Lecture 4 : Virtualization Techniques

Dr. Bibhas Ghoshal

Assistant Professor

Department of Information Technology

Indian Institute of Information Technology Allahabad

Some Slides Used in this Lecture have been adapted from slides of Professor Mythilli Vutukuru, Dept. Of CSE, IIT Bombay delivered for the course Virtualization and Cloud Computing

Hardware Assisted Virtualization

- Original x86 CPUs did not support virtualization

- Modern technique, after hardware support for virtualization introduced in CPUs - Intel VT-X or AMD-Vsupport is widely available in modern systems

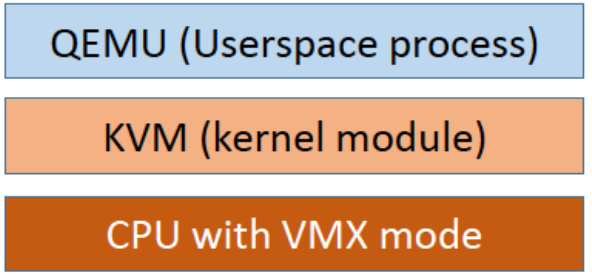
- Special CPU mode of operation called VMX mode for running VMs
- Many hypervisors use this H/W feature - QEMU/KVM in Linux

Works with binary translation if no hardware support

Sets up guest VM memory as part of userspace process

When invoked, KVM switches to VMX mode to run guest

CPU switches between VMX and non-VMX root modes

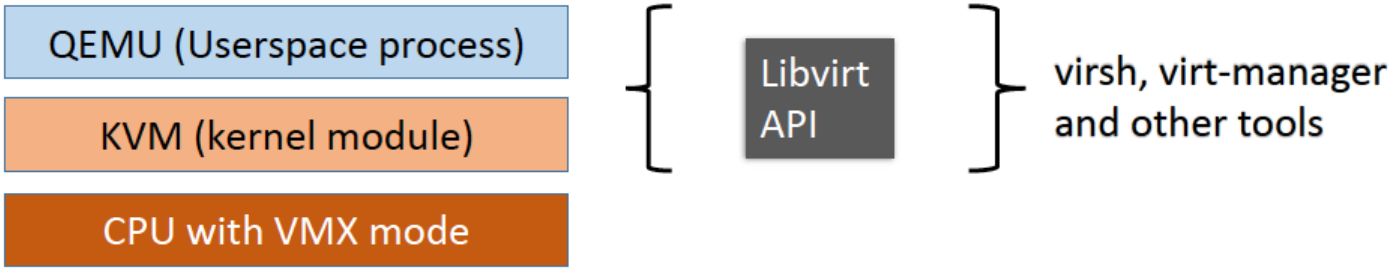


Slide Author : Mythilli Vutukuru

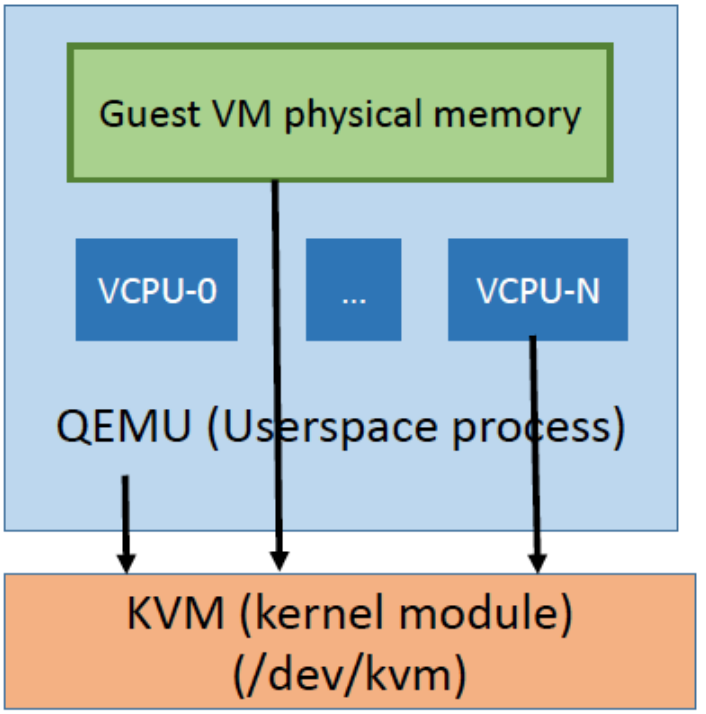
Hardware Assisted Virtualization

- Install QEMU/KVM on Linux => libvirtis is installed

- A set of tools manage hypervisors, including QEMU/KVM
- A daemon runs on the system and communicates with hypervisors
- Exposes an API using which hypervisors can be managed, VM created etc.
- Command linetool (virsh) and GUI (virt-manager) use this API to manage VMs



QEMU Architecture



- QEMU is userspace process
- KVM exposes a dummy device
- QEMU talks to KVM via open/ioctlsyscalls
- Allocates memory via mmapfor guest VM physical memory
- Creates one thread for each virtual CPU (VCPU) in guest
- Multiple file descriptors to /dev/kvm

(one for QEMU, one for VM, one for VCPU and so on)

- ioctlon fdsto talk to KVM
- Host OS sees QEMU as a regular multi-threaded process

QEMU Operation

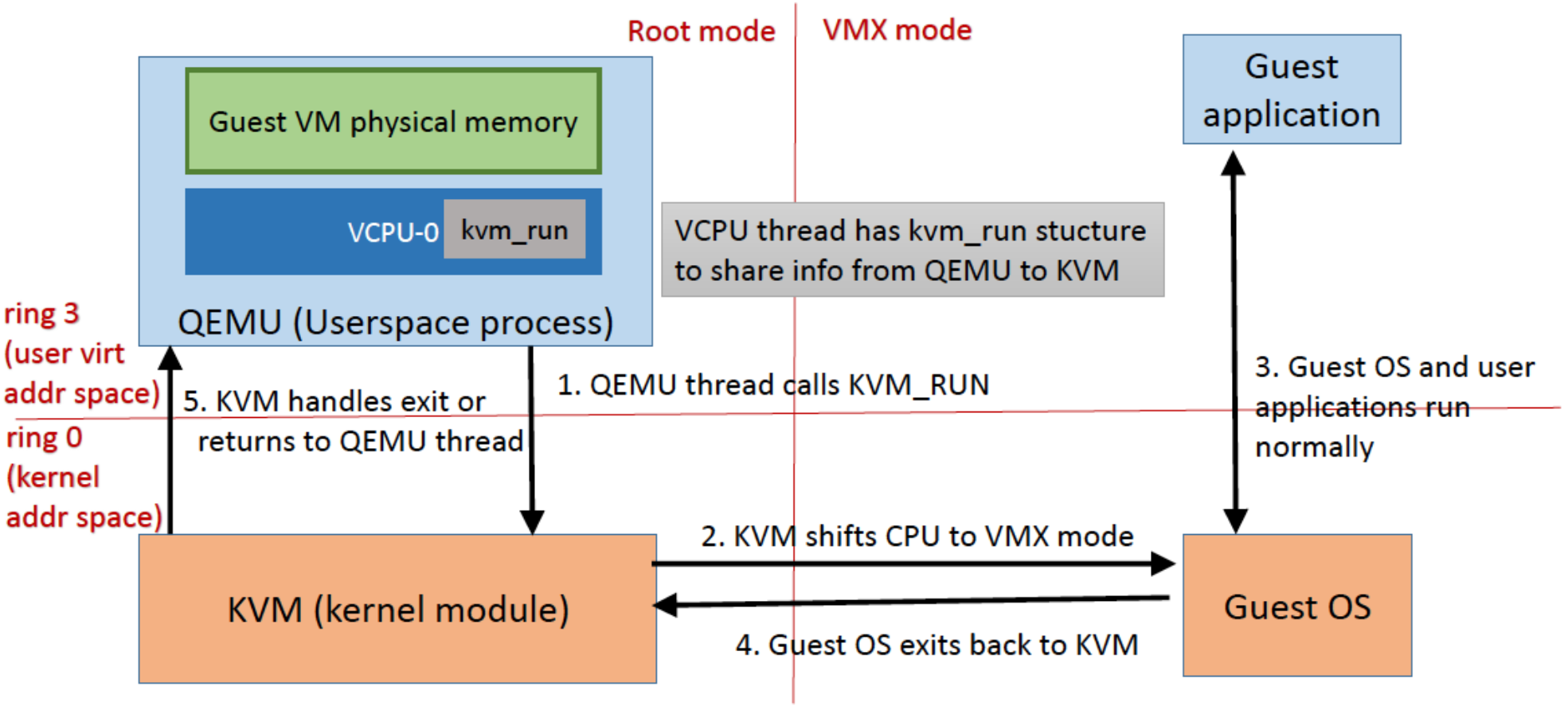
```
open(/dev/kvm)
ioctl(qemu_fd, KVM_CREATE_VM)
ioctl(vm_fd, KVM_CREATE_VCPU)

for(;;) { //each VCPU runs this loop
 ioctl(vcpu_fd, KVM_RUN)
 switch(exit_reason) {
   case KVM_EXIT_IO: //do I/O
   case KVM_EXIT_HLT:
 }
}
```

This ioctlsystem call blocks this thread, KVM switches to VMX mode, runs guest VM

Returns to QEMU on host when VM exits from VMX mode.

QEMU handles exit and returns to guest VM

QEMU Operation



VMX Mode

- Special CPU instructions to enter and exit VMX mode
  - VMLAUNCH, VMRESUME invoked by KVM to enter VMX mode
  - VMEXIT invoked by guest OS to exit VMX mode
- On VMX entry/exit instructions, CPU switches context between host OS to guest OS
  - Page tables (address space), CPU register values etc. switched
  - Hardware manages the mode switch
- Where is CPU context stored during mode switch?
  - Cannot be stored in host OS or guest OS data structures alone (why?)
  - VMCS (VM control structure), also called VMCB (VM control block)

VMCS

- What is VMCS?
  - Common memory area accessible in both modes
  - One VMCS per VM (KVM tells CPU which VMCS to use)

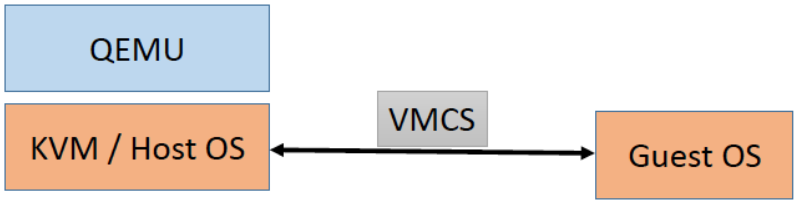
- What is stored in VMCS?
  - Host CPU context:Stored when launching VM, restored on VM exit
  - Guest CPU context:Stored on VM exit, restored when VM is run
  - Guest entry/execution/exit control area:KVM can configure guest Memory and CPU context, which instructions and events should cause VM to exit
  - Exit information:Exit reason and any other exit-related information
- VMCS information exchanged with QEMU via kvm_runstructure
  - VMCS only accessible to KVM in kernel mode, not to QEMU userspace

VMX Mode of Execution

- How is guest OS execution in VMX mode different?
- Restrictions on guest OS execution, configurable exits to KVM
  - Guest OS exits to KVM on certain instructions (e.g., I/O device access)
- No hardware access to guest, emulated by KVM
  - Guest OS usually exits on interrupts (interrupts handled by KVM, assigned to the appropriate host or guest OS)
  - KVM can inject virtual interrupts to guest OS during VMX mode entry
- All of the above controlled by KVM via VMCS
- Mimics the trap-and-emulate architecture with hardware support
  - Guest runs in a (special) ring 0, but trap-and-emulate achieved

Host View

- Host sees QEMU as regular multithreaded process
  - Process that has memory-mapped memory, talks to KVM device via ioctl calls
  - Multiple QEMU VCPU threads can be scheduled in parallel on multiple cores
- When KVM launches a VM, host OS context is stored in VMCS
  - Host OS execution is suspended (all host processes stop)
  - CPU loads guest OS context and guest OS starts running
- When guest OS exits, host OS context is restored from VMCS
  - Host OS resumes in KVM, where it stopped execution
  - KVM can return to QEMU, or host can switch to another process
  - Host OS is not aware of guest OS execution

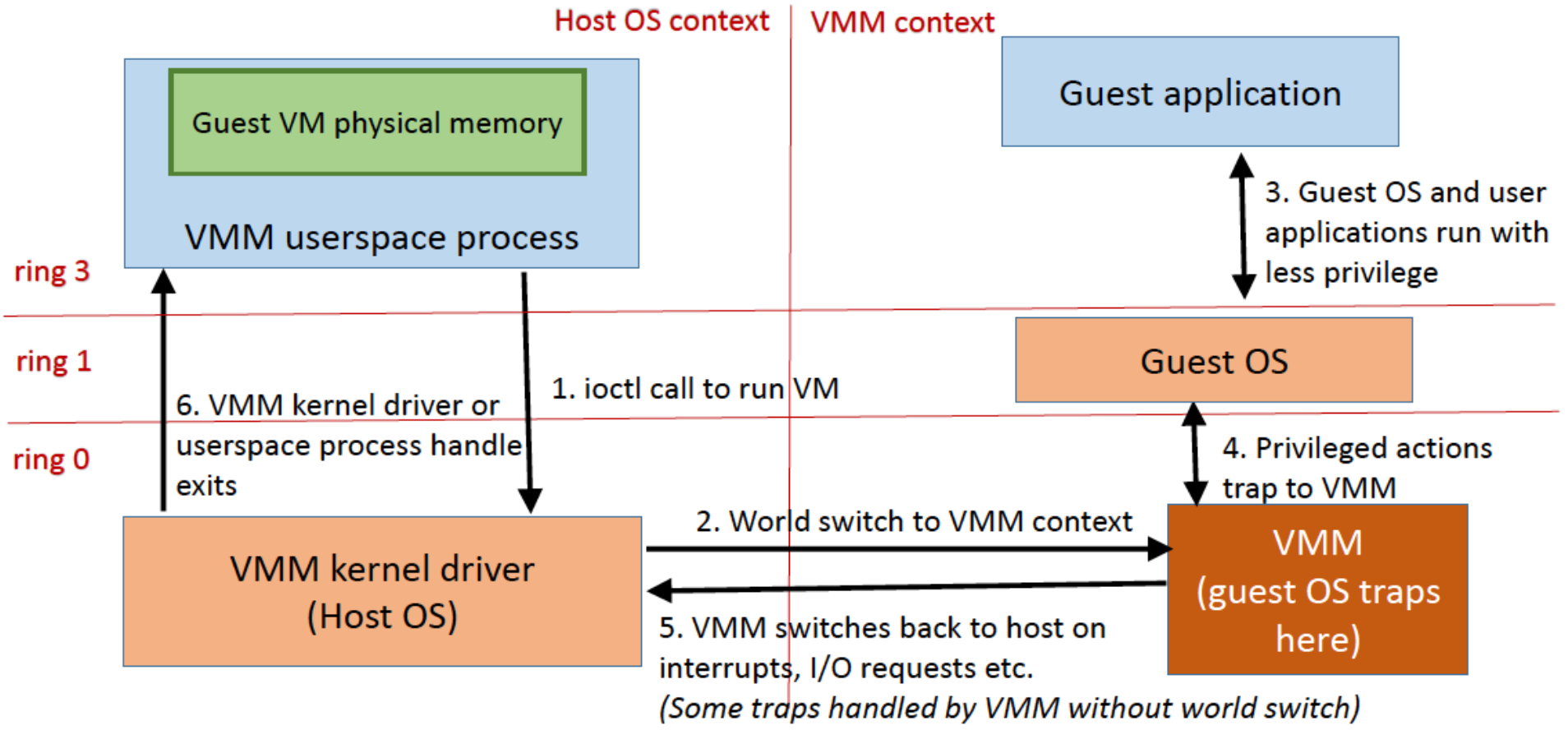


Full Virtualization

- Dynamic(on a need basis) binary(not source) translation of OS instructions
  - Problematic OS instructions translated before execution

- VMWare workstation first to solve the problem of virtualization existing operating systems on x86 (basis for this lecture)
  - Type 2 hypervisor based on trap-and-emulate approach
  - Binary translation is higher overhead than hardware-assisted virtualization
  - Used when hardware support not available

Full Virtualization VMM Architecture

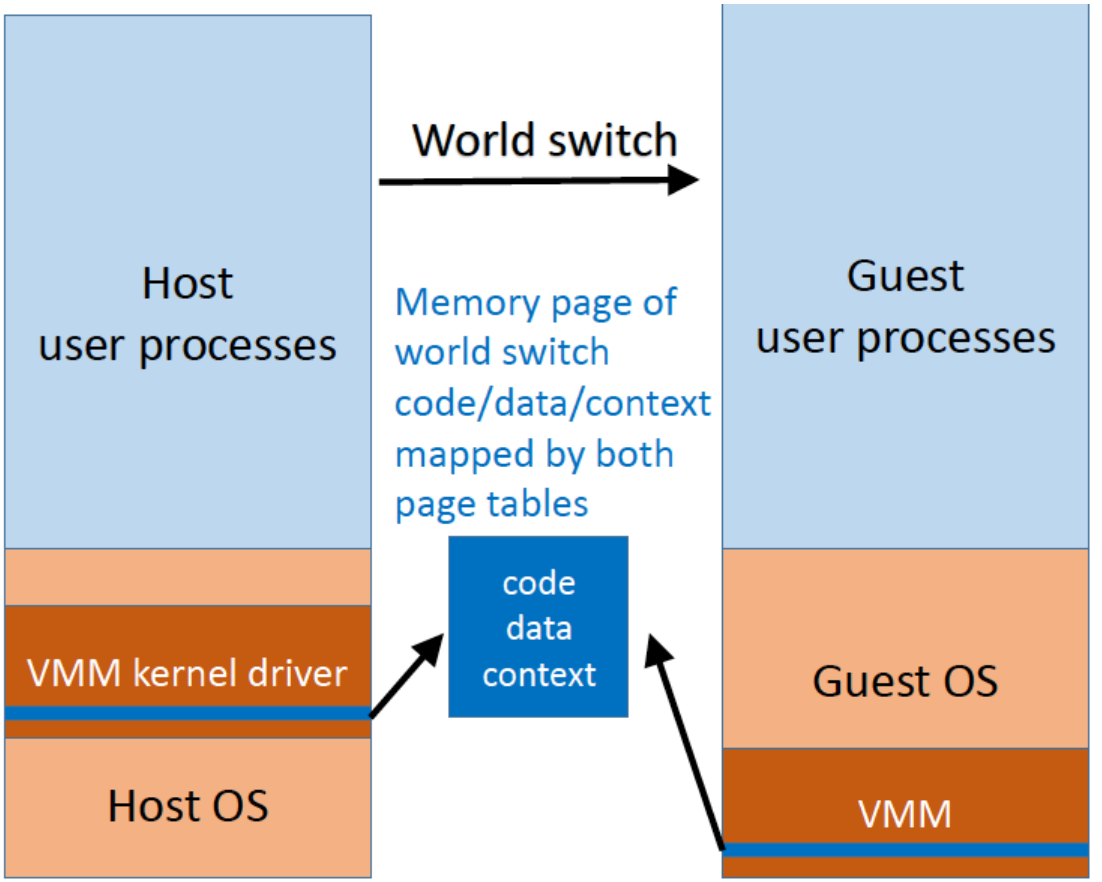


Host and VMM contexts

- Each context has separate page tablesCPU registers etc.
- VMM context :

VMM occupies top 4MB of address space

- Memory page containing code/data of world switch mapped in both contexts
- Host/VMM context saved/restored in this special “cross” page by VMM



Full Virtualization versus QEMU/KVM

- Where is context saved?

• Common cross page mapped into both host and guest address spaces
• KVM: Common memory (VMCS) accessible by CPU in both contexts via special instructions

- Privilege level of guest OS?

• Guest OS runs in ring 1 (lower privilege). Instructions that do not run correctly at lower privilege level are suitably translated to trap to VMM
• KVM: Guest OS runs in VMX ring 0. Some privileged instructions trigger exit to KVM

- How to trap to VMM?

• VMM is located in top 4MB of guest address space , guest OS traps to VMM for privileged ops. World switch to host if VMM cannot handle trap in guest context
• KVM : VMM is not in guest context, guest traps to VMM in host via VM exit

Binary Translation

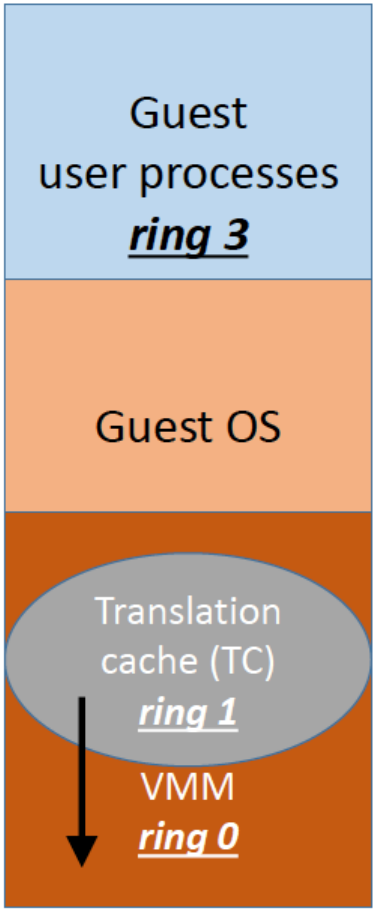
• Guest OS binary is translated instruction-by-instruction and stored in translation cache (TC)

•Part of VMM memory

•Most code stays same, unmodified

•OS code modified to work correctly in ring 1

•Sensitive but unprivileged instructions modified to trap



Guest OS code executes from TC in ring 1

•Privileged OS code traps to VMM

•E.g., I/O, set IDT, set CR3, other privileged ops

•Emulated in VMM context or by switching to host

•VMM sets sensitive data structures like IDT etc. (maintains shadow copies)

Dynamic Binary Translation

• VMM translator logic (ring 0) translates guest code one basic block at a time to produce a compiled code fragment (CCF)

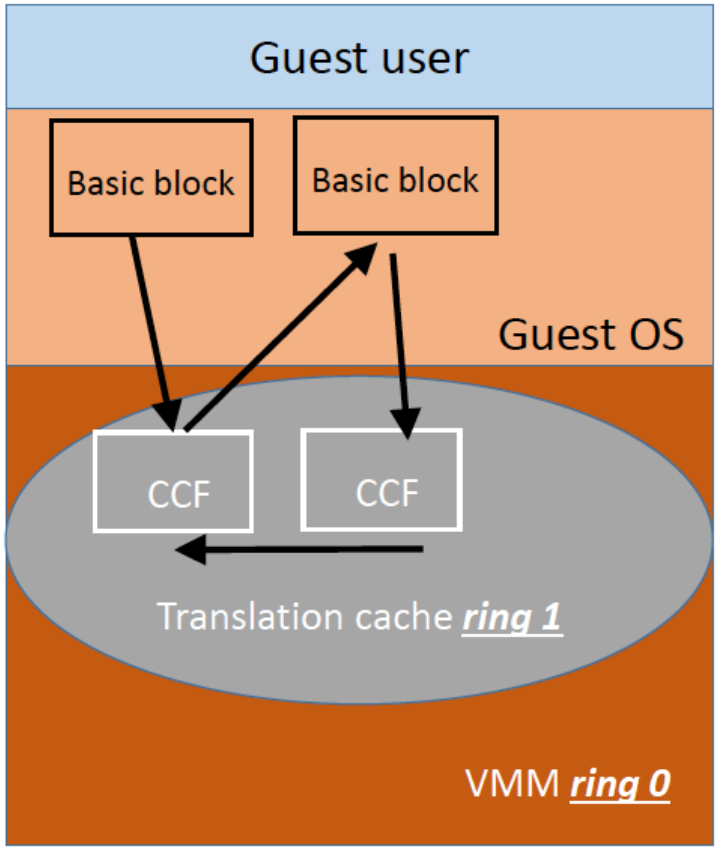
•Basic block = sequence of instructions until a jump/return

•Once CCF is created, move to ring 1 to run translated guest code

•Once CCF ends, “call out” to VMM logic, compute next instruction to jump to, translate, run CCF, and so on

•If next CCF present in TC already, then directly jump to it without invoking VMM translator logic

•Optimization called chaining

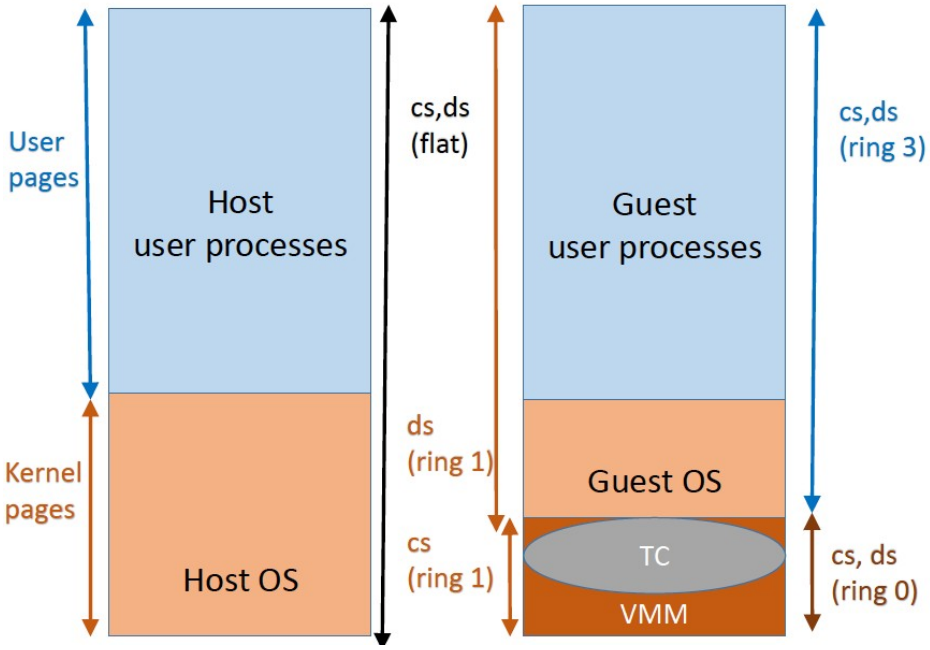


Use of Segmentation for Protection

- Paging protects user code from kernel code via bit in page table entry
- Segments are"flat"
- Separate flat segments for user and kernel modes

Segmentation is used to protect VMM from guest

- Flat segments truncated to exclude VMM
- CS of guest OS (ring 1) points to VMM
- VMM (ring 0) segments point to top 4MB



Dynamic Binary Translation

• VMM translator logic (ring 0) translates guest code one basic block at a time to produce a compiled code fragment (CCF)

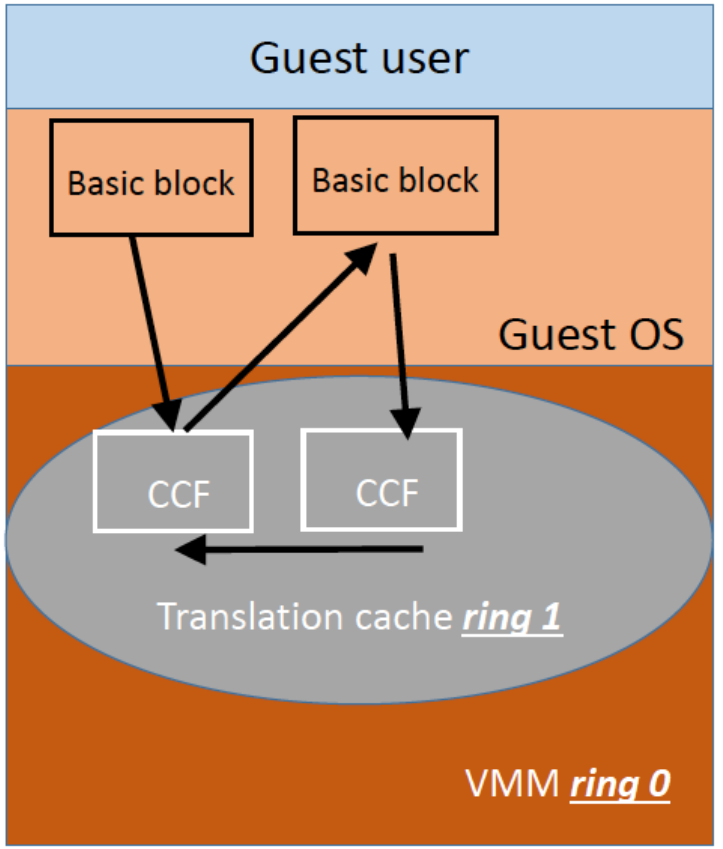
•Basic block = sequence of instructions until a jump/return

•Once CCF is created, move to ring 1 to run translated guest code

•Once CCF ends, “call out” to VMM logic, compute next instruction to jump to, translate, run CCF, and so on

•If next CCF present in TC already, then directly jump to it without invoking VMM translator logic

•Optimization called chaining



Dynamic Binary Translation

• VMM translator logic (ring 0) translates guest code one basic block at a time to produce a compiled code fragment (CCF)

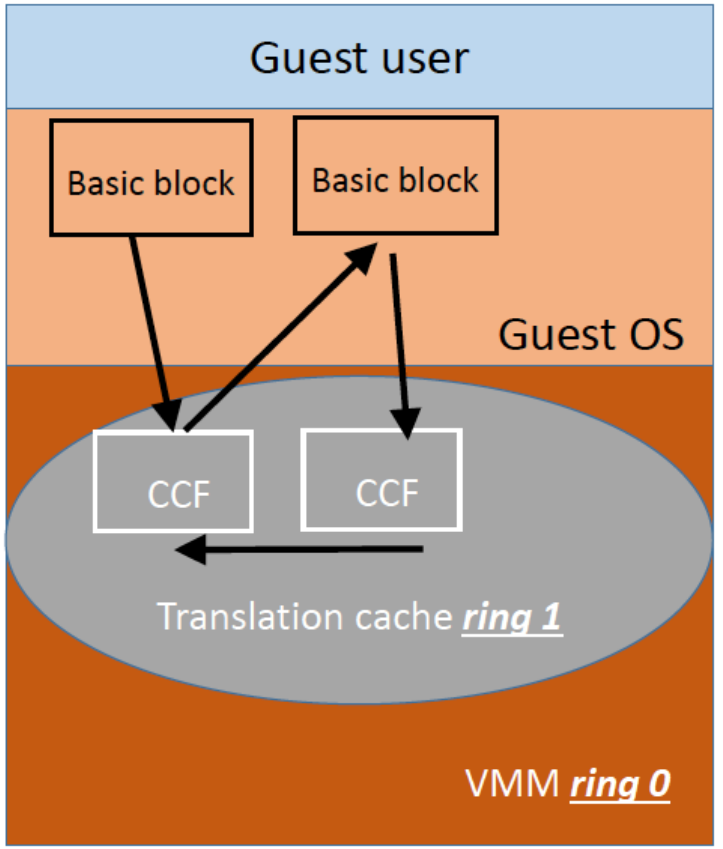
•Basic block = sequence of instructions until a jump/return

•Once CCF is created, move to ring 1 to run translated guest code

•Once CCF ends, “call out” to VMM logic, compute next instruction to jump to, translate, run CCF, and so on

•If next CCF present in TC already, then directly jump to it without invoking VMM translator logic

•Optimization called chaining



Para Virtualization

- Guest operating system is modified to use only instructions that can be virtualized.

- **Reasons for para virtualization :**

  Some aspects of the hardware cannot be virtualized

  Improved performance

  Present a simpler interface

- Examples: Xen, Denaly

Xen and Para Virtualization

- Para virtualization: modify guest OS to be amenable to virtualization

- XenoLinux is a modified Linux OS that runs on Xen hypervisor
- Application interface need not change

- Benefits: better performance than binary translation
- Disadvantages: requires source code changes to OS, porting effort

Xen - A VMM Based on Para Virtualization

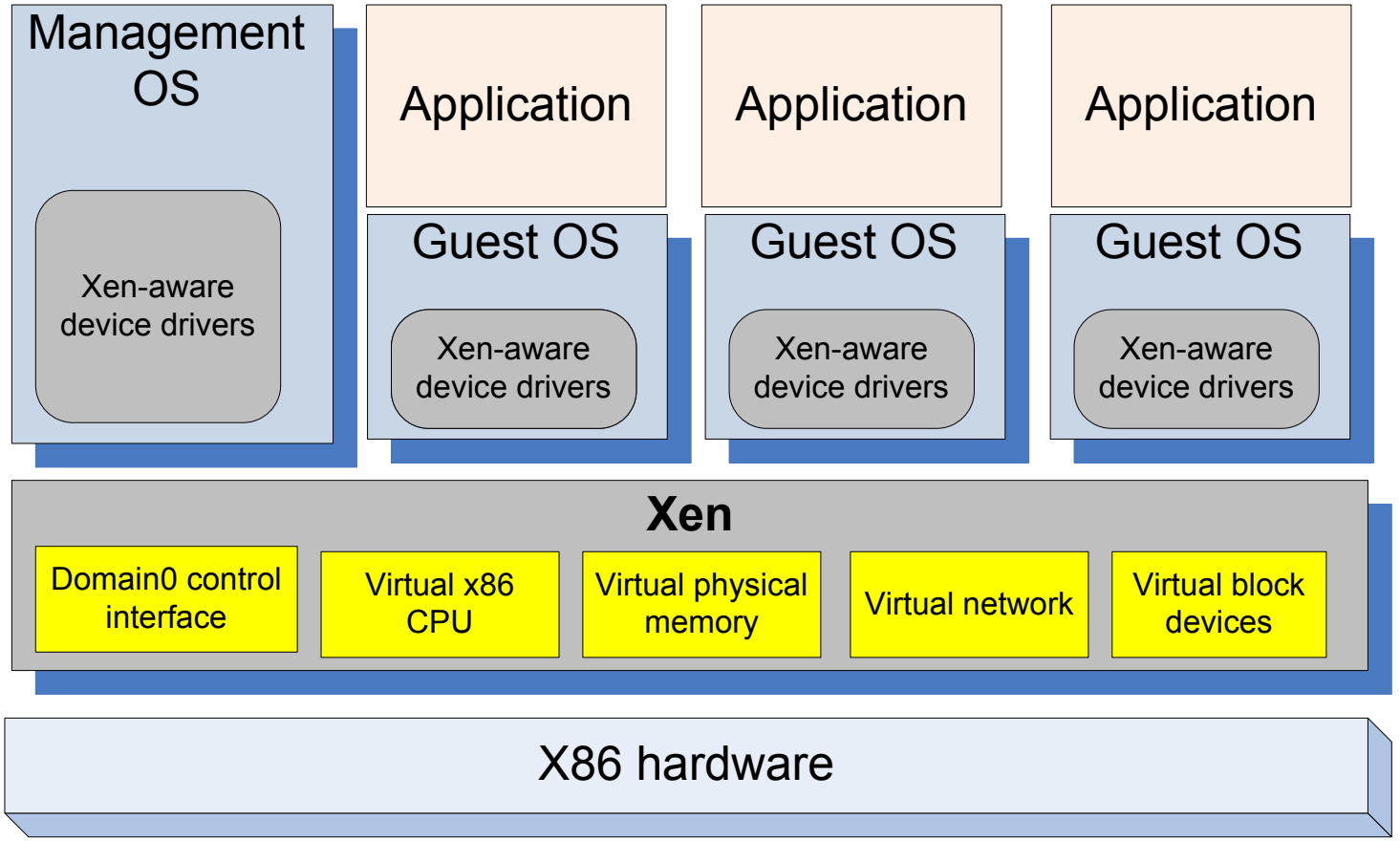


Figure Source : Cloud Computing, Theory and Practice, Dan Marinescu

Xen Architecture

- Type 1 hypervisor: runs directly over hardware
- Trap-and-emulate architecture
  - Xen runs in ring 0, guest OS in ring 1
  - Xen sits in the top 64MB of address space of guests
  - Guest OS traps to Xent o perform privileged actions
- A guest VM is called a domain
  - Special domain called dom0runs control/management software

Dom 0 Components : Xen Store and Tool Stack

- Xen Store – a Dom0 process.
- Supports a system-wide registry and naming service.
- Implemented as a hierarchical key-value storage.
- A watch function informs listeners of changes of the key in storage they have subscribed to.
- Communicates with guest VMs via shared memory using Dom0 privileges.

- Tool stack - responsible for creating, destroying, and managing the resources and privileges of VMs.
- To create a new VM, a user provides a configuration file describing memory and CPU allocations and device configurations.
- Tool stack parses this file and writes this information in Xen Store.
- Takes advantage of Dom0 privileges to map guest memory, to load a kernel and virtual BIOS and to set up initial communication channels with Xen Store and with the virtual console when a new VM is created.

Figure Source : Cloud Computing, Theory and Practice, Dan Marinescu

Xen : Strategies for Virtualization

Function	Strategy
Paging	A domain may be allocated discontinuous pages. A guest OS has direct access to page tables and handles pages faults directly for efficiency; page table updates are batched for performance and validated by *Xen* for safety.
Memory	Memory is statically partitioned between domains to provide strong isolation. `XenoLinux` implements a *balloon driver* to adjust domain memory.
Protection	A guest OS runs at a lower priority level, in ring 1, while *Xen* runs in ring 0.
Exceptions	A guest OS must register with *Xen* a description table with the addresses of exception handlers previously validated; exception handlers other than the page fault handler are identical with *x86* native exception handlers.
System calls	To increase efficiency, a guest OS must install a "fast" handler to allow system calls from an application to the guest OS and avoid indirection through *Xen*.
Interrupts	A lightweight event system replaces hardware interrupts; synchronous system calls from a domain to *Xen* use *hypercalls* and notifications are delivered using the asynchronous event system.
Multiplexing	A guest OS may run multiple applications.
Time	Each guest OS has a timer interface and is aware of "real" and "virtual" time.
Network and I/O devices	Data is transferred using asynchronous I/O rings; a ring is a circular queue of descriptors allocated by a domain and accessible within *Xen*.
Disk access	Only *Dom0* has direct access to IDE and SCSI disks; all other domains access persistent storage through the Virtual Block Device (VBD) abstraction.

Figure Source : Cloud Computing, Theory and Practice, Dan Marinescu

CPU Virtualization in Xen

Guest OS code modified to not invoke any privileged instruction

- Any privileged operation traps to Xen in ring 0

- Hyper calls: guest OS voluntarily invokes Xen to perform privileged ops
- Much like system calls from user process to kernel
- Synchronous: guest pauses while Xen services the hypercall

- Asynchronous event mechanism: communication from Xen to domain

- Much like interrupts from hardware to kernel
- Used to deliver hardware interrupts and other notifications to domain
- Domain registers event handler callback functions

Trap Handling in Xen

When trap/interrupt occurs, Xen copies the trap frame onto the guest OS kernel stack, invokes guest interrupt handler

- Guest registers an interrupt descriptor table with Xen to handle traps
  - Interrupt handlers validated by Xen(check that no privileged segments loaded)
- Guest trap handlers work off information on kernel stack, no modifications needed to guest OS code
  - Except page fault handler, which needs to read CR2 register to find faulting address (privileged operation)
  - Page fault handler modified to read faulting address from kernel stack (address placed on stack by Xen)
- What if interrupt handler still invokes privileged operations?
  - Traps to Xenagain and Xendetects this "double fault" (trap followed by another trap from interrupt handler code) and terminates misbehaving guest

Figure Source : Cloud Computing, Theory and Practice, Dan Marinescu

Memory Virtualization in Xen

One copy of combined GVA→HPA page table maintained by guest OS

- CR3 points to this page table
- Like shadow page tables, but in guest memory, not in VMM

- Guest is given read-only access to guest "RAM" mappings (GPA→HPA)
  - Using this, guest can construct combined GVA→GPA mapping

- Guest page table is in guest memory, but validated by Xen
  - Guest marks its page table pages as read-only, cannot modify
  - When guest needs to update, it makes a hypercall to Xen to update page table
  - Xen validates updates (is guest accessing its slice of RAM?) and applies them
  - Batched updates for better performance

- Segment descriptor tables are also maintained similarly
  - Read-only copy in guest memory, updates validated and applied by Xen
  - Segments truncated to exclude top 64MB occupied by Xen

Xen Abstractions for Network and I/O

- Each domain has one or more Virtual Network Interfaces (VIFs) which support the functionality of a network interface card. A VIF is attached to a Virtual Firewall-Router (VFR).
- Split drivers have a front-end in the DomU and the back-end in Dom0; the two communicate via a ring in shared memory.
- Ring - a circular queue of descriptors allocated by a domain and accessible within Xen. Descriptors do not contain data, the data buffers are allocated off-band by the guest OS.
- Two rings of buffer descriptors, one for packet sending and one for packet receiving, are supported.
- To transmit a packet :
  - a guest OS enqueues a buffer descriptor to the send ring,
  - then Xen copies the descriptor and checks safety,
  - copies only the packet header, not the payload, and
  - executes the matching rules.

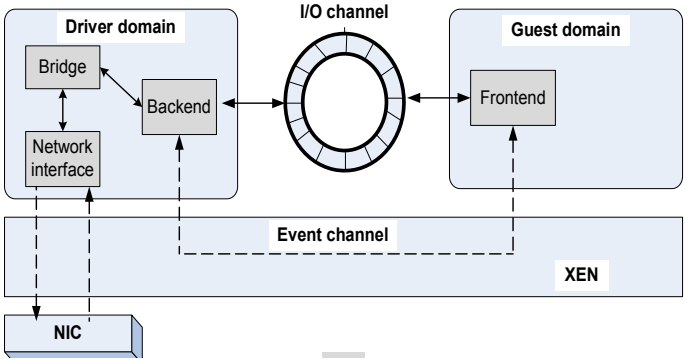


Figure Source : Cloud Computing, Theory and Practice, Dan Marinescu

Performance Comparison of VMs

The questions examined are:
(a) how performance scales up with the load;
(b) the impact of a mix of applications;
(c) implications of the load assignment on individual servers?

Main conclusions:

1. Xen virtualization overhead of is higher than OpenVZ primarily due to L2-cache misses.

2. Performance degradation when the workload increases is also noticeable for Xen.

3. Hosting multiple tiers of the same application on the same server is not an optimal solution.

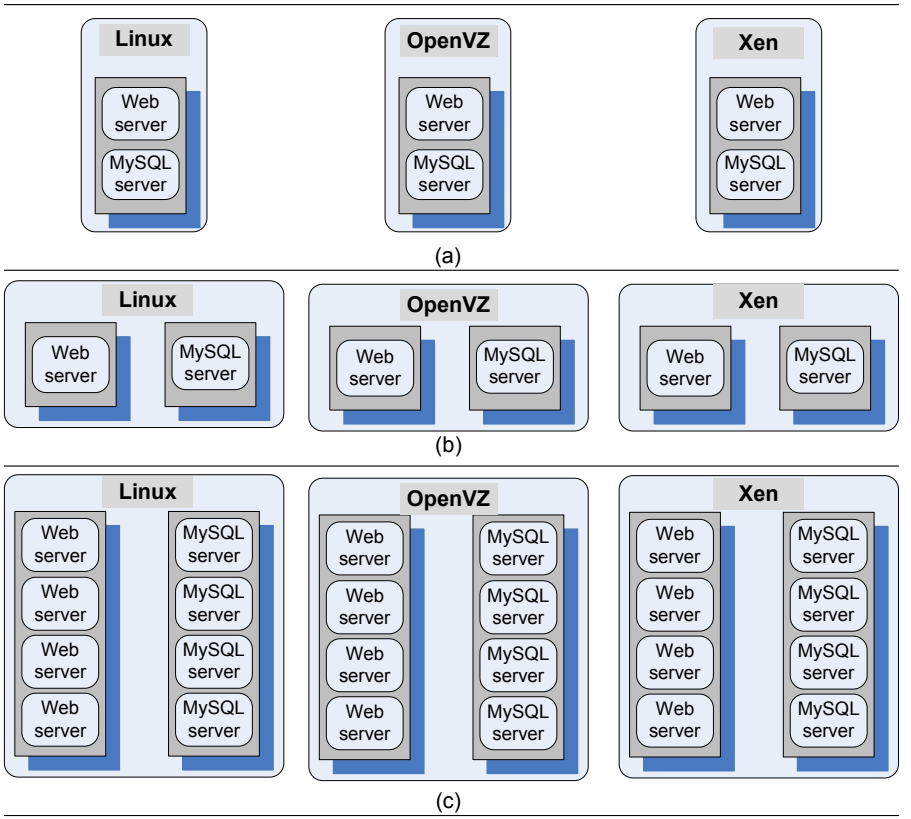


Figure Source : Cloud Computing, Theory and Practice, Dan Marinescu