Virtualization technologies: design and implementation

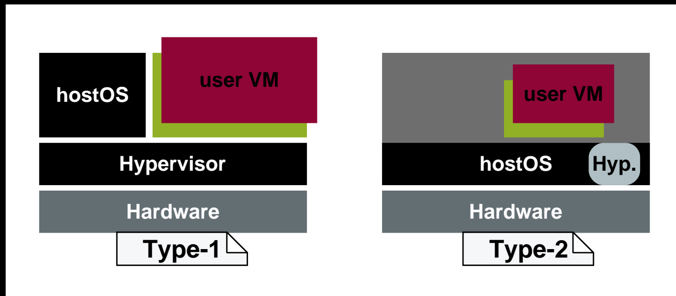
Alain Tchana - (alain.tchana@grenoble-inp.fr)

krakOS - France

Virtualization system design

System-level virtualization

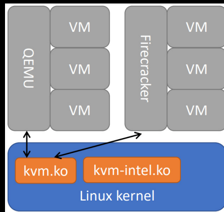
- Bare-metal hypervisor (type-1, microkernel design): Xen, hyper-V, VMware ESXi
- Hosted hypervisor (type-2): Linux/KVM



Quick presentation of a hosted hypervisor (type-2)

Linux KVM

- Composed of two loadable modules
  - *kvm.ko:* the general purpose hypervisor
  - an additional module per architecture. e.g., *kvm-intel.ko*)
- An additional user space hypervisor (QEMU, Firecracker, kvmtool)



Linux KVM advantages and limitations

Advantages

- Based on Linux
  - large community, thus can evolve very quickly

Limitations

- Based on Linux
  - large TCB (around 25M LOCs[a]): bugs, vulnerabilities
  - VMs are managed as processes, thus best effort, not in the spirit of cloud (multi-tenants). This could lead to performance unpredictability. Need a lot of tuning. e.g., explain the issue related to the page cache

[a] https://www.phoronix.com/scan.php?page=news_item&px=Linux-Kernel-Commits-2017

Why Xen (type-1) for this class?

Advantages

- Dedicated to virtualization
- Small TCB (around 0.150M LOC)
- Popular in both industry (used by Amazon, among others) and research (used by Alain Tchana, among others)

Limitations

- Small community, compared to Linux/KVM

I think that type-1 is the best virtualization model. I need a paper to demonstrate that!

Why Xen (type-1) for this class?

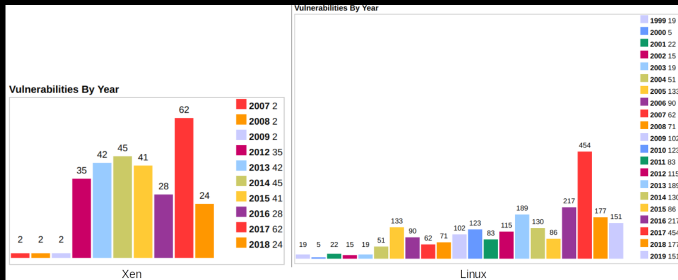
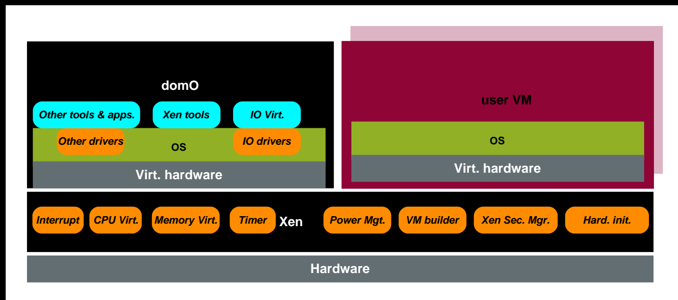


Figure: *From CVE website, visited on Sep. 14th, 2019 at 11am*

1. Focus on Xen (type-1 virtualization)

Xen general architecture



- Xen starts first
- Then it builds and starts *dom0*
- *dom0* is an extension of Xen

Installation

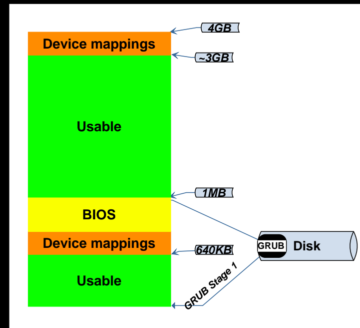
- Xen is installed as a normal kernel
  - in /boot
  - taken into account by GRUB
- The original kernel is configured as a boot module
  - it services as dom0 (includes Xen tools)

Demo and practical class by students (Xen installation)

```
git clone
https://gitlab.com/lenapster/ensl-cr03-virtualization.git
```

Boot process

1. BIOS is loaded into RAM
2. BIOS initializes and checks the hardware (e.g., keyboard, memory)
3. BIOS loads (from the MBR) into RAM another program, which is generally the boot loader (e.g., GRUB). The MBR is the firsts 512 bytes of the boot device
4. GRUB runs in two phases (because the memory space reserved for the boot loader is very small)
5. If selected, Xen boots like a classical kernel



The boot process

- Xen checks the hardware and does some initialization operations in order to be able to handle earlier dom0 and hardware requests.
- Xen builds dom0, loads and starts it.
- dom0 kernel does the remaining hardware initialization operations
  - starts `/sbin/init`, as any Linux kernel does
  - `/sbin/init` runs all init scripts in `/etc/rc*.d` folders
  - Xen related scripts are thus launched: Xen commons (`xencommons`), network virtualization scripts, `xendomains` (responsible for starting user VMs at machine boot time)

Demo, ring 0 and VMX (explain)

Code reading of Xen version 4.10.0

- *config, m4, and scripts:* configurations for compilation
- *docs:* some documentations
- *extra:* a mini guest operating system
- *stubdom:* implementation of several driver domains (called stubdomains). Used when dom0 is disaggregated.
- *tools:* implementation of Xen tools, will be hosted by dom0
- *unmodified_drivers:* para-virtualized drivers for domU kernel versions under 2.6.36. Now Linux code naturally includes Xen codes for paravirtualized drivers
- *xen:* implementation of the bare-metal hypervisor

Demo

Code reading of Xen version 4.10.0

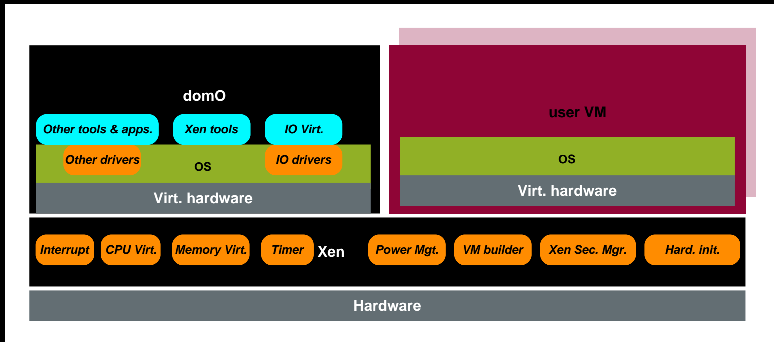
xen folder

- arch/x86: specific code for x86 architectures
- arch/x86/boot: head.S and x86_64.S
- arch/x86/: setup.c, domain.c, domctl.c, dom0_build.c, mm.c, trap.c, hypercall.c
- arch/hvm: for handling VMX transitions (entry.S). vmcs.c manages VMCS. vmx.c implements VMX. (see latter)
- arch/mm: memory virtualization

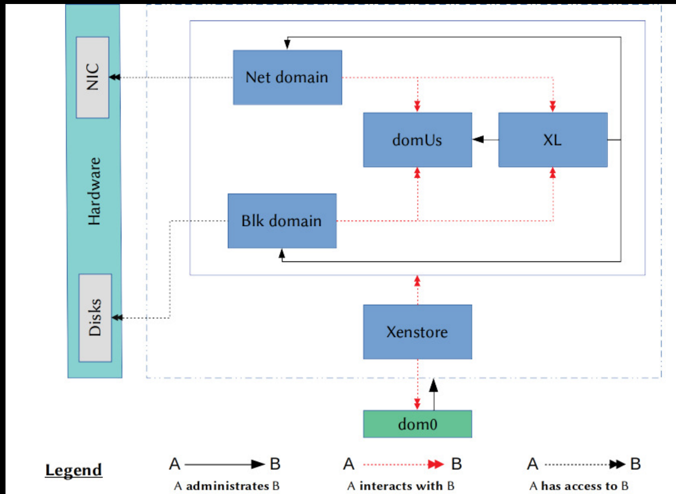
Demo

dom0 architecture

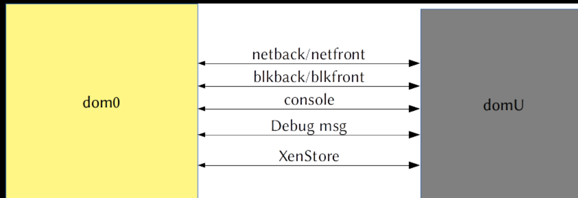
- IO drivers
- VM administration tool (xl)
- XenStore
- Cloud manager agent



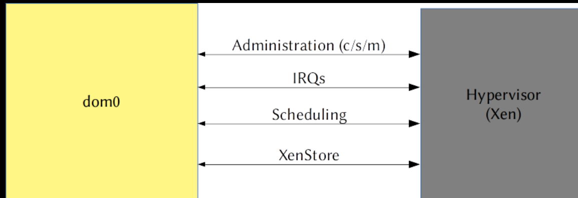
dom0 architecture



dom0 architecture



dom0 architecture



Xen tools

tools folder

- `xl`: implements administration commands
- `libxl` (Xen light library): factorized code related to administration commands
- `libxc` (Xen control library): implements the low level logic for invoking Xen, using hypercalls

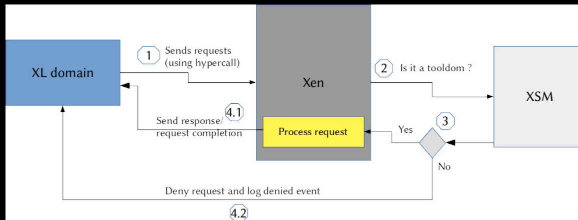
xl CLI

libxl

libxenctrl

Xen

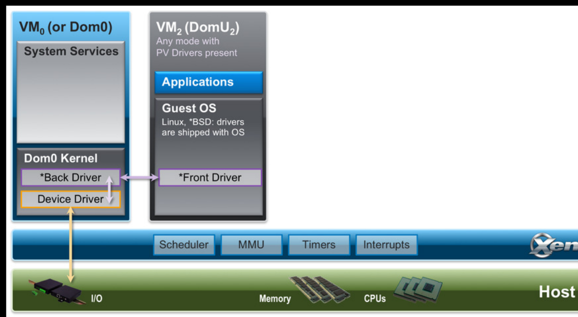
Xen tools



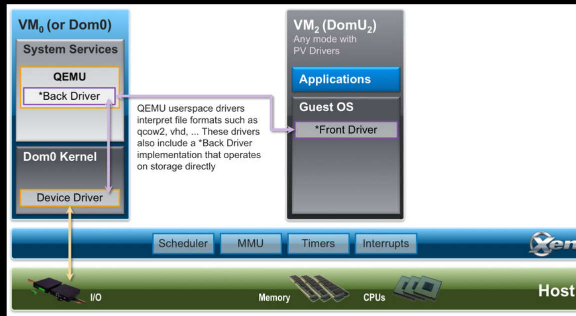
Main concepts

- Syscall virtualization (hypercall)
- Interrupt virtualization (event-channel)
- CPU virtualization (vCPU scheduling)
- Memory virtualization
- I/O virtualization

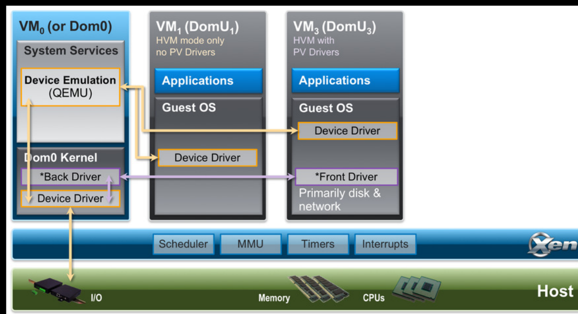
IO virtualization (split-driver approach, PV domUs)



IO virtualization (split-driver approach, PV domUs)



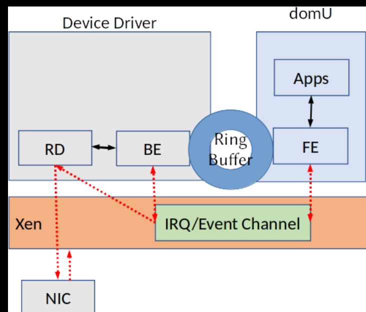
IO virtualization (emulation approach, HVM domUs)



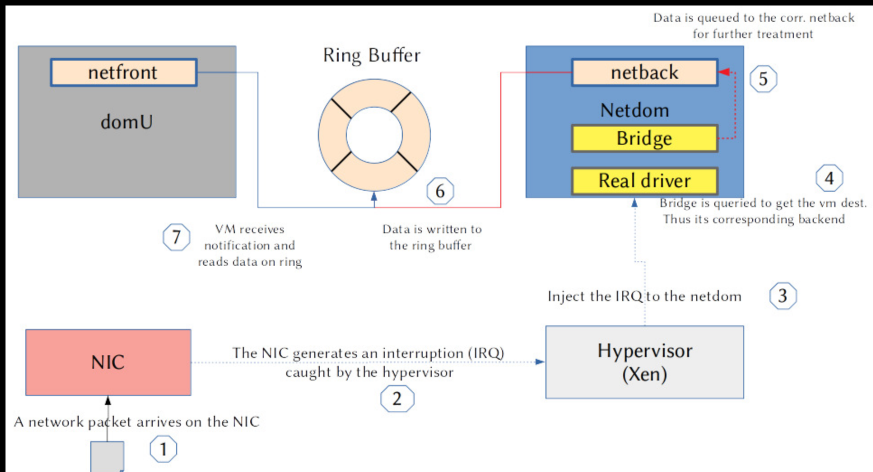
IO virtualization (split-driver approach)

Basic idea

In each VM, a netdevice frontend is configured to exchange requests with a backend inside the driver domain (dom0). They communicate via shared memory regions (ring buffers) and notification mechanisms (event channels) provided by Xen.



IO virtualization (split-driver approach)



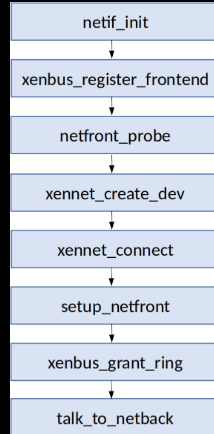
IO virtualization (split-driver approach)

Initialization

Each time a VM is launched, the domain which manages the NIC (called Netdom) is registered in Xenstore. Then a ring buffer (shared memory) is initialized between the frontend driver inside the VM and a backend driver inside the driver domain.

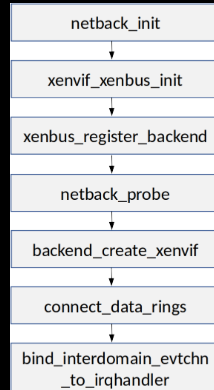
Sending path - Frontend

A number of queues netfront-queues are created at VM boot time. For each queue, receive buffers, ring buffers for communication with the backend are initialized, tx and rx irq handlers are registered with their corresponding #irq. Ring buffers grant references and #irq are then communicated to the driver domain which initializes a virtual interface (vif) based on these informations.



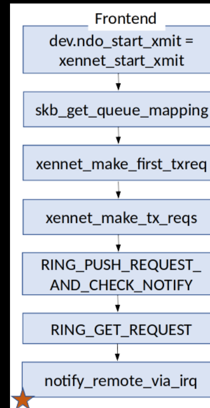
Sending path - Backend

The backend domain registers a new virtual interface vifdomid.netdomid (e.g. vif1.0). Then, it reads data relative to frontend number of queues, ring buffer grant references, tx and rx irqs from Xenstore. It maps the ring buffers for each queue in his memory address space and sets the handler for each corresponding irqs `xenvif_tx_interrupt`.



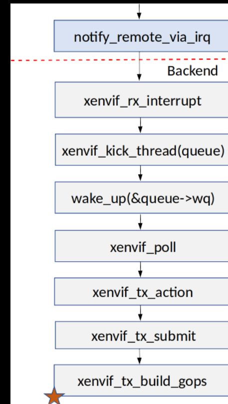
Sending path (VM⇒NIC)

As a real driver, when a packet arrives at the frontend to be sent, the `ndo_start_xmit` function registered at the initialization is called. For the frontend, this is `xennet_start_xmit`. It chooses a queue to be used for processing, then constructs transmit requests, pushes them to the corresponding ring buffer and sets the shared ring `req_prod` counter. If `req_prod <= req_cons` (before update), then the corresponding backend is notified via the `tx_irq`, otherwise it means the backend is still processing requests, so no need for notification.



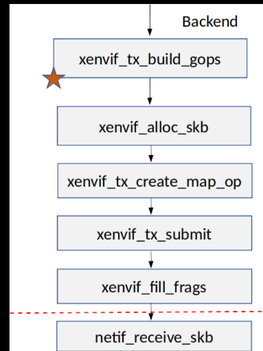
Sending path (VM$\Rightarrow$NIC)

Once the interrupt is acknowledged, the handler `xenvif_rx_interrupt` is launched. It then starts the kthread associated with the corresponding queue, which upon started begins a polling process with `xenvif_poll`. It retrieves request information (grant references, etc) from the corresponding ring buffer and applies the grants (done by the hypervisor).



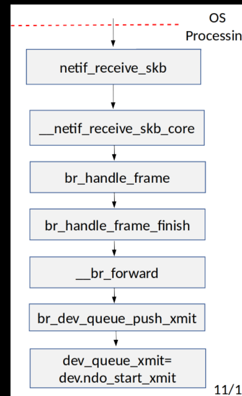
Sending Path (VM⇒NIC)

If enough scheduling credit time, an `sk_buff` structure is created and the header is filled in respect with the information retrieved from the ring. Then, the skb⇒dev attribute is set to the name of the corresponding vif (e.g. vif1.0) in `xenvif_tx_submit`. The remaining fragments of the skb are attached to the current `sk_buff` structure and the callback `xenvif_zerocopy_callback` pushes a `XEN_NET_RSP_OKAY` to the ring buffer to notify that the corresponding request has been correctly processed (the shared `req_cons` counter is updated). The `skb_buff` is then passed to `netif_receive_skb` (as the normal process).



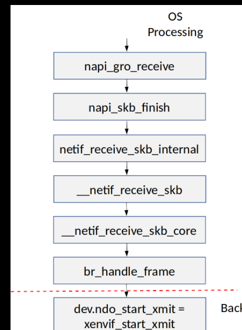
Send Path (VM⇒NIC)

The skb structure is passed to `br_handle_frame` which calls the `rx_handler` registered for skb⇒dev. Depending on the type of networking used bridging/NAT and the protocol of the skb, the latter is subject to several transformation (passing all the protocol layers). The final skb is passed to `__br_forward` which sends skb⇒dev to the real device attached to the vif (e.g e1000) and enqueued in the tx queue of the corresponding dev. The `.ndo_start_xmit` of the real device driver is then called to transmit the packet e.g `e1000_xmit_frame` (for e1000).



Receive Path (NIC⇒VM) - Real driver

Upon receiving a packet, the NIC generates an IRQ which is captured by the hypervisor and forwarded to the driver domain (dom0). The handler of the interrupt (`NET_RX_SOFTIRQ`) is specific for each driver. Here, we focus on the e1000 driver (`e1000_intr`). The latter first verifies if the driver works correctly by running a test function `e1000_info`. If everything is okay, it then schedules a `watchdog_task` which runs ever $2 \times 100 jiffies = 2s$, and calls the `__napi_schedule` routine. Otherwise, it restarts the driver `e1000_down()` followed by `e1000_up()`.

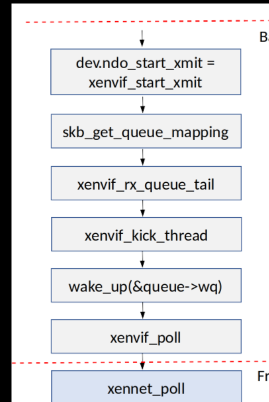


Receive Path (NIC⇒VM) - Real driver

For each processed packet, information about the skb is reconstructed, then skb passes through the protocol layers to set skb⇒dev (for the destination). This is done by `br_handle_frame` (for the bridging virtualization approach) with the same mechanism as the sending path. Once skb⇒dev is set, its .ndo_start_xmit is called (`xenvif_start_xmit` routine).

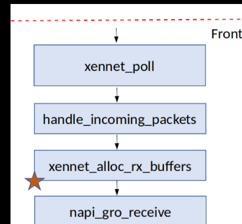
Receive Path (NIC⇒VM) - Backend

`xenvif_start_xmit` first chooses a queue for processing this skb. If no queue is set, the packet is dropped (`NET_RX_DROP` returned). Once a queue is chosen, the skb is enqueued in the corresponding rx queue. The kthread associated to the queue is then launched and starts the polling process. For each element in the queue, it constructs a request and pushes to the corresponding ring buffer, updates the shared `rsp_prod` counter and notifies the frontend.



Receive Path (NIC⇒VM) - Frontend

Once the frontend acknowledges the interrupt, the handler of the corresponding interrupt is launched `xennet_poll` on the corresponding queue. It retrieves information about the packet on the corresponding ring buffer, then prepares rx buffers to be used for the flipping process. Once the rx buffers are ready, it copies the grants (ownership grants) on the ring buffer and both the frontend and backend request the hypervisor to apply those grants. Once this is achieved, the skb is now ready to be used by the frontend. It now processes the skb as in a native environment.



Main concepts

- Syscall virtualization (hypercall)
- Interrupt virtualization (event-channel)
- CPU virtualization (vCPU scheduling)
- Memory virtualization
- ~~I/O virtualization~~

Main concepts

- Next class
- Read the following books
  - Hardware and Software Support for Virtualization [Edouard Bugnion, Jason Nieh, Dan Tsafrir - 2017]
  - The Definitive Guide to the Xen Hypervisor [David Chisnall - 2007]
- Read the following research papers for the project
  - Dune: safe user-level access to privileged CPU features [OSDI 2012]
  - Flexible Page-level Memory Access Monitoring Based on Virtualization Hardware [VEE 2017]