Linux and Xen

Andrea Sarro

andrea.sarro(at)quadrics.it

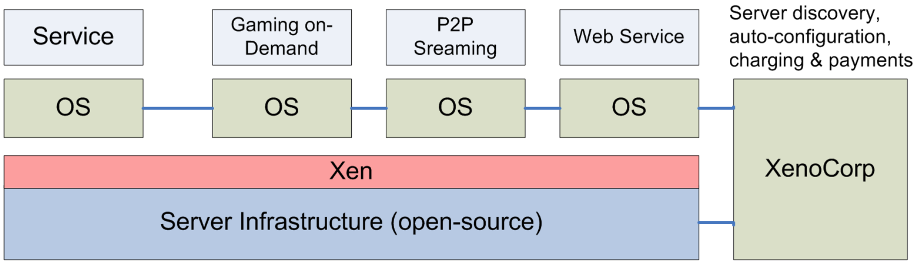
Linux Kernel Hacking Free Course IV Edition

What is Xen?

Xen is an open source *platform virtualization* solution:

- It was born as part of University of Cambridge XenoServers research project
  - XenoServers is aimed to provide a public infrastructure for distributed computing
  - XenoServers offers the possibility to deploy untrusted services and execute untrusted code on a shared platform with resource sharing and without security issues

What is Xen? (cont)


Service
Gaming on-Demand
P2P Sreaming
Web Service
Server discovery, auto-configuration, charging & payments
OS
OS
OS
OS
XenoCorp
Xen
Server Infrastructure (open-source)

What is (Platform) “Virtualization”?

1. Virtualization provides the ability to run several isolated execution environments over a single physical machine
2. Virtualization hides the physical characteristics of computational resources to simplify interaction mechanisms with other systems, applications and end users
3. Virtualization is the logical representation of resources without physical constraints

Why Virtualize?

Main virtualization goals:

- Increase resource utilization and throughput
- Decrease operational costs
- Provide isolation between applications and between users

Main applications:

- Server consolidation
- Legacy applications execution
- Untrusted code execution
- Concurrent execution of different OS instances
- Software migration (for HA and Fault Tolerance)
- Software appliances

Virtualization History

From the beginnings (mainframes era)...

- ’60s: IBM M44/44X
- 1972: IBM VM/370 for System/370

to modern times (PCs era)...

- 2001: VMware Enterprise ESX Server
- 2005: Intel VT-x and AMD-V
- 2006: Gartner “Top 10 Strategic Technologies List”: 1st place assigned to *virtualization technology*

Virtualization History - Xen

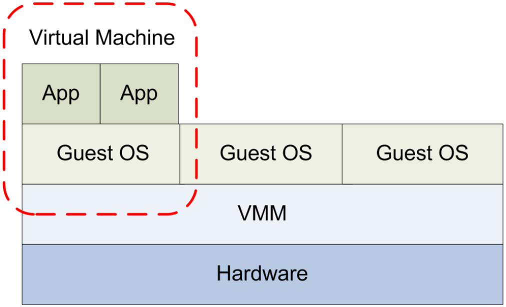
Main Xen *milestones*:

- 2003: First Xen public release. Official presentation at SOSP (Symposium on Operating System Principles)
- Early 2005: XenSource Inc. was founded by project developers
- End of 2005: Xen 3.0 release with *hardware virtualization technologies* support
- 2007: XenSource acquisition by Citrix Systems; concomitant creation of the "Xen Advisory Board"

Virtualization Key Concepts

Typical components of a platform virtualization solution:

- Hardware (host)
- Virtual Machine Monitor (VMM)
- Virtual Machine (VM)
- Guest Software (guest OS + guest applications)



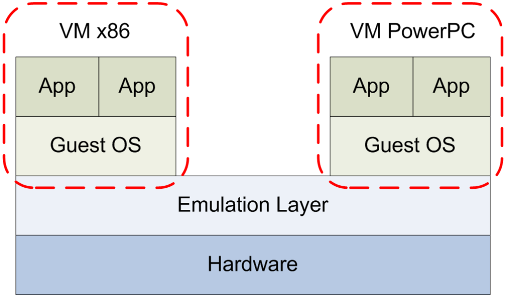
Virtualization Techniques

There are several platform virtualization techniques, each one with pros and cons:

- System Emulation
- Full Virtualization
- Paravirtualization
- Operating System Level Virtualization

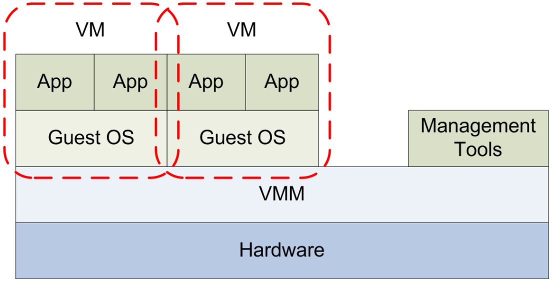
System Emulation

- All instructions are translated by the emulation layer
- It allows execution of code for a different architecture than the physical processor one
- Emulation imposes a huge performance overhead due to full code translation



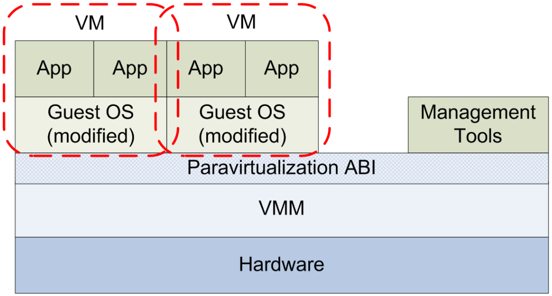
Full Virtualization

- Instructions are translated only when necessary, otherwise they are executed unaltered by the processor
- It allows execution of not modified guest operating systems that support the physical processor architecture
- It uses *trap & emulate* and *binary rewriting* techniques



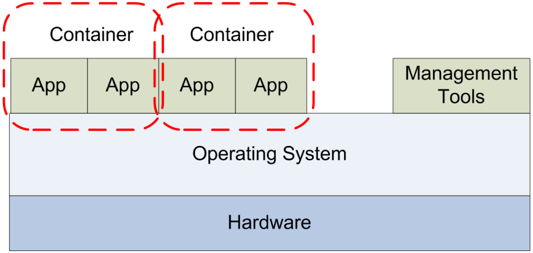
Paravirtualization

- It requires cooperation between VMM and guest OS
- The VMM exports an interface accessible from the VMs
- The guest OS has to be modified to access these new interfaces



Operating System Level Virtualization

- It uses only one instance of the operating system
- The OS guarantees isolation and independence of guest software execution environments (*containers*)
- From a performance perspective, it resembles the native unmodified operating system



Xen Hypervisor

Xen is an open source Virtual Machine Monitor (*hypervisor*) that adopts the paravirtualization technique. It requires Linux to be modified in order to allow execution inside a virtual machine.

Xen virtualization solution for x86 architecture:

- The analyzed processor architecture is IA-32, also known as x86. Xen also supports IA-32e, also known as x86-64 (not analyzed in this presentation)
- Xen 3.1 is the Virtual Machine Monitor (VMM), also called *hypervisor*
- Linux 2.6.20 is the guest operating system

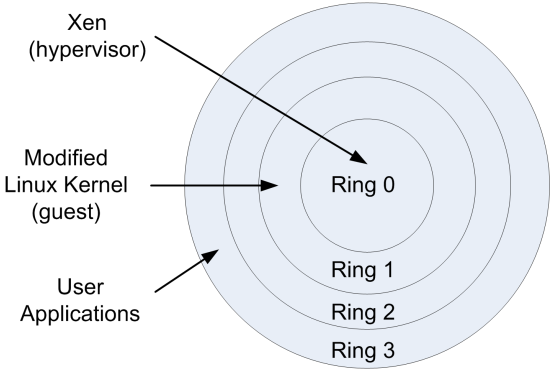
Ring Deprivileging

Xen adopts a *ring deprivileging* strategy, assigning IA-32 protection rings in a different fashion than Linux:

- Ring 0 is assigned to the Xen hypervisor, while in Linux vanilla is used by the kernel for kernel mode execution
- Ring 1 is assigned to the Linux kernel, requiring modifications to *behavior sensitive* code (instructions that change silently their behavior if executed with $CPL \neq 0$)
- Ring 3 is used for user mode execution like in Linux vanilla, so applications don't have to be modified to be executed inside a virtual machine

Ring Deprivileging (cont)

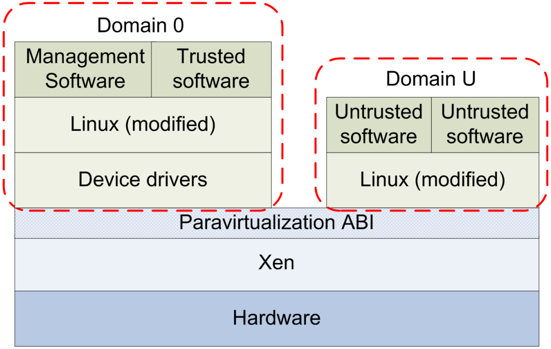
Ring deprivileging is necessary in order to allow exclusive hardware control only to the Xen hypervisor.



Xen Architecture

Xen calls the virtual machines *domains*. There are two kinds of domains:

- Domain 0 (“privileged” domain), built during system startup
- Domain U (“unprivileged” domain(s)), launched by administration tools in *domain 0*



Linux and Xen

Due to ring deprivileging, Linux as a guest OS has to be modified in the following aspects:

- Execution of behavior sensitive code (and also most of privileged code for performance reasons) has to be performed via *hypercalls*
- Handling mechanisms for interrupts and exceptions have to be defined
- A *split driver architecture* for I/O device drivers has to be implemented
- Memory management should take into account memory virtualization

Hypercalls

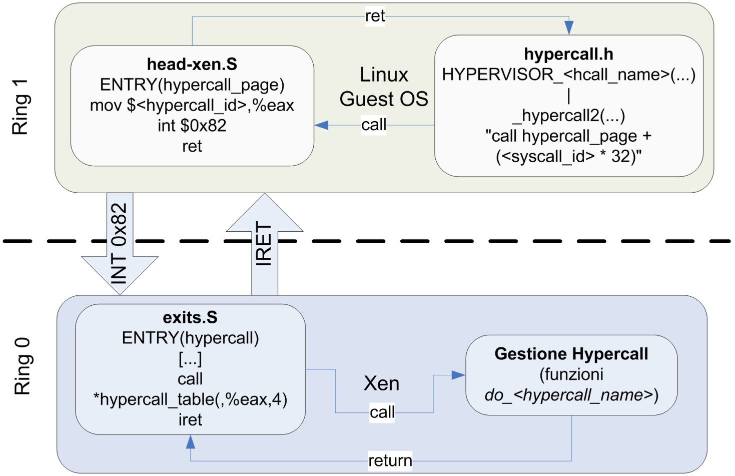
While privileged instructions (that require *CPL* $= 0$) generate a *trap* to the hypervisor via GPFs, there are mainly two reasons for explicitly calling the hypervisor:

- Reduce performance hit due to *trap & emulate*
- Execute behavior sensitive code

Hypercalls are similar to (legacy) Linux system calls:

- They represent an entry point to the hypervisor facilities
- The switch between the guest operating system and the hypervisor is demanded to a software interrupt (`int 0x82`)

Hypercalls (cont)



Exceptions

In Linux vanilla the IDT is initialized in `trap_init()` using `set_<type>_gate()` functions.

Because Xen handles the IDT, it requires all calls to these function to be replaced with a single call to the `HYPERVISOR_set_trap_table()` hypercall.

`HYPERVISOR_set_trap_table()` accepts as a parameter the virtual IDT of the guest, represented by the `trap_table` structure (of type `struct trap_info`) in *traps-xen.c*.

`struct trap_info` resembles a *trap* or *interrupt gate*, having fields for vector, handler segment selector and offset.

Exceptions (cont)

When an exception occurs the processor transfers control to the Xen hypervisor, using the Xen exception handlers in *entry.S*.

If the exception was caused by the guest OS, these handlers call `do_guest_trap`.

`do_guest_trap`:

1. Gets from the guest context the gate for the exception
2. Creates the *exception frame* required by the guest OS to process the exception

Then `iret` is executed to return control to the guest OS exception handler

Event Channels

Event channels are the Xen facility used to notify events between VMM and domains.

They are used for hardware interrupt notification to domains and for signaling a pending event, like a termination request.

Event channel:

- An event channel represent a binary information (1 bit)
- When an event occurs this information changes from 0 to 1, behaving as a notification flag (*pending flag*)

Event Channels (cont)

Event channel operations are invoked through the `HYPERVISOR_event_channel_op` hypercall, specifying a command and a list of arguments.

An event channel can be viewed as a communication channel through which an event is transmitted:

- Event channel instances are called *ports*
- Event channel operations include *bind* (to another port or to an IRQ), *send*, *close*

Interrupts

In Xen interrupts to be notified to the Linux guest OS are handled through the event channels notification mechanism.

During startup the guest OS installs two handlers (*event* and *failsafe*) via the `HYPERVISOR_set_callbacks` hypercall:

- The *event* callback is the handler to be called to notify an event to the guest OS
- The *failsafe* callback is used when a fault occurs when using the *event* callback

The guest OS can install a handler for a physical IRQ through the `HYPERVISOR_event_channel_op` hypercall, specifying as operation `EVTCHNOP_bind_pirq`.

Interrupts (cont)

When an interrupt occurs control passes to the Xen `common_interrupt` routine, that calls the Xen `do_IRQ` function.

`do_IRQ`:

Checks who has the responsibility to handle the interrupt:

- The VMM: the interrupt is handled internally by the VMM
- One ore more guest OS: it calls `__do_IRQ_guest` function

`__do_IRQ_guest`:

For each domain that has a binding to the IRQ sets to 1 the *pending flag* of the event channel via `send_guest_pirq`

Interrupts (cont)

The entry point in Linux is the `hypervisor_callback` function (is the *event* callback handler installed at startup), that calls `evtchn_do_upcall`.

`evtchn_do_upcall`:

1. Checks for *pending* events
2. Resets to zero the *pending flag*
3. Uses the `evtchn_to_irq` array to identify the IRQ binding for the event channel
4. Calls Linux `do_IRQ` interrupt handler function

I/O device driver model

Xen doesn't include device drivers (except those strictly required to boot) for the hardware platform, but dispatches this task to the *domain 0* Linux operating system.

To support the *split driver architecture*, Linux has to provide two kinds of drivers:

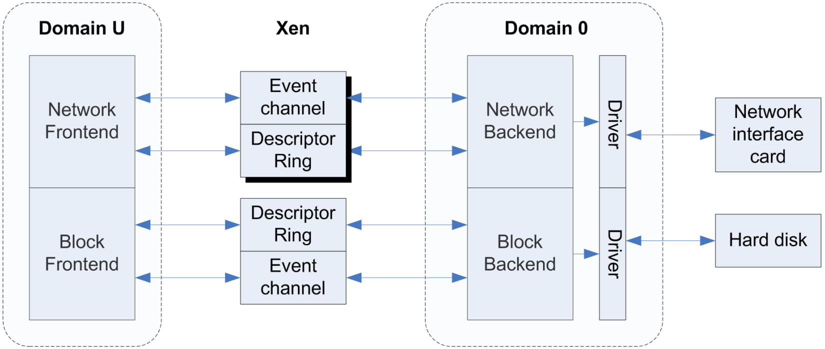
- Backend drivers (executed in *domain 0*): they receive requests from the *domain U frontend drivers*, doing multiplexing and sequencing and communicating with the real device driver. Responses are demultiplexed and sent back to the *domain U frontend drivers*.
- Frontend drivers (executed in *domain U*): they substitute the real device drivers for the end applications, exporting a compatible interface. All requests are forwarded to the *domain 0 backend drivers*.

I/O device driver model (cont)

A *frontend* driver communicates with the corresponding *backend* driver through two facilities, managed by Xen:

- Shared memory (and *descriptor rings*)
- Event channels (interdomain)

I/O device driver model (cont)

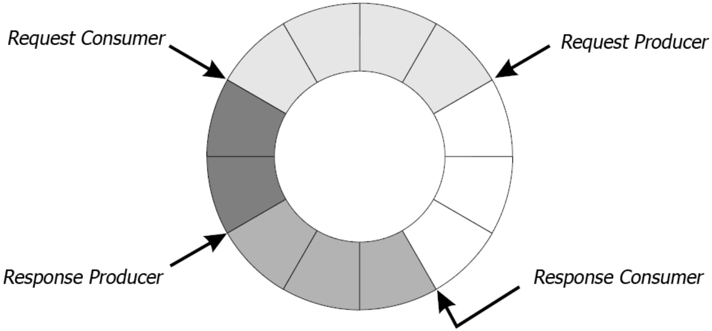


I/O device driver model (cont)

A descriptor ring is a circular buffer used in a producer/consumer fashion to support transfer of requests and responses.

Notification has to be performed using event channels, providing the opportunity of batching requests/responses.

I/O device driver model (cont)



Memory management

Memory virtualization is a hard task to accomplish on IA-32 architecture because:

- IA-32 TLB can't be managed by software
- A *TLB flush* is required at each domain switch

Guidelines followed during Xen memory management strategy development:

- Each guest operating system should be responsible for its page table management, keeping at minimum Xen intervention
- Top 64 MB of the address space in each domain are reserved to Xen, in order to not require a *TLB flush* during transitions between guests and VMM

Memory management (cont)

Xen distinguishes between two kinds of memory:

- Machine memory: it's the whole physical memory in the system
- Pseudo-physical memory: it's an abstraction valid for each domain, allowing the guest OS to consider it's own memory as a contiguous block, hiding the underlying potentially fragmented physical layout (machine memory)

Two translation tables are defined:

- machine-to-pseudophysical: maintained by Xen, maps machine addresses to pseudo-physical addresses
- pseudophysical-to-machine: maintained by each domain, maps pseudo-physical addresses to machine addresses

Memory management (cont)

Using this kind of memory organization only guest OS routines that require the knowledge of the real physical layout have to be modified (i.e. page table management), while other routines can continue to operate unmodified on pseudo-physical memory.

Xen handles page tables in three different ways:

- Read Only Page Tables
- Writable Page Tables
- Shadow Page Tables

Concluding remarks

Xen is a powerful platform virtualization solution.

- Because it adopts the paravirtualization technique it requires the guest OS (i.e. Linux) to be modified in order to use the interfaces exported by the VMM to the virtual machines.
- Beginning from version 3.0, Xen supports *hardware virtualization technologies*, like Intel VT-X, though it's HVM (Hardware Virtual Machine) component:
  - When adopting these technologies Xen is a full virtualization solution, that can run unmodified guest operating systems (except for *domain 0)*.
  - When HVM is enabled, Xen pays a substantial performance overhead in I/O operations and memory management compared to Xen PV (paravirtualized).

Concluding remarks (cont)

Xen performance overhead compared to Linux vanilla (preliminary results from simple *real world* benchmarks, like *kernel compile time* and *kernel archive decompress time*):

- Xen PV has a small overhead [3-10%]
- Xen HVM has a huge overhead [30-60%]

Further performance studies are being developed to understand the performance overhead in real scenarios and to evaluate scalability and security of virtualization solutions.

A Xen PV drawback: Xen patches for the Linux kernel are not always available for the last stable version of the kernel.