虚拟机、QEMU与KVM

目录

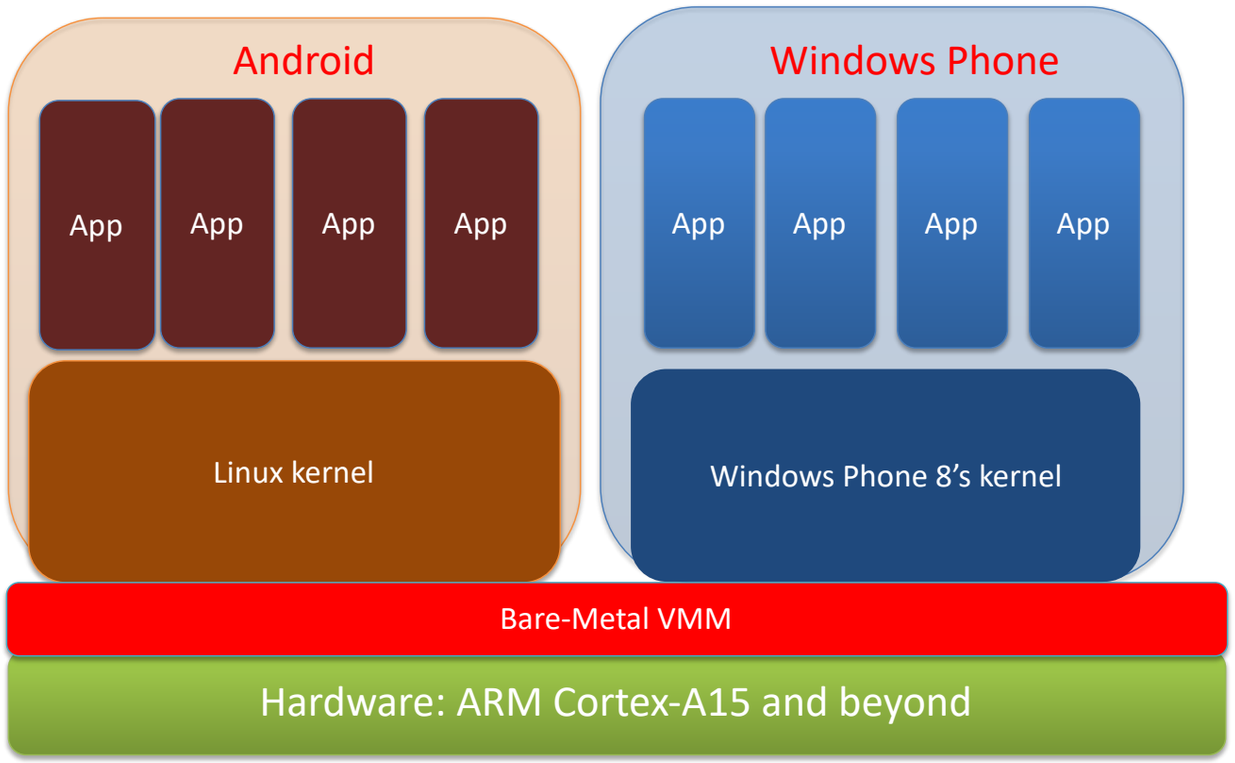
虚拟机



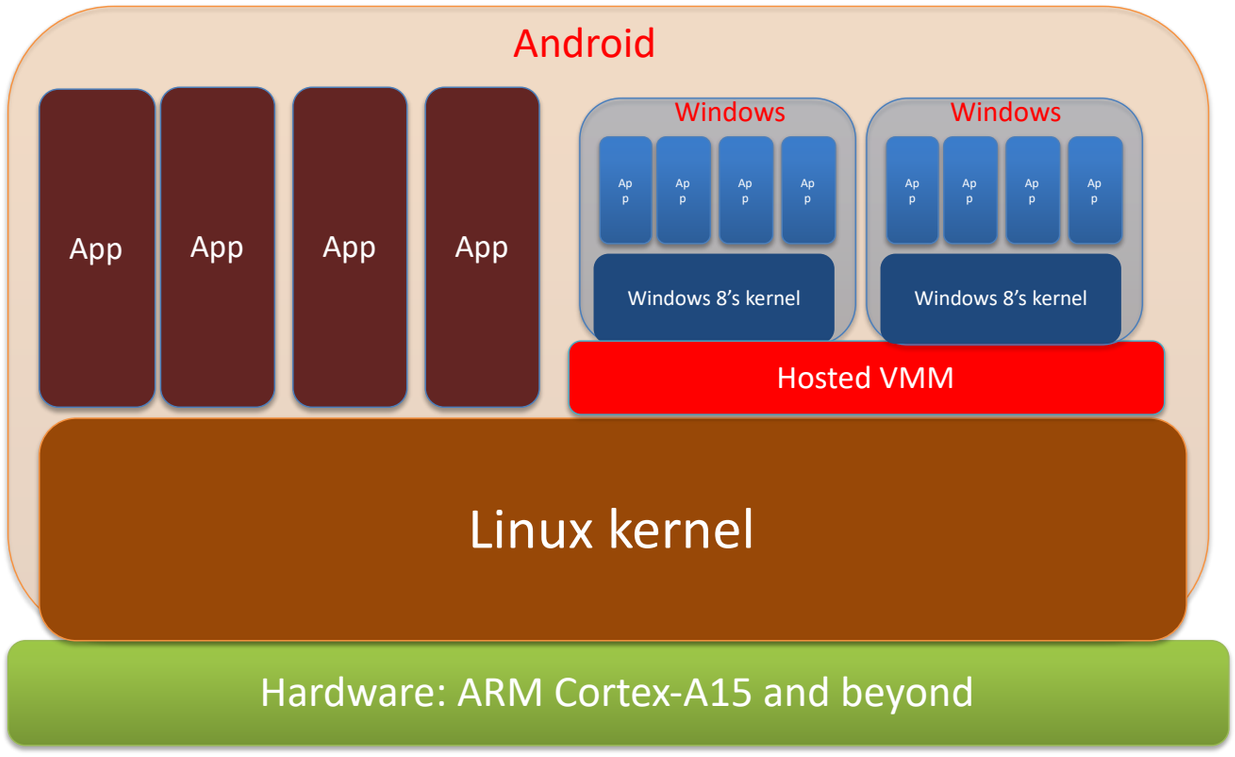
为什么要有虚拟机?

虚拟机分类

VMM：Virutal machine monitor



Bare-metal VMM

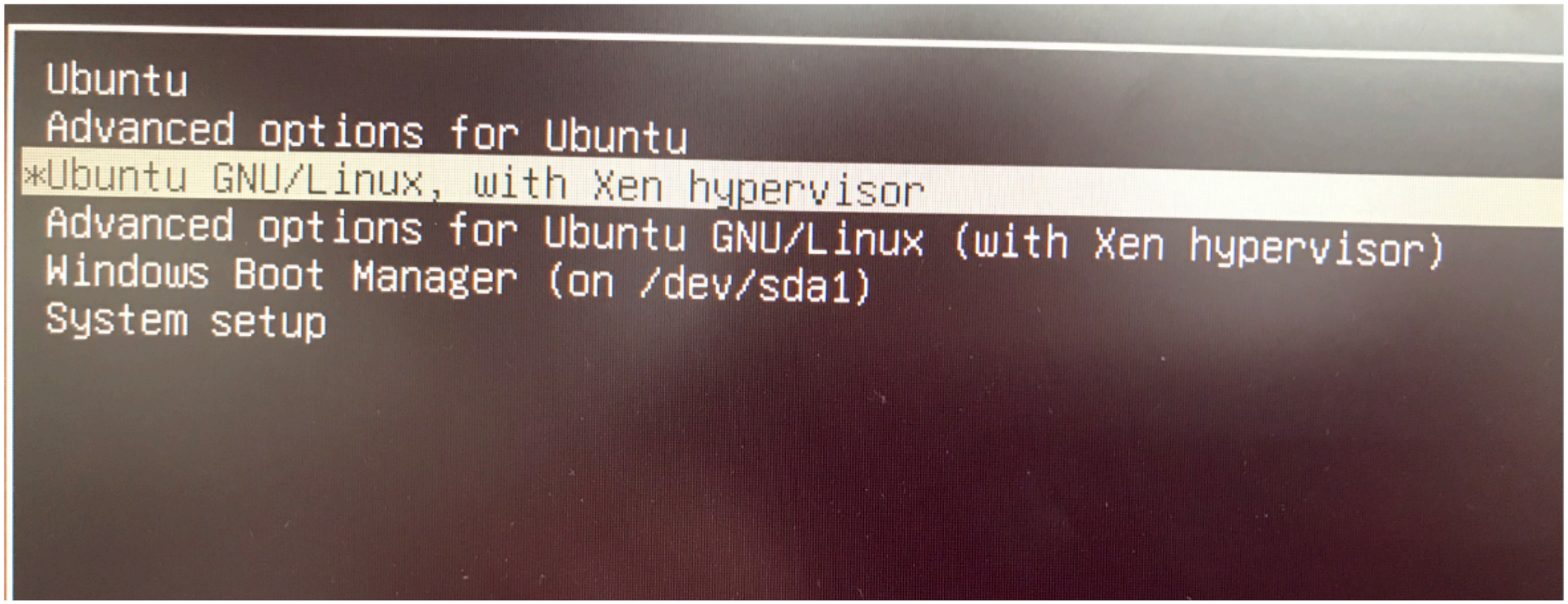


Hosted VMM

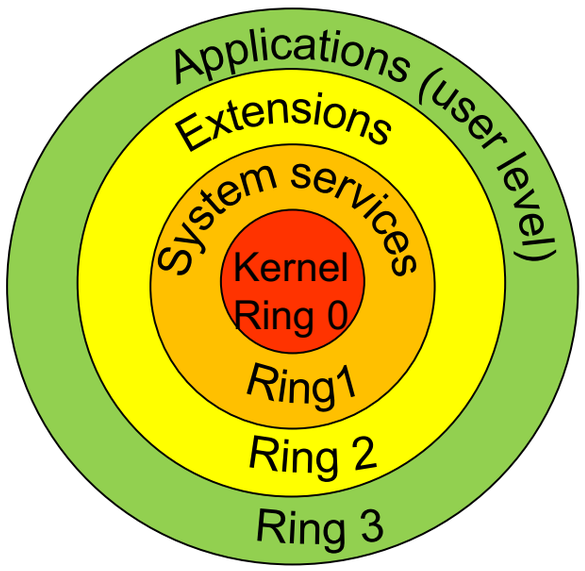
Xen
Vmware workstation
Hyper-V

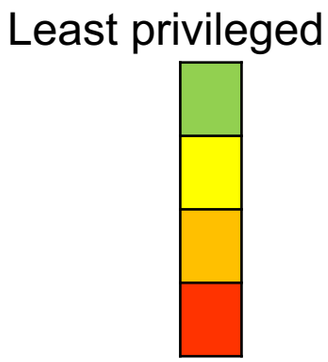
KVM
VMware player
Parallers

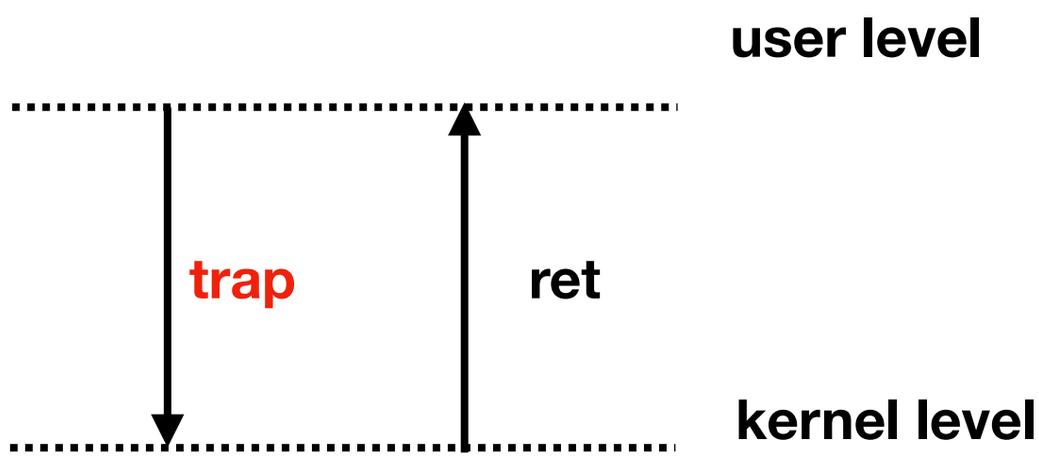
虚拟机分类


Ubuntu
Advanced options for Ubuntu
*Ubuntu GNU/Linux, with Xen hypervisor
Advanced options for Ubuntu GNU/Linux (with Xen hypervisor)
Windows Boot Manager (on /dev/sda1)
System setup

特权级







trap类型:

1.系统调用: read, write, ioctl…

2.硬件中断: hardware events

3.异常: 当cpu执行指令时检测到一个或多个预先定义的错误，运算错误、指令权限不够、page fault。

Protection ring

特权级指令

printf(“3/0 = %d\n”, 3/0)

```
xulin@xulin:~/test$ ./privilege_test
Floating point exception (core dumped)
xulin@xulin:~/test$ dmesg  |tail -n 1
[265568.796473] traps: privilege_test[20162] trap divide error ip:55a113a4e144 sp:7ffcaea85960 error:0 in privilege_
```

asm volatile(“cli”)
asm volatile(“sti”)

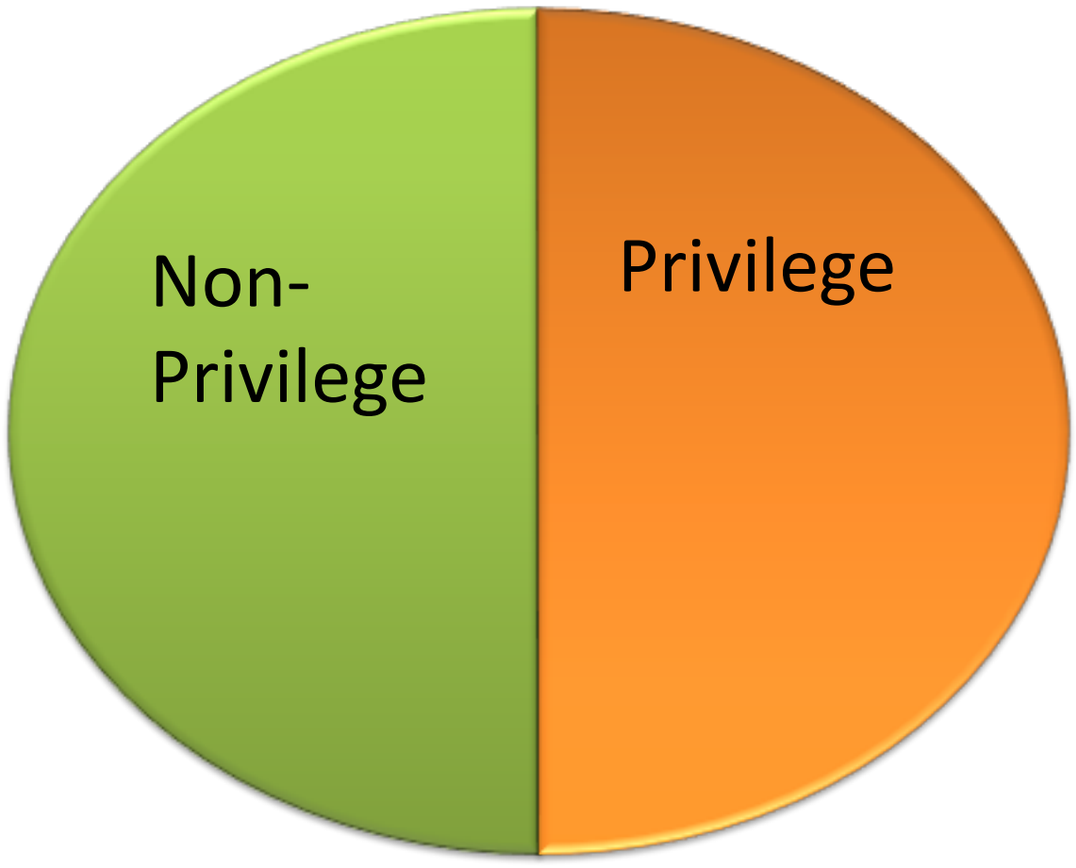
```
xulin@xulin:~/test$ ./privilege_test
Segmentation fault (core dumped)
xulin@xulin:~/test$ dmesg  |tail -n 1
[265653.111577] traps: privilege_test[20233] general protection fault ip:563579357129 sp:7fff527f24f0 error:0 in privilege_
```

```
 int err;
 printk(KERN_INFO "Hello, i'm in the kernel world !\n");

 printk(KERN_INFO "I'm going to do some dangerous things!!\n");
 asm volatile("cli\n\t");
 asm volatile("sti\n\t");
```

```
xulin@xulin:~/test/privilege_test$ sudo insmod privilege_test.ko
xulin@xulin:~/test/privilege_test$ sudo rmmod privilege_test.ko
xulin@xulin:~/test/privilege_test$ dmesg | tail -n 5
[266546.453654] I'm going to do some dangerous things!!
[266575.998028] Oh, I'm going to exit !
[266599.228024] Hello, i'm in the kernel world !
[266599.228024] I'm going to do some dangerous things!!
[266603.307725] Oh, I'm going to exit !
```

特权级指令

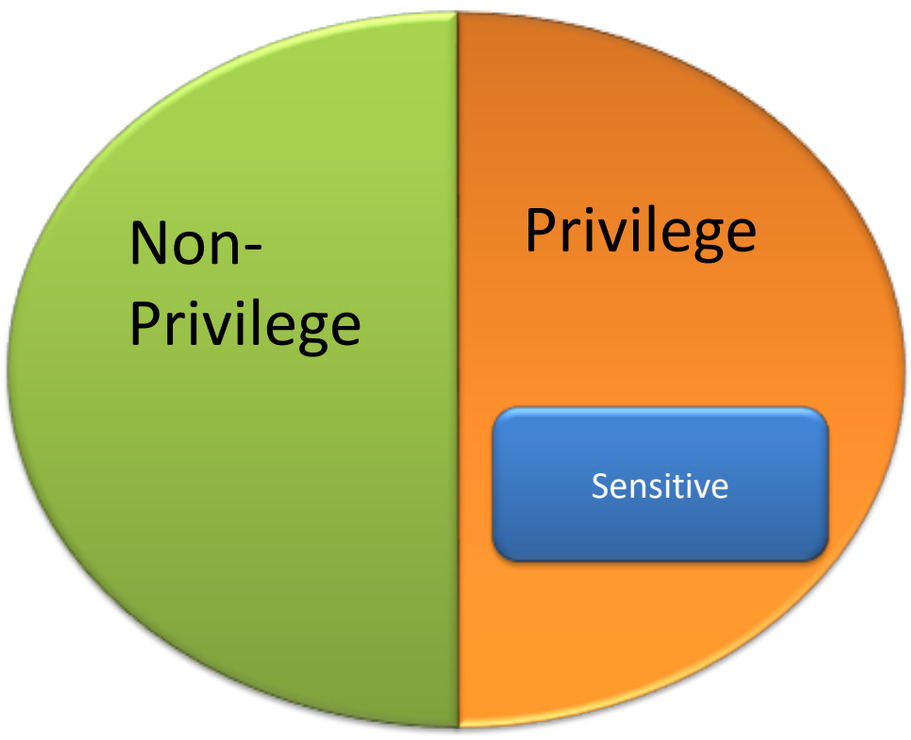


privilge: 在用户态运行时会trap到内核态的指令.
Non-privilege: 所有其他指令.

Decidede by CPU designer

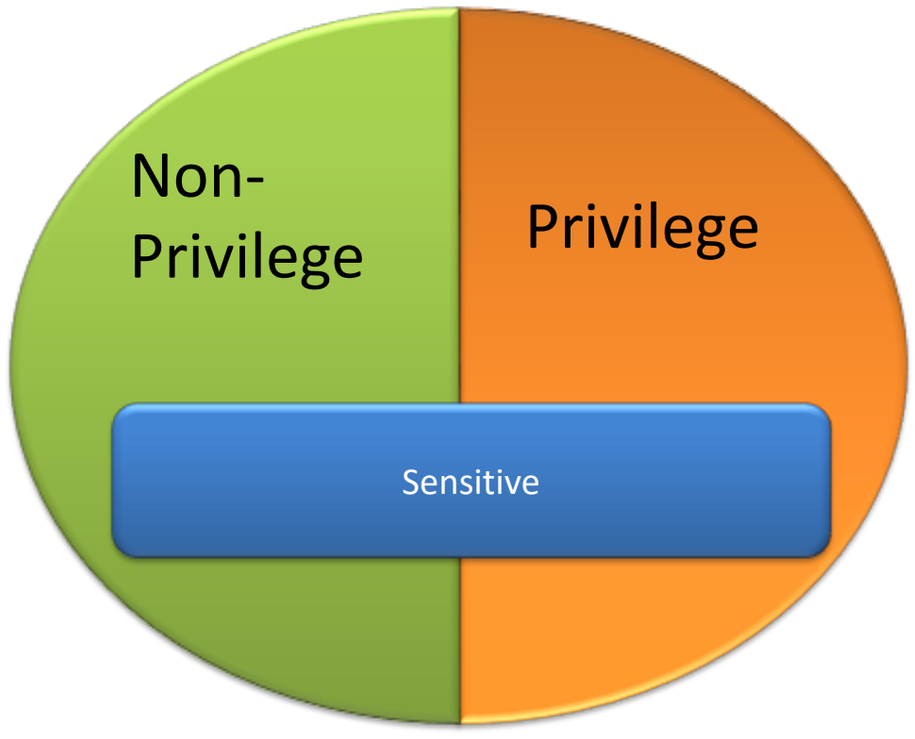
敏感指令

敏感指令：指令和操作硬件相关，例如修改页表基地址、控制中断控制器等
非敏感指令：其他指令



所有的敏感指令都是特权级指令。
IBM PowerPC
IBM S/390

可虚拟化的CPU

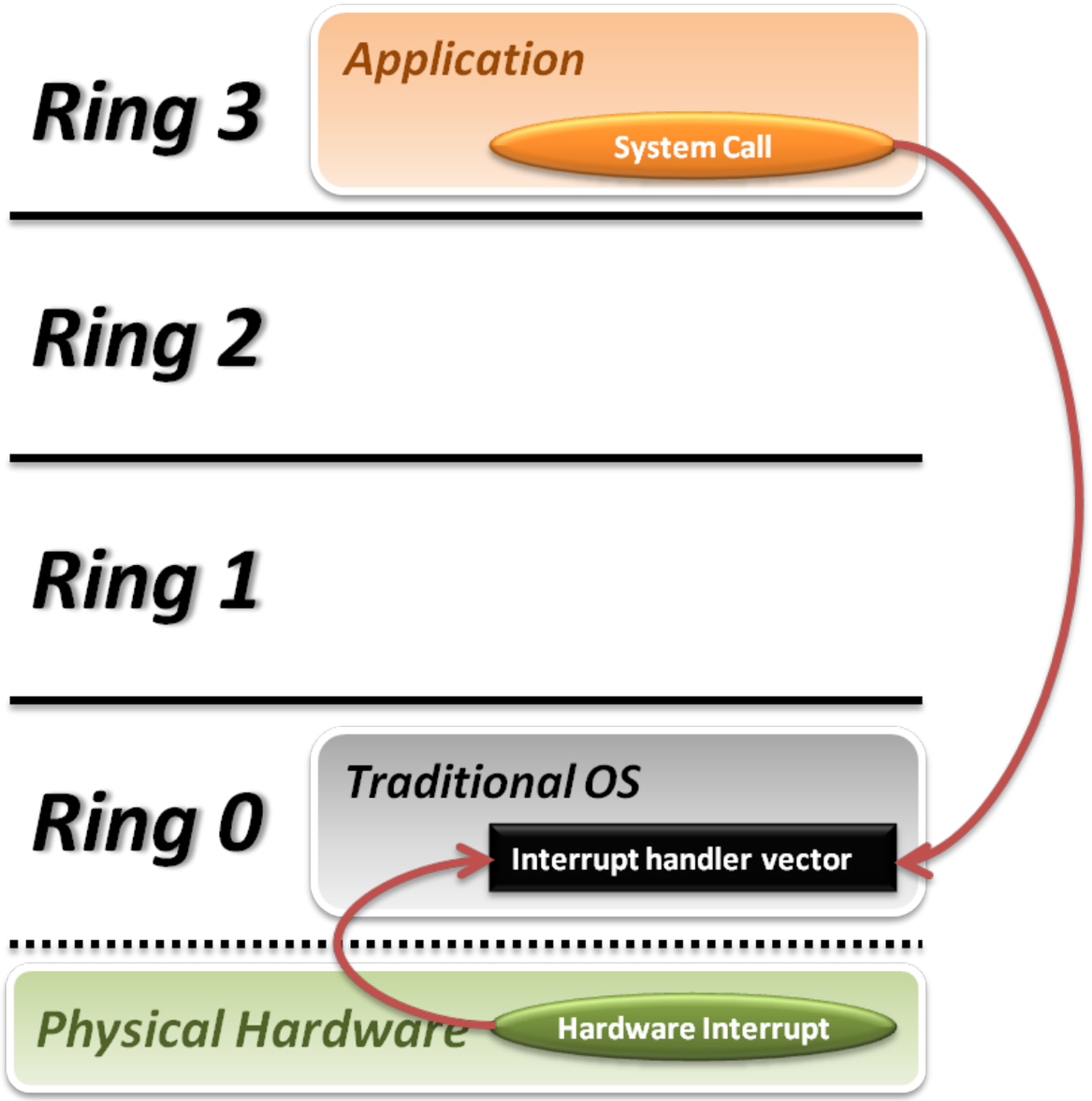


一些敏感指令不是特权级指令。
X86
ARM

存在虚拟化漏洞的CPU

对于可虚拟化的CPU，如何来设计VMM?

陷入再模拟



正常的操作系统

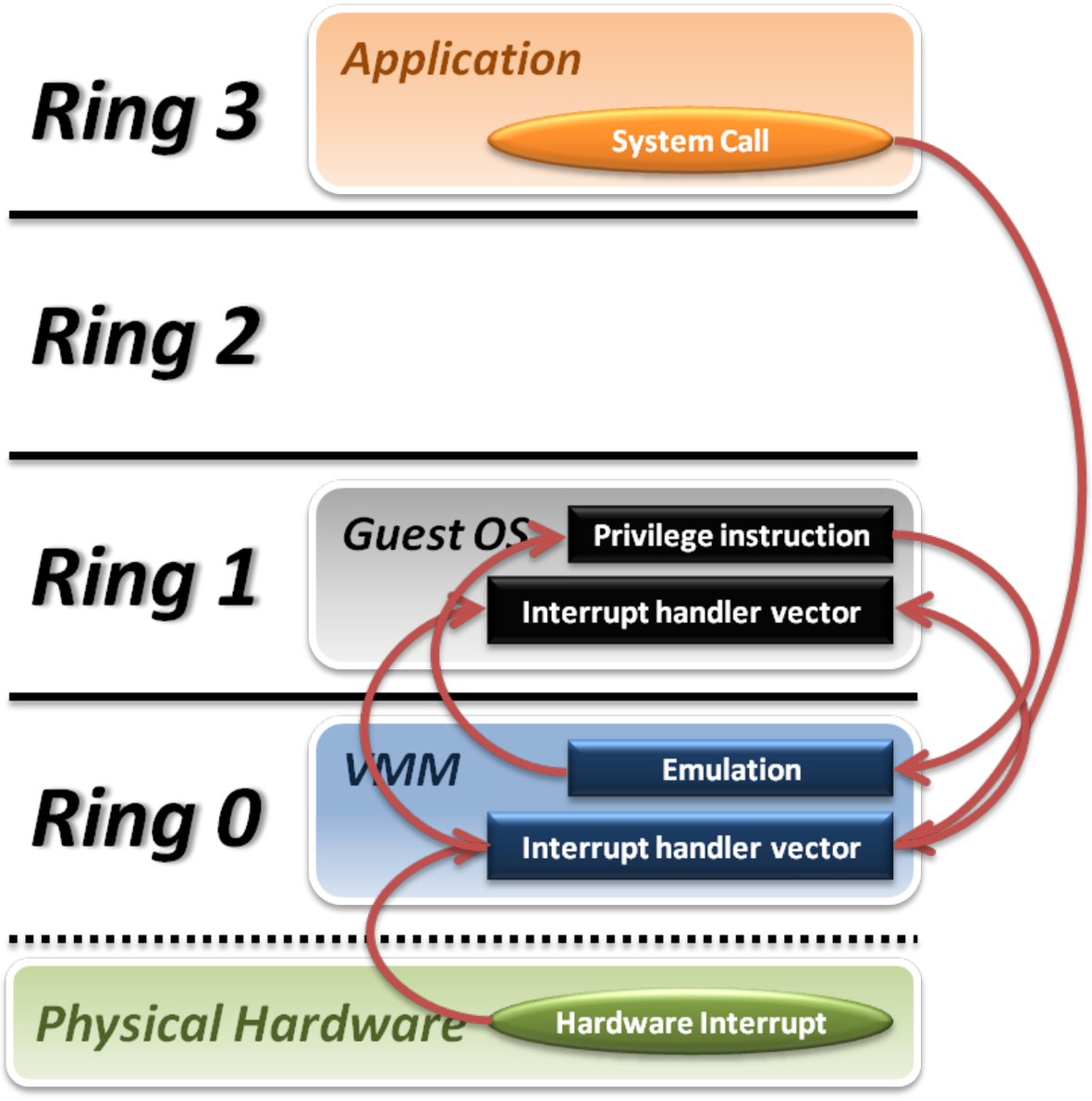
在产生系统调用时：

- CPU跳转到中断向量表
- CPU切换到内核态运行

在产生硬件中断时：

- 硬件会打断CPU的执行，跳转到中断向量表

陷入再模拟



虚拟机

在产生系统调用时：

1. CPU跳转到VMM的中断向量表
2. 接着CPU跳转到Guest OS中

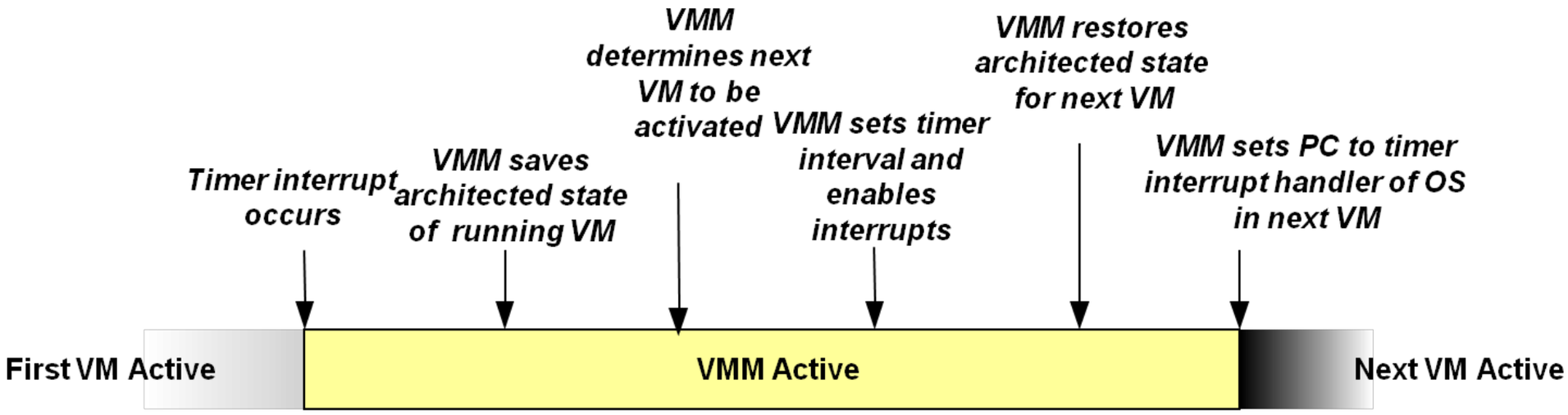
在产生硬件中断时：

1. 硬件会打断CPU的执行，跳转到VMM的中断向量表
2. 接着跳转到Guest OS的中断向量表

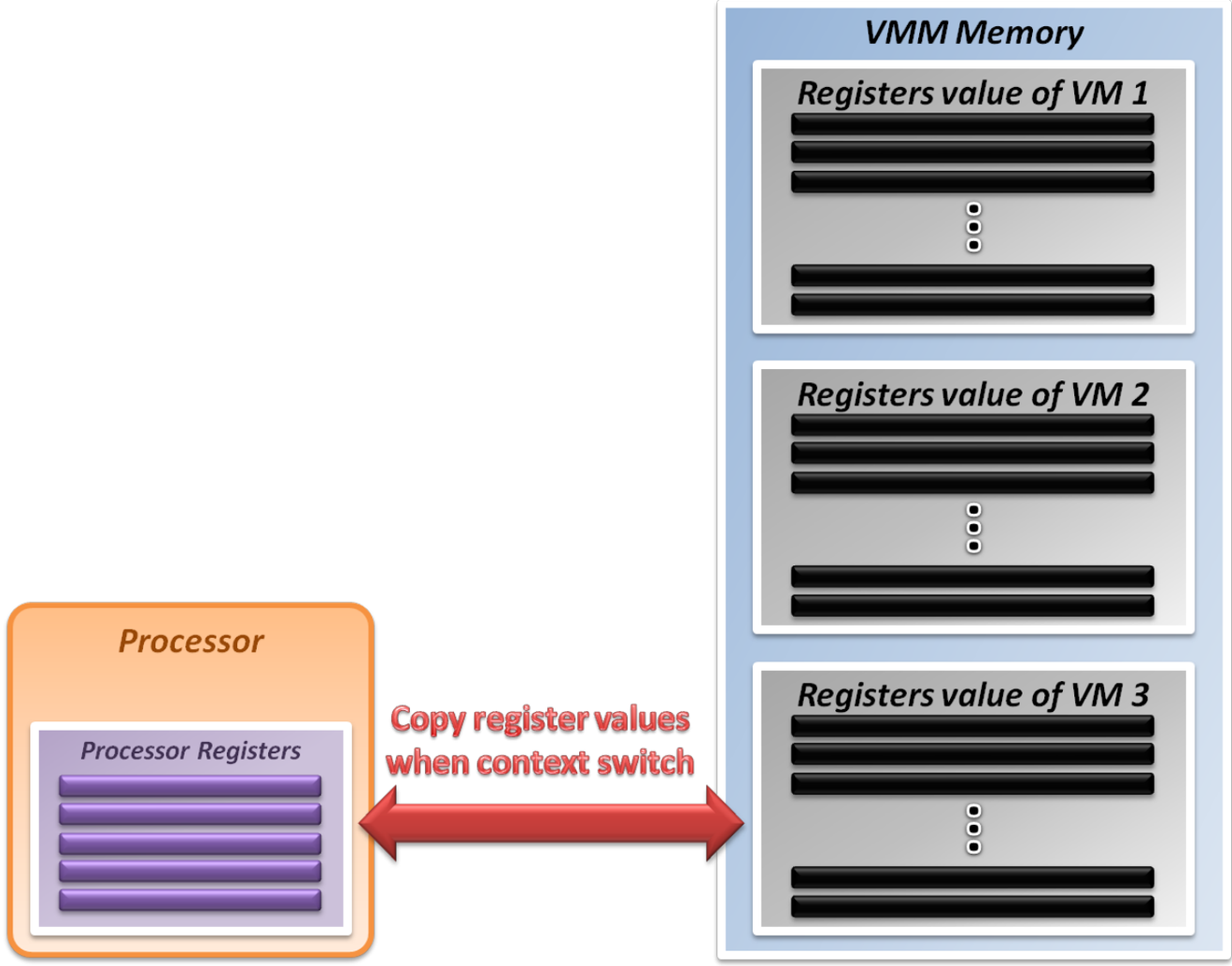
在执行特权指令时：

1. CPU陷入到VMM中进行指令模拟
2. 指令模拟结束后，跳转到Guest OS中

虚拟机调度



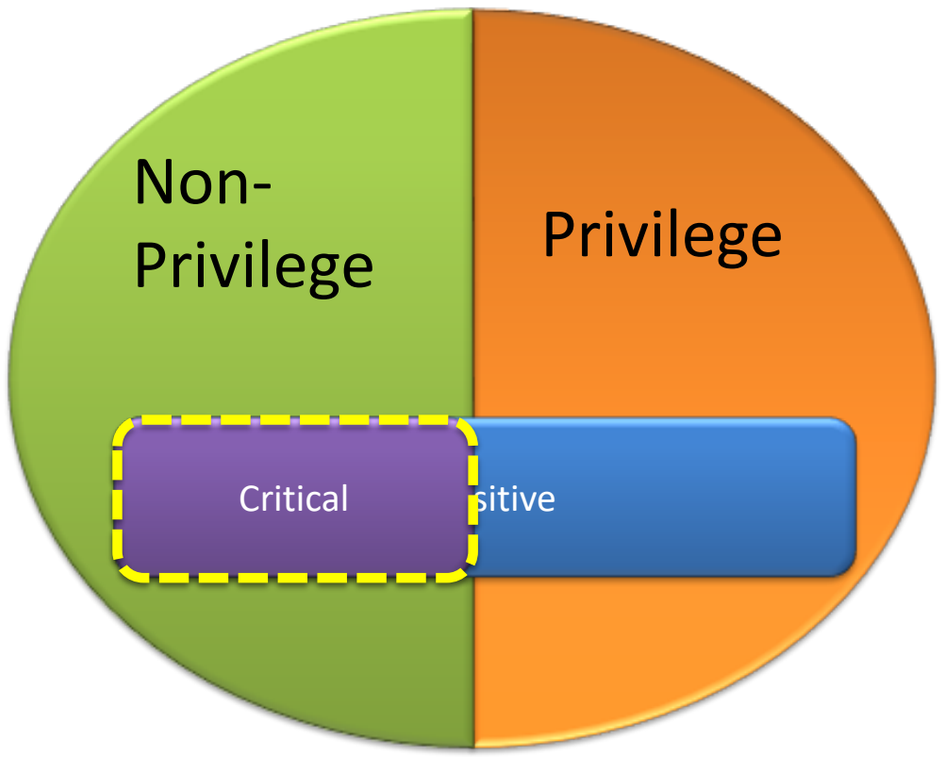
1. 定时器中断
2. 上下文切换到VMM
3. VMM保存当前VM的上下文
4. VMM找到下一个需要运行的VM
5. 更新定时器中断
6. 恢复下一个VM的上下文
7. 跳转到下一个VM的时钟中断handler处



对于存在虚拟化漏洞的CPU，如何来设计VMM?

敏感指令

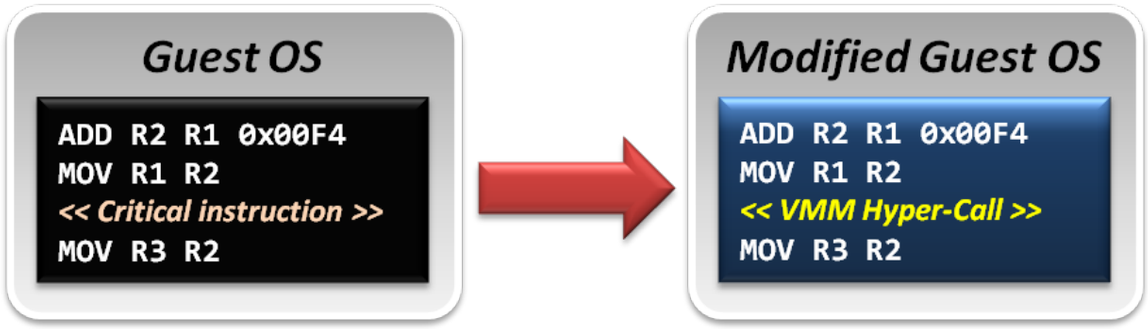
可以在非特权极运行的敏感指令称为critical instruction



敏感指令

半虚拟化：

1. 修改guest os的代码，替换critical instruction。
2. 实现Hyper-call，让guest os可以陷入到VMM中。



指令翻译：

1. 在翻译guest os的代码时，把这些critial instruction翻译为privilege instruction，这样就可以陷入到VMM中。

硬件辅助的方式：

1. 使用CPU提供的虚拟化扩展指令集
2. Intel:VT-x
3. ARM: EL2、hvc、2-stage-mmu

VT-x

VMX Root Operation(Root模式)

-所有指令的表现和之前的一致

-VMM运行在此模式

VMX Non-root opertaion(Non-root模式)

-所有的敏感指令被重新定义

-所有敏感指令的执行都会陷入到Root模式

-虚拟机运行在此模式

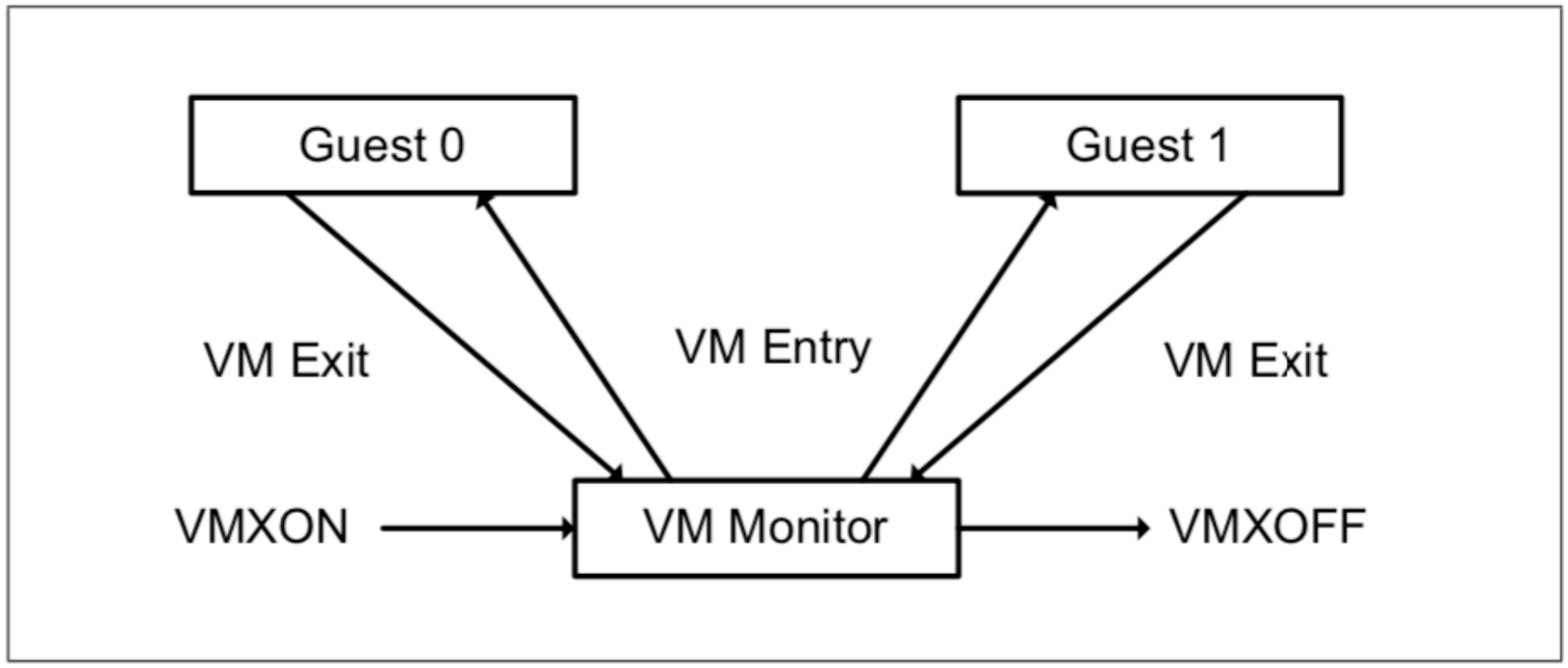
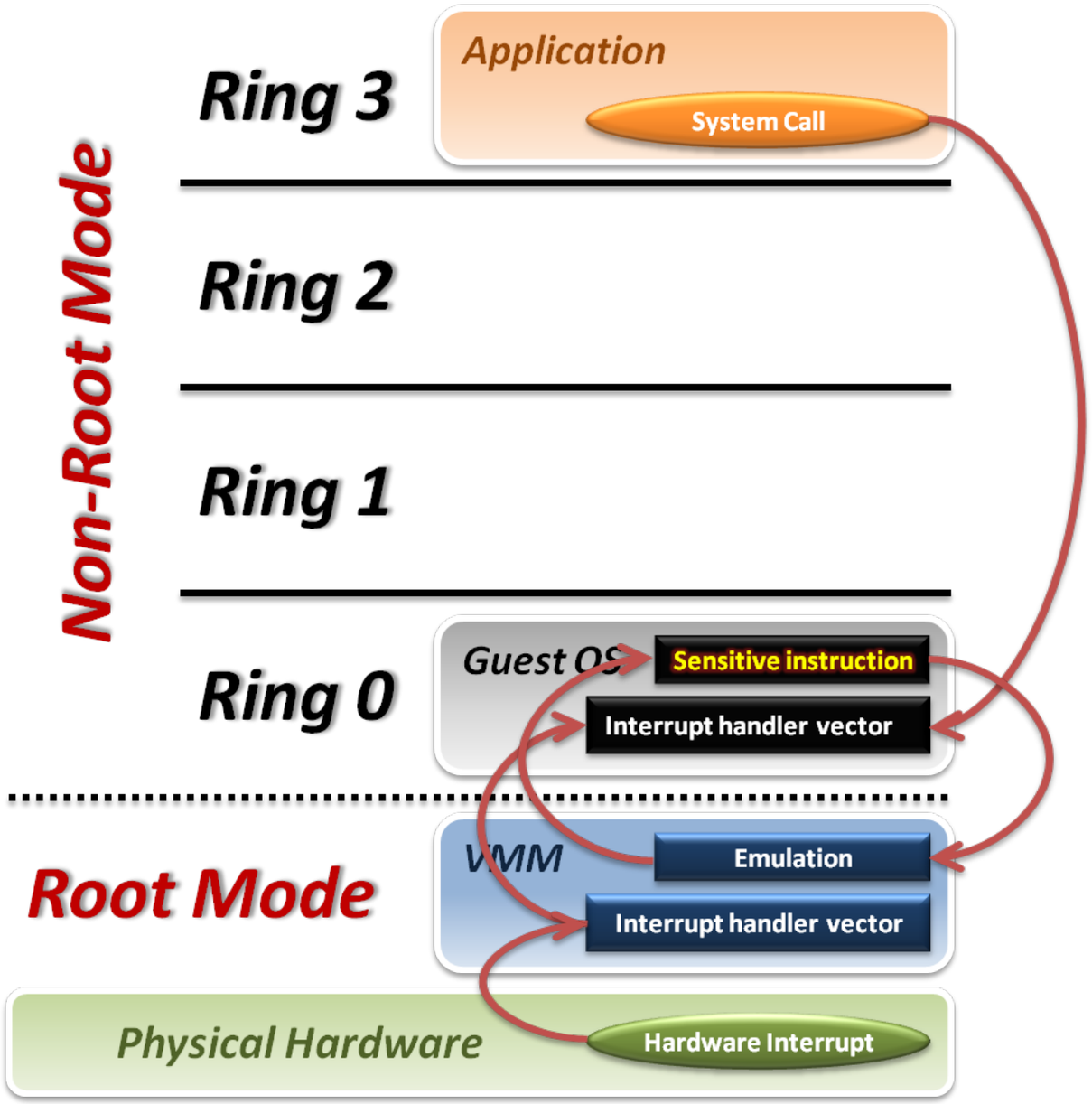


Figure 23-1. Interaction of a Virtual-Machine Monitor and Guests

VM Entry: VMLAUCH or VMRESUME

Intel 64 and IA-32 Architecutres Software Developer's Manual volume 3 chapter 23

VT-x



虚拟机 VT-x

在产生系统调用时：

- CPU跳转到虚拟机的中断向量表

在产生硬件中断时：

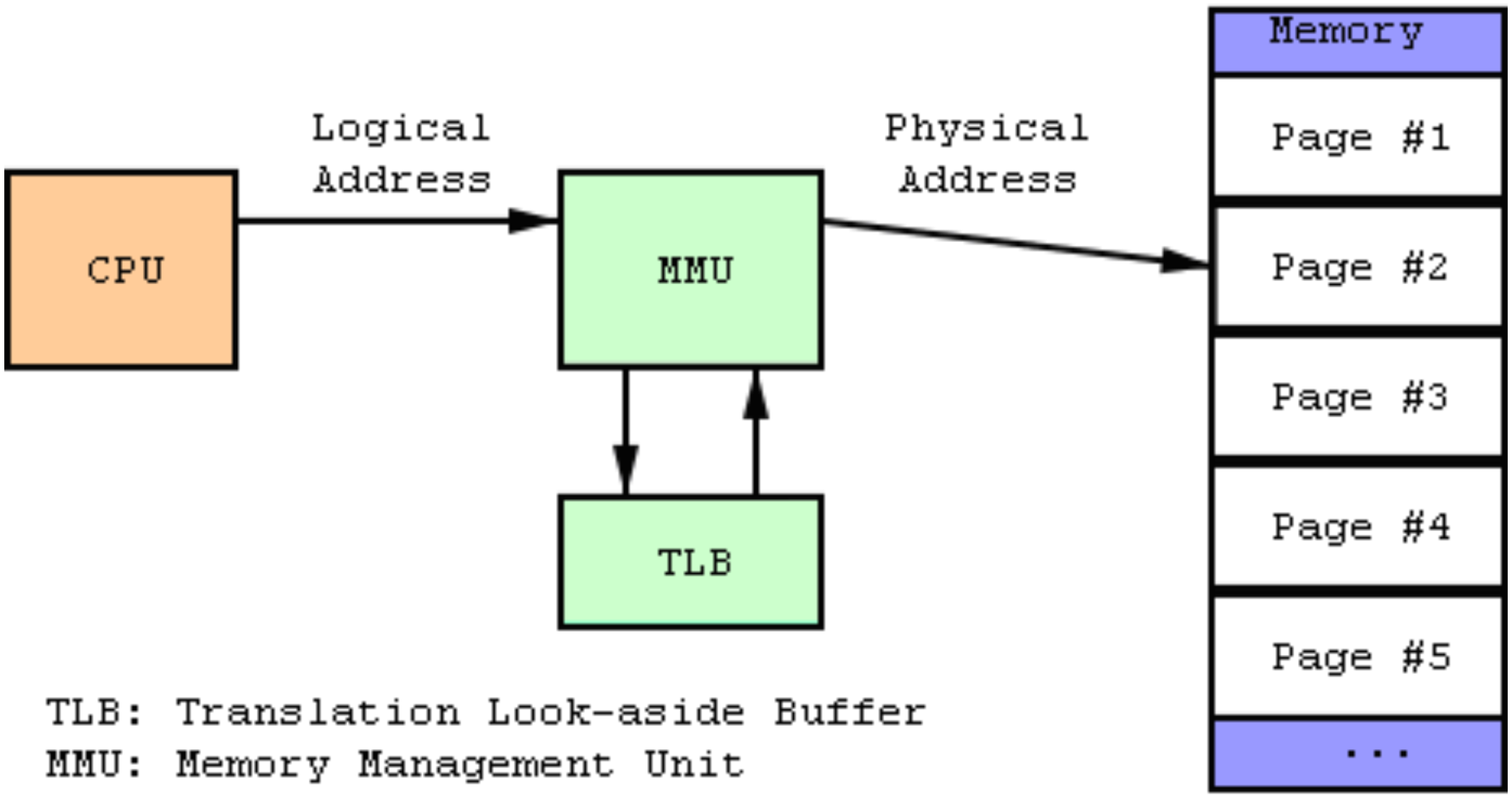
- 硬件会打断CPU的执行，跳转到VMM的中断向量表
- 跳转到Guest OS的中断向量表

在执行特权指令时：

- 只有当CPU执行敏感指令时才陷入到VMM中进行指令模拟
- 指令模拟结束后，跳转到Guest OS中

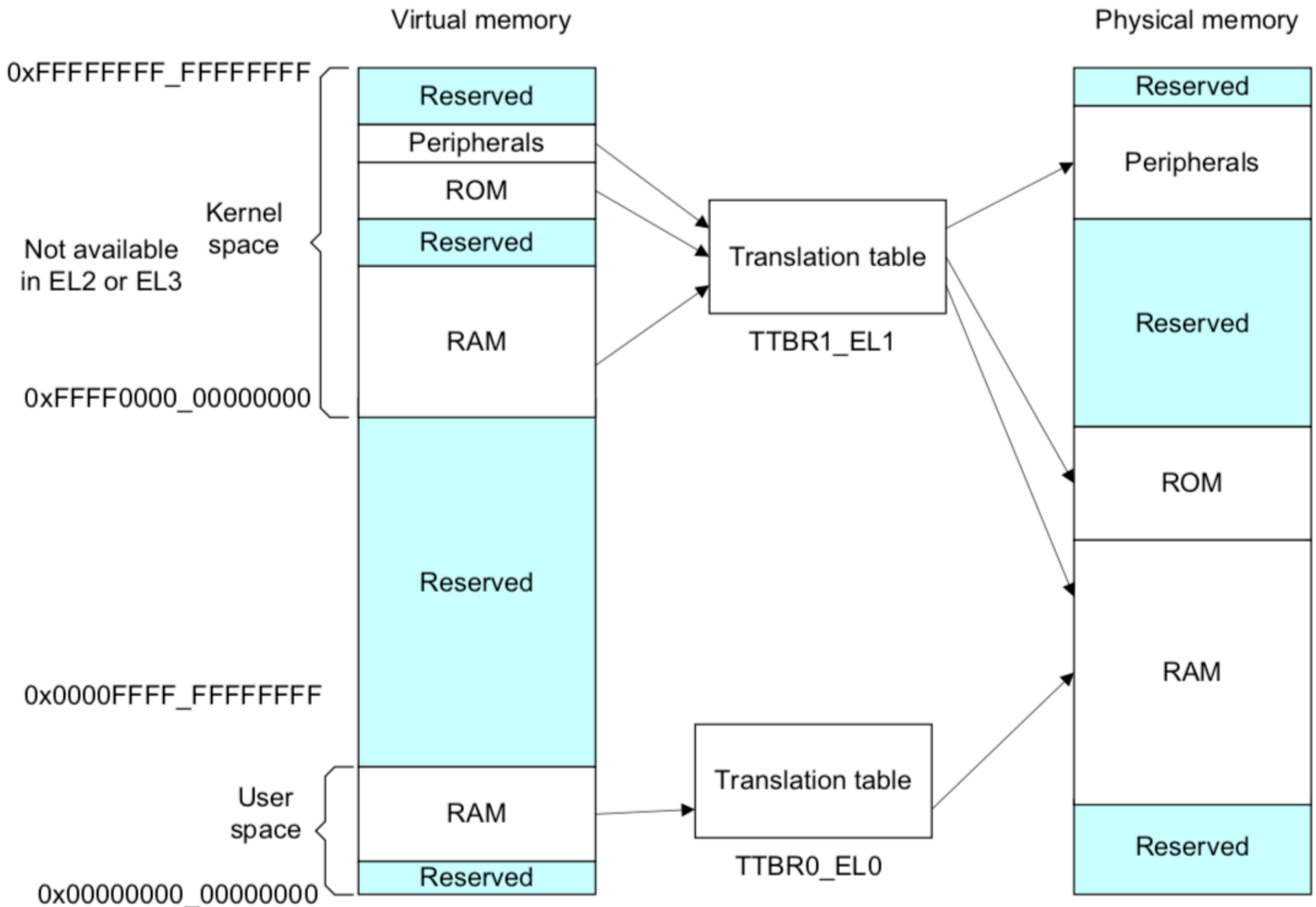
内存虚拟化

虚拟内存



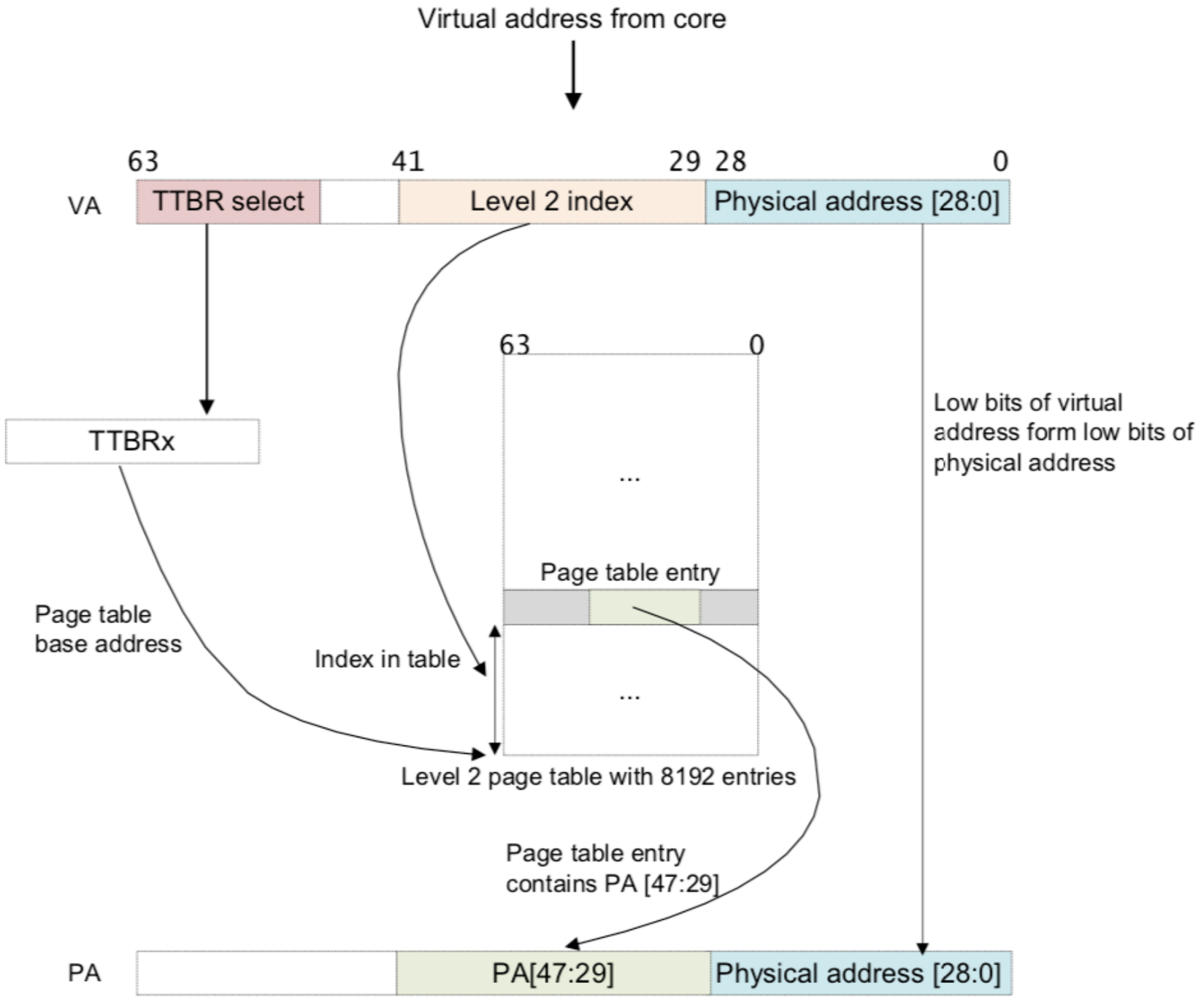
TLB: Translation Look-aside Buffer
MMU: Memory Management Unit
CPU: Central Processing Unit

虚拟内存



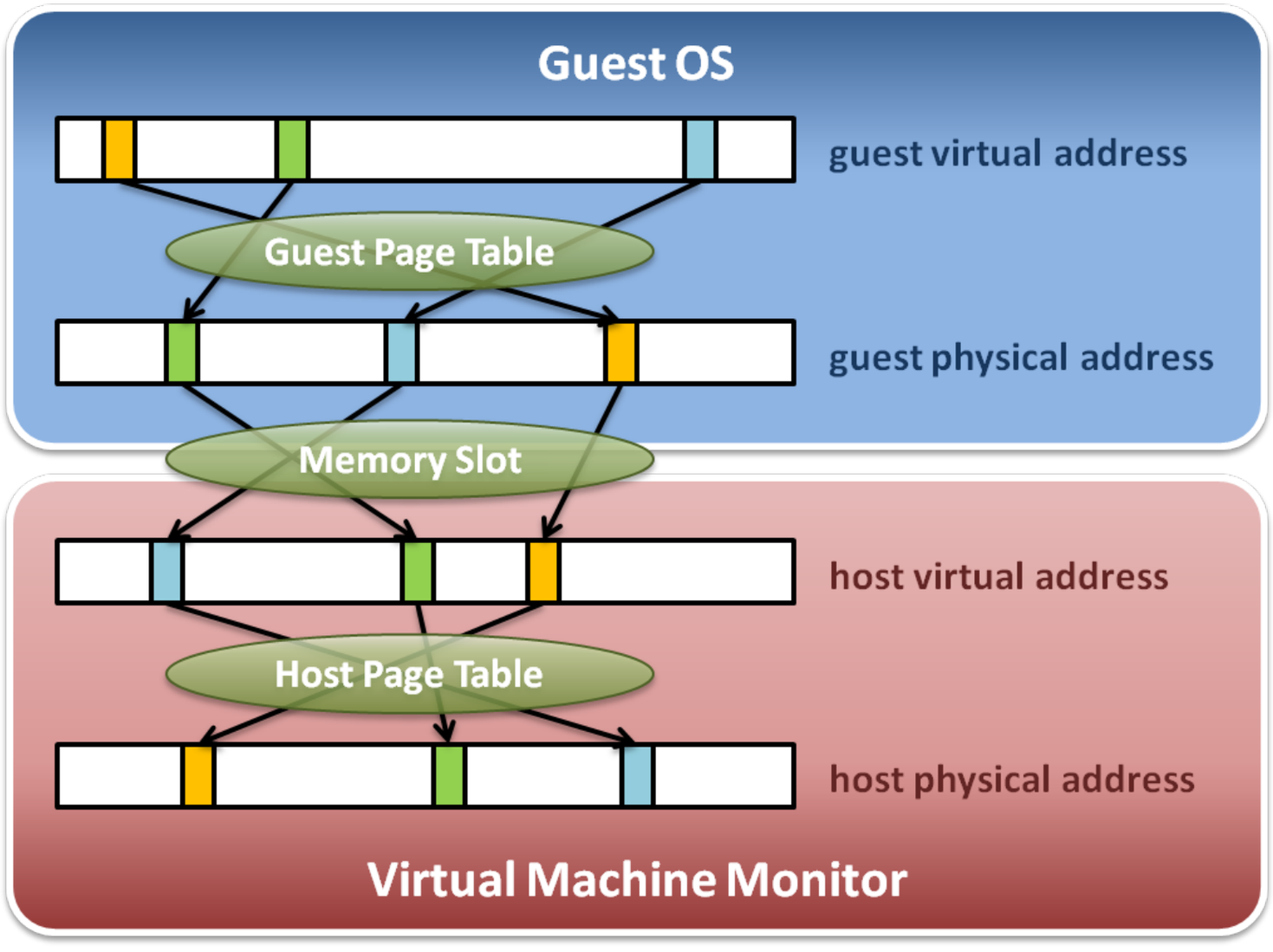
arm v8 address translation

虚拟内存



arm v8 address translation

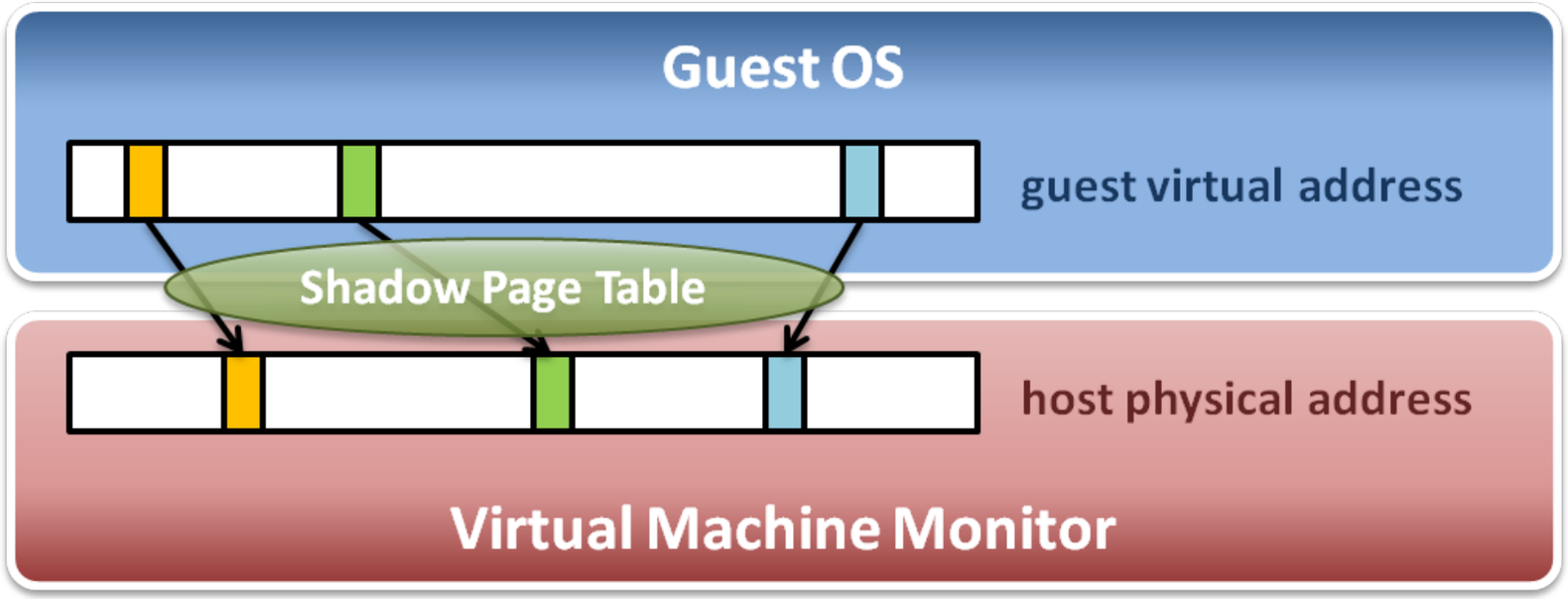
内存虚拟化



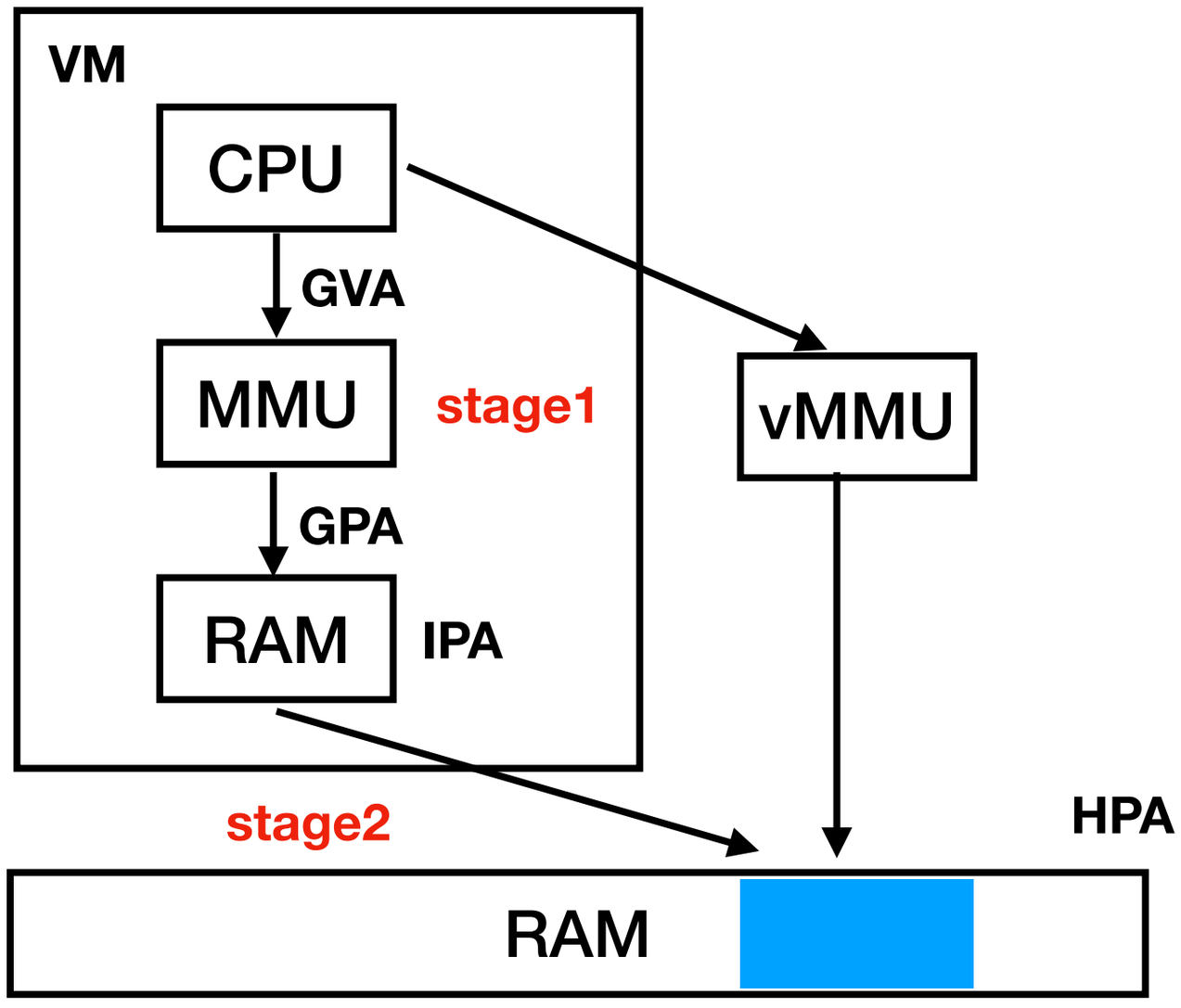
Low performance!

内存虚拟化

1. 影子页表
2. 硬件辅助的内存虚拟化

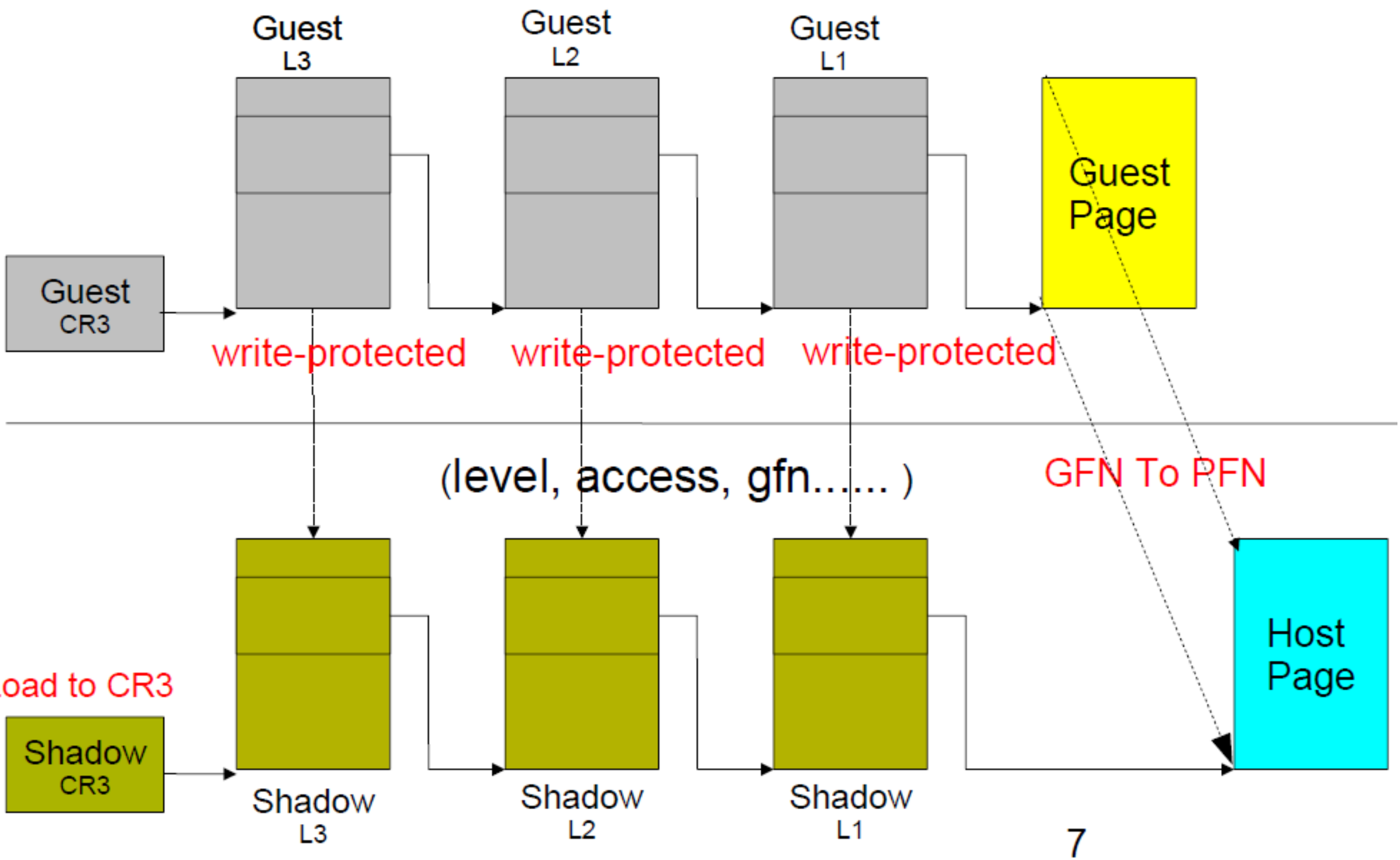


内存虚拟化

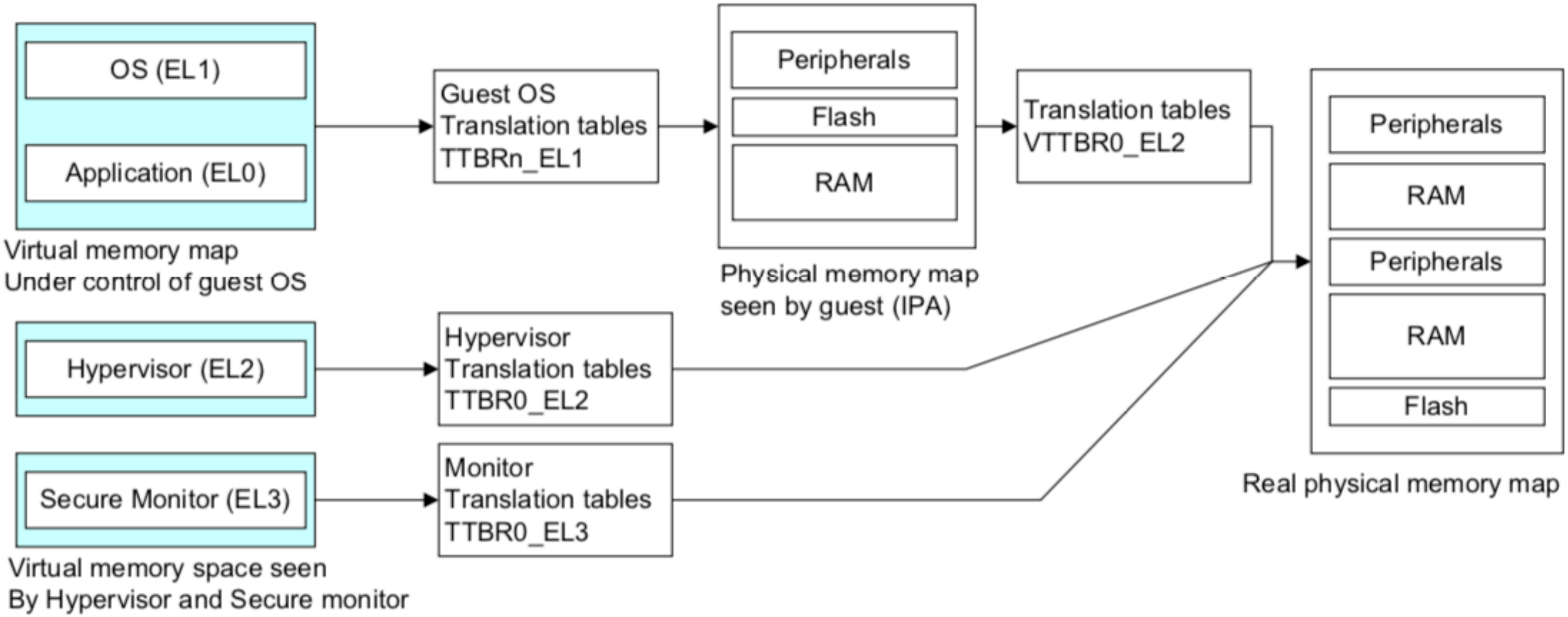


GVA: guest virtual address
GPA: guest physical address
IPA: immediate physical address
HPA: hypervisor physical address

内存虚拟化


Guest
L3
Guest
L2
Guest
L1
Guest
Page
Guest
CR3
write-protected
write-protected
write-protected
(level, access, gfn.....)
GFN To PFN
Host
Page
Load to CR3
Shadow
CR3
Shadow
L3
Shadow
L2
Shadow
L1

内存虚拟化


OS (EL1)
Application (EL0)
Virtual memory map
Under control of guest OS
Guest OS
Translation tables
TTBRn_EL1
Peripherals
Flash
RAM
Physical memory map
seen by guest (IPA)
Translation tables
VTTBR0_EL2
Peripherals
RAM
Peripherals
RAM
Flash
Real physical memory map
Hypervisor (EL2)
Hypervisor
Translation tables
TTBR0_EL2
Secure Monitor (EL3)
Monitor
Translation tables
TTBR0_EL3
Virtual memory space seen
By Hypervisor and Secure monitor

2. QEMU

QEMU

QEMU是一个设备模拟器(CPU)

```
flynn@francisco:~/qemu$ ls hw/
9pfs    block   display  i386    isa             microblaze  nvram      ppc     sh4      tpm         watchdog
acpi    bt      dma      ide     Kconfig         mips        openrisc   rdma    smbios   tricore     xen
adc     char    gpio     input   lm32            misc        pci        riscv   sparc    unicore32   xenpv
alpha   core    hppa     intc    m68k            moxie       pci-bridge s390x   sparc64  usb         xtensa
arm     cpu     hyperv   ipack   Makefile.objs   net         pci-host   scsi    ssi      vfio
audio   cris    i2c      ipmi    mem             nios2       pcmcia     sd      timer    virtio
```

```
allwinner-a10.c          exynos4_boards.c  Makefile.objs     omap1.c          smmuv3-internal.h  trace.h-timestamp
armsse.c                 fsl-imx25.c       mcimx6ul-evk.c    omap2.c          spitz.c            trace.o
armv7m.c                 fsl-imx31.c       mcimx7d-sabre.c   omap_sx1.c       stellaris.c        versatilepb.c
aspeed.c                 fsl-imx6.c        microbit.c        palm.c           stm32f205_soc.c    vexpress.c
aspeed_soc.c             fsl-imx6ul.c      mps2.c            pxa2xx.c         strongarm.c        virt-acpi-build.c
bcm2835_peripherals.c    fsl-imx7.c        mps2-tz.c         pxa2xx_gpio.c    strongarm.h        virt.c
bcm2836.c                gumstix.c         msf2-soc.c        pxa2xx_pic.c     sysbus-fdt.c       xilinx_zynq.c
boot.c                   highbank.c        msf2-som.c        raspi.c          tosa.c             xlnx-versal.c
collie.c                 imx25_pdk.c       musca.c           realview.c       trace.c            xlnx-versal-virt.c
cubieboard.c             integratorcp.c    musicpal.c        sabrelite.c      trace.c-timestamp  xlnx-zcu102.c
digic_boards.c           Kconfig           netduino2.c       smmu-common.c    trace.d            xlnx-zynqmp.c
digic.c                  kzm.c             nrf51_soc.c       smmu-internal.h  trace-events       z2.c
exynos4210.c             mainstone.c       nseries.c         smmuv3.c         trace.h
```

一个例子

1. 创建一个add.S文件

```
flynn@francisco:~/test/qemu_arm$ cat add.s
.text
start:                          @ Label, not really required
        mov    r0, #5           @ Load register r0 with the value 5
        mov    r1, #4           @ Load register r1 with the value 4
        add    r2, r1, r0       @ Add r0 and r1 and store in r2

stop:   b stop                  @ Infinite loop to stop execution
```

2. 交叉编译：

arm-linux-gnueabihf-as add.s -o add.o

3. 链接：

```
arm-linux-gnueabihf-ld -Ttext=0x0 -o add.elf add.o
```

4.制作bin二进制文件：

```
arm-linux-gnueabihf-objcopy -O binary add.elf add.bin
```

5.制作flash文件：

```
dd if=/dev/zero of=flash.bin bs=4096 count=4096
```

6.将add.bin文件拷贝到flash中：

```
dd if=add.bin of=flash.bin bs=4096 conv=notrunc
```

7. 启动qemu，模拟arm运行

```
qemu-system-arm -M connex -pflash flash.bin -nographic -serial /dev/null
```

一个例子

```
(qemu) info registers
R00=00000005 R01=00000004 R02=00000009 R03=00000000
R04=00000000 R05=00000000 R06=00000000 R07=00000000
R08=00000000 R09=00000000 R10=00000000 R11=00000000
R12=00000000 R13=00000000 R14=00000000 R15=0000000c
PSR=400001d3 -Z-- A svc32
FPSCR: 00000000
(qemu) xp /4iw 0x0
0x00000000:  e3a00005  mov       r0, #5
0x00000004:  e3a01004  mov       r1, #4
0x00000008:  e0812000  add       r2, r1, r0
0x0000000c:  eafffffe  b         #0xc
(qemu)
```

QEMU

```
flynn@francisco:~/qemu$ ls accel/kvm/
kvm-all.c  Makefile.objs  sev-stub.c  trace.c
```

KVM

KVM (for Kernel-based Virtual Machine) is a full virtualization solution for Linux on x86 hardware containing virtualization extensions (Intel VT or AMD-V). It consists of a loadable kernel module, **kvm.ko**, that provides the core virtualization infrastructure and a processor specific module, **kvm-intel.ko** or **kvm-amd.ko**.

```
flynn@francisco:/mnt/my-linux$ ls virt/kvm/
arm  async_pf.c  async_pf.h  coalesced_mmio.c  coalesced_mmio.h  eventfd.c  irqchip.c  Kconfig  kvm_main.c  vfio.c
```

```
flynn@francisco:/mnt/my-linux$ ls arch/x86/kvm/*.c
arch/x86/kvm/assigned-dev.c  arch/x86/kvm/i8254.c   arch/x86/kvm/irq.c         arch/x86/kvm/mmu.c      arch/x86/kvm/pmu_intel.c
arch/x86/kvm/cpuid.c         arch/x86/kvm/i8259.c   arch/x86/kvm/irq_comm.c    arch/x86/kvm/mtrr.c     arch/x86/kvm/svm.c
arch/x86/kvm/emulate.c       arch/x86/kvm/ioapic.c  arch/x86/kvm/lapic.c       arch/x86/kvm/pmu_amd.c  arch/x86/kvm/vmx.c
arch/x86/kvm/hyperv.c        arch/x86/kvm/iommu.c   arch/x86/kvm/mmu_audit.c   arch/x86/kvm/pmu.c      arch/x86/kvm/x86.c
```

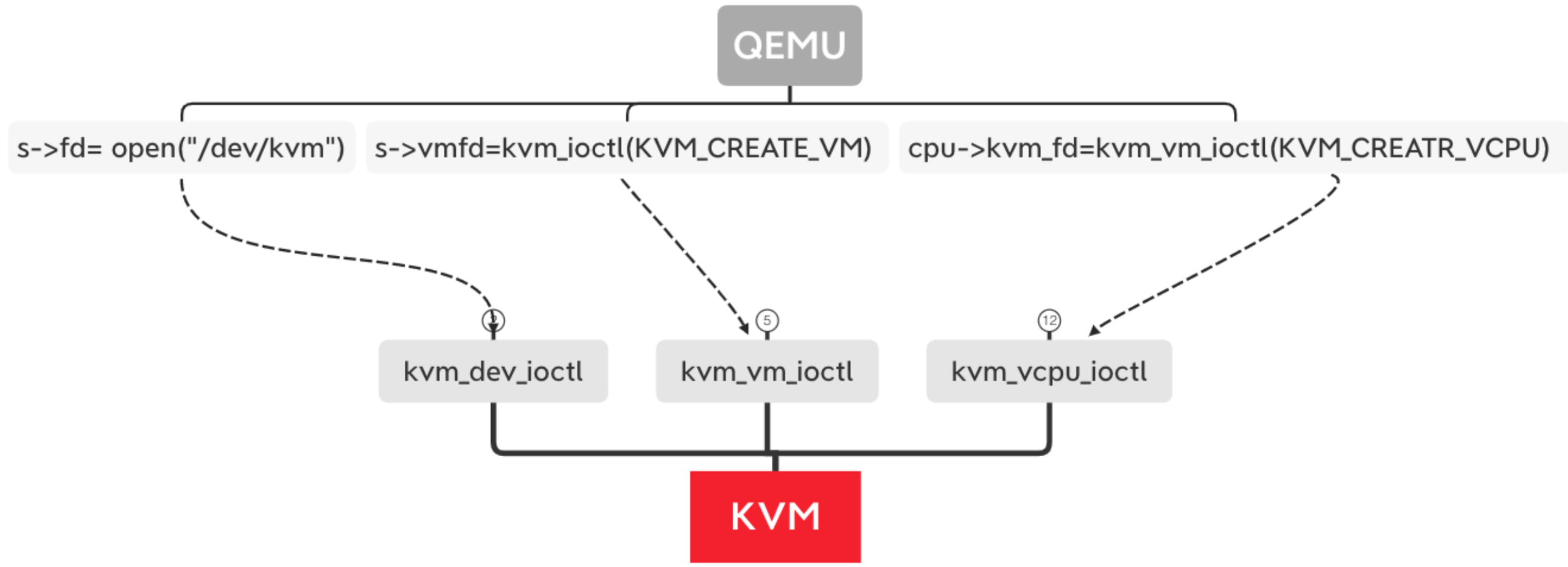
***pmu**:performance counter monitor*

KVM

```
1
2  ccflags-y += -Iarch/x86/kvm
3
4  CFLAGS_x86.o := -I.
5  CFLAGS_svm.o := -I.
6  CFLAGS_vmx.o := -I.
7
8  KVM := ../../../virt/kvm
9
10 kvm-y                  += $(KVM)/kvm_main.o $(KVM)/coalesced_mmio.o \
11                         $(KVM)/eventfd.o $(KVM)/irqchip.o $(KVM)/vfio.o
12 kvm-$(CONFIG_KVM_ASYNC_PF)  += $(KVM)/async_pf.o
13
14 kvm-y                   += x86.o mmu.o emulate.o i8259.o irq.o lapic.o \
15                          i8254.o ioapic.o irq_comm.o cpuid.o pmu.o mtrr.o \
16                          hyperv.o
17
18 kvm-$(CONFIG_KVM_DEVICE_ASSIGNMENT) += assigned-dev.o iommu.o
19 kvm-intel-y             += vmx.o pmu_intel.o
20 kvm-amd-y               += svm.o pmu_amd.o
21
22 obj-$(CONFIG_KVM)    += kvm.o
23 obj-$(CONFIG_KVM_INTEL) += kvm-intel.o
24 obj-$(CONFIG_KVM_AMD)    += kvm-amd.o
```

KVM

kvm_init
kvm_exit
kvm_sched_in
kvm_sched_out



QEMU & KVM

QEMU & KVM

L3

L0

QEMU

Create vm

Update Context

Create vcpu

Map memory

RUN/STOP

Raise IRQ

MMIO

I/O

Bad status

Backend driver

Handle user space tasks

Userspace signal

guest user space

IOCTL
/dev/kvm

User
space
task

signal

HOST
Kernel

Change guest state

IRQ

Suspend/Resume

Update status

Start/Stop

...

Guest state handling

Virt/kvm

In kernel I/O

VMMU

APIC

net

Kernel dependent task

Kernel
task

Host Kernel signal

Host IRQ

IRQ

Guest kernel space

Arch/x86/kvm

Save host state

Load guest state

Guest enter handler

VMCS

others

load host state

store guest state

Guest exit handler

Root

VM exit

Non-Root

VM entry

QEMU & KVM

```
static void *qemu_kvm_cpu_thread_fn(void *arg)
{
+-- 11 lines: CPUState *cpu = arg;----------------
    r = kvm_init_vcpu(cpu);
+-- 12 lines: if (r < 0) {------------------------
    do {
        if (cpu_can_run(cpu)) {
            r = kvm_cpu_exec(cpu);
            if (r == EXCP_DEBUG) {
                cpu_handle_guest_debug(cpu);
            }
        }
        qemu_wait_io_event(cpu);
    } while (!cpu->unplug || cpu_can_run(cpu));

    qemu_kvm_destroy_vcpu(cpu);
```

```
do {
    MemTxAttrs attrs;

    if (cpu->vcpu_dirty) {
        kvm_arch_put_registers(cpu, KVM_PUT_RUNTIME_STATE);
        cpu->vcpu_dirty = false;
    }

    kvm_arch_pre_run(cpu, run);
    if (atomic_read(&cpu->exit_request)) {
        DPRINTF("interrupt exit requested\n");
        /*
         * KVM requires us to reenter the kernel after IO ex
         * instruction emulation. This self-signal will ensu
         * leave ASAP again.
         */
        kvm_cpu_kick_self();
    }

    /* Read cpu->exit_request before KVM_RUN reads run->imme
     * Matching barrier in kvm_eat_signals.
     */
    smp_rmb();

    run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);

    attrs = kvm_arch_post_run(cpu, run);
```

QEMU & KVM

L3

L0

QEMU

Create vm

Update Context

Create vcpu

Map memory

RUN/STOP

Raise IRQ

MMIO

I/O

Bad status

Backend driver

Handle user space tasks

Userspace signal

guest user space

IOCTL
/dev/kvm

User
space
task

signal

HOST
Kernel

Change guest state

IRQ

Suspend/Resume

Update status

Start/Stop

...

Guest state handling

Virt/kvm

In kernel I/O

VMMU

APIC

net

Kernel dependent task

Kernel
task

Host Kernel signal

Host IRQ

IRQ

Guest kernel space

Arch/x86/kvm

Save host state

Load guest state

Guest enter handler

VMCS

others

load host state

store guest state

Guest exit handler

Root

VM exit

Non-Root

VM entry

QEMU & KVM

```
static int vcpu_run(struct kvm_vcpu *vcpu)
{
    int r;
    struct kvm *kvm = vcpu->kvm;

    vcpu->srcu_idx = srcu_read_lock(&kvm->srcu);
    vcpu->arch.l1tf_flush_l1d = true;

    for (;;) {
        if (kvm_vcpu_running(vcpu)) {
            r = vcpu_enter_guest(vcpu);
        } else {
            r = vcpu_block(kvm, vcpu);
        }

        if (r <= 0)
            break;

        kvm_clear_request(KVM_REQ_PENDING_TIMER, vcpu);
        if (kvm_cpu_has_pending_timer(vcpu))
            kvm_inject_pending_timer_irqs(vcpu);

        if (dm_request_for_irq_injection(vcpu) &&
            kvm_vcpu_ready_for_interrupt_injection(vcpu)) {
            r = 0;
            vcpu->run->exit_reason = KVM_EXIT_IRQ_WINDOW_OPEN;
            ++vcpu->stat.request_irq_exits;
            break;
        }

        kvm_check_async_pf_completion(vcpu);

        if (signal_pending(current)) {
            r = -EINTR;
            vcpu->run->exit_reason = KVM_EXIT_INTR;
            ++vcpu->stat.signal_exits;
            break;
```

```
static int vcpu_enter_guest(struct kvm_vcpu *vcpu)
{
    int r;
    bool req_int_win =
        dm_request_for_irq_injection(vcpu) &&
        kvm_cpu_accept_dm_intr(vcpu);

    bool req_immediate_exit = false;

    if (kvm_request_pending(vcpu)) {
+-- 92 lines: if (kvm_check_request(KVM_REQ_GET_VMCS12_PAG

    if (kvm_check_request(KVM_REQ_EVENT, vcpu) || req_int_
        ++vcpu->stat.req_event;
        kvm_apic_accept_events(vcpu);
+-- 38 lines: if (vcpu->arch.mp_state == KVM_MP_STATE_INIT
    r = kvm_mmu_reload(vcpu);
    if (unlikely(r)) {
        goto cancel_injection;
    }

    preempt_disable();

    kvm_x86_ops->prepare_guest_switch(vcpu);
+-- 67 lines: ----------------------------------------------
    kvm_x86_ops->run(vcpu);
```

QEMU & KVM

L3

L0

QEMU

Create vm

Update Context

Create vcpu

Map memory

RUN/STOP

Raise IRQ

MMIO

I/O

Bad status

Backend driver

Handle user space tasks

Userspace signal

guest user space

IOCTL
/dev/kvm

User
space
task

signal

HOST
Kernel

Change guest state

IRQ

Suspend/Resume

Update status

Start/Stop

...

Guest state handling

Virt/kvm

In kernel I/O

VMMU

APIC

net

Kernel dependent task

Kernel
task

Host Kernel signal

Host IRQ

IRQ

Guest kernel space

Arch/x86/kvm

Save host state

Load guest state

Guest enter handler

VMCS

others

load host state

store guest state

Guest exit handler

Root

VM exit

Non-Root

VM entry

QEMU & KVM

```
ic void __noclone vmx_vcpu_run(struct kvm_vcpu *vcpu)

13 lines: struct vcpu_vmx *vmx = to_vmx(vcpu);-----------
if (vmx->ple_window_dirty) {
    vmx->ple_window_dirty = false;
    vmcs_write32(PLE_WINDOW, vmx->ple_window);
}
53 lines: -------------------------------------------------
asm(
    /* Store host registers */
    "push %%" _ASM_DX "; push %%" _ASM_BP ";"
    "push %%" _ASM_CX " \n\t" /* placeholder for guest rc
    "push %%" _ASM_CX " \n\t"
    "cmp %%" _ASM_SP ", %c[host_rsp](%0) \n\t"
    "je 1f \n\t"
    "mov %%" _ASM_SP ", %c[host_rsp](%0) \n\t"
    /* Avoid VMWRITE when Enlightened VMCS is in use */
    "test %%" _ASM_SI ", %%" _ASM_SI " \n\t"
    "jz 2f \n\t"
    "mov %%" _ASM_SP ", (%%" _ASM_SI ") \n\t"
    "jmp 1f \n\t"
    "2: \n\t"
    __ex(ASM_VMX_VMWRITE_RSP_RDX) "\n\t"
    "1: \n\t"
    /* Reload cr2 if changed */
    "mov %c[cr2](%0), %%" _ASM_AX " \n\t"
    "mov %%cr2, %%" _ASM_DX " \n\t"
    "cmp %%" _ASM_AX ", %%" _ASM_DX " \n\t"
    "je 3f \n\t"
    "mov %%" _ASM_AX", %%cr2 \n\t"
    "3: \n\t"
    /* Check if vmlaunch of vmresume is needed */
    "cmpl $0, %c[launched](%0) \n\t"
    /* Load guest registers.  Don't clobber flags. */
    "mov %c[rax](%0), %%" _ASM_AX " \n\t"
    "mov %c[rbx](%0), %%" _ASM_BX " \n\t"
    "mov %c[rdx](%0), %%" _ASM_DX " \n\t"
    "mov %c[rsi](%0), %%" _ASM_SI " \n\t"
```

```
                                                    cmpl $0, %c[launched](%0) \n\t
                                                    /* Load guest registers.  Don't clobber flags.
                                                    "mov %c[rax](%0), %%" _ASM_AX " \n\t"
                                                    "mov %c[rbx](%0), %%" _ASM_BX " \n\t"
                                                    "mov %c[rdx](%0), %%" _ASM_DX " \n\t"
                                                    "mov %c[rsi](%0), %%" _ASM_SI " \n\t"
                                                    "mov %c[rdi](%0), %%" _ASM_DI " \n\t"
                                                    "mov %c[rbp](%0), %%" _ASM_BP " \n\t"
#ifdef CONFIG_X86_64                                #ifdef CONFIG_X86_64
                                                    "mov %c[r8](%0),  %%r8  \n\t"
                                                    "mov %c[r9](%0),  %%r9  \n\t"
                                                    "mov %c[r10](%0), %%r10 \n\t"
                                                    "mov %c[r11](%0), %%r11 \n\t"
                                                    "mov %c[r12](%0), %%r12 \n\t"
                                                    "mov %c[r13](%0), %%r13 \n\t"
                                                    "mov %c[r14](%0), %%r14 \n\t"
                                                    "mov %c[r15](%0), %%r15 \n\t"
#endif                                              #endif
                                                    "mov %c[rcx](%0), %%" _ASM_CX " \n\t" /* kills

                                                    /* Enter guest mode */
                                                    "jne 1f \n\t"
                                                    __ex(ASM_VMX_VMLAUNCH) "\n\t"
                                                    "jmp 2f \n\t"
                                                    "1: " __ex(ASM_VMX_VMRESUME) "\n\t"
                                                    "2: "
                                                    /* Save guest registers, load host registers, k
                                                    "mov %0, %c[wordsize](%%" _ASM_SP ") \n\t"
                                                    "pop %0 \n\t"
                                                    "setbe %c[fail](%0)\n\t"
                                                    "mov %%" _ASM_AX ", %c[rax](%0) \n\t"
                                                    "mov %%" _ASM_BX ", %c[rbx](%0) \n\t"
                                                    __ASM_SIZE(pop) " %c[rcx](%0) \n\t"
                                                    "mov %%" _ASM_DX ", %c[rdx](%0) \n\t"
                                                    "mov %%" _ASM_SI ", %c[rsi](%0) \n\t"
                                                    "mov %%" _ASM_DI ", %c[rdi](%0) \n\t"
                                                    "mov %%" _ASM_BP ", %c[rbp](%0) \n\t"
```

QEMU & KVM

L3

L0

QEMU

Create vm

Update Context

Create vcpu

Map memory

RUN/STOP

Raise IRQ

MMIO

I/O

Bad status

Backend driver

Handle user space tasks

Userspace signal

guest user space

IOCTL
/dev/kvm

User
space
task

signal

HOST
Kernel

Change guest state

IRQ

Suspend/Resume

Update status

Start/Stop

...

Guest state handling

Virt/kvm

In kernel I/O

VMMU

APIC

net

Kernel dependent task

Kernel
task

Host Kernel signal

Host IRQ

IRQ

Guest kernel space

Arch/x86/kvm

Save host state

Load guest state

Guest enter handler

VMCS

others

load host state

store guest state

Guest exit handler

Root

VM exit

Non-Root

VM entry

Thank you