Virtualization

MOS WS 2020/21

Goals

- Give you an overview about:
  - Virtualization and VMs in General
  - Hardware Virtualization on x86

Goals

- Give you an overview about:
  - Virtualization and VMs in General
  - Hardware Virtualization on x86
- Not in this lecture:
  - Lots and lots of Details
  - Language Runtimes
  - How to use Xen/KVM/…

History

Erik Pitti, CC-BY, www.flickr.com/people/24205142@N00

History

- Pioneered with IBM's CP/CMS in ~1967 running on System/360 and System/370
- CP: Control Program (provided S/360 VMs)
  - Memory Protection between VMs
  - Preemptive scheduling
- CMS: Cambridge Monitor System (later Conversational Monitor System) – Single User OS
- At the time more flexible & efficient than time-sharing multi-user systems!

Applications

- Consolidation (improve server utilization)
- Isolation (incompatibility or security reasons)
- Reuse (legacy software)
- **Development**

… but was confined to the mainframe-world for a long time!

Why?

Imagine you want to write an operating system, that is:

- Secure
- Trustworthy
- Small
- Fast
- Fancy

but, …

Why?

Users expect to run their favourite software („legacy“):

- Browsers
- Word
- iTunes
- Certified Business Applications
- Gaming (Windows/DirectX to DOS)

Porting/Rewriting is not an option!

Why?

„By virtualizing a commodity OS […] we gain support for legacy applications, and devices we don’t want to write drivers for.“

„All this allows the research community to finally escape the straitjacket of POSIX or Windows compatibility […]“

Roscoe, Elphinstone, and Heiser, 2007

What is Virtualization?

Suppose you develop on your x86-based workstation running a system ***H****ost,* a system ***G****uest* which is supposed to run on ARM-based phones.

An emulator for G running H precisely emulates G's:

- CPU
- Memory (subsystem)
- I/O devices

Ideally, programs running on the emulated G exhibit the same behaviour, except for timing, as when run on a real system G.

What is Virtualization?

The emulator:

- interprets every instruction in software as it is executed,
- prevents G to access H’s resources directly,
- maps G’s devices onto H’s devices,
- may run multiple times on H.

What is Virtualization?

Emulation:

- Different Instruction Sets

- Different Hardware Devices

➡ Emulation can be slow & and complex, depending on fidelity.

What is Virtualization?

- What if H=G?
- Interpreting/Emulating every instruction unnecessary?
- Faster?

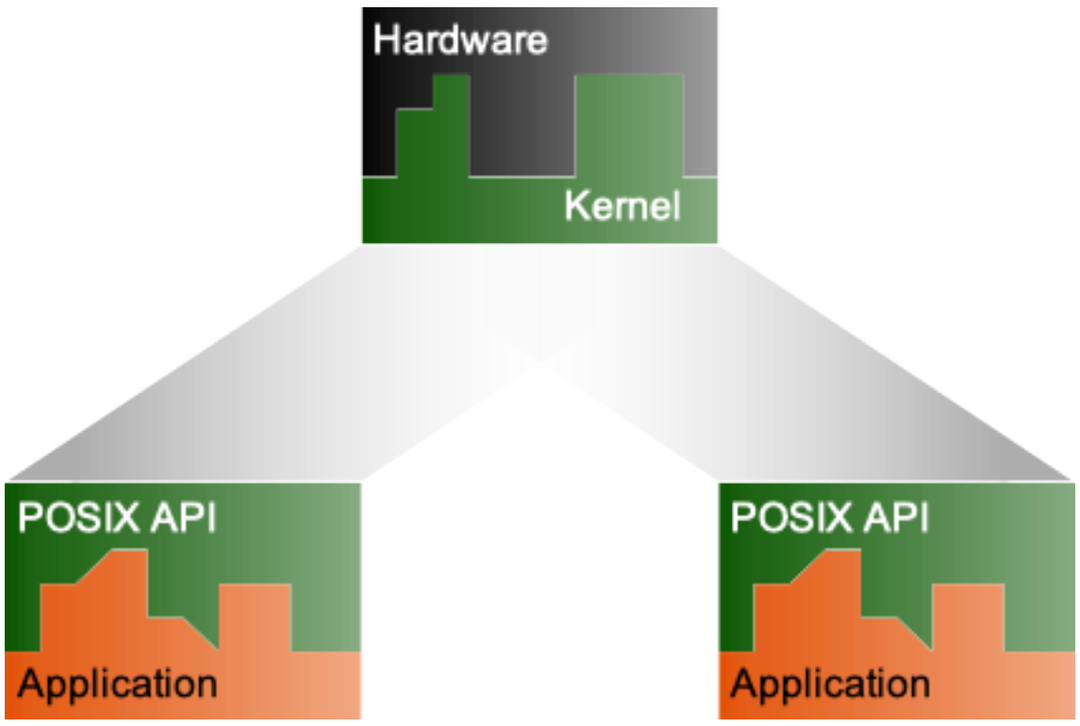
How?

- Guest runs as normal user process

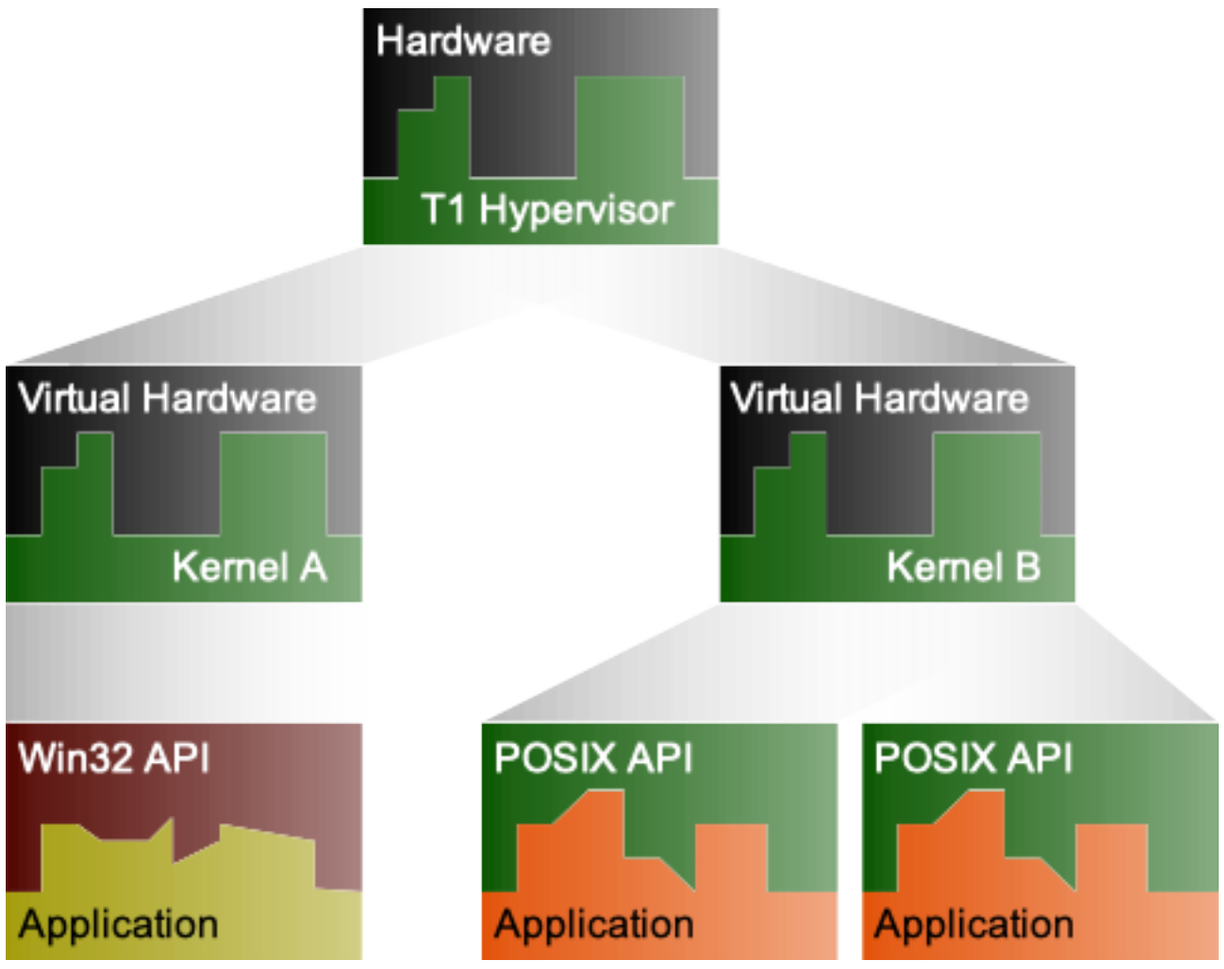
How?

- Guest runs as normal user process
- It’s not just instructions! We need to *emulate* virtual hardware. The software providing the illusion of a real machine is the *Virtual Machine Monitor* (VMM)

Where to put the VMM?

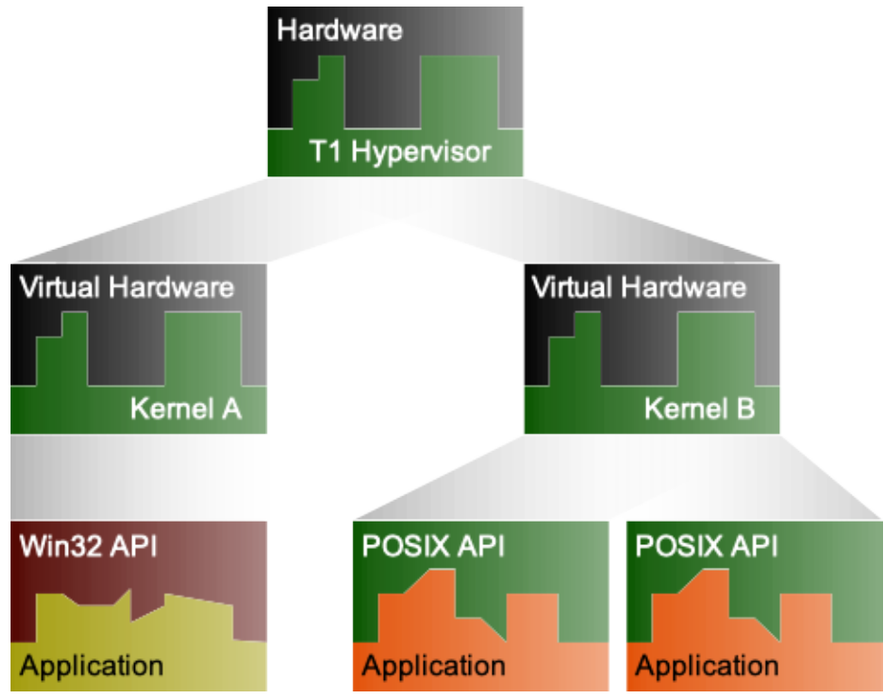


Type-1 Hypervisor

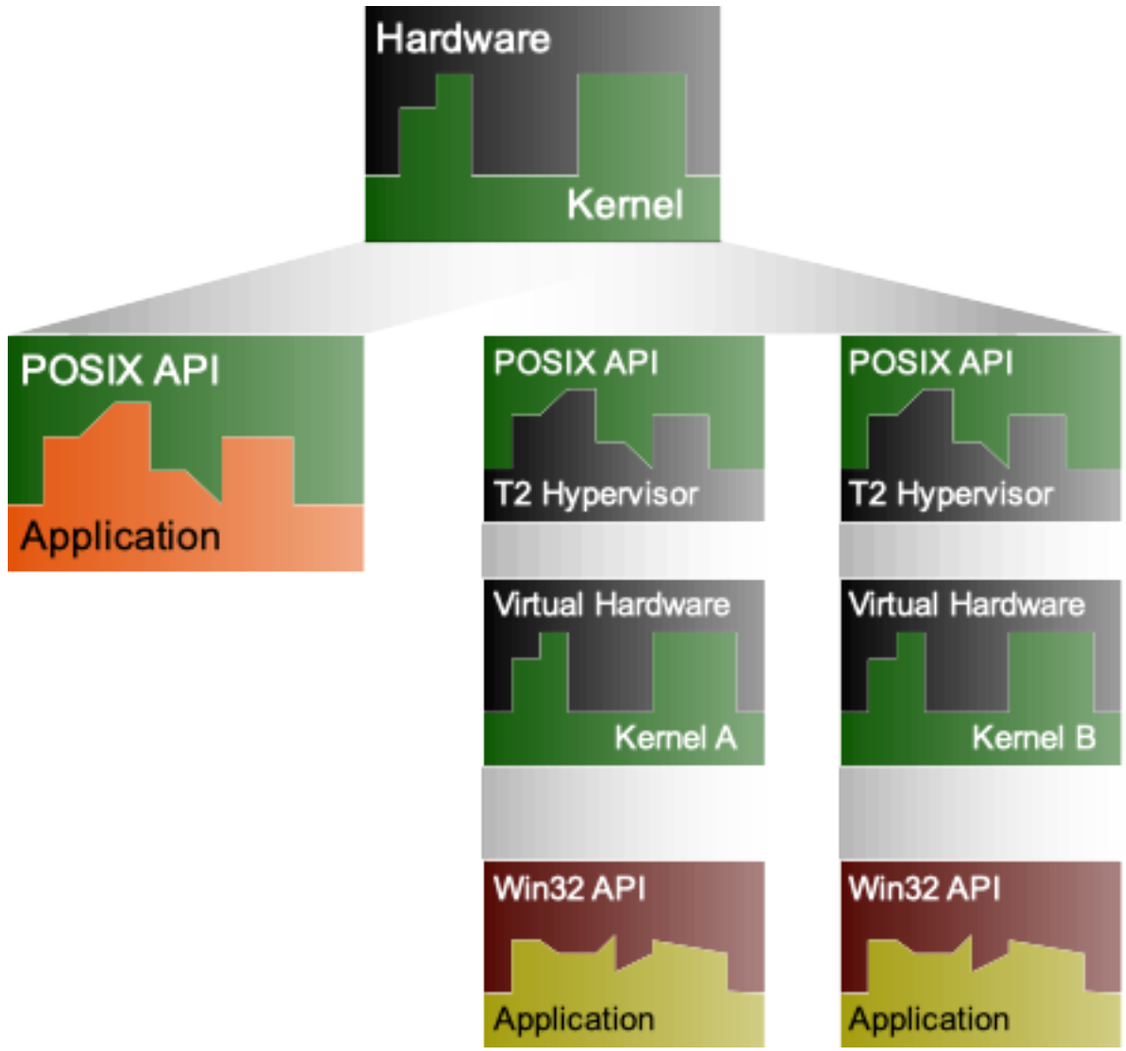


Type-1 Hypervisor

- „Bare-metal“ Hypervisors
- No OS-Overhead
- Complete Control over Host ressources
- High maintenance
- Examples: Xen, VMWare ESXi

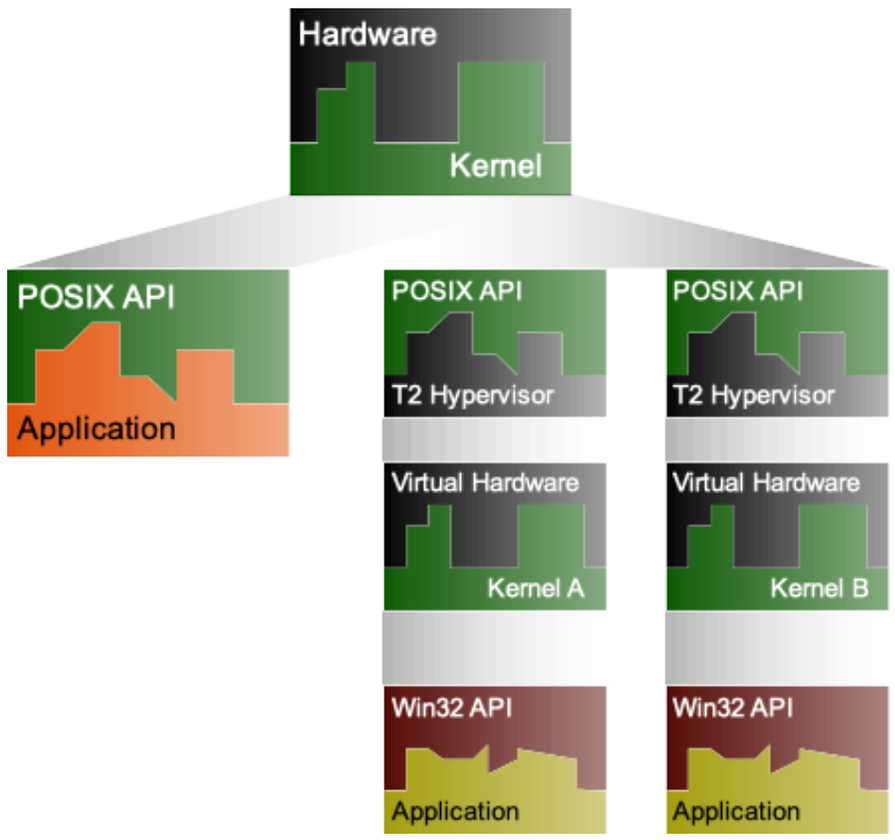


Type-2 Hypervisor



Type-2 Hypervisor

- „Hosted“ Hypervisor
- Doesn’t re-invent the wheel
- Performance tradeoff
- Requires Host-OS support for CPU’s virt features
- Examples: KVM, VMware Server/Workstation, VirtualBox, …



Alternative: Paravirtualisation

Too complicated? Just „port“ the Guest OS to the interface of your choice!

- Better performance
- Simplify VMM
- Maintenance
- Source of Guest OS required
- Tradeoff: Paravirtualized drivers for I/O performance (KVM virtio, VMware)
- Examples: Usermode Linux, Xen/XenoLinux, DragonFlyBSD, VKERNEL, **L⁴Linux**

Virtualized ABI!

Why deal with the Guest OS-kernel at all? Re-implement it’s interface!

- Example: Wine virtualises Windows ABI
- Run unmodified Windows binaries
- Windows API calls are mapped to Host-OS’s (Linux, MacOS, BSD, …) equivalents
- Huge moving target / maintenance effort!
- API-Virtualization: Re-compile Windows applications from source; link against winelib

Virtualizability

Suppose our ISA has an instruction, `OUT`, that writes to a device in kernel mode.

But we're running (virtualized) in user space … ?

- Just do nothing?
- *Trap* to kernel mode

Virtualizability

VMM needs to handle:

- Address Space changes
- Device accesses
- System calls
- ...

These already *trap* to the host kernel (SIGSEGV).

Virtualizability

Easy, right?

- `push %cs` pushes CS register onto stack

Virtualizability

Easy, right?

- `push %cs` pushes CS register onto stack
- CS register contains current privilege level

Virtualizability

Easy, right?

- `push %cs` pushes CS register onto stack
- CS register contains current privilege level

➡ Virtual Guest in Ring 3 can detect it is not in Ring 0!

Virtualizability

Easy, right?

- `push %cs` pushes CS register onto stack
- CS register contains current privilege level

➡ Virtual Guest in Ring 3 can detect it is not in Ring 0!

➡ Our VM is not a duplicate of a real machine, hence not a VM at all ☹️

Virtualizability

… is a property of the ISA. Instructions are divided into two classes:

Virtualizability

… is a property of the ISA. Instructions are divided into two classes:

- Privileged Instructions
- cause a trap in user mode

Virtualizability

… is a property of the ISA. Instructions are divided into two classes:

- Privileged Instructions
  - cause a trap in user mode
- Sensitive instructions

Virtualizability

… is a property of the ISA. Instructions are divided into two classes:

- Privileged Instructions
  - cause a trap in user mode
- Sensitive instructions
  - Behaviour depends on or changes the processor’s configuration or mode

Virtualizability

An ISA is *virtualizable*, i.e. a VMM can be written, if all sensitive instructions are privileged.

- Execute guest in user/unprivileged mode

Virtualizability

An ISA is *virtualizable*, i.e. a VMM can be written, if all sensitive instructions are privileged.

- Execute guest in user/unprivileged mode
- Emulate instructions that cause traps (***Trap & Emulate***)

Virtualizability

An ISA is *virtualizable*, i.e. a VMM can be written, if all sensitive instructions are privileged.

- Execute guest in user/unprivileged mode
- Emulate instructions that cause traps (***Trap & Emulate***)

➡ „Formal Requirements for Virtualizable Third-Generation Architectures“
Popek & Goldberg, 1974
http://portal.acm.org/citation.cfm?id=361073

Virtualizing x86

- x86 originally not virtualizable (push, pushf/popf, … 17 instructions on the Pentium)
- Trapping is expensive!

Virtualizing x86

- x86 originally not virtualizable (push, pushf/popf, … 17 instructions on the Pentium)
- Trapping is expensive!
- First commercial virtualisation solution for x86: VMware Workstation (~1999)
  - Translate problematic instructions to calls into the VMM on the fly (*Binary re-writing*)
  - Can avoid traps for privileged instructions
  - Performance good, but complex runtime translation engine; only common guests (commercially) supported.
- Examples: KQemu, VirtualBox, Valgrind

Hardware Support

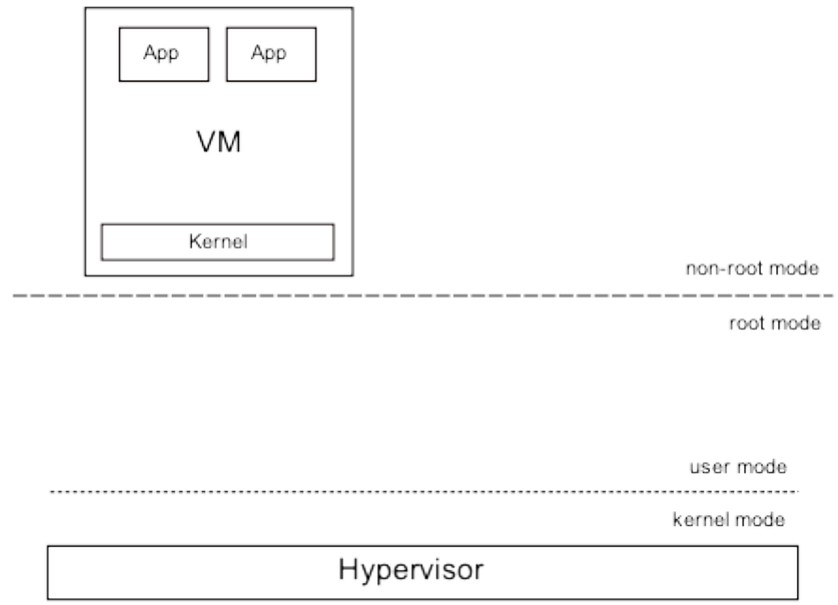
- „Hardware-assisted Virtualization“
- CPU virtualization:
  - All guest instructions are virtualizable
  - Processor provides virtual CPU mode, including kernel mode

Hardware Support

- „Hardware-assisted Virtualization“
- CPU virtualization:
  - All guest instructions are virtualizable
    - Processor provides virtual CPU mode, including kernel mode
- Memory Virtualization
- Typically, VMs have very few (if any) VM-exits for CPU/ memory virtualization

Hardware Support

- Late P4 introduced hardware support in 2004: Intel VT (AMD-V similar)
- *root/non-root mode* duplicate x86 protection rings
- Root mode runs HV, non-root mode runs Guest



Hardware Support

- Late P4 introduced hardware support in 2004: Intel VT (AMD-V similar)
- *root/non-root mode* duplicate x86 protection rings
  - Root mode runs HV, non-root mode runs Guest
  - Everything Intel VT cannot handle traps to root mode
  - Special memory regions (VMCS/VMCB) holds guest state
  - Reduces Software complexity

Instruction Emulation

- Running 16-bit Code (BIOS/Boot loaders)
  - Not in AMD-V/latest Intel VT
- Handling memory-mapped I/O
  - Realized as non-present page
  - Page fault
  - Emulate offending instruction
- …

MMU Virtualization

- Early versions of VT do not virtualise the MMU; VMM has to handle guest virtual memory!

- Four different types of addresses (Host/Guest x Physical/Virtual): hPA, hVA, gPA, gVA

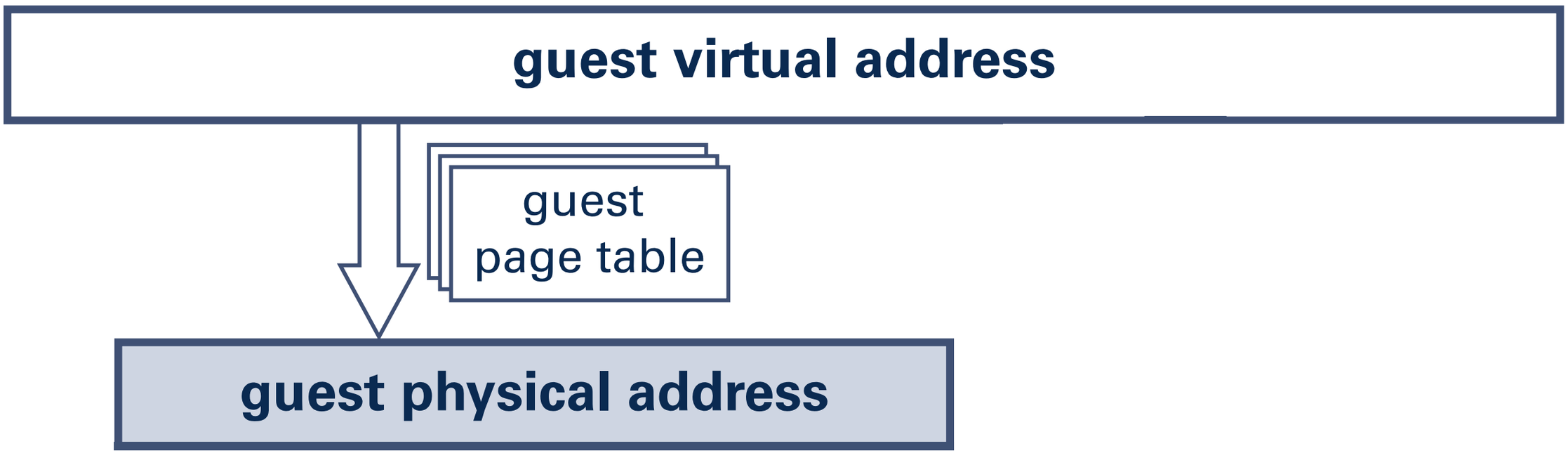
- hVA -> hPA and gVA -> gPA mapped by page tables

- Mapping from Guest-Physical to Host-Virtual usually simple (identity or constant offset)

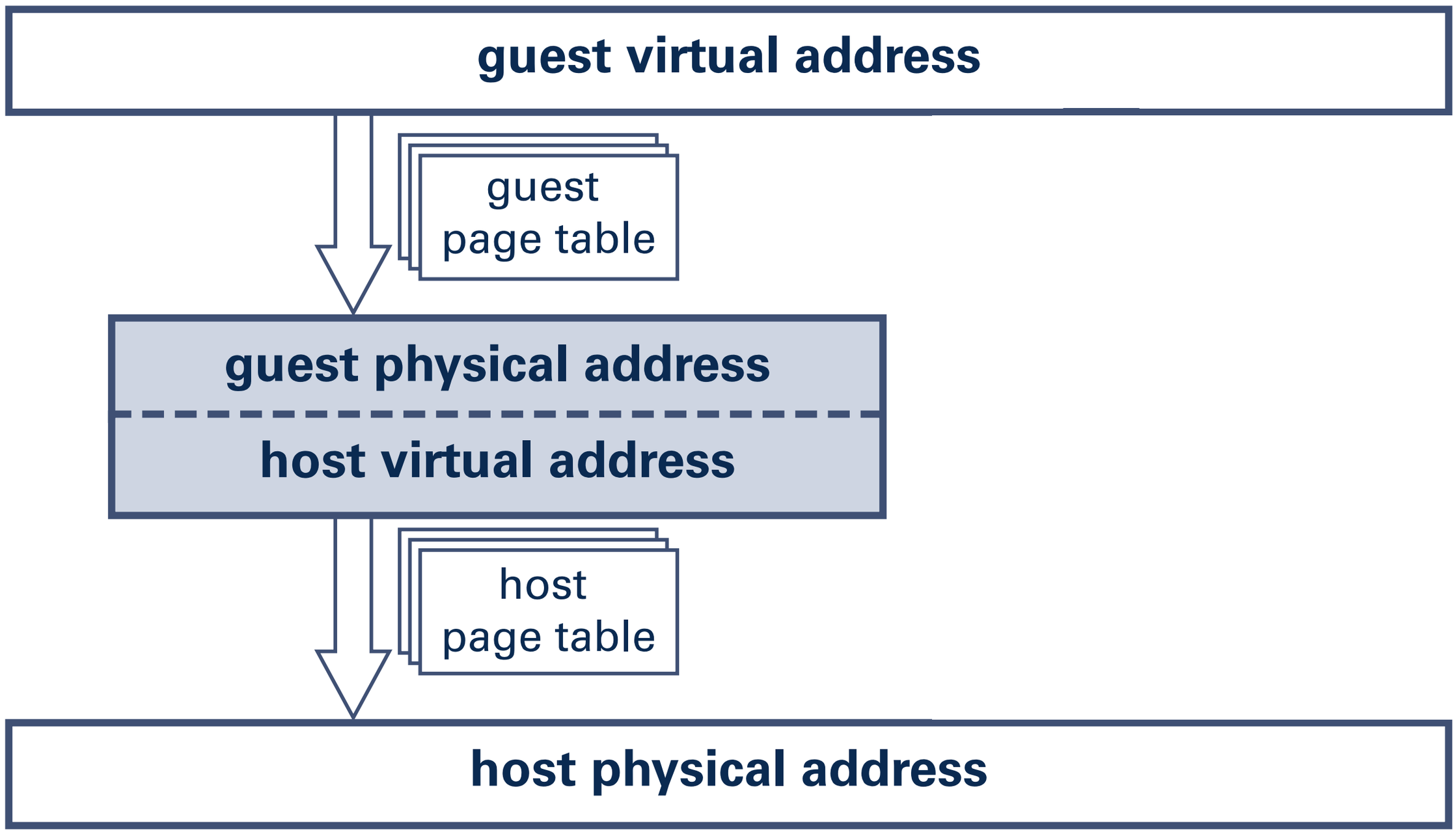
MMU Virtualization

- MMU not virtualized; can handle only one page table
- Hypervisor must maintain a page table, that
  - Maps from Guest Virtual to Host Physical („merging“ guest and host page table)
  - Must be adapted on VM layout changes

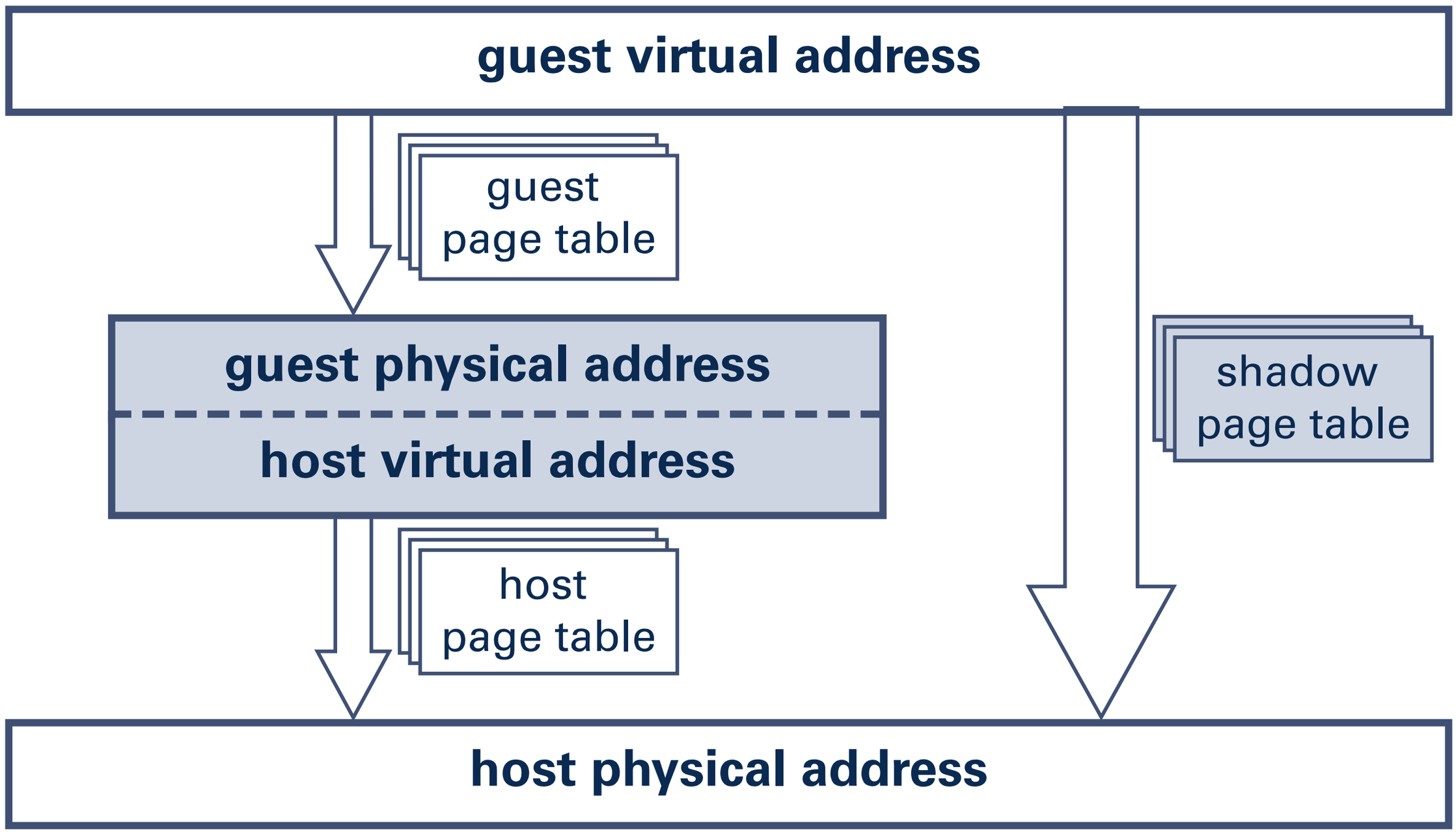
Memory Virtualization



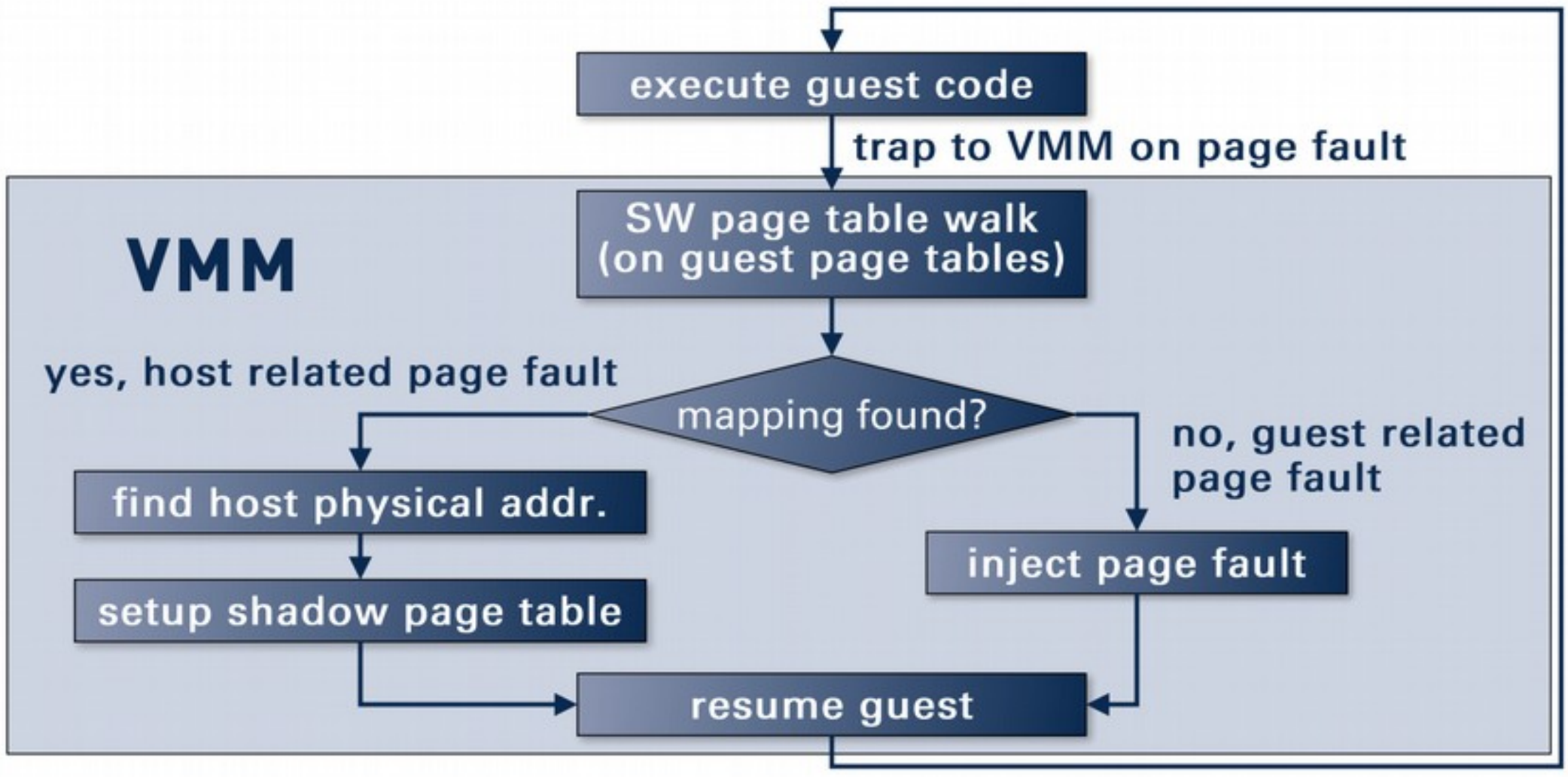
Memory Virtualization



Memory Virtualization



Shadow Paging



Shadow Paging

- Update or re-creation on
  - Guest Page Table modification
  - Guest Address Space switch

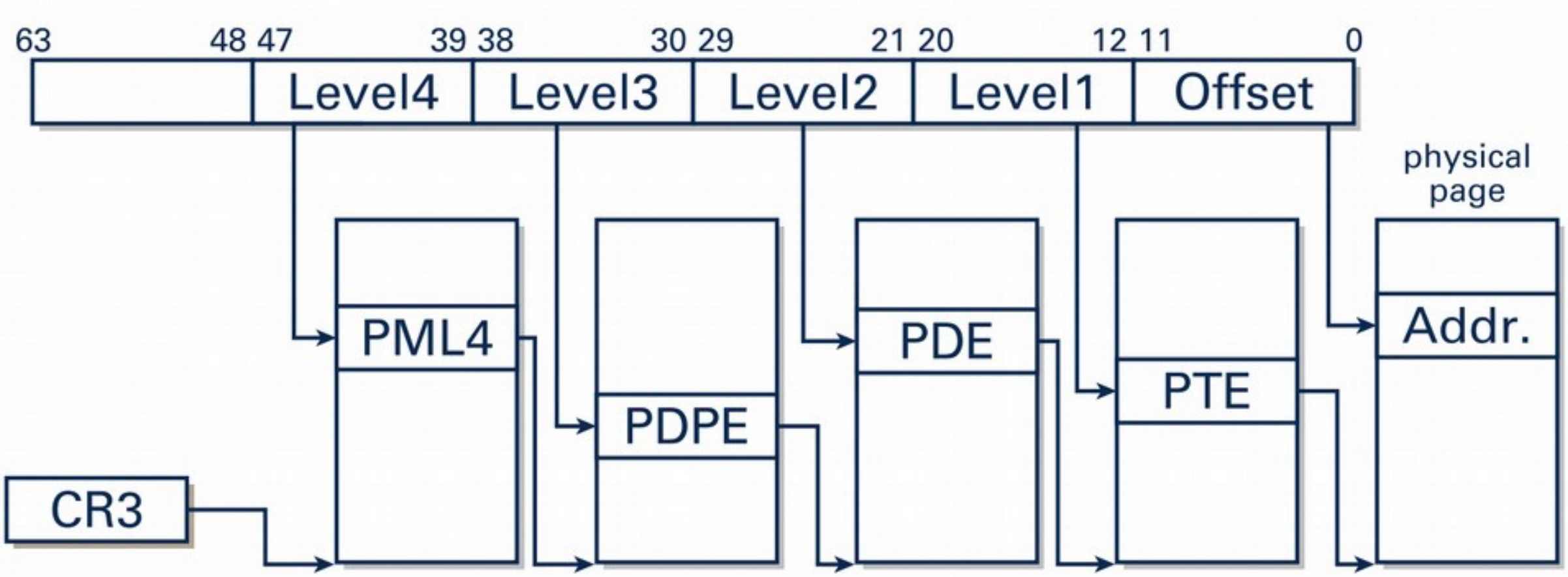
➡ Significant Overhead; certain workloads are penalised

➡ Hardware Support!

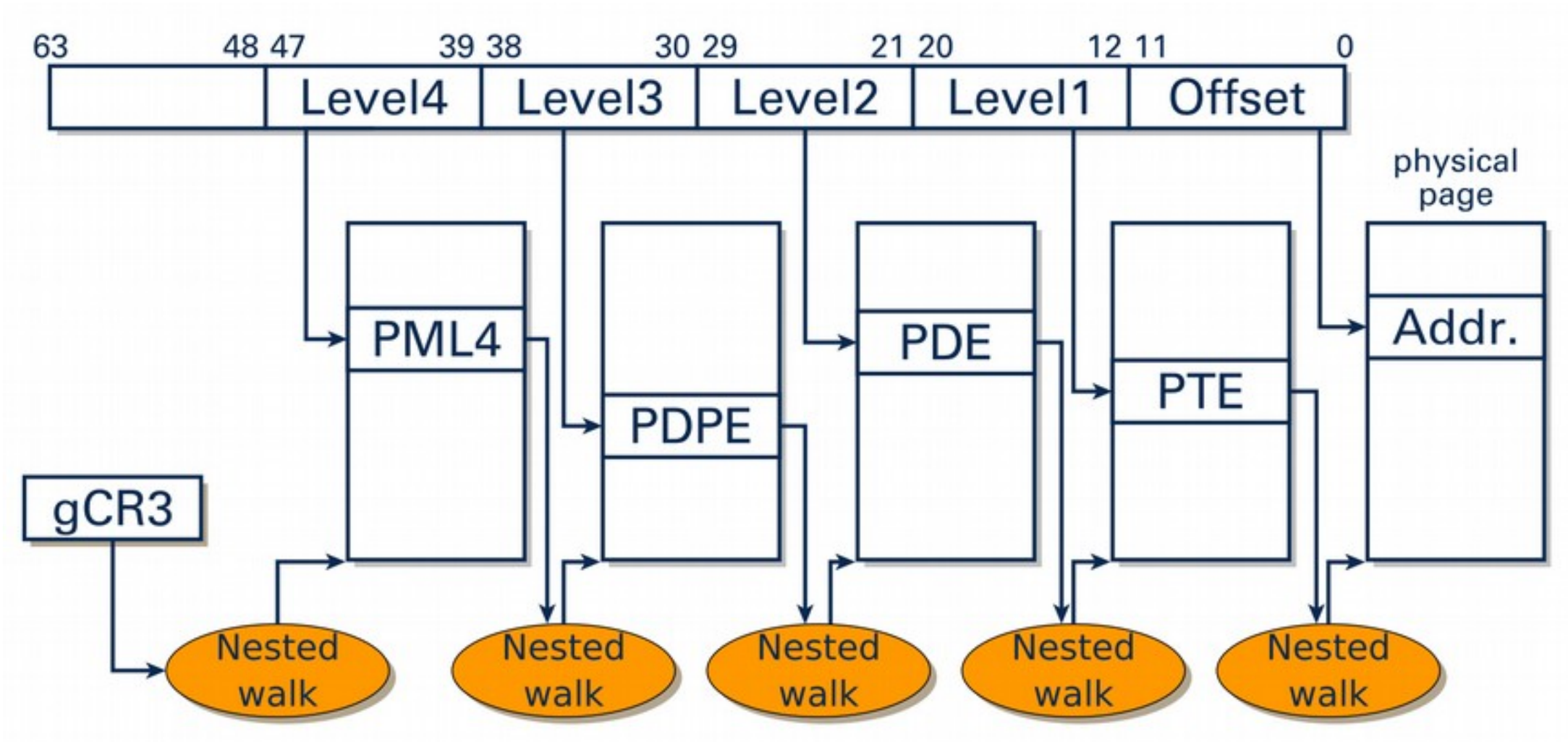
MMU Virtualization

Intel Nehalem (EPT) and AMD Barcelona (Nested Paging) introduce hardware support for MMU virtualisation. The CPU can handle Guest and Host page table at the same time, which can reduce VM Exits by two orders of magnitude but introduces measurable constant overhead (<1%).

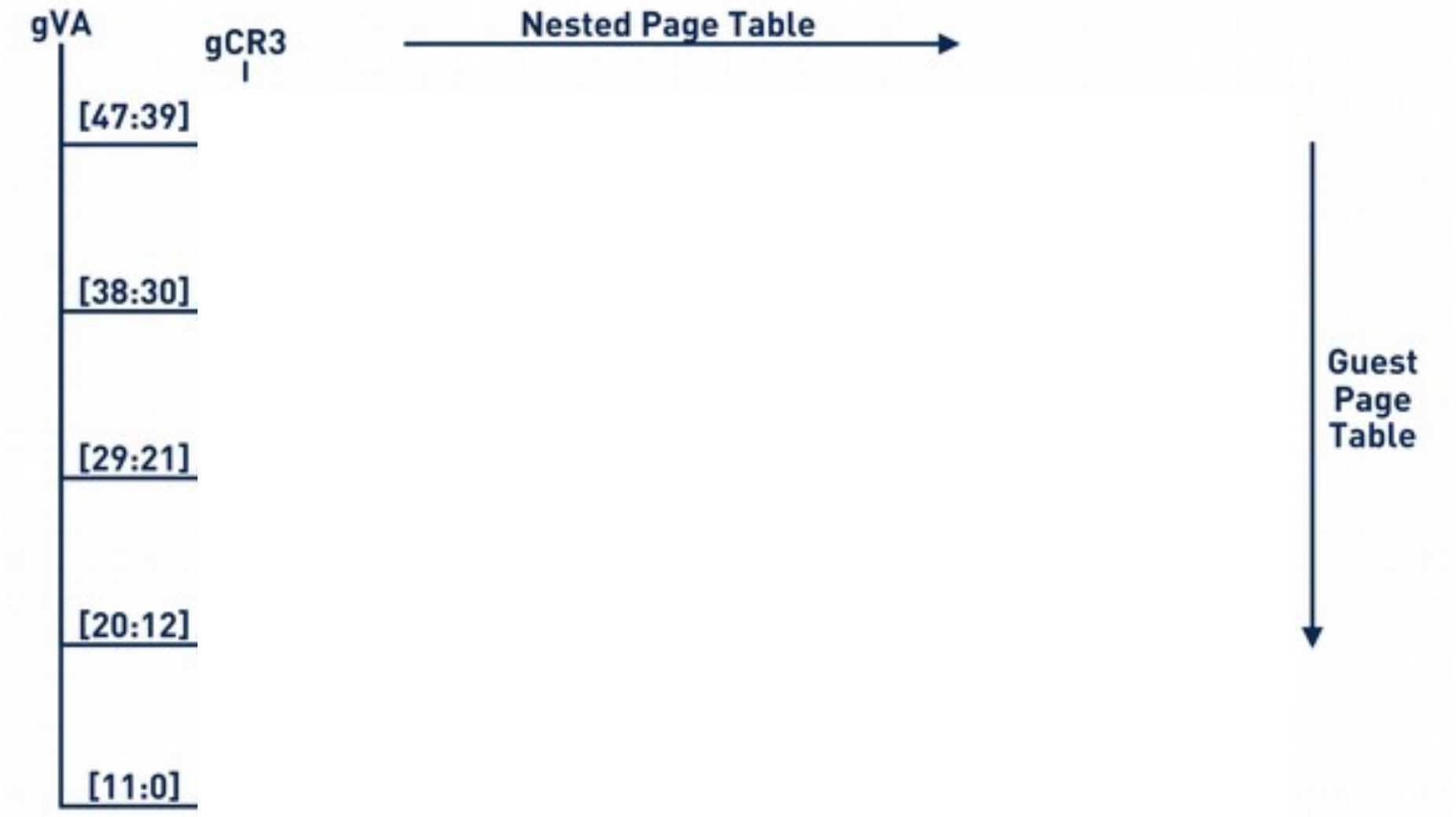
Recap: Address Translation



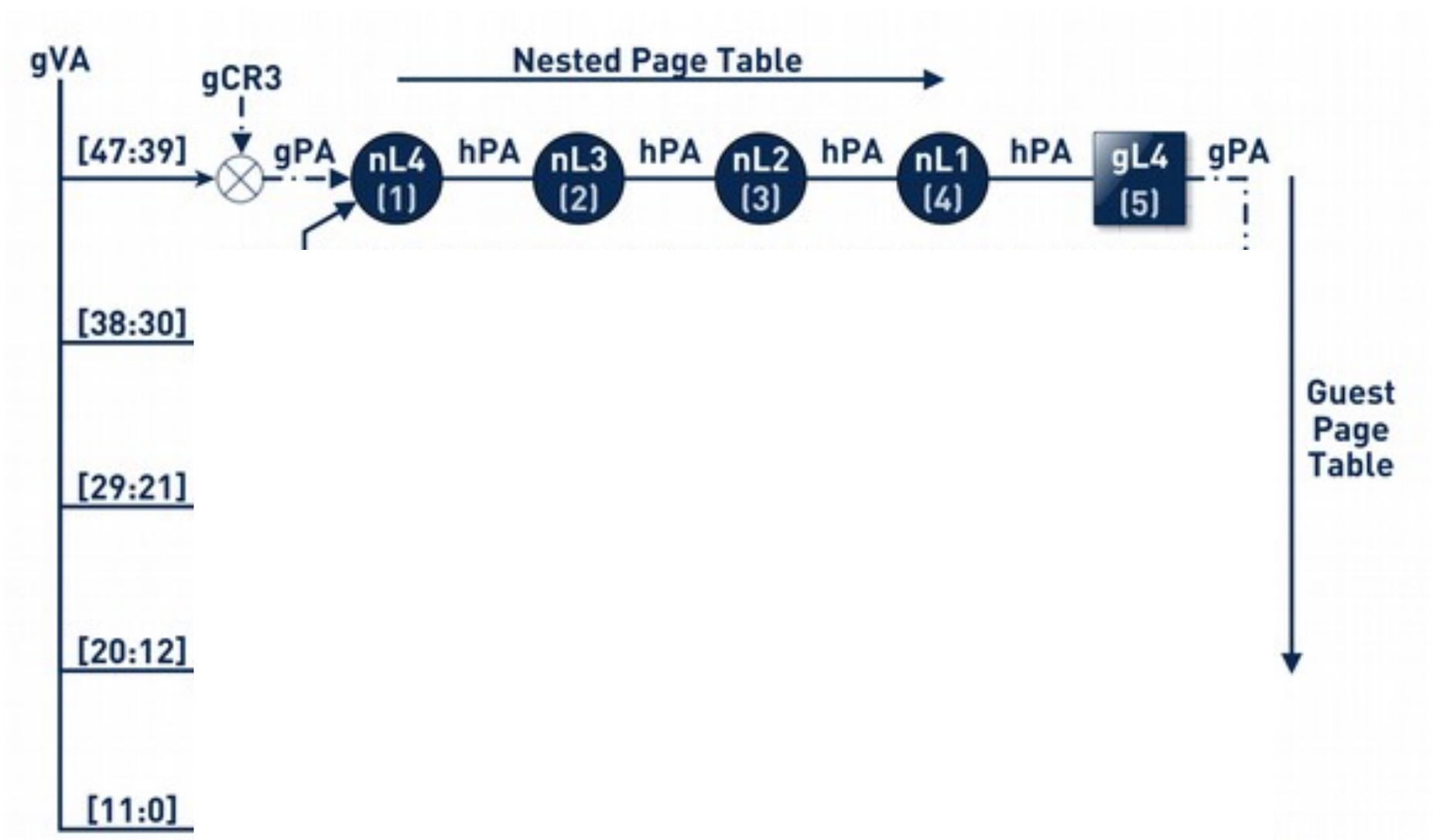
Guest Address Translation



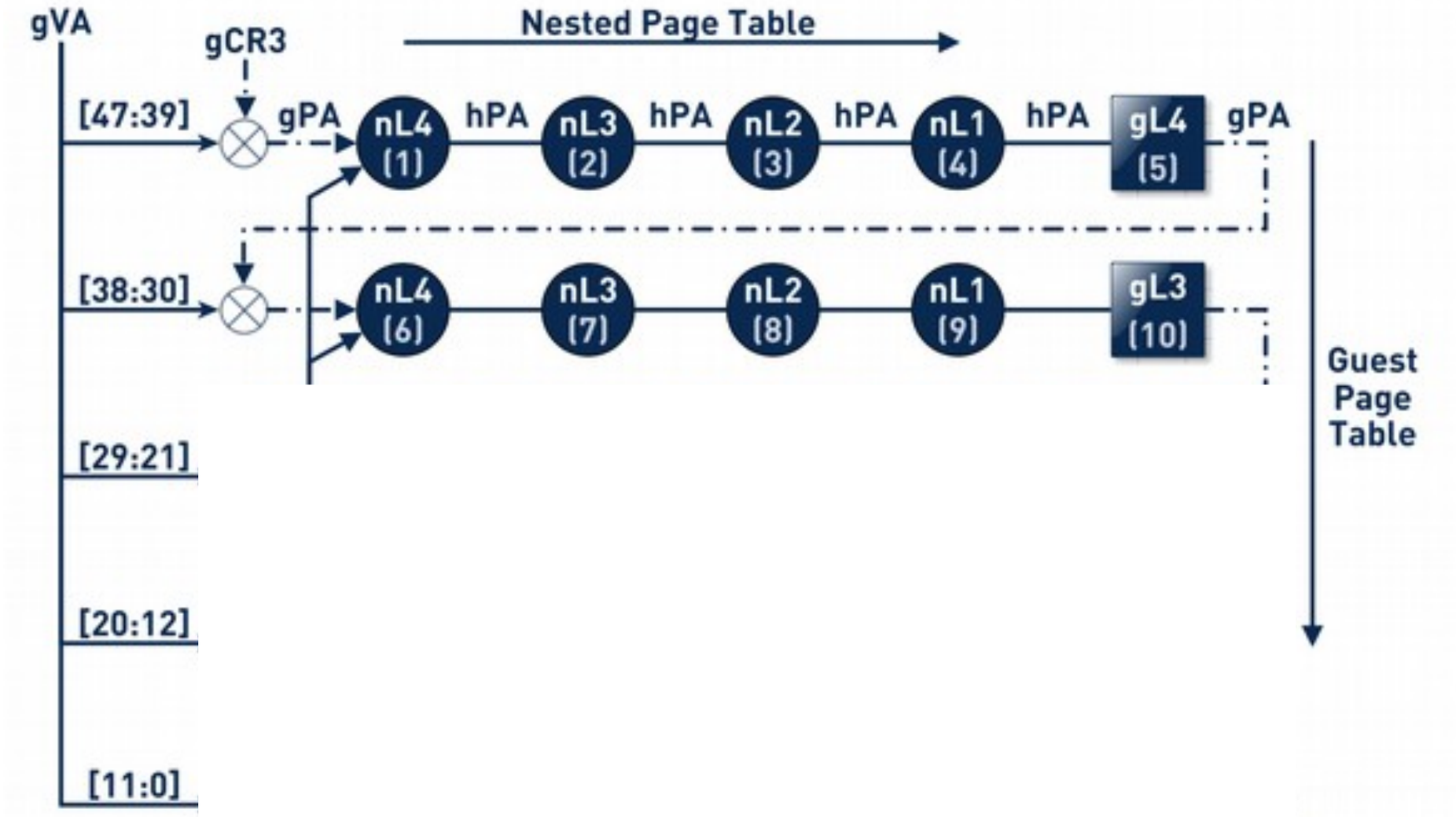
2D Page Table Walk!



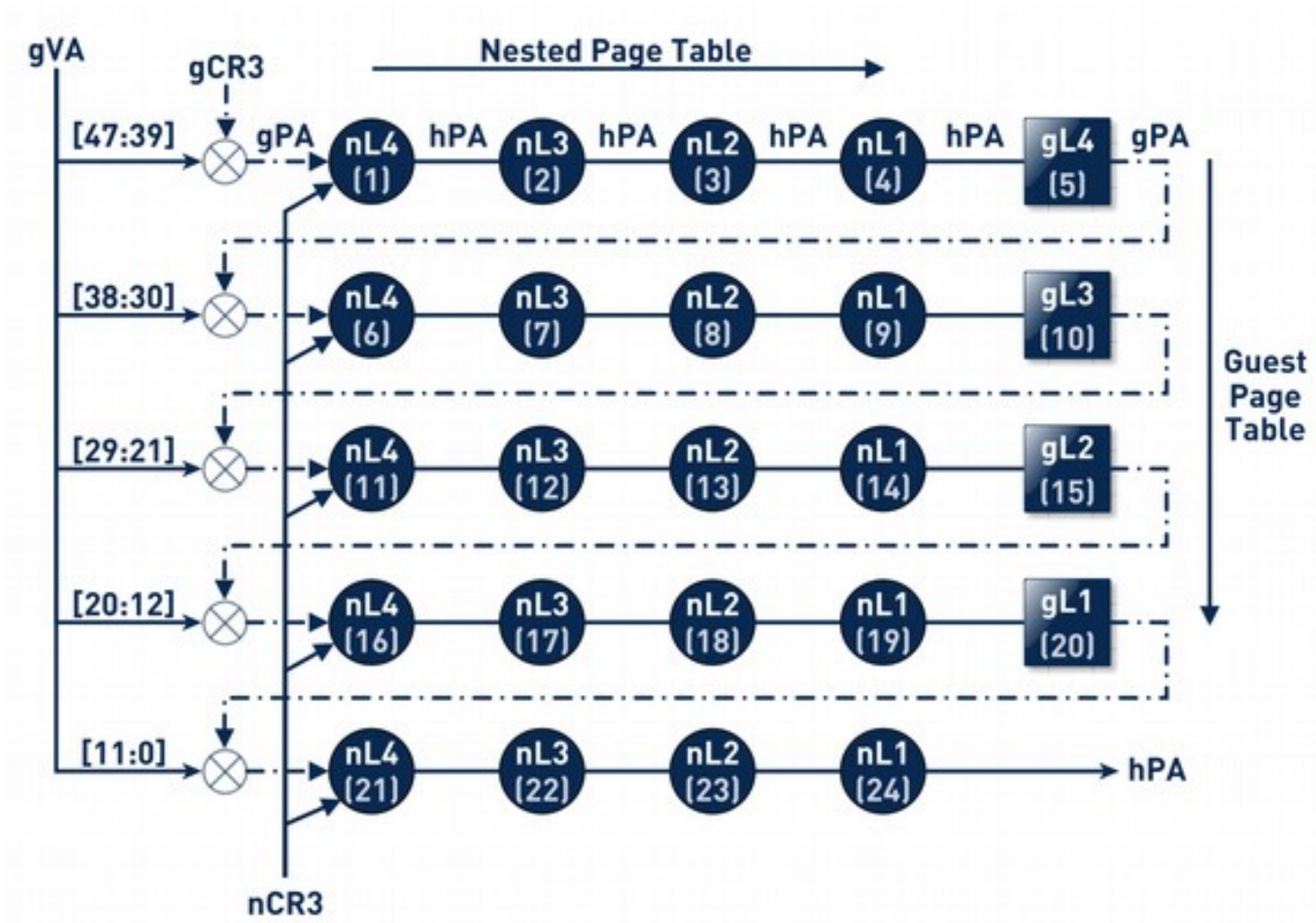
2D Page Table Walk!



2D Page Table Walk!



2D Page Table Walk!



vTLB vs. Nested Paging

Event	Shadow Paging	Nested Paging
vTLB Fill	181,966,391	
Guest Page Fault	13,987,802	
CR Read/Write	3,000,321	
vTLB Flush	2,328,044	
INVLPG	537,270	
Hardware Interrupts	239,142	174,558
Port I/O	723,274	610,589
Memory-Mapped I/O	75,151	76,285
HLT	4,027	3,738
Interrupt Window	3,371	2,171
Sum	202,864,793	867,341
Runtime (seconds)	645	470
Exit/s	314,519	1,845

Steinberg and Kauer 2010

Recap: Virtualization

- Classification
  - Target? Hardware, OS ABI/API, …
  - Modified Guest? Paravirtualization
    - Emulation/Virtualization: Interpret all Instructions?
- Popek & Goldberg: „A VM is an *efficient*, *isolated duplicate* of real machine“
- Hypervisors: Type 1 (bare-metal, kernel) &
  Type 2 (hosted, application on conventional OS)

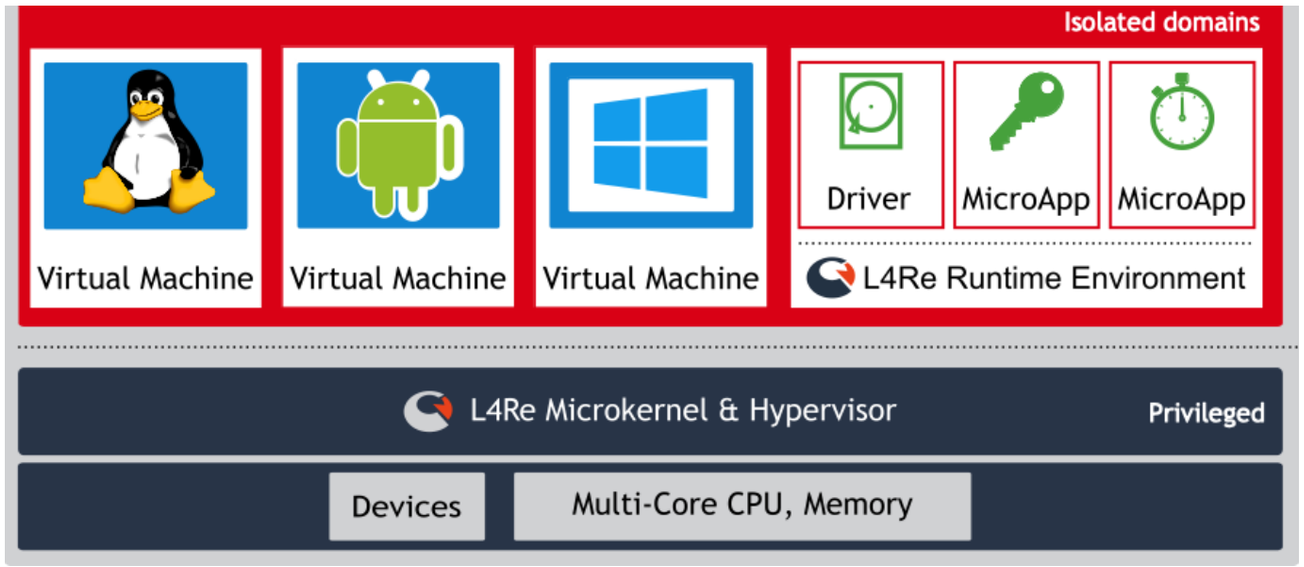
Arm

- Virtualisation Support since Cortex A15 (~2010)
- New processor mode „HYP“ (PL2/EL2) – different to x86
- Nested paging from the start
- No processor-defined state layout (VMCS/VMCB)

  ➡ Hypervisor saves/restores all registers
- Interrupt Controller (GIC) and Generic Timer have built-in virtualisation support

Recap: Microkernels

- Small is beautiful: Small TCB; Security & Safety, Application-specific TCBs
- Real-time, Multi-server, Modular Frameworks, Fault containment
- L4Re: OS Framework
  - L4Re Microkernel
  - L4RE User-level infrastructure
  - … includes virtualisation



Apply Microkernel Principles to Virtualisation

„Hypervisor“ and „VMM“ do not *need* to be synonymous!

Apply Microkernel Principles to Virtualisation

„Hypervisor“ and „VMM“ do not *need* to be synonymous!

Hypervisor:

- Kernel-part
- Provides & ensures isolation
- Mechanism, no policy!

Apply Microkernel Principles to Virtualisation

„Hypervisor“ and „VMM“ do not *need* to be synonymous!

Hypervisor:

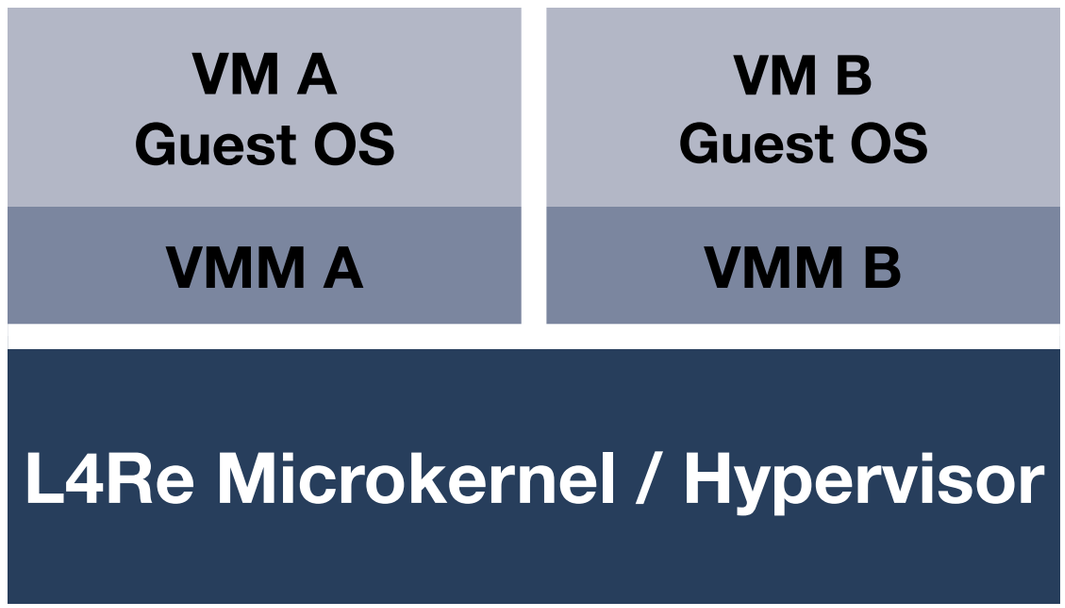
- Kernel-part
- Provides & ensures isolation
- Mechanism, no policy!

Virtual Machine Monitor:

- User space-part
- Platform & device emulation
- Design options!

VMM Design Options

- Typical: One VMM per VM (multi-VM VMMs possible)
- Application-specific: simple vs. feature-rich
- VMM is an **untrusted** user application
- Border between guest and VMM is not the only one

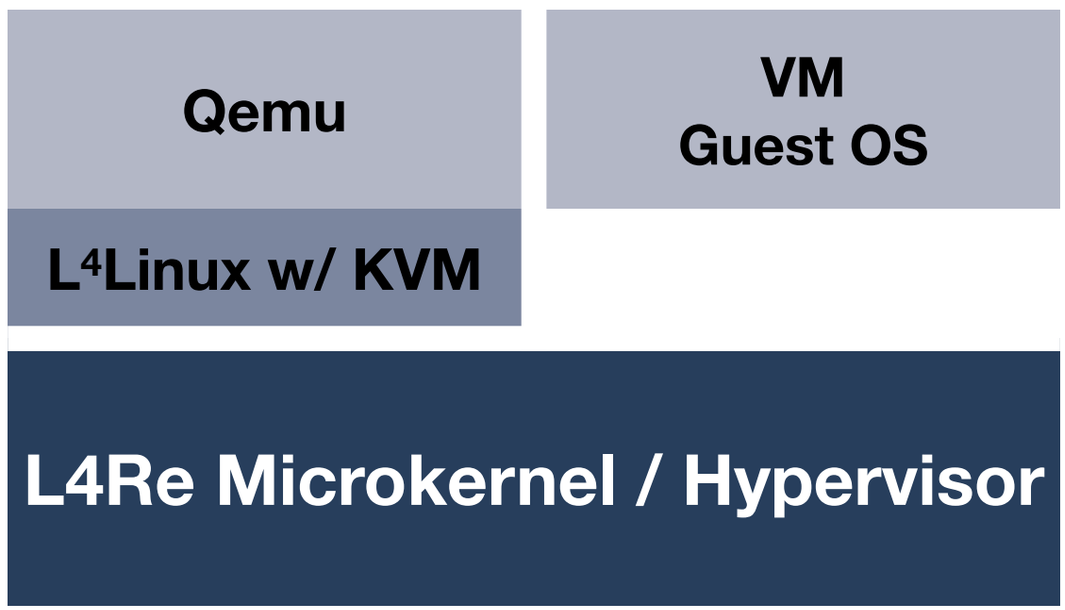


L4Re: uvmm

- VMM for Arm, MIPS, and x86
- Small
- Uses virtio for Guests
- Mainly Linux as Guest OS, but other Guests on request
- Runs (unmodified) Arm Linux

L4Re: KVM/L4

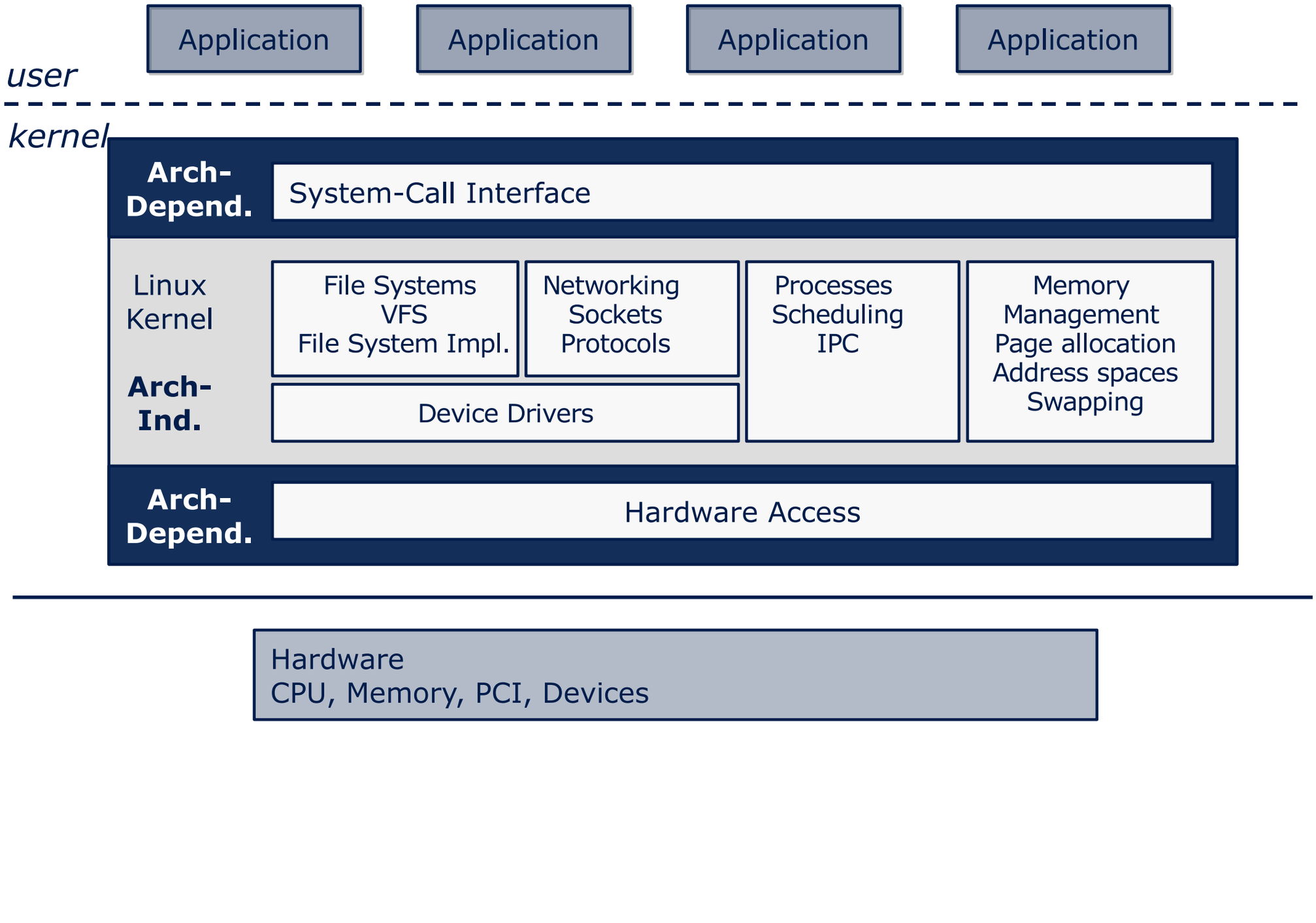
- Complex and feature-rich VMM
- Uses L⁴Linux to run KVM + Qemu
- x86
- Runs Windows
- Used in production



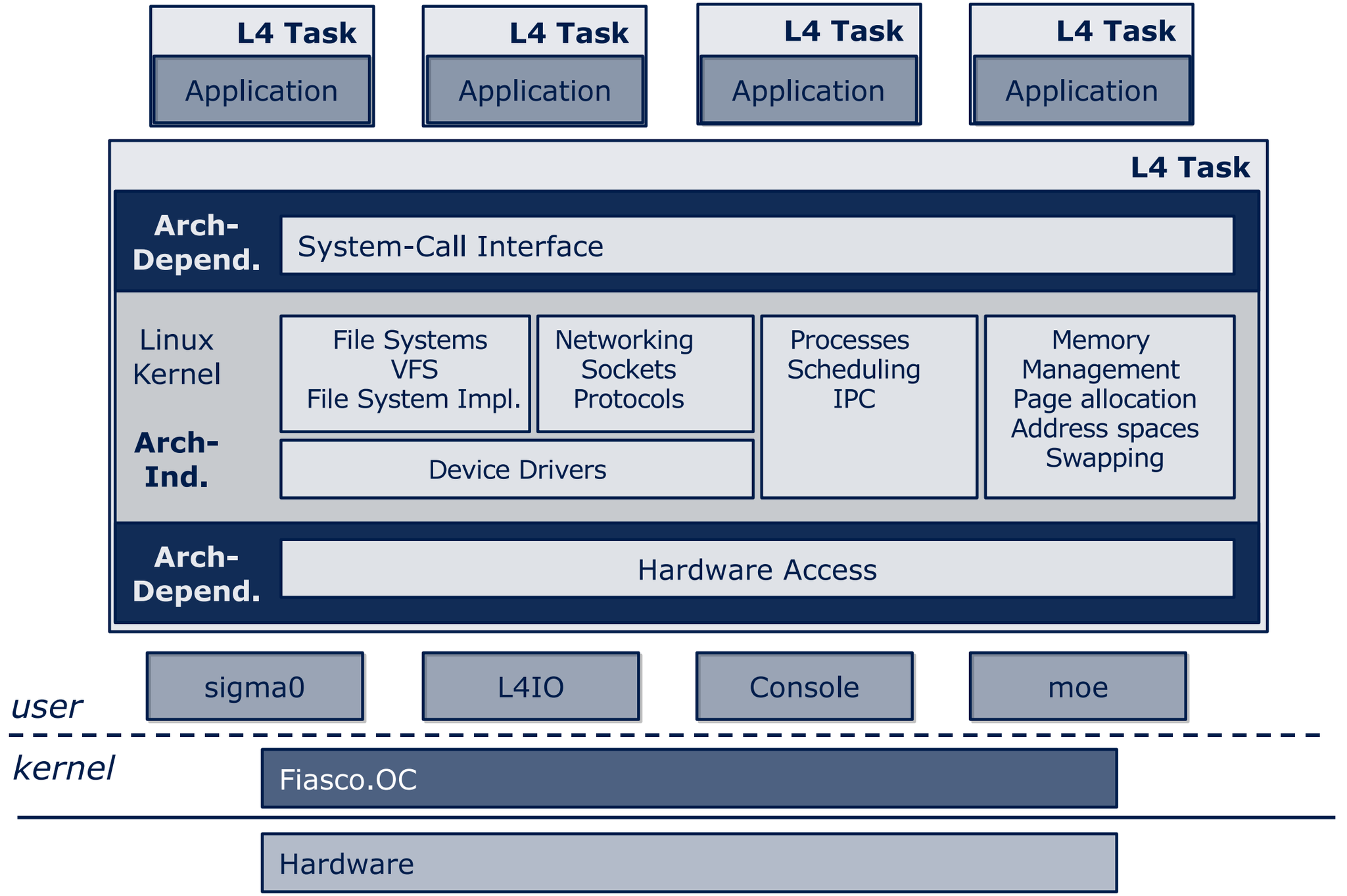
Paravirt Example: L⁴Linux

- Paravirtualized Linux on top of L4Re; presented at SOSP'97
- Regard „L4Re“ as new hardware platform and implement
  - Syscall interface (kernel entry, signal delivery, copy from/to userspace)
  - Hardware Access (CPU state/features, MMU, interrupts, MMIO & port I/O)

Paravirt Example: L⁴Linux



Paravirt Example: L⁴Linux



Software Abstractions

- Interface between kernel/hypervisor and user-level/VMM
- Requirements:
  - Asynchronous execution model of OS kernels
  - Hardware-assisted + paravirtualization
  - Nicely integrate into system

vCPU

- „Legacy“ (synchronous) L4 Thread:
  - Executes ^ Waits for {Events, Messages, IRQs}

  ➡ Hard to map OS kernel onto

- vCPU:
  - Interruptible Thread
  - Similar to how a processor works: Executes **and** get interrupts

L4Re threads can become a vCPU!

vCPU Details

- vCPU is a thread; every thread can become a vCPU
- Interrupt-style execution
  - Events transition the execution to user-defined entry points
  - Virtual interrupt flag (Interrupts disabled == normal thread)
- Virtual User Mode
  - A vCPU can switch to a different L4 task (address space) for execution
  - Returns to „home task“/kernel for any received event
- State save area: Memory area to hold CPU & message state

vCPU

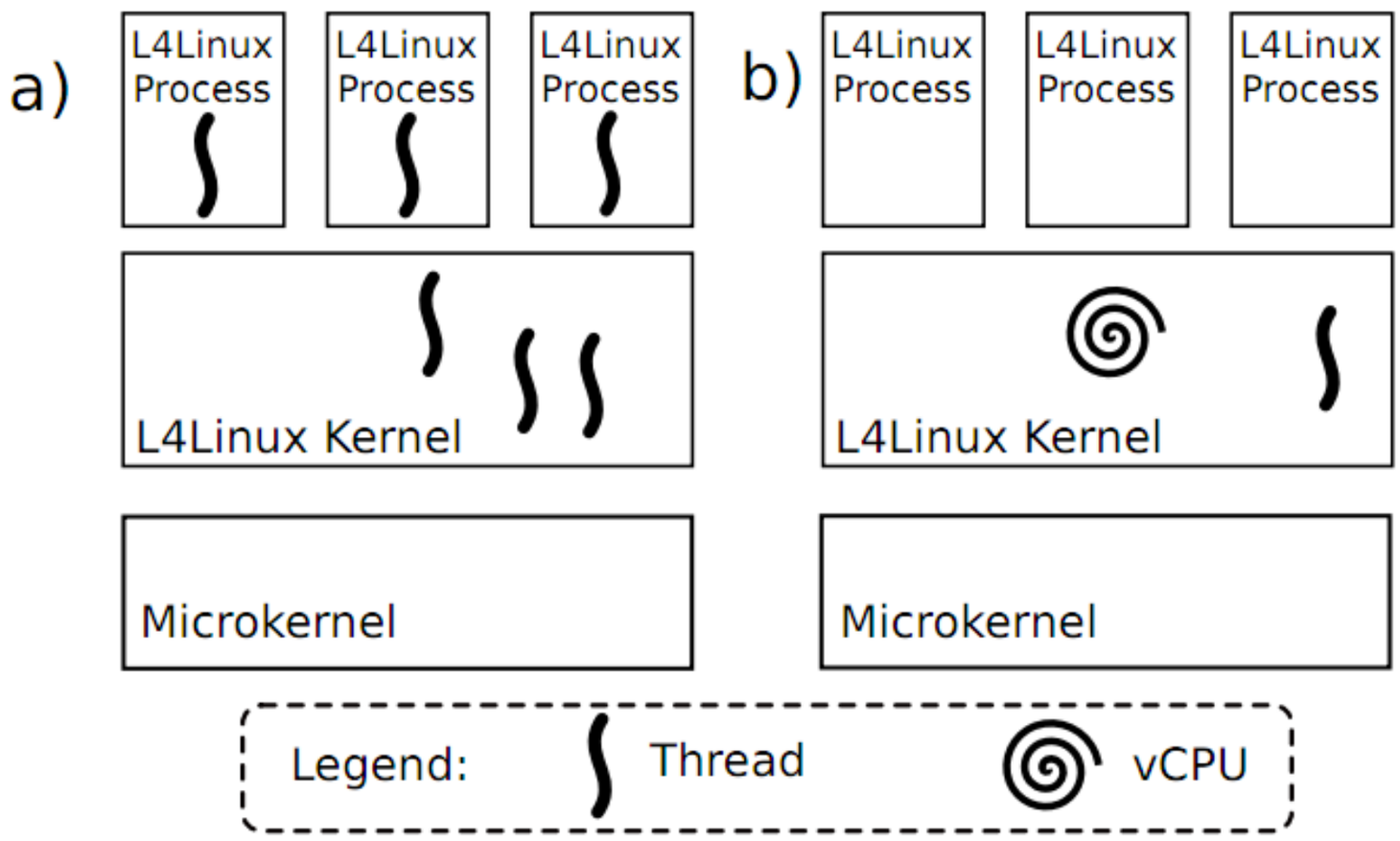


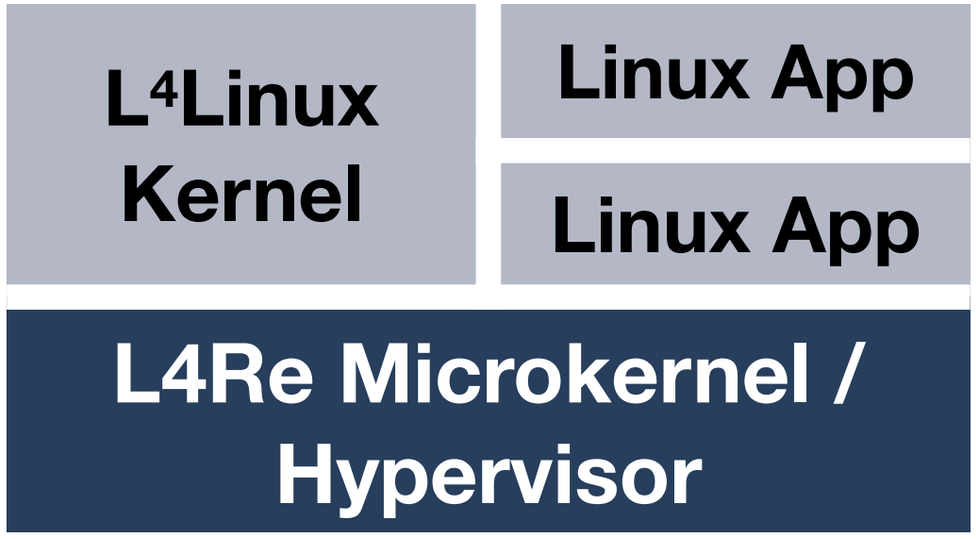
FIGURE 3: *(a) L4Linux implemented with threads and (b) L4Linux implemented with vCPUs.*

Paravirt: Challenges

- Fundamental problem: Mapping 3 logical levels of privilege (Linux App, Linux Kernel, L4Re Microkernel/ Hypervisor) onto 2 levels the platform provides (User/ Kernel mode)

- CPU: Run Linux Kernel + App in microkernel user land

- Memory: Linux kernel manages memory for Linux Apps



L⁴Linux: Performance

- 1997 publication reported <5% overhead
- Events need to be bounced through the micro kernel
  - Native: 2 privilege leves, 0 AS switches
  - L4Linux: 4 privilege levels, 2 AS switches

➡ Hardware-assisted Virtualization

Hardware-assisted Virtualisation

- Intel: VT-x, AMD: SVM, Arm: VE, MIPS: VZ
- Nicely integrates into vCPU abstraction
- Save state area (x86: VMCS/VMCB, Arm/MIPS: hypervisor-mode state & interrupt controller state)
- Memory: nested paging by L4::Task/L4::VM

Device Access

- Options: Exclusive vs Sharing
  - Exclusive: Pass-through
  - Sharing: Microkernel Service / Driver + Guest Interface (VirtIO)
- Pass through resources:
  - MMIO (direct mapping)
  - Interrupts via Microkernel/Hypervisor
  - Direct guest-delivered interrupts on some recent hardware

IOMMU

- Important hardware building block
- MMU for devices
- Indirection & Protection
  - Guest can use gPA (instead of hPA) to program DMA
  - Prevent DMA attacks by evil guests, evil devices, evil firmware, … Limit device accessibility to memory
- Programmed by assigning L4::Task to device

VirtIO

- Common standard for virtual devices
- Defines common data structures
- Wide range of Support (Linux, *BSD, Windows, QNX, …)
- Optimised for virtualisation setups, but can also be used for hardware devices

References

- „Hype and Virtue“, Roscoe, Elphinstone, and Heiser, 2007 http://dl.acm.org/citation.cfm?id=1361397.1361401

- „Formal Requirements for Virtualizable Third Generation Architectures“, Popek and Goldberg, 1974, http://doi.acm.org/10.1145/361011.361073

- „Survey of Virtual Machine Research“, Goldberg, 1974, http://dx.doi.org/10.1109/MC.1974.6323581

- „NOVA: A Microhypervisor-based Secure Virtualization Architecture“, Steinberg and Kauer, 2010, http://www.hypervisor.de/eurosys2010.pdf

- „Virtual Processors as Kernel Interface“, Lackorzynski, Warg, and Peter, 2012, https://www.osadl.org/fileadmin/dam/rtlws/12/Lackorzynski.pdf

References

- „Binary Translation Using Peephole Superoptimizers“, Bansal and Aiken, 2008, http://dl.acm.org/citation.cfm?id=1855741.1855754

- „Virtual machine monitors: current technology and future trends“, Rosenblum and Garfinkel, 2005, http://xenon.stanford.edu/~talg/papers/COMPUTER05/virtual-future-computer05.pdf

- „The Turtles Project: Design and Implementation of Nested Virtualization“, Ben-Yehuda, Day, et al., 2010, https://www.usenix.org/conference/osdi10/turtles-project-design-and-implementation-nested-virtualization

- „The Evolution of an x86 Virtual Machine Monitor“, Agesen et al., 2010, http://doi.acm.org/10.1145/1899928.1899930

References

- „The performance of µ-kernel-based systems“, Härtig et al., 1997, http://dl.acm.org/citation.cfm?id=266660

- „Pre-Virtualization: Slashing the Cost of Virtualization“, LeVasseur et al, 2005, http://www.l4ka.org/downloads/publ_2005_levasseur-ua_cost-of-virtualization.pdf

- „Lightweight Virtualization on Microkernel-based Systems“, Liebergeld, 2010, http://os.inf.tu-dresden.de/papers_ps/liebergeld-diplom.pdf

VMM

- Instruction Emulator
- Timers: PIT, RTC, HPET, PMTimer
- Interrupt Controller: PIC, LAPIC, IOAPIC
- PCI host bridge
- keyboard, mouse, VGA
- Network
- SATA or IDE disk controller
- ...

VMM

VMM needs to emulate (parts of) BIOS/EFI (mostly for boot loaders + early platform discovery):

- Memory layout
- Screen output
- Keyboard
- Disk access
- ACPI tables