Virtualization in Embedded Systems
Lecture for the Embedded Systems Course
CSD, University of Crete (April 7 & 11, 2025)

- Manolis Marazakis (maraz@ics.forth.gr)

FORTH-ICS Institute of Computer Science (ICS)
Foundation for Research and Technology – Hellas (FORTH)

Today's lecture

CS-428 focus shift in remainder of lectures: from "simple" to "complex" embedded

- Introduction of concepts
  - Virtualization (ISA/ABI/API, VM, VMM/Hypervisor)
  - Taxonomies of virtualization approaches
- Motivation in the context of embedded systems
  - H/W + S/W co-design
  - Use-cases (mostly from mobile)
- Virtualization techniques (& overheads)
  - Dynamic Binary Translation
  - (De-)Privileged execution, Traps (instr. & trace faults)
  - Memory management, Primary vs. Shadow structures

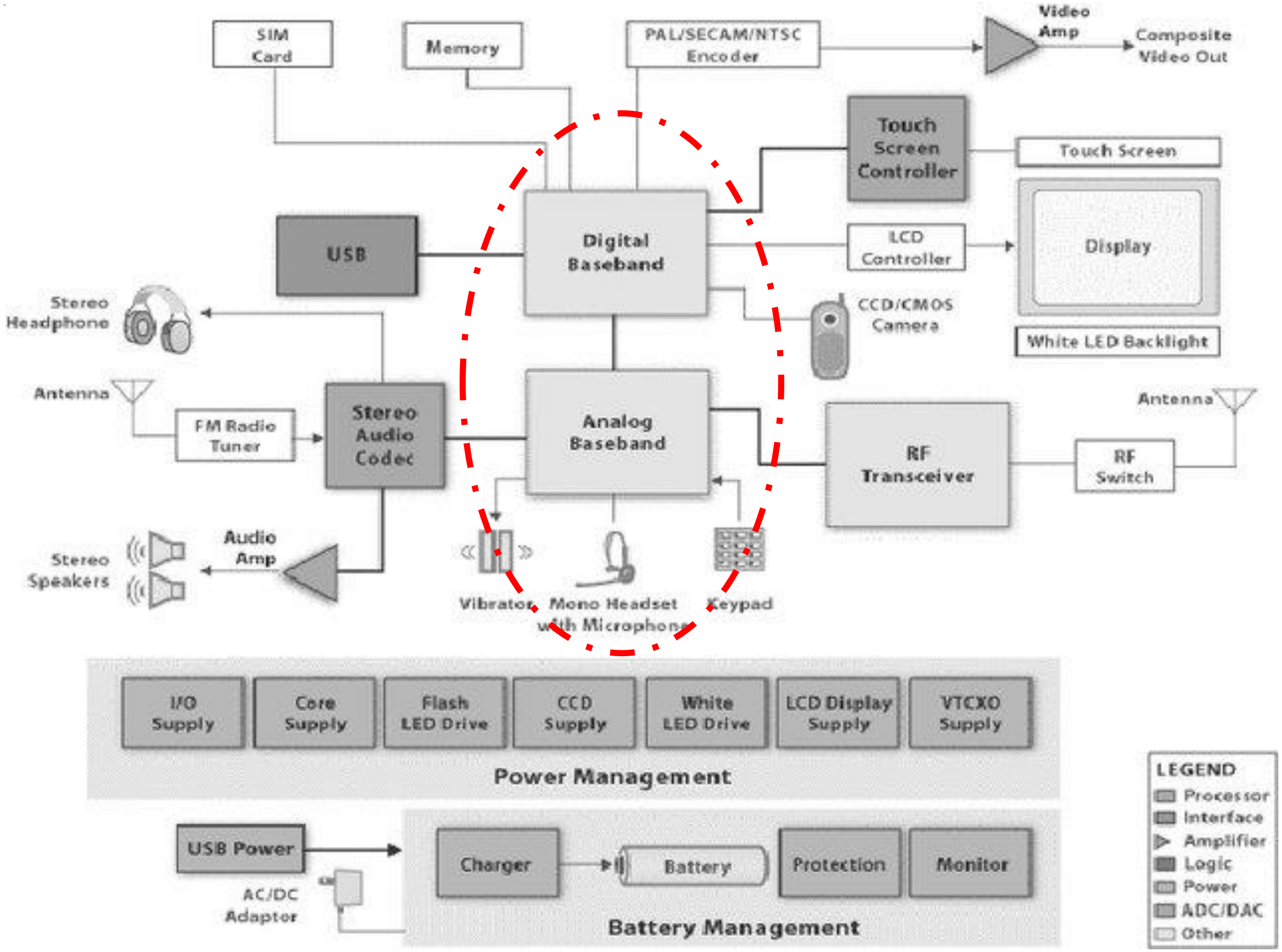
Architecture viewpoint x86, ARM (+ RISC-V)

OS viewpoint: Focusing on Linux, in purpose-built embedded systems

Virtualization to enable H/W-S/W co-design

- How to **co-design/co-develop H/W + S/W** for a system ?
  - Limited availability
  - Bugs in the production environment cannot be reproduced in the laboratory
  - Difficult to debug on-site
  - Narrow time windows
  - Sometimes in a dangerous environment …
- **Debugging** challenges
    - Is it a problem in the driver or in the device?
    - Is the firmware faulty? Is it wrongly loaded/configured?
    - Is the hardware damaged?
    - How can we reproduce the bug?
    - Do we have easy access to the environment?
    - Is it remotely located?

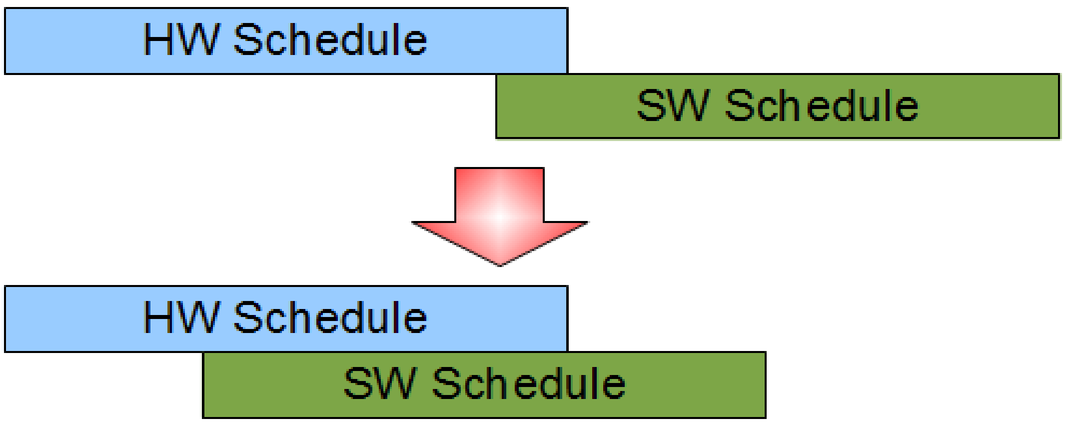
Block diagram of a basic mobile phone



Writing (and testing) device drivers … without hardware

Shift Left

- Hardware + Software = Complete product
- Feature-complete software by A-0 silicon
- Software needs to happen earlier



[source: PJ Waskiewicz & Shannon Nelson - Linux Plumbers Conference, 2011]

Embedded Linux System : Outline

BusyBox

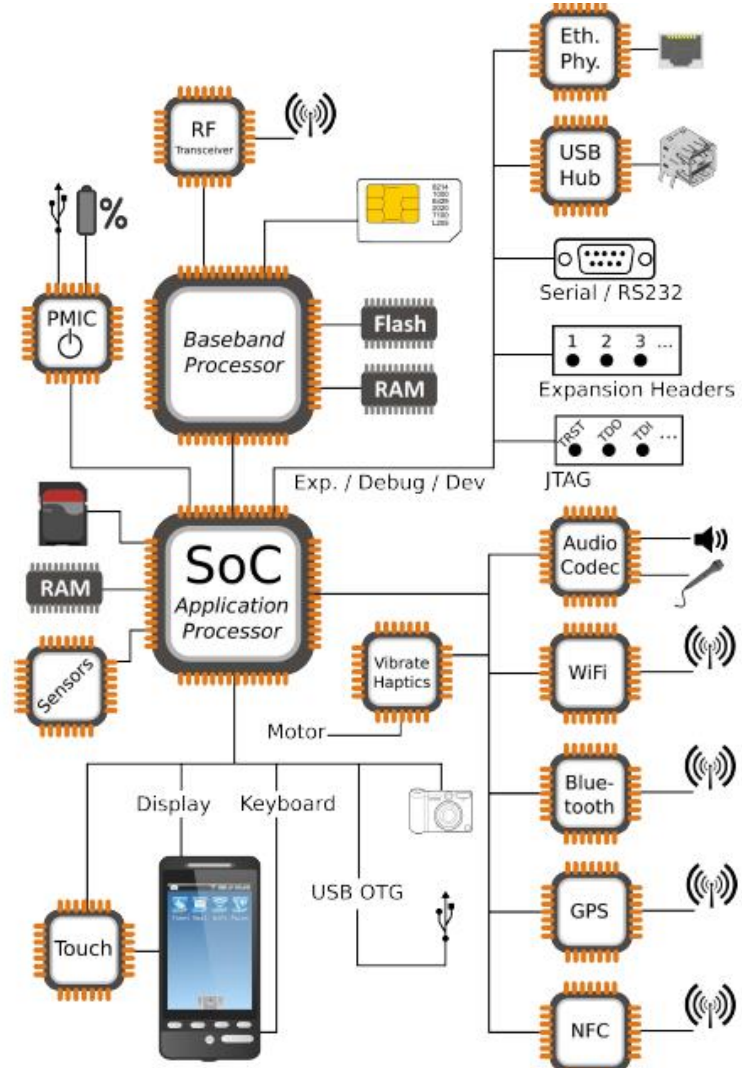
Custom Application(s)

Libc

uClibc / eglibc / musl / glibc

Linux kernel

Process & Memory mgmt, HW support, …



Embedded Linux system development

+ Build Host to create cross-chain:
binutils, kernel headers, C/C++ libraries, gcc, gdb

Development Host

- **Cross-compilation toolchain**
- **Debugger**
- **Misc. tools**

JTAG, Serial

Ethernet

Embedded System Target

Application

Application

Library

Library

Library

C Library

Linux OS kernel + drivers

Bootloader

- Board support package (BSP) development
- System integration
- Development of applications

Embedded Linux := the usage of the Linux kernel and various open-source components in embedded systems

Board Support Package (BSP)

- Collection of components specific to a hardware platform
  - Bootloader (e.g., U-Boot)
  - OS kernel with hardware-specific drivers
  - Device tree files
  - Hardware abstraction layers
  - Flash memory layout definitions
- Provided by board vendor or created by development team

Device Tree

- Data structure that describes HW components in a system
  - separates hardware-specific configuration from the kernel code
  - Before device trees, HW details were hardcoded in the kernel
    - … requiring different kernel builds for different boards using the same SoC
  - With device trees, a single kernel binary can support multiple hardware configurations through externalized HW descriptions.
- Hierarchically organized
  - Nodes, each with Properties (key-value pairs)
  - Paths identify Nodes in the hierarchy

```
# dtc -I dts -O dtb -o output.dtb input.dts
# cpp -nostdinc -I include -undef -x assembler-with-cpp input.dts | dtc -I dts -O dtb -o output.dtb
```

Device Tree Example

```
cpus {   #address-cells = <1>;       #size-cells = <0>;
  cpu@0 { compatible = "arm,cortex-a9";  reg = <0>; };
  cpu@1 { compatible = "arm,cortex-a9";  reg = <1>;  }; };

memory@80000000 {  device_type = "memory";
  reg = <0x80000000 0x20000000>; /* 512 MB */    };

uart@10009000 { compatible = "vendor,uart";
  reg = <0x10009000 0x1000>; interrupts = <36>;
  status = "okay";    };

i2c@10018000 { compatible = "vendor,i2c";
  reg = <0x10018000 0x1000>;   interrupts = <40>; clock-frequency = <100000>;
status = "okay";
   eeprom@50 { compatible = "atmel,24c256";  reg = <0x50>; };  };
```

Example Boot Sequence (Arm)

- 1. **Bootloader** (eg. U-Boot) loads the kernel image and DTB into memory
- 2. Bootloader passes **DTB address** to the kernel
  - via register (r2 on Arm)
- 3. **Kernel** validates DTB & creates internal representation.
- 4. <u>Kernel uses the device tree</u> to:
  - Configure memory regions
  - Identify and initialize platform devices
  - Set up interrupt mappings
  - Detect available buses and connected devices

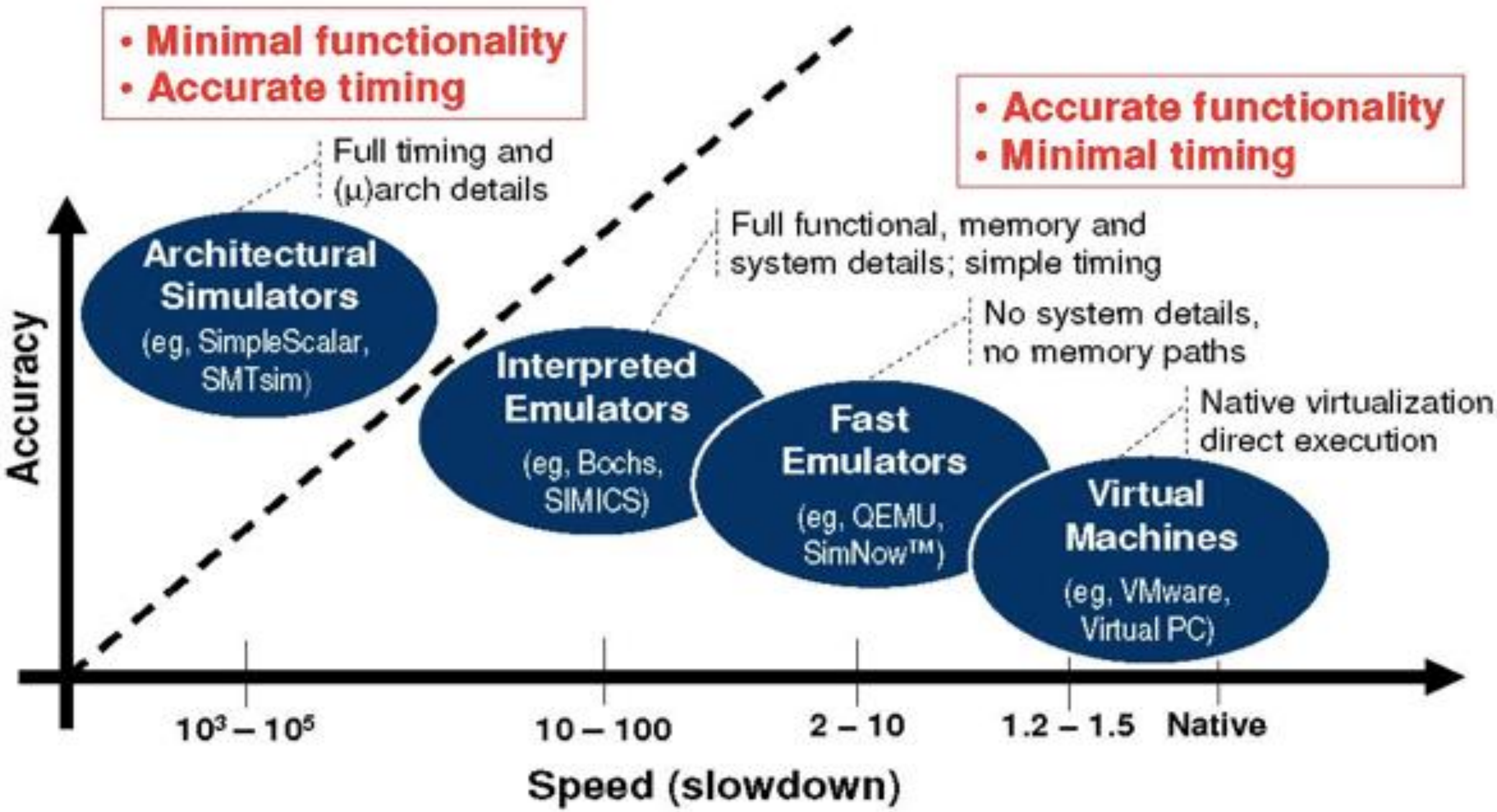
Uses of QEMU (Quick EMUlator)

- Emulate various target architectures
  - ARM, MIPS, PowerPC, RISC-V, x86, …
- Test embedded Linux systems without physical hardware
- Accelerate development and debugging cycles
- Serve as both a target device emulator and build host

Virtualization Definitions

- Virtualization
  - A layer mapping its visible interface and resources onto the underlying layer or system on which it is implemented
  - Purposes: abstraction, replication, isolation
- Virtual Machine (VM)
  - An efficient, isolated duplicate of a real machine
    - Programs should not be able to distinguish between execution on real or virtual H/W (except for: fewer/variable resources, and device timing)
    - VMs should execute without interfering with each other
    - Efficiency requires that most instructions execute directly on real H/W
- Hypervisor / Virtual Machine Monitor (VMM)
  - Partitions a physical machine into multiple “virtual machines”
    - Host : machine and / or software on which the VMM is implemented
    - Guest : the OS which executes under the control of the VMM

Virtualization alternatives & their performance



FORTH-ICS Institute of Computer Science

OS vs Hypervisor (VMM)

- Hypervisor / Virtual Machine Monitor (VMM)
  - Software that supports virtual machines on a physical machine
  - Determines how to map VM resources to physical ones
  - Physical resources may be time-shared, partitioned, or emulated
- The OS has complete control of the (physical) system
  - Impossible for >1 operating systems to be executing on the same platform
  - OS provides execution environment for processes
- Hypervisor (VMM) "virtualizes" the hardware interface
  - GuestOS's do not have complete control of the system
  - VMM provides execution environment for OS
    - "virtual hardware"

What needs to be emulated for a VM? [Hardware]

- CPU and memory hierarchy
  - ISA, Register state, Memory state
  - Privilege levels, Exceptions/Traps, Interrupts
- Memory Management Unit (MMU)
  - Page tables, segments → virtual memory support
  - Controlled via special registers, and via page tables
- Platform
  - Interrupt controller, timers, peripheral buses
- Firmware (BIOS)
- Peripheral devices
  - Disk, network interface, serial line
  - Programmed I/O, Direct Memory Access (DMA)
  - Events delivered to software via polling or interrupts

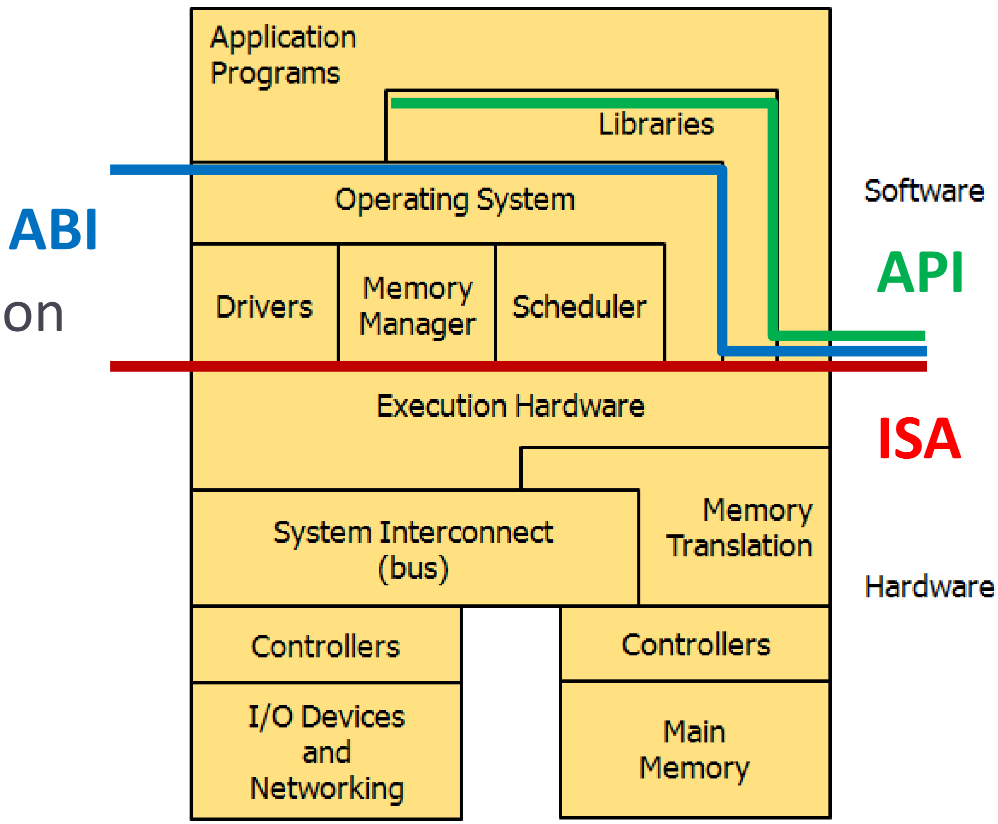
Hardware is not (commonly) designed be multiplexed → Loss of isolation

What needs to be emulated for a VM? [OS, App]

- OS
  - OS issues instructions to control hardware devices
  - … interacts with hardware devices using “sensitive” instructions
  - Allocate and manage hardware resources on behalf of programs
  - … OS runs at higher privilege level than applications
  - Expose system call interface to applications
  - … implemented using low-level H/W interfaces
- Application
  - Relies on the system call interface, runs in unprivileged mode
  - Special instruction(s) to call into OS code
  - OS provides a program with the illusion of its own memory
    - Virtual address spaces (implemented via MMU) → isolation
      - from OS and other App’s
  - Most instructions run directly on the CPU
    - Sensitive instructions cause the CPU to throw an exception to the OS

Computing systems are built on levels of abstraction

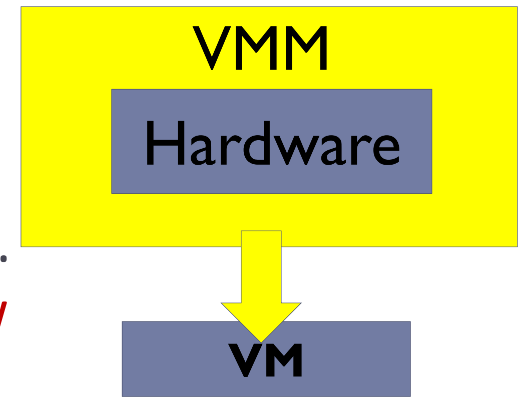
- Different perspectives on what a “machine” is
  - OS → **ISA**: Instruction Set Architecture
    - h/w – s/w interface
  - Compiler → **ABI**: Application Binary Interface
    - User ISA + OS calls
    - Calling conventions
  - Application → **API**: Application Programming Interface
    - User ISA + Library calls



By Glenford Myers (1982)

"Classic" VM (Popek & Goldberg, 1974) (1/4)

- Essentials of a Virtual Machine Monitor (VMM)
  - An efficient, isolated duplicate of the real machine.
  - **Equivalence**
    - Software on the VMM executes identically to its execution on hardware, barring timing effects.
      i.e. Running on VMM == Running directly on HW



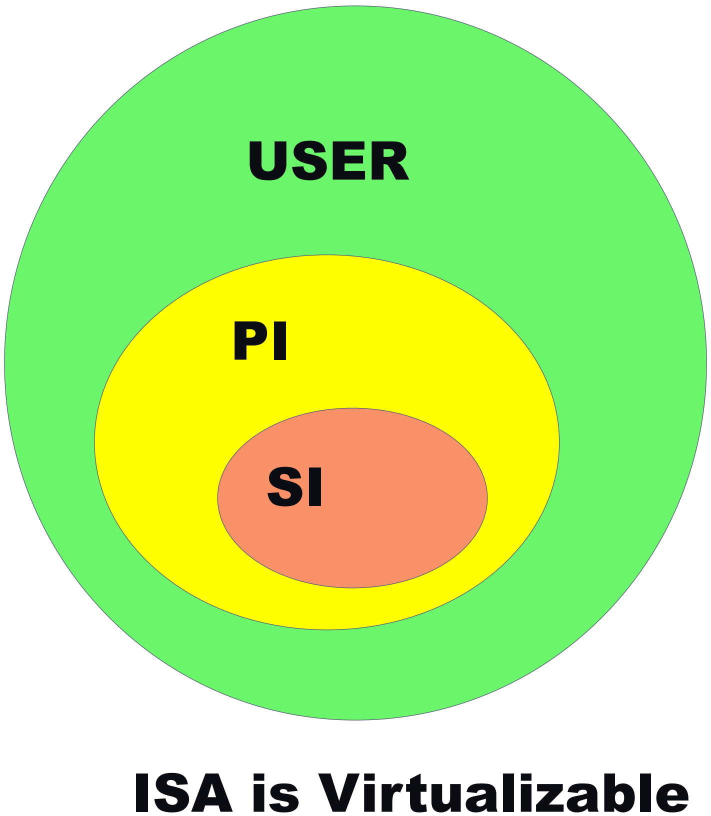
  - **Performance**
    - Non –Privileged instructions can be executed directly by the real processor, with no software intervention by the VMM.
      i.e. Performance on VMM == Performance on HW
  - **Resource control**
    - The VMM must have complete control of the <u>virtualized</u> resources.

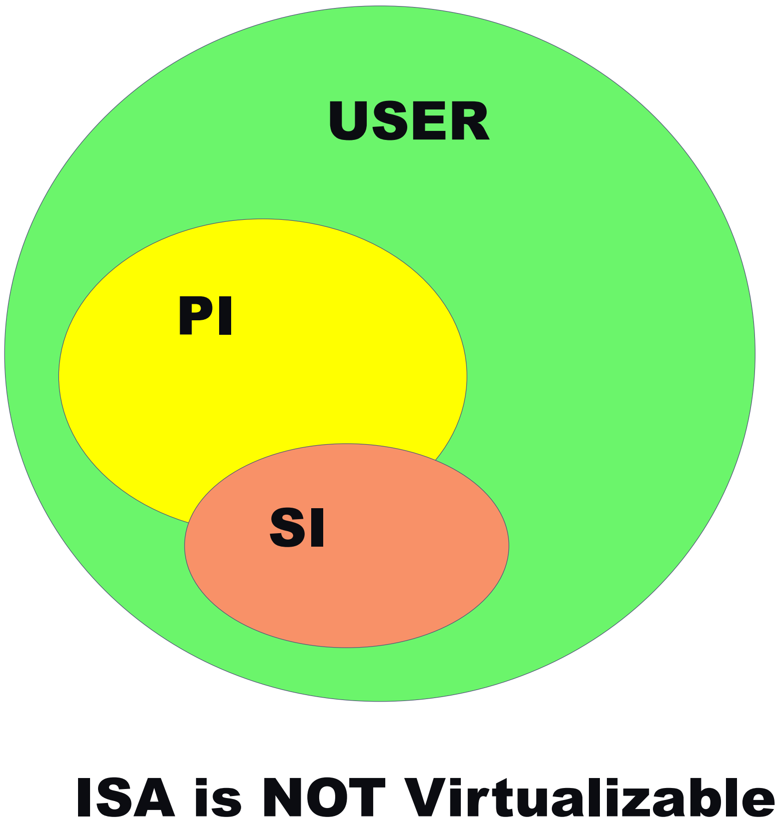
“Classic” VM (Popek & Goldberg, 1974) (2/4)

- Instruction types
  - Privileged instructions: generate trap when executed in any but the most-privileged level
    - Execute in privileged mode, trap in user mode
    - E.g. x86 LIDT : load interrupt descriptor table address
  - Privileged state: determines resource allocation
    - Privilege mode, addressing context, exception vectors, …
  - Sensitive instructions: instructions whose behavior depends on the current privilege level, or modify H/W state
    - Control sensitive: change privileged state
    - Behavior sensitive: exposes privileged state
    - E.g. x86 POPF : pop stack to EFLAGS (in user-mode, the ‘interrupt enable’ bit is not over-written)

“Classic” VM (Popek & Goldberg, 1974) (3/4)

Theorem 1: A VMM may be constructed if the set of SI’s is a subset of the set of PI’s





“Classic” VM (Popek & Goldberg, 1974) (4/4)

- To build a VMM, it is sufficient for all instructions that affect the correct functioning of the VMM (SI’s) always trap and pass control to the VMM.
  - This guarantees the “resource control property”
  - Non-privileged instructions are executed without VMM intervention
  - Equivalence property: We are not changing the original code, so the output will be the same.

Mostly-virtualizable Architectures ☹

- x86
  - Sensitive push/pop instructions are not privileged
  - Segment and interrupt descriptor tables in virtual memory
- Itanium
  - Interrupt vectors table in virtual memory
- MIPS
  - User-accessible kernel registers k0, k1 (save/restore state)
- ARM
  - PC is a general-purpose register
  - Exception returns to PC (no trap)

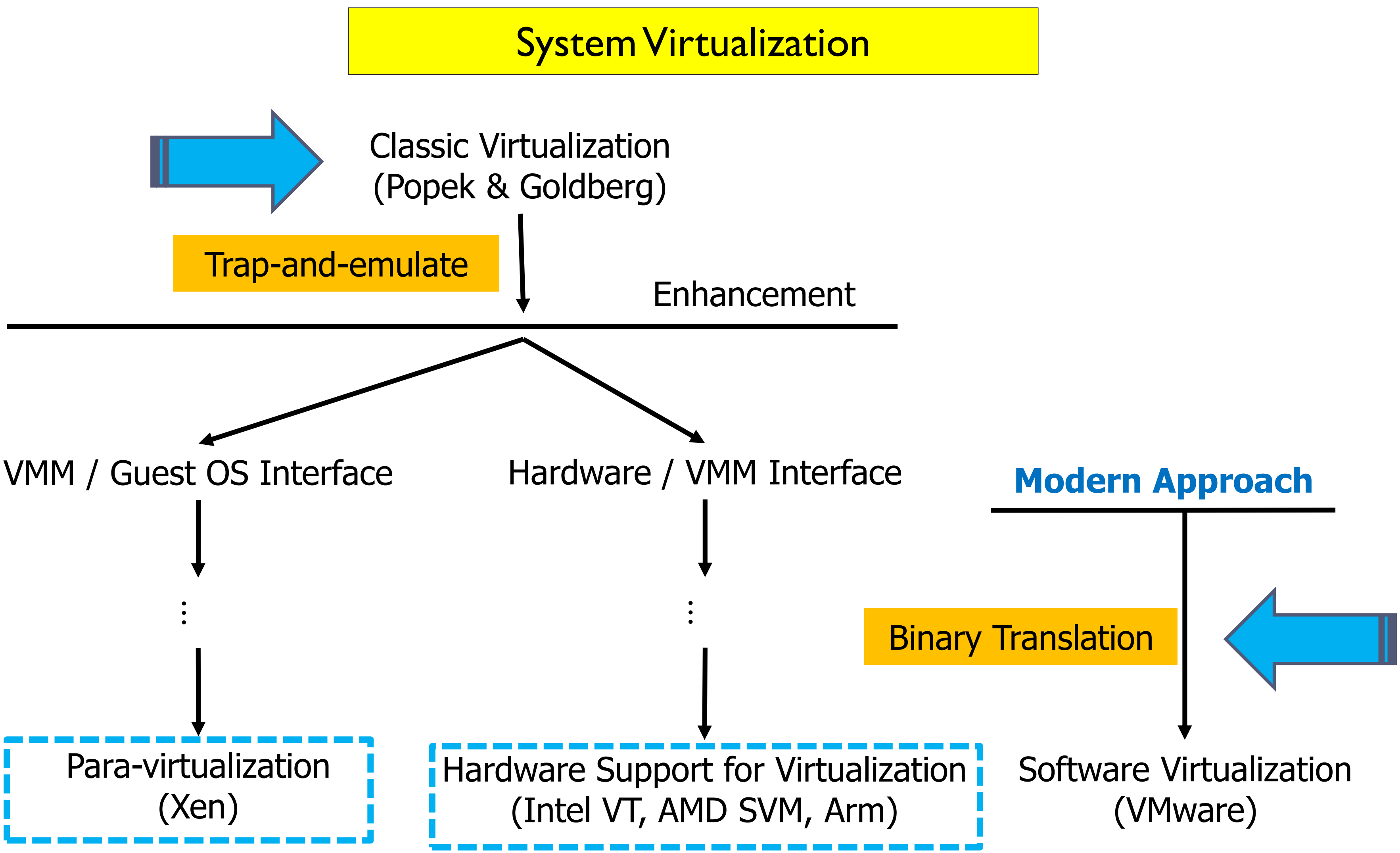
Virtualization overheads

- VMM maintains virtualized privileged machine state
  - Processor status, addressing context, device state, …
- VMM emulates privileged instructions
  - Translation between virtual and real privileged state
    - E.g. guest-to-real page tables
- Traps are expensive
  - Several 100s cycles (for x86)
- Certain important OS operations involve several traps
  - Interrupt enable/disable for mutual exclusion
  - Page table setup/updates for fork()

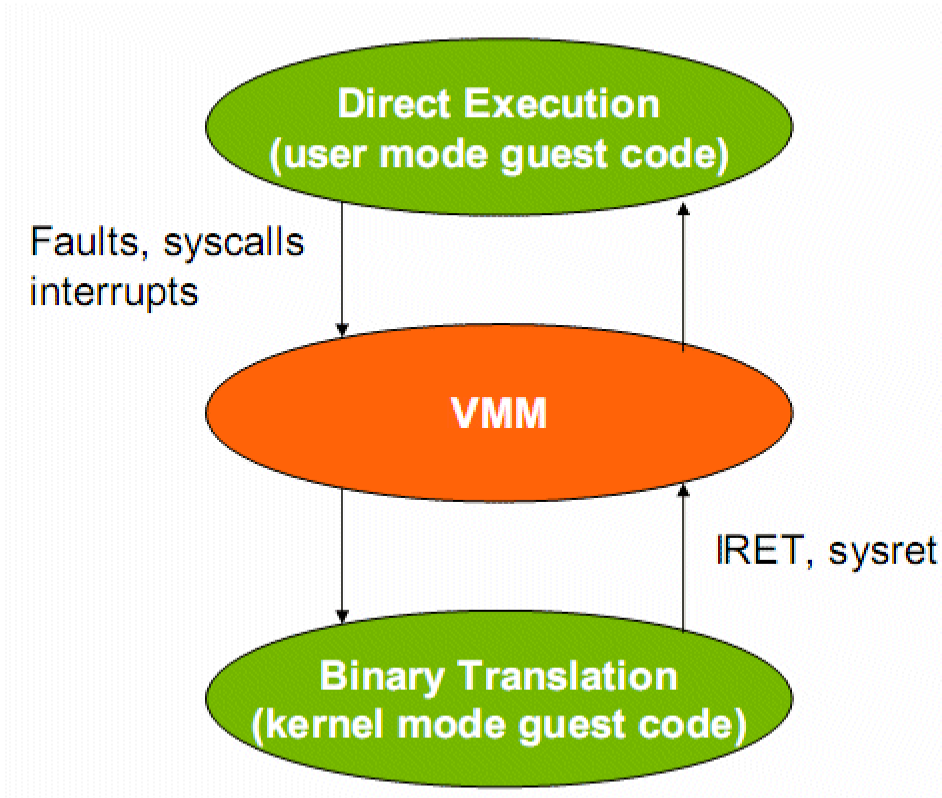
How to achieve safe –and- fast virtualization?

- Emulation
  - Interpret each instruction
- Paravirtualization
  - Modify the guest OS to avoid non-virtualizable instructions
- Binary translation (instead of trap-and-emulate)
  - Static vs Dynamic
- Change processor architecture
  - Intel VT , AMD Pacifica → extend x86 to make "Classic Virtualization" possible [VM/370 origins !]
  - Add a new CPU mode to distinguish VMM from guest/app

Evolution of System Virtualization



Binary Translation



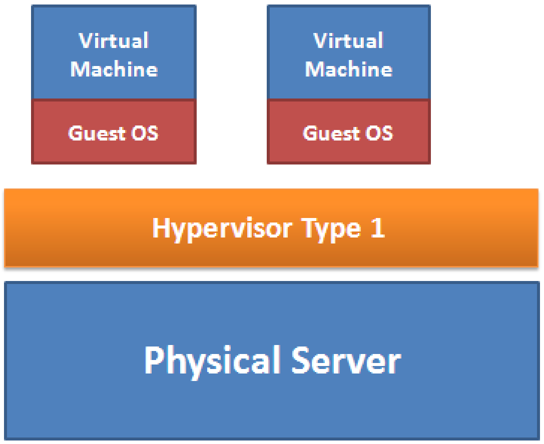
+ translator cache

+ trace cache

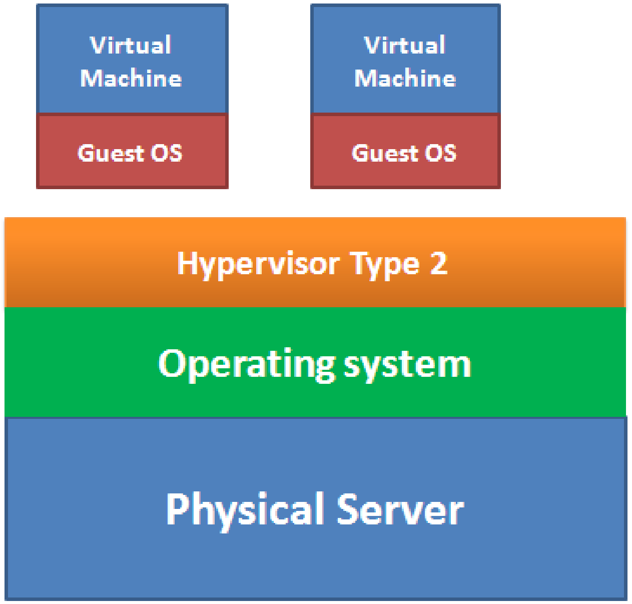
- User applications are not translated, but run directly.
- Binary Translation only happens when the guest OS kernel gets called.

Hypervisor (VMM) types

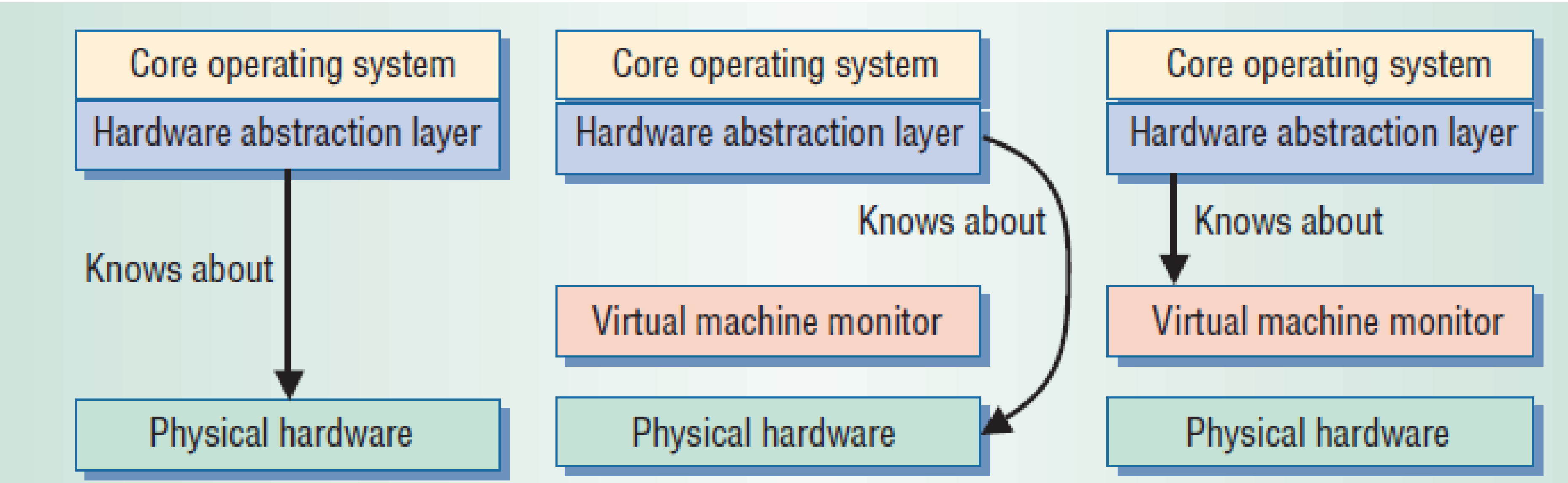
- Type I: run directly on hardware (minimal OS)
  - Bare-metal (minimal OS)
  - e.g. XEN (Citrix XenServer), Microsoft Hyper-V, IBM LPAR, VMware ESXi (vmkernel)
  - Monolithic (kernel + device drivers+ I/O stack)
  - Microkernel –based: I/O stack and HW-specific device drivers in "parent" partition



- Type II: run on host OS
  - Hosted
  - one user-space process, or one user-space process per VM
  - e.g. VMware Workstation, VirtualBox, KVM (Linux), QEMU



VMM architectures



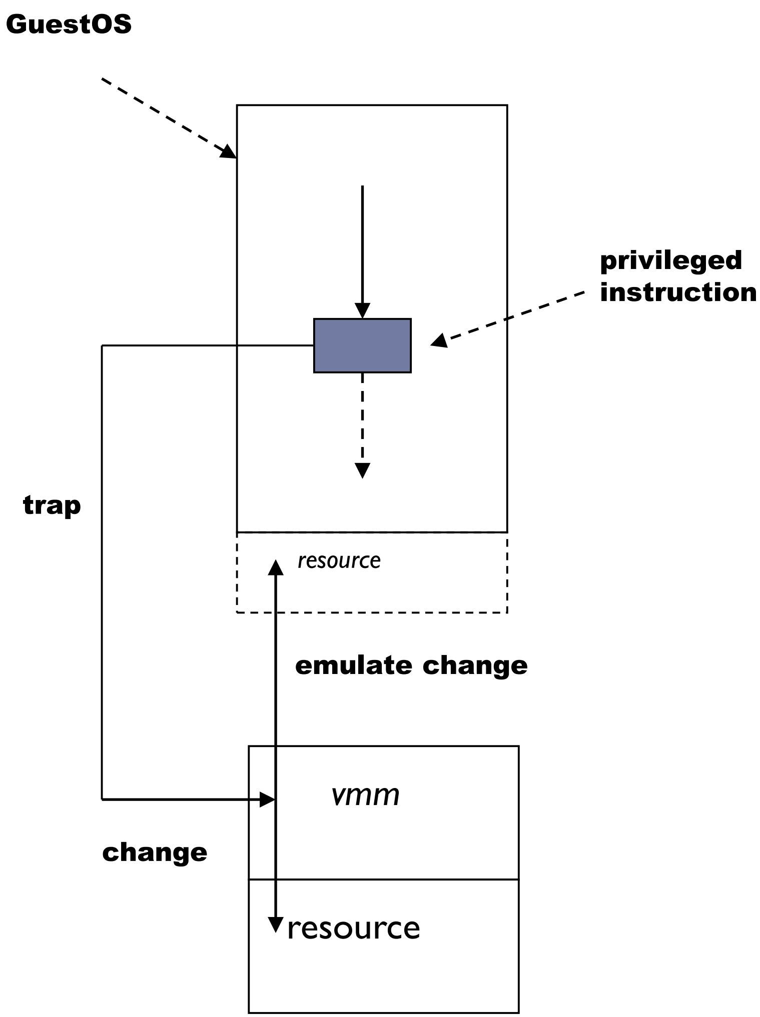
Only OS knows about H/W **Unmodified view of H/W** **Modified view of H/W**

Paravirtualized VMM

VMM provides a HW/SW interface to guest OSs :

- Full virtualization: trapping & emulating sensitive instructions
- Para-virtualization: OS-assisted ("hyper-calls")
- HW-accelerated virtualization (unmodified Guest OS)

Key Techniques (1/3): De-privileging

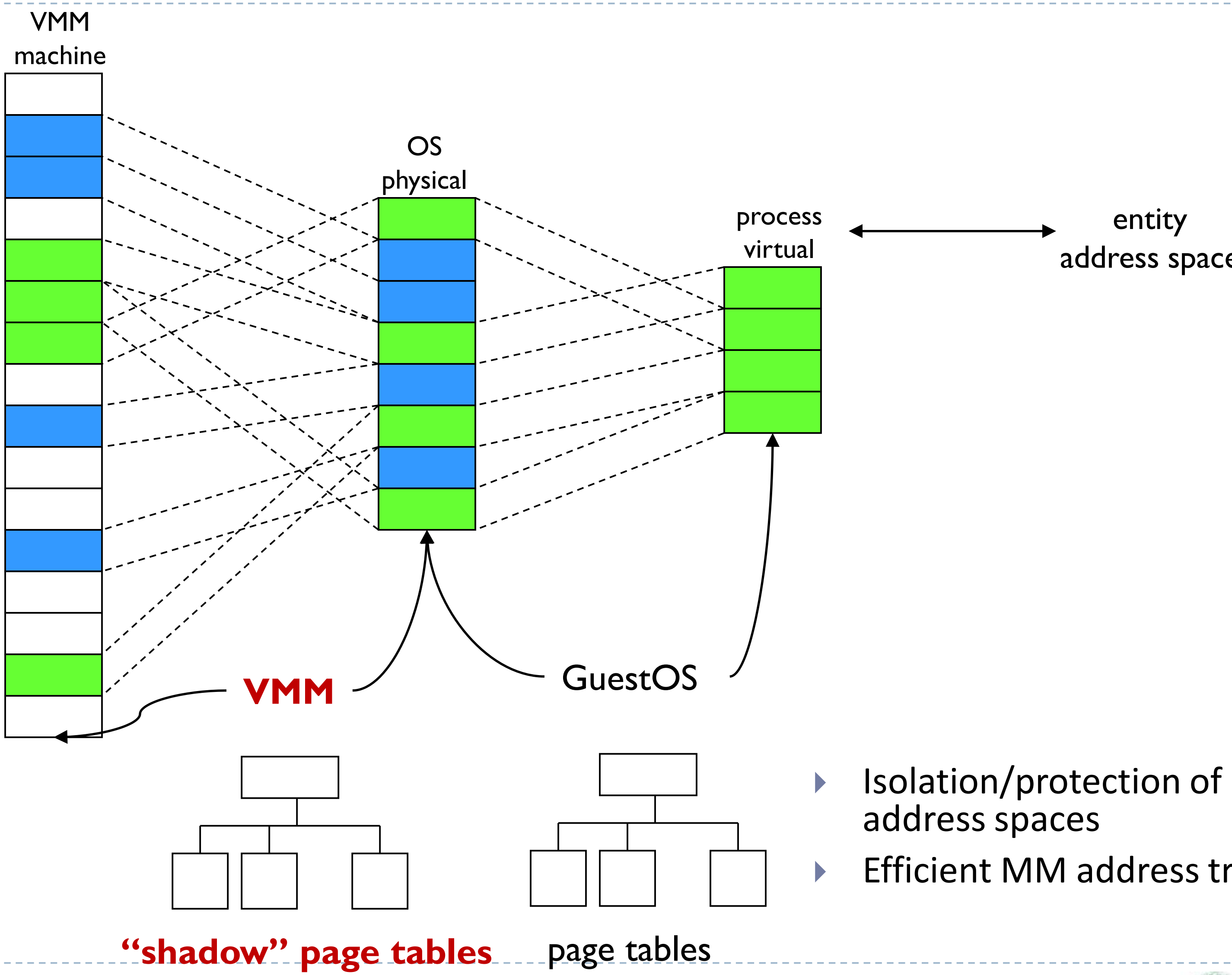


- VMM emulates the effect on system/hardware resources of privileged instructions whose execution traps into the VMM
  - aka **trap-and-emulate**
- Typically achieved by running GuestOS at a lower hardware priority level than the VMM
  - “Normal” instructions run directly on processor
  - “Privileged” instructions trap into VMM (for safe emulation)
- Problematic on architectures where privileged instructions do not trap when executed at deprivileged priority!

Key Techniques (2/3): Primary vs Shadow Structures

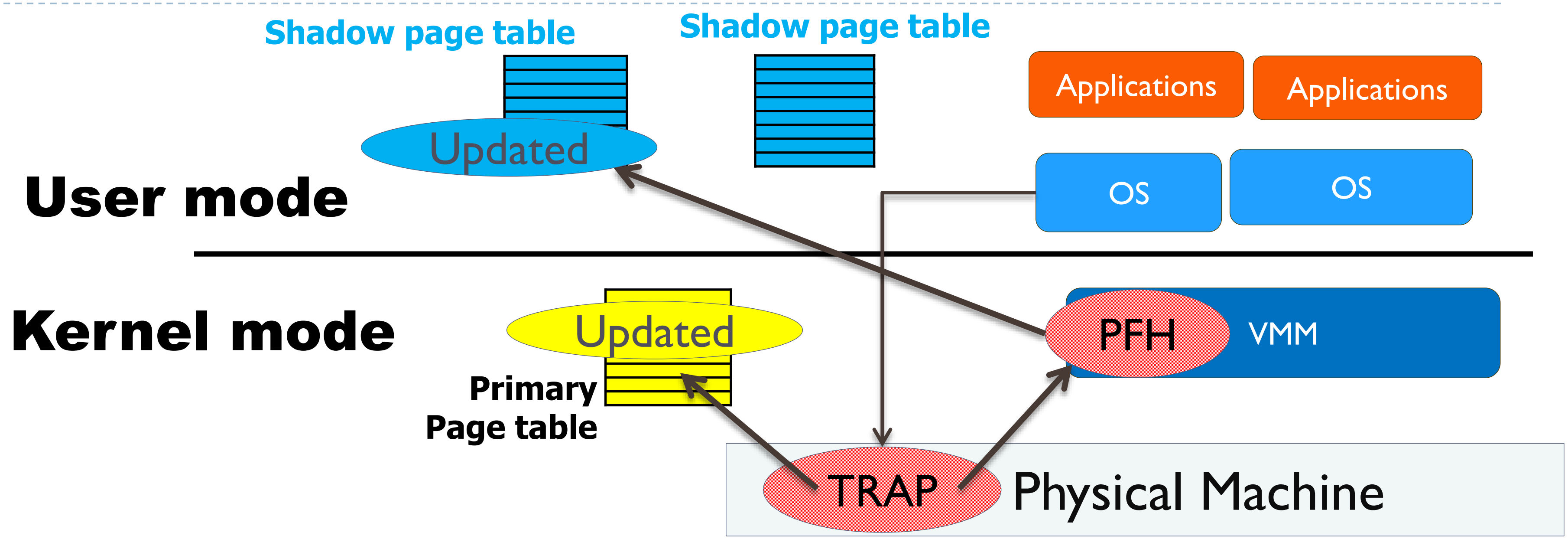
- VMM maintains "shadow" copies of critical structures whose "primary" versions are manipulated by GuestOS
  - e.g., page tables
- Primary copies needed to insure correct environment visible to GuestOS

Memory Management by the VMM



- Isolation/protection of Guest OS address spaces
- Efficient MM address translation

Key Techniques (3/3): Memory Tracing (Trace faults)



- Control access to memory so that the shadow and primary structures remain coherent
  - Write-protect primary structure so that update operations cause page faults → caught, interpreted, emulated by the VMM
  - VMM typically use hardware page protection mechanisms to trap accesses to in-memory primary structures

Sources

- James E. Smith, Ravi Nair, **The Architecture of Virtual Machines**, IEEE Computer, vol.38, no.5, May 2005
- Mendel Rosenblum, Tal Garfinkel, **Virtual Machine Monitors: Current Technology and Future Trends**, IEEE Computer, May 2005.
- A. Whitaker, R.S. Cox, M. Shaw, S.D. Gribble, **Rethinking the Design of Virtual Machine Monitors**, IEEE Computer, vol.38, no.5, May 2005.
- Kirk L. Kroeker, **The Evolution of Virtualization**, CACM, vol.52, no. 3, March 2009
- G.J. Popek, and R.P. Goldberg, **Formal Requirements for Virtualizable Third Generation Architectures**, CACM, vol. 17 no. 7, 1974.
- Jim Smith and Ravi Nair, **Virtual Machines: Versatile Platforms for Systems and Processes**, ISBN-10: 1558609105, Elsevier, 2005