INTRODUCTION TO VIRTUALIZATION

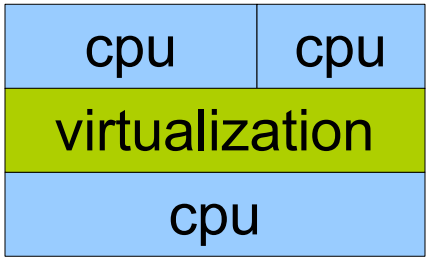
(ISSISP 2014 tutorial)

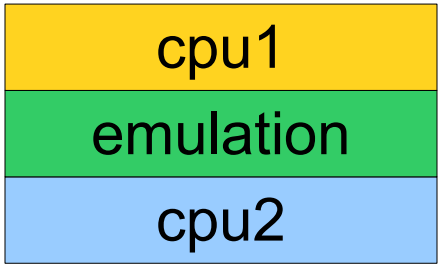
Federico Maggi — @phretor
federico.maggi@polimi.it

BASIC CONCEPTS

What is Virtualization?

- **Virtualization** is a very similar concept to **emulation**
  - **Emulation:** the system pretends to be **another system**
    - e.g., “executing” ARM instructions using a program compiled for and running on a x86 processor
  - **Virtualization:** the system pretends to be **one or more systems**
    - e.g., multiple **operating systems** sharing the same CPU





  - Emulation & virtualization **can co-exist**, and multiple virtual machines of different architectures can run concurrently on the same physical hardware
    - **Example:** an x86 and an ARM guest running on top of an x86 host

Emulation example (CPU)

- CPU **emulation** can be accomplished in different ways, but the **underlying concept** is the same:
  - **Binary rewriting** (or translation): Take the instruction stream, and **generate another instruction stream**.
- This approach is generally **slow**, because every instruction—and the I/O operations—must be entirely **handled in software**

Emulation: QEMU (example)

- QEMU efficiently emulates a couple of dozens of **architectures**, including PowerPC, x86, ARM, MIPS, Sparc, Alpha, etc.
- In addition to CPU emulation, QEMU, as well as other emulators, provide **device emulation**, e.g.:
  - VGA display
  - PS/2 mouse and keyboard
  - block devices
- We will see how QEMU (and other emulators) can be used to **virtualize** an entire machine.

CPU Emulation

- **Simple to describe**, but very **challenging** to design and difficult to implement. A few examples:
  - management of the **translated code**
  - **register** allocation
  - code **optimization**
  - **memory** partitioning and management
  - **self-modifying code** support
  - **exception** handling
  - hardware **interrupts**
- The way **QEMU** performs CPU emulation via **binary translation** makes it stand out from the crowd for its good **efficiency** and **ease of portability**.

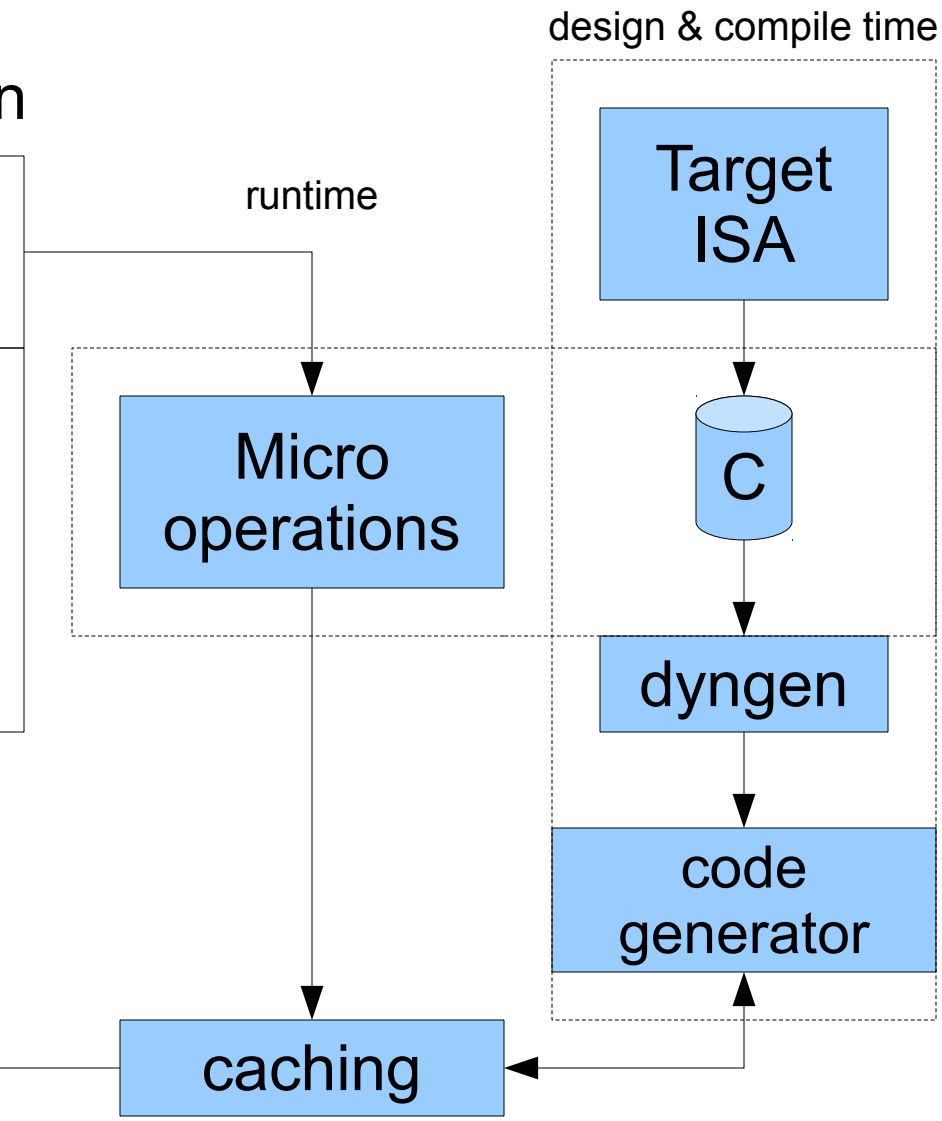
QEMU Binary Translation Example

- BT via dynamic compilation
- Guest code (PowerPC)

```
addi r1,r1,-16    # r1 = r1 - 16
```

- Target code (x86)

```
mov 0x4(%ebp),%ebx
add $0xfffffff0,%ebx
mov %ebx,0x4(%ebp)
```



Micro Operations

- Chosen so that their **number is much smaller than all the combinations of instructions and operands** of the target CPU.
- **Translation** from target CPU to micro operations is **hand coded** in C and then **compiled with GCC** in the target CPU object file
- From the previous `addi r1,r1,-16` `# r1 = r1 - 16`

```
movl_T0_r1          # T0 = r1
addl_T0_im -16      # T0 = T0 - 16
movl_r1_T0          # r1 = T0
```

- **Easy portability** is ensured by GCC backends

Micro Operation Implementation

```
void op_movl_T0_r1(void)    /* movl_T0_r1 */
{
    T0 = env->regs[1];      // virtual registers
}

extern int __op_param1;
void op_addl_T0_im(void)    /* op_addl_T0_im */
{
    T0 = T0 + ((long)(&__op_param1));
}
```

Runtime Translation

```
# movl_T0_r1
# ebx = env->regs[1]
mov    0x4(%ebp),%ebx

# addl_T0_im -16
# ebx = ebx - 16
add    $0xfffffff0,%ebx

# movl_r1_T0
# env->regs[1] = ebx
mov    %ebx,0x4(%ebp)
```

Back to Virtualization

Let's give a loose definition:

"virtualization is a framework or methodology of ***dividing the resources of a computer into multiple execution environments***, by applying one or more concepts or technologies such as hardware and software partitioning, time-sharing, partial or complete machine simulation, emulation, quality of service, and many others"

—Amit Singh, An Introduction to Virtualization (Jan, 2004)

Virtualization is an Old Idea

- The **concept** of *virtualization* is already applied to modern operating systems (OSs).
- Examples:
  - **Scheduling time-sharing technique** (CPU Virtualization)
    - e.g., each **process** thinks that it has exclusive access to the CPU, but the OS's scheduler makes sure that each process gets a fair share
  - **Virtual memory layout** (Memory Virtualization)
    - e.g., each **process** does not compute the physical memory addresses on its own, but the OS and the CPU “virtualize” the physical memory
  - **Screen multiplexing** (Window managers)
    - e.g., in a **multi-window system**, each program draws using pixels within an area, without checking if other windows are using the same pixels

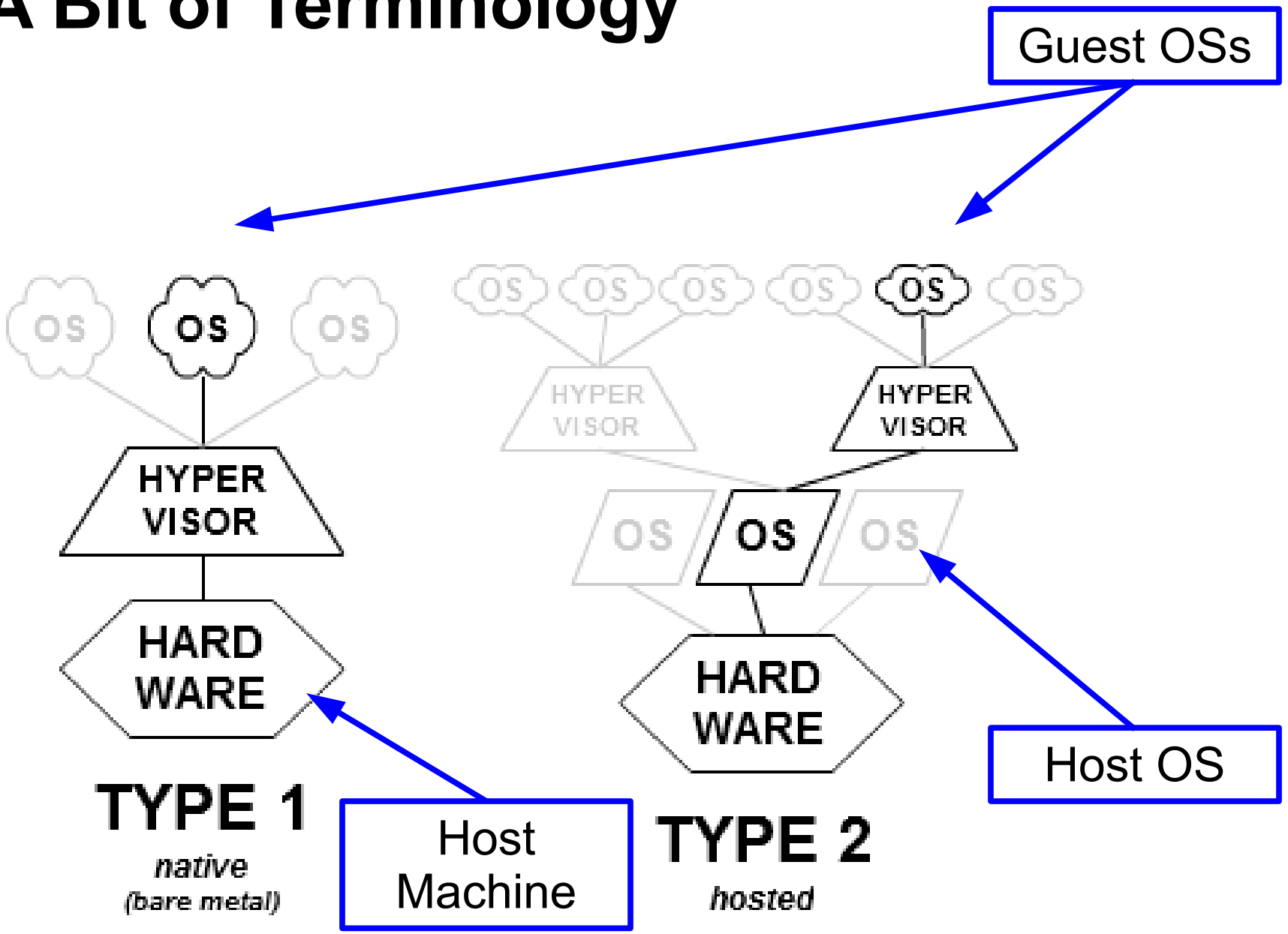
Why Virtualize?

- The basic **reasons** for virtualization are:
  - **Migration** due to HW faults
    - (the state of) virtual machines can be **serialized and resumed**
  - **Cloning** for testing environment, patches etc.
    - e.g., **old** or unsupported architectures
    - e.g., **save the state** of a virtual machine, **modify, rollback**
  - **Power usage**, exploiting maximum computation power
    - e.g., two server each utilizing 20% CPU waste power, **consolidating** them to one physical machine with two virtual machines is easy and saves power
  - Very high degree of **isolation** (Security features)
    - e.g., an **infected kernel** (e.g., rootkit) only affects the processes running on *that* virtual machine
- Can you think of **other reasons**?

Virtualizing a Machine

- An OS is designed to be in **total control** of the hardware resources that it manages
- **Concurrently running multiple OSs** on the same hardware is simply **conflicting** with the basic idea of OS
- Virtualizing a machine, with respect to the OS, means **introducing another layer** that does a very similar job
  - this layer is usually called **Virtual Machine Monitor (VMM)**
  - depending on where it sits, the VMM can take different actions
- **Ideally**, the VMM has to **monitor each CPU instruction** and ensure that **each OS is unaffected**
  - each OS thinks that it has **exclusive access to the hardware**
- The VMM must be able to have **full observability**

A Bit of Terminology



Virtualizing a Machine

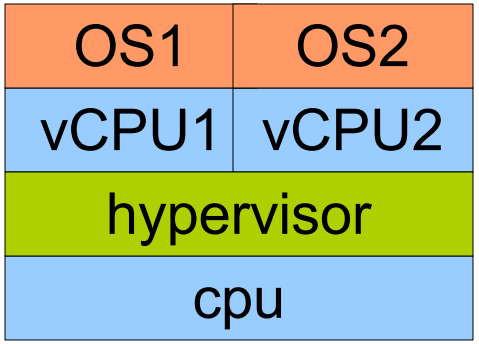
- **CPU virtualization**
  - how to run instructions “concurrently”?
- **Memory virtualization**
  - how to isolate each guest OS in a physical memory space?
- **Resource management**
  - how to create, run, destroy, migrate guests?

- There is actually much more, but this is just an intro ;-)

CPU VIRTUALIZATION

CPU Virtualization

- **Conceptually very easy**
  - run a process (i.e., application process, or the kernel),
  - interrupt the process,
  - save the CPU state,
  - run another process, and so on.

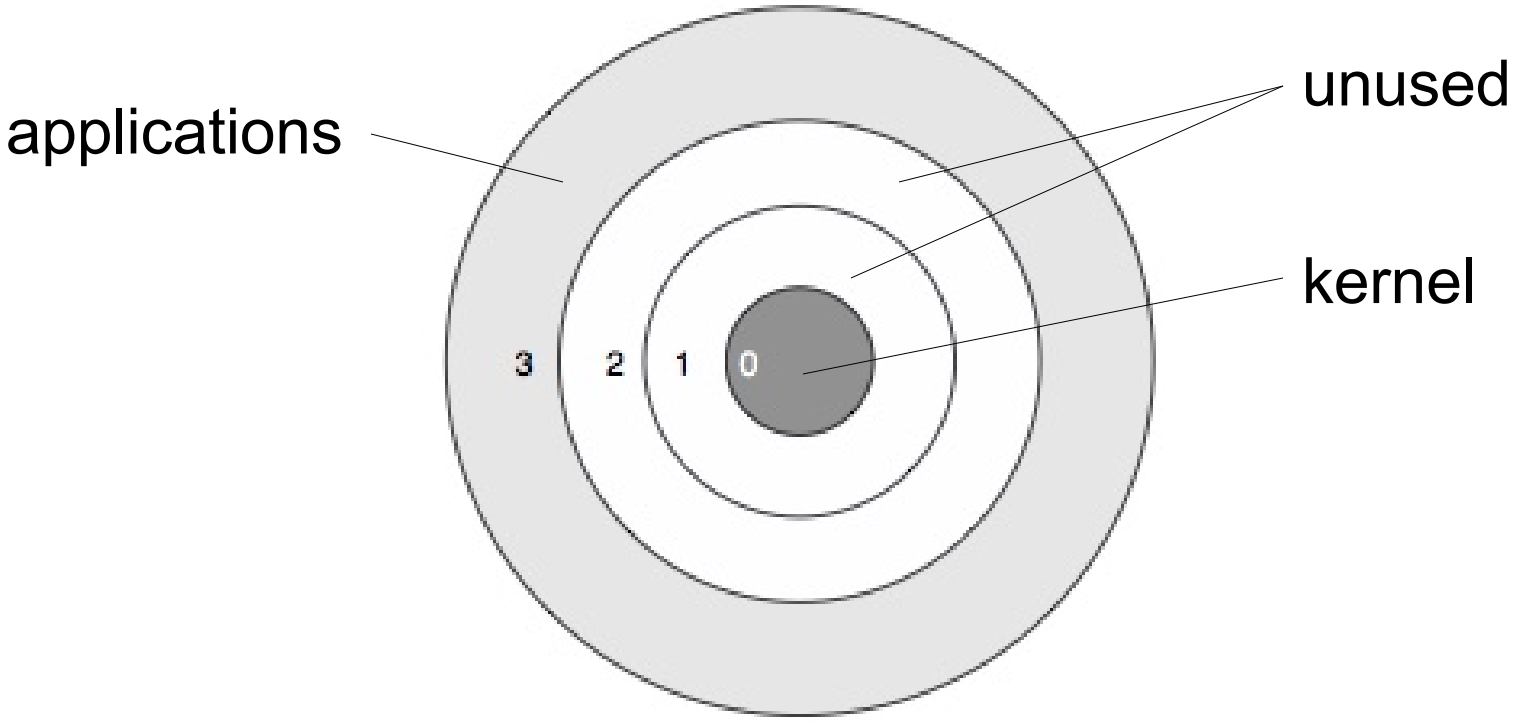


- **Problem**
  - the **physical CPU** and the **virtual CPU** are not identical
    - **example:** when the CPU is in **privileged mode**, the OS is allowed to interfere with the physical hardware (e.g., physical memory)
  - the **hypervisor** must ensure that each vCPU1 and vCPU2 do not conflict when their respective OS1 and OS2 are accessing the physical hardware

Privileges: Ring Permission (1/3)

- **Protection rings**, are mechanisms to **protect data and functionality** from faults and malicious behavior.
- A protection ring is one of two, or more, **hierarchical levels** or layers of privilege within the architecture of a computer system.
- This is generally **hardware enforced** by some CPU architectures that provide different CPU modes at the hardware or microcode level.

Privileges: Ring Permission (2/3)



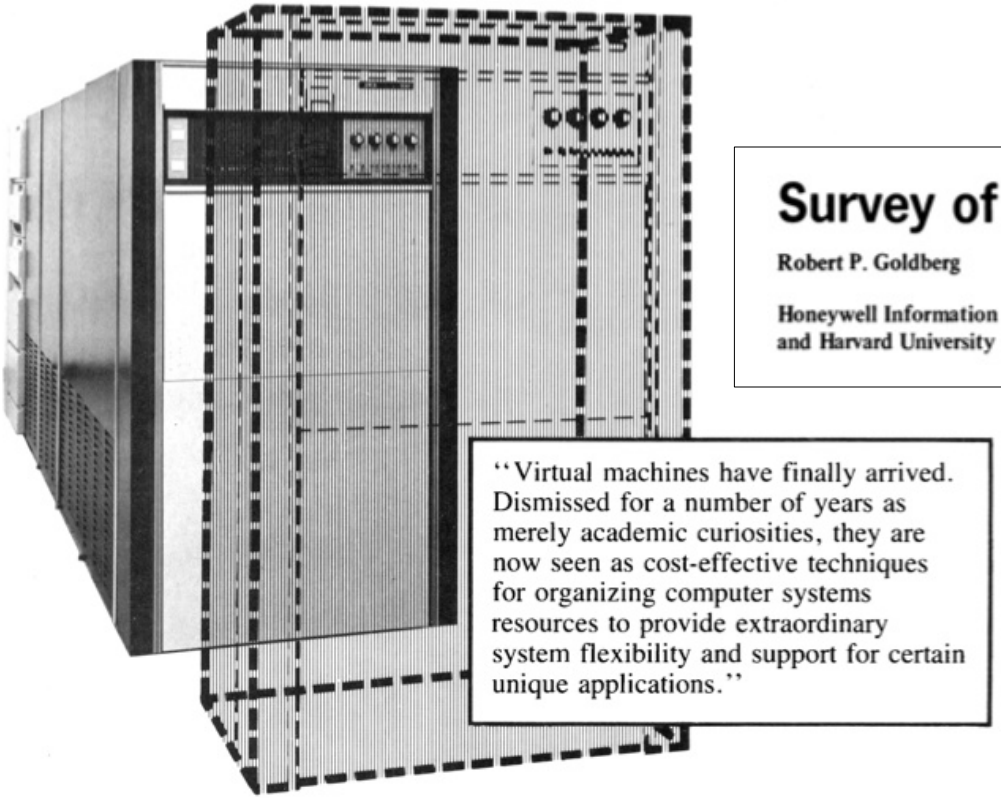
- **Special gates** (e.g., system calls) between rings are provided to allow an outer ring to access an inner ring's resources in a **controlled** manner, as opposed to allowing arbitrary usage.

Privileges: Ring Permission (3/3)

- **Privileged (or supervisor) mode**
  - hardware-mediated flag that can be changed by code running in system-level software (e.g., OS code).
  - **System-level tasks** or threads will have this flag set while they are running (ring 0),
  - whereas user-space applications will not (ring 3).
- This flag determines whether it would be possible to *execute machine code operations* such as
  - **modifying registers** for various descriptor table
  - performing operations such as **disabling interrupts**.
- Speaking of instructions...

Virtualization Requirements

"Formal Requirements for Virtualizability Third Generation Architectures" (1974). Popek and Goldberg defined **a set of requirements** that must be met.



Survey of Virtual Machine Research

Robert P. Goldberg

Honeywell Information Systems
and Harvard University

"Virtual machines have finally arrived. Dismissed for a number of years as merely academic curiosities, they are now seen as cost-effective techniques for organizing computer systems resources to provide extraordinary system flexibility and support for certain unique applications."

Privileged and Sensitive Instructions

- **Privileged instructions**
  - may execute in a privileged mode (ring 0),
  - but will *trap* if executed outside this mode (ring >0).
- **Control-sensitive instructions**
  - attempt to *change the configuration* of resources in the system
  - e.g., physical memory assigned to a program.
- **Behavior-sensitive instructions**
  - behave in a different way *depending on* the configuration of resources
  - e.g., load and store operations that act on virtual memory
  - Related to so-called “side effects”.

CPU Virtualization: Requirements

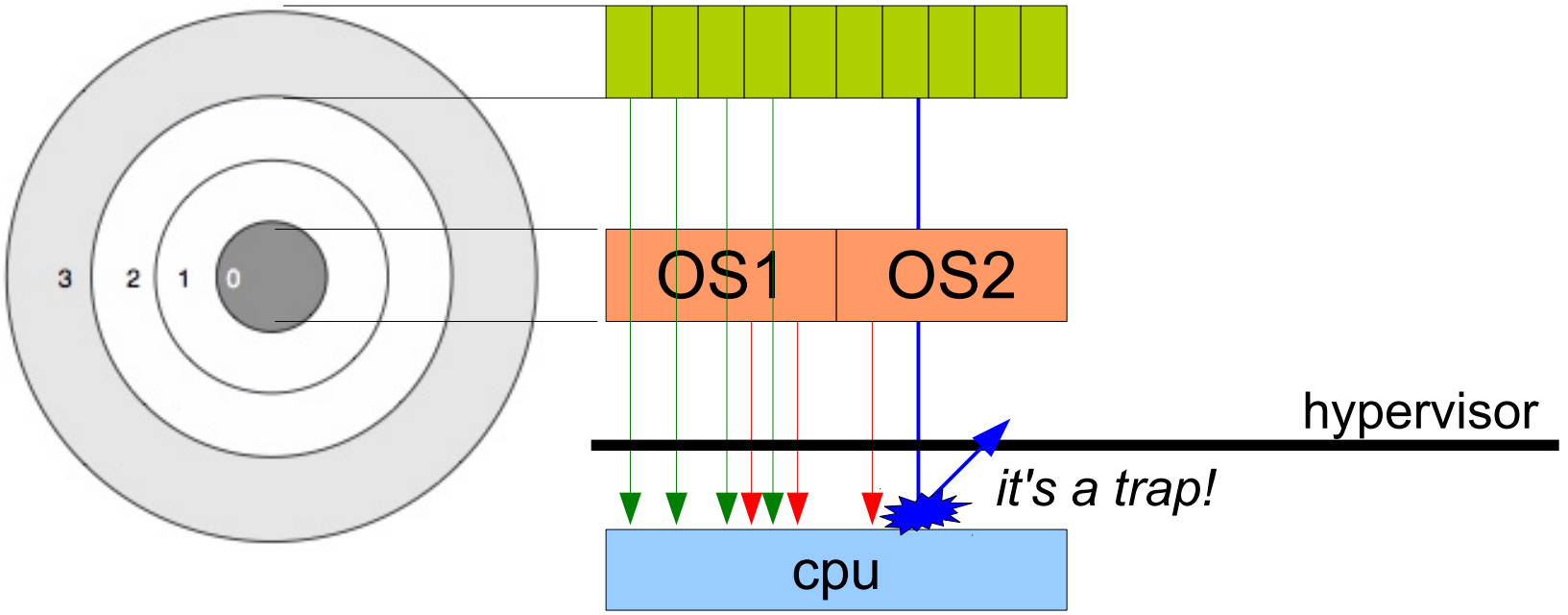
- All **sensitive instructions must also be privileged instructions**
  - so that they trap if executed in non-privileged mode
- Why? A hypervisor must be able to intercept any instructions that **change the state of the machine** in a way that impacts other processes.
- The job of the hypervisor is to **keep track of the state** of the CPU, and **expose a different state** to each OS
  - when **switching between two OSs**, we need to **save the state** of the CPU exposed to the first OS, and **restore the state** for the new one. And so on.

CPU Virtualization: Requirements

Non-sensitive instructions executed in non-privileged mode

Sensitive instructions executed in privileged mode

Sensitive instructions executed in non-privileged mode ~> trap!



Problems of x86 Virtualization

- There is a set of **17 instructions in the x86** instruction set that does **not** have this property.
  - Sensitive register instructions
    - Example: the LAR and LSL instructions **load information about a specified memory segment**. Because these **cannot be trapped**, there is **no way for the hypervisor to rearrange the memory layout** without a guest OS finding out.
  - Protection system instructions
    - Example: **SIDT**, set the values of certain condition registers, but **have no corresponding load instructions**.
      - So, every time they execute they must be trapped and the new **value stored elsewhere** as well, so it can be **restored when the virtual machine is re-activated**.

Some Solutions for x86 Virtualization

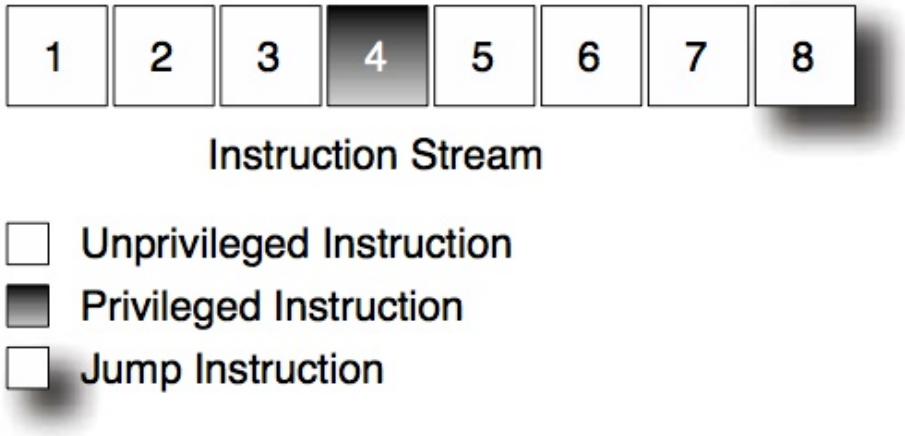
- x86 is a **very attractive** architecture because is very widespread
- In order to overcome the issue of x86 architecture we can use 3 **possible solutions**:
  - **Binary Rewriting** or Translation ~> workaround
  - **Paravirtualization** ~> workaround
  - **Hardware-assisted Virtualization ~> makes x86 virtualizable**
- The **price to pay** for the workarounds is either
  - performance penalty (binary rewriting)
  - modify the OS (paravirtualization)

Binary Translation for Virtualization

- It's **conceptually like emulation**, but we don't translate to another architecture's ISA. QEMU can help as well.
- The instruction stream is scanned by the virtualization environment and **privileged instructions are identified**.
- Every **privileged instruction is rewritten** to execute on an emulated CPU rather than on the real CPU.
- Basically, **the guest executes on an interpreter** rather than directly on the physical CPU.
  - The interpreter correctly implements **non-trapping instructions**
  - Essentially, the interpreter **separates the physical state from the virtual state**.

Binary Translation for Virtualization

- It **inserts breakpoints** on any jump and on any privileged instruction.
- When it gets to a jump, the instruction stream reader needs to **quickly scan the next part for privileged instructions** and mark them.
- When it reaches a privileged instruction, it has to **emulate it**.



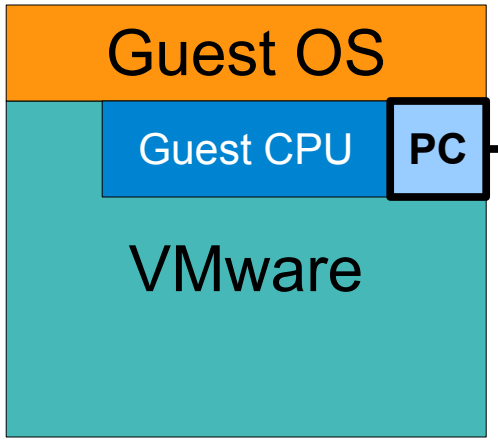
VMware's Approach to BT

Observation: non-privileged instructions are the majority. Let's optimize those.

- *Binary*
  - Obviously, we're translating x86 op codes.
- *Dynamic*
  - Translation happens at runtime, interleaved with execution of the generated code.
- *On demand (lazy translation)*
  - Code is translated only when it is about to execute. Side-steps the problem of telling code and data apart.
- *System level*
  - Makes no assumption about the guest code. It just translates and execute whatever code. Requires no OS modification.

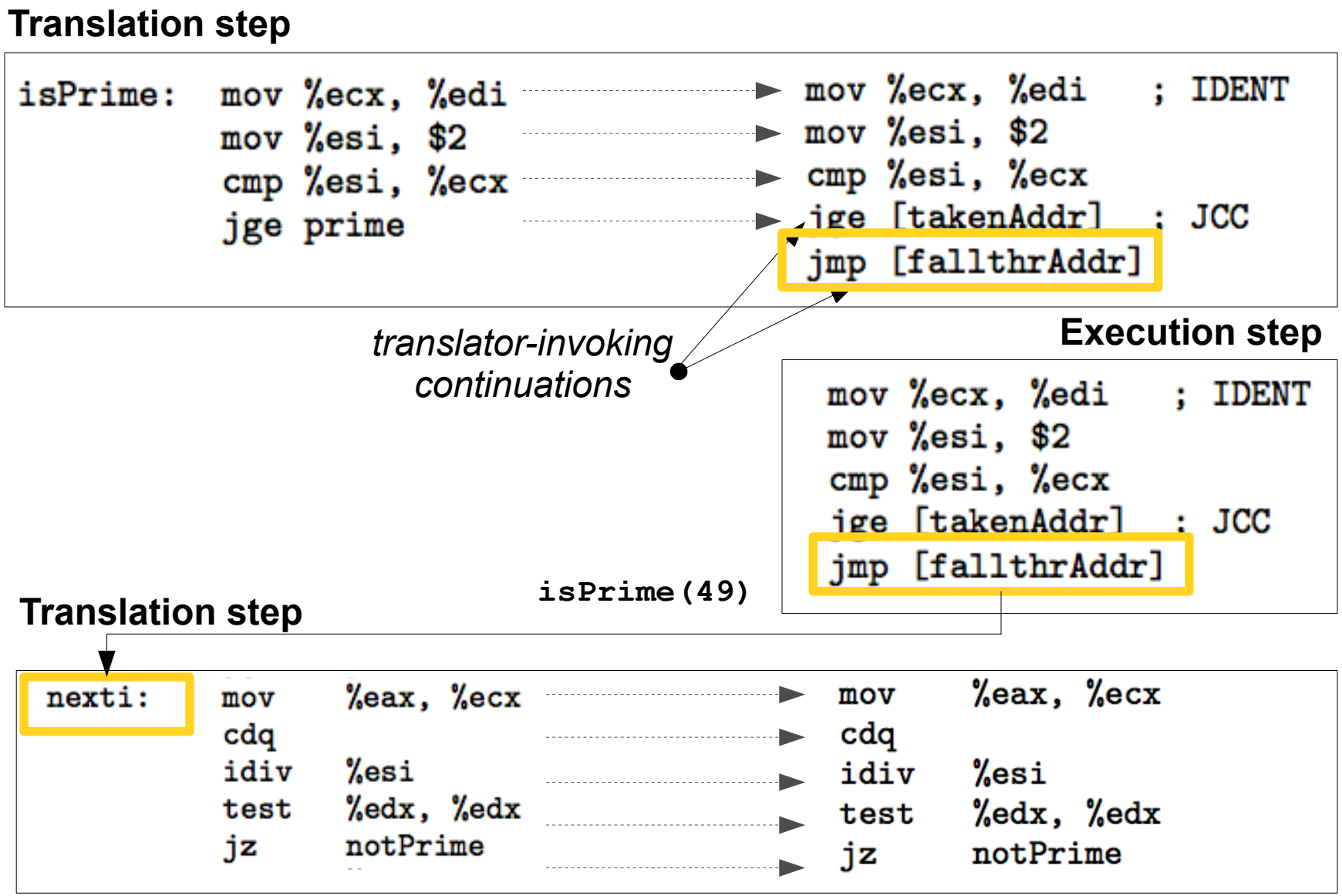
VMware's Translation Example

```
int isPrime(int a) {
  for (int i = 2; i < a; i++) {
    if (a % i == 0) return 0;
  }
  return 1;
}
```




```
isPrime:  mov    %ecx, %edi ; %ecx = %edi (a)
          mov    %esi, $2   ; i = 2
          cmp    %esi, %ecx ; is i >= a?
          jge    prime      ; jump if yes
nexti:    mov    %eax, %ecx ; set %eax = a
          cdq               ; sign-extend
          idiv   %esi       ; a % i
          test   %edx, %edx ; is remainder zero?
          jz     notPrime   ; jump if yes
          inc    %esi       ; i++
          cmp    %esi, %ecx ; is i >= a?
          jl     nexti      ; jump if no
prime:    mov    %eax, $1   ; return value in %eax
          ret
notPrime: xor    %eax, %eax ; %eax = 0
          ret
```

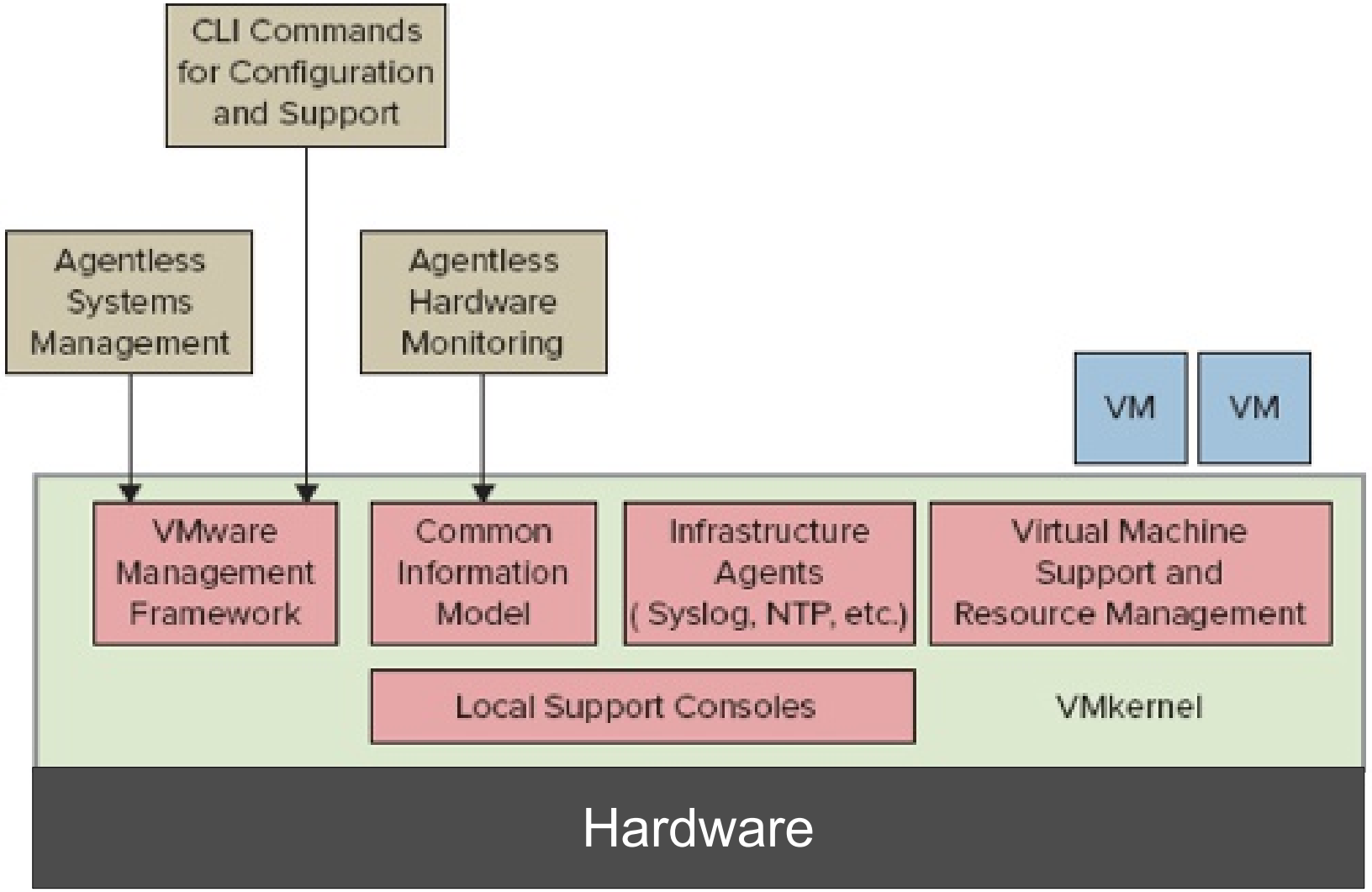
VMware's Translation Example



Performance Issues

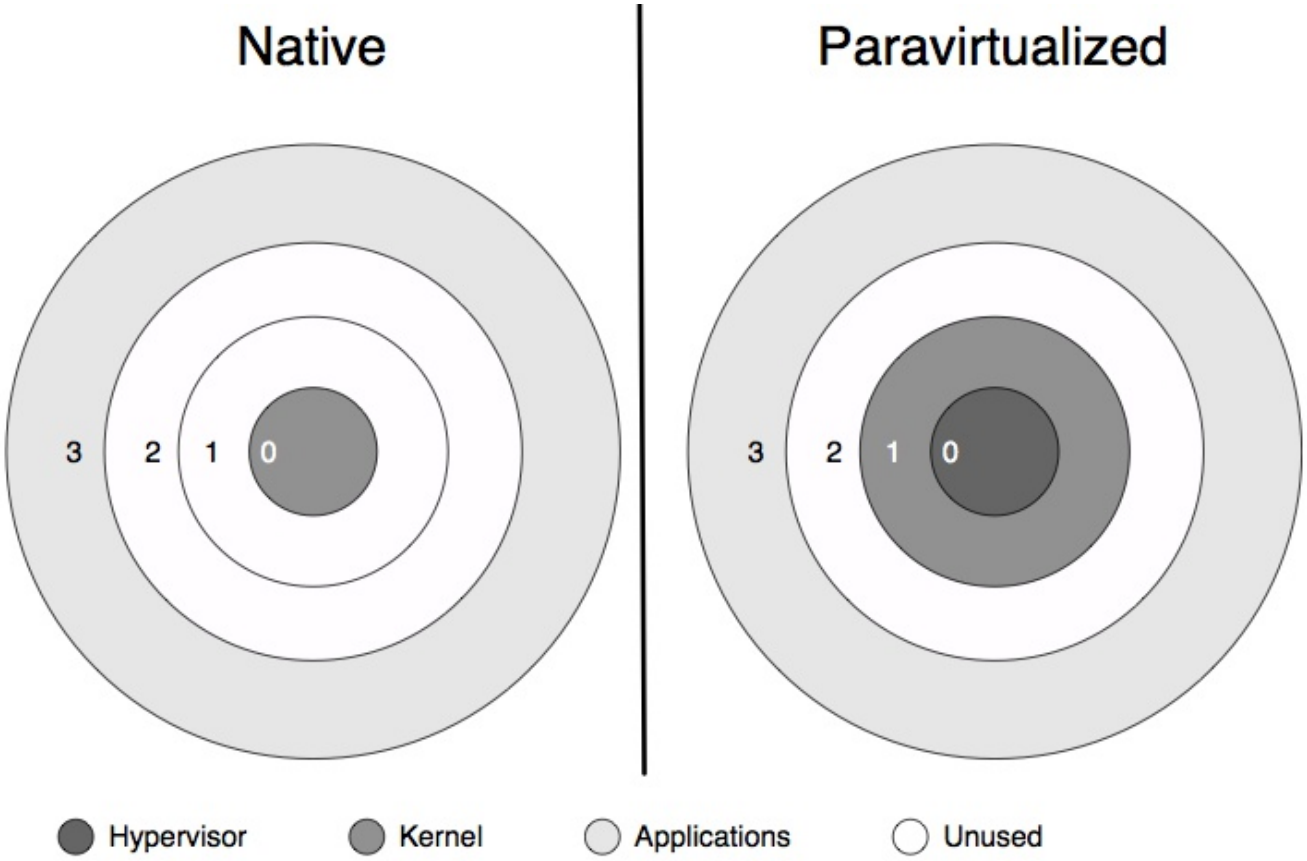
- **Consequence:** the **new control transfers** that are added
  - change the **code layout** (may result in more jumps),
  - imply new **calls to the translator**.
- The **more** translation-execution-translation switches, the more **speed penalty** is introduced.
- **Mitigations:**
  - keep a **cache** of the translated code blocks, and
  - **allow** translated code blocks to **jump** between each others when it's safe to do so, **without invoking the translator** all the time

VMware Resulting Architecture



Paravirtualization (Xen Approach)

- Except for the problematic instructions, x86 is virtualizable
- Ring 1 is unused: lets use it!



Paravirtualization (Xen Approach)

- **Approach:** ignore the problematic instructions and let the OS deal with them (i.e., inform the hypervisor).
- If a **guest system** executes an instruction that doesn't trap while inside a **paravirtualized** environment, the guest **has to deal with the consequences**.
- Conceptually, this is similar to the binary rewriting approach, except that here the **rewriting happens at compile time (or design time)**, rather than at runtime.
- The OS is designed with support for running in ring 0, but it runs in ring 1 now
  - it means that it **cannot execute privileged instruction**s, because the will simply fail

Privileged Instructions via Ring 1

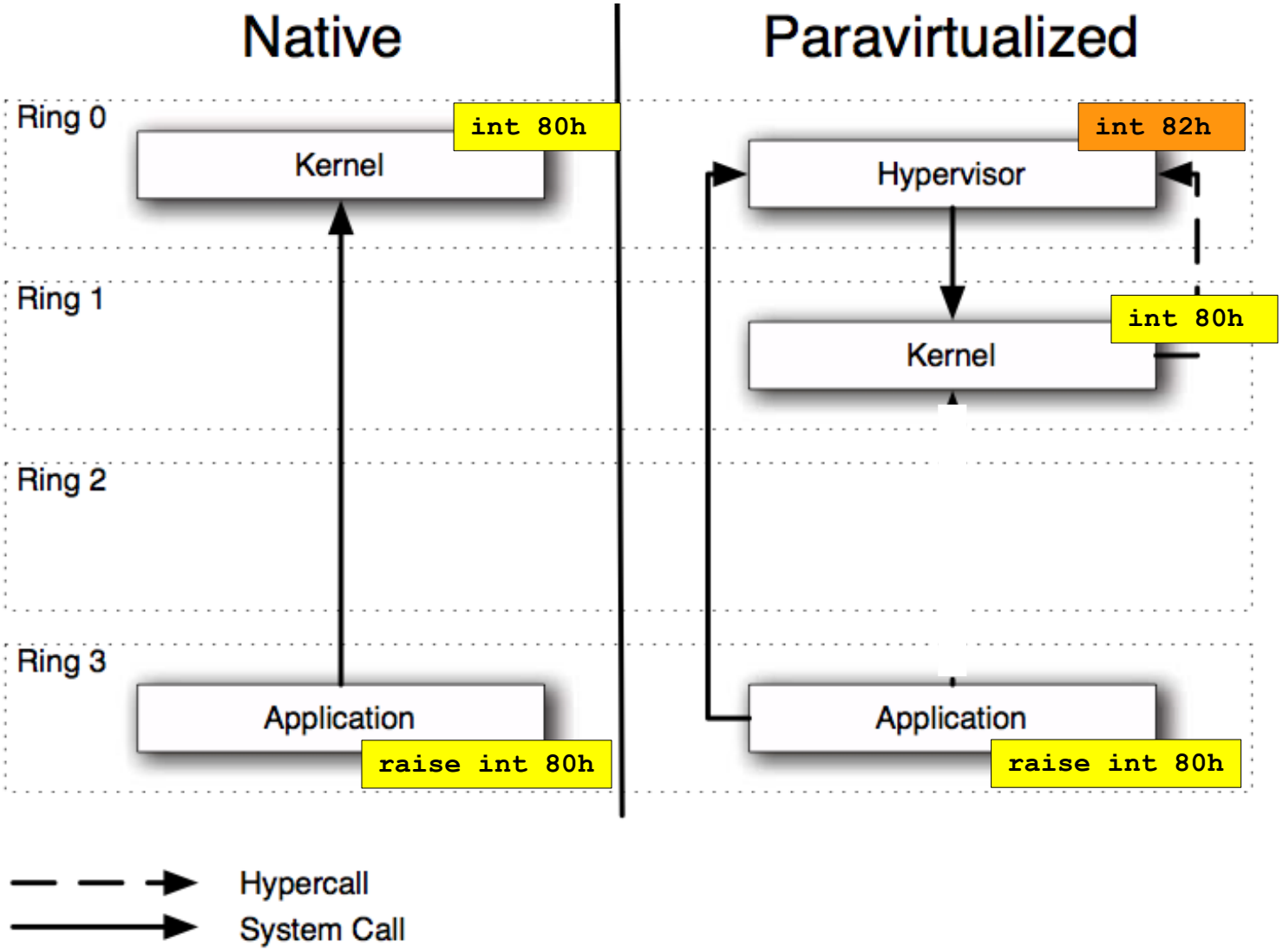
- In order to **simulate a privileged instruction** the hypervisor exposes a set of **hypercalls**.
- A **hypercall** is conceptually similar to a system call. On UNIX systems, the convention for invoking a system call is to push the values and then raise an interrupt.

  A regular **system call** from ring 3 (process) to ring 0 (OS)

```
push dword 0   # push parameters
mov eax, 1     # set the system call
push eax       # push syscall identifier
int 80h        # raise software interrupt
```

  The OS has **interrupt handlers** that take care of each system call.

Ring Transitions with Hypercalls

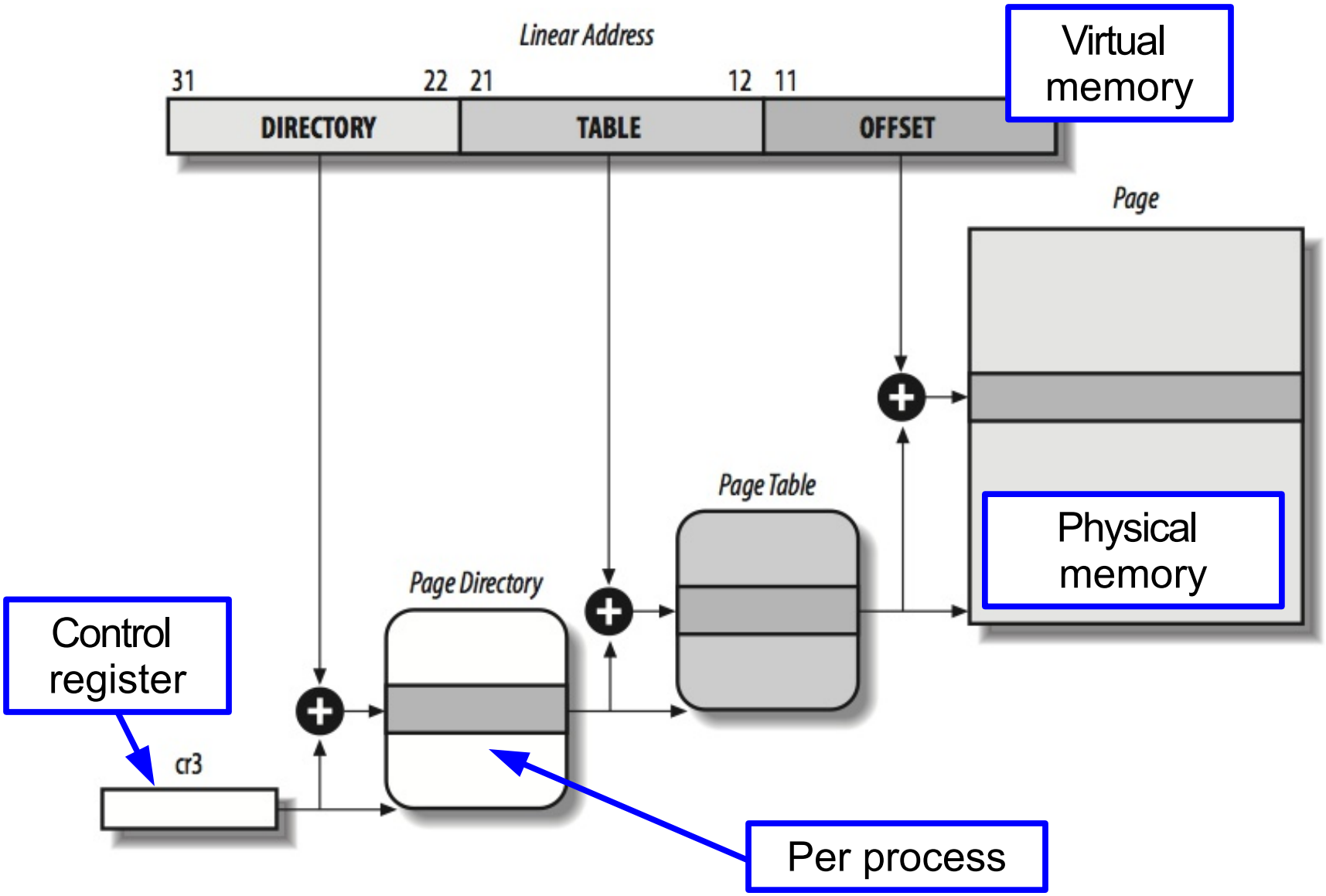


MEMORY VIRTUALIZATION

Memory Virtualization

- Conceptually easy, but more difficult than CPU virtualization
- **Two main problems**, in essence
  - (problem 1) the **hypervisor must be aware of the physical memory allocation**, to confine each guest OS in an address space
  - (problem 2) modern CPUs **cache the virtual-to-physical address** translation to speedup memory accesses
- **Consequence**
  - require significant modification of the guest OS, which needs to inform the hypervisor before allocating memory
  - require significant work from the hypervisor, which needs to manage the swapping of processes of different OSs

Virtual Memory in Regular OSs



x86 Problems and Solutions

- **Solution** by user-space hypervisors (e.g., VMware)
  - each **guest** is assigned a **virtual page table**
  - every **update** to the virtual page table **goes through the hypervisor**
  - the hypervisor **multiplexes** the **updates of the real page table** for each guest OS
  - **Price to pay:** increased overhead
- **Solution** by ring-0 hypervisors using PV (e.g., Xen)
  - use **real page tables** as if each guest OS runs in hardware
  - give each guest OS **read-only access** to the page tables (no overhead for reads)
  - **Price to pay:** modify the guest OS to **inform the hypervisor** when page **updates** are needed (using a hypercall).

How About Cached Virtual-to-Physical Translations?

- The **second problem** is that the real CPU manages **cache misses** in hardware by **walking the OS's page table**
  - but...which OS?
  - The CPU is unaware that there is more than one OS running!
  - Which CR3? The hypervisor intervenes to write a “physical” address in it, which points to the real physical address.
- Plus, the **CPU does not allow to modify its cache**
- In **paravirtualized guests**, this is **solved** by flushing the cache at each OS switch (~> performance penalty)
- The **guest OS must be modified** to inform the hypervisor every time a new process is allocated

Performance of Paravirtualization

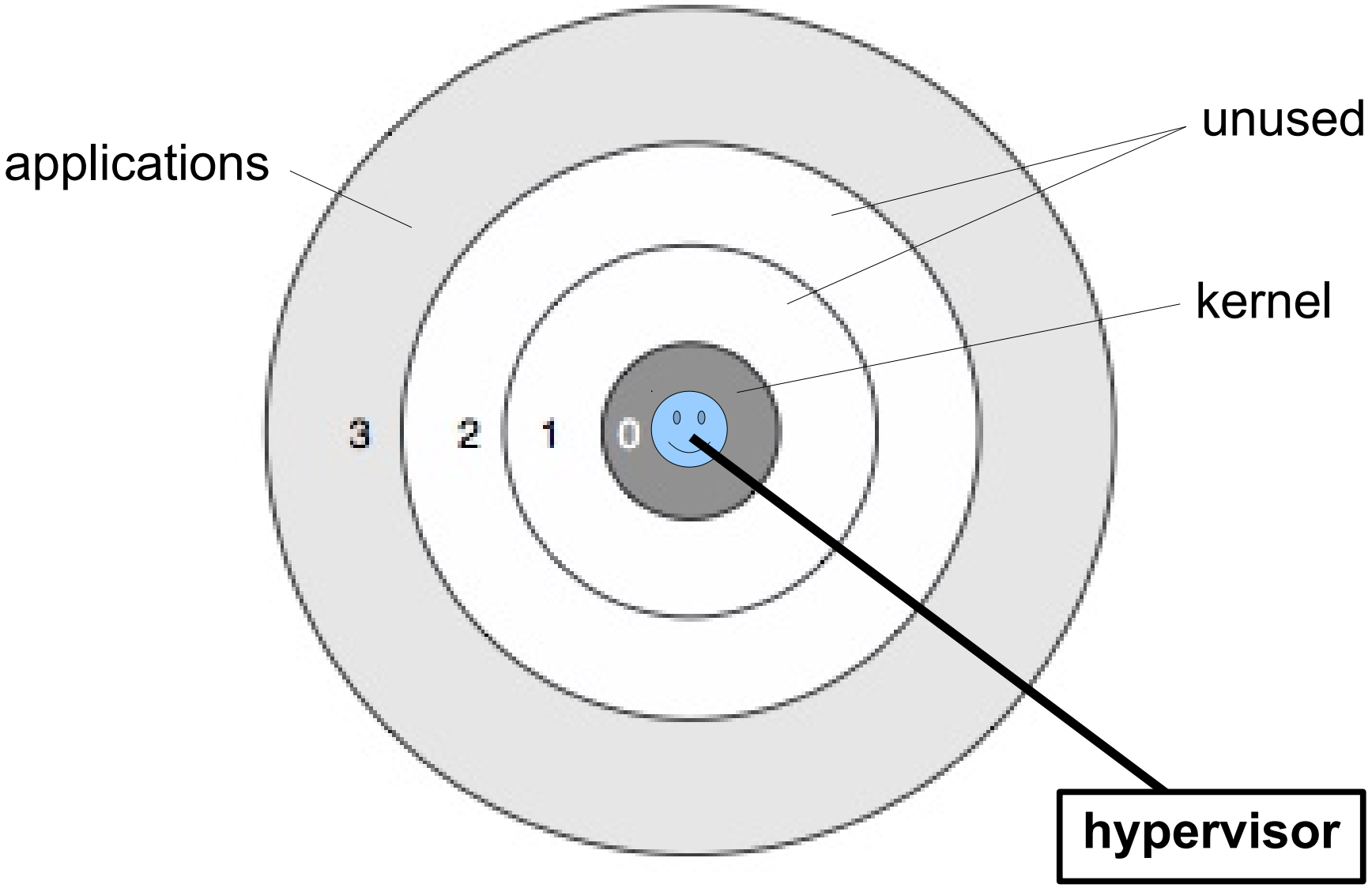
- Most of the instructions **run directly in hardware**
- **Issues**
  - extra transition step from ring 0 to ring 1 imposes a bit of overhead
  - Less overhead than emulation in binary translation
  - Need to port the OS to Xen in order to use the hypercalls an other few details
    - how about closed-source OSs?

Recap on Problems and Solutions

- The issues that we have seen generally boil down to the fact that **hardware**
  - was **designed to run one OS**
  - has **internal state** that is **not observable** or **not modifiable** from the outside
- This problem applies to other components such as:
  - I/O devices
    - modern video cards have a lot of internal states
  - time (not really a device, but a very important concept)
    - real time vs. CPU time vs. virtual machine time

HARDWARE-ASSISTED VIRTUALIZATION

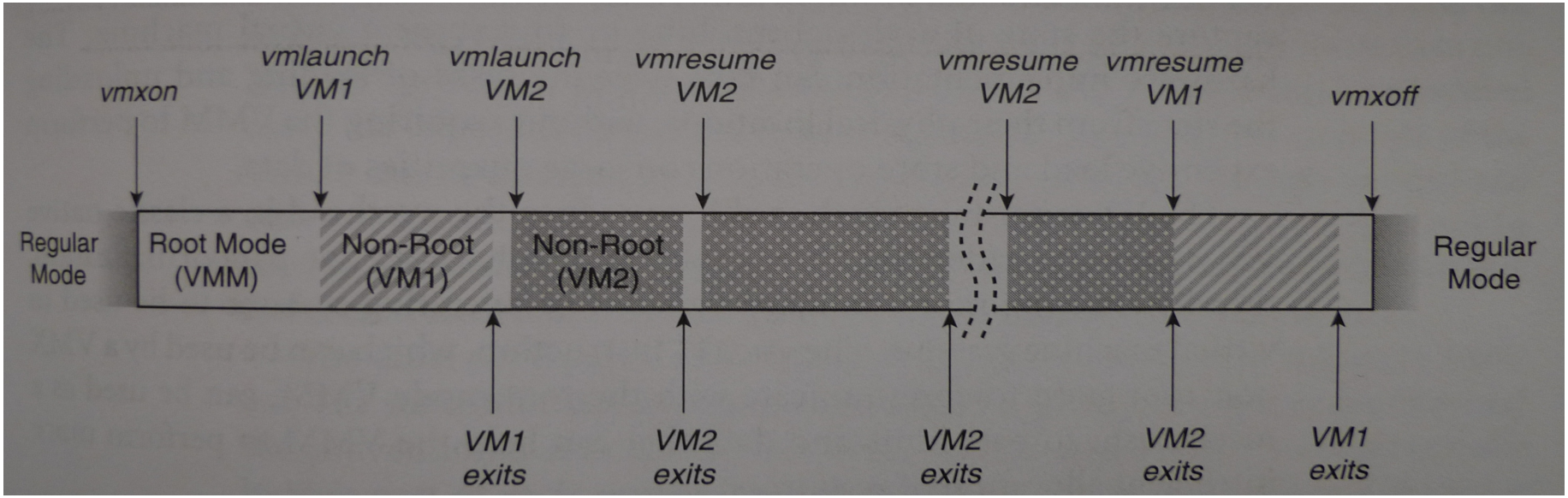
Hardware-Assisted Virtualization



Hardware-Assisted Virtualization

- **Radically** solved the aforementioned problem for CPUs
- **Intel** and **AMD**
  - added specific instructions to make virtualization easier for x86
  - AMD ~> AMD-V, formerly *Pacifica*
  - Intel ~> IVT or VT
- **Result:** adding a “ring -1” above ring 0
  - the OS stays where it expects to be
  - all **sensitive instructions** can be **trapped** with no side effects
  - no need to modify the OS, at all, yet with the same benefits of the paravirtualization approach.

Hardware-Assisted Virtualization



- We have two modes:
  - VMX Root Mode = VMM at hypervisor level
  - VMX Non-Root Operation = Normal execution into the virtual machine

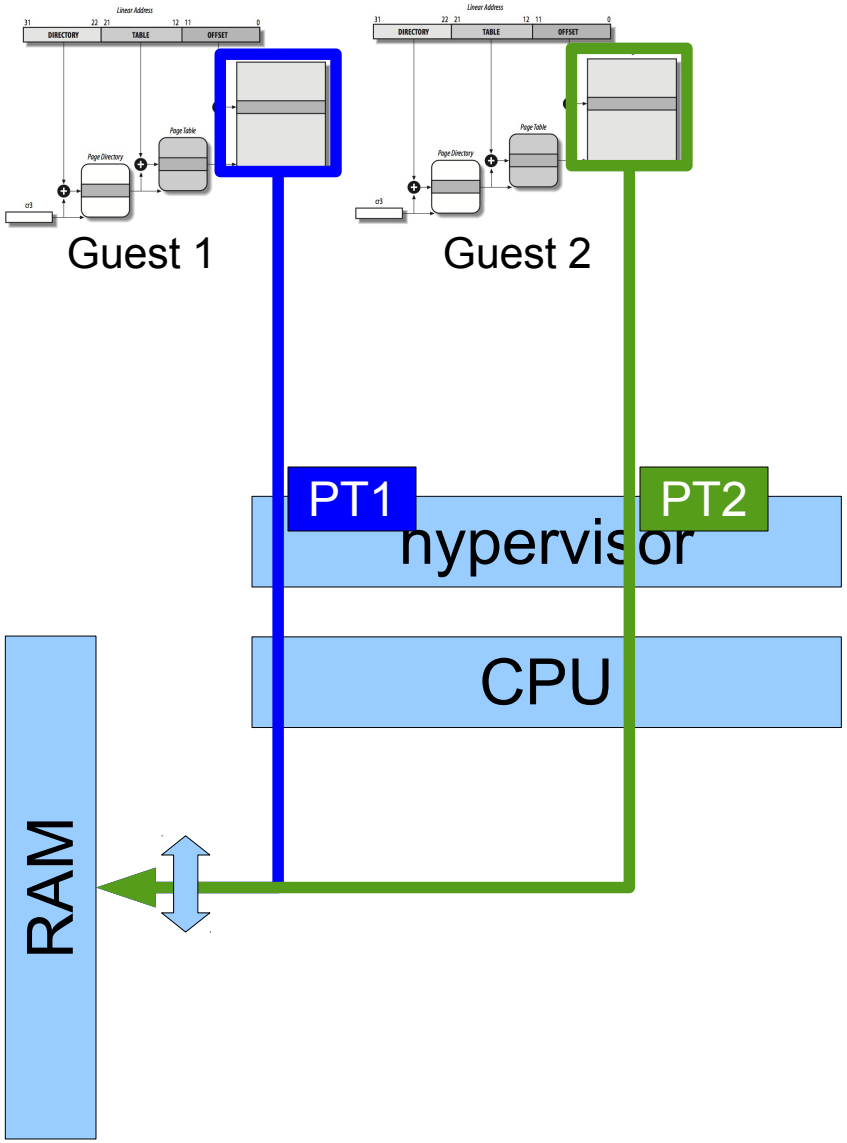
Hardware-Assisted Virtualization

- IVT adds a **new mode** to the processor, called VMX
- A hypervisor can run in **VMX root mode** and be **invisible** to the operating system, running in ring 0.
- New instructions set
  - enable ~> VMX root-mode is enabled and CPU is configured to execute the VMM in root-mode.
  - vmexit
    - access to privileged CPU state
    - interrupt virtualization
    - I/O device virtualization
    - **page-table virtualization**
  - vmresume ~> back to non-root operation
  - vmlaunch ~> execute virtual machine non-root mode

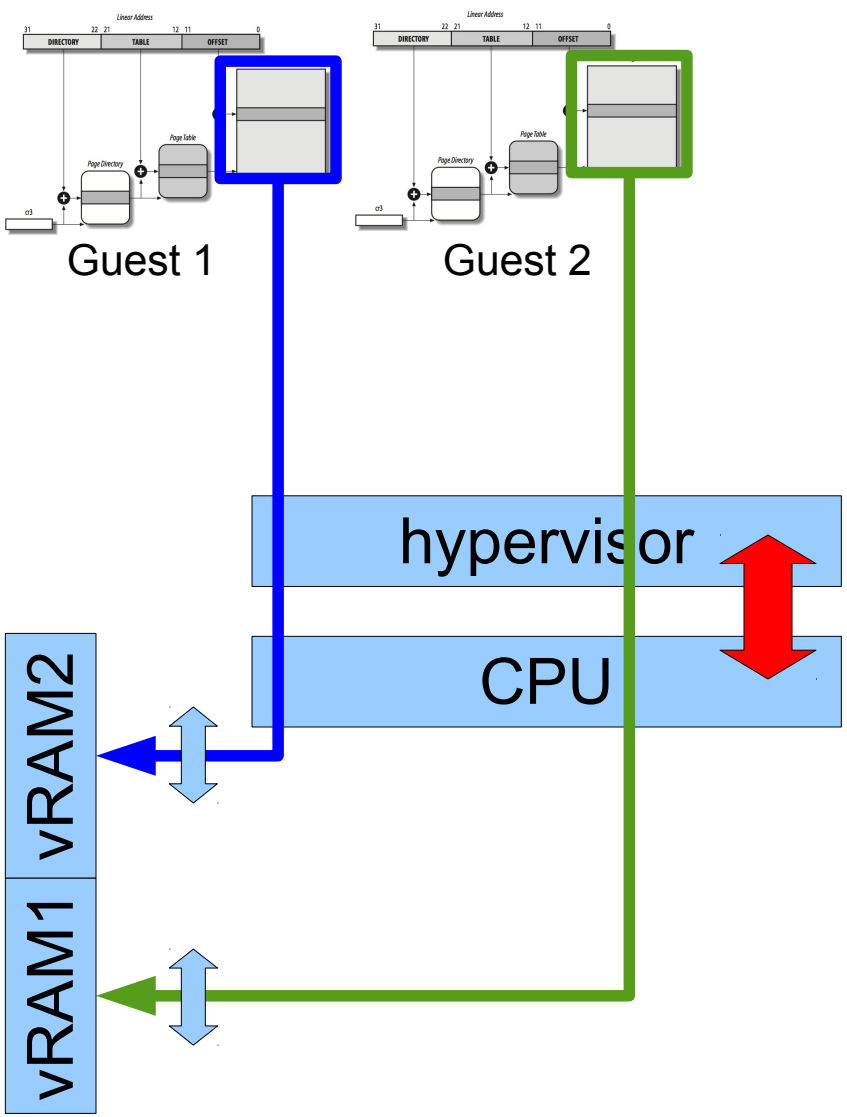
Hardware-assisted Memory Mgmt

- Shadow page tables
  - implement **Xen's page-table update** mechanism in hardware
  - trap into the hypervisor whenever the **guest OS attempts to update the page table** and change the mapping
- Nested page tables (best)
  - Adds **another level of indirection** in memory addressing
  - “physical” addresses are **not really physical**, but those assigned to the guest
  - the CPU transparently handles all this for us!

Without nested page tables



With nested page tables

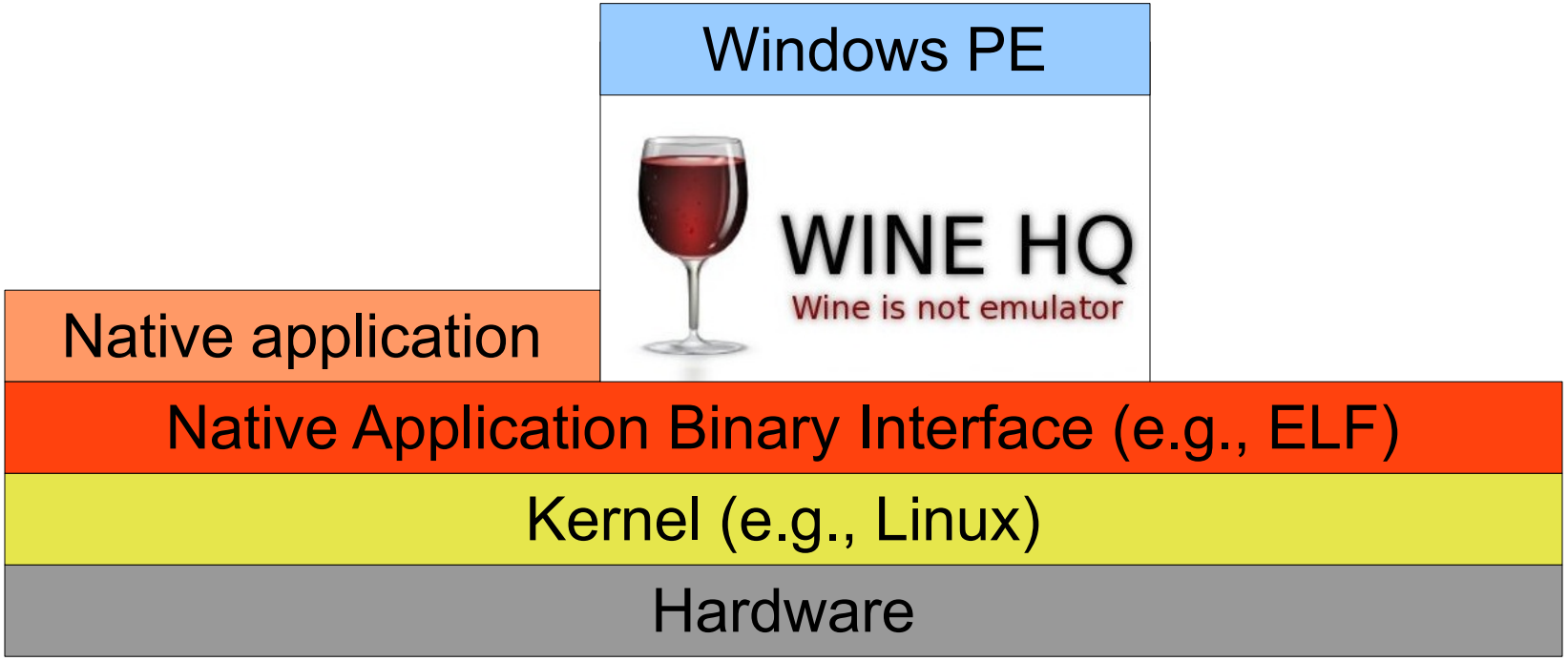


OTHER FORMS OFVIRTUALIZATION

(examples)

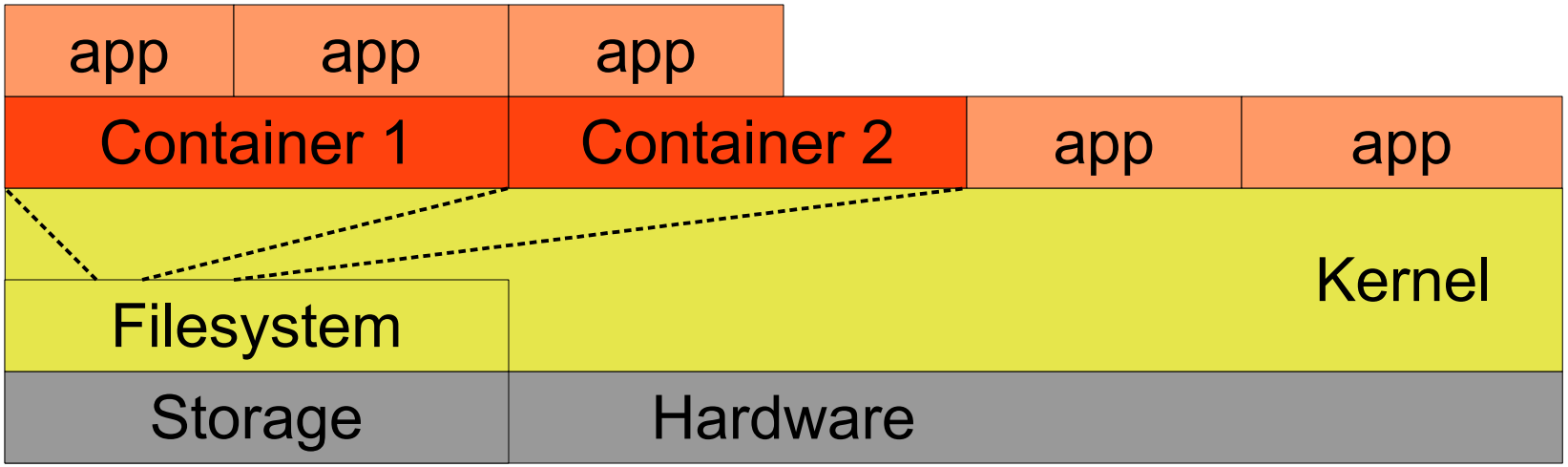
Other Forms of Virtualization (1/3)

- ABI – Application Binary Interface Virtualization
  - allow the execution of binary formats on other OSs
  - e.g., **Wine:** run Windows PE binaries on Linux
  - wrapper around Linux system calls to emulate Windows API



Other Forms of Virtualization (2/3)

- Containers (e.g., Jail, LXC)
  - isolated application-level environments
  - very lightweight, fast to create and destroy
  - multiplex the ABI and filesystem without changing the OS
  - increasingly popular for deploying apps in multi-tenant OSs



Other Forms of Virtualization (3/3)

- Language based
  - define an intermediate target language, often called bytecode, for compiled applications
    - usually closer to the source code than to the machine code
  - define a semantic and virtual machine for executing the intermediate language
  - allow complete isolation at the “instruction” level
  - each application runs in a small virtual machine, which is actually a process in the guest operating system
  - allow program portability across different guests, at the cost of porting the virtual machine
  - popular examples: Java, Dalvik (Android), .NET

APPLICATION OF VIRTUALIZATION
(example)

Using Virtualization for Dynamic Analysis of Unknown Binaries

- **Problem**
  - x86 executable binary that does something **probably bad** to our system.
  - we don't know what it does, and have **no time to manually analyze it** (as there are ~180 million suspicious binaries out there)
- **Approach**
  - let it **run** on a real machine and observe **what it does**
  - **interesting actions:** system calls, because they imply that privileged code is executed
    - open a network connection, write to a file, place a call, send SMS

Example Process Trace (Mac OS X)

```
0   = issetugid(0x0, 0x0, 0x0)

0 = geteuid(0x0, 0x0, 0x0)

0 = csops(0x0, 0x0, 0xBFFFF7F4)

12  = shared_region_check_np(0xBFFFD790, 0x0, 0xBFFFF7F4)

0 =  __sysctl(0xBFFFD630, 0x2, 0xBFFFD5F8)

0   = stat64("CoreFoundation", 0xBFFFE7B8, 0x1)

...

193084 =  write(0x4, "\316\372\355\376\a\0", 0x2F23C)
```

Naïve Solution

- **Modify** an open-source OS to log every time a system call is invoked
- Alternatively, use **process-auditing tools** (ptrace, DTrace) to do the same job without modifying the OS
- Prepare a real system, with the modified OS installed and a few applications
- Launch the binary (revert, repeat for every sample)
- Do you see the **problems**?
  - what if the OS is **not open source** or have no process auditing?
  - a **smart malware** can realize that a process is being traced, or that the OS has been modified.

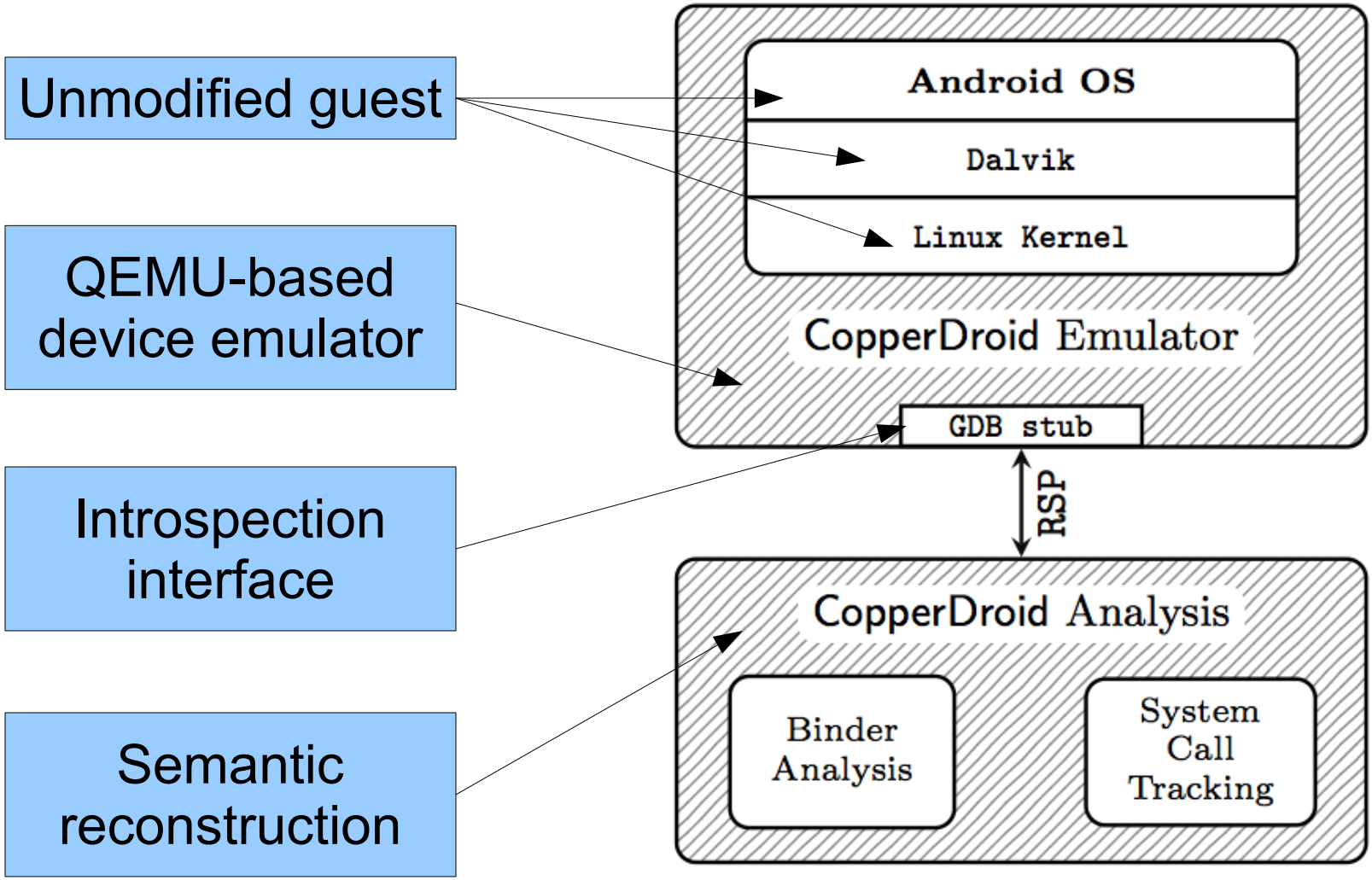
Root of the Problem

- The malware is a user-space process
- Thus has access to the same information that any process has
- Can invoke system calls (ring 0), inspect their results
- The tracing happens in ring 3, or in the best case in ring 0
- This makes the tracing non transparent with respect to any process running in ring 3 or ring 0

Virtualization to the Rescue

- We run an unmodified OS on top of a hypervisor
- We work in a virtual ring -1, that is into the hypervisor
- Tools like QEMU or HyperDBG make this very very easy
- What can we do?
  - Modify the hypervisor, or use the API provided by the hypervisor
  - We can
    - intercept “every” instruction
    - inspect the memory, the CPU state, etc.
  - All without the ring 3 noticing ~> transparent debugging
  - Some hypervisors (e.g., WindRiver's) allow backward execution!

Example: CopperDroid



Semantic Gap Problem

- The instruction trace is not very informative

```
movl    $0x200005d, %eax
movq    %rcx, %r10
syscall
jae     0x7fff988ac9b4
movq    %rax, %rdi
jmpq    0x7fff988a919a
```

- We need to know the **semantic of each instruction** and CPU state
- From that, we can **reverse-engineer what system call was invoked** each time
- Most of the time we will have to inspect the memory and **unmarshal (i.e., deserialize) to the objects** of interest

System Calls in Linux ARM

- Like on Intel, on ARM architecture invoking a **system call induces a user-to-kernel transition**
- On ARM, invoked through the **`swi`** interrupt (SoftWare Interrupt)
- Registers:
  - **r7** contains the number of the invoked system call
  - **r0-r5** contain syscall parameters
  - **lr** contains the return address

Example Decoded Trace

```
fork( ) = 0x125
getpgid(0x41) = 0x23
setpgid(0x125, 0x23) = 0x0
getuid32( ) = 0x0
open("/acct/uid/0/tasks", …) = ...
fstat64(0x13, 0xbef7f910) = 0x0
mprotect(0x40008000, 0x1000, 0x3) = 0x0
```

Conclusive Remarks

- Virtualization is basically **resource management**
- Resources could be anything, from process to hardware
- The **hypervisor** must have unrestricted read/write access to the resource to be virtualized
- Virtualizing some CPUs was tricky, but now is easy
- Virtualization is very useful for transparent, dynamic **program analysis**
- Malicious programs have started to **detect virtual machines**, and refuse to run

THANKS*

Enjoy the rest of the school!

Federico Maggi — @phretor
<federico.maggi@polimi.it>

*thanks to Andrea Lanzi for providing me some material on CPU virtualization.

References

Most of these slides are my personal view on the content presented in the following papers, books and technical blog posts:

- Fabrice Bellard, *"QEMU, a Fast and Portable Dynamic Translator"*, Procs. of USENIX ATC, 2005.
- Barham et al., *"Xen and the Art of Virtualization",* Procs. of SOSP, 2013.
- David Chisnall, *"The Definitive Guide to the Xen Hypervisor"*, Prentice Hall, 2008.
- Adam K. & Agesen O., *"A Comparison of Software and Hardware Techniques for x86 Virtualization"*, Procs. of ASPLOS, 2006.
- Robert P. Goldberg, *"A survey of Virtual Machine Research*", Computer, 1974.
- Amit Singh, *"An Introduction to Virtualization*", available at http://kernelthread.com, written in January 2004.

References (continued)

- Bovet and Cesati, *"Understanding the Linux Kernel"*, 3rd edition, O'Reilly, 2006.
- Reina et al., *"A System Call-Centric Analysis and Stimulation Technique to Automatically Reconstruct Android Malware Behaviors"*, Procs. of EuroSec 2013. Tool available online at http://copperdroid.isg.rhul.ac.uk/
- Matthew Portnoy, *"Virtualization Essentials",* Wiley, 2012