Hypervisors

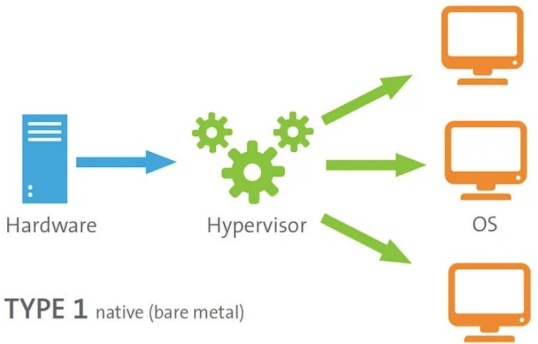
Agenda

- Virtualization & Hypervisors Concepts
- HW Support Virtualization & KVM-QEMU
- QEMU Internals
- KVM Virtual Machines

Virtualization

What is virtualization?

- In computer science,
  virtual means "a hardware environment that is not real".
- We duplicate the functions of physical hardware and present them to an operating system.
- The physical system that runs the virtualization software (***hypervisor*** or ***Virtual Machine Monitor VMM***) is called a **host** and the virtual machines installed on top of the hypervisor are called ***guests***.
- Different hypervisors are there in the market
  Xen, Kvm-Qemu, Vmware hypervisors, Oracle virtualbox,…



Virtualization

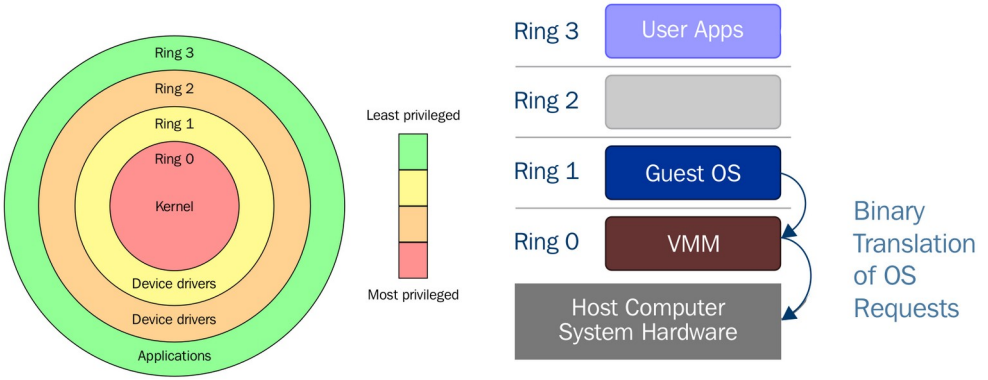
Types of virtualization

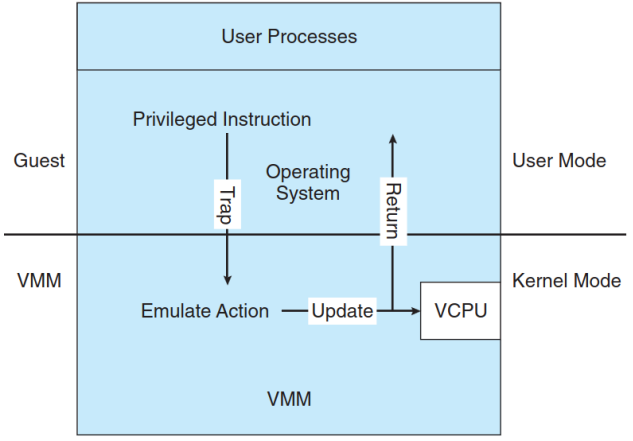
- Full virtualization
- Para-virtualization
- Hardware assisted virtualization

Virtualization

Types of virtualization

- Full virtualization
  - It relies on ***binary translation*** techniques.
  - In binary translation, some system calls are interpreted and dynamically rewritten.
  - **Guest** OS in Ring 1 can access the host for privileged instructions through the **VMM** in Ring 0.
  - With this approach, the **critical instructions** *(i.e I/O, system calls, interrupts, exceptions, page faults,...)* are discovered dynamically at runtime and replaced with **traps** into the **VMM** that are to be emulated in software.
  - This technique is called **Trap&Emulate**.
    Trap&Emulate VMM:
    Guest OS runs at lower privilege level than **VMM**, traps to **VMM** for privileged operation.

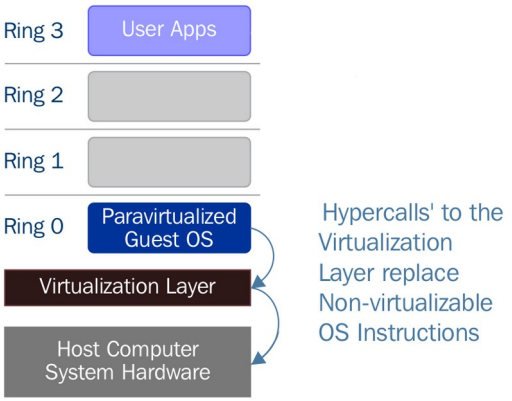




Virtualization

Types of virtualization

- Para-virtualization
  - In para-virtualization, the guest operating system source needs to be **modified** "**Front-end**"in order to allow those instructions to access **Ring 0**.
  - The operating system needs to be modified to communicate between the VMM/hypervisor and the guest through the "**Back-end**" (**hypercalls**) path.
  - The hypervisor provides an API and the OS of the guest virtual machine calls that API which require host operating system modifications.
  - In this case, the modified guest operating system can run in Ring 0.
  - This technique results in **greater performance** compared to full virtualization, however requires specialized **guest kernel which is aware** of para virtualization technique and come with needed software support.



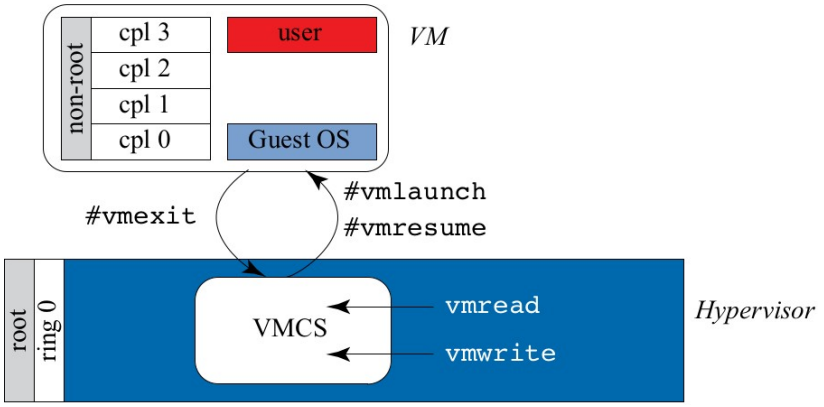
Virtualization

Types of virtualization

- Hardware assisted virtualization
  - **i.e**
    Intel and **AMD** created new processor extensions of the x86 architecture, called Intel **VT-x** and **AMD-V** respectively.

    Introducing Virtualization Technology processors for x86 brought an extra instruction set called ***Virtual Machine Extensions*** or ***VMX***.

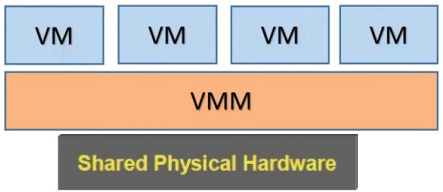
    So, x86 has 2 modes of execution **non-root** mode, and **root** mode where both have 4 cpu **rings** or **levels**.
    VM runs in a **non-root** mode and the hypervisor runs in **root-mode**
  - *Will talk about **x86-64** and **ARM** CPU virtualization later in details.*



Hypervisor

What's Hypervisor?

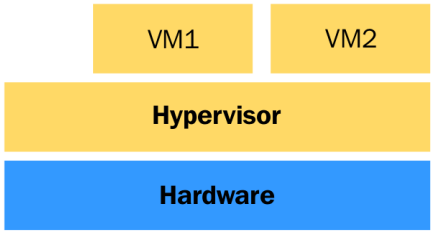
- Hypervisor is the ***VMM*** (Virtual Machine Monitor).
- So hypervisor is the VM monitor that is responsible for,
  - Providing virtual hardware.
  - Switching between **VMs** and save switching context.
  - Memory translation and I/O mapping.
  - VM life cycle management.
  - VMs migration.
  - ...

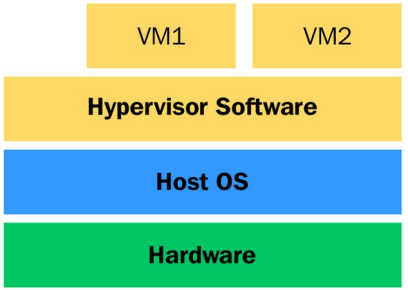


Hypervisor

Hypervisor Types

- There is no clear or standard definition of ***Type 1*** and ***Type 2*** hypervisors, however generally,
- ***Type 1***
  - If the VMM/hypervisor runs directly on top of the hardware, it's considered to be a Type 1.
  - Small in size, easy to install and less overhead.
  - Ex. QNX Hypervisor, Xen, Hyper-V, Oracle VM Server.
- ***Type 2***
  - Type 2 hypervisor resides on top of the operating system.
  - Wide range of hardware support, because the underlying host OS is controlling hardware access.
  - Ex. Kvm-Qemu, Virtualbox.



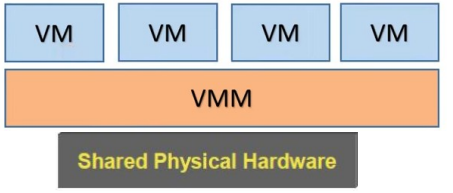


Hardware Support
Virtualization

Popek/Goldberg Theorem

What's Popek/Goldberg Theorem

- In 1974, Gerald Popek and Robert Goldberg published a paper called "***Formal Requirements for Virtualizable Third-Generation Architectures***" that defines the necessary and sufficient formal requirements to ensure that a VMM can be constructed.
- The Theorem
  - Determines whether a given instruction set architecture (ISA) can be virtualized by a VMM using **multiplexing**.
  - For any architecture that meets the hypothesis of the theorem, any operating system directly running on the hardware can also run inside a virtual machine, **without modifications.**
  - Guest operating systems could run directly in virtual machines, without having to resort to software translation or **para-virtualization**.
  - Actually different **terminologies** proposed, however for now no need to deep dive into the theorem details.

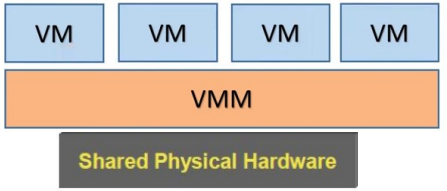


Command	Operands	Operation	Description
ADC	Rd, Rr	Rd←Rd + Rr + C	Add two registers with carry
ADD	Rd,Rr	Rd←Rd + Rr	Add two registers
ADIW	Rh:Rl, K	Rh:Rl←Rh:Rl + K	Add immediate to Word (0 ≤ K ≤ 63)
AND	Rd,Rr	Rd←Rd • Rr	Logical AND two registers
ANDI	Rd, K	Rd←Rd • K	Logical AND with immediate (16 ≤ d ≤ 31)
ASR	Rd	C←Rd(0), Rd(6…0) ←Rd(7…1), Rd(7) ←Rd(7)	Arithmetic shift right

Popek/Goldberg Theorem

The Model

- The theorem must ensure compliance with the following three criteria,
- **Equivalence**
  - Any **program** running within the virtual machine, i.e., any guest operating system and application combination, should exhibit identical behavior as if that program had run directly on the underlying hardware.
- **Safety**
  - The VMM must be in complete control of the hardware at all times.
  - A virtual machine is isolated from the underlying hardware.
  - Ensuring that there is no shared state within the VMM between two virtual machines.
- **Performance**
  - The efficiency requirement implies that the execution speed of the program in a virtualized environment is at worst a minor decrease over the execution time when run directly on the underlying hardware.



x86-64: CPU Virtualization with VT-x

Design Requirements

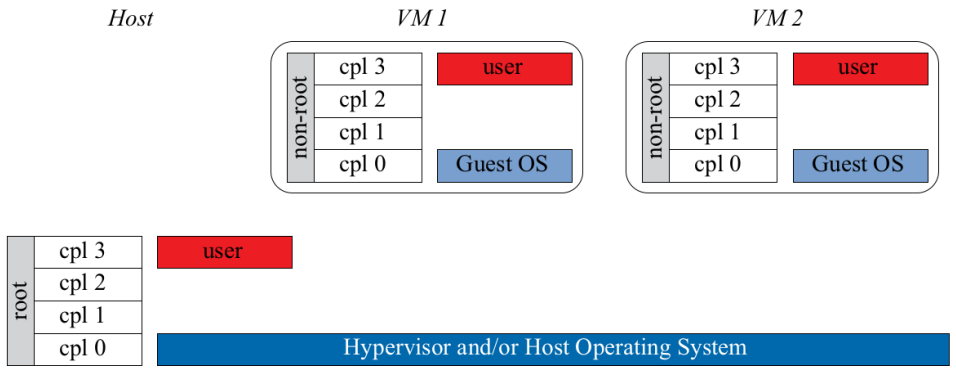
- In designing VT-x, Intel's central design goal is to fully meet the requirements of the Popek/Goldberg theorem.
  - **Equivalence**
  - **Safety**
  - **Performance**

> A central design goal for Intel Virtualization Technology is to eliminate the need for CPU paravirtualization and binary translation techniques, and thereby enable the implementation of VMMs that can support a broad range of unmodified guest operating systems while maintaining high levels of performance.
>
> R. Uhlig et al., *IEEE Computer*, 2005 [171]

x86-64: CPU Virtualization with VT-x

VT-X Architecture

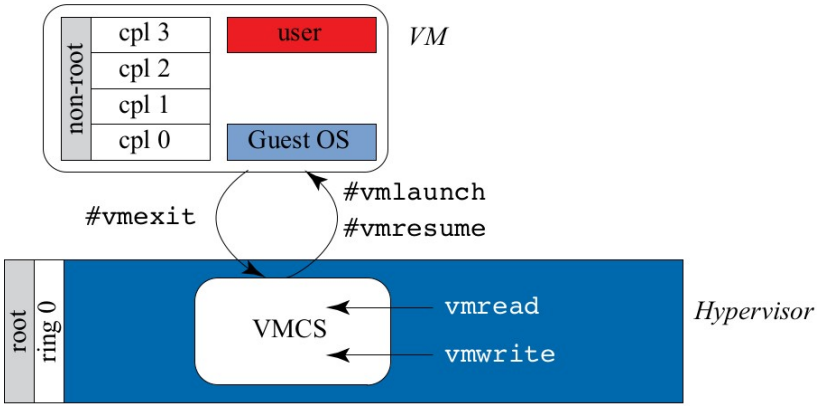
- The architecture of VT-x is based on a central design decision: do not change the semantics of individual instructions of the **ISA**.
- VT-x **duplicates** the entire architecturally visible state of the processor and introduces a new mode of execution: the **root mode**.
- **Hypervisors** and **host operating** systems run in **root mode** whereas **virtual machines** execute in **non-root mode (VMX)**.
- Architectural extension major **properties**
  - This new mode (root vs. non-root) is only used for virtualization.
  - It is orthogonal to all other modes of execution of the CPU (e.g., real mode, v8086 mode, protected mode), which are available in both modes. It is also **orthogonal** to the protection levels of protected mode (e.g cpl=0--cpl=3) with all four levels separately available to each mode.
  - Each mode defines its own distinct, complete 64-bit linear address space. Each **address space** is defined by a **distinct page table** tree with a distinct page table register. Only the address space corresponding to the current mode is active in the **TLB**, and the **TLB** changes atomically as part of the transitions.
  - **External interrupts** are generally delivered in **root mode** and trigger a **transition** from non-root mode if necessary. The transition occurs even when non-root interrupts are disabled.



x86-64: CPU Virtualization with VT-x

Transition Between ROOT and NON-ROOT Modes

- The state of the virtual machine is stored in a dedicated structure in physical memory called the Virtual Machine Control Structure (**VMCS**).
- Once initialized, a virtual machine resumes execution through a **vmresume** instruction.
- This **privileged** instruction loads the state from the **VMCS** (***Virtual Machine Control Structure***) in memory into the **register file** and performs an atomic transition between the host environment and the guest environment.
- The virtual machine then executes in non-root mode until the **first trap** that must be handled by the **hypervisor** or the next external interrupt.
- This transition from non-root mode to root mode is called a **vmexit**.



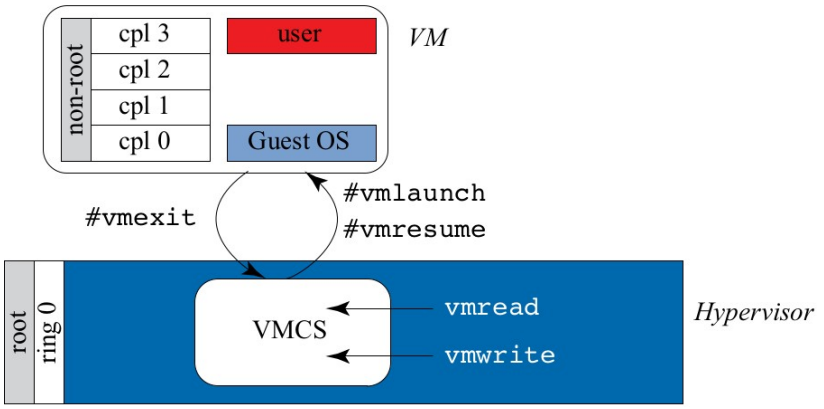
Intel instruction	purpose
VMXON	Enable VMX
VMXOFF	Disable VMX
VMLAUNCH	Start/enter VM
VMRESUME	Re-enter VM
VMCLEAR	Null out/reinitialize VMCS
VMPTRLD	Load the current VMCS
VMPTRST	Store the current VMCS
VMREAD	Read values from VMCS
VMWRITE	Write values to VMCS
VMCALL	Exit virtual machine to VMM
VMFUNC	Invoke a VM function in VMM without exiting guest operation

x86-64: CPU Virtualization with VT-x

Transition Between ROOT and NON-ROOT Modes

- List of possible **reasons** for a **vmexit.**

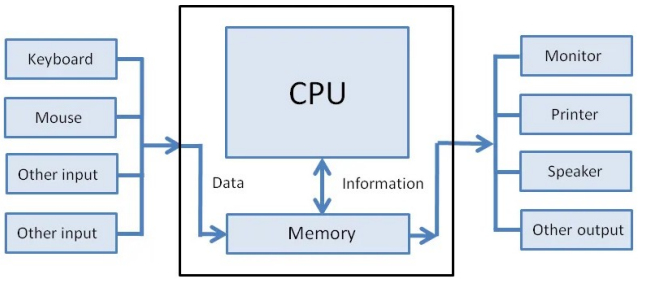
Category	Exit Reason	Description
Exception	0	Any guest instruction that causes an exception
Interrupt	1	The exit is due to an external I/O interrupt
Triple fault	2	Reset condition (bad)
Interrupt window	7	The guest can now handle a pending guest interrupt
Legacy emulation	9	Instruction is not implemented in non-root mode; software expected to provide backward compatibility, e.g., `task switch`
Root-mode Sensitive	11-17, 28-29, 31-32, 46-47:	x86 privileged or sensitive instructions: `getsec, hlt, invd, invlpg, rdpmc, rdtsc, rsm, mov-cr, mov-dr, rdmsr, wrmsr, monitor, pause, lgdt, lidt, sgdt, sidt, lldt, ltr, sldt`
Hypercall	18	`vmcall` : Explicit transition from non-root to root mode
VT-x new	19-27, 50, 53	ISA extensions to control non-root execution: `invept, invvpid, vmclear, vmlaunch, vmptrld, vmptrst, vmreas, vmresume, vmwrite, vmxoff, vmxon`
I/O	30	Legacy I/O instructions
EPT	48-49	EPT violations and misconfigurations



Virtualization Components

Basic Virtualized Components

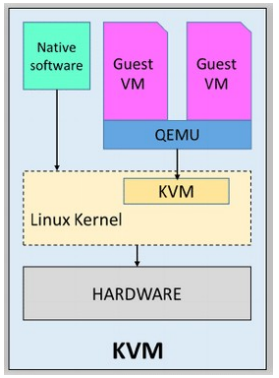
- CPUs
- Memories & MMU
- Input/Output (IO) devices

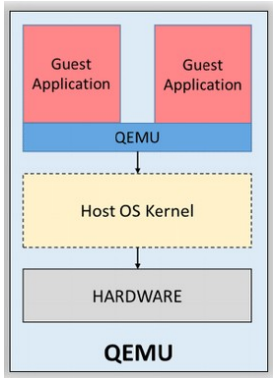


x86-64: CPU Virtualization with VT-x

KVM vs QEMU

- KVM is the Linux-based Kernel Virtual Machine.
- KVM is the most relevant open-source **type-2 hypervisor**.
- KVM lets you turn Linux into a hypervisor that allows a host machine to run multiple, isolated virtual environments called guests or virtual machines (**VMs**).
- QEMU (Quick Emulator) is a free and open-source HW emulator.
- QEMU code can also emulate number of CPUs (Out of scope).
- **QEMU** alone is a complete machine simulator with support for cross-architectural **binary translation of the CPU**, and a complete set of I/O device models.





QEMU Emulation

Emulated IO

- Input
- Display
- Character / Serial
- Audio
- Block
- Networking

Bus emulation

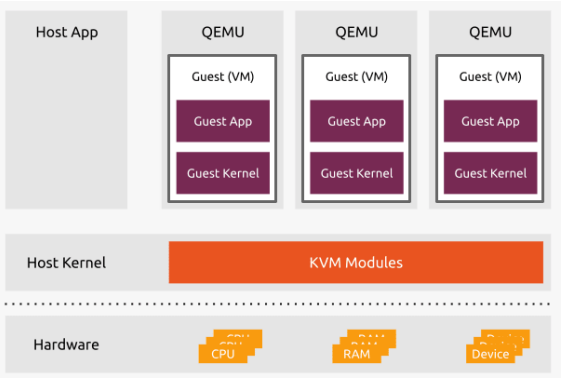
- MMIO
- PCI / PCIe
- SSI (SPI)
- I2C
- PCMCIA
- SCSI
- USB

Alpha
Arm (arm, aarch64)
AVR
Cris
Hexagon
PA-RISC (hppa)
x86 (i386, x86_64)
Loongarch
m68k
Microblaze
MIPS (mips*)
OpenRISC

x86-64: CPU Virtualization with VT-x

KVM: A hypervisor For VT-X

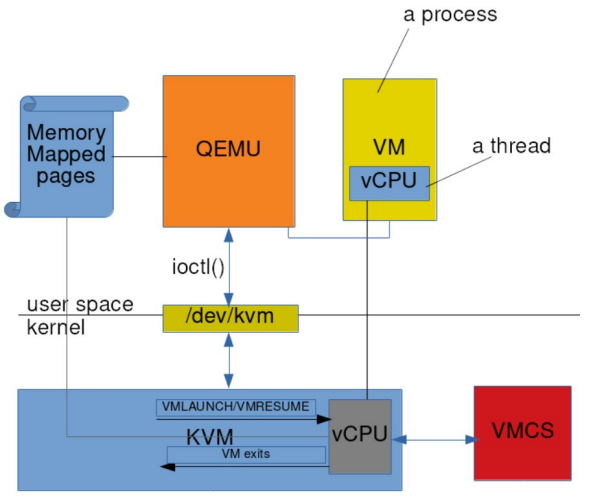
- Together **KVM** & **QEMU**,
  - This combination is considered a **type-2 hypervisor**.
  - QEMU responsible for the user-space implementation of all **I/O** device emulation.
  - The KVM kernel module responsible to multiplex the **CPU** and **MMU** of the processor.



x86-64: CPU Virtualization with VT-x

KVM: A hypervisor For VT-X

- Together **KVM** & **QEMU**,
  - QEMU is just running in a user-space process.
  - KVM exposes a sudo device file entry in ***/dev/kvm.***
  - QEMU allocates memory via **mmap** for guest VM physical memory.
  - QEMU creates one **thread** for each virtual CPU(**VCPU**) in the guest.
  - Host OS sees QEMU as a regular **multi-threaded** process.
  - QEMU talks to **/dev/kvm** normally through **ioctl**.
  - *Will talk about **QEMU-KVM interaction** later in details.*



Simplified host-kvm

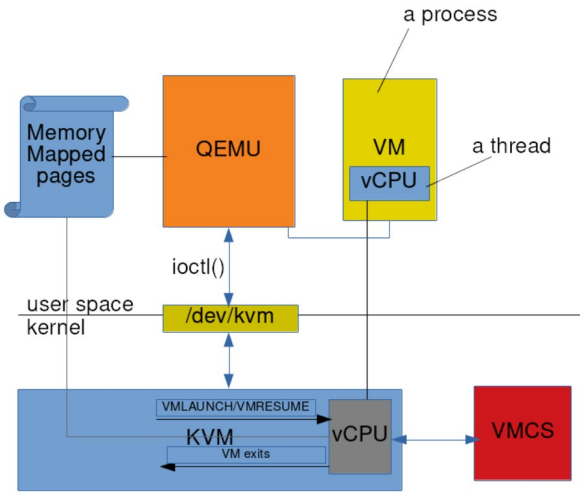
```
kvm_fd = open("/dev/kvm", O_RDWR);
vm_fd = ioctl(kvm_fd, KVM_CREATE_VM, 0);
mmap(vm_img_sz);
vcpu_fd = ioctl(vm_fd, KVM_CREATE_VCPU, 0);
mmap(sz_for_each_vcpu);
for (;;) {
  ioctl(vcpu_fd, KVM_RUN, 0);
  switch (run->exit_reason) {
    case KVM_EXIT_IO:
      /* Emulate IO */
      break;
    case KVM_EXIT_SHUTDOWN:
    /* Exit */
  }
}
```

x86-64: CPU Virtualization with VT-x

KVM: A hypervisor For VT-X

- KVM adopted **Popek/Goldberg's** three core attributes of a virtual machine.
  - **Equivalence**
    A KVM virtual machine should be able to run any x86 operating system (32-bit or 64-bit) and all of its applications without any modifications
  - **Safety**
    KVM virtualizes all resources visible to the virtual machine, including CPU, physical memory, I/O busses and devices, and BIOS firmware. The KVM hypervisor remains in complete control of the virtual machines at all times.
  - **Performance**
    KVM's design makes a careful tradeoff to ensure that all performance critical components are handled within the KVM kernel module while limiting the complexity of that kernel module.

    KVM kernel module handles only the core platform functions associated with the emulation of the x86 processor, the **MMU, the interrupt subsystem (**inclusive of the **APIC, IOAPIC**, etc.).
    All functions responsible for **I/O emulation** are handled in user-space (**QEMU**).



x86-64: CPU Virtualization with VT-x

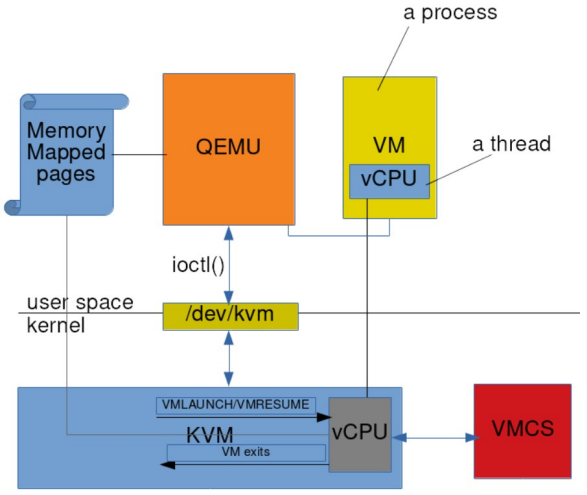
KVM Kernel Module *(kvm.ko)*

- Create vm

/virt/kvm/kvm_main.c

```
static long kvm_dev_ioctl(struct file *filp,
                          unsigned int ioctl, unsigned long arg)
{
    int r = -EINVAL;
    switch (ioctl) {
    case KVM_CREATE_VM:
        r = kvm_dev_ioctl_create_vm(arg);
    ...
    }
    ...
}

static int kvm_dev_ioctl_create_vm(unsigned long type)
{
    ...
    kvm = kvm_create_vm(type, fdname);
    ...
}

static struct kvm *kvm_create_vm(unsigned long type, const char *fdname)
{
    struct kvm *kvm = kvm_arch_alloc_vm();
    ...
}

static inline struct kvm *kvm_arch_alloc_vm(void)
{
    return kzalloc(sizeof(struct kvm), GFP_KERNEL_ACCOUNT);
}
```

```
struct kvm {

    struct mm_struct *mm; /* userspace tied to this vm */
    unsigned long nr_memslot_pages;
    /* The two memslot sets - active and inactive (per address space) */
    struct kvm_memslots __memslots[KVM_MAX_NR_ADDRESS_SPACES][2];
    /* The current active memslot set for each address space */
    struct kvm_memslots __rcu *memslots[KVM_MAX_NR_ADDRESS_SPACES];
    struct xarray vcpu_array;
    ...
    int max_vcpus;
    int created_vcpus;
    ..
};
```



Simplified host-kvm

```
kvm_fd = open("/dev/kvm", O_RDWR);
vm_fd = ioctl(kvm_fd, KVM_CREATE_VM, 0);
mmap(vm_img_sz);
vcpu_fd = ioctl(vm_fd, KVM_CREATE_VCPU, 0);
mmap(sz_for_each_vcpu);
for (;;) {
  ioctl(vcpu_fd, KVM_RUN, 0);
  switch (run->exit_reason) {
    case KVM_EXIT_IO:
      /* Emulate IO */
      break;
    case KVM_EXIT_SHUTDOWN:
    /* Exit */
  }
}
```

x86-64: CPU Virtualization with VT-x

KVM Kernel Module *(kvm.ko)*

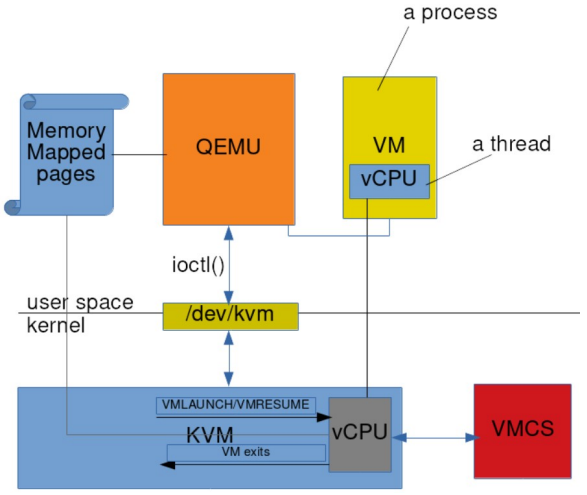
- Create vcpu

/virt/kvm/kvm_main.c

```
static long kvm_vm_ioctl(struct file *filp,
           unsigned int ioctl, unsigned long arg)
{
    struct kvm *kvm = filp->private_data;
    void __user *argp = (void __user *)arg;
    int r;
    switch (ioctl) {
    case KVM_CREATE_VCPU:
        r = kvm_vm_ioctl_create_vcpu(kvm, arg);
        break;
    ...
    }
    ...
}

static int kvm_vm_ioctl_create_vcpu(struct kvm *kvm, u32 id)
{
    struct kvm_vcpu *vcpu;
    struct page *page;
    ...
    vcpu = kmem_cache_zalloc(kvm_vcpu_cache, GFP_KERNEL_ACCOUNT);
    page = alloc_page(GFP_KERNEL_ACCOUNT | __GFP_ZERO);
    vcpu->run = page_address(page);
    kvm_vcpu_init(vcpu, kvm, id);
    r = kvm_arch_vcpu_create(vcpu);
    ...
}
```

/arch/x86/kvm/x86.c

```
int kvm_arch_vcpu_create(struct kvm_vcpu *vcpu)
{
    struct page *page;
    int r;
    ...
    kvm_gpc_init(&vcpu->arch.pv_time, vcpu->kvm);
    r = kvm_mmu_create(vcpu);
    r = kvm_create_lapic(vcpu, lapic_timer_advance_ns);
    ...
}
```



Simplified host-kvm

```
kvm_fd = open("/dev/kvm", O_RDWR);
vm_fd = ioctl(kvm_fd, KVM_CREATE_VM, 0);
mmap(vm_img_sz);
vcpu_fd = ioctl(vm_fd, KVM_CREATE_VCPU, 0);
mmap(sz_for_each_vcpu);
for (;;) {
  ioctl(vcpu_fd, KVM_RUN, 0);
  switch (run->exit_reason) {
    case KVM_EXIT_IO:
      /* Emulate IO */
      break;
    case KVM_EXIT_SHUTDOWN:
    /* Exit */
  }
}
```

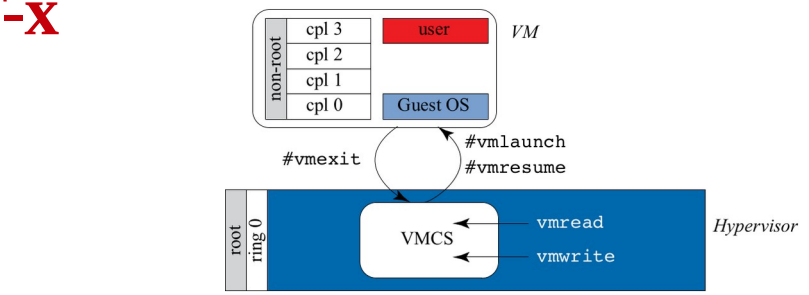
x86-64: CPU Virtualization with VT-x

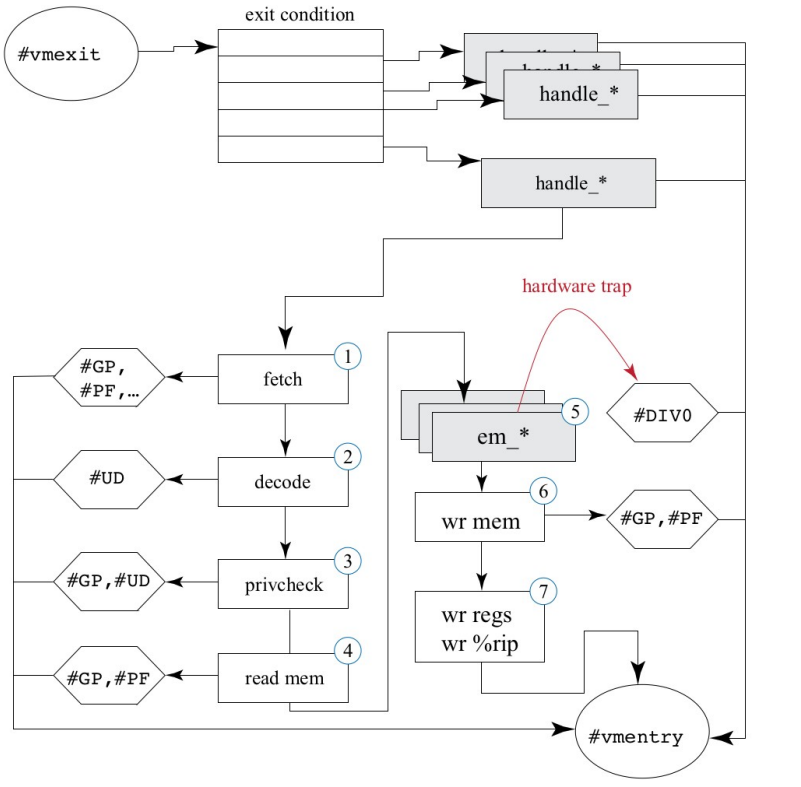
KVM Kernel Module *(kvm.ko)*

- KVM flow from the original **vmexit** until the **vmresume** instruction returns to **non-root mode**.
- Upon the **vmexit**, KVM first saves all the vcpu state in memory.
- KVM then performs dispatches : (***handler_****) for each exit reason. ***handler_**** rely on **VMCS** fields to determine the necessary emulation steps to perform.

```
/*
 * The exit handlers return 1 if the exit was handled fully and guest execution
 * may resume.  Otherwise they set the kvm_run parameter to indicate what needs
 * to be done to userspace and return 0.
 */
static int (*kvm_vmx_exit_handlers[])(struct kvm_vcpu *vcpu) = {
        [EXIT_REASON_EXCEPTION_NMI]           = handle_exception_nmi,
        [EXIT_REASON_EXTERNAL_INTERRUPT]      = handle_external_interrupt,
        [EXIT_REASON_TRIPLE_FAULT]            = handle_triple_fault,
        [EXIT_REASON_NMI_WINDOW]              = handle_nmi_window,
        [EXIT_REASON_IO_INSTRUCTION]          = handle_io,
        [EXIT_REASON_CR_ACCESS]               = handle_cr,
        [EXIT_REASON_DR_ACCESS]               = handle_dr,
        [EXIT_REASON_CPUID]                   = kvm_emulate_cpuid,
        [EXIT_REASON_MSR_READ]                = kvm_emulate_rdmsr,
        [EXIT_REASON_MSR_WRITE]               = kvm_emulate_wrmsr,
        [EXIT_REASON_INTERRUPT_WINDOW]        = handle_interrupt_window,
        [EXIT_REASON_HLT]                     = kvm_emulate_halt,
        [EXIT_REASON_INVD]                    = kvm_emulate_invd,
        [EXIT_REASON_INVLPG]                  = handle_invlpg,
        [EXIT_REASON_RDPMC]                   = kvm_emulate_rdpmc,
        [EXIT_REASON_VMCALL]                  = kvm_emulate_hypercall,
        [EXIT_REASON_VMCLEAR]                 = handle_vmx_instruction,
        [EXIT_REASON_VMLAUNCH]                = handle_vmx_instruction,
        [EXIT_REASON_VMPTRLD]                 = handle_vmx_instruction,
        [EXIT_REASON_VMPTRST]                 = handle_vmx_instruction,
        [EXIT_REASON_VMREAD]                  = handle_vmx_instruction,
```

/arch/x86/kvm/vmx/vmx.c





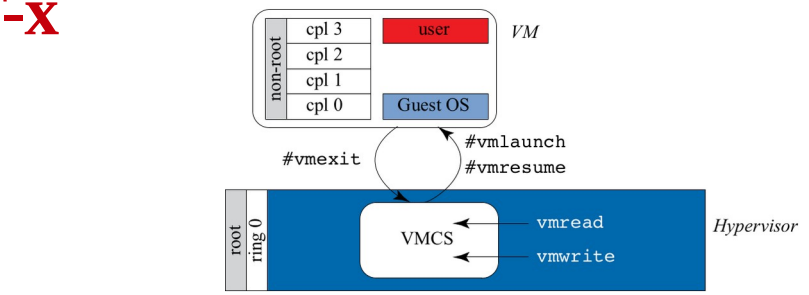
x86-64: CPU Virtualization with VT-x

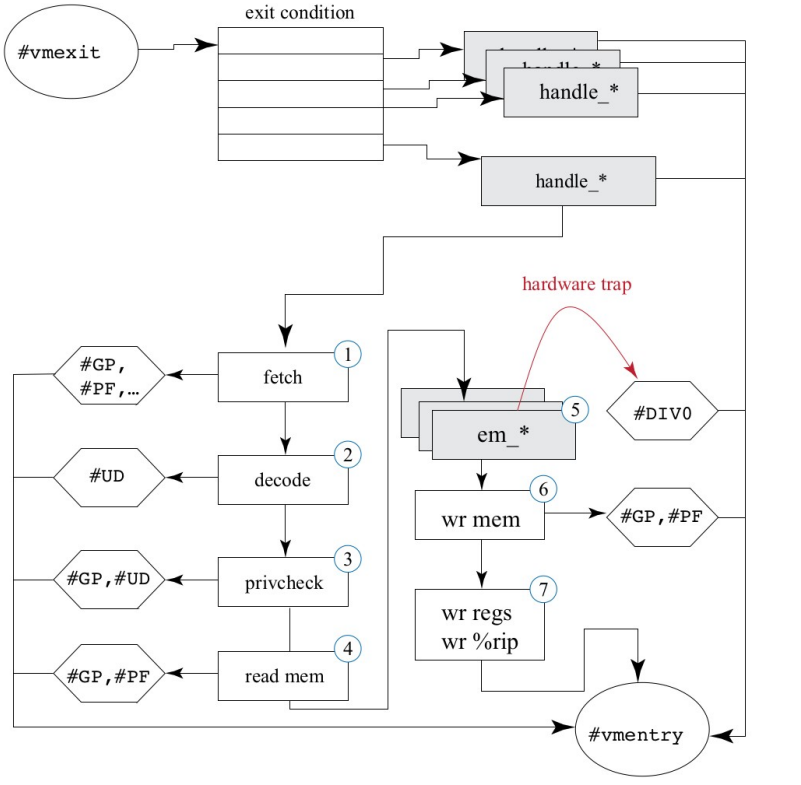
KVM Kernel Module *(kvm.ko)*

- KVM may need to:
  - Do nothing, for example when an external interrupt/fault occurs, which is handled by the underlying **host** operating system.

```
/*
 * The exit handlers return 1 if the exit was handled fully and guest execution
 * may resume.  Otherwise they set the kvm_run parameter to indicate what needs
 * to be done to userspace and return 0.
 */
static int (*kvm_vmx_exit_handlers[])(struct kvm_vcpu *vcpu) = {
        [EXIT_REASON_EXCEPTION_NMI]           = handle_exception_nmi,
        [EXIT_REASON_EXTERNAL_INTERRUPT]      = handle_external_interrupt,
        [EXIT_REASON_TRIPLE_FAULT]            = handle_triple_fault,
        [EXIT_REASON_NMI_WINDOW]              = handle_nmi_window,
        [EXIT_REASON_IO_INSTRUCTION]          = handle_io,
        [EXIT_REASON_CR_ACCESS]               = handle_cr,
        [EXIT_REASON_DR_ACCESS]               = handle_dr,
        [EXIT_REASON_CPUID]                   = kvm_emulate_cpuid,
        [EXIT_REASON_MSR_READ]                = kvm_emulate_rdmsr,
        [EXIT_REASON_MSR_WRITE]               = kvm_emulate_wrmsr,
        [EXIT_REASON_INTERRUPT_WINDOW]        = handle_interrupt_window,
        [EXIT_REASON_HLT]                     = kvm_emulate_halt,
        [EXIT_REASON_INVD]                    = kvm_emulate_invd,
        [EXIT_REASON_INVLPG]                  = handle_invlpg,
        [EXIT_REASON_RDPMC]                   = kvm_emulate_rdpmc,
        [EXIT_REASON_VMCALL]                  = kvm_emulate_hypercall,
        [EXIT_REASON_VMCLEAR]                 = handle_vmx_instruction,
        [EXIT_REASON_VMLAUNCH]                = handle_vmx_instruction,
        [EXIT_REASON_VMPTRLD]                 = handle_vmx_instruction,
        [EXIT_REASON_VMPTRST]                 = handle_vmx_instruction,
        [EXIT_REASON_VMREAD]                  = handle_vmx_instruction,
```

/arch/x86/kvm/vmx/vmx.c





x86-64: CPU Virtualization with VT-x

KVM Kernel Module *(kvm.ko)*

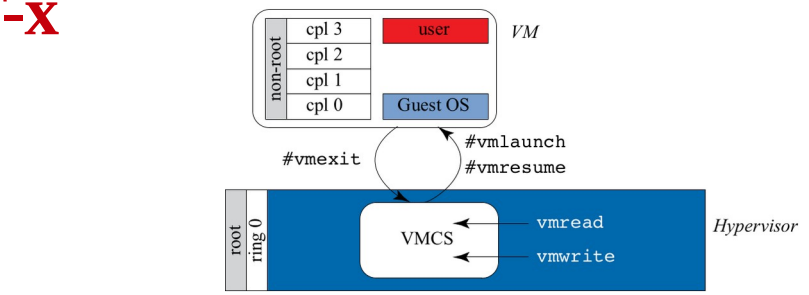
- KVM may need to:
  - Emulate the semantics of that instruction or inject an event to VM.

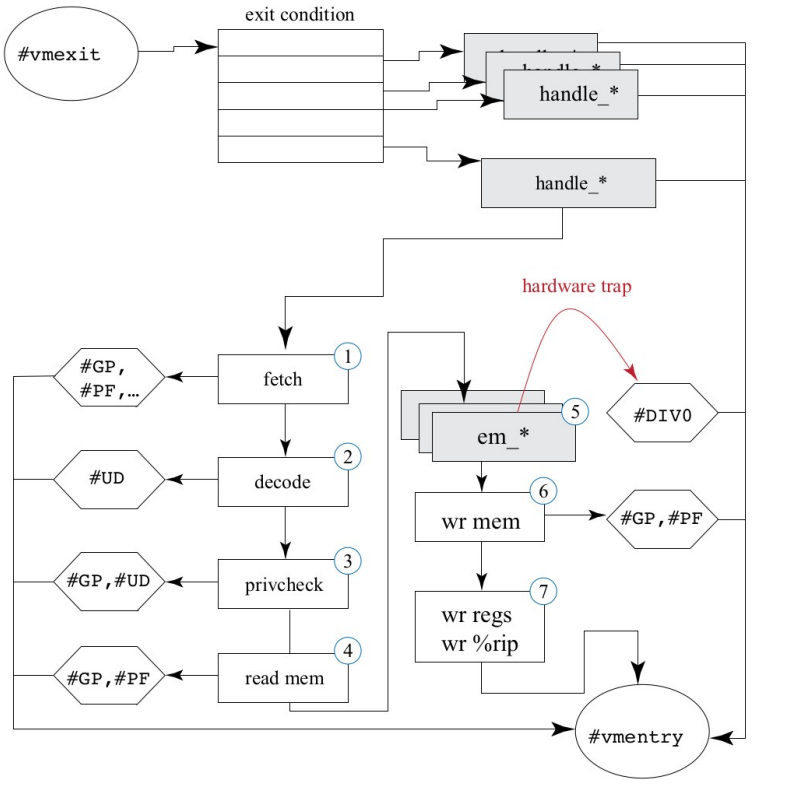
```
int emulator_task_switch(struct x86_emulate_ctxt *ctxt,
                         u16 tss_selector, int idt_index, int reason,
                         bool has_error_code, u32 error_code)
{
        int rc;

        invalidate_registers(ctxt);
        ctxt->_eip = ctxt->eip;
        ctxt->dst.type = OP_NONE;

        rc = emulator_do_task_switch(ctxt, tss_selector, idt_index, reason,
                                     has_error_code, error_code);

        if (rc == X86EMUL_CONTINUE) {
                ctxt->eip = ctxt->_eip;
                writeback_registers(ctxt);
        }

        return (rc == X86EMUL_UNHANDLEABLE) ? EMULATION_FAILED : EMULATION_OK;
}
```

arch/x86/kvm/emulate.c

arch/x86/kvm/vmx/vmx.c

```
static struct kvm_x86_ops vmx_x86_ops __initdata = {
    .name = "kvm_intel",
    ...
    .get_msr = vmx_get_msr,
    .set_msr = vmx_set_msr,

    .inject_exception = vmx_inject_exception,
    ...
}
```





x86-64: CPU Virtualization with VT-x

KVM Kernel Module *(kvm.ko)*

- KVM may need to:
  - Emulate the semantics of that instruction.
    After **vmexit**, kvm needs to understand the instruction that caused it.

    1) **Fetch** the instruction from guest virtual memory.

    2) **Decode** the instruction, extracting its operator and operands.

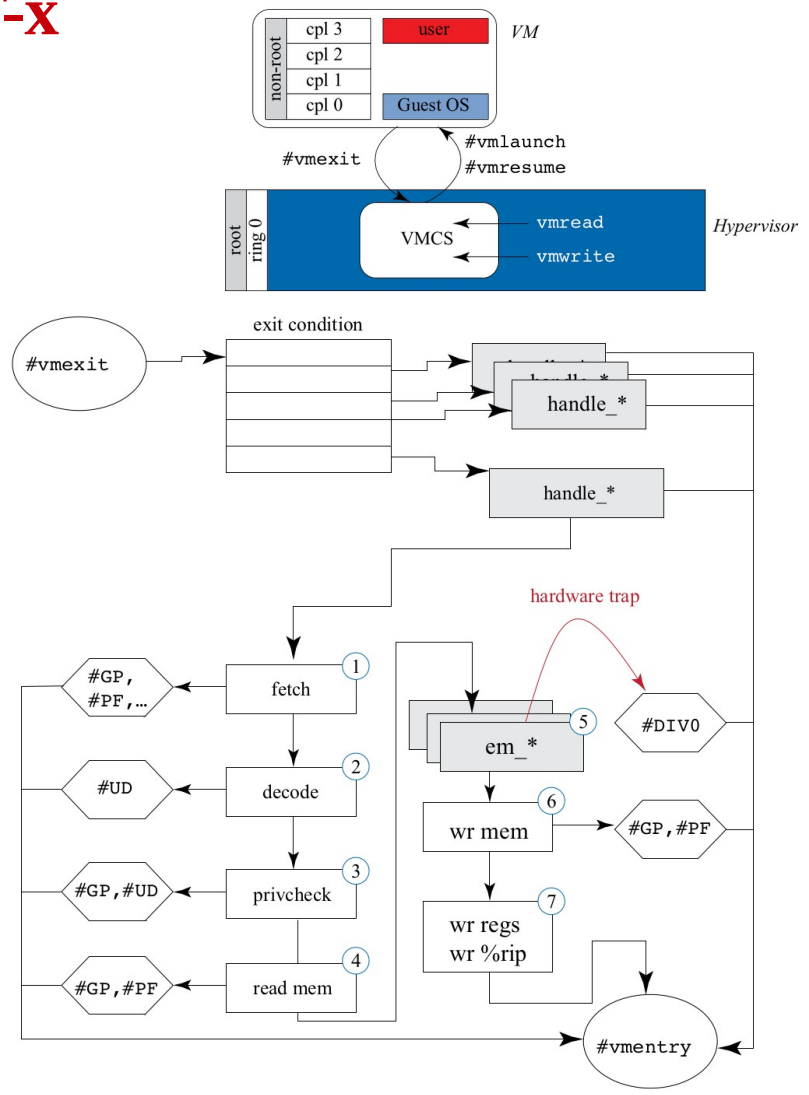
    3) **Verify** whether the instruction can execute given the current state of the virtual CPU, e.g., privileged instructions can only execute if the virtual CPU is at cpl0.

    4) **Read** any memory read-operands from memory.

    5) **Emulate** the decoded instruction.

    6) **Write** any memory write-operands back to the **guest** virtual machine

    7) **Update** guest registers and the instruction pointer as needed.

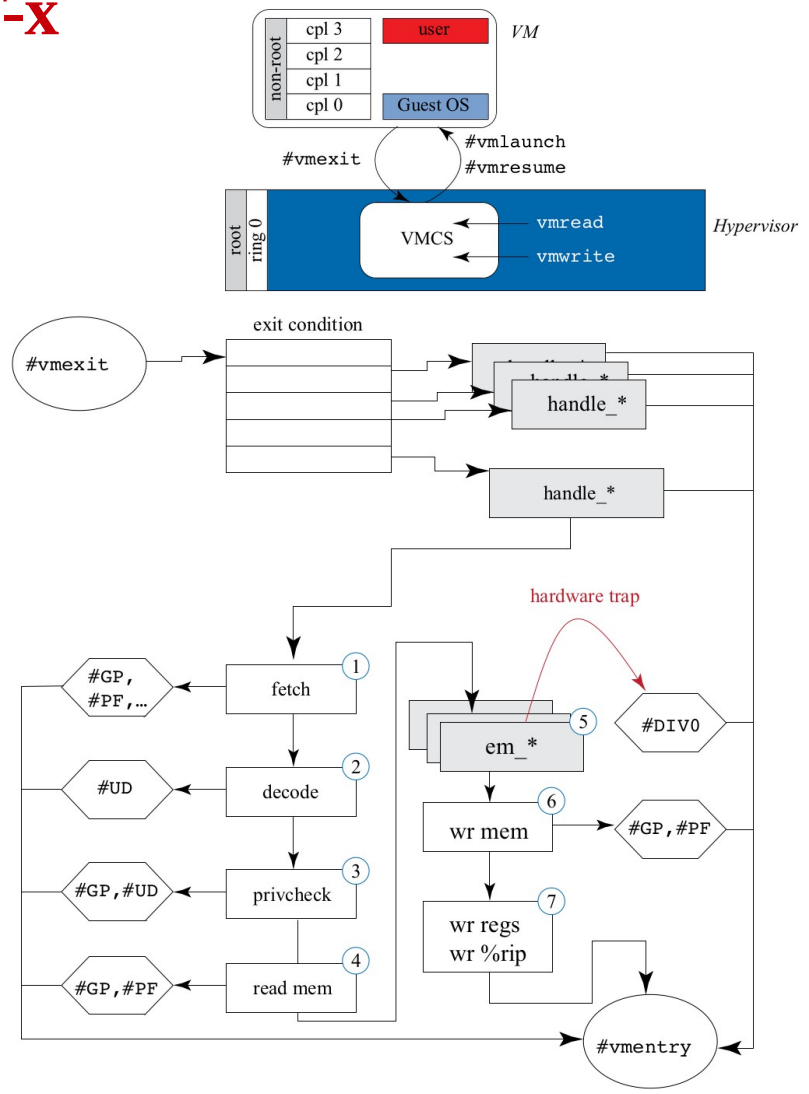


x86-64: CPU Virtualization with VT-x

KVM Kernel Module *(kvm.ko)*

- Every previous stage may contain a possible exception case that lead the KVM should inject **a fault in the guest virtual CPU**.
  GP : General Protection Fault.
  PF : Page Fault.
  UD : Undefined Instruction.

- In some rare cases, a fault in the actual hardware may happen such as when **dividing by zero** (*hardware trap*).



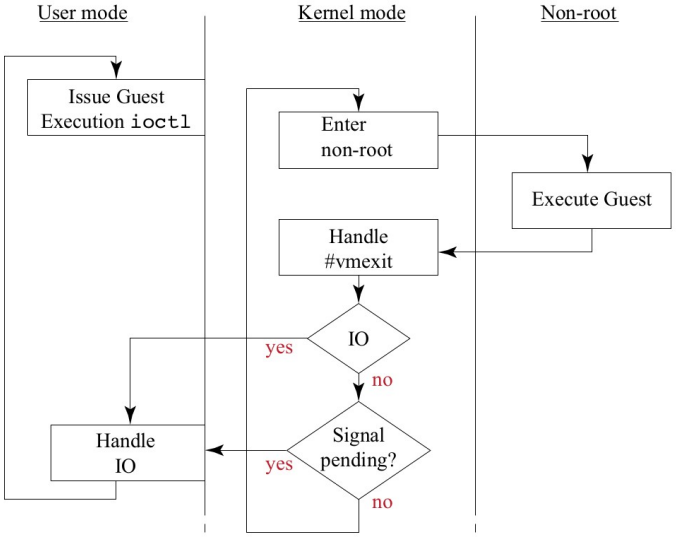
x86-64: CPU Virtualization with VT-x

Role Of The Host (QEMU)

- Generic Execution Loop
  - Generally **QEMU** is responsible for the userspace management.
  - Userspace triggers Kvm module ***/dev/kvm*** through **ioctl**.
  - KVM enters **non-root** mode then executes the **guest** code until either

    1) **Guest** initiates **I/O** using an I/O instruction or memory-mapped I/O.

    2) **Host** receives an external I/O or timer interrupt.
  - QEMU then emulates the initiated I/O (if required).
  - In case of external I/O, Qemu may simply return back to the KVM by using another **ioctl**.



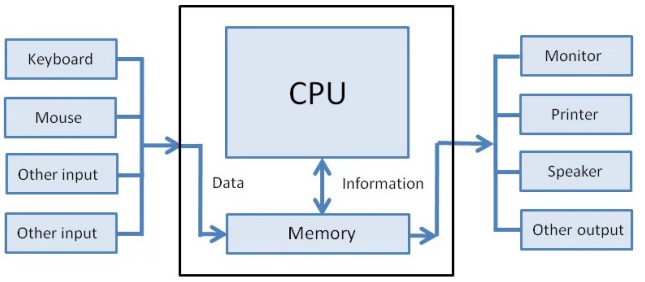
Simplified host-kvm

```
kvm_fd = open("/dev/kvm", O_RDWR);
vm_fd = ioctl(kvm_fd, KVM_CREATE_VM, 0);
mmap(vm_img_sz);
vcpu_fd = ioctl(vm_fd, KVM_CREATE_VCPU, 0);
mmap(sz_for_each_vcpu);
for (;;) {
  ioctl(vcpu_fd, KVM_RUN, 0);
  switch (run->exit_reason) {
    case KVM_EXIT_IO:
      /* Emulate IO */
      break;
    case KVM_EXIT_SHUTDOWN:
    /* Exit */
  }
}
```

Virtualization Components

Basic Virtualized Components

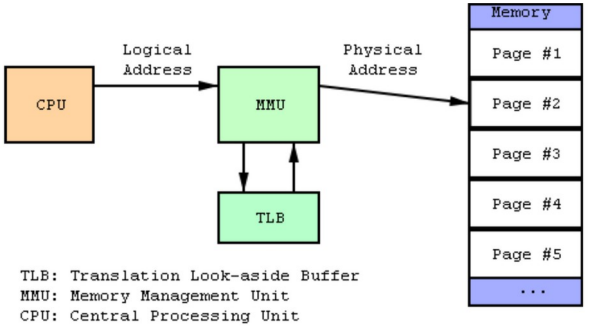
- CPUs
- Memories & MMU
- Input/Output (IO) devices

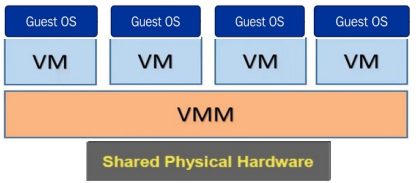


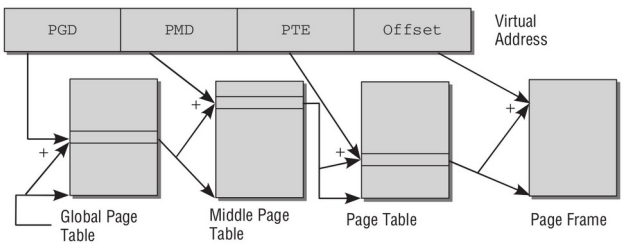
Memory Virtualization

What's the problem?

- We used to see operating systems like Linux uses ***MMU*** and ***virtual addresses*** and manages all the resources at normal **Host** machine.
- Page tables are used for address translation (VA → PA).
- Now we have different **VMs** running **guest** OSs and **VMM/Host** that have to virtualize physical memory**.**
- This creates a two-dimensional problem
  - Guest operating system defines mapping between virtual memory and **guest-physical** memory (**GVA → GPA**).
  - Hypervisor then independently defines mappings between **guest-physical** memory and **host-physical** memory (**GPA → HPA**).
- *Which page table should MMU use?*





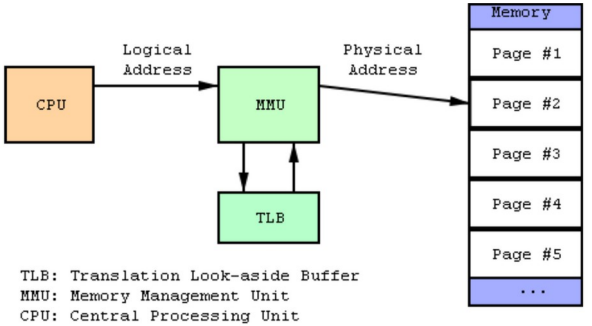


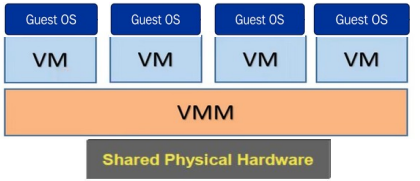
Memory Virtualization

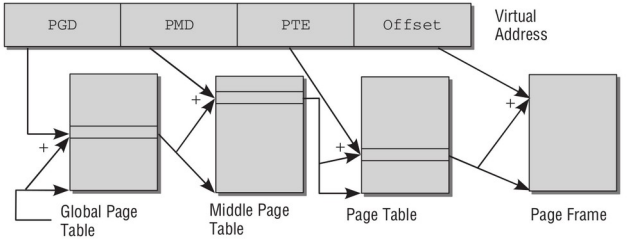
Memory Virtualization Techniques

- Shadow paging
  In the absence of any architecture support in the MMU, hypervisors rely on ***Shadow Paging*** to virtualize memory.

- Extended page tables (EPT)
  Implemented by HW (MMU extension).



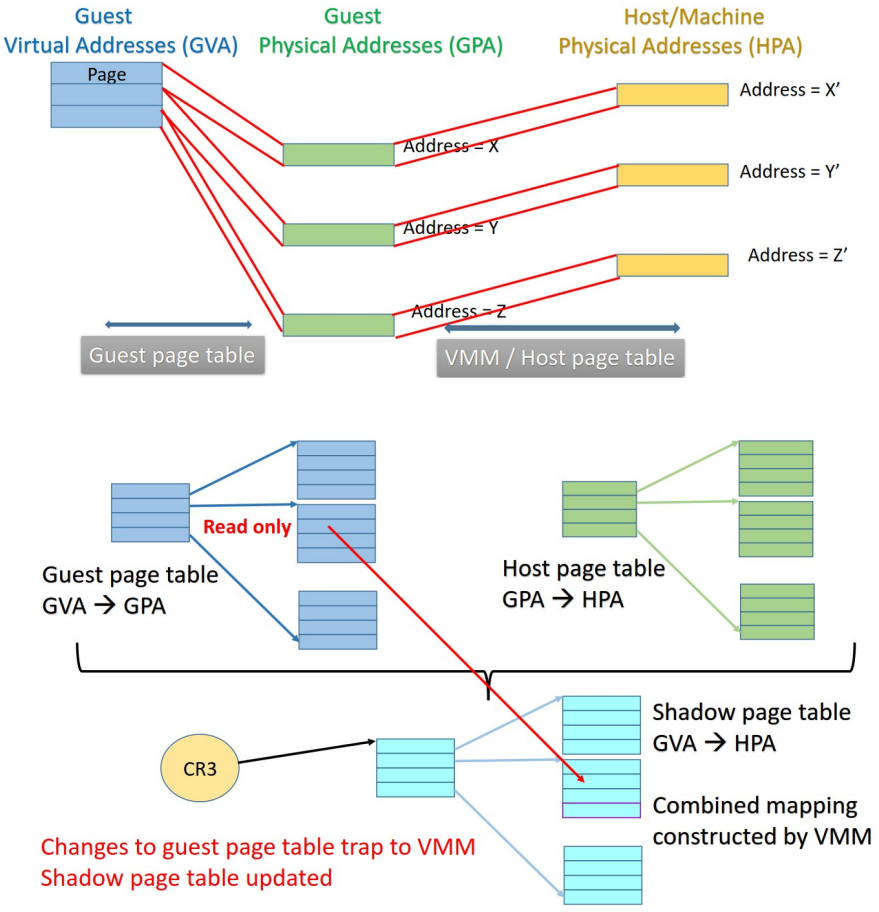




Memory Virtualization

Shadow Paging (SW Implementation)

- VMM shall create a combined mapping **GVA → HPA** and MMU is given a pointer to this page table.
- VMM tracks changes to guest page table and updates ***shadow page table***.
- Guest writes to **CR3** in x86 for example, privileged operation **traps** to VMM.
- VMM marks the guest page table pages as **read-only**.
- VMM constructs shadow page table, sets CR3 to it.
- ***Shadow page table*** can be built on demand, starts with empty page table, and then add entries on **page faults**.
- Guest changes page table, traps to VMM, shadow entry updated.



Memory Virtualization

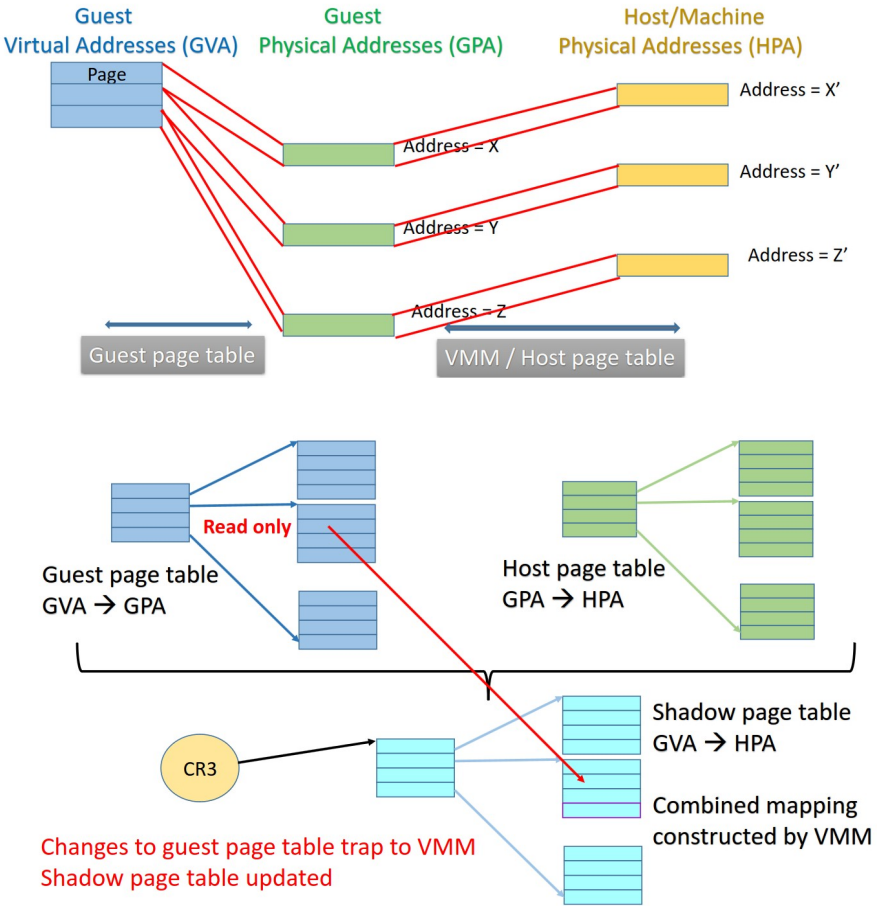
Shadow Paging (SW Implementation)

- *What about shadow page tables? How many in memory?. It's a design choice!*

  1) VMM can maintain multiple shadow page tables of active processes (overhead to track changes to all page table pages)

  2) VMM can discard old shadow page table on context switch, and rebuild it later (overhead during context switch).

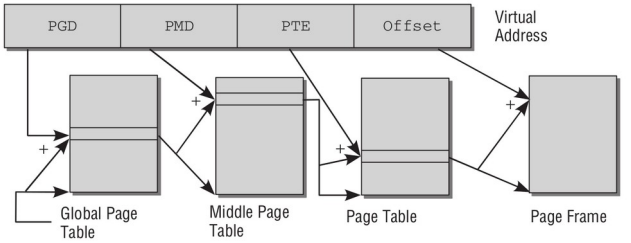
- For more details, check *Virtualization and Cloud Computing*

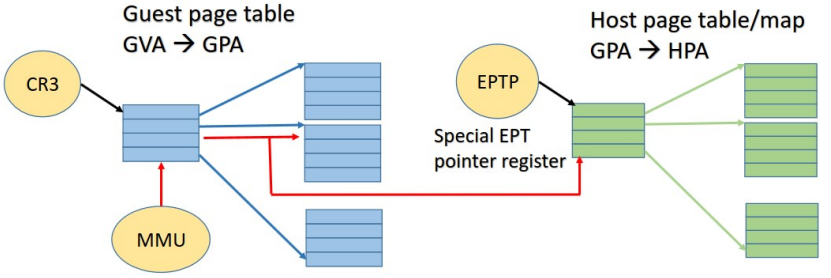


Memory Virtualization

Extended Paging (MMU Support EPT x86-64)

- **Extended page tables**, also known as **nested page tables**, eliminates the need for software-based **shadow paging**.
- MMU takes pointers to two separate page tables, such that address translation walks both page tables.
- As we know the page table has different **levels** (2, 3, 4, …).
- MMU starts walking guest page table resolves each level using **GVA** and get **GPA**.
- Use **GPA**, walk host page table to find **HPA**.
- Lets consider gust page table & host page table have 3 levels.

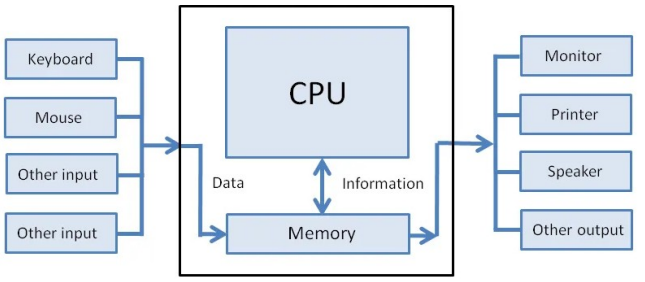




Virtualization Components

Basic Virtualized Components

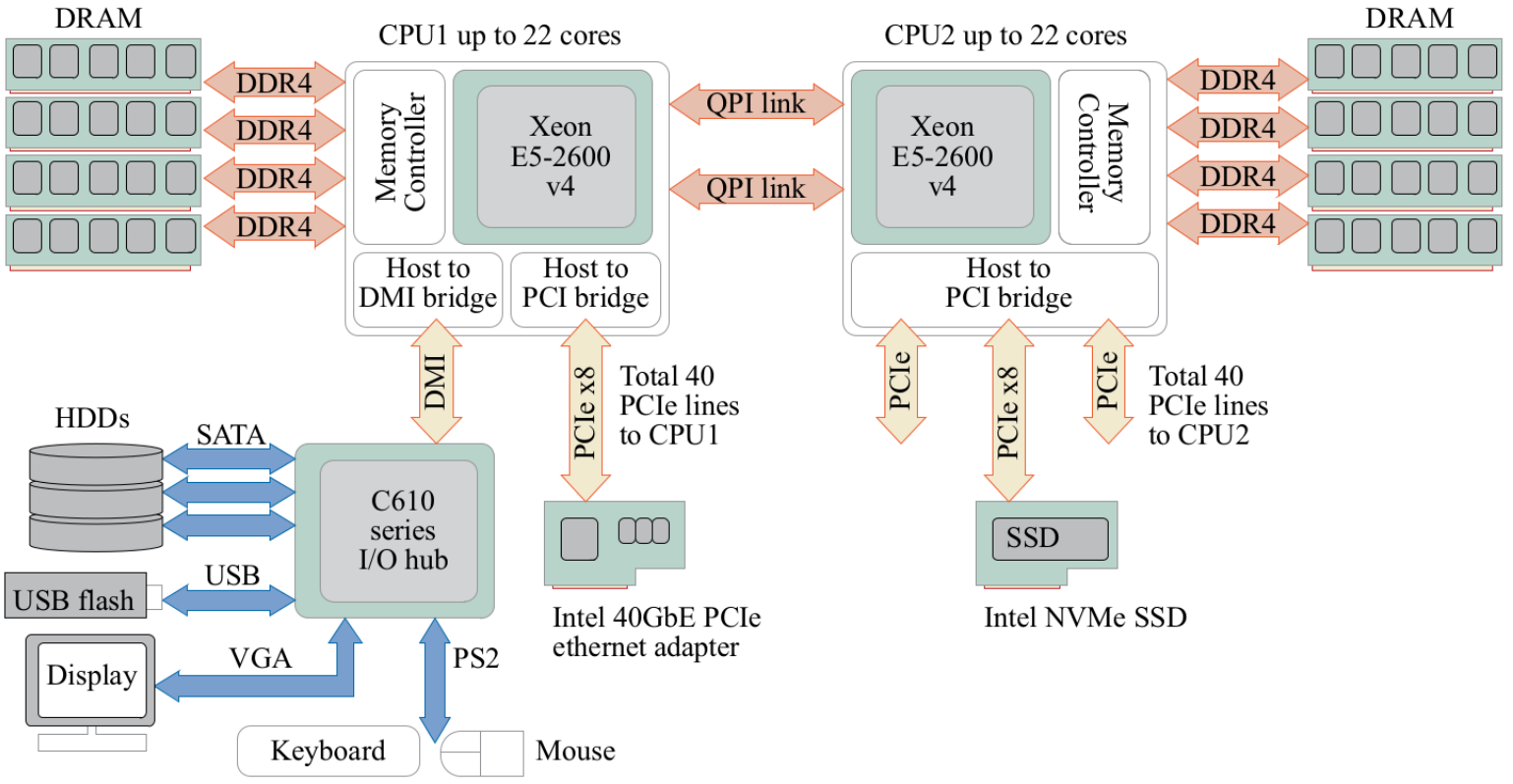
- CPUs
- Memories & MMU
- Input/Output (IO) devices



x86-64: I/O Virtualization

Physical I/O

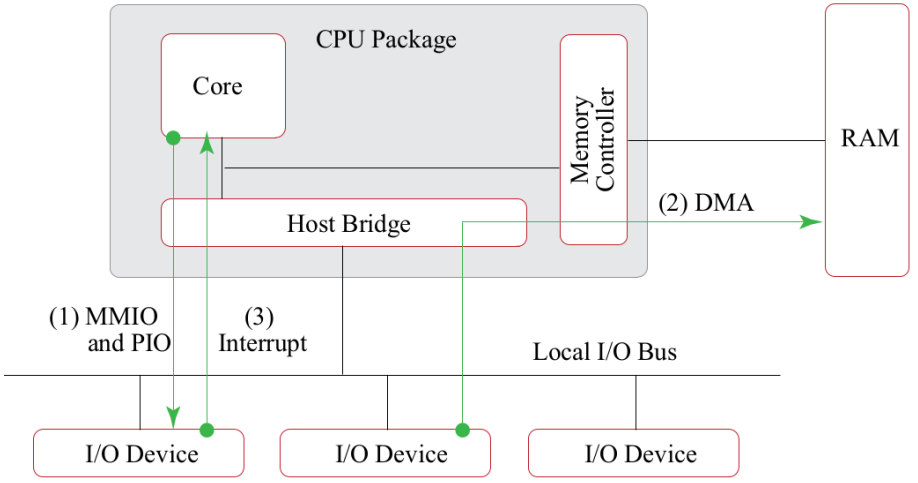
- High-level view of generic internal organization of Intel servers.



x86-64: I/O Virtualization

Physical I/O

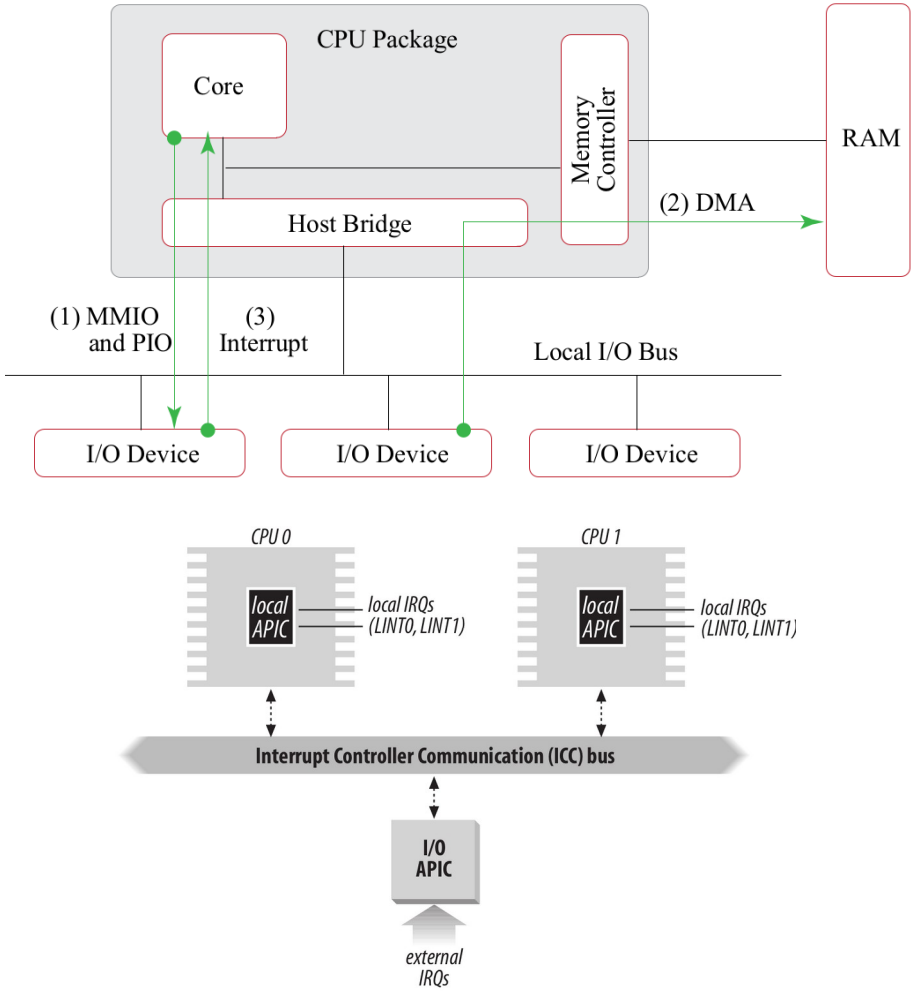
- Generally I/O devices interact with the CPU and the memory in three ways.
  - **MMIO** and **PIO** allow CPU cores to communicate with I/O devices.
  - **DMA** allows I/O devices to access the memory.
  - **Interrupts** allow I/O devices to communicate with CPU cores.
- **PIO** (Port-mapped I/O) and **MMIO** (memory-mapped I/O)
  - ***PIO*** devices communications via special instructions, **OUT** and **IN** x86 to write/read from I/O device. i.e, Ports **0x0060-0x0064** are used to read from and send commands to keyboards and mice with PS/2 connectors.
  - ***MMIO*** device registers are associated with physical memory addresses, so, use regular **load** and **store** x86 operations through the memory bus.



x86-64: I/O Virtualization

Physical I/O

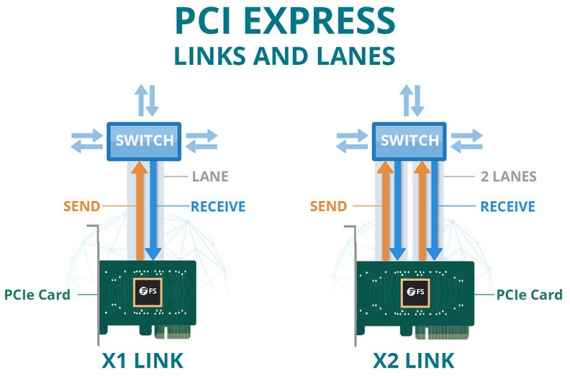
- **DMA** (Direct Memory Access)
  - CPU can initiate a DMA operation, to transfer data buffers from memory to I/O devices, and asking the I/O device for asynchronous notification when the operation completes.
- **Interrupts**
  - I/O devices trigger asynchronous event notifications directed at the CPU cores by issuing interrupts.
  - When an interrupt fires, the hardware invokes the associated routine on the target core, using the interrupt vector to index the **IDT** (Interrupt Descriptor Table).
  - **LAPIC** (Local Advanced Programmable Interrupt Controller), It's per core controller responsible for notifying the CPU with interrupts, and Disable/Enable interrupts notifications.
  - I/O APIC acts like a router with respect to the local APIC's.

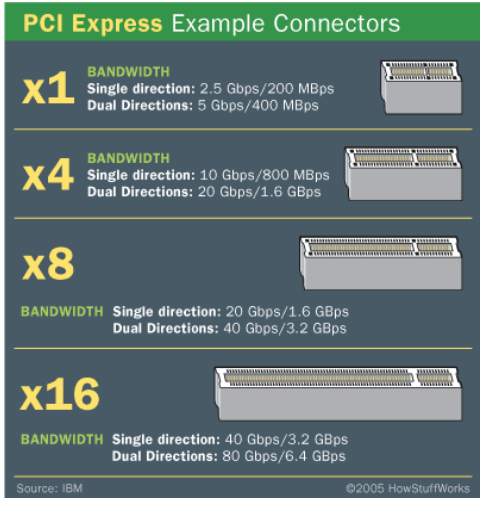


x86-64: I/O Virtualization

Physical I/O

- **PCIe**
  Peripheral Component Interconnect Express
  Terminologies
  - PCIe lanes
    A PCIe lane is a data channel.
    Lane refers to a set of **differential signal pairs** (transmit and receive).
  - PCIe slots
    PCIe x1, PCIe x4, PCIe x8, PCIe x16
    determine the number of **data lanes** available for communication.
    Ex.
    PCIe x1 → Wi-Fi adapters
    PCIe x4 → Sound cards
    PCIe x8 → SSD disks
    PCIe x16 → GPUs





x86-64: I/O Virtualization

Physical I/O

- **PCIe**
  Peripheral Component Interconnect Express
  Terminologies
  - PCIe generations

PCIe 1.0 / 1.1:

- Released: 2003 (1.0), 2005 (1.1)
- Max Bandwidth per Lane: 250 MB/s (1.0), 250 MB/s (1.1)
- Max Bandwidth x16 Slot: 4 GB/s (1.0), 4 GB/s (1.1)
- Notable for being the first generation of PCIe, introducing higher speeds than the previous PCI interface.

PCIe 2.0:

- Released: 2007
- Max Bandwidth per Lane: 500 MB/s
- Max Bandwidth x16 Slot: 8 GB/s
- Doubled the data rate per lane compared to PCIe 1.0/1.1, resulting in improved overall bandwidth.

PCIe 3.0:

- Released: 2010
- Max Bandwidth per Lane: 1 GB/s
- Max Bandwidth x16 Slot: 16 GB/s
- Introduced further doubling of data rate per lane compared to PCIe 2.0, providing higher performance for demanding applications.

PCIe 4.0:

- Released: 2017
- Max Bandwidth per Lane: 2 GB/s
- Max Bandwidth x16 Slot: 32 GB/s
- Doubled the data rate per lane again compared to PCIe 3.0, offering even more bandwidth for high-performance graphics cards and storage solutions.

PCIe 5.0:

- Released: 2019
- Max Bandwidth per Lane: 4 GB/s
- Max Bandwidth x16 Slot: 64 GB/s
- Doubled the data rate per lane compared to PCIe 4.0, reaching impressive bandwidth levels suitable for emerging data-intensive applications.

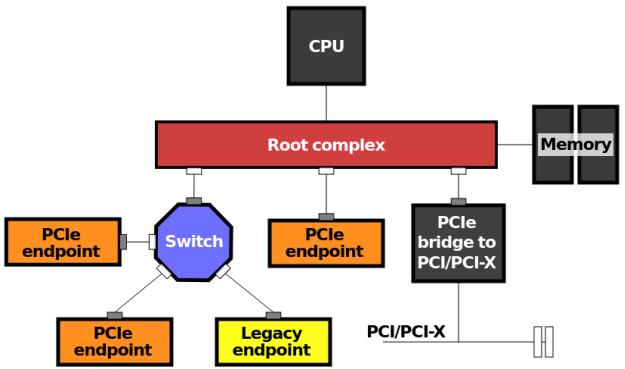
PCIe 6.0:

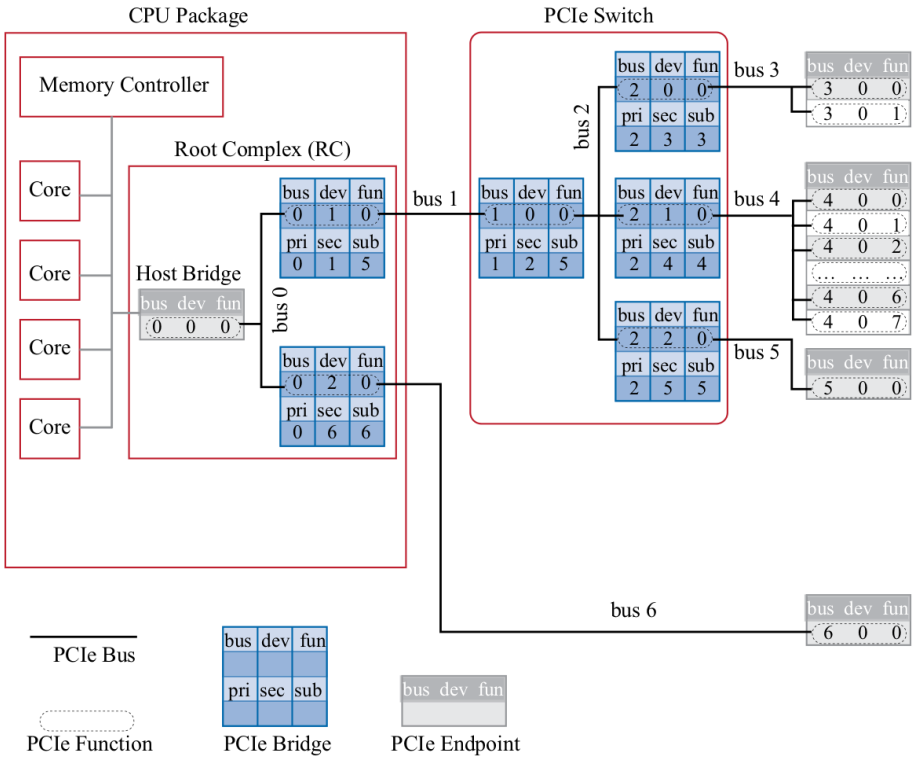
- Released: 2022
- Max Bandwidth per Lane: 8 GB/s
- Max Bandwidth x16 Slot: 128 GB/s
- Doubled the data rate per lane compared to PCIe 5.0, pushing the boundaries of high-speed data transfer for future systems.

x86-64: I/O Virtualization

Physical I/O

- **PCIe** general Fan-out topology





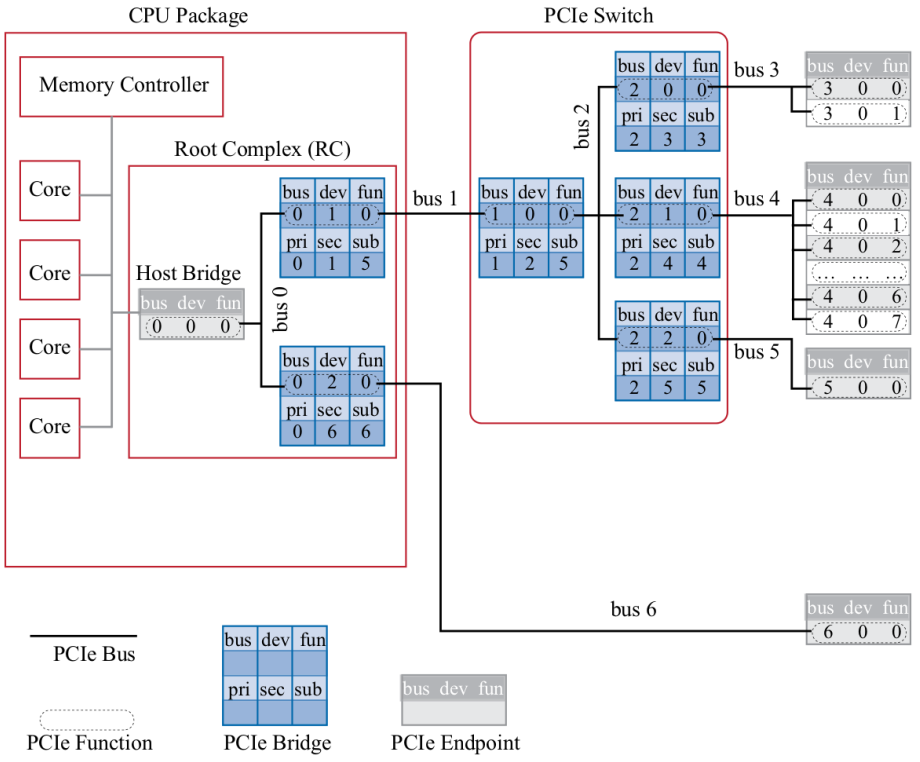
x86-64: I/O Virtualization

Physical I/O

- **PCIe Node (Device) Enumeration**
  - Every PCIe node in the PCIe graph is uniquely identified by the form of ***bus:device.function*** (BDF), consist of ***8, 5, and 3 bits***.
- **PCIe Edge (Bus) Enumeration**
  - The maximal number of buses in the system is 256 (as bus has 8 bits).
  - Every PCIe bridge G to be associated with three buses: ***primary***, ***secondary***, and ***subordinate***.

    ex. The buses under bridge 1:0.0 range from 2 (***secondary***) to 5 (***subordinate***).

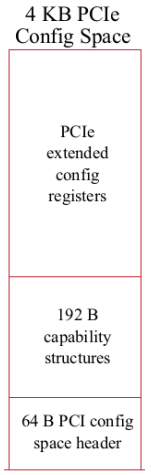
    ex. The buses under bridge 2:5.5 range from 5 (***secondary***) to 5 (***subordinate***) as well, only one bus associated.



x86-64: I/O Virtualization

Physical I/O

- **PCIe Configuration Space**
  - Each PCIe node has 4 KB configuration space identified by BDF ***bus:device.function*** consists of 3 parts
    1) 64 B : PCI conifg header.
    2) 192 B : Extended capability structures.
    3) Rest of 4KB : Extended config registers.
  - Capabilities :
    Each type has an ID (Power Cap, Graphics, MSI Cap,…).



Capability IDs

ID	Capability
00h	Null Capability – This capability contains no registers other than those described below. It may be present in any Function. Functions may contain multiple instances of this capability. The Null Capability is 16 bits and contains an 8-bit Capability ID followed by an 8-bit Next Capability Pointer.
01h	PCI Power Management Interface – This Capability structure provides a standard interface to control power management features in a device Function. It is fully documented in the *PCI Bus Power Management Interface Specification*.
02h	AGP – This Capability structure identifies a controller that is capable of using Accelerated Graphics Port features. Full documentation can be found in the *Accelerated Graphics Port Interface Specification*.
03h	VPD – This Capability structure identifies a device Function that supports Vital Product Data. Full documentation of this feature can be found in the *PCI Local Bus Specification*.
04h	Slot Identification – This Capability structure identifies a bridge that provides external expansion capabilities. Full documentation of this feature can be found in the *PCI-to-PCI Bridge Architecture Specification*.
05h	Message Signaled Interrupts – This Capability structure identifies a device Function that can do message signaled interrupt delivery. Full documentation of this feature can be found in the *PCI Local Bus Specification*.
06h	CompactPCI Hot Swap – This Capability structure provides a standard interface to control and sense status within a device that supports Hot Swap insertion and extraction in a CompactPCI system. This Capability is documented in the *CompactPCI Hot Swap Specification PICMG 2.1, R1.0* available at http://www.picmg.org.
07h	PCI-X – Refer to the *PCI-X Protocol Addendum to the PCI Local Bus Specification* for details.

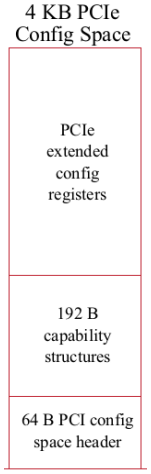
31 – 16	15 – 0	
Device ID	Vendor ID	000h
Status	Command	004h
Class Code	Rev ID	008h
BIST / Header / Lat Timer	Cache Ln	00Ch
Base Address Register 0		010h
Base Address Register 1		014h
Base Address Register 2		018h
Base Address Register 3		01Ch
Base Address Register 4		020h
Base Address Register 5		024h
Cardbus CIS Pointer		028h
Subsystem ID	Subsystem Vendor ID	02Ch
Expansion ROM Base Address		030h
Reserved	CapPtr	034h
Reserved		038h
Max Lat / Min Gnt / Intr Pin	Intr Line	03Ch
PM Capability	NxtCap / PM Cap	040h
Data / BSE / PMCSR		044h
MSI Control	NxtCap / MSI Cap	048h
Message Address (Lower)		04Ch
Message Address (Upper)		050h
Reserved	Message Data	054h
PE Capability	NxtCap / PE Cap	058h
PCI Express Device Capabilities		05Ch
Device Status	Device Control	060h
PCI Express Link Capabilities		064h
Link Status	Link Control	068h
Reserved Legacy Configuration Space (Returns 0x00000000)		06Ch–0FFh
Next Cap / Capability Version	PCI Express Extended Capability - DSN	100h
PCI Express Device Serial Number (1st)		104h
PCI Express Device Serial Number (2nd)		108h
Reserved Extended Configuration Space (Returns Completion with 0x00000000)		10Ch–FFFh

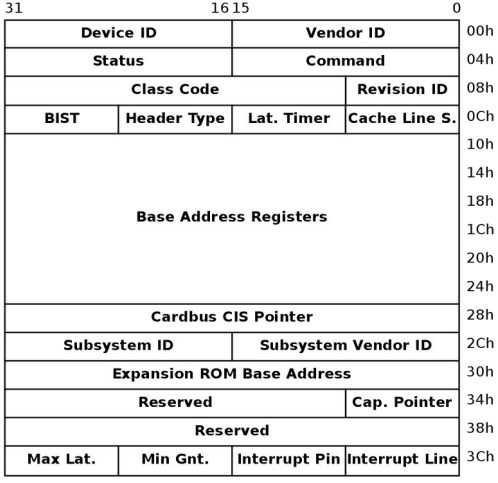
Optional Returns 0 if not implemented

x86-64: I/O Virtualization

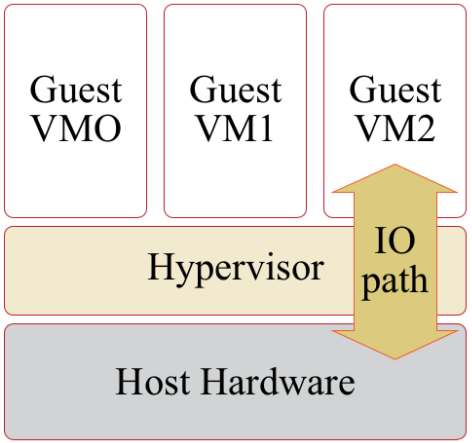
Physical I/O

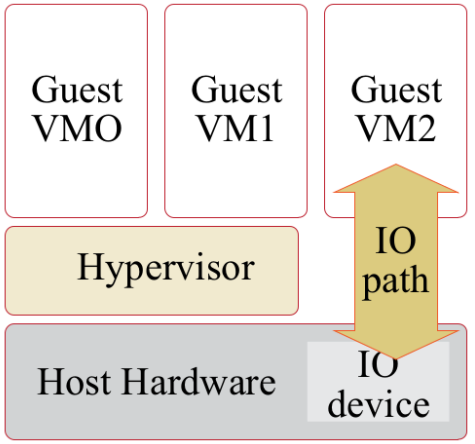
- **PCIe Configuration Space**
  - BARs (Base address Registers) : address registers to read, write data/configurations to the device.
    Normal devices : Up to 6 BARs.
    Bridges : 2 BARs
- **PCIe MSI**
  - Message Signaled Interrupts (MSI) allow a device to send a PCIe packet whose destination is a LAPIC of a core.
  - It's similar to a DMA operation, but instead of targeting the memory, it targets a LAPIC.
- *PCIe has many other details, hope talk about it later in details in Linux drivers sessions, for now it's sufficient.*

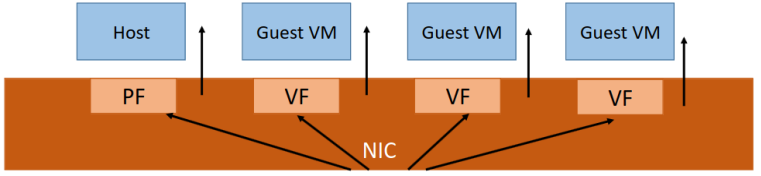




x86-64: I/O Virtualization



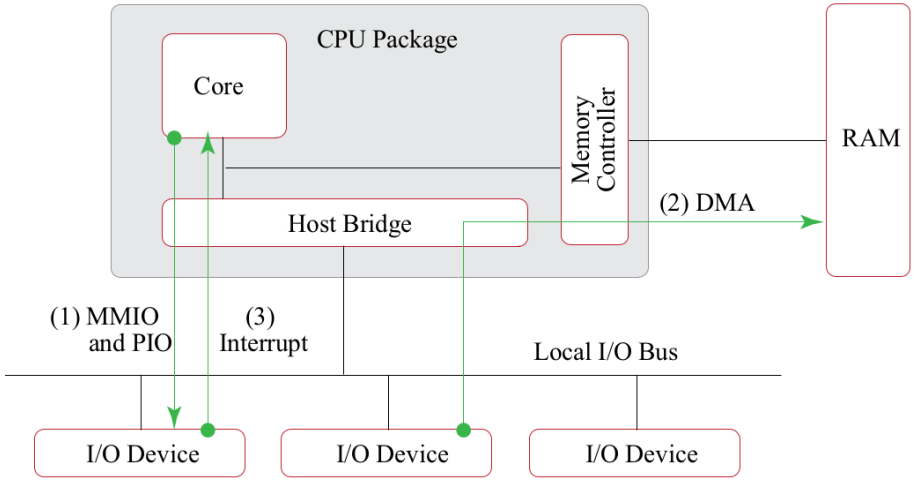




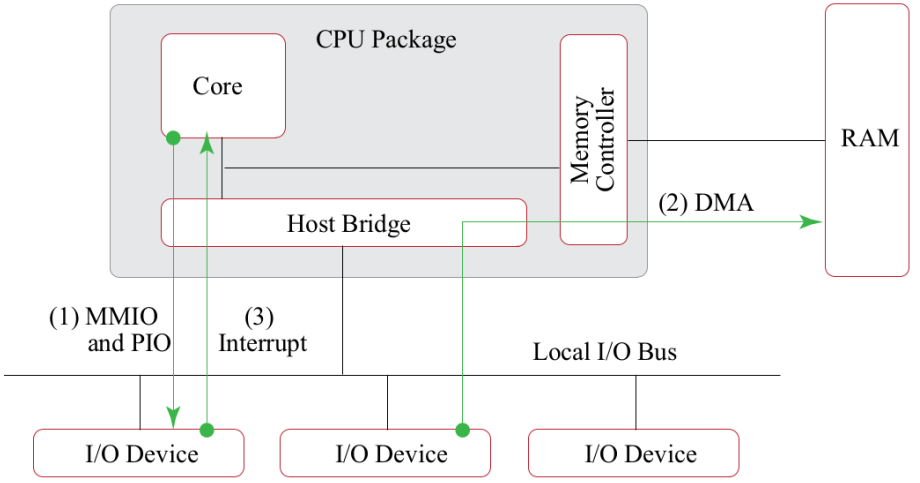
x86-64: I/O Virtualization

Virtual I/O without HW Support

- **I/O Emulation (Full Virtualization)**
  Let's remember,
  - OS talks to I/O devices by using MMIO and PIO operations.
  - PIO : **IN, OUT** x86 to read/write from I/O device.
  - MMIO : **load** and **store** x86 operations through the memory bus.
  - The I/O devices respond by triggering interrupts and by reading/writing data to/from memory via DMAs.

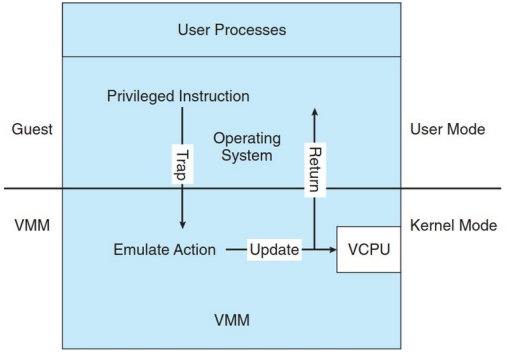


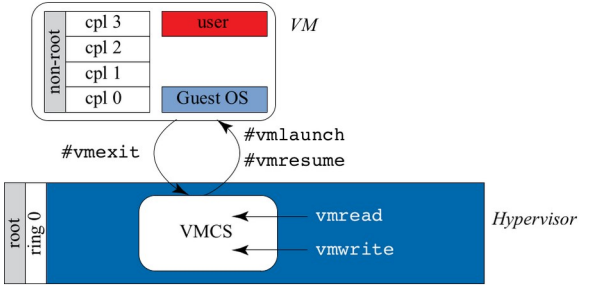
x86-64: I/O Virtualization



Virtual I/O without HW Support

- **I/O Emulation (Full Virtualization)**
  - *But, how hypervisor can emulate,*
    1) DMA Operation
    2) Guest's MMIOs
    3) Guest's PIOs
    - DMA: Emulating DMAs to/from guest memory by hypervisor can be done easily by reading from and writing to this memory.
    - Guest's MMIOs: Regular loads/stores from/to guest memory pages, so the hypervisor can arrange for these memory accesses to **trap** by mapping the **pages** as **reserved/non-present** (both **loads** and **stores** trigger **exits**) or as **read-only** (only **stores** trigger exits).
    - Guest's PIOs: Already **privileged** instructions (**IN, OUT**), and the hypervisor can configure the guest's **VMCS** to **trap** upon them.



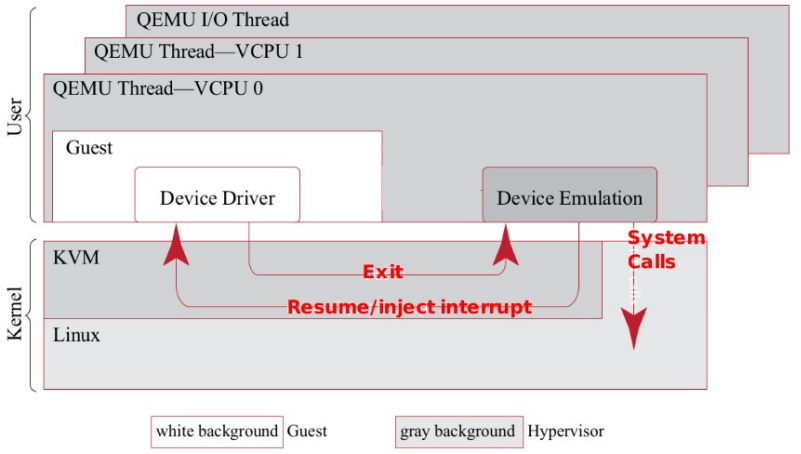


x86-64: I/O Virtualization

Virtual I/O without HW Support

- **I/O Emulation (Full Virtualization)**
  - KVM/QEMU Interaction Sequence
    - Every hosted virtual machine is encapsulated within a QEMU process.
    - A QEMU **process** represents the **VCPUs** (virtual cores) of its VM using different threads.
    - For every **virtual device**, QEMU spawns another thread, denoted as "**I/O thread**".
    - When the **guest** VM device driver issues **MMIOs/PIOs** to drive the device, these operations read/write-protected memory locations, triggering **exits** that suspend the VM VCPU context and invoke **KVM**.
    - KVM relays the **events** back to the same VCPU host **thread** context.
    - QEMU's device emulation layer processes these events, using the physical resources of the system, through regular system calls.

```
_start:
mov %ax, 0
loop:
    # out value, port
    out %ax, $0x10
    inc %ax
    jmp loop
```

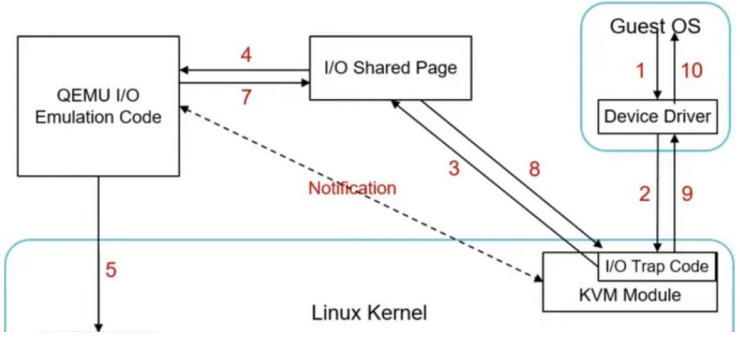


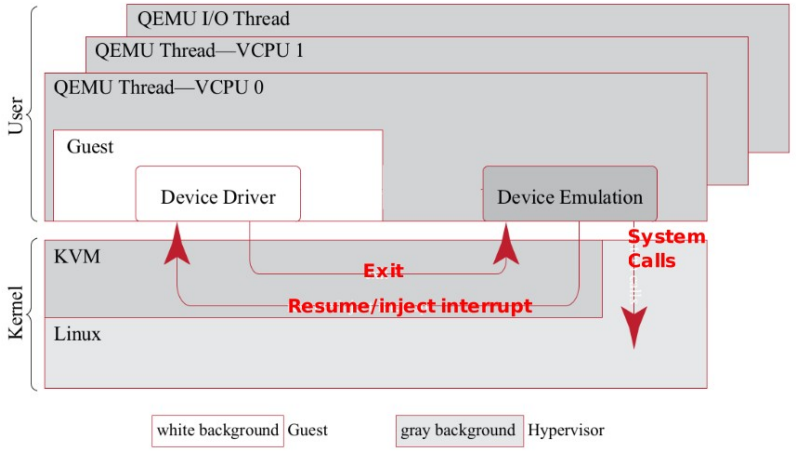

```
static int run_and_emulate(int vcpu_fd, struct kvm_run *run)
{
  int ret = E_OK;
  for (;;) {
    int ret = ioctl(vcpu_fd, KVM_RUN, 0);
    if (ret < 0) {
      perror("KVM_RUN failed\n");
      ret = E_NOT_OK;
      goto err;
    }

    switch (run->exit_reason) {
    case KVM_EXIT_IO:
      char * p = (char *)run;
      printf("Counter loop : %x\n", *(int *)(p + run->io.data_offset));
      sleep(1);
      break;
    case KVM_EXIT_SHUTDOWN:
      ret = E_NOT_OK;
      goto err;
    }
  }

err:
  return ret;
}
```

x86-64: I/O Virtualization

Virtual I/O without HW Support

- **I/O Emulation (Full Virtualization)**
  - KVM/QEMU Interaction Sequence, *Cont'd*
    - The emulation layer emulates the request using the I/O buffers, which are accessible by **shared memory**.
    - It then **resumes** the guest execution context via KVM, possibly **injecting** an **interrupt** to the guest that I/O events occurred.



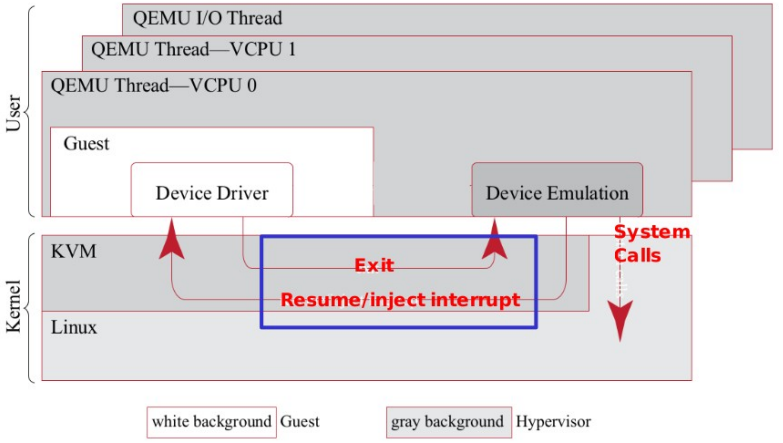


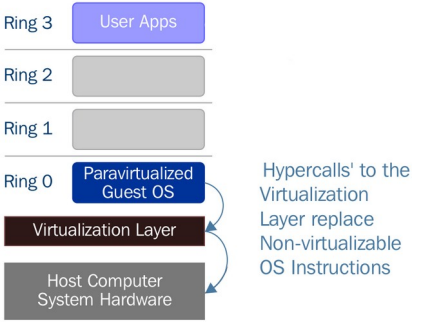

```c
static int run_and_emulate(int vcpu_fd, struct kvm_run *run)
{
  int ret = E_OK;
  for (;;) {
    int ret = ioctl(vcpu_fd, KVM_RUN, 0);
    if (ret < 0) {
      perror("KVM_RUN failed\n");
      ret = E_NOT_OK;
      goto err;
    }

    switch (run->exit_reason) {
    case KVM_EXIT_IO:
      char * p = (char *)run;
      printf("Counter loop : %x\n", *(int *)(p + run->io.data_offset));
      sleep(1);
      break;
    case KVM_EXIT_SHUTDOWN:
      ret = E_NOT_OK;
      goto err;
    }
  }

err:
  return ret;
}
```

x86-64: I/O Virtualization

Virtual I/O without HW Support

- **I/O Para-virtualization**
  - **Para-virtualization** introduced due to performance overheads by I/O Full virtualization.
  - For example, I/O Full virtualization
    Sending/receiving a single Ethernet frame involves multiple register accesses, which translate to multiple **exits** per frame in virtualized setups.
  - This observation underlies **I/O para-virtualization**, whereby guests and hosts agree upon a (virtual) device **specification** to be used for I/O emulation, with the explicit goal of **minimizing overheads**.
  - The guest OS must install a special **device driver** that is compatible with its hypervisor.

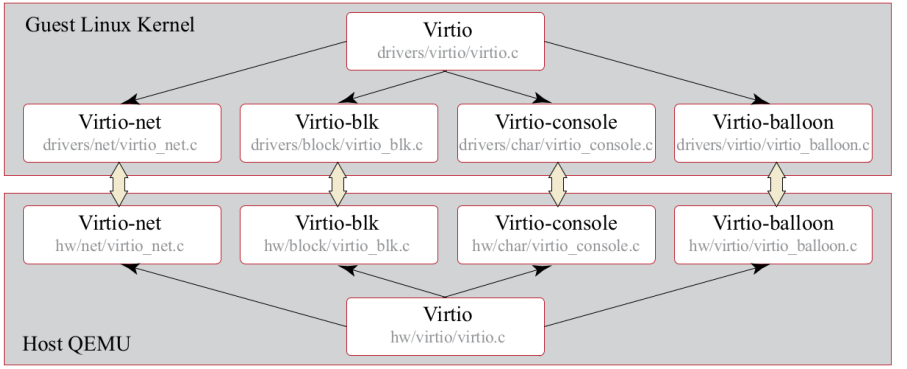


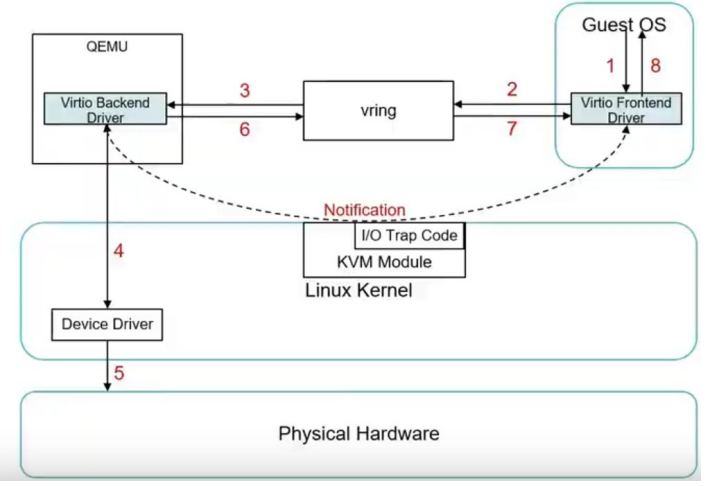


x86-64: I/O Virtualization

Virtual I/O without HW Support

- **I/O Para-virtualization**
  - **Virtio**
    - The framework of para-virtual I/O devices of KVM/QEMU is called **virtio,** offering a common guest-host interface and communication mechanism.
    - **virtio** can be divided into several parts ***Front-end driver***, ***Back-end driver***, ***Transport (or virtqueue)***
    - The central construct of **virtio** is ***virtqueue*** (or ***vring***).
    - In I/O Full virtualization, MMIO/PIO access implicitly triggers an exit each time.
    - In contrast, guests that use ***virtqueues*** generally never trigger exits.





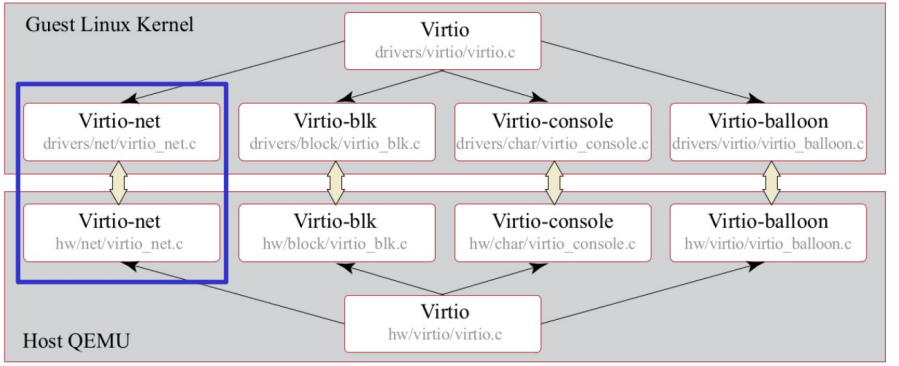
x86-64: I/O Virtualization

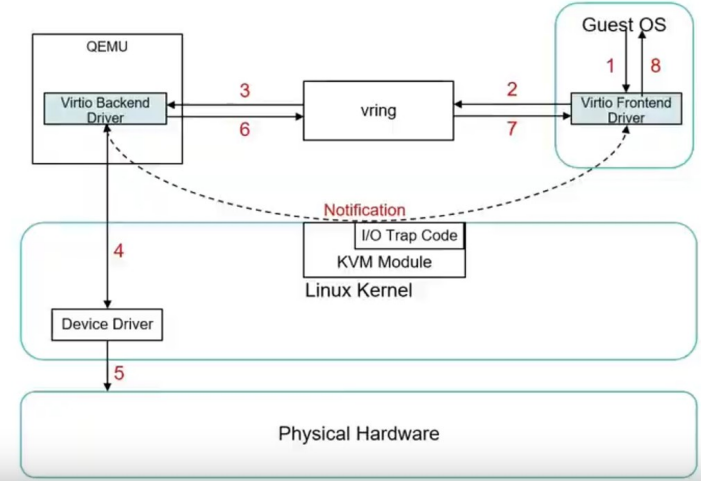
Virtual I/O without HW Support

- **I/O Para-virtualization**
  - **Virtio**
    - ***Virtqueue*** (or ***vring***) has **2 modes** of execution, ***VIRTQ_AVAIL_F_NO_INTERRUPT*** ***VIRTQ_USED_F_NO_NOTIFY,***
    - **NO_INTERRUPT**
      When guest turns on the NO_INTERRUPT mode, it informs the **host** to **refrain** from delivering interrupts associated with the para-virtual device until the **guest** turns off this mode.

      ex. **Virtio-net**
      It uses aggressively the **NO_INTERRUPT** mode in transmission virtqueue Tx, because it does not care when the transmission finishes.

      Only when the virtqueue is nearly full, the guest turn off the NO_INTERRUPT mode, thereby enabling interrupts.





x86-64: I/O Virtualization

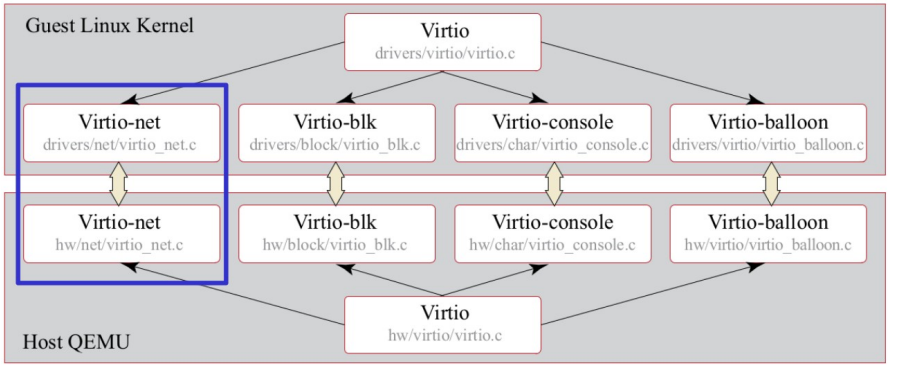
Virtual I/O without HW Support

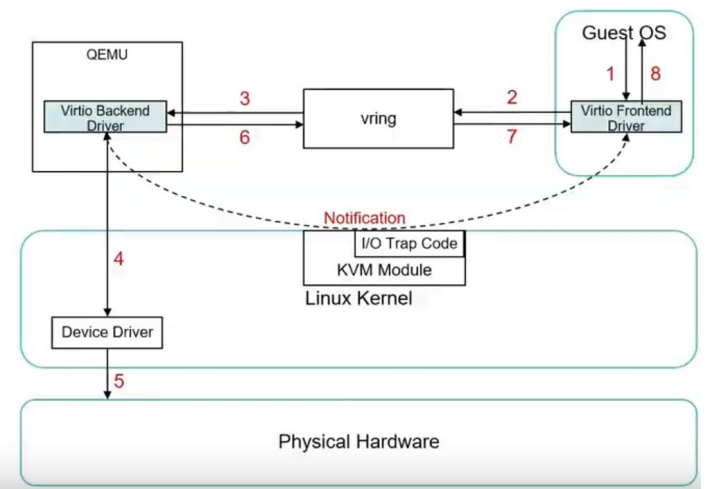
- **I/O Para-virtualization**
  - **Virtio**
    - **NO_NOTIFY**
      When the host turns on the NO_NOTIFY mode, it informs the guest to refrain from kicking it.

      ex. **Virtio-net**
      It uses this execution mode, because TCP traffic tends to be bursty.

      When a burst of frames is sent by the guest, the host only needs the first kick.

      The host needs no additional kicks, Hence, it turns NO_NOTIFY on and processing the **vring** until the end of burst of frames and thereby minimizes virtualization overheads.



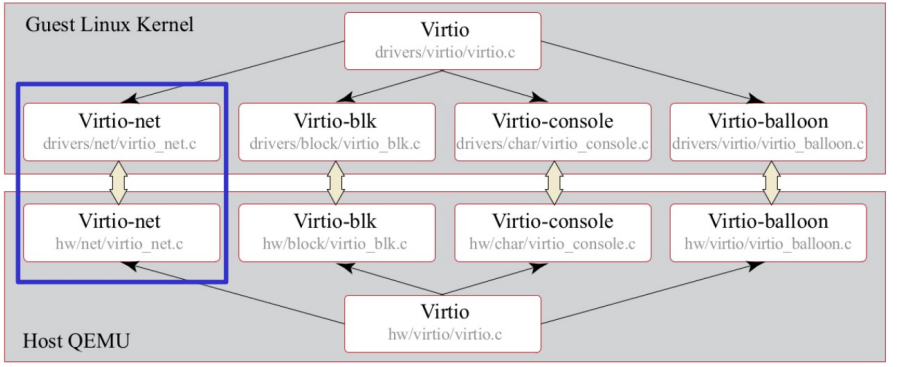


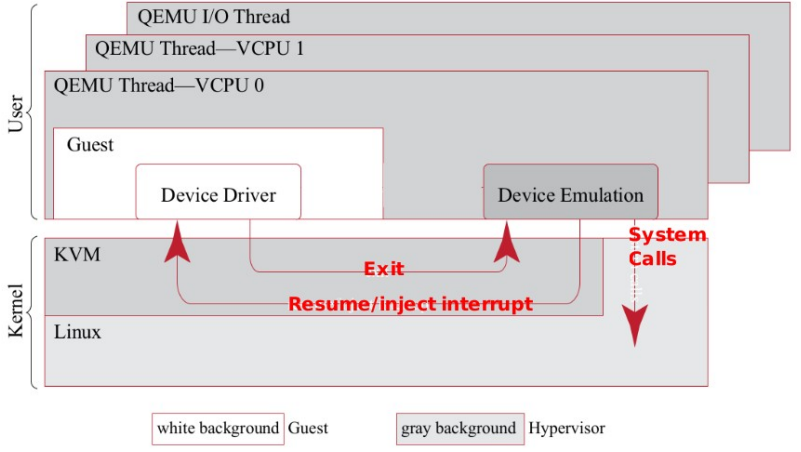
x86-64: I/O Virtualization

Virtual I/O without HW Support

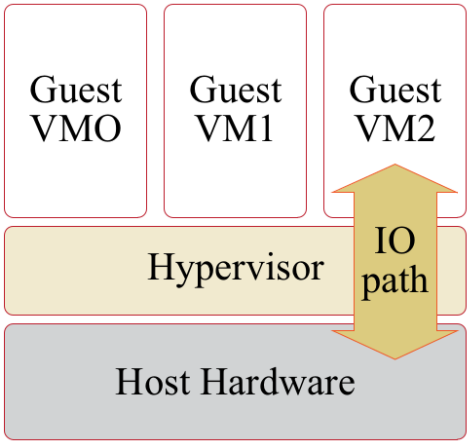
- **I/O Para-virtualization**
  - **Virtio** Performance

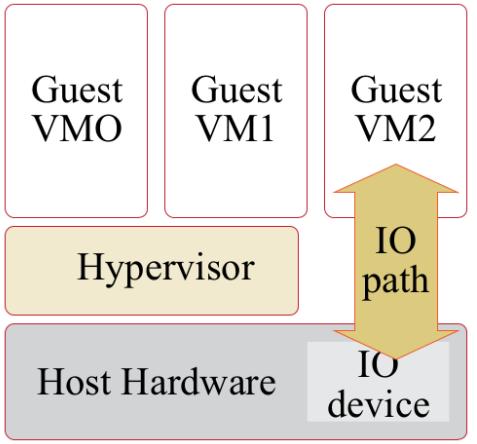
	Metric	e1000	Virtio-net	Ratio
Guest	throughput (Mbps)	239	5,230	22x
	exits per second	33,783	1,126	1/30x
	interrupts per second	3,667	257	1/14x
TCP segments	per exit	1/9	25	225x
	per interrupt	1	118	118x
	per second	3,669	30,252	8x
	avg. size (bytes)	8,168	21,611	3x
	avg. processing time (cycles)	652,443	79,132	1/8x
Ethernet frames	per second	23,804	–	–
	avg. size (bytes)	1,259	–	–

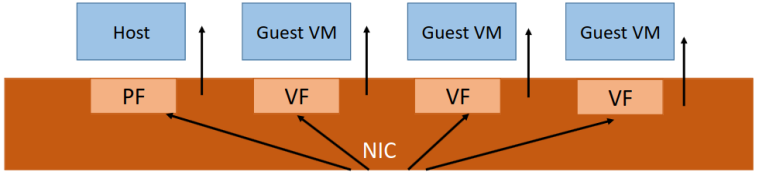




x86-64: I/O Virtualization



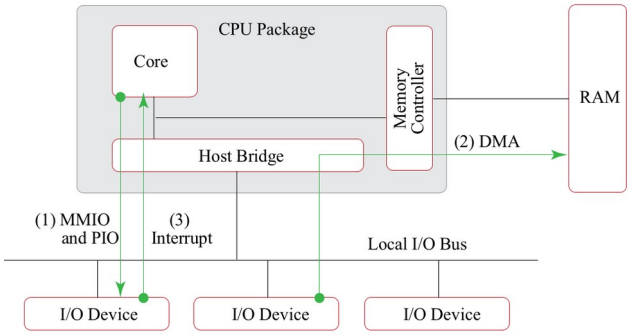


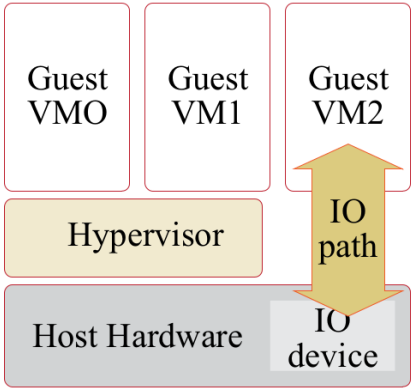


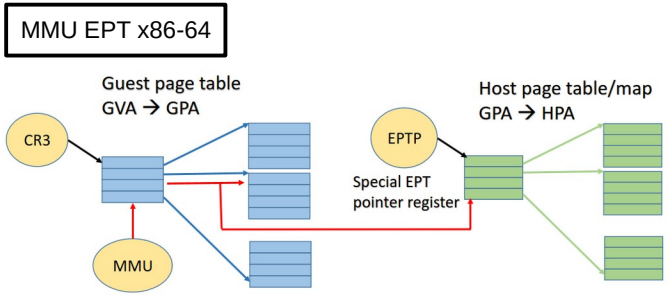
x86-64: I/O Virtualization

Virtual I/O HW Support

- What if we allow the VMs to directly (pass-through) access physical devices, and get the overhead of virtualization is significantly reduced.
- This approach, denoted **direct device assignment.**
- But this approach should be **secure** and **scalable** , so that,
  - Security is obtained with the help of the I/O Memory Management Unit (**IOMMU**).
  - Scalability is obtained via Single-Root I/O Virtualization (**SRIOV**).
  - *Note :*
    *The above 2 techniques regarding Intel x86-64 architecture where **PCIe** is a major component.*



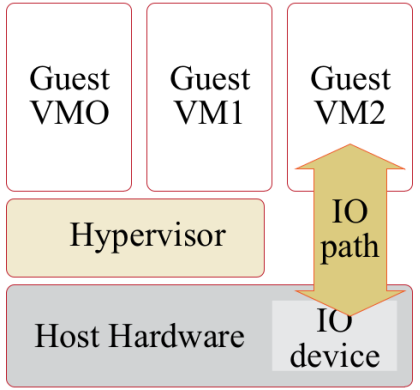




x86-64: I/O Virtualization

Virtual I/O HW Support

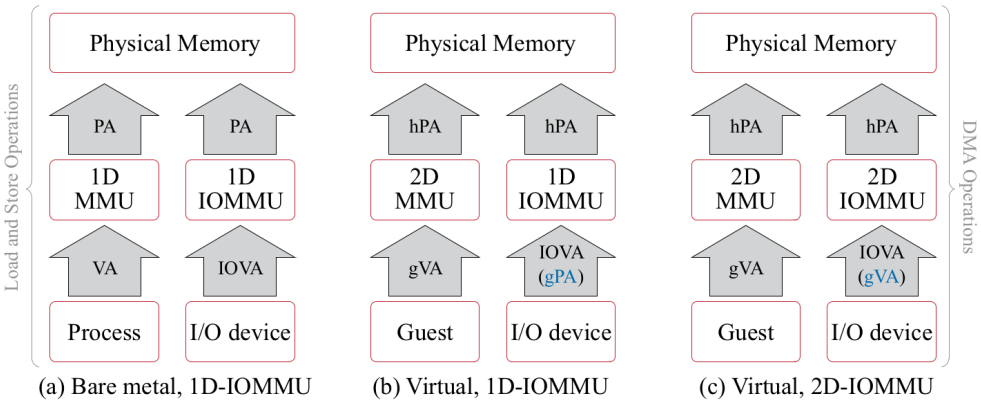
- **IOMMU**
  - IOMMU manages 2 main things
    1) DMA can read/write any memory location,while,Vms do not know the (real) physical location of their DMA buffers!!!
    2) VMs can indirectly trigger any interrupt vector they wish.
  - The IOMMU consists of two main components
    1) **DMA Remapping Engine (DMAR)**
       Allows **DMAs** to be carried out with I/O virtual addresses (**IOVAs**), which the IOMMU translates into physical addresses according to page tables that are set by the **hypervisor**.
    2) **Interrupt Remapping Engine (IR)**
       Translates interrupt vectors fired by devices based on an interrupt translation table configured by the **hypervisor**.



x86-64: I/O Virtualization

Virtual I/O HW Support

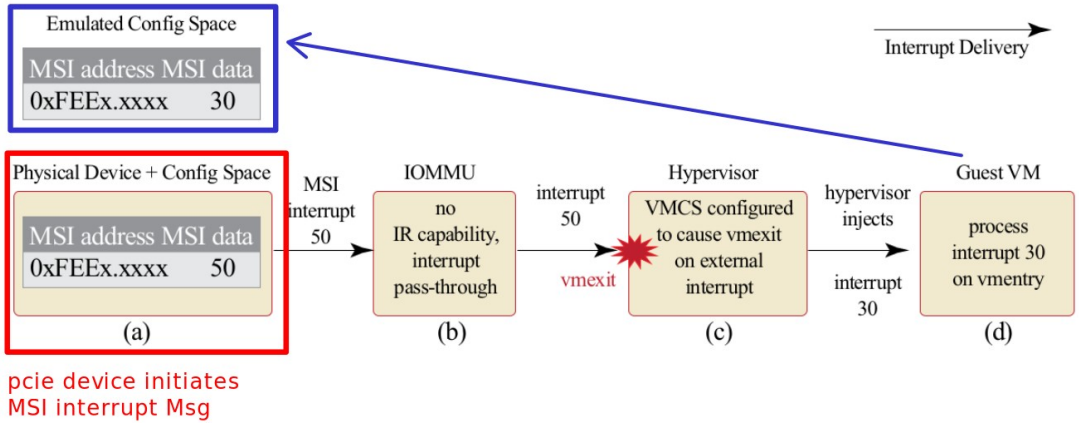
- **IOMMU**
  - DMAR (DMA Remapping)
    - Generally IOMMU plays for I/O devices is similar to the role the regular MMU plays for processes, as illustrated.
    - I/O devices access the memory via DMAs associated with IOVAs, which are translated into physical addresses by the IOMMU.
    - When given a target memory buffer of a **DMA**, the OS associates the physical address (**PA**) of that buffer with some **IOVA**, then, the OS maps the **IOVA** to the **PA** by inserting the **IOVA** => **PA** translation into IOMMU data structures.
    - ***(a)*** *In non-virtual setups, the IOMMU is for devices what the MMU is for processes.*
    - ***(b)*** *When virtual machines are involved, the usage model of the IOMMU (for facilitating direct device assignment) depends on whether the IOMMU supports 1D or 2D page walks.*
      1D IOMMU : The hypervisor does not expose an IOMMU to the guest, causing it to program DMAs with guest physical addresses.
    - ***(c)*** *With a 2D IOMMU, the guest has its own IOMMU, to be programmed.*

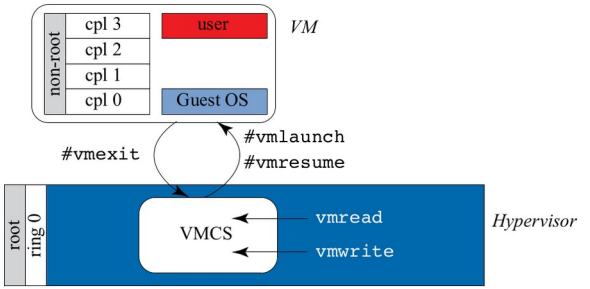


x86-64: I/O Virtualization

Virtual I/O HW Support

- **IOMMU**
  - IR (Interrupt Remapping)
    Ex. Let's see how IR remaps and guarantees the authenticity and legitimacy of the interrupts.
    No IR Capability
    - MSI address : Target LAPIC.
      MSI data : Interrupt vector.
    - The **hypervisor** decided that device physical interrupt vector is **50**.
    - The **VM** configured this vector to be **30** via its emulated configuration space.
    - **MSI** raised by device, then MSI passes through the IOMMU unchanged.
    - It then reaches the target **LAPIC**, thereby triggering an **vmexit** that invokes the hypervisor.
    - Hypervisor injects the interrupt **30** to the VM.



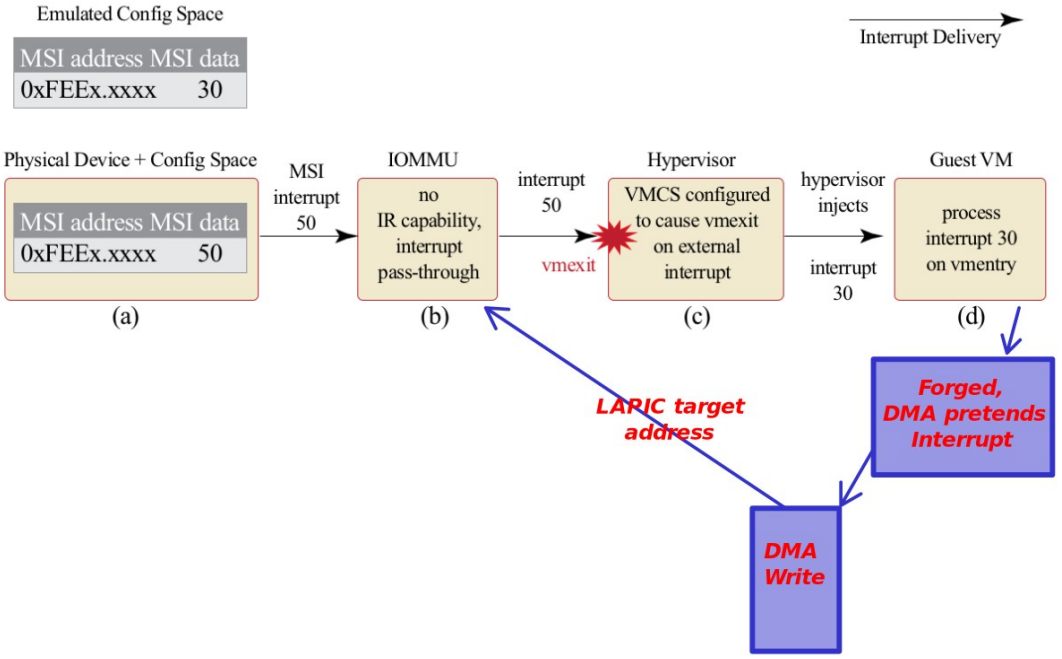


x86-64: I/O Virtualization

Virtual I/O HW Support

- **IOMMU**
  - IR (Interrupt Remapping)
    Ex. Let's see how IR remaps and guarantees the authenticity and legitimacy of the interrupts.
    No IR Capability
    - ***Issue:***
      Interrupt delivery procedure isn't safe.
      Safety rule is broken.

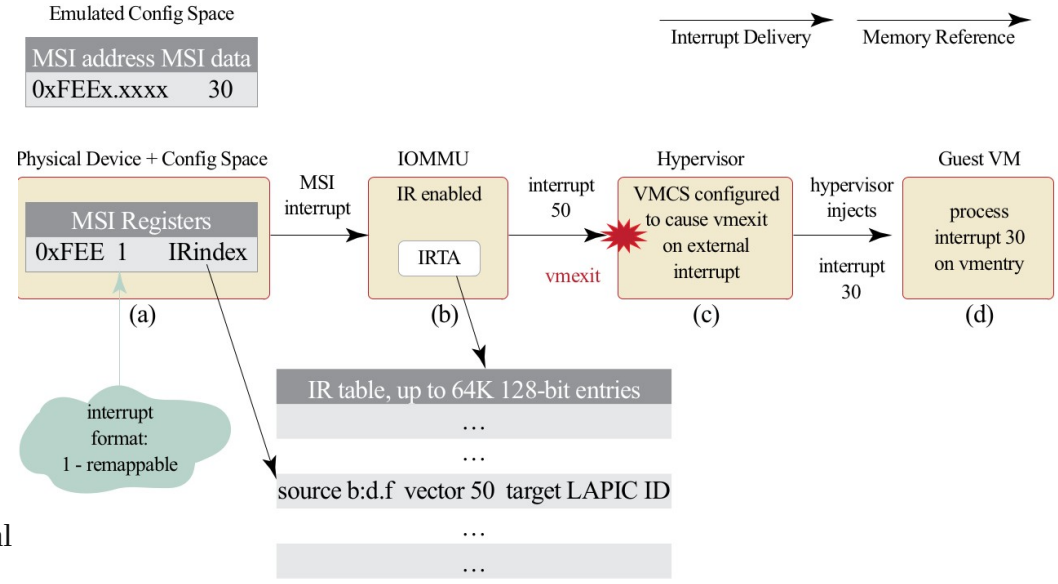
      Because the device is directly assigned to VM, the **VM** can program the device to DMA-write any value (any vector) into the interrupt space, by using ***0xFEEx_xxxx LAPIC*** as a DMA target address
    - Without IR the IOMMU cannot distinguish between a genuine MSI interrupted fired by the device and a rogue DMA that just **pretends to be an interrupt**.



x86-64: I/O Virtualization

Virtual I/O HW Support

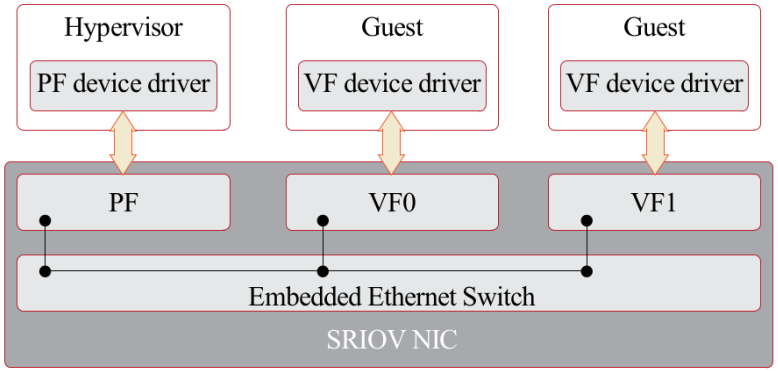
- **IOMMU**
  - IR (Interrupt Remapping)
    IR Capability
    - Instead of storing the physical interrupt vector and target LAPIC, MSI registers now store an Irindex, which is an index to the IR table, pointed to by the IR Table Address (IRTA) register at the IOMMU.
    - When a device fires an interrupt directed at 0xFEEx_xxxx, the IOMMU deduces the Irindex form this “address” and “data”.
    - The IOMMU then determines the target LAPIC and physical interrupt vector based on the associated IR table entry.
    - The IR table entry contains device’s BDF (PCIe Bus, Device, Function) for anti-spoofing, to indicate that the device is indeed allowed to raise the interrupt associated with this Irindex.
    - Only after verifying the authenticity and legitimacy of the interrupt, the IOMMU delivers it to the hypervisor.

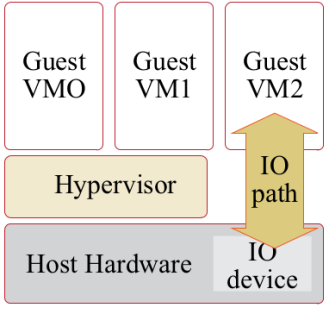


x86-64: I/O Virtualization

Virtual I/O HW Support

- **SRIOV (Single Root I/O Virtualization)**
  - The PCI-SIG standardized the SRIOV specification, which extends **PCIe** to support devices that can “self-virtualize”
  - SRIOV-capable I/O device can present **multiple instances** of itself to software. Each instance can then be assigned exclusively to a different VM.
  - An SRIOV device is defined to have at least one **Physical Function** (PF) and multiple **Virtual Functions** (VFs), which serve as **device instances**.
  - A PF is a standard PCIe function, It has a standard configuration space, and the host software manages it as it would any other PCIe function.
  - The PF also allows the host to allocate, de-allocate, and configure Vfs.
  - A VF is a lightweight PCIe function that implements only a subset of the components of a standard PCIe function as, it cannot de-allocate other Vfs.
  - When a VF is assigned to a VM, the VF provides the VM the ability to do direct I/O and the VM can safely initiate DMAs, such that the hypervisor remains **uninvolved** in the I/O path.
  - Intel implementations of SRIOV NICs enable up to 128 VFs per device.





x86-64: I/O Virtualization

Further Topics

- Exitless Interrupts
- Posted Interrupts
- I/O Page Faults

ARM
Virtualization Support

Design Principles

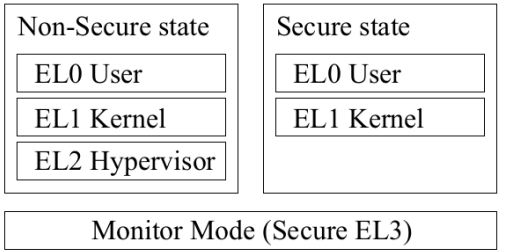
Popek/Goldberg Theorem

- ARM’s central design goal is to fully meet the requirements of the **Popek/Goldberg** theorem.
  - Equivalence
  - Safety
  - Performance

CPU Virtualization

Virtualization Extension

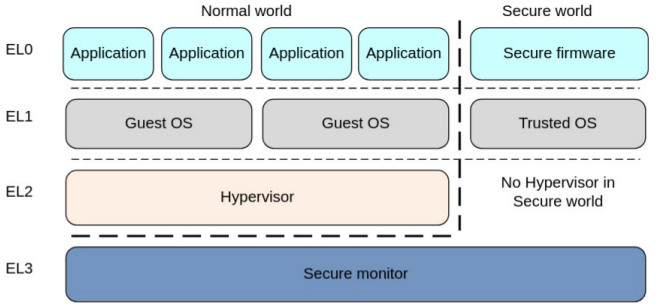
- Generally the overall ARM architecture can be split into 2 modes **secure** and **non-secure** modes.
- A special mode, **monitor mode**, is provided to switch between the secure and non-secure worlds.
- The **EL2** was introduced as a Trap&Emulate mechanism to support virtualization in the **non-secure** mode.
  - The **non-secure** state is where most user applications and operating systems run.
  - **Trap&Emulate** is not actually supported in secure mode as there is no means to trap operations executed in the non-secure world to the secure world.
  - ARM enables virtualization for general-purpose workloads.
- The **Virtualization Extensions** introduces a new **more privileged** mode of execution of the processor, referred to as **HYP** (hypervisor) mode in the **ARMv7**, and **EL2** in **ARMv8** architecture.



CPU Virtualization

ARM Virtualization

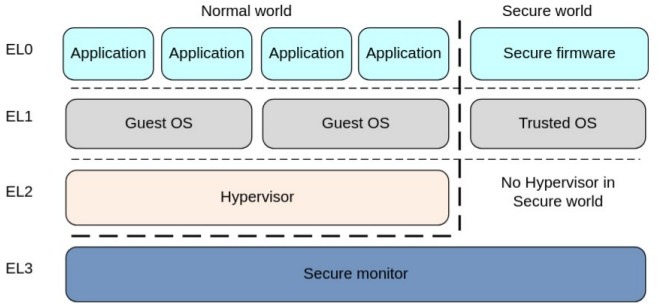
- To support VMs, hypervisor running in EL2 can configure the hardware to **trap** to EL2 from EL0 or EL1 on various sensitive instructions and hardware interrupts.
- So, a hypervisor enables the virtualization features in EL2 before switching to a VM.
- VM will execute normally in EL0 and EL1 until some condition is reached that requires intervention of the hypervisor.
- The ARM architecture allows each trap to be configured to trap directly into a VM's EL1 instead of going through EL2.
  Ex.
  System calls traps, Page faults,…
  they can be handled by the **guest OS** without intervention of the hypervisor.

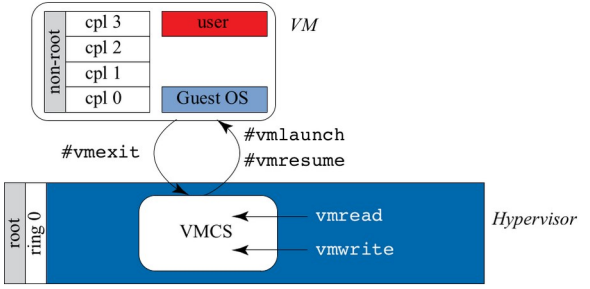


CPU Virtualization

ARM & Intel Virtualization

- Compared to Intel's VT-x, ARM a separate **more privileged** CPU mode, **EL2.** In contrast, Intel has **root** and **non-root** mode, which are orthogonal to the CPU protection modes.
- Intel provides specific hardware support for a **VM control block (VMCS)** which is automatically **saved&restored** when switching to and from root mode.
- ARM hypervisors should implement save&restore VM context by SW.
- The hardware-supported VM control block feature is most likely faster.
- However SW provides some flexibility in what is saved and restored in switching to and from EL2.

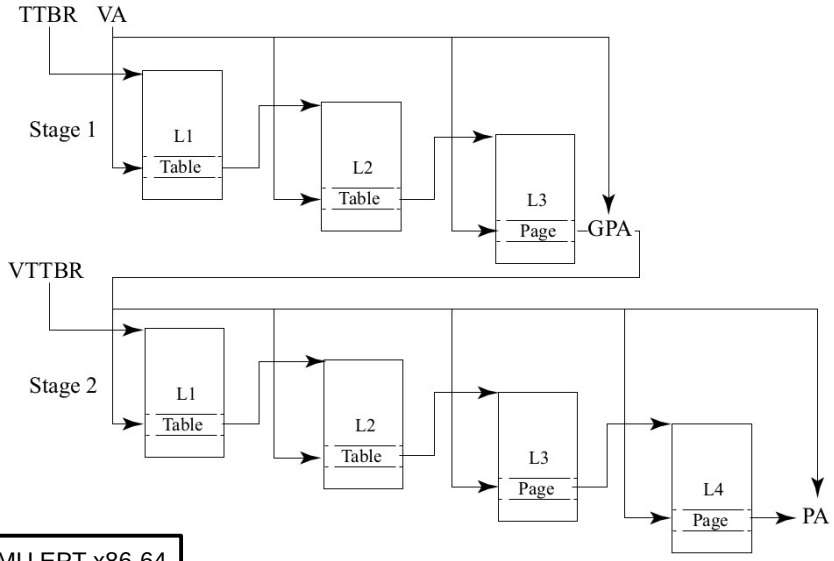




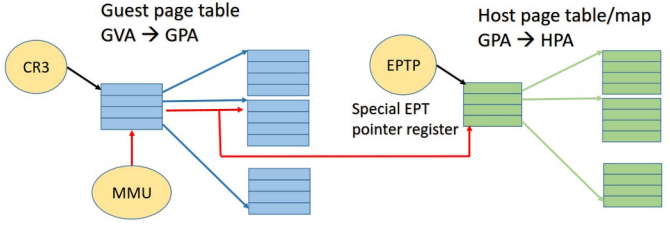
Memory Virtualization

MMU & Page Tables

- ARM and Intel are quite similar in their support for virtualizing physical memory.
- Extended Page Table (**EPT**) or nested page table is supported (Two Stages page tables).
- Intel GPA (Guest Virtual Address) → ARM IPA (Intermediate Physical Address).
- Three levels of page tables are used for **Stage-1** translation from virtual to guest physical addresses.
- Four levels of page tables are used for **Stage-2** translation from guest to host physical addresses.
- **TTBR** : Translation Table Base Register.
- **VTTBR** : Virtualization Translation Table Base Register.
- As seen, the virtual address (VA) is first totally translated into a guest physical address (gPA) and finally into a host physical address (hPA).



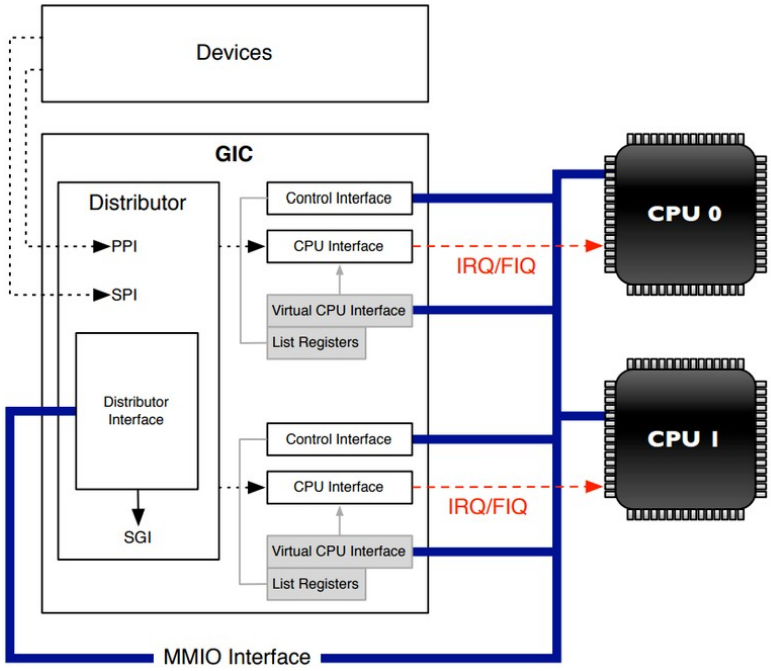
MMU EPT x86-64



Interrupt Virtualization

VGIC

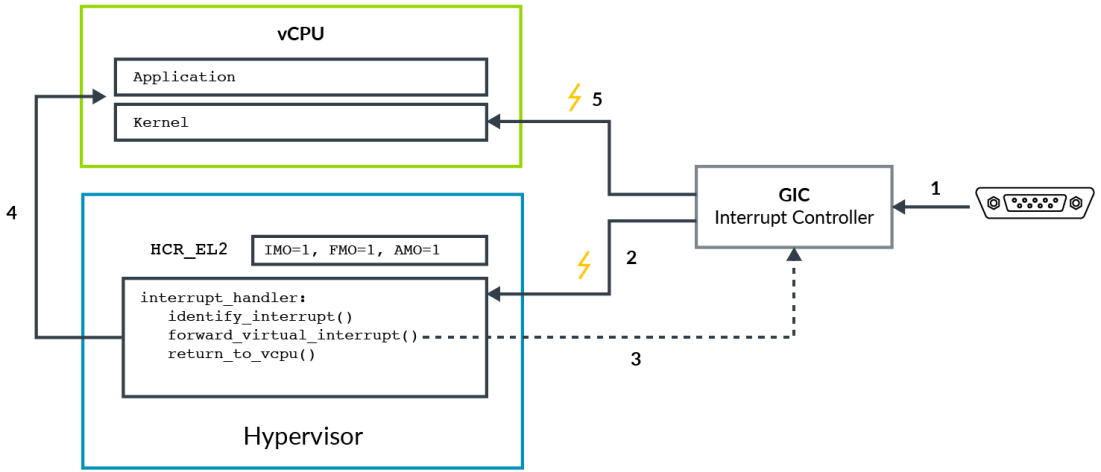
- The **GIC** (Generic Interrupt Controller) routes interrupts from devices to CPUs and CPUs query the GIC to discover the source of an interrupt.
- The GIC is split in two parts, the **distributor** and the CPU **interfaces**.
- There is only one distributor in a system, but each CPU core has a GIC CPU interface.
- Both the CPU **interfaces** and the **distributor** are accessed over a Memory-Mapped interface (**MMIO**).
- The GIC includes hardware virtualization support in the form of a virtual GIC (**VGIC**).
- Each CPU has **VGIC interface,** VMs are configured to see the VGIC CPU interface instead of the GIC CPU interface.
- Virtual interrupts are generated by writing to special registers, the ***list registers***, in the **VGIC** hypervisor control interface, and the VGIC CPU interface raises the virtual interrupts directly to a VM's kernel mode.

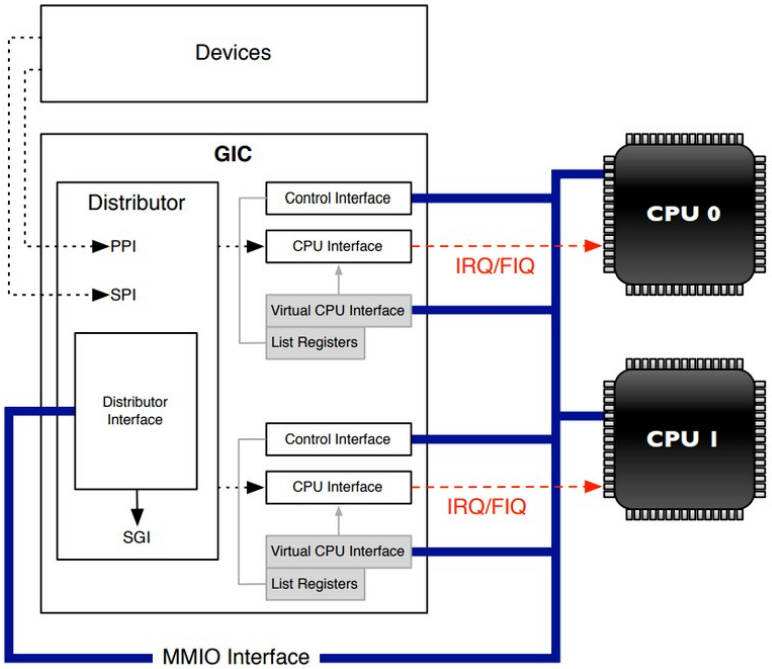


Interrupt Virtualization

VGIC

- The **distributor** must still be emulated in software and all accesses to the distributor by a VM must still trap to the hypervisor.
- For example, when a virtual CPU sends a virtual IPI to another virtual CPU, this will cause a trap to the hypervisor, which emulates the distributor access in software and programs the ***list registers*** on the receiving CPU's VGIC hypervisor control interface.
- The ***virtualized interrupts*** feature on ARM is little bit similar to ***Posted Interrupts*** feature in x86-64.

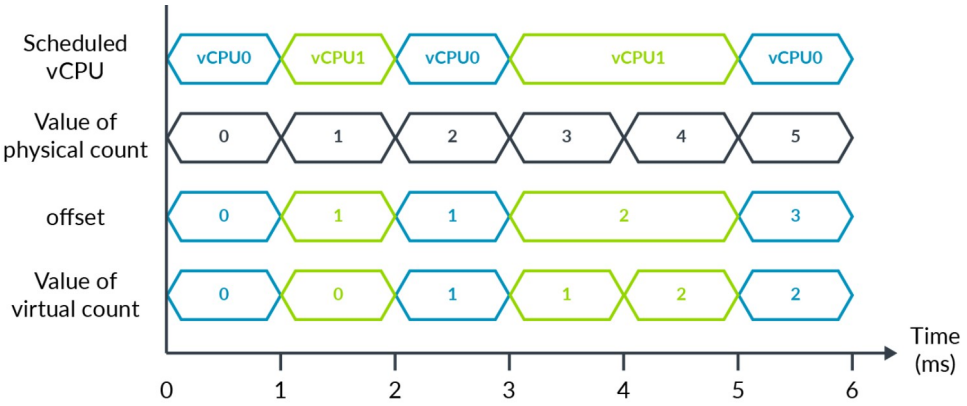


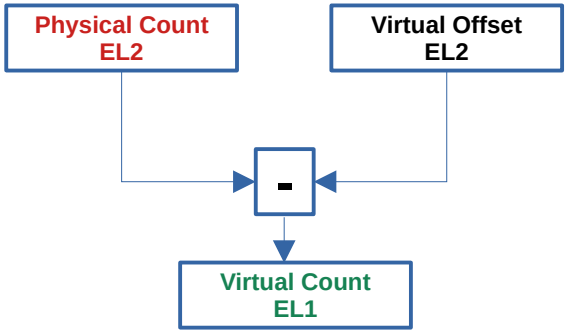


Timer Virtualization

Virtualizing Generic Timers

- ARM Generic Timer Architecture, provides a timer for each CPU, which is programmed to raise an interrupt to the CPU after a certain amount of time has passed (***physical timer***).
- ARM provides virtualization support for the timers by introducing a new timer called the ***virtual timer***.
- A hypervisor can be configured to use ***physical timers*** while **VMs** are configured to use ***virtual timers***.
- EL2 configures an ***offset register***, which is subtracted from the physical counter and returned as the value when reading the virtual counter.
- *What's the problem?*



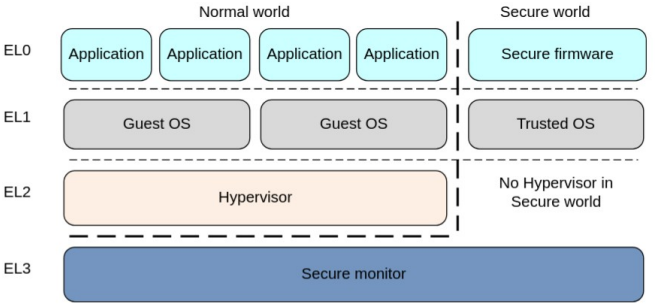


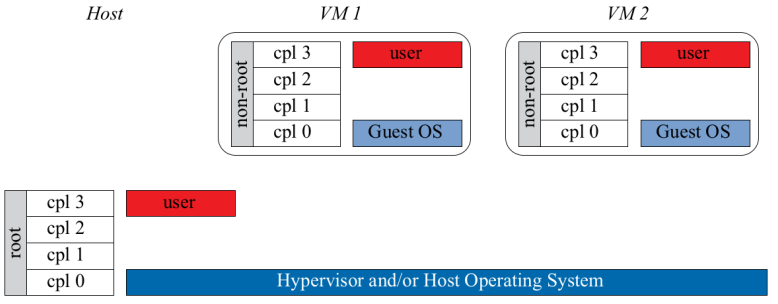
KVM/ARM

Run Kvm (Linux) In EL2 Mode?!!

- Running KVM/ARM in EL2 implies running the Linux kernel in EL2.
- *Why is this a problematic?*
  - Low-level architecture dependent code in Linux is written to work in kernel mode (EL1) and EL2 is completely different.
  - To preserve compatibility with hardware without EL2 and to run Linux as a guest OS, low-level code would have to be written to work in both modes, potentially resulting in slow and convoluted code paths.
    Ex.
    A **page fault** handler needs to obtain the virtual address causing the page fault.
    In EL2, this address is stored in a different **register** than in kernel mode.
  - These problems do not occur for **x86** hardware virtualization.

    The entire Linux **kernel** can run in **root mode** as a **hypervisor** because the same set of CPU modes available in **non-root** mode are available in root mode.

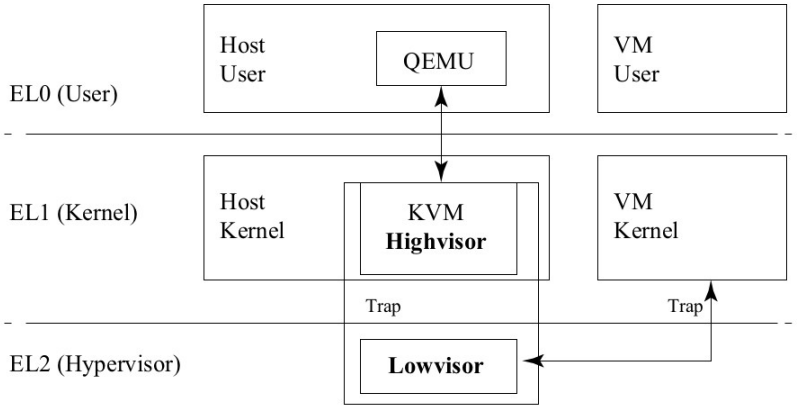




KVM/ARM

Split-mode Virtualization

- A new approach to hypervisor design that **splits the core hypervisor** so that it runs across different privileged CPU modes to take advantage of the specific benefits and functionality offered by each CPU mode.
- Splitting the hypervisor into two components, the **Lowvisor** (EL1) and the **Highvisor** (EL2).
- **Lowvisor (EL2)**
  - Directly interacts with hardware protection features.
  - Switches from a **VM execution** context to the **host execution** context and vice-versa.
  - Provides a virtualization **trap handler**, which handles interrupts and exceptions that must trap to the hypervisor.



KVM/ARM

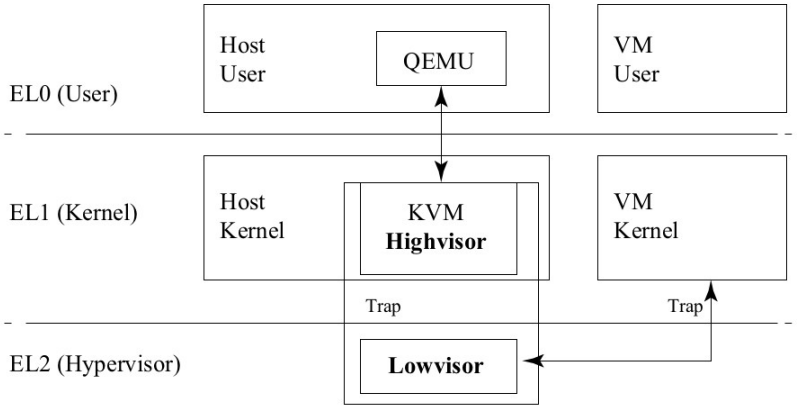
Split-mode Virtualization

- **Highvisor (EL1)**
  - Leverage existing Linux functionality such as the **scheduler**, kernel **data structures** and **mechanisms** such as **locking** mechanisms and **memory allocation** functions.
  - As the lowvisor provides a low-level trap-handler and switch from mode to another, the **highvisor** handles Stage-2 page faults from the VM and performs instruction **emulation**.
- Because the hypervisor is split across kernel mode (EL1) and EL2, switching between a **VM** and the **highvisor** involves multiple mode transitions.

  1) Trap while running the VM to run lowvisor (EL2).
  2) Lowvisor causes another trap to run the highvisor (EL1).

  And from **highvisor** (EL1) to **VM**:

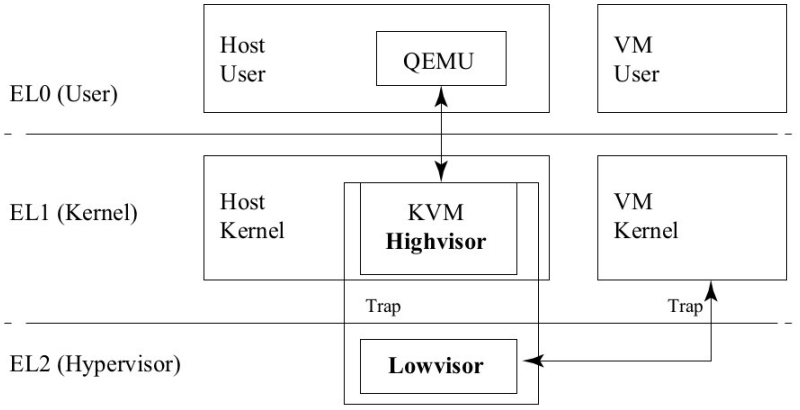
  1) Trapping from kernel mode to EL2 (lowvisor).
  2) EL2 (lowvisor) switching to the VM.



KVM/ARM

Split-mode Virtualization

- KVM/ARM uses a memory mapped interface to share data between the **highvisor** and **lowvisor** as necessary.
  - It leverages the highvisor's ability to use the existing memory management subsystem in Linux to manage memory for both the highvisor and lowvisor.
  - Managing the lowvisor's memory is very challenging, because it requires managing EL2's **separate address space**.
  - One simplistic approach would be to reuse the host kernel's page tables and also use them in EL2 to make the address spaces identical.
  - This unfortunately does not work, because EL2 uses a different page table format from kernel mode.
  - Therefore, the highvisor explicitly manages the EL2 page tables to map any code executed in EL2 and any data structures shared between the highvisor and the lowvisor to the same virtual addresses in EL2 and in kernel mode.



QEMU
Internals

KVM
Virtual Machines