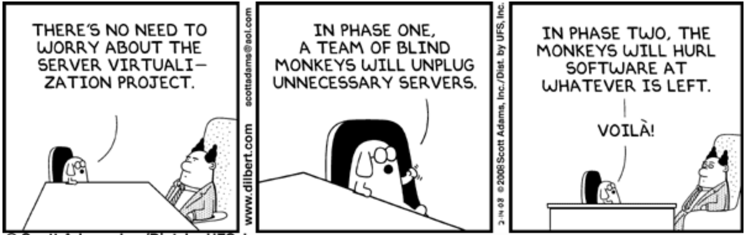

THERE'S NO NEED TO WORRY ABOUT THE SERVER VIRTUALI- ZATION PROJECT.
scottadams@aol.com
www.dilbert.com
IN PHASE ONE, A TEAM OF BLIND MONKEYS WILL UNPLUG UNNECESSARY SERVERS.
© 2008 Scott Adams, Inc./Dist. by UFS, Inc.
IN PHASE TWO, THE MONKEYS WILL HURL SOFTWARE AT WHATEVER IS LEFT.
VOILÀ!

Microkernel-based Operating Systems
— Virtualisation —

Jan Bierbaum

Hannes Weisbach
Julian Stecklina

2024-12-17

Goals

Give you an overview about:

- Virtualisation and virtual machines in general
- Hardware-assisted virtualisation on x86

Goals

Give you an overview about:

- Virtualisation and virtual machines in general
- Hardware-assisted virtualisation on x86

We will *not* discuss:

- Lots and lots of details
- Language runtimes
- How to use Xen/KVM/...

"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."

"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."

—Goldberg: "Survey of Virtual Machine Research", 1974

Early History: IBM

Early History: IBM

Virtualisation was pioneered with IBM's CP/CMS in ~1967 running on System/360 and System/370:

CP Control Program that provided System/360 virtual machines

- Memory protection between VMs
- Preemptive scheduling

CMS Cambridge (later Conversational) Monitor System — single-user OS

Early History: IBM

Virtualisation was pioneered with IBM's CP/CMS in ~1967 running on System/360 and System/370:

CP Control Program that provided System/360 virtual machines
- Memory protection between VMs
- Preemptive scheduling

CMS Cambridge (later Conversational) Monitor System — single-user OS

At the time more flexible and efficient than time-sharing multi-user systems.

Early History: IBM

Virtualisation was pioneered with IBM's CP/CMS in ~1967 running on System/360 and System/370:

CP Control Program that provided System/360 virtual machines
- Memory protection between VMs
- Preemptive scheduling

CMS Cambridge (later Conversational) Monitor System — single-user OS

At the time more flexible and efficient than time-sharing multi-user systems.

Gave rise to IBM's VM line of operating systems:

- First release: 1972
- Latest release: z/VM 7.4 (20 September 2024)

Virtualisation is Great

- Consolidation: improve server utilization
- Isolation: isolate services for security reasons or because of incompatibility
- Reuse: run legacy software
- Development

Virtualisation is Great

- Consolidation: improve server utilization
- Isolation: isolate services for security reasons or because of incompatibility
- Reuse: run legacy software
- Development

. . . but was confined to the mainframe world for a very long time.

Starting Point

You want to write a new operating system that is

- secure
- trustworthy
- small
- fast
- fancy

but . . .

Commodity Applications

Users expect to run all the software they are used to ("legacy support"):

- Browsers
- MS Word
- iTunes
- certified business applications
- new (Windows/DirectX) and ancient (DOS) games

Porting or rewriting all applications is infeasible!

Commodity Hardware

Users expect to run an (x86) OS on any PC which requires support for:

- input devices (USB, PS2, keyboards, mice, tablets, ...)
- graphics adapters (AMD, Intel, nVidia, ...)
- disks (SATA, SAS, USB, Thunderbolt, ...)
- network (Ethernet, Wifi, ...)
- printer, scanner, webcams, ...

Porting or rewriting all drivers is infeasible!

One Solution: Virtualisation

"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, Heiser: "Hype and virtue" (2007)

Virtualisation

"Virtual is most generally used to describe something as being the same as something else in almost every way, except perhaps in name or some other minor, technical sense."

—`https://www.dictionary.com/`

Virtualisation

"Virtual is most generally used to describe something as being the same as something else in almost every way, except perhaps in name or some other minor, technical sense."

—`https://www.dictionary.com/`

"A virtual machine is taken to be an efficient, isolated duplicate of the real machine."

—Popek, Goldberg: "Formal requirements[. . .]", 1974

Virtualisation

"Virtual is most generally used to describe something as being the same as something else in almost every way, except perhaps in name or some other minor, technical sense."

—`https://www.dictionary.com/`

"A virtual machine is taken to be an efficient, isolated duplicate of the real machine."

—Popek, Goldberg: "Formal requirements[...]", 1974

"All problems in computer science can be solved by another level of indirection."

—Butler Lampson, 1972

Virtualisation

“Virtual is most generally used to describe something as being the same as something else in almost every way, except perhaps in name or some other minor, technical sense.”

—`https://www.dictionary.com/`

“A virtual machine is taken to be an efficient, isolated duplicate of the real machine.”

—Popek, Goldberg: “Formal requirements[. . .]”, 1974

“All problems in computer science can be solved by another level of indirection.”

—Butler Lampson, 1972

“...except for the problem of too many layers of indirection.”

—David Wheeler

Emulation

Suppose you develop on your x86 workstation (**H**ost) an operating system that is to run on an ARM-based mobile device (**G**uest).

Emulation

Suppose you develop on your x86 workstation (**H**ost) an operating system that is to run on an ARM-based mobile device (**G**uest).

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

- CPU
- Memory subsystem
- I/O devices

Emulation

Suppose you develop on your x86 workstation (**H**ost) an operating system that is to run on an ARM-based mobile device (**G**uest).

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

- CPU
- Memory subsystem
- I/O devices

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

Emulation

The emulator

- simulates every instruction in software as it is executed,
- prevents G from directly accessing to H's resources,
- maps G's devices onto H's devices, and
- may run multiple times on H.

Mapping G to H

G and H may have considerably different

- instructions sets
- hardware devices

Mapping G to H

G and H may have considerably different

- instructions sets
- hardware devices

making emulation slow and complex (depending on emulation fidelity).

What if G == H?

What if G == H?

If host and emulated hardware architecture is (about) the same,

- interpreting every executed instruction seems unnecessary
- near-native execution speed should be possible.

Virtualisation

What if G == H?

If host and emulated hardware architecture is (about) the same,

- interpreting every executed instruction seems unnecessary
- near-native execution speed should be possible.

This is (easily) possible, if the architecture is *virtualisable*.

Idea: Executing the guest as a user process

Run the guest operating system as a normal user process on the host.

[1] Often used synonymously with “hypervisor”. We’ll come back to tht later.

Idea: Executing the guest as a user process

Run the guest operating system as a normal user process on the host.

But this is not just about executing instructions!

[1] Often used synonymously with “hypervisor”. We’ll come back to tht later.

Idea: Executing the guest as a user process

Run the guest operating system as a normal user process on the host.

But this is not just about executing instructions!

We need to emulate virtual hardware. The software providing the illusion of a real machine is the Virtual Machine Monitor (VMM)[1].

[1] Often used synonymously with "hypervisor". We'll come back to tht later.

A hypothetical instruction: `OUT`

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

A hypothetical instruction: `OUT`

Suppose our system has an instruction `OUT` that writes to a device register in kernel mode. But we run it (virtualised) in user mode. How should `OUT` behave?

A hypothetical instruction: `OUT`

Suppose our system has an instruction `OUT` that writes to a device register in kernel mode. But we run it (virtualised) in user mode. How should `OUT` behave?

Option 1

Just do nothing!

Option 2

Cause a trap to kernel mode!

A hypothetical instruction: `OUT`

Suppose our system has an instruction `OUT` that writes to a device register in kernel mode. But we run it (virtualised) in user mode. How should `OUT` behave?

Option 1

~~Just do nothing!~~

Option 2

Cause a trap to kernel mode!

Otherwise device access cannot be (easily) virtualized.

Virtualizability

. . . is a property of the *Instruction Set Architecture* (ISA).

Virtualizability

. . . is a property of the *Instruction Set Architecture* (ISA).
Instructions are divided into two classes:

Sensitive

A sensitive instruction changes or depends in its behavior on the processor's configuration or mode.

Privileged

A privileged instruction causes a trap when executed in user mode.

Trap & Emulate

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

Trap & Emulate

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

Approach:

- Execute guest in unprivileged mode
- Emulate all instructions that cause traps

Popek, Goldberg: "Formal Requirements for Virtualizable Third-Generation Architectures", 1973

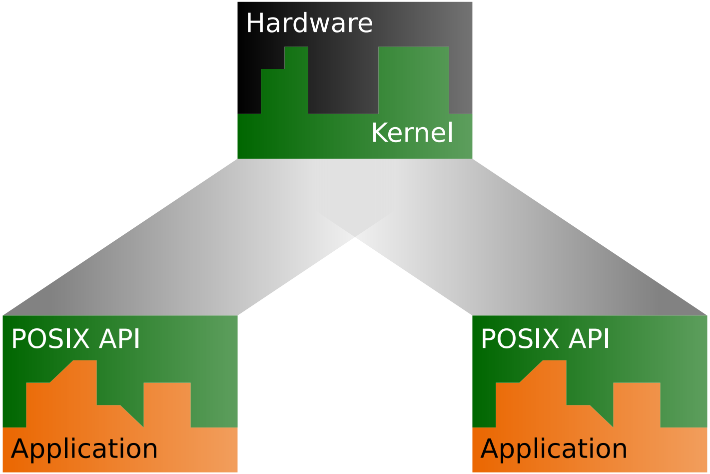
VMM

The Virtual Machine Monitor (VMM) needs to handle:

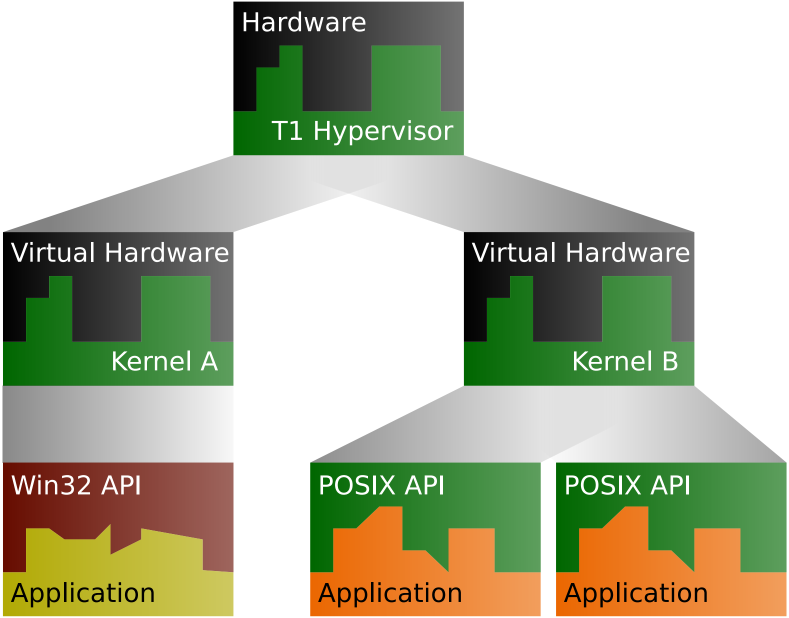
- address space changes
- device accesses
- system calls
- ...

Most of these are not problematic, because they trap to the host kernel.

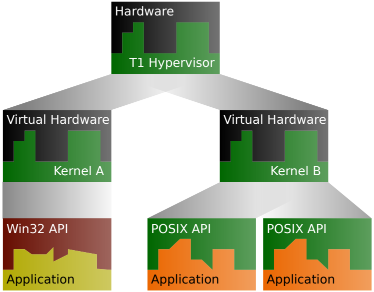
Where to put the VMM?



Type-1/Bare-Metal/Native Hypervisor



Type-1/Bare-Metal/Native Hypervisor

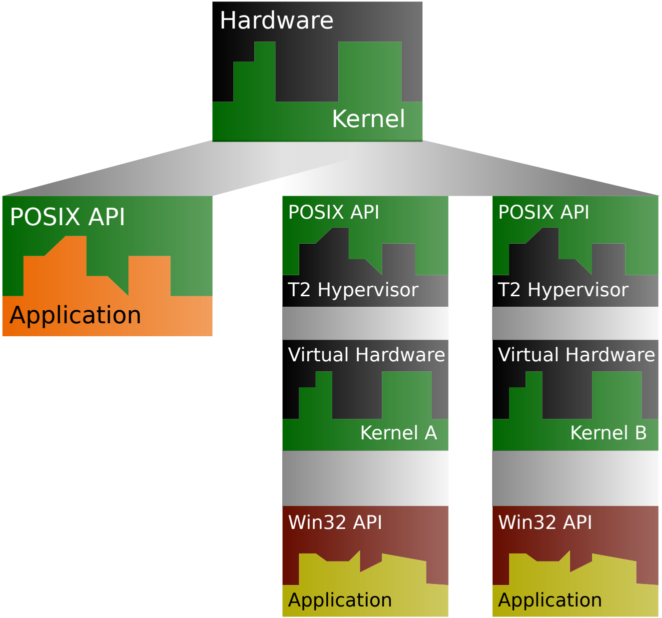


- Implemented directly on hardware
- No OS overhead
- Complete control over host resources
- High maintainance effort

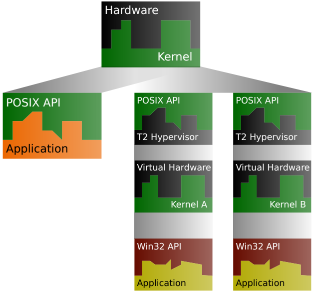
Popular examples:

- Xen
- Hyper-V
- VMware ESXi

Type-2/Hosted Hypervisor



Type-2/Hosted Hypervisor



- Implemented as normal process on top of an OS
- Doesn't reinvent the wheel
- Performance may suffer
- Requires Host-OS support for CPU's virtualisation features

Popular examples:

- KVM,
- VMware Server/Workstation,
- VirtualBox,
- ...

Paravirtualisation

Why all the trouble?
Just “port” a guest operating system to the interface of your choice.

Paravirtualisation

Why all the trouble?
Just "port" a guest operating system to the interface of your choice.

+ Better performance
+ Simplified VMM
- Maintainance cost
- Source code of guest OS required (& modification allowed)

Paravirtualisation

Why all the trouble?
Just "port" a guest operating system to the interface of your choice.

+ Better performance
+ Simplified VMM
- Maintainance cost
- Source code of guest OS required (& modification allowed)

Compromise: Paravirtualized drivers for I/O performance (KVM virtio, VMware)

Examples: Usermode Linux, L^4Linux, Xen/XenoLinux, DragonFlyBSD VKERNEL, ...

Virtualised ABI

Why deal with the OS kernel at all? Just reimplement its interface!

Example: Wine

- Reimplements (virtualizes) Windows ABI
- Run unmodified Windows binaries
- Windows API calls are mapped to host OS's (Linux/MacOS/*BSD/...) equivalents
- Huge moving target/maintainance effort

Virtualised ABI

Why deal with the OS kernel at all? Just reimplement its interface!

Example: Wine

- Reimplements (virtualizes) Windows ABI
- Run unmodified Windows binaries
- Windows API calls are mapped to host OS's (Linux/MacOS/*BSD/...) equivalents
- Huge moving target/maintainance effort

Also: API "virtualisation": Recompile Windows applications as native applications linking to `winelib`

Recap — Virtualisation

- Classification criteria:
  - Target? Hardware, OS ABI, OS API, . . .
  - Modified guest? Paravirtualisation
  - Emulation vs. Virtualisation (Interpret all or only some instructions?)

Recap — Virtualisation

- Classification criteria:
  - Target? Hardware, OS ABI, OS API, ...
  - Modified guest? Paravirtualisation
  - Emulation vs. Virtualisation (Interpret all or only some instructions?)
- Popek, Goldberg: "*A virtual machine is an efficient, isolated duplicate of a real machine.*" implemented by a Virtual Machine Monitor (hypervisor).
  - Type 1/bare-metal hypervisors run as kernel
  - Type 2/hosted hypervisors run as applications on a conventional OS

Is x86 Virtualizable?

- x86 originally not virtualizable (`push`, `pushf`/`popf`, ... 17 instructions on the Pentium)
- Trapping on every privileged instruction too expensive

Is x86 Virtualizable?

- x86 originally not virtualizable (`push`, `pushf`/`popf`, ... 17 instructions on the Pentium)
- Trapping on every privileged instruction too expensive
- First commercial virtualisation solution for x86: VMware Workstation (~1999)
  - Translate problematic instructions into appropriate calls to the VMM on the fly (binary rewriting)

Is x86 Virtualizable?

- x86 originally not virtualizable (`push`, `pushf`/`popf`, ... 17 instructions on the Pentium)
- Trapping on every privileged instruction too expensive
- First commercial virtualisation solution for x86: VMware Workstation (~1999)
  - Translate problematic instructions into appropriate calls to the VMM on the fly (binary rewriting)
  - Avoid costly traps for privileged instructions

Is x86 Virtualizable?

- x86 originally not virtualizable (`push`, `pushf`/`popf`, ... 17 instructions on the Pentium)
- Trapping on every privileged instruction too expensive
- First commercial virtualisation solution for x86: VMware Workstation (~1999)
  - Translate problematic instructions into appropriate calls to the VMM on the fly (binary rewriting)
  - Avoid costly traps for privileged instructions
  - Decent performance but complex runtime translation engine; only common guests (commercially) supported

Is x86 Virtualizable?

- x86 originally not virtualizable (`push`, `pushf`/`popf`, ... 17 instructions on the Pentium)
- Trapping on every privileged instruction too expensive
- First commercial virtualisation solution for x86: VMware Workstation (~1999)
  - Translate problematic instructions into appropriate calls to the VMM on the fly (binary rewriting)
  - Avoid costly traps for privileged instructions
  - Decent performance but complex runtime translation engine; only common guests (commercially) supported

Is x86 Virtualizable?

- x86 originally not virtualizable (`push`, `pushf`/`popf`, ... 17 instructions on the Pentium)
- Trapping on every privileged instruction too expensive
- First commercial virtualisation solution for x86: VMware Workstation (~1999)
  - Translate problematic instructions into appropriate calls to the VMM on the fly (binary rewriting)
  - Avoid costly traps for privileged instructions
  - Decent performance but complex runtime translation engine; only common guests (commercially) supported
- Other examples: KQemu, Virtual Box, Valgrind

Special Hardware Support

- "Hardware-assisted virtualisation"
- CPU
  - Virtual CPU mode, including kernel mode
  - All guest instructions are virtualisable
- Memory

Special Hardware Support

- "Hardware-assisted virtualisation"
- CPU
  - Virtual CPU mode, including kernel mode
  - All guest instructions are virtualisable
- Memory
- Typically, VMs have very few (if any) VM-exits for CPU/memory virtualisation

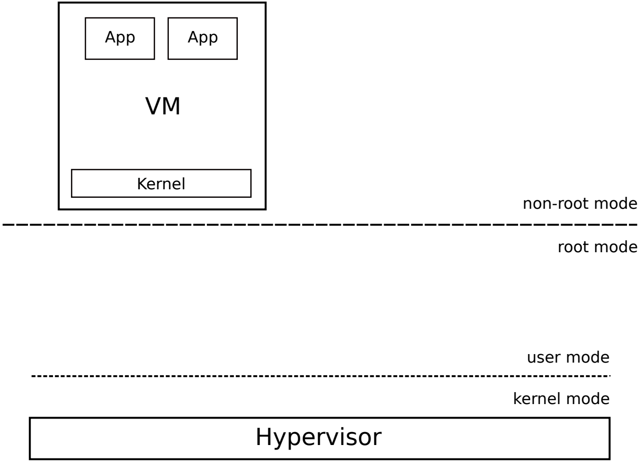
Special Hardware Support

Pentium 4 introduced hardware support for virtualisation in 2004:
Intel VT (AMD-V very similar)

Special Hardware Support

Pentium 4 introduced hardware support for virtualisation in 2004:
Intel VT (AMD-V very similar)

- Root mode vs. non-root mode
  - Duplicate x86 protection rings
  - Root mode runs hypervisor
  - Non-root mode runs guest



Special Hardware Support

Pentium 4 introduced hardware support for virtualisation in 2004:
Intel VT (AMD-V very similar)

- Root mode vs. non-root mode
  - Duplicate x86 protection rings
  - Root mode runs hypervisor
  - Non-root mode runs guest
- Situations that Intel VT cannot handle trap to root mode (VM Exit)

Special Hardware Support

Pentium 4 introduced hardware support for virtualisation in 2004:
Intel VT (AMD-V very similar)

- Root mode vs. non-root mode
  - Duplicate x86 protection rings
  - Root mode runs hypervisor
  - Non-root mode runs guest
- Situations that Intel VT cannot handle trap to root mode (VM Exit)
- Special memory region (VMCS/VMCB) holds guest state

Special Hardware Support

Pentium 4 introduced hardware support for virtualisation in 2004:
Intel VT (AMD-V very similar)

- Root mode vs. non-root mode
  - Duplicate x86 protection rings
  - Root mode runs hypervisor
  - Non-root mode runs guest
- Situations that Intel VT cannot handle trap to root mode (VM Exit)
- Special memory region (VMCS/VMCB) holds guest state
- Reduced software complexity

Special Hardware Support

Pentium 4 introduced hardware support for virtualisation in 2004:
Intel VT (AMD-V very similar)

- Root mode vs. non-root mode
  - Duplicate x86 protection rings
  - Root mode runs hypervisor
  - Non-root mode runs guest
- Situations that Intel VT cannot handle trap to root mode (VM Exit)
- Special memory region (VMCS/VMCB) holds guest state
- Reduced software complexity

Supported by all major virtualisation solutions today.

Instruction Emulation

Intel VT and AMD-V still require an instruction emulator, e.g. for

- Running 16-bit code (not in AMD-V, current Intel VT)
  - BIOS
  - Boot loaders
- Handling memory-mapped I/O
  - Realized as non-present page
  - Page fault on access
  - Emulate offending instruction
- ...

MMU Virtualisation

Early versions of Intel VT do not completely virtualize the MMU. The VMM has to handle guest virtual memory.

MMU Virtualisation

Early versions of Intel VT do not completely virtualize the MMU. The VMM has to handle guest virtual memory.

Four different types of memory addresses:

- **hPA** Host physical address
- **hVA** Host virtual address
- **gPA** Guest physical address
- **gVA** Guest virtual address

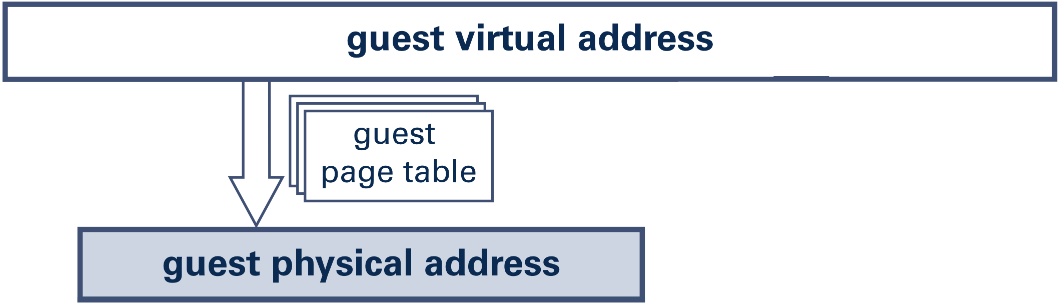
MMU Virtualisation

Early versions of Intel VT do not completely virtualize the MMU. The VMM has to handle guest virtual memory.

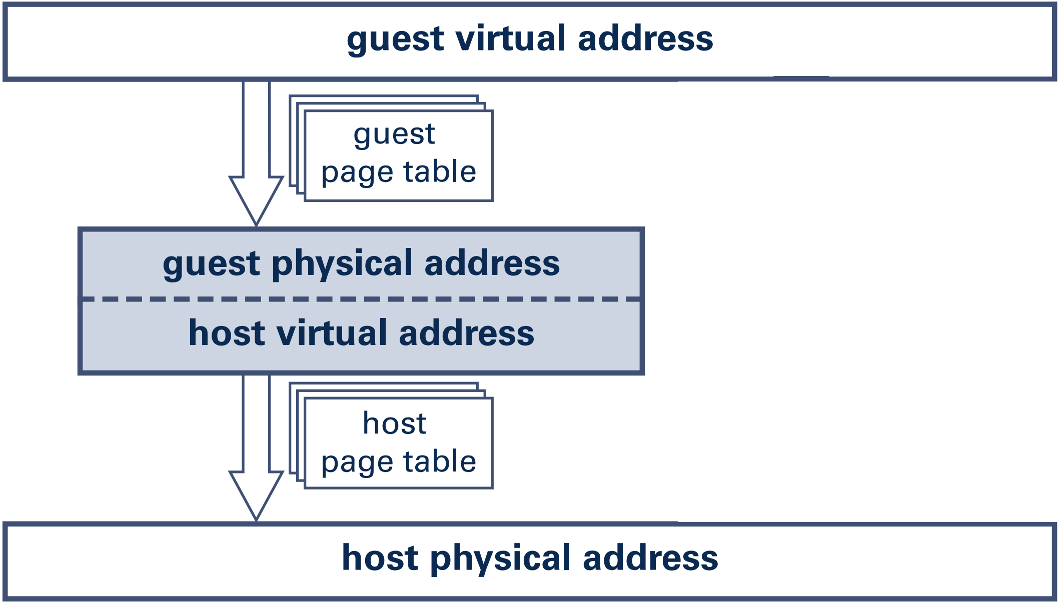
Four different types of memory addresses:

- **hPA** Host physical address
- **hVA** Host virtual address
- **gPA** Guest physical address
- **gVA** Guest virtual address

Usually GPA == HVA or other simple mapping (e.g. constant offset).


guest virtual address
guest
page table
guest physical address

MMU Virtualisation

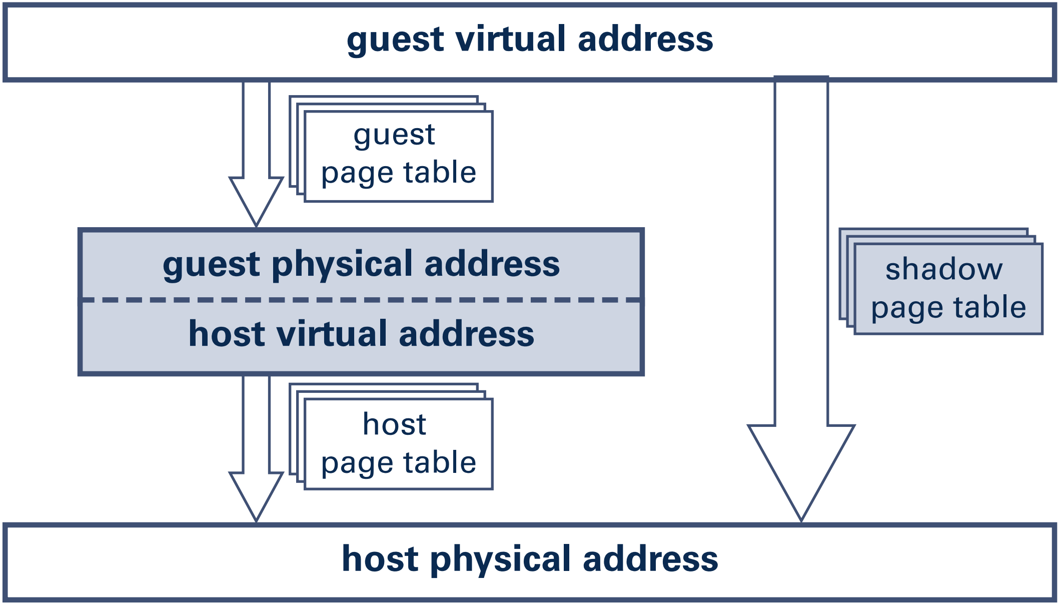


Shadow Page Tables

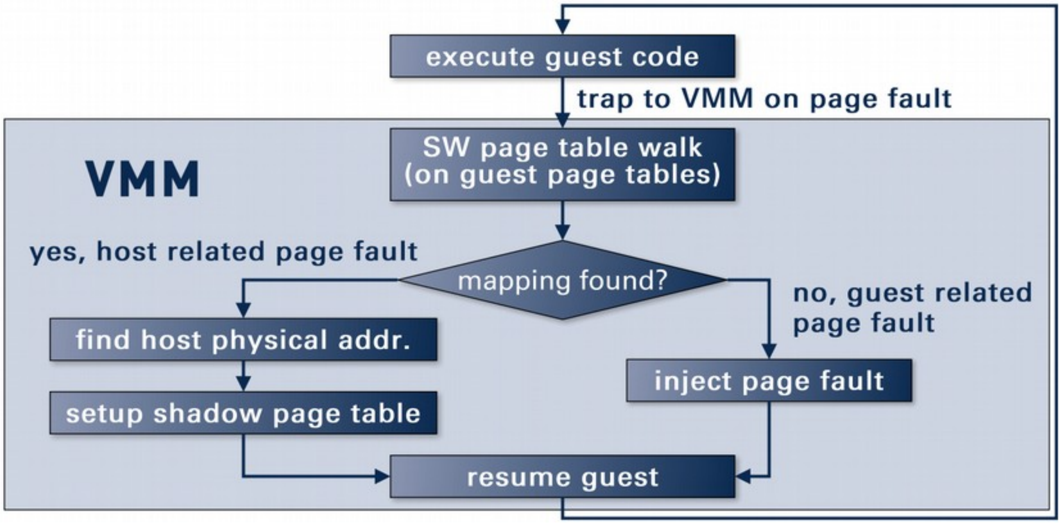
If the hardware can handle only one page table, the hypervisor must maintain a shadow page table that

- maps from GVA to HPA ("merging" guest and host page table),
- must be adapted on changes to virtual memory layout.

MMU Virtualisation



Shadow Paging in a Nutshell



Drawbacks of Shadow Paging

Maintaining shadow page tables causes significant overhead, because they need to be updated or recreated on

- guest page table modification,
- guest address space switch.

Certain workloads are penalized.

Second Level Address Translation (SLAT)

Intel *Nehalem* (Extended Page Table, EPT) and AMD *Barcelona* (Nested Paging) microarchitectures introduced hardware support for MMU virtualisation.

Second Level Address Translation (SLAT)

Intel *Nehalem* (Extended Page Table, EPT) and AMD *Barcelona* (Nested Paging) microarchitectures introduced hardware support for MMU virtualisation.

The CPU can handle the guest and host page table at the same time and thus reduce VM Exits by two orders of magnitude.

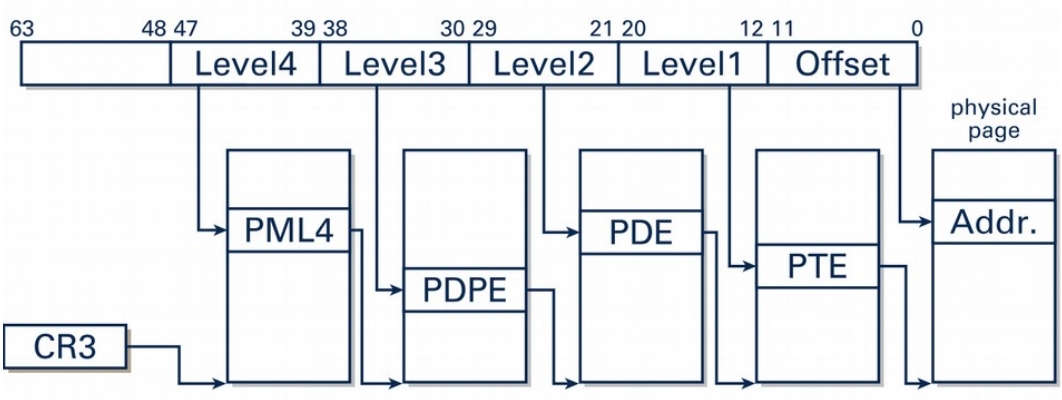
Second Level Address Translation (SLAT)

Intel *Nehalem* (Extended Page Table, EPT) and AMD *Barcelona* (Nested Paging) microarchitectures introduced hardware support for MMU virtualisation.

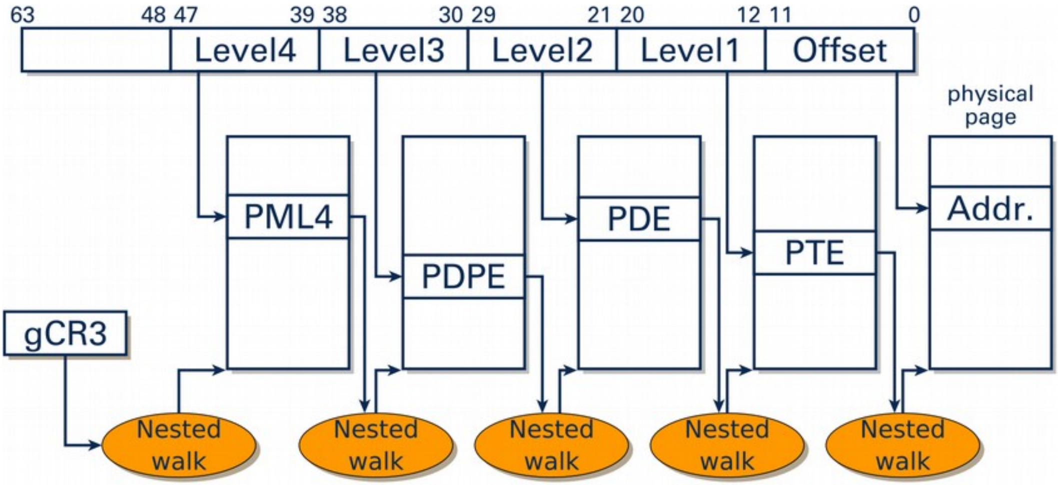
The CPU can handle the guest and host page table at the same time and thus reduce VM Exits by two orders of magnitude.

This feature introduces a measurable constant overhead ($< 1\%$).

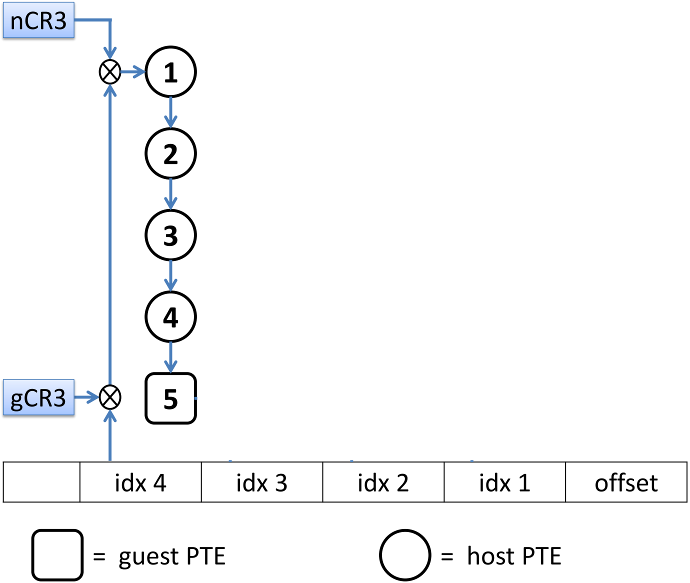
Guest Address Translation



Guest Address Translation

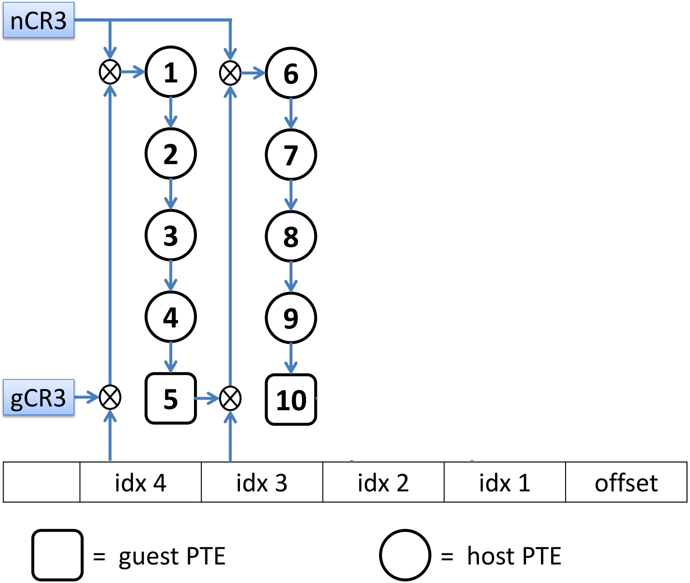


2D Page Table Walk



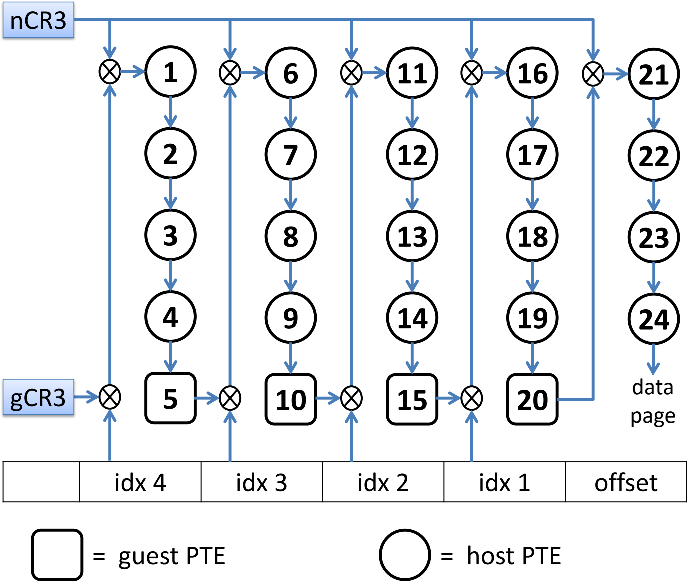
"Hash, Don't Cache (the Page Table)", 2016
— Yaniv, Tsafrir

2D Page Table Walk



— Yaniv, Tsafrir
"Hash, Don't Cache (the Page Table)", 2016

2D Page Table Walk



— Yaniv, Tsafrir
"Hash, Don't Cache (the Page Table)", 2016

Shadow Paging vs. SLAT

Event	Shadow Paging	EPT
vTLB Fill	181,966,391	
Guest Page Fault	13,987,802	
CR Read/Write	3,000,321	
vTLB Flush	2,328,044	
Port I/O	723,274	610,589
INVLPG	537,270	
Hardware Interrupts	239,142	174,558
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 (sec)	645	470
Exit/sec	314,519	1,845

—Linux kernel compilation, from Steinberg, Kauer
"NOVA: A Microhypervisor-Based Secure Virtualization Architecture", 2010

Arm

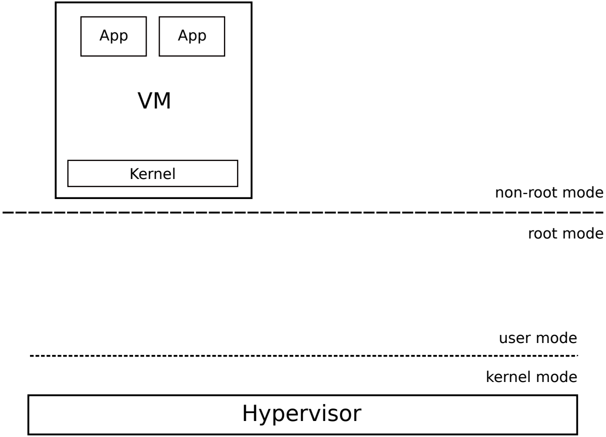
- Virtualisation Support since Cortex A15 (2010)
- New processor mode “HYP” (PL2/EL2) — different from x86
- Nested paging from the start
- No processor-defined state layout (VMCS/VMCB) ⇒ Hypervisor saves/restores all registers
- Interrupt controller (GIC) and generic timer have built-in virtualisation support
- Hardware support for nested virtualisation

TCB of Virtual Machines

The *Trusted Computing Base* of a Virtual Machine is the hardware and software components you have to trust to guarantee this VM's security.

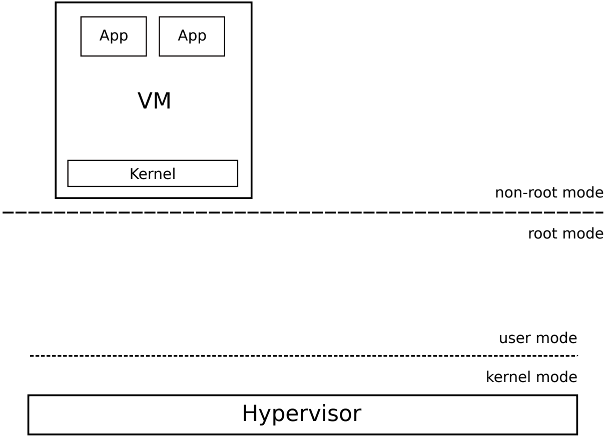
TCB of Virtual Machines

The *Trusted Computing Base* of a Virtual Machine is the hardware and software components you have to trust to guarantee this VM's security.



TCB of Virtual Machines

The *Trusted Computing Base* of a Virtual Machine is the hardware and software components you have to trust to guarantee this VM's security.



For e.g. KVM this (conservatively) includes the Linux kernel and Qemu.

Recap: Microkernels & L4Re

- Small is beautiful: small TCB; security & safety, application-specific TCBs

Recap: Microkernels & L4Re

- Small is beautiful: small TCB; security & safety, application-specific TCBs
- Real-time, multi-server, modular frameworks, fault containment, ...

Recap: Microkernels & L4Re

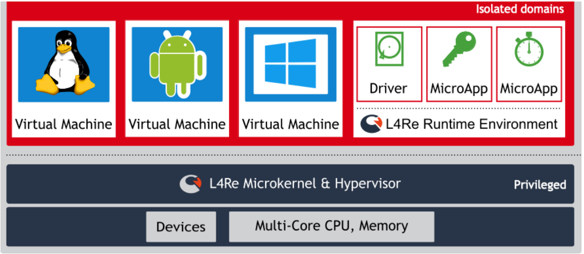
- Small is beautiful: small TCB; security & safety, application-specific TCBs
- Real-time, multi-server, modular frameworks, fault containment, ...

Recap: Microkernels & L4Re

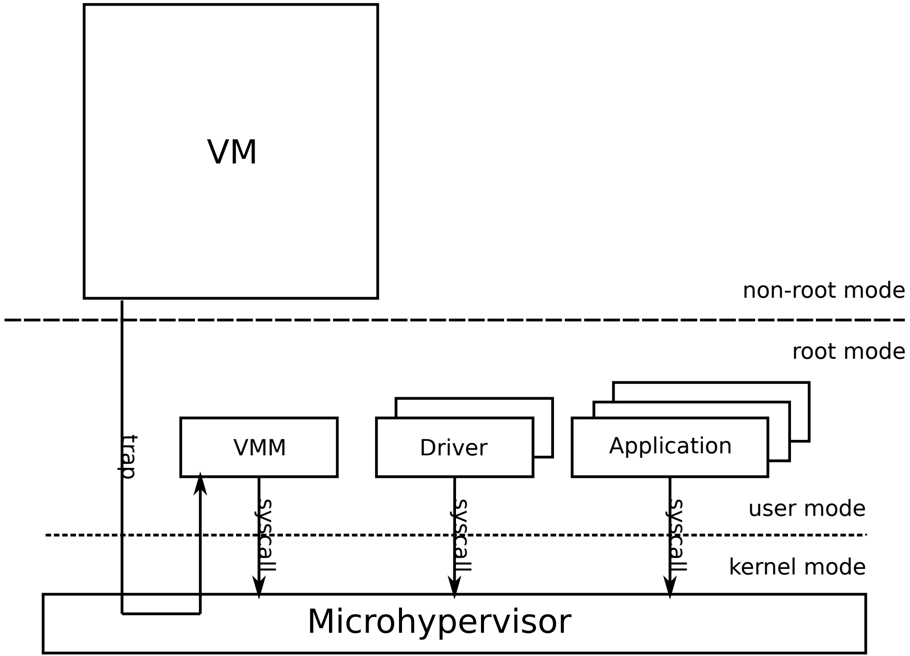
- Small is beautiful: small TCB; security & safety, application-specific TCBs
- Real-time, multi-server, modular frameworks, fault containment, ...

L4Re: OS Framework

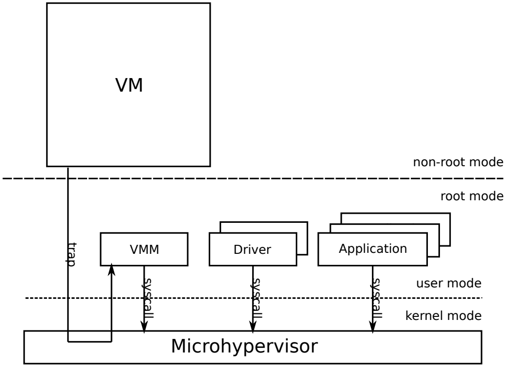
- Fiasco/L4Re Microkernel
- L4Re user-level infrastructure
- ...supports virtualisation



Shrinking the Hypervisor



Shrinking the Hypervisor



What needs to be in the Microhypervisor? Ideally nothing, but

- VT-x instructions are privileged
- Hypervisor has to validate guest state to enforce isolation

Microhypervisor vs. VMM

"Hypervisor" and "VMM" do not need to be synonymous...

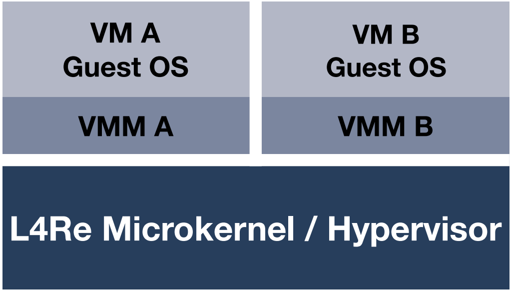
Microhypervisor

- "Kernel part"
- Provides & ensures isolation
- Enables safe access to virtualisation features to userspace
- Mechanisms, no policies!

VMM

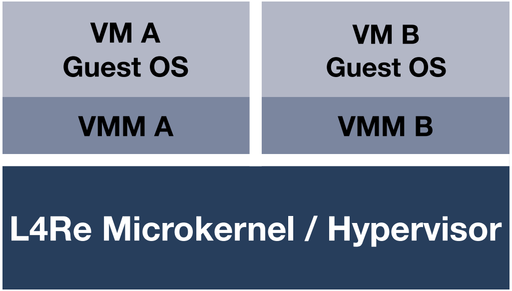
- "User-space part"
- Platform & device emulation
- Design options!

VMM Design Options



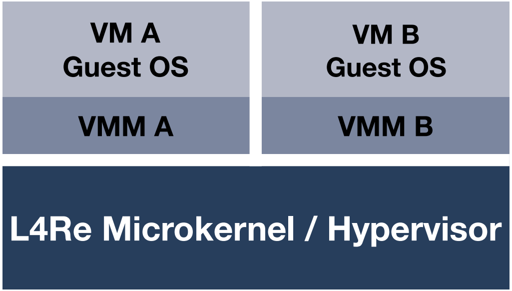
- Typical: One VMM per VM (multi-VM VMMs possible)

VMM Design Options



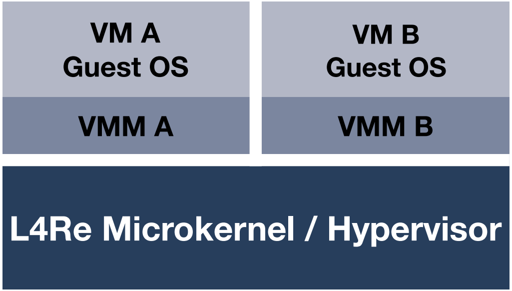
- Typical: One VMM per VM (multi-VM VMMs possible)
- Application-specific: simple vs. feature-rich

VMM Design Options



- Typical: One VMM per VM (multi-VM VMMs possible)
- Application-specific: simple vs. feature-rich
- VMM is an untrusted user application

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, RISC V, and x86

L4Re: uvmm

- VMM for Arm, MIPS, RISC V, and x86
- Small

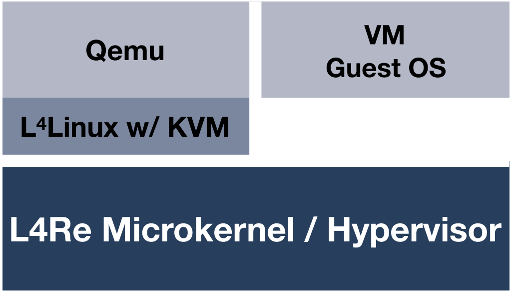
L4Re: uvmm

- VMM for Arm, MIPS, RISC V, and x86
- Small
- Uses virtio for guests

L4Re: uvmm

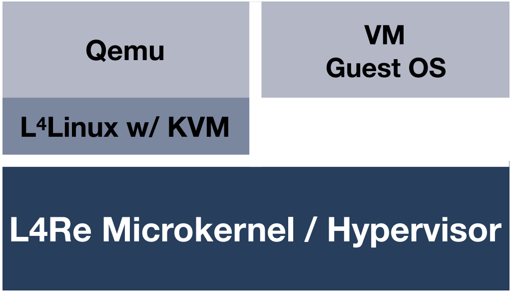
- VMM for Arm, MIPS, RISC V, and x86
- Small
- Uses virtio for guests
- Mainly (unmodified) Linux as guest OS, but others on request

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



L4Re: KVM/L4

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



Shows flexibility of L4Re architecture:
integration of existing virtualization solutions, e.g. KVM

L^4Linux

. . . is a paravirtualized Linux running as a user-level application on top of L4Re; first presented at SOSP'97

- Regard "L4Re" as new *hardware* platform in Linux and implement
  - Syscall interface: kernel entry, signal delivery, copy from/ to userspace
  - Hardware access: CPU state/features, MMU, interrupts, MMIO & port I/O

L^4Linux

. . . is a paravirtualized Linux running as a user-level application on top of L4Re; first presented at SOSP'97

- Regard "L4Re" as new *hardware* platform in Linux and implement
  - Syscall interface: kernel entry, signal delivery, copy from/ to userspace
  - Hardware access: CPU state/features, MMU, interrupts, MMIO & port I/O
- Stub drivers to connect to other L4 services

L^4Linux

. . . is a paravirtualized Linux running as a user-level application on top of L4Re; first presented at SOSP'97

- Regard "L4Re" as new *hardware* platform in Linux and implement
  - Syscall interface: kernel entry, signal delivery, copy from/ to userspace
  - Hardware access: CPU state/features, MMU, interrupts, MMIO & port I/O
- Stub drivers to connect to other L4 services
- Slowdown of ~5% for typical loads

L^4Linux

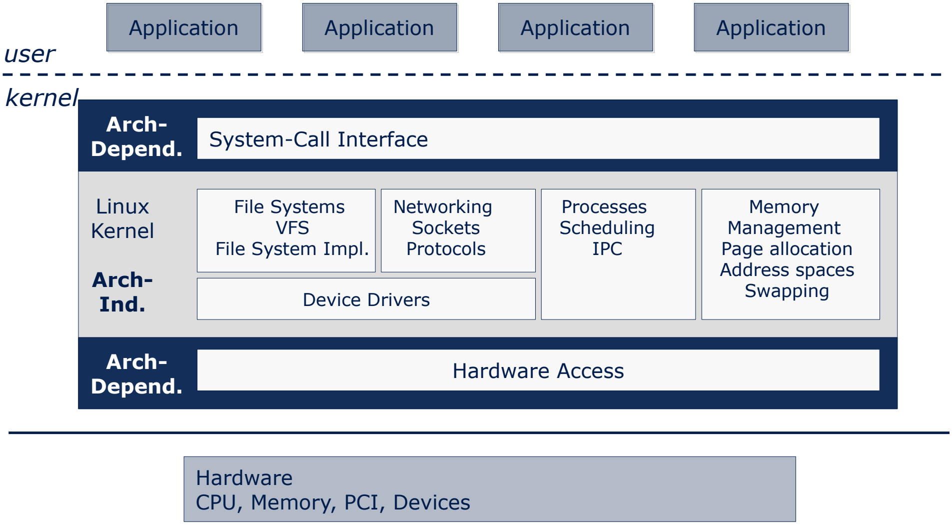
. . . is a paravirtualized Linux running as a user-level application on top of L4Re; first presented at SOSP'97

- Regard "L4Re" as new *hardware* platform in Linux and implement
  - Syscall interface: kernel entry, signal delivery, copy from/ to userspace
  - Hardware access: CPU state/features, MMU, interrupts, MMIO & port I/O
- Stub drivers to connect to other L4 services
- Slowdown of ~5% for typical loads
- Supports x86-32, x86-64, ARM32 and ARM64 (aarch64), including SMP

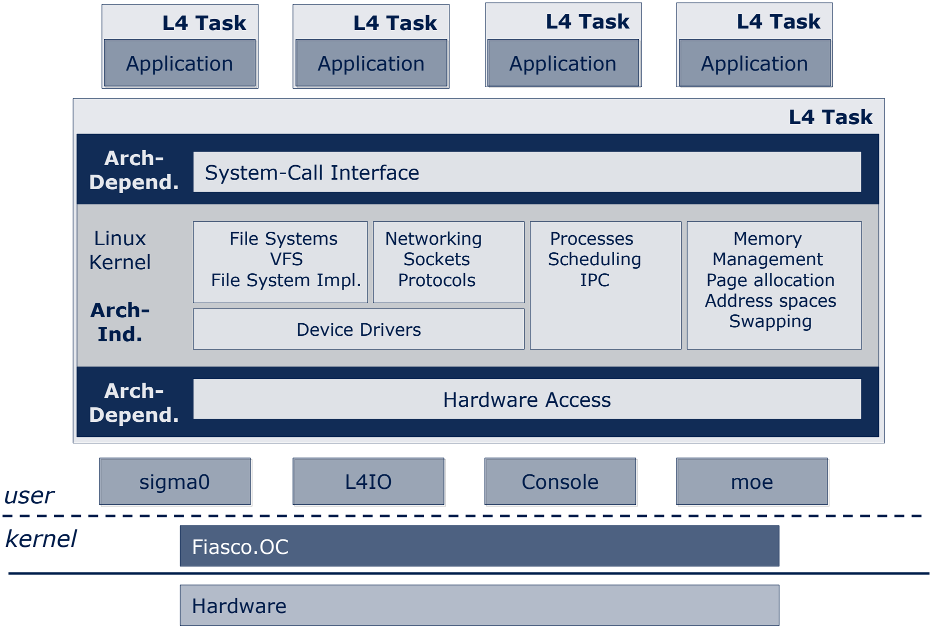
…is a paravirtualized Linux running as a user-level application on top of L4Re; first presented at SOSP'97

- Regard "L4Re" as new *hardware* platform in Linux and implement
  - Syscall interface: kernel entry, signal delivery, copy from/ to userspace
  - Hardware access: CPU state/features, MMU, interrupts, MMIO & port I/O
- Stub drivers to connect to other L4 services
- Slowdown of ~5% for typical loads
- Supports x86-32, x86-64, ARM32 and ARM64 (aarch64), including SMP
- Actively maintained (latest release based on Linux 6.10) and used in production

L^4Linux Architecture



L[4]Linux Architecture



Software Abstractions

Interface between kernel/microhypervisor and user-level/VMM

Requirements:

- Asynchronous execution model of OS kernels (IRQs)
- Paravirtualization + hardware-assisted virtualisation
- Smooth integration into system

Challenges

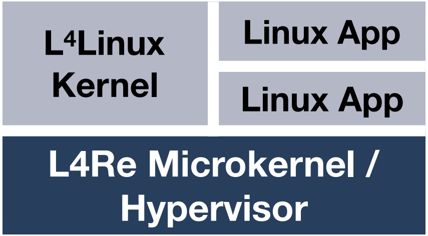
Fundamental problem: How to map three logical levels of privilege (Linux application, Linux kernel, L4Re microkernel/hypervisor) onto the two privilege levels the platform provides (user/kernel mode)?

Challenges

Fundamental problem: How to map three logical levels of privilege (Linux application, Linux kernel, L4Re microkernel/hypervisor) onto the two privilege levels the platform provides (user/kernel mode)?

CPU: Run Linux kernel & applications in microkernel user land

Memory: Linux kernel manages memory for Linux applications



vCPU

Regular/“Legacy” L4 Thread

- Executes XOR waits (for event, messages, IRQs)
- Hard to map OS kernel onto

vCPU

Regular/"Legacy" L4 Thread

- Executes XOR waits (for event, messages, IRQs)
- Hard to map OS kernel onto

vCPU

- Similar to how a CPU works: executes AND get interrupts
- "Interruptible thread"

vCPU

Regular/"Legacy" L4 Thread

- Executes XOR waits (for event, messages, IRQs)
- Hard to map OS kernel onto

vCPU

- Similar to how a CPU works: executes AND get interrupts
- "Interruptible thread"

- L4 thread (any thread can become a vCPU)

vCPU

Regular/“Legacy” L4 Thread

- Executes XOR waits (for event, messages, IRQs)
- Hard to map OS kernel onto

vCPU

- Similar to how a CPU works: executes AND get interrupts
- “Interruptible thread”

- L4 thread (any thread can become a vCPU)
- Interrupt-style excution

vCPU

Regular/“Legacy” L4 Thread

- Executes XOR waits (for event, messages, IRQs)
- Hard to map OS kernel onto

vCPU

- Similar to how a CPU works: executes AND get interrupts
- “Interruptible thread”

- L4 thread (any thread can become a vCPU)
- Interrupt-style excution
    - Events transition execution to user-defined entry points (“entry vector”)

vCPU

Regular/"Legacy" L4 Thread

- Executes XOR waits (for event, messages, IRQs)
- Hard to map OS kernel onto

vCPU

- Similar to how a CPU works: executes AND get interrupts
- "Interruptible thread"

- L4 thread (any thread can become a vCPU)
- Interrupt-style excution
  - Events transition execution to user-defined entry points ("entry vector")
  - Virtual interrupt flag (IRQs disabled == normal thread)

vCPU

Regular/"Legacy" L4 Thread

- Executes XOR waits (for event, messages, IRQs)
- Hard to map OS kernel onto

vCPU

- Similar to how a CPU works: executes AND get interrupts
- "Interruptible thread"

- L4 thread (any thread can become a vCPU)
- Interrupt-style excution
  - Events transition execution to user-defined entry points ("entry vector")
  - Virtual interrupt flag (IRQs disabled == normal thread)
- Virtual user mode

vCPU

Regular/"Legacy" L4 Thread

- Executes XOR waits (for event, messages, IRQs)
- Hard to map OS kernel onto

vCPU

- Similar to how a CPU works: executes AND get interrupts
- "Interruptible thread"

- L4 thread (any thread can become a vCPU)
- Interrupt-style excution
  - Events transition execution to user-defined entry points ("entry vector")
  - Virtual interrupt flag (IRQs disabled == normal thread)
- Virtual user mode
  - vCPU can switch to a different L4 task (address space) for execution

vCPU

Regular/“Legacy” L4 Thread

- Executes XOR waits (for event, messages, IRQs)
- Hard to map OS kernel onto

vCPU

- Similar to how a CPU works: executes AND get interrupts
- “Interruptible thread”

- L4 thread (any thread can become a vCPU)
- Interrupt-style excution
  - Events transition execution to user-defined entry points (“entry vector”)
  - Virtual interrupt flag (IRQs disabled == normal thread)
- Virtual user mode
  - vCPU can switch to a different L4 task (address space) for execution
  - Returns to “home task”/kernel for any received event

vCPU

Regular/“Legacy” L4 Thread

- Executes XOR waits (for event, messages, IRQs)
- Hard to map OS kernel onto

vCPU

- Similar to how a CPU works: executes AND get interrupts
- “Interruptible thread”

- L4 thread (any thread can become a vCPU)
- Interrupt-style excution
  - Events transition execution to user-defined entry points (“entry vector”)
  - Virtual interrupt flag (IRQs disabled == normal thread)
- Virtual user mode
  - 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

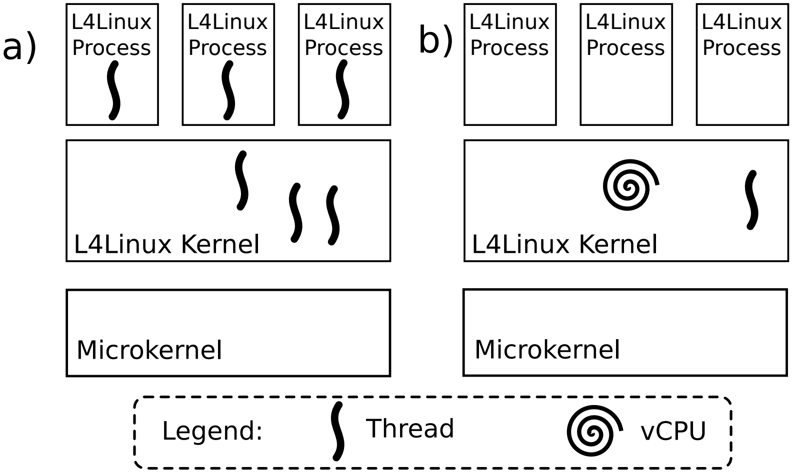
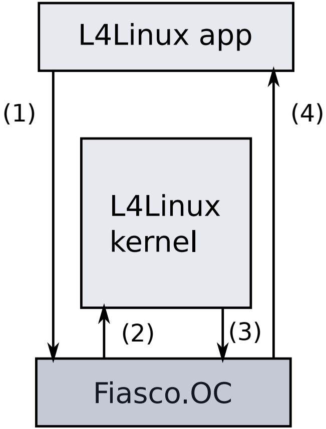


FIGURE 3: *(a) L4Linux implemented with threads and (b) L4Linux implemented with vCPUs.*

—from Lackorzynski, Warg, Peter "Virtual Processors as Kernel Interface", 2010

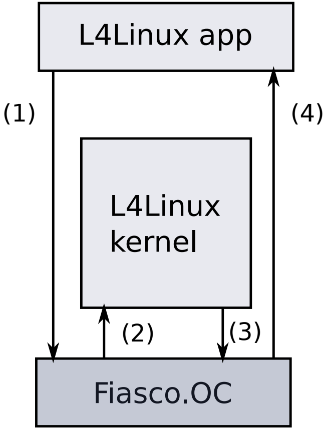
Event Processing, e.g. Syscalls

- Linux syscall interface (`int 0x80`) causes trap



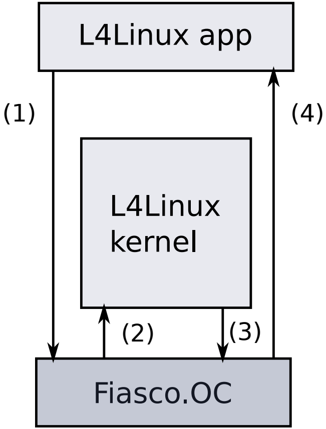
Event Processing, e.g. Syscalls

- Linux syscall interface (`int 0x80`) causes trap
- L^4Linux server receives exception IPC



Event Processing, e.g. Syscalls

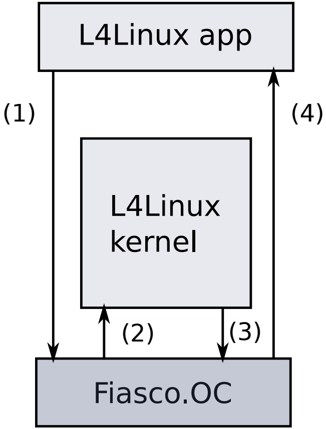
- Linux syscall interface (`int 0x80`) causes trap
- L4Linux server receives exception IPC
- Heavyweight compared to native Linux system calls:
  - Two address space switches (native: zero)
  - Two Fiasco kernel entries + exits (native: one)



Event Processing, e.g. Syscalls

- Linux syscall interface (`int 0x80`) causes trap
- L^4Linux server receives exception IPC
- Heavyweight compared to native Linux system calls:
  - Two address space switches (native: zero)
  - Two Fiasco kernel entries + exits (native: one)

⇒ Hardware-assisted virtualisation
  - Nicely integrates into vCPU abstraction
  - Nested paging by `L4::Task`/`L4::VM`



Device Access

- Options:
  - Exclusive: Pass-through
  - SR-IOV: Hardware-assisted sharing (virtualised hardware)
  - Sharing: Microkernel-based service/driver + guest interface (VirtIO)

Device Access

- Options:
  - Exclusive: Pass-through
  - SR-IOV: Hardware-assisted sharing (virtualised hardware)
  - Sharing: Microkernel-based service/driver + guest interface (VirtIO)
- Pass through resources:
  - MMIO (direct mapping)
  - Interrupts via Microkernel/Hypervisor
  - Interrupts delivered directly to guest on recent hardware

VirtIO

- Standard for virtual devices
- Defines common data structures
- Widely supported (Linux, *BSD, Windows, QNX, . . .)
- Optimised for virtualisation, but also usable for hardware devices

IOMMU

- Important hardware building block
- MMU for devices
- Indirection & Protection
  - Limit device access to memory → prevents DMA attacks by guests, devices/firmware, . . .
  - Guest can use gPA (instead of hPA) to program DMA
- Programmed by assigning `L4::Task` to device

L^4Linux as a Toolbox

Reuse large parts of code from Linux:

- Filesystems
- Network stack
- Device drivers
- ...

L^4Linux as a Toolbox

Reuse large parts of code from Linux:

- Filesystems
- Network stack
- Device drivers
- ...

Hybrid applications can provide these services to native L4 applications.

Research Paths

- Realtime: fully preemptive kernel, realtime drivers & services

Research Paths

- Realtime: fully preemptive kernel, realtime drivers & services
- Security: capability system, reduced TCB

Research Paths

- Realtime: fully preemptive kernel, realtime drivers & services
- Security: capability system, reduced TCB
- HPC/Cloud: scalability, OS-noise/execution variability, isolation

Research Paths

- Realtime: fully preemptive kernel, realtime drivers & services
- Security: capability system, reduced TCB
- HPC/Cloud: scalability, OS-noise/execution variability, isolation

- Techniques

Research Paths

- Realtime: fully preemptive kernel, realtime drivers & services
- Security: capability system, reduced TCB
- HPC/Cloud: scalability, OS-noise/execution variability, isolation

- Techniques
  - Combining critical & non-critical applications in a single system

Research Paths

- Realtime: fully preemptive kernel, realtime drivers & services
- Security: capability system, reduced TCB
- HPC/Cloud: scalability, OS-noise/execution variability, isolation

- Techniques
  - Combining critical & non-critical applications in a single system
  - Split off critical parts from applications and run them on L4Re

Research Paths

- Realtime: fully preemptive kernel, realtime drivers & services
- Security: capability system, reduced TCB
- HPC/Cloud: scalability, OS-noise/execution variability, isolation

- Techniques
  - Combining critical & non-critical applications in a single system
  - Split off critical parts from applications and run them on L4Re
  - Use VM to provide comon runtime, then *decouple* critical applications