CS 524 A

Introduction to Cloud Computing

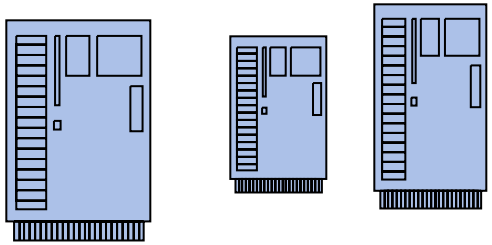
Lecture 4: Virtual Machines Part II

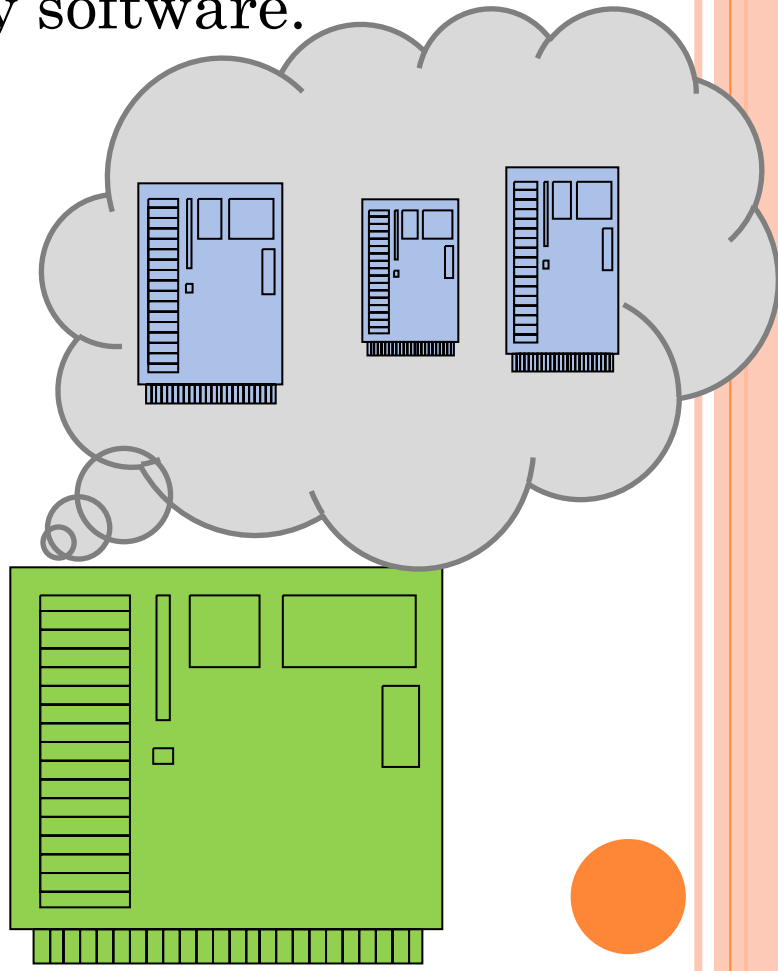
Outline

- History and motivation
- Mechanisms to achieve virtualization and the obstacles in their ways
- The *Popek and Goldberg* requirements
- The hypervisor architecture (Xen and VMware case studies)
- Non-virtualizable CPUs: the *x86* case study
  - Consequence: need for instruction emulation or new instructions
  - Rewrite *vs. paravirtualization*
  - Hardware-assisted virtualization
- Virtual memory challenges and solutions
  - Shadow page tables
  - Nested paging
- I/O challenges and solutions
  - I/O MMU
  - Case study: Xen approach to I/O virtualization
- Security
- New hypervisor technologies in development
- Containers
- Some open problems

What do we want to achieve with VMs?

For one thing, replace several machines with (a bigger) one without changing any software.



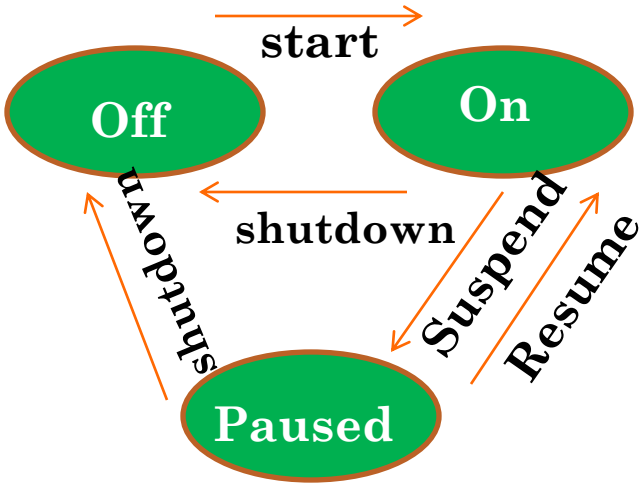


IBM did that with VM/370 in 1972

Why do that?

- Save the cost (space, energy, personnel) of running several machines in place of one—a **green** aspect, too!
- Use the (otherwise wasted) CPU power
- *Clone* servers (for example, for debugging) at low cost
- *Migrate* a machine (for example, when the load increases) at low cost (e.g., with a memory stick!)
- *Isolate* a machine—a server for a specific purpose (such as security)—without buying new hardware

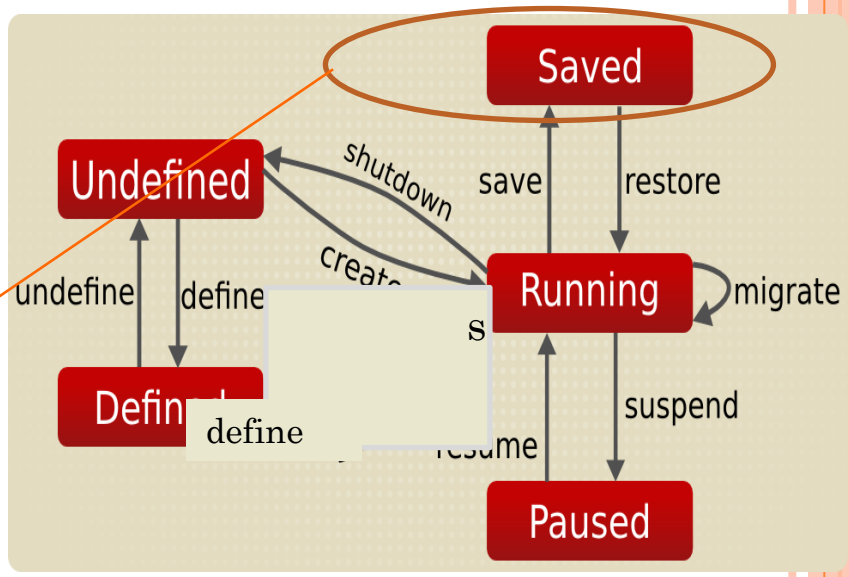
Case study: A VM Lifecycle



Physical

Virtual

after
http://wiki.libvirt.org/page/VM_lifecycle#Virtual_Machine_Lifecycle



Makes a big difference:
1) Create an appliance image
2) *Migrate* the machine

So why not run a virtual machine (the operating system and all) as a user process?

Because

- A process is aware of other processes (by design)— and so isolation fails
- A user process runs in the *user* mode and therefore its code cannot contain privileged instructions, but the operating system **must execute** these!

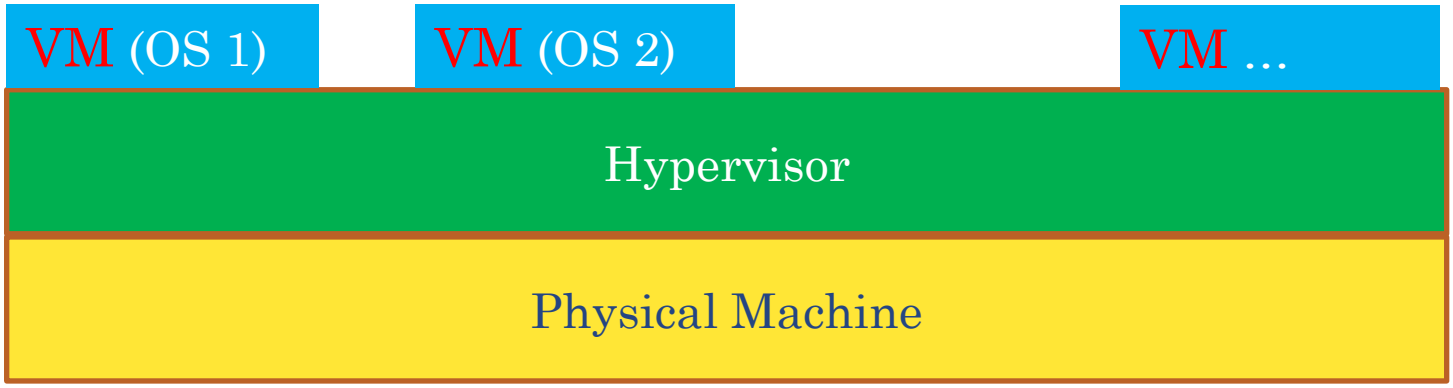
Popek and Goldberg Virtualization Requirements

Hence the need to *virtualize* a CPU, subject to the *Popek and Goldberg Virtualization Requirements* set forth in Gerald J. Popek and Robert P. Goldberg (1974). "Formal Requirements for Virtualizable Third Generation Architectures". *Communications of the ACM* **17** (7): 412 –421

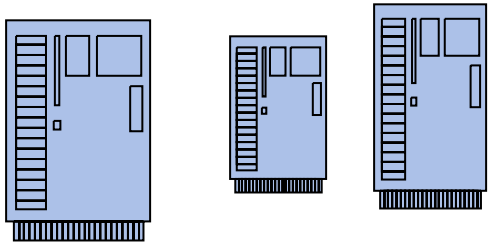
These requirements have been met in the third-generation CPUs (IBM 360, Honewell 6000, DEC PDP-10, etc., but then they were *not* met (not surprising?) for the Intel processor developed for the IBM PC)

Hypervisors

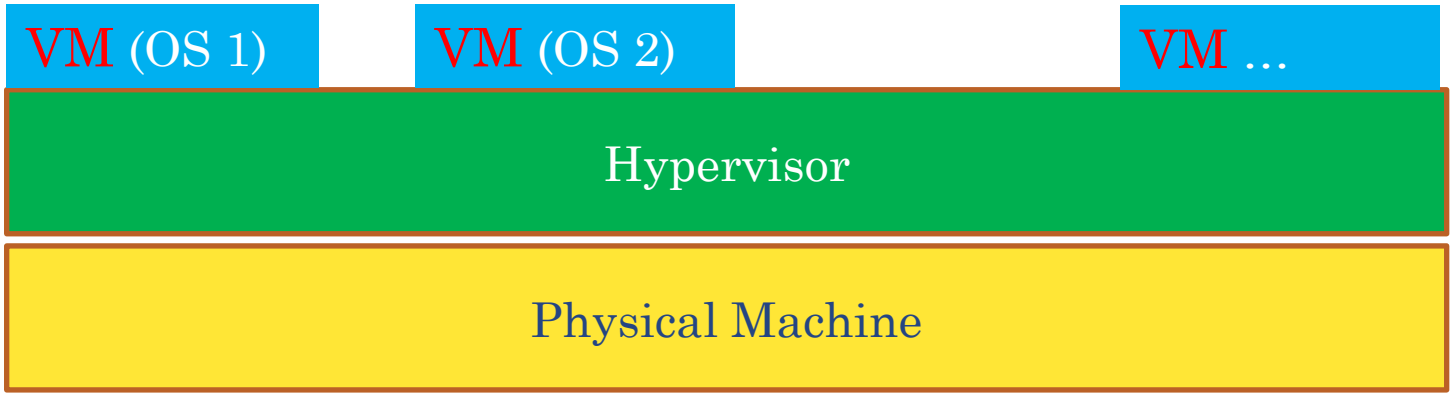
A Hypervisor (or *Virtual Machine Monitor [VMM]*) is *software charged with creating and maintaining virtual machines*. It runs *below* the OS level (of course!).



Note on reliability

This  is less reliable than

that



because a hypervisor is much simpler than an OS (the number of instructions is two orders of magnitude smaller)!

The three major requirements that hypervisors must meet

- *Equivalence* (or *fidelity*): A program run on a VM must behave exactly as though it runs on a physical machine
- *Efficiency:* A statistically dominant [sic!] fraction of machine instructions must be executed without the hypervisor's intervention.
- *Resource control:* The hypervisor must be in complete control of the resources:
  - No program running on a VM may access resources other than those specifically allocated to it
  - A hypervisor may take away from a virtual machine any resource allocated to it

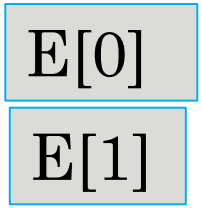
The formal Model

State of memory *PC* *Relocation-base register [MMU]*

State $S = (E, M, P, R)$

CPU mode (user/system)

Stack for saving the (M, P, R) triplets



Instructions and Traps

The instruction i is a mapping

$$i: \mathcal{E} \rightarrow \mathcal{E}$$

$$i(E_1, M_1, P_1, R_1) = (E_2, M_2, P_2, R_2).$$

The instruction is said to *trap*, if the storage is left unchanged, except for the top of the stack:

$$E_2[0] = (M_1, P_1, R_1); \text{ (saved status)}$$

$$(M_2, P_2, R_2) = E_2[1].$$

An instruction is said to *memory-trap* if it traps when accessing out-of-bound memory.

An instruction is called

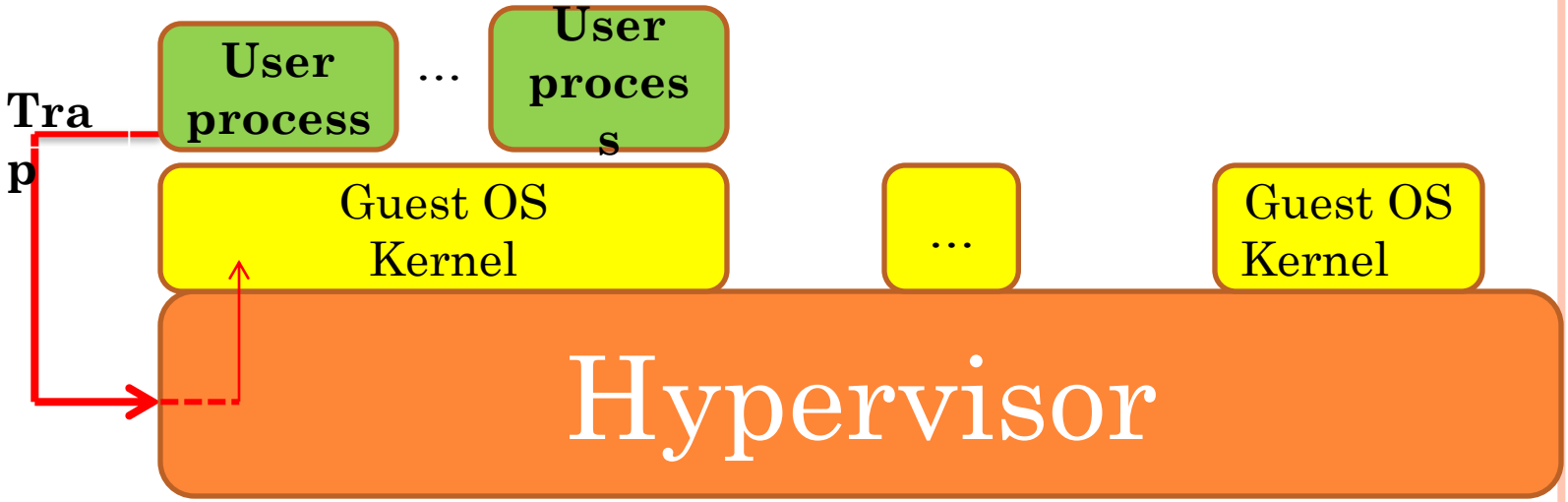
- ***Privileged***, if it does not memory-trap, but traps only when executed in user mode;
- ***Control-sensitive***, if it attempts to change the amount of memory available or affects the CPU mode
- ***Behavior-sensitive,*** if its execution depends on the value of controls set by a control-sensitive instruction
- ***Sensitive,*** if it is either behavior-sensitive or control sensitive. Otherwise, it is called ***Innocuous***.

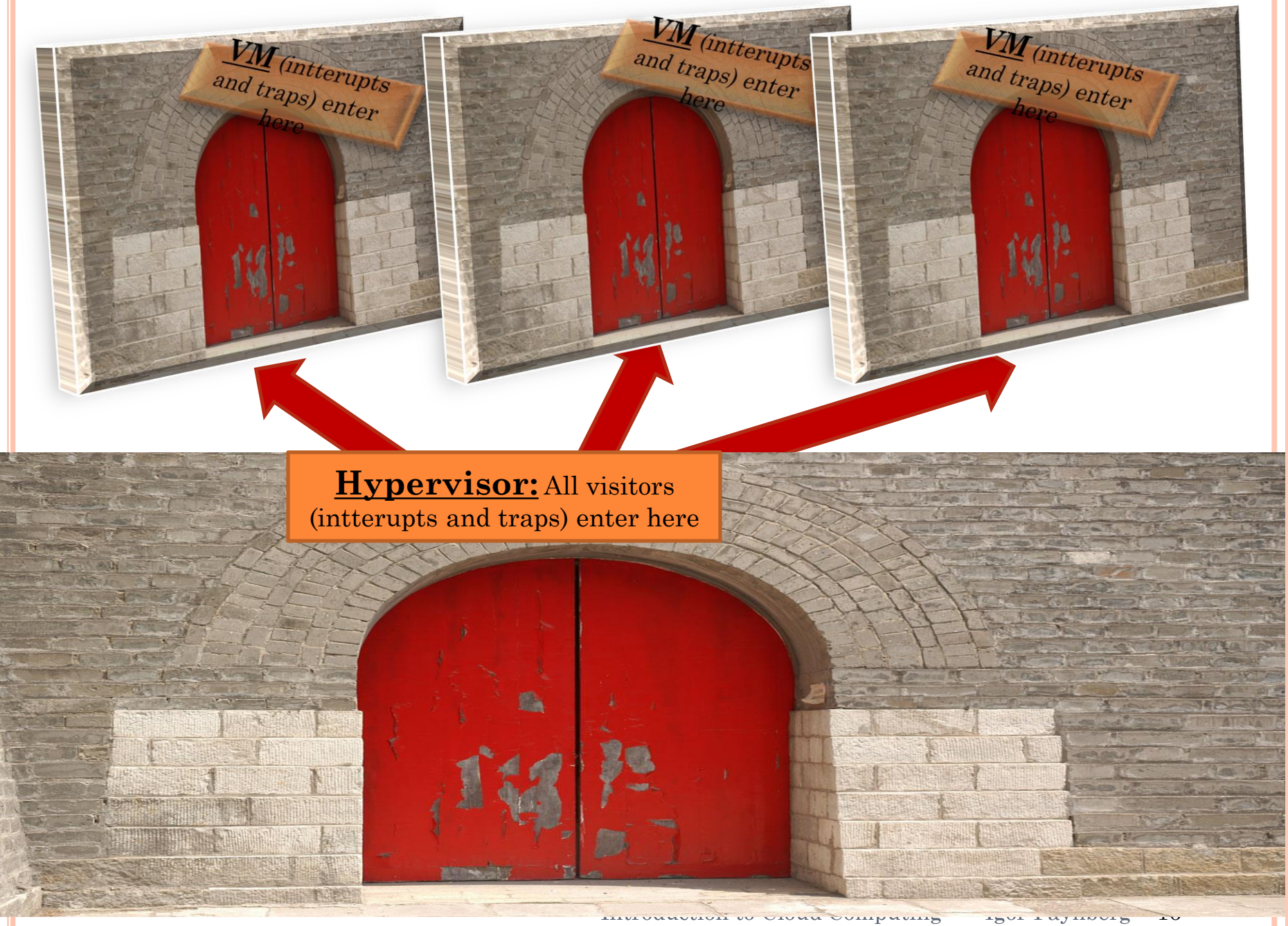
Examples (original) of a behavior-sensitive instruction

- IBM 360 *Load Real Address (LRA)*
- PDP-11/45 *Move From Previous Instruction Space (MFPI)*
- …more in the homework!

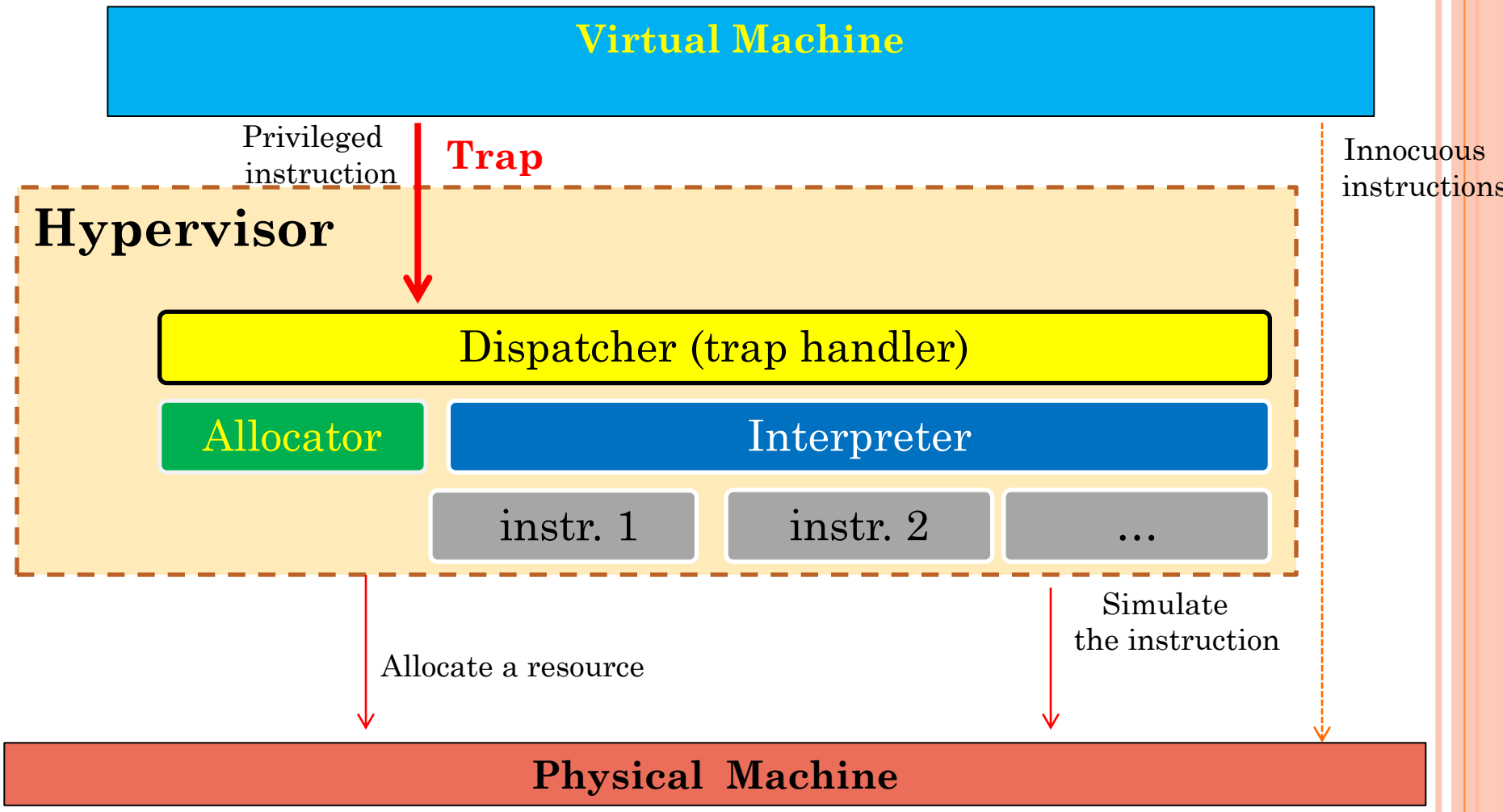
How it works

- In short, instructions are
  - *Control-sensitive,* if they can change the configuration of resources (e.g., contents of segment relocation registers)
  - *Behavior-sensitive,* if their result depends on the configuration of *resources*
- If all sensitive instructions are privileged, a hypervisor can be successfully implemented



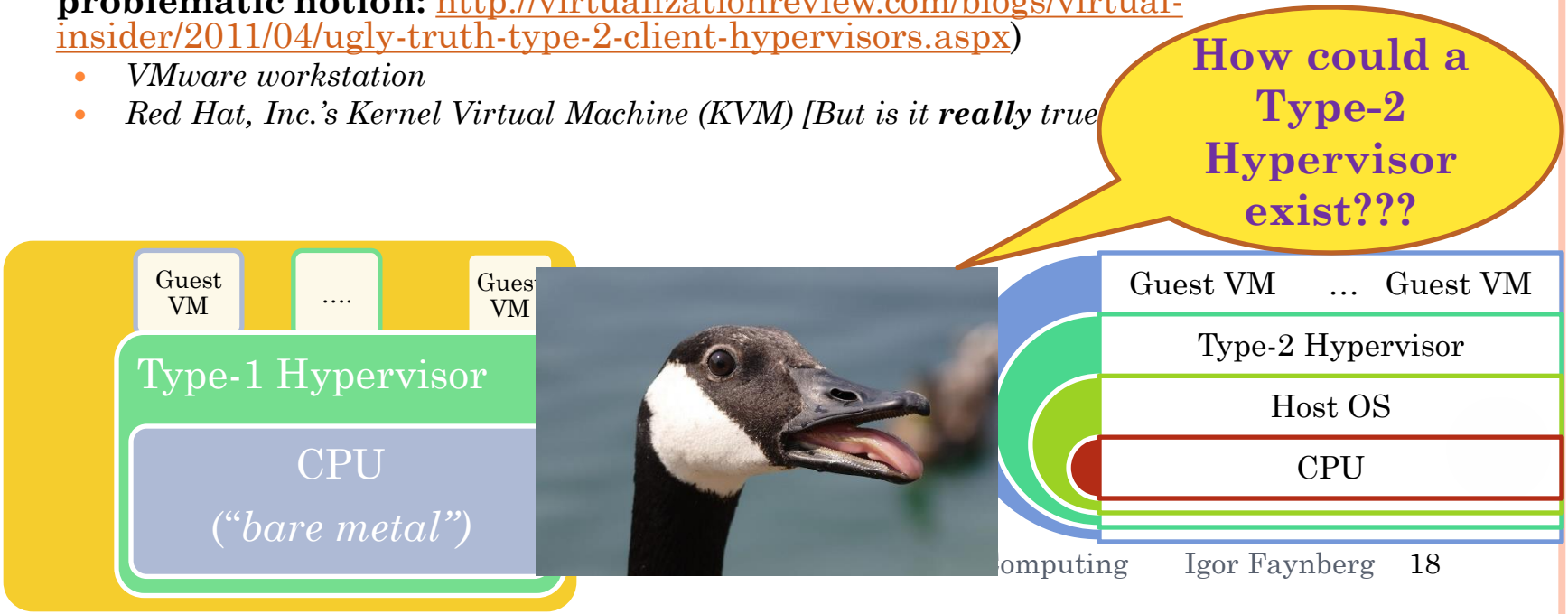

VM (intterupts and traps) enter here
VM (intterupts and traps) enter here
VM (intterupts and traps) enter here
Hypervisor: All visitors (intterupts and traps) enter here

More detail on execution (after *Popek and Goldberg*)



But where *exactly* does a Hypervisor run?

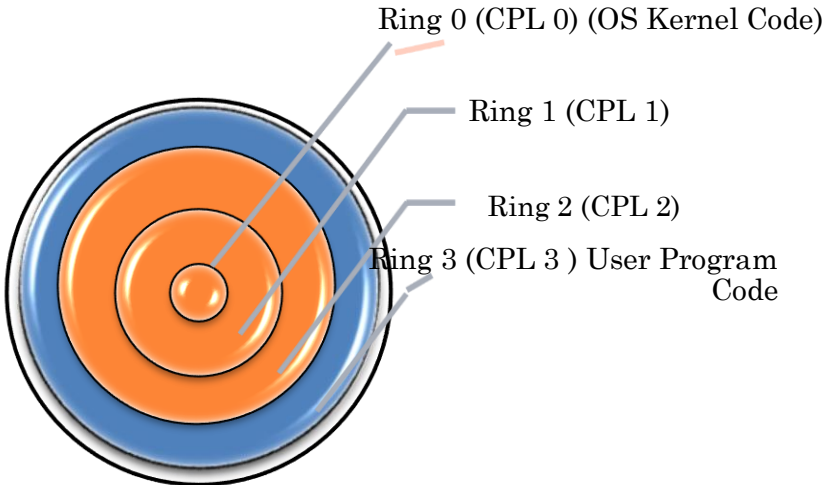
- *A Type-1 Hypervisor* (or a *bare-metal Hypervisor*) runs directly on the machine:
  - The IBM *Control Program/Conversation Monitor System (CP/CPM)* which introduced S/360 and culminated in VM/370 in1972
  - *VMware ESX* and *ESX*i (http://www.vmware.com/files/pdf/VMware-ESX-and-VMware-ESXi-DS-EN.pdf)
  - Microsoft **Hyper-V** (http://www.microsoft.com/en-us/server-cloud/hyper-v-server/default.aspx)
  - Citrix Xen Server (http://www.citrix.com/English/ps2/products/product.asp?contentID=683148)
- *A Type-2 Hypervisor* runs on top of a *host* operating system (a **somewhat problematic notion:** http://virtualizationreview.com/blogs/virtual-insider/2011/04/ugly-truth-type-2-client-hypervisors.aspx)
  - *VMware workstation*
  - *Red Hat, Inc.'s Kernel Virtual Machine (KVM) [But is it **really** true*



x86

- (The family of CPUs based on Intel 8086 CPU architecture)
- Historically, 80386 was designed to provide virtualization for the 16-bit 8086 mode (to support DOS applications); no one expected the 32-bit 80386 to have a need for virtualizaiton!
- Specifically, *Intel Pentium* has 17 instructions that are **not** virtualizable according to the *Popek and Goldberg' requirements*

The x-86 Privilege Level Rings



- Four Code Privilege Levels (CPLs), from 0 to 3, in the descending privilege order.
- The code running at a higher privilege has access to the resources available to the code at lower privilege
- The practice of the two-level operating systems has been to execute the system code at CPL 0 and the user code at CPL 3.
- Now, the hypervisor can be put in Ring 0, but guest OSs in Ring 1—still keeping user processes in Ring 3.

CLASSES OF NON-VIRTUALIZABLE *X.86* INSTRUCTIONS

1. Instructions that read and write the values of the segment table registers as well as the interrupt table register.

   Problem: the processor has only one register for each of these, which means that they need to be replicated for each virtual machine with all the overhead to effect this.

2. Instructions that copy parts of the STATUS register into either general registers or memory (including the stack).

   Problem: a virtual machine can find out that it is virtual!

3. Instructions that depend on the memory protection and address relocation mechanisms.

   Problem: These instructions need to ensure that the current privilege level and the requested privilege level are both greater than the privilege level of a segment, but a virtual machine does not execute at CPL 0.

4. Instructions that are supposed to execute only in system mode and do not trap when executed in the user mode

   Problem: The hypervisor does not get control and so system functions are not performed.

Let us look at some example instructions

(https://www.intel.com/content/dam/www/public/us/en/documents/manuals/64-ia-32-architectures-software-developer-instruction-set-reference-manual-325383.pdf)

- *Store Interrupt Descriptor Table Register (SIDT)* allows condition registers to be set but has no *load* counterpart (thus no way to restore old values)
- *Load Access Rights Byte (*LAR) : loads access rights from segment descriptor into a general purpose register
- Load Segment Limit (LSL) : loads *unscrambled* segment limit from segment descriptor into a general purpose register

Because these do not trap, there is no way for a hypervisor to gain control (let alone to maintain values specific to each guest VM)

What can be done then?

- *Binary rewriting* (practiced by *VMWare*):
  - *Inspect* the instruction stream (block-by-block) to identify a problematic instruction
  - *Rewrite* a problematic instructions to trap into the hypervisor where it will get emulated
- *Paravirtualization* (practiced by *Xen*):

  Still rewrite the OS kernel to replace problematic instructions with *hypercalls* (i.e., defined calls to the hypervisor), but do that **compilaton before running the program.**

  Hadware-assisted virtualization

Hardware-assisted Virtualization

Additional set of capabilities

- *Intel Virtualization Technology (IVT)*
  - *adds a new* ***mode, VMX,*** to the processor
  - Supplies extra instructions for starting, saving, and stopping a VM, and saving CPU state
  - Indicates which interrupt should be passed to a specific VM for processing
- *Advanced Micro Devices, Inc. (AMD) Pacifica (*now *AMD V)*
  - Augments IVT with an extra level of (VM-specific) page translation
  - Provides the *Device Exclusive Vector Interface* (to map into VM-specific device address space)

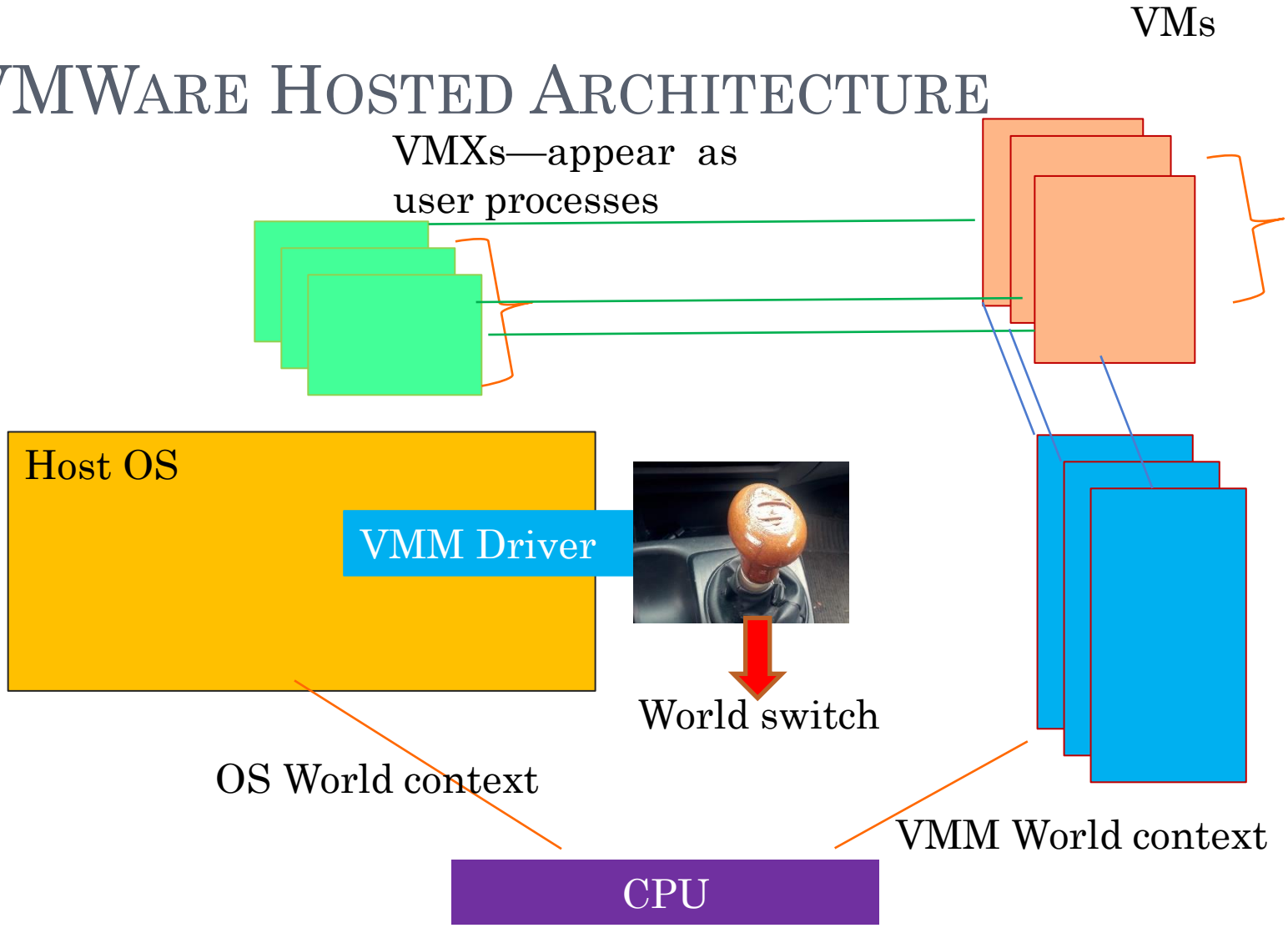
With these capabilities, the hypervisor runs in the *Ring -1 now*, invisible to the OS Kernel still running in *Ring 0*

Case study: VMware Workstation

VMware is a "type-2" hypervisor (*but is it*?) that has two modules:

- Virtual Machine Monitor (VMM) (one for each VM)
  - actually *runs* the virtual machine (using combination of *emulation*—actually executing the VM instructions rather than giving them to hardware and *direct execution*
  - emulates the hardware devices and multiplexes the actual physical platform among all machines
  - presents the virtualized *front end* to the virtual machines, while the actual *back end* is managed by the host OS
- Virtual Machine eXtension (VMX) (a user-space program, which provides the user interface to and manages a VM)
- VMM driver, essentially a *root kit* installed as a driver in the host OS, which performs the ***world switch***

VMWare Hosted Architecture



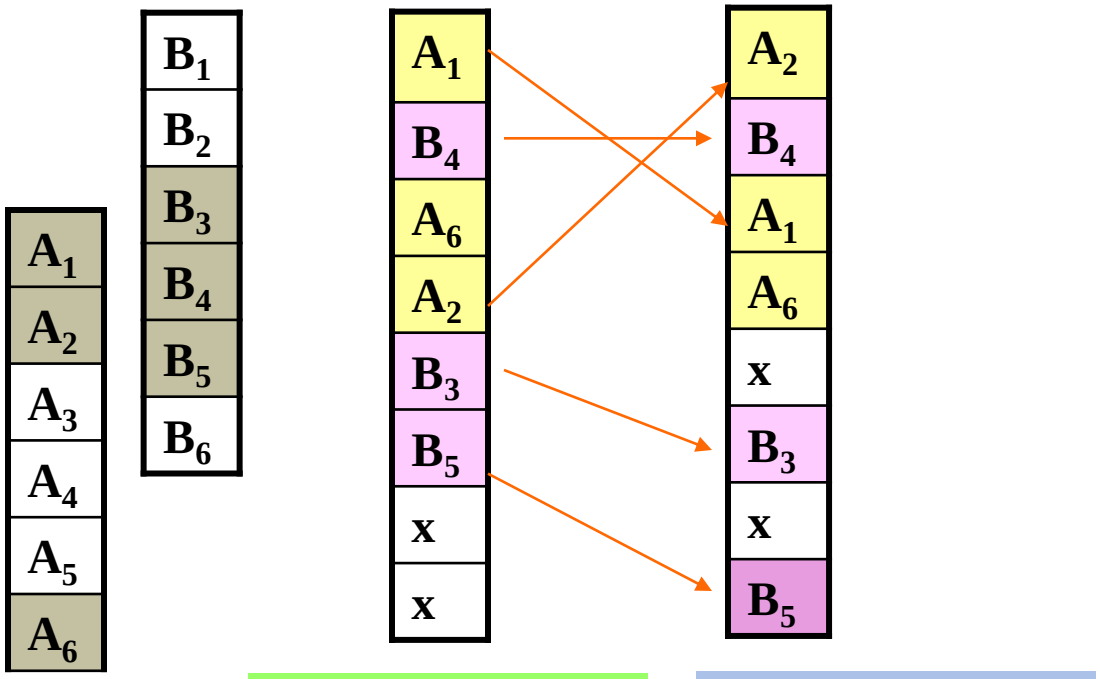
An observation

At any given time, either

- the VMM owns the CPU (i.e., catches all interrupts, etc.) or
- the Host OS owns the CPU

Consequently, the VMM does **not** exactly run on top of the OS

DEALING WITH THE VM'S PAGE TABLES: *SHADOW TABLES*



VM page tables

VM "Physical" Memory

Shadow page table in the Hypervisor

Alternative: Nested Page Tables (AMD) = Extended Page Tables (Intel)

- The MMU hardware supports nesting:

guest virtual address -->

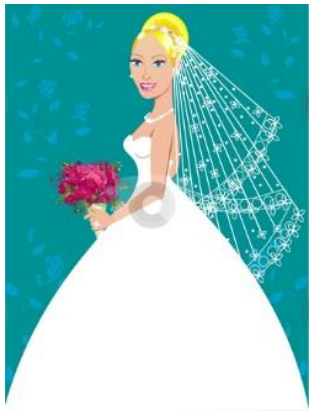
guest "physical" address -->

machine physical address

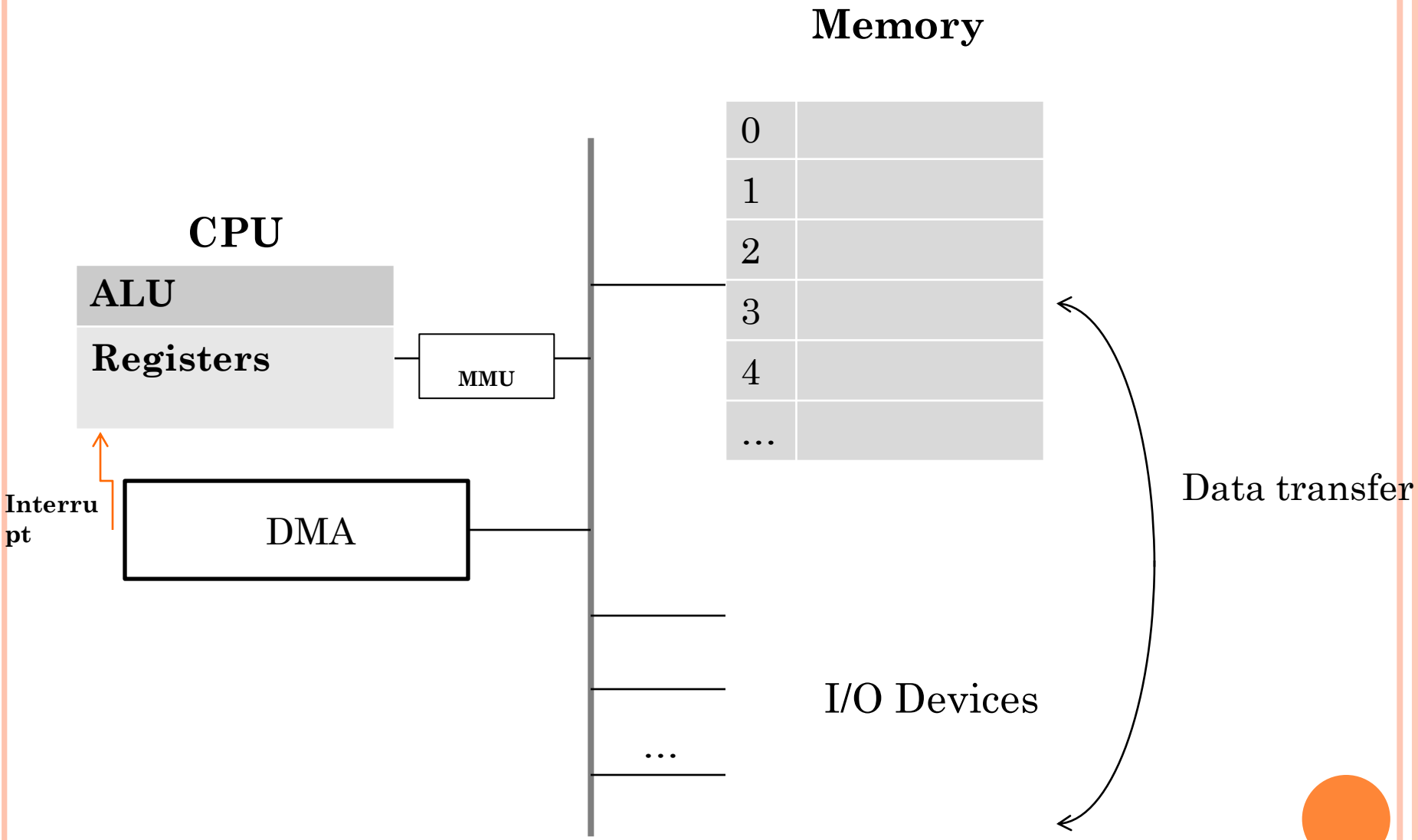
- The hypervisor still has to maintain the tables, but the CPU "walks tables" and translates addresses itself

I/O Virtualization Problems

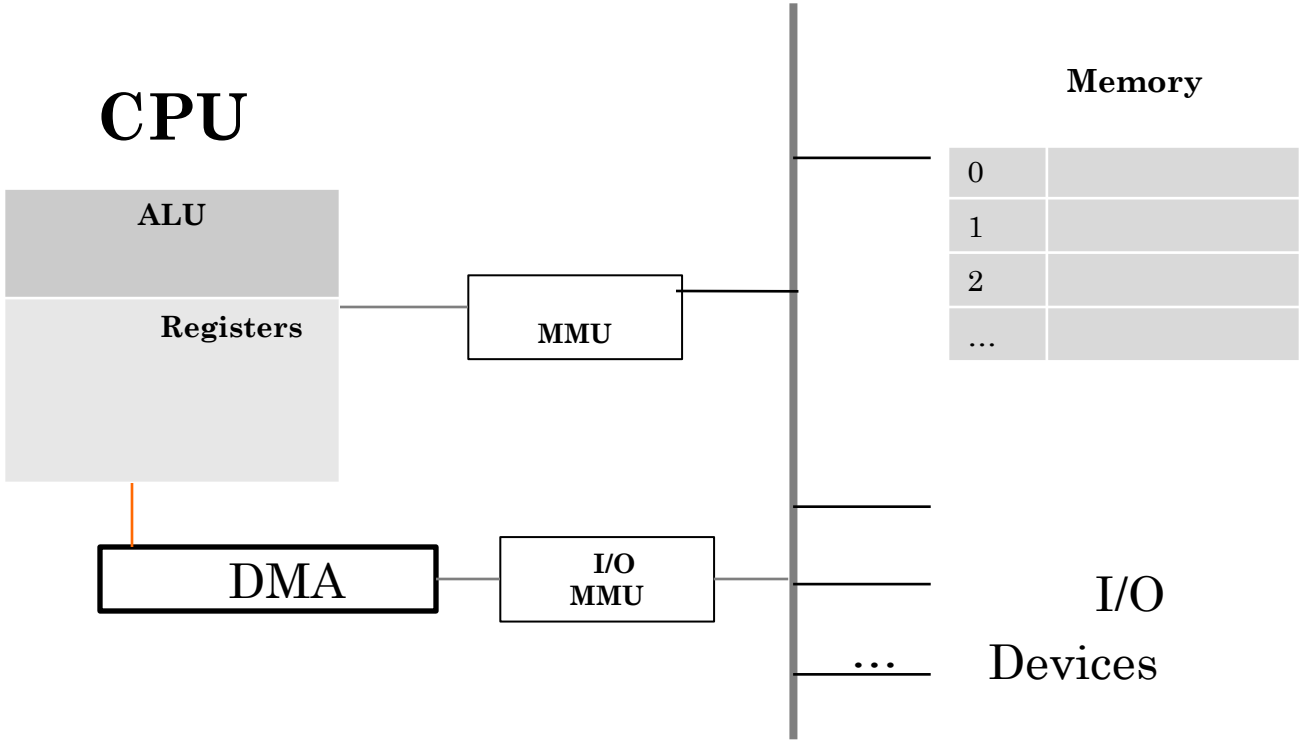
- A virtual machine may not have direct access to an I/O device since each I/O device is "married" to its machine.
- Therefore a hypervisor must **either** directly emulate a device or use *paravirtualized* drivers



And what about DMA, which needs real memory access?



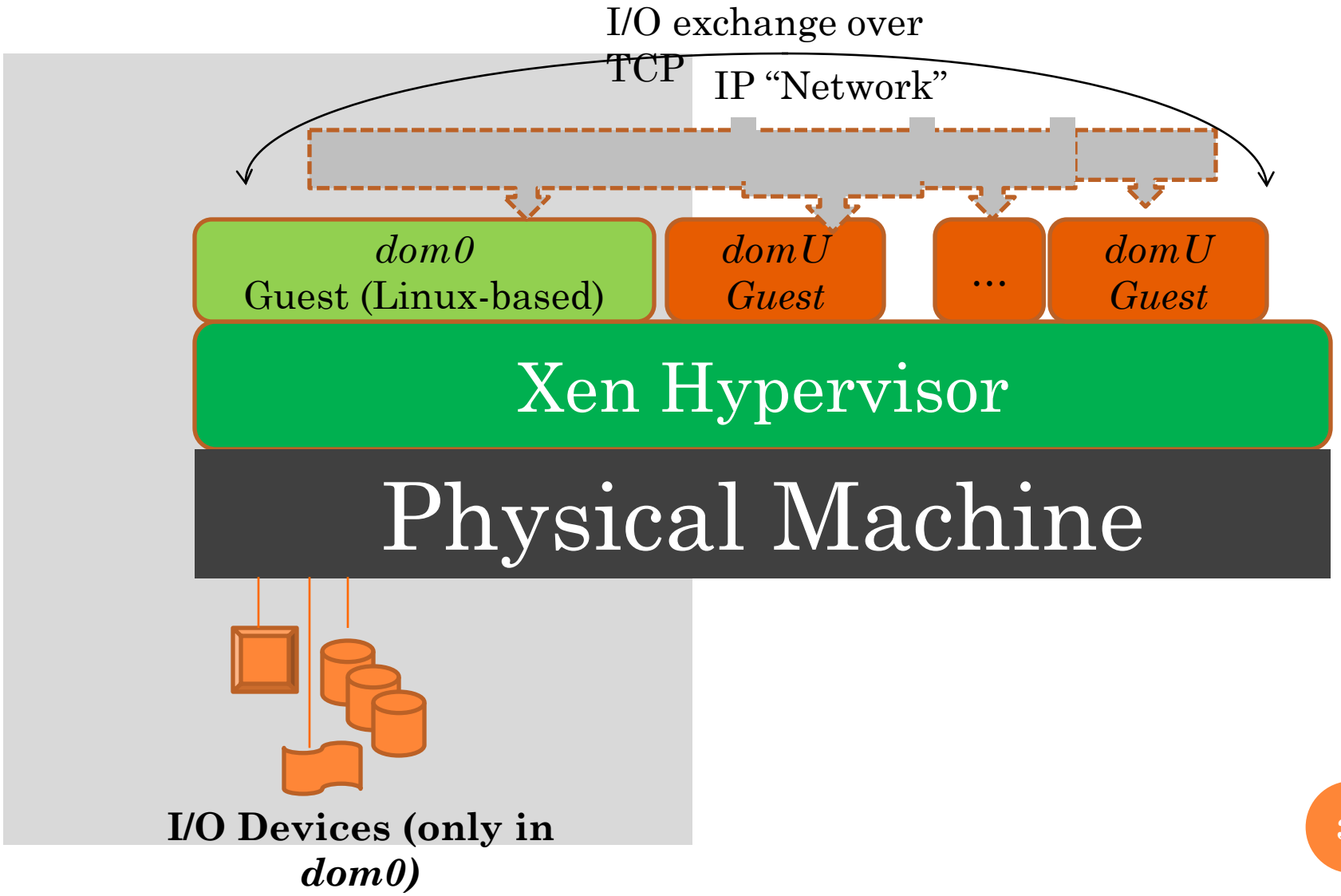
A solution: I/O MMU



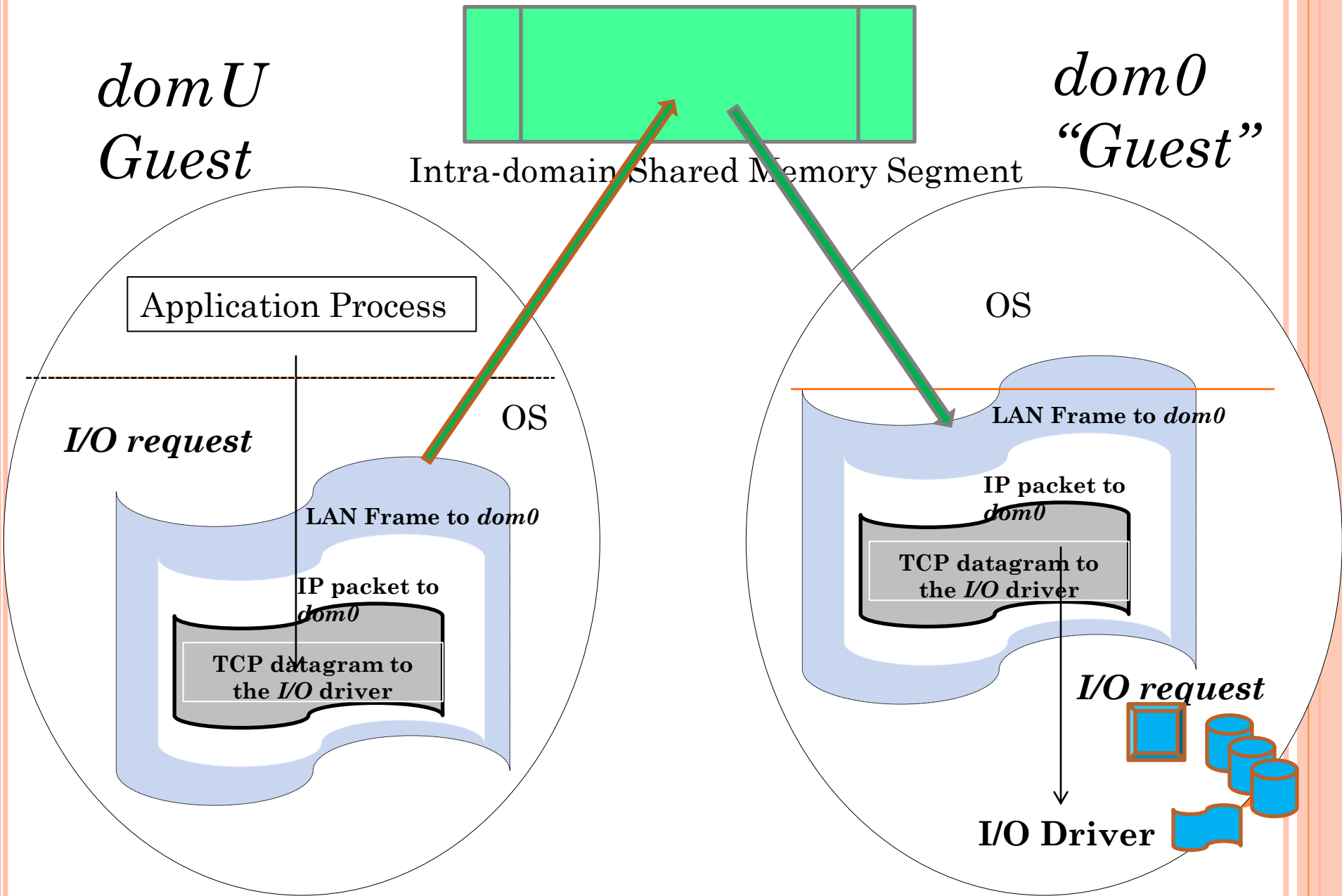
A Case Study in Elegance: *XEN* domains

- *XEN* runs *guest systems* in the environments called *domains*
- There are two domains: *Domain 0* and Domain *U*
- *Domain 0* hosts one guest system (typically, *Linux*) that handles all device drivers
  - *XEN* does not handle device drivers by itself
  - Most devices cannot be handled by more than one operating system
  - The guest system runs here at a higher level of privilege than **any other** *Xen* guest
- *Domain U* hosts all other guest systems, which *think* that they access devices over the network interface
- *XEN* intercepts the I/O requests and routes them to the guest in *Domain 0*
- *Domain 0* handles the requests

Virtual Machine I/O support in Xen



Xen network I/O optimization using shared memory



Security Outline

- NIST and the Federal Information Processing Standard [**FIPS**] recommendations on hypervisor security
- A bit more on I/O (BIOS)
- Trust and Trusted Platform Modules
- Hardware Security Modules
- Hypervisor introspection

NIST Recommendations on Hypervisor Security

- Install all updates to the hypervisor as they are released by the vendor… Centralized patch management solutions can also be used to administer updates.
- Restrict administrative access to the management interfaces of the hypervisor. Protect all management communication channels using a dedicated management network or the management network communications is authenticated and encrypted using FIPS 140-2 validated cryptographic modules.
- Synchronize the virtualized infrastructure to a trusted authoritative time server.
- Disconnect unused physical hardware from the host system… Disconnect unused [*Network Interface Controller*] NICs from any network.
- Disable all hypervisor services such as clipboard- or file-sharing between the guest OS and the host OS unless they are needed.
- Consider using introspection capabilities to monitor the security of each guest OS.
- Consider using introspection capabilities to monitor the security of activities occurring among guest OSs.
- Carefully monitor the hypervisor itself for signs of compromise. This includes using self-integrity monitoring capabilities that hypervisors may provide, as well as monitoring and analyzing hypervisor logs on an ongoing basis.

Basic Input/Output System (BIOS) adapters

- The booting process must be protected to ensure against root-kits and to comply with licenses.
- Without specialized hardware, hypervisors need to emulate the low-level firmware components, such as graphic- and network adapters such as *Basic Input/Output System (BIOS)* adapters.
- BIOS is responsible, among other things, for selecting the booting device and booting the system.
- BIOS is typically implemented as *firmware* on a hardware chip, and it provides access to the (physical) machine hardware configuration. **There is only one BIOS on a physical machine.**
- **There is only one BIOS on a physical machine**, and so its function must be replicated by a hypervisor for each virtual machine…

THE CONCEPT OF *TRUST*

"*Trust is the expectation that a device will behave in a particular manner for a specific purpose* "

A definition form the *Trusted Computing Group*

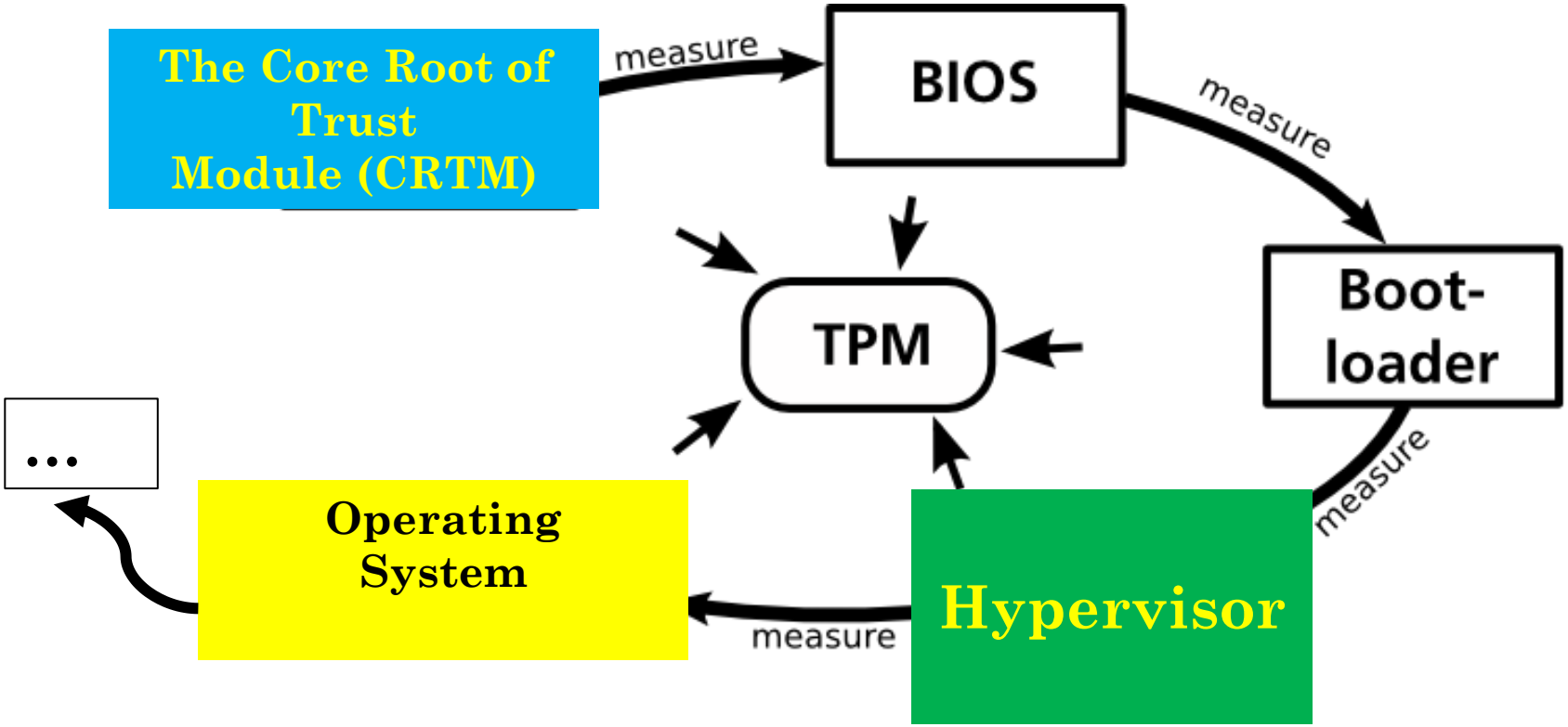
- The idea is to assess a device's (or the whole platform's) future behavior based on its current state
- The task is to obtain securely the current status of such a platform.
- A platform that is able to securely report its own state is referred to as "Trusted Platform".

The Trusted Platform Module (TPM)

A major difference from a *smartcard!*

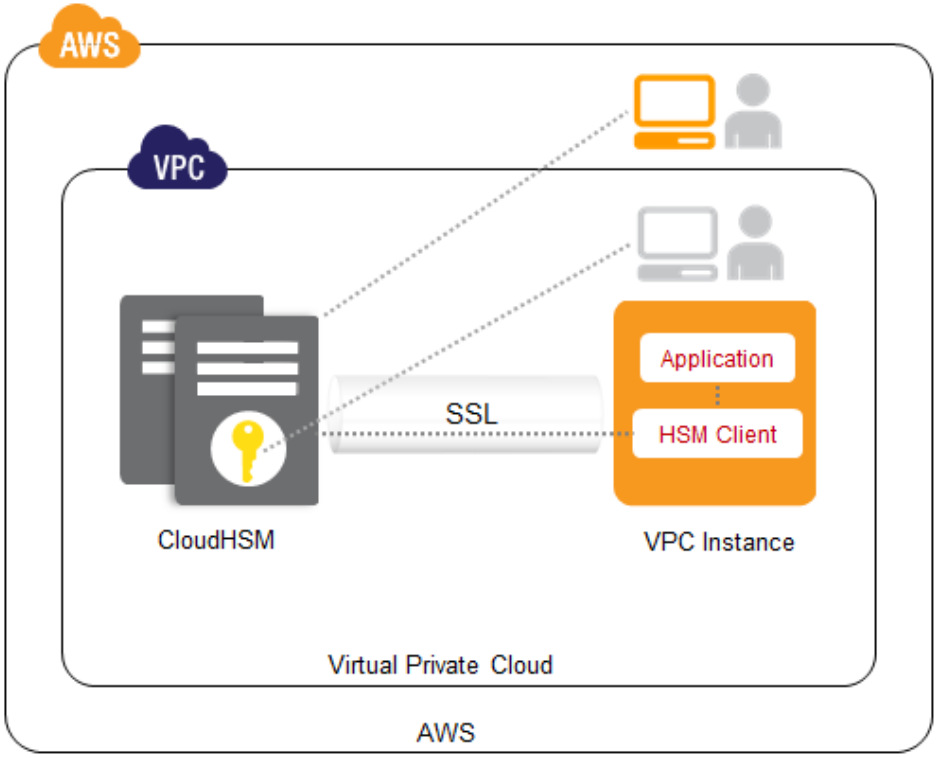
- A TPM is a *passive* hardware chip soldered to the motherboard: It just receives commands and data, performs the requested operation, and returns a result
- TPM provides volatile and non volatile memory, a true random number generator, a SHA-1 engine and an asymmetric key (e.g., RSA) engine
- A TPM enables *integrity measurements* and thus *trusted boot* and *attestation* thus allowing to develop the *chain of trust*.

The measurements process



After source: http://trust.f4.hs-hannover.de/

Hardware Security Module (HSM)



Source: Amazon AWS
https://aws.amazon.com/blogs/aws/aws-cloud-hsm-secure-key-storage-and-cryptographic-operations/

- Secure cryptographic key generation
- Secure cryptographic key storage and management
- Offloading application servers for complete asymmetric and symmetric

Hypervisor Introspection

All Virtual Machines live in glass houses!

Hypervisor

Source:
OddityMall.com

Introspection in Open Source

- Fairly straightforward in an emulated environment with QEMU [1], but considerably harder in the non-emulated environment.
  - A recent research publication [2] points to an *OpenStack* project, apparently still within the QEMU framework. Performance cited as "most CPU-intensive component."
  - [3] builds an intrusion detection framework using AI methods, but performance is unclear.
- In the non-emulated environment, apart from hypervisor's explicit copying VM snapshots to an appliance's memory (expensive and potentially dangerous), the only way to access memory maps and resources is by trapping into the hypervisor and executing at its level. Known solution: LibVMI
  - *LibVMI* (*http://libvmi.com/api/*) has no documentation yet. Described in [4] and extended to *Python* (*PyVMI*). Works with both KVM and XEN.
  - Performance remains a serious problem: much work to do on each API call. Example: *vmi_read_ksym* call "must handle reads around page boundaries, resolve the kernel symbol, translate the kernel symbol to a physical address, and perform the actual read from the hypervisor."
  - An application certainly needs to develop a complex optimization mechanism, which is likely to fail in some circumstances.

Notes on Introspection Deployment

- Opening up introspection API is considered dangerous. The API must be carefully protected—but can one be *too careful?*
- Hypervisor-based Introspection is expensive in terms of performance
- From a few operator sources: VMware introspection API is disabled in known deployments.
- *NEW* From ETSI NFV SEC: Introspection is a major problem in the way of Lawful Interception. (A new WI has been just created to harness new hardware technologies to enforce multi-layer access.)
- .

References (introspection)

[1] F. Bellard. (2005) *QEMU, a Fast and Portable Dynamic Translator. USENIX'05.*

[2] M. Baig, C. Fitzsimons, S. Balasubramanian, R. Sion, and D. Porter. (2014) *CloudFlow: Cloud-wide policy enforcement using fast VM introspection* . IEEE Conference on Cloud Engineering (IC2E 2014). http://www3.cs.stonybrook.edu/~porter/pubs/cloudflow.pdf

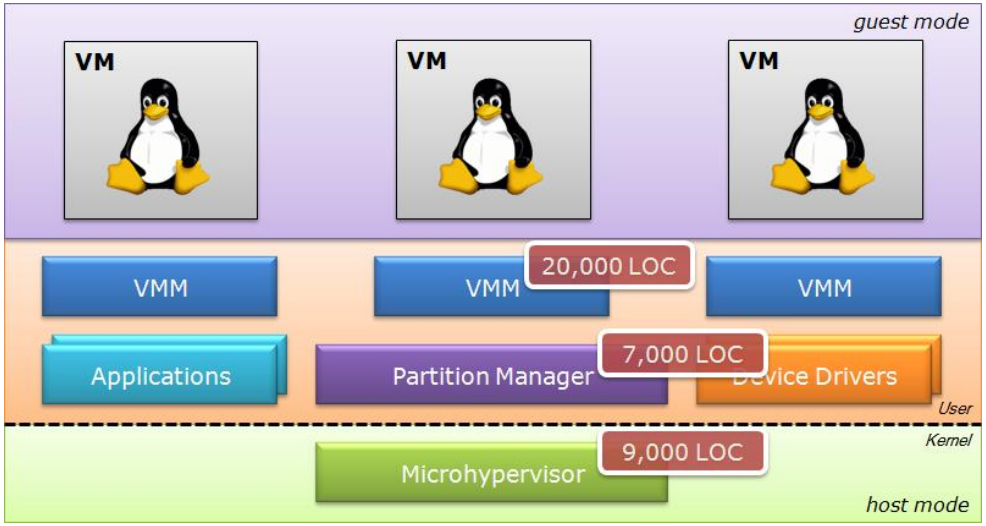
[3] S. Lee and F. Yu. (2014) *Securing KVM-Based Cloud Systems via Virtualization Introspection.* 47 Hawaii International Conference on Systems Sciences (HICSS). http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=6759220&tag=1.

[4] B. D. Payne. (2012) *Simplifying Virtual Machine Introspection Using LibVMI.* SANDIA REPORT SAND2012-7818 Unlimited Release Printed September 2012 Bryan D. Payne. http://prod.sandia.gov/techlib/access-control.cgi/2012/127818.pdf

New Hypervisor Technologies in Development

- NOVA microkernel hypervisor
  - Developed in Dresden Technical University (2006-2012)
  - Developed in Intel Labs after that
  - Available in open source: https://github.com/udosteinberg/NOVA
  - Most effective on Intel VT-x but runs in other configurations including QEMU
- *NoHype*
  - Under development in Princeton University
  - Based on a stripped version of XEN and a modified version of Linux
  - Dependent (presently) on the Intel VT-x

NOVA

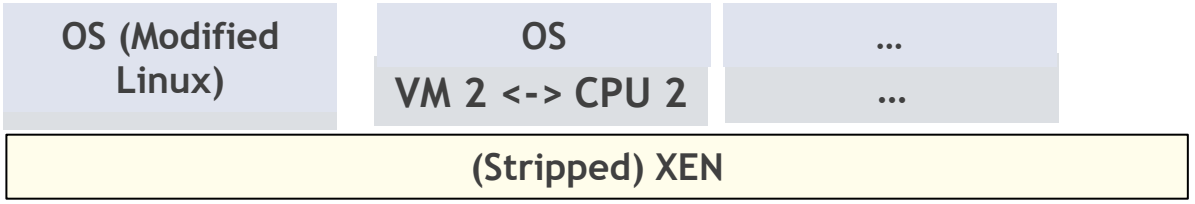


Vs.

- **over 100,000 LOC in XEN (not counting device drivers)**
- **over 200,000 LOC in KVM/Linux Kernel (not counting device drivers)**
- **about 200,000 LOC in VMware VMM**

- **A VM may compromise only its associated VMM**
- **Hypervisor resources are not exposed to user level**

NoHype



Hypervisor (stripped XEN 4.0)
- Always runs in layer -1
- Places a VM into its own core and sets the core's outer MMU to the limits of the VMs separate memory
- Intercepts any attempt to get to unassigned memory

Key ideas:
(i) pre-allocation of processor cores and memory resources
(ii) use of virtualized I/O devices
(iii) minor modifications to the guest OS to perform all system discovery during bootup, and
(iv) avoiding indirection by bringing the guest virtual machine in more direct contact with the underlying hardware

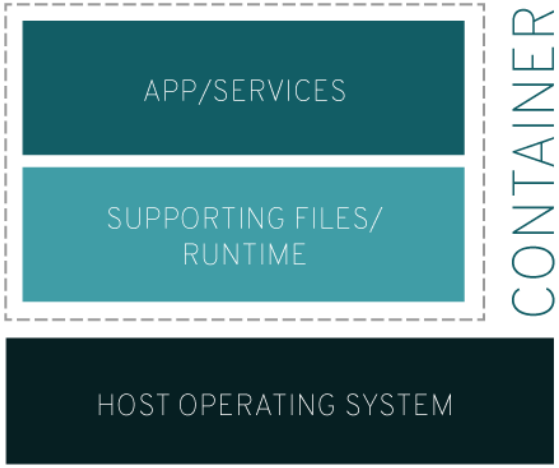
NoHype supports the model in which customers specify resource requirements ahead of time.

Containers

- The idea: instead of running an application as a *virtual appliance* (on a separate VM with its own OS) run it as a process that has been *isolated*.
- This type of isolation requires a new set of features.
- *Linux* has implemented those. (And since most servers run *Linux* anyway, all OSs in VMs are the same, anyway. Makes sense!)

Definition

- A Linux® container is a set of processes that are isolated from the rest of the system, running from a distinct image that provides all files necessary to support the processes. By providing an image that contains all of an application's dependencies, it is portable and consistent as it moves from development, to testing, and finally to production.



t.com/en/containers/whats-a-linux-container

More on "OS-level virtualization"

- Linux containers were not deemed secure before version 3.8, the *root* user of the guest system could run arbitrary code on the host system with root privileges
- There are other "OS-level virtualization" technologies on Linux such as OpenVZ and Linux-VServer, as well as those on other operating systems such as FreeBSD jails, AIXWorkload Partitions and Solaris Containers.

Conclusion: Some open problems in virtualization

- Software licensing
- I/O performance
- Security
- Availability of data to a given virtual machine at a specific location (a legal matter to be addressed later)
- What may and may not be guaranteed in service level agreements (also a legal issue)