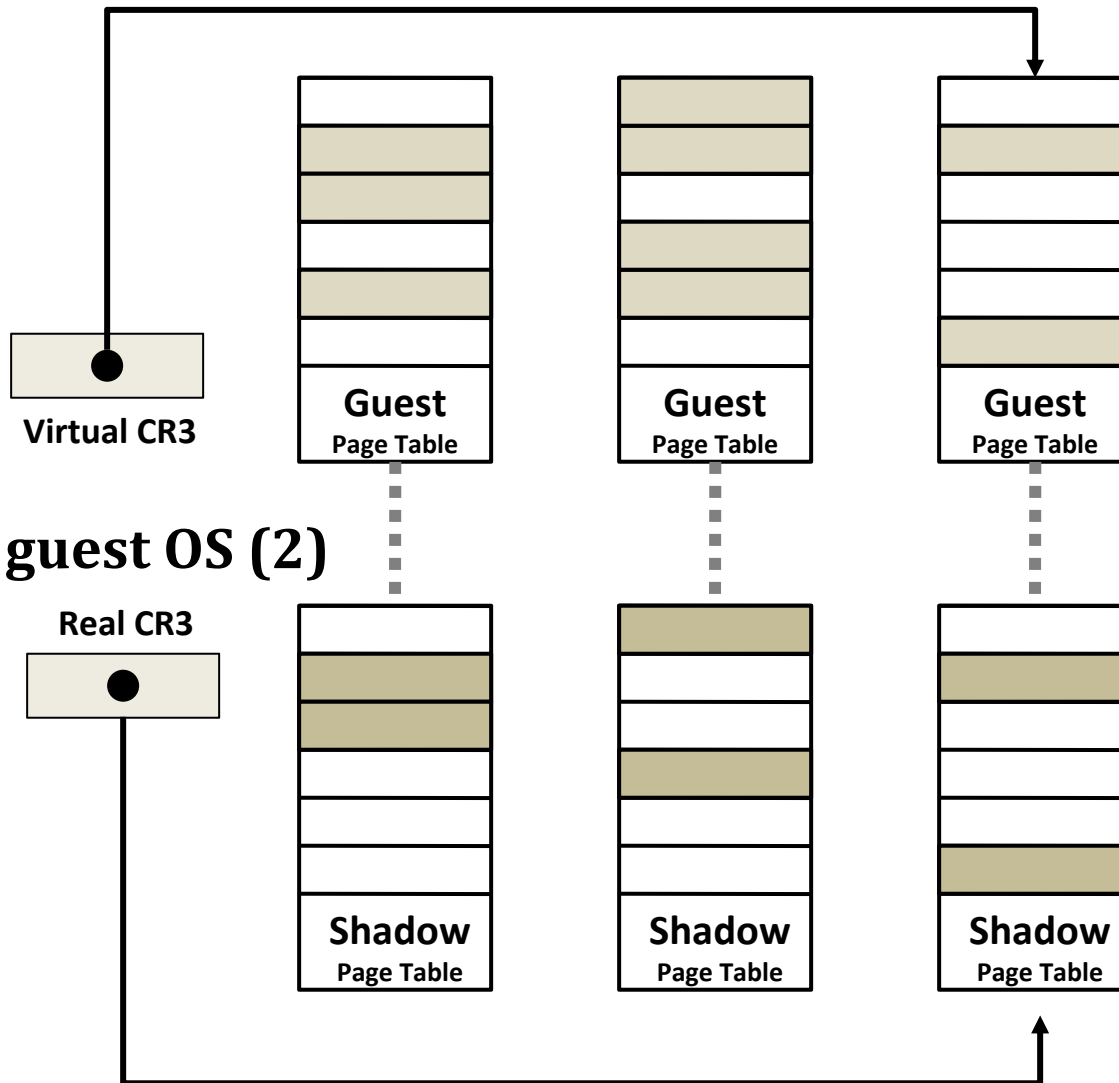
# Shadow Page Tables



**Set CR3 by guest OS (1)**



# Shadow Page Tables



Set CR3 by guest OS (2)



# Shadow Page Tables: Drawbacks

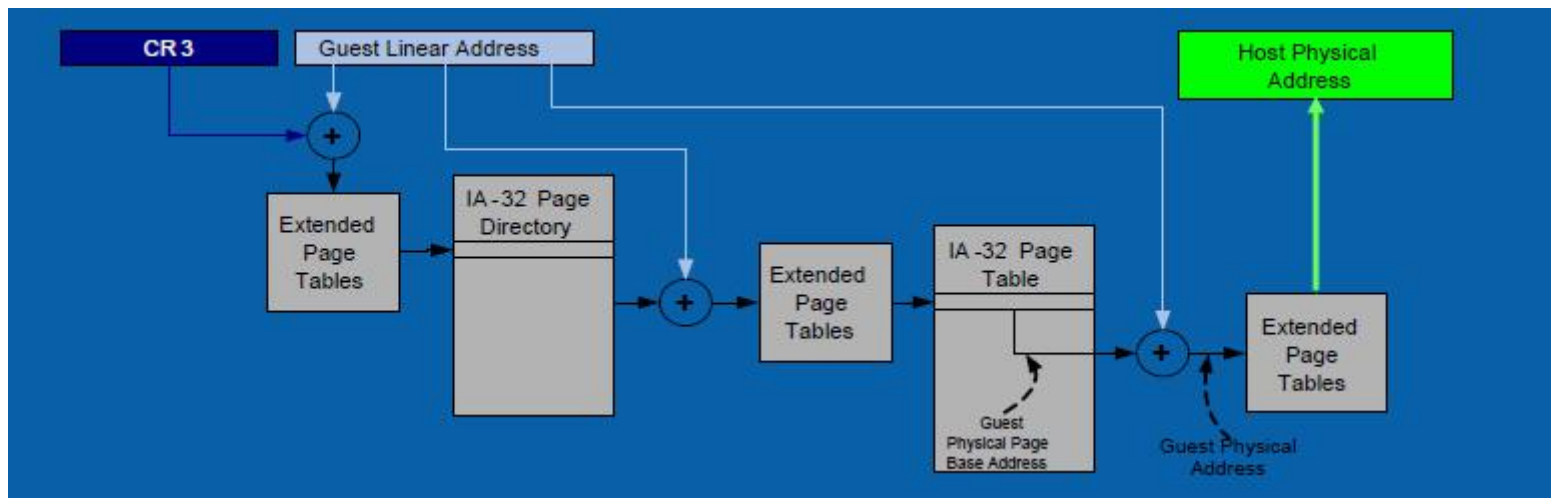
- Maintaining consistency between guest page tables and shadow page tables leads to an overhead: VMM traps
- Loss of performance due to TLB flush on every “world-switch”
- Memory overhead due to shadow copying of guest page tables



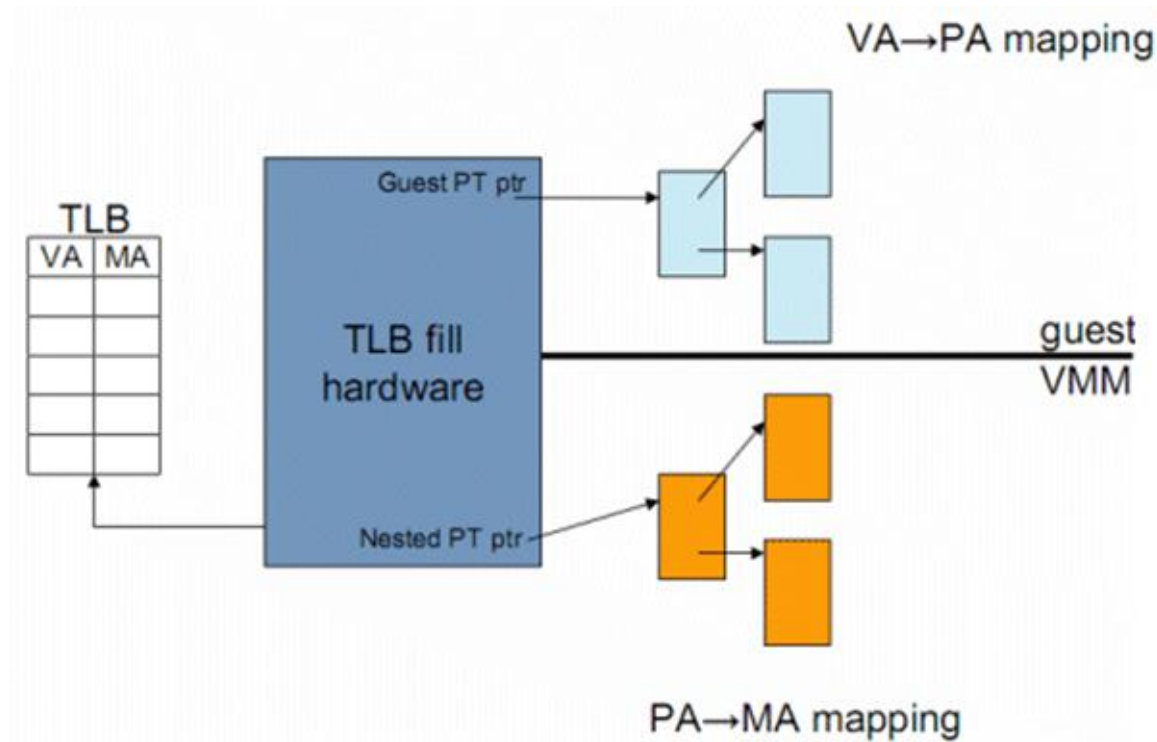


# Nested / Extended Page Tables

- The Extended Page-Table mechanism (EPT) is used to support the virtualization of physical memory
- 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

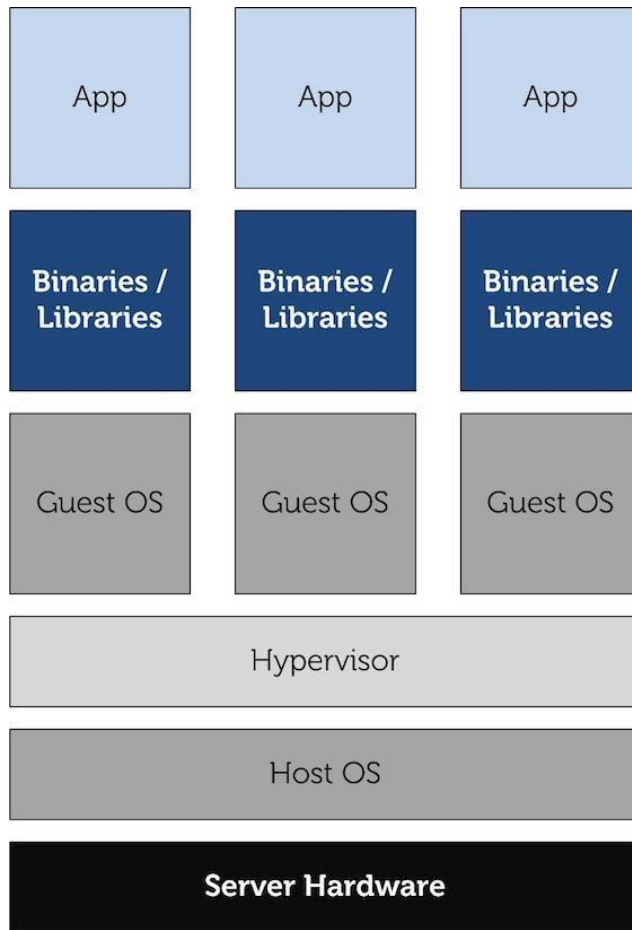


# Considerations on EPT

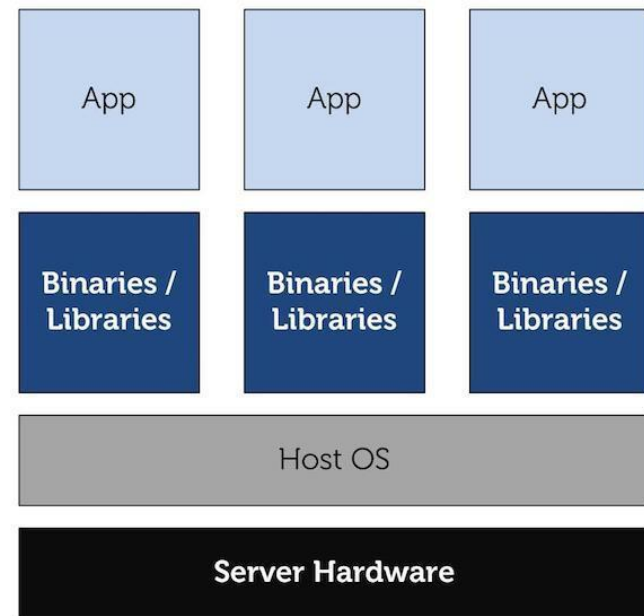
- Advantages:
  - Simplified VMM design
  - Guest page table modifications need not to be trapped, hence VM exits reduced
  - Reduced memory footprint compared to shadow page table algorithms
- Disadvantages:
  - TLB miss is very costly since guest-physical address to machine address needs an extra EPT walk for each stage of guest-virtual address translation



# Linux Containers



Virtualization



Containers



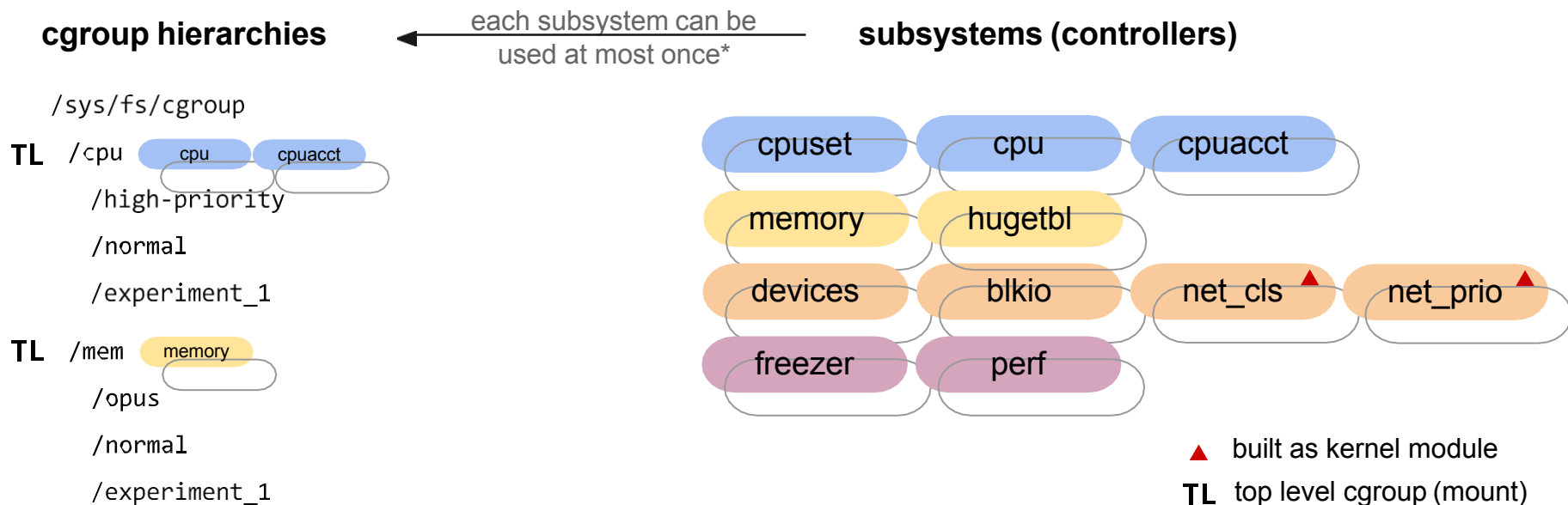
# Underlying Kernel Mechanisms

- 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 available to processes*



# cgroups (as seen from userspace)

- low-level filesystem interface similar to sysfs and procfs
- new filesystem type “cgroup”, default location in /sys/fs/cgroup



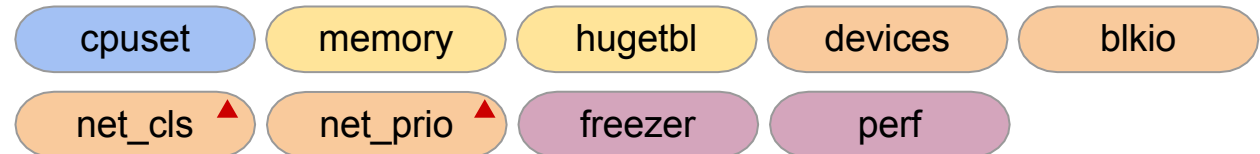
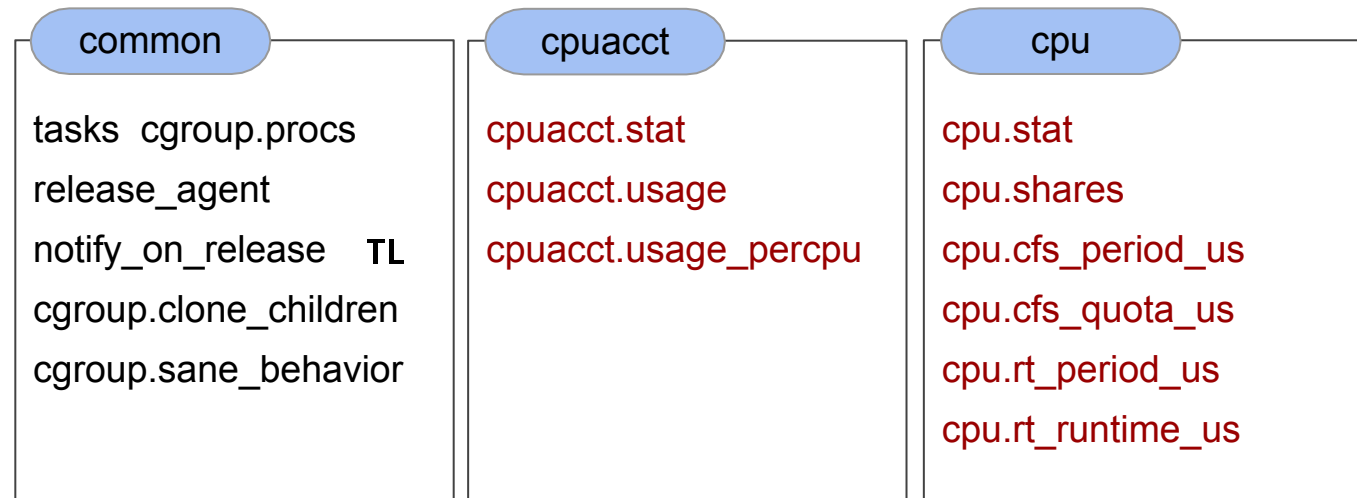
# cgroups (as seen from userspace)

## cgroup hierarchies

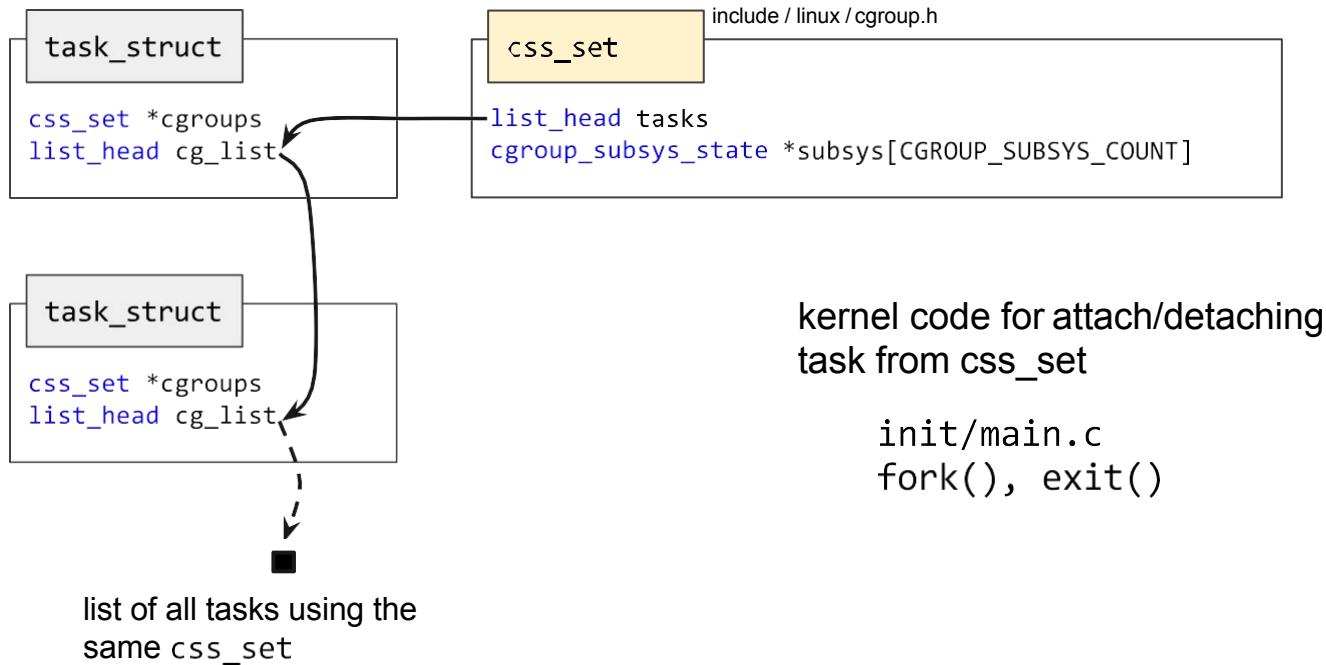
/sys/fs/cgroup

**TL** /cpu **cpu** **cpuacct**  
    /high-priority  
    /normal  
    /experiment\_1

**TL** /mem **memory**  
    /opus  
    /normal  
    /experiment\_1

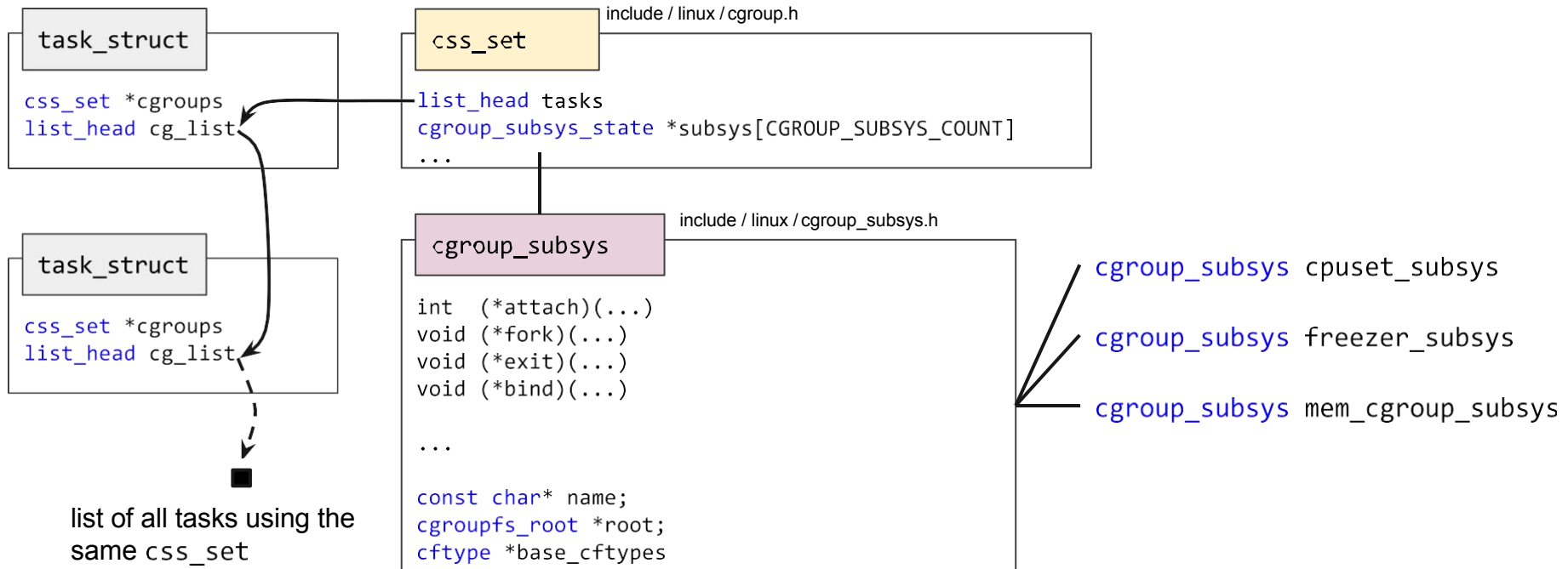


# cgroups (as managed by kernel)





# cgroups (as managed by kernel)



# cgroups (as managed by kernel)

## cgroup\_subsys

include / linux / cgroup\_subsys.h

```
int (*attach)(...)  
void (*fork)(...)  
void (*exit)(...)  
void (*bind)(...)  
  
...  
  
const char* name;  
cgroupfs_root *root;  
cftype *base_cftypes
```

```
cgroup_subsys cpuset_subsys  
    .base_cftypes = files
```

```
1819 static struct cftype files[] = {  
1820     --{  
1821     ---.name = "cpus",  
1822     ---.seq_show = cpuset_common_seq_show,  
1823     ---.write_string = cpuset_write_resmask,  
1824     ---.max_write_len = (100U + 6 * NR_CPUS),  
1825     ---.private = FILE_CPULIST,  
1826     --},  
1827  
1828     --{  
1829     ---.name = "mems",  
1830     ---.seq_show = cpuset_common_seq_show,  
1831     ---.write_string = cpuset_write_resmask,  
1832     ---.max_write_len = (100U + 6 * MAX_NUMNODES),  
1833     ---.private = FILE_MEMLIST,  
1834     --},  
1835  
1836     --{  
1837     ---.name = "cpu_exclusive",  
1838     ---.read_u64 = cpuset_read_u64,  
1839     ---.write_u64 = cpuset_write_u64,  
1840     ---.private = FILE_CPU_EXCLUSIVE,  
1841     --},  
1842  
1843     --{  
1844     ---.name = "mem_exclusive",  
1845     ---.read_u64 = cpuset_read_u64,  
1846     ---.write_u64 = cpuset_write_u64,  
1847     ---.private = FILE_MEM_EXCLUSIVE,  
1848     --},
```



# namespaces (as seen from userspace)

- namespaces limit the scope of kernel-level *names* and *data structures* at process granularity
- Some examples:
  - mnt (mount points, file systems) `CLONE_NEWNS`
  - pid (processes) `CLONE_NEWPID`
  - net (network stack) `CLONE_NEWNET`
  - ipc (System V IPC) `CLONE_NEWIPC`
  - uts (unix timesharing) `CLONE_NEWUTS`
  - user (UIDs) `CLONE_NEWUSER`



# 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
  - symbolic links in `/proc/<pid>/ns`



# namespaces (as managed by 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
```

**nsproxy** include / linux / nsproxy.h

```
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 / nsproxy.h

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

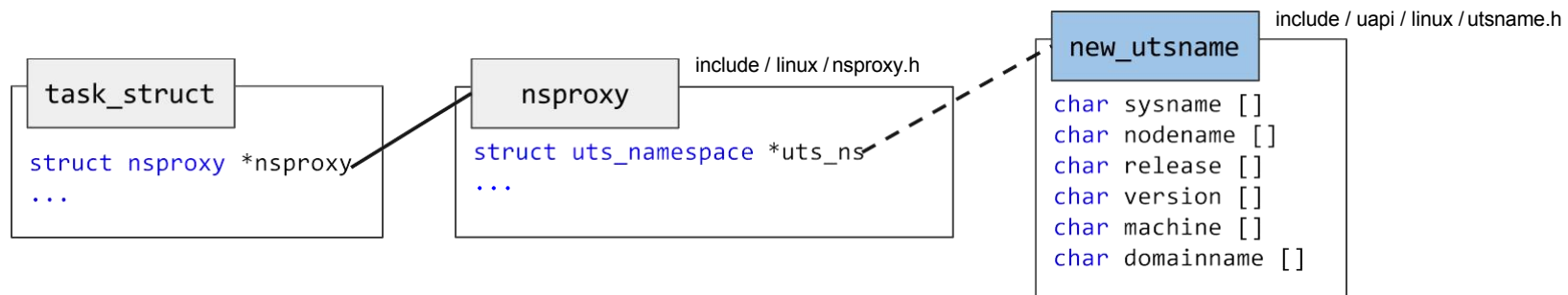
**cred** include / linux / cred.h

```
...
struct user_namespace *user_ns
```



# namespaces (as managed by kernel)

- Example for the UTS namespace

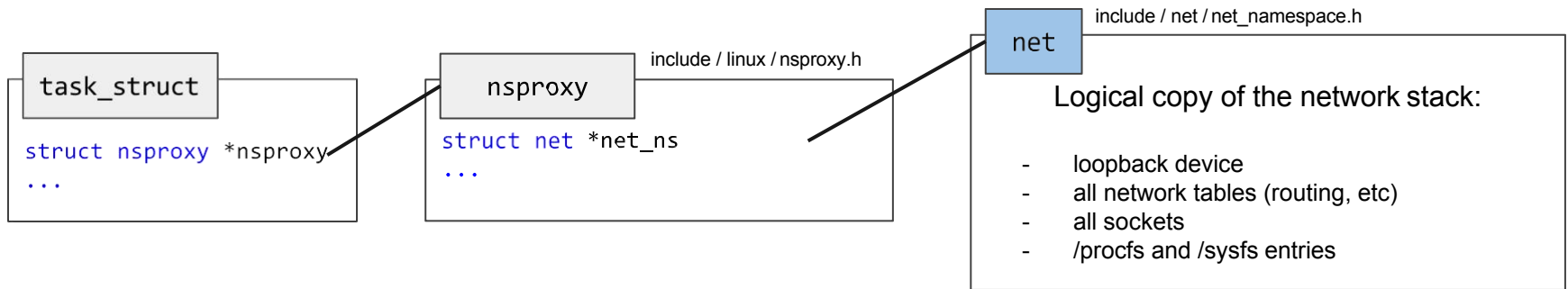


- Global access to hostname: `system_utsname.nodename`
- Namespace-aware access to hostname:  
`&current->nsproxy->uts_ns->name->nodename`



# namespaces (as managed by kernel)

- Example for the `net` namespace

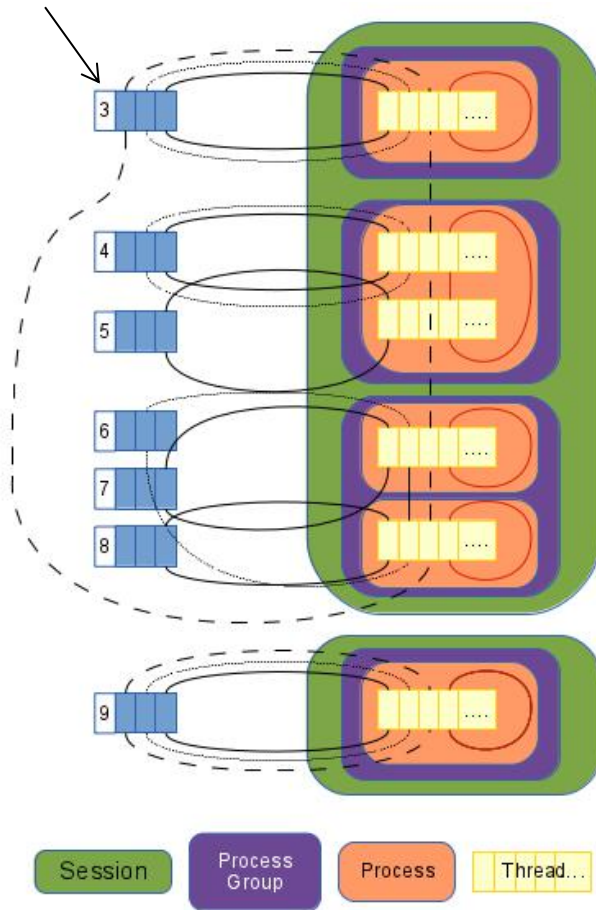


- A network device belongs to exactly one namespace
- A socket belongs to exactly one namespace
- A new namespace only includes the loopback device
- Communications between namespaces are handled via veth or unix sockets



# pids and namespaces

struct pid

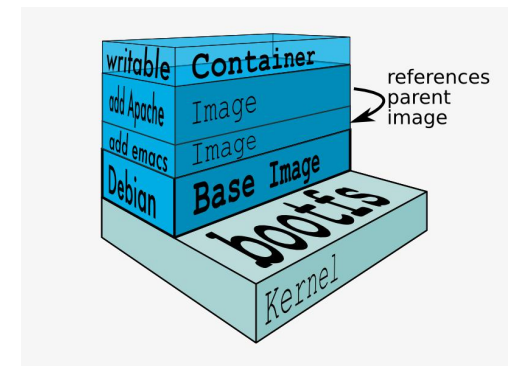


- `struct pid` links together pids in the namespace world



# Containers

- A light form of resource virtualization based on kernel mechanisms
- 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)
- some implementations offer support for:
  - container templates
  - deployment / migration
  - union filesystems



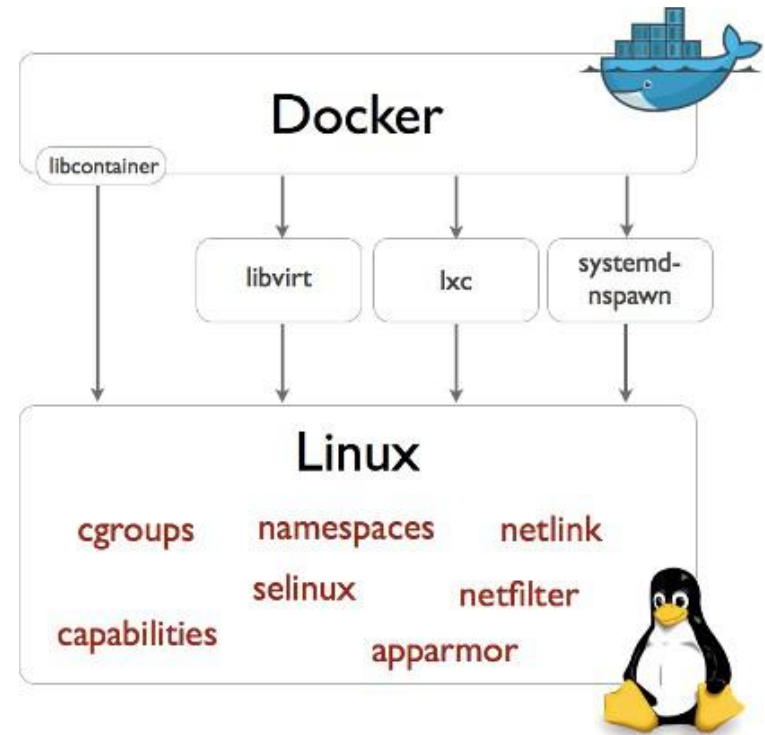
# Container Solutions: LXC

- 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`
- Shell scripts:
  - `lxc-create -t ubuntu -n containerName`



# Container Solutions: Docker

- A Linux container engine
- Multiple backend drivers
- Application-centric
- Diff-based deployment of updates (AUFS)
- Links (tunnels) between containers



# Kernel Samepage Merging

- COW is used by the kernel to share physical frames with different virtual mappings
- If the kernel has no knowledge on the usage of memory, a similar behaviour is difficult to put in place
- KSM exposes the `/dev/ksm` pseudofile
- By means of `ioctl()` calls, programs can register portions of their address spaces
- An additional `ioctl()` call enables the page sharing mechanism, and the kernel starts looking for pages to share



# Kernel Samepage Merging

- The KSM driver (in a kernel thread) picks one registered region and starts scanning it
  - A SHA1 hash is used to compare frames
  - If a similarity is found, all processes "sharing" the page will point to the same frame (in COW mode)
- A host running several guest Windows machines can overcommit its memory 300% without affecting performance
  - Windows zeroes all free'd memory

