

系统虚拟化

陈海波 / 夏虞斌

上海交通大学并行与分布式系统研究所

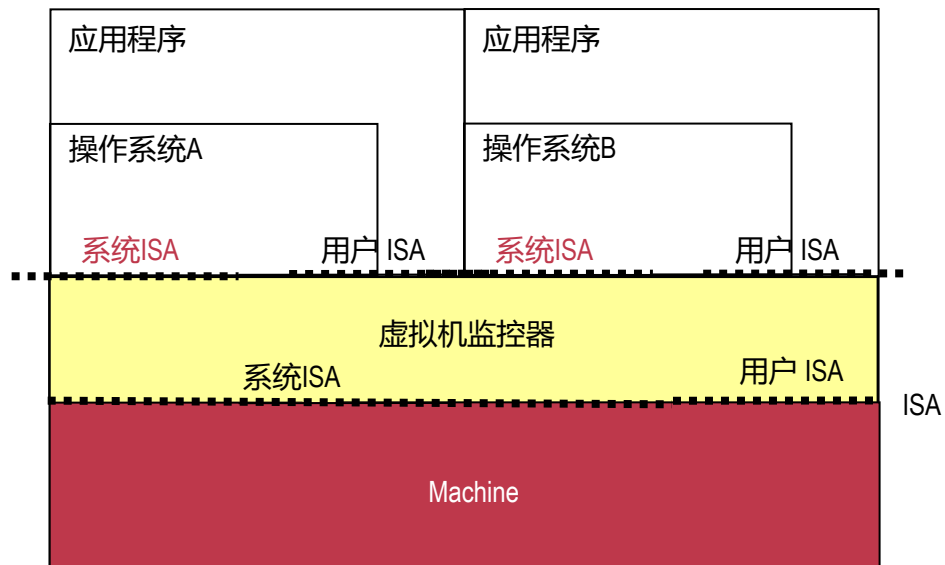
<https://ipads.se.sjtu.edu.cn>

版权声明

- 本内容版权归**上海交通大学并行与分布式系统研究所**所有
- 使用者可以将全部或部分本内容免费用于非商业用途
- 使用者在使用全部或部分本内容时请注明来源：
 - 内容来自：上海交通大学并行与分布式系统研究所+材料名字
- 对于不遵守此声明或者其他违法使用本内容者，将依法保留追究权
- 本内容的发布采用 Creative Commons Attribution 4.0 License
 - 完整文本：<https://creativecommons.org/licenses/by/4.0/legalcode>

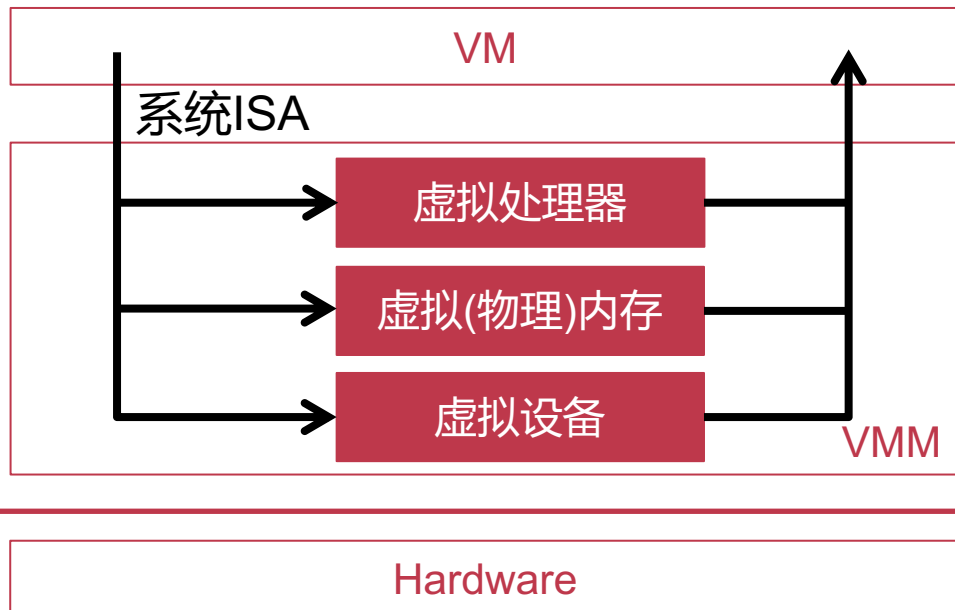
Review: 系统ISA

- 读写敏感寄存器
 - sctrl_el1、ttbr0_el1/ttbr1_el1...
- 控制处理器行为
 - 例如: WFI(陷入低功耗状态)
- 控制虚拟/物理内存
 - 打开、配置、安装页表
- 控制外设
 - DMA、中断



Review: 系统虚拟化的流程

- **第一步**
 - 捕捉所有系统ISA并陷入(Trap)
- **第二步**
 - 由具体指令实现相应虚拟化
 - 控制虚拟处理器行为
 - 控制虚拟内存行为
 - 控制虚拟设备行为
- **第三步**
 - 回到虚拟机继续执行



Review: 系统虚拟化技术

- **处理器虚拟化**
 - 捕捉系统ISA
 - 控制虚拟处理器的行为
- **内存虚拟化**
 - 提供“假”物理内存的抽象
- **设备虚拟化**
 - 提供虚拟的I/O设备

Review: ARM不是严格的可虚拟化架构

- 在ARM中：不是所有敏感指令都属于特权指令
- 例子: **CPSID/CPSIE**指令
 - CPSID和CPSIE分别可以关闭和打开中断
 - 内核态执行：PSTATE.{A, I, F} 可以被CPS指令修改
 - 在用户态执行：CPS 被当做NOP指令，不产生任何效果
 - 不是特权指令

Review: 如何处理这些不会下陷的敏感指令？

处理这些不会下陷的敏感指令，使得虚拟机中的操作系统能够运行在用户态（EL-0）

- 方法1：解释执行
- 方法2：二进制翻译
- 方法3：半虚拟化
- 方法4：硬件虚拟化（改硬件）

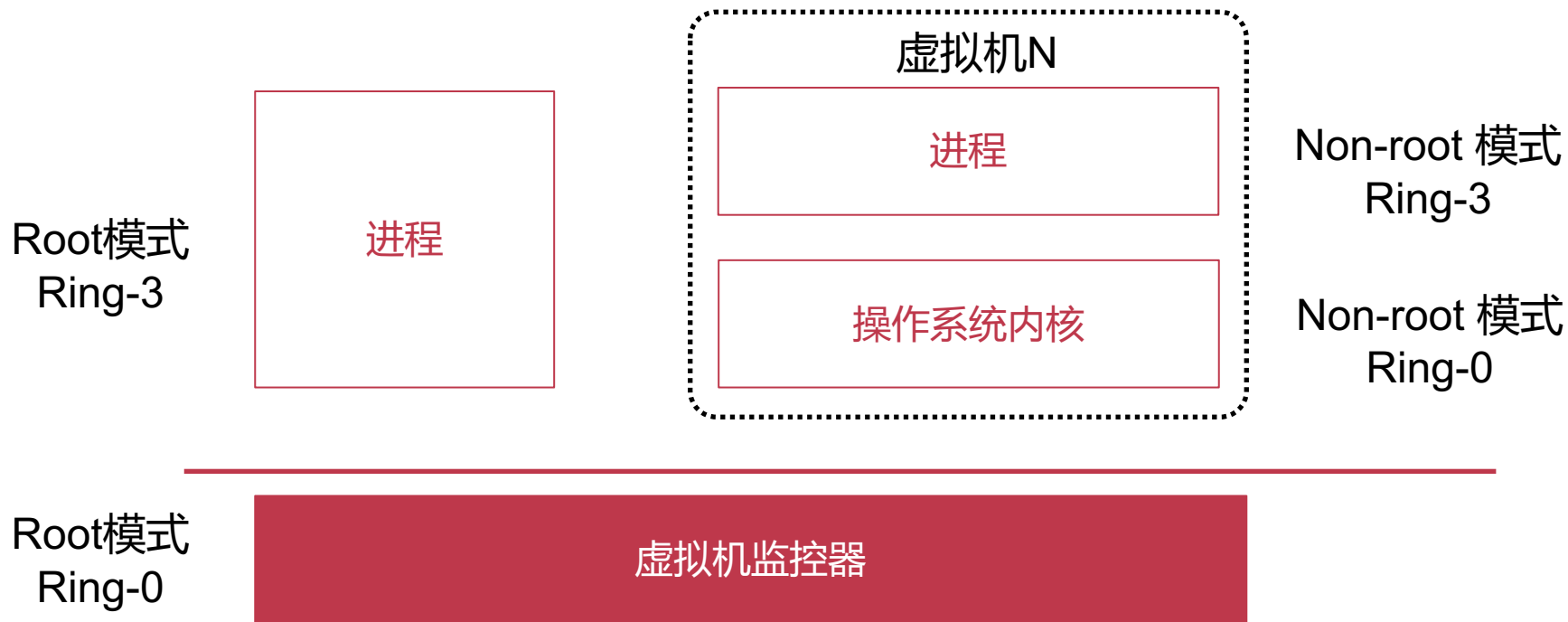
Review: 方法4：硬件虚拟化

- **x86和ARM都引入了全新的虚拟化特权级**
- **x86引入了root模式和non-root模式**
 - Intel推出了VT-x硬件虚拟化扩展
 - Root模式是最高特权级别，控制物理资源
 - VMM运行在root模式，虚拟机运行在non-root模式
 - 两个模式内都有4个特权级别：Ring0~Ring3
- **ARM引入了EL2**
 - VMM运行在EL2
 - EL2是最高特权级别，控制物理资源
 - VMM的操作系统和应用程序分别运行在EL1和EL0



INTEL VT-X

VT-x的处理器虚拟化



Virtual Machine Control Structure (VMCS)

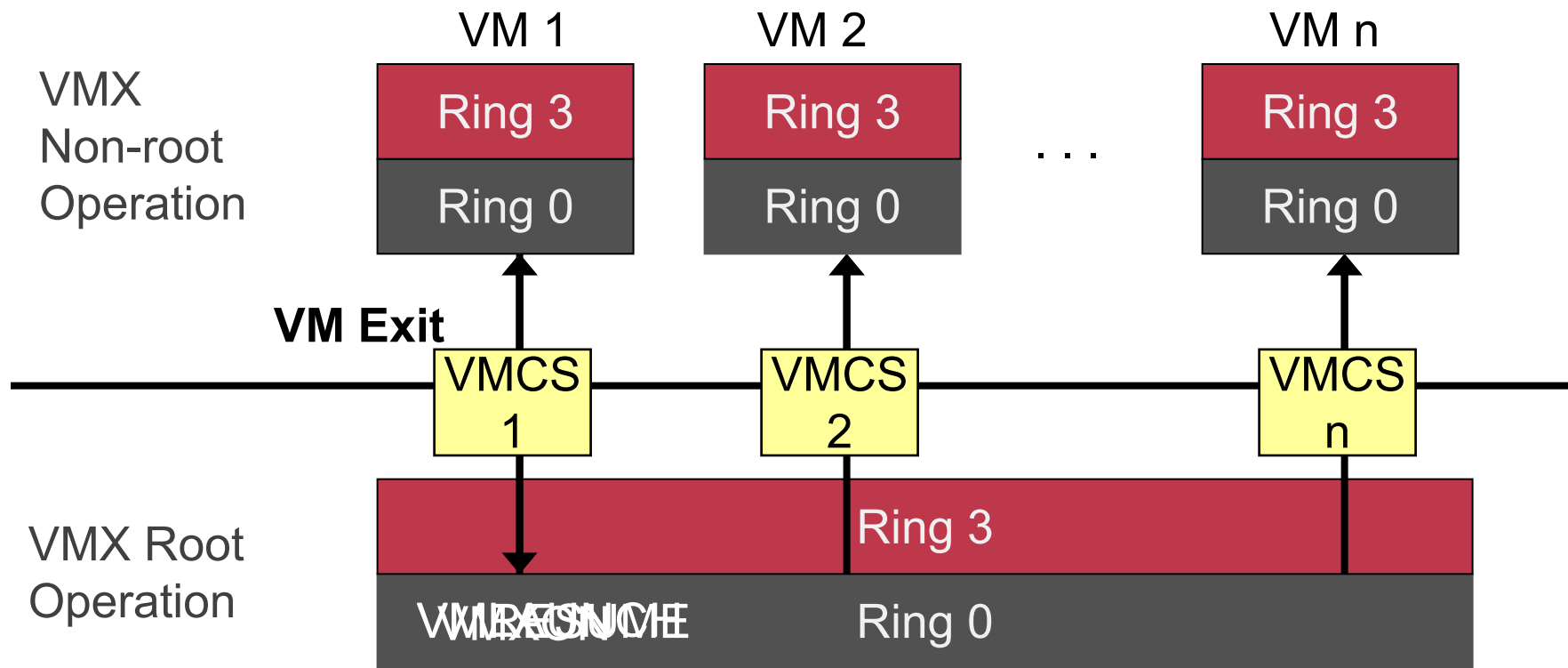
- **VMM提供给硬件的内存页（4KB）**
 - 记录与当前VM运行相关的所有状态
- **VM Entry**
 - 硬件自动将当前CPU中的VMM状态保存至VMCS
 - 硬件自动从VMCS中加载VM状态至CPU中
- **VM Exit**
 - 硬件自动将当前CPU中的VM状态保存至VMCS
 - 硬件自动从VMCS加载VMM状态至CPU中

VT-x VMCS的内容

- 包含6个部分

- Guest-state area: 发生VM exit时，CPU的状态会被硬件自动保存至该区域；发生VM Entry时，硬件自动从该区域加载状态至CPU中
- Host-state area：发生VM exit时，硬件自动从该区域加载状态至CPU中；发生VM Entry时，CPU的状态会被自动保存至该区域
- VM-execution control fields：控制Non-root模式中虚拟机的行为
- VM-exit control fields：控制VM exit的行为
- VM-entry control fields：控制VM entry的行为
- VM-exit information fields：VM Exit的原因和相关信息（只读区域）

VT-x的执行过程



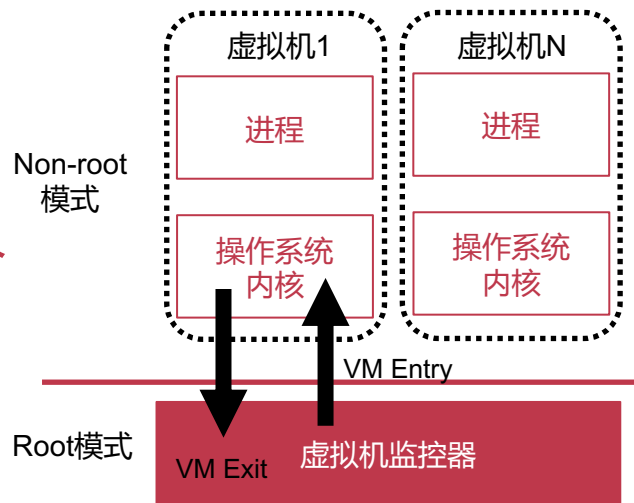
x86中的VM Entry和VM Exit

- **VM Entry**

- 从VMM进入VM
- 从Root模式切换到Non-root模式
- 第一次启动虚拟机时使用**VMLAUNCH**指令
- 后续的VM Entry使用**VMRESUME**指令

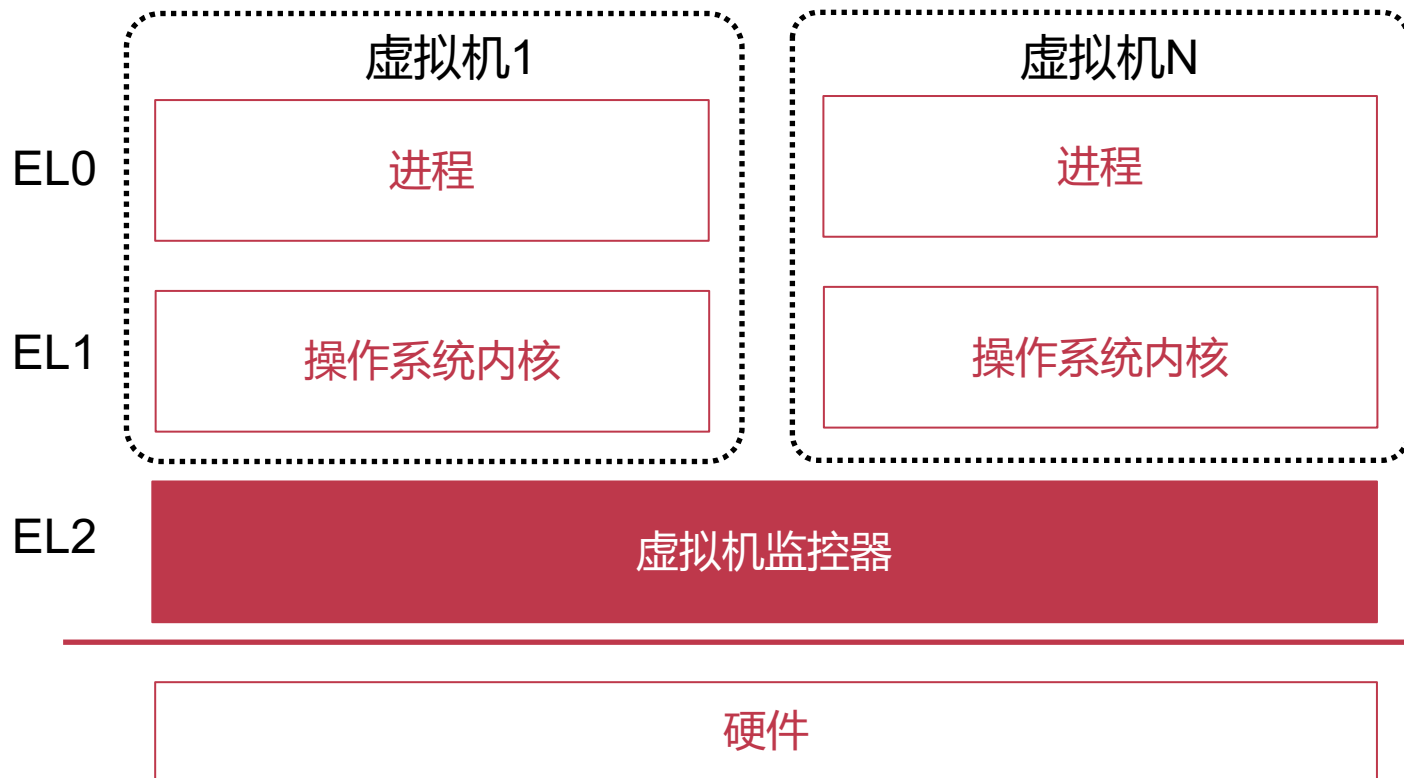
- **VM Exit**

- 从VM回到VMM
- 从Non-root模式切换到Root模式
- 虚拟机执行敏感指令或发生事件(如外部中断)



ARM的虚拟化技术

ARM的处理器虚拟化



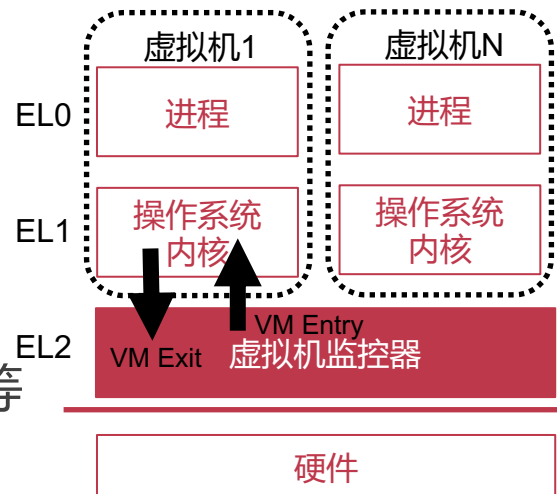
ARM的VM Entry和VM Exit

- VM Entry

- 使用ERET指令从VMM进入VM
- 在进入VM之前，VMM需要**主动**加载VM状态
 - VM内状态：通用寄存器、系统寄存器、
 - VM的控制状态：HCR_EL2、VTTBR_EL2等

- VM Exit

- 虚拟机执行敏感指令或收到中断等
- 以Exception、IRQ、FIQ的形式回到VMM
 - 调用VMM记录在vbar_el2中的相关处理函数
- 下陷第一步：VMM**主动**保存所有VM的状态



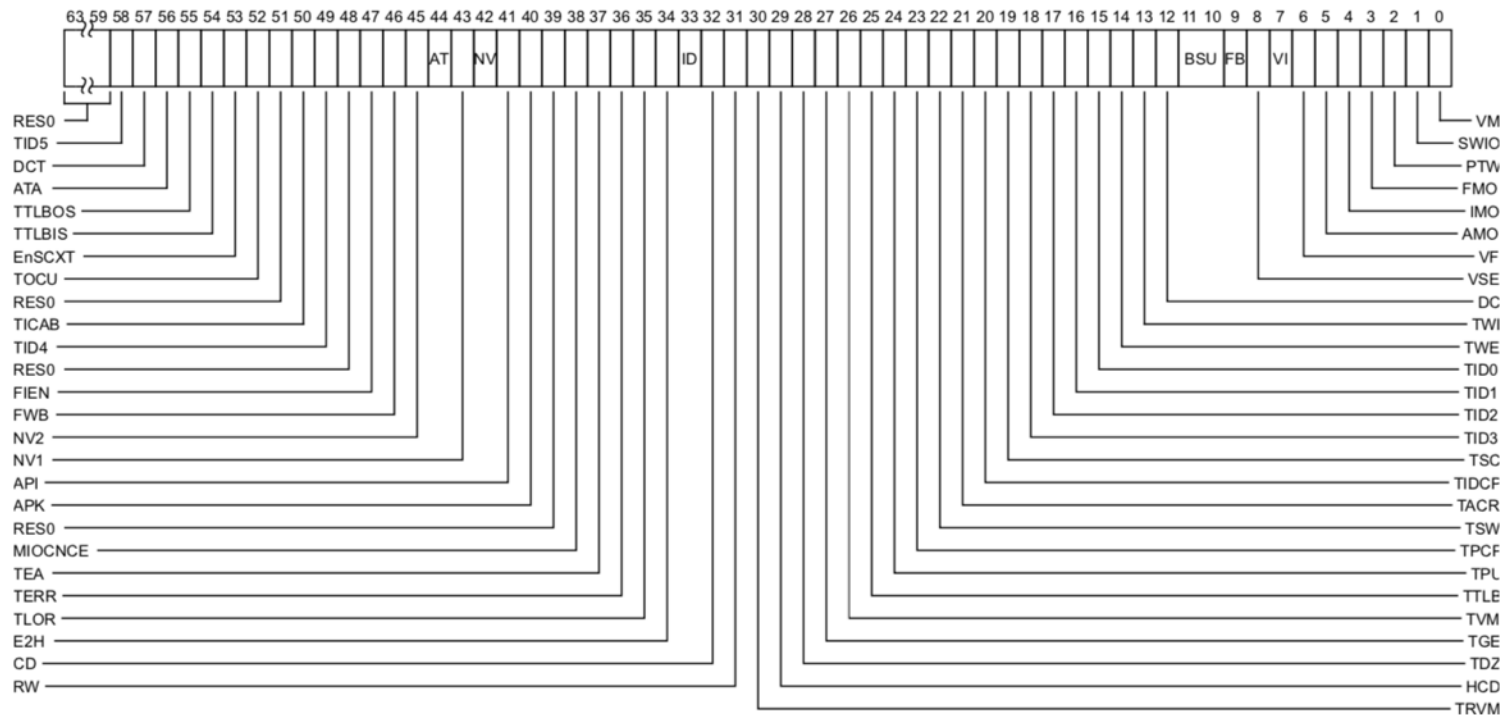
ARM硬件虚拟化的新功能

- **ARM中没有VMCS**
- **VM能直接控制EL1和EL0的状态**
 - 自由地修改PSTATE(VMM不需要捕捉CPS指令)
 - 可以读写TTBR0_EL1/SCTRL_EL1/TCR_EL1等寄存器
- **VM Exit时VMM仍然可以直接访问VM的EL0和EL1寄存器**
- **思考题1：为什么ARM中可以不需要VMCS？**
- **思考题2：ARM中没有VMCS，对于VMM的设计和实现来说有什么优缺点？**

HCR_EL2寄存器简介

- **HCR_EL2 : VMM控制VM行为的系统寄存器**
 - VMM有选择地决定VM在某些情况时下陷
 - 和VT-x VMCS中VM-execution control area类似
- **在VM Entry之前设置相关位，控制虚拟机行为**
 - TRVM(32位)和TVM(26位): VM读写内存控制寄存器是否下陷，例如SCTRL_EL1、TTBR0_EL1
 - TWE(14位)和TWI(13位)：执行WFE和WFI指令是否下陷
 - AMO(6位)/IMO(5位)/FMO(4位)：Exception/IRQ/FIQ是否下陷
 - VM(0位): 是否打开第二阶段地址翻译

HCR_EL2寄存器简介

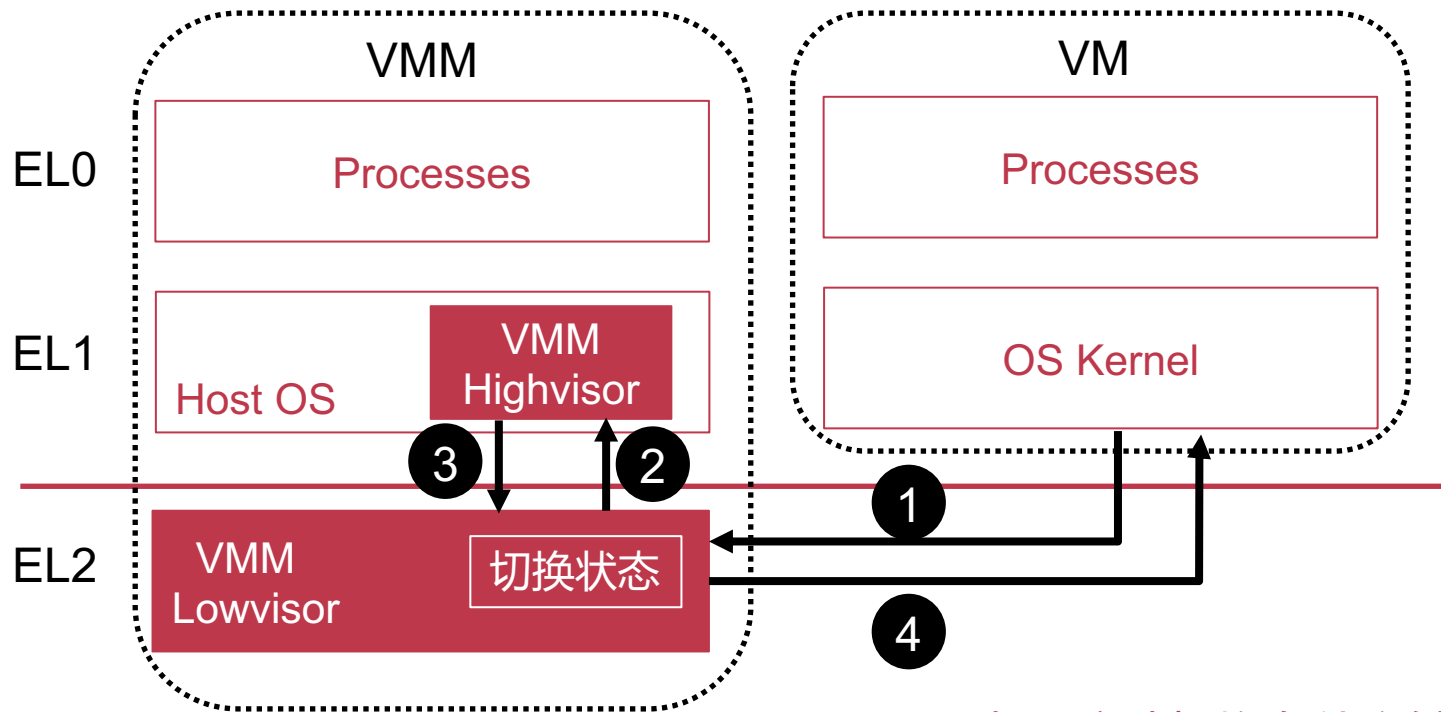


ARM硬件虚拟化发展

- **ARMv8.0**

- 增加EL2¹特权级
- EL2只能运行VMM，不能运行一般操作系统内核
 - OS一般只使用EL1的寄存器，在EL2中不存在对应的寄存器
 - EL1 : TTBR0_EL1、TTBR1_EL1
 - EL2 : TTBR_EL2
 - EL2不能与EL0共享内存
- 因此：无法在EL2中运行Type-2虚拟机监控器的Host OS
 - 或者说，Host OS需要大量修改才能运行在EL2

ARMv8.0中的Type-2 VMM架构



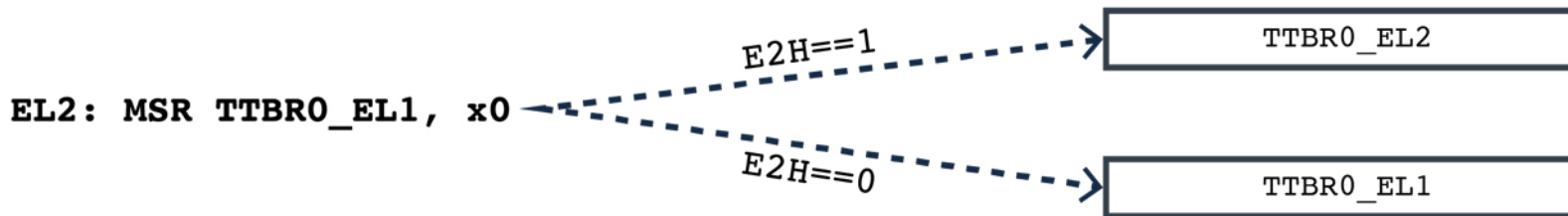
思考：这种架构有什么缺点？

部件	代码行
KVM/ARM Highvisor	5094
KVM/ARM Lowvisor	718

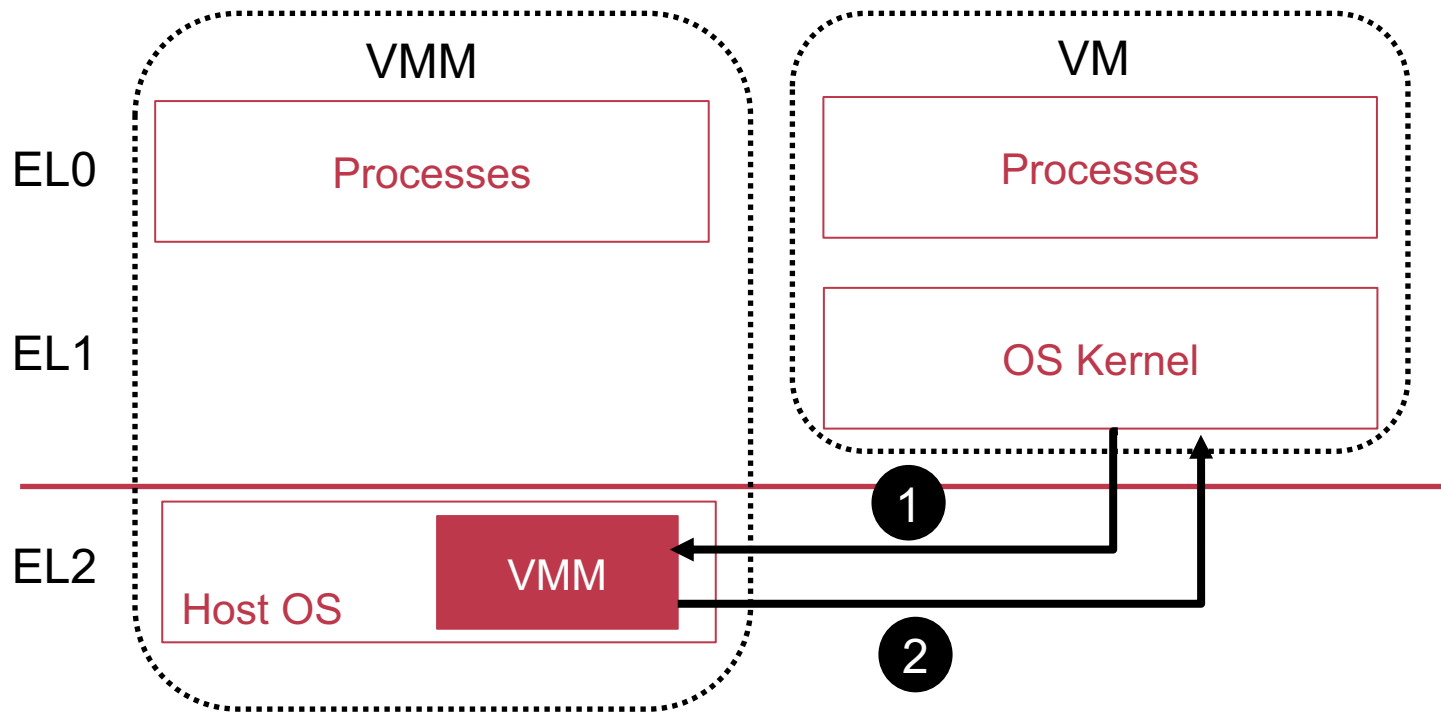
ARMv8.1中的Type-2 VMM架构

- **ARMv8.1**

- 推出Virtualization Host Extensions(VHE), 在HCR_EL2.E2H打开
 - 寄存器映射
 - 允许与EL0共享内存
- 使EL2中可直接运行未修改的操作系统内核 (Host OS)



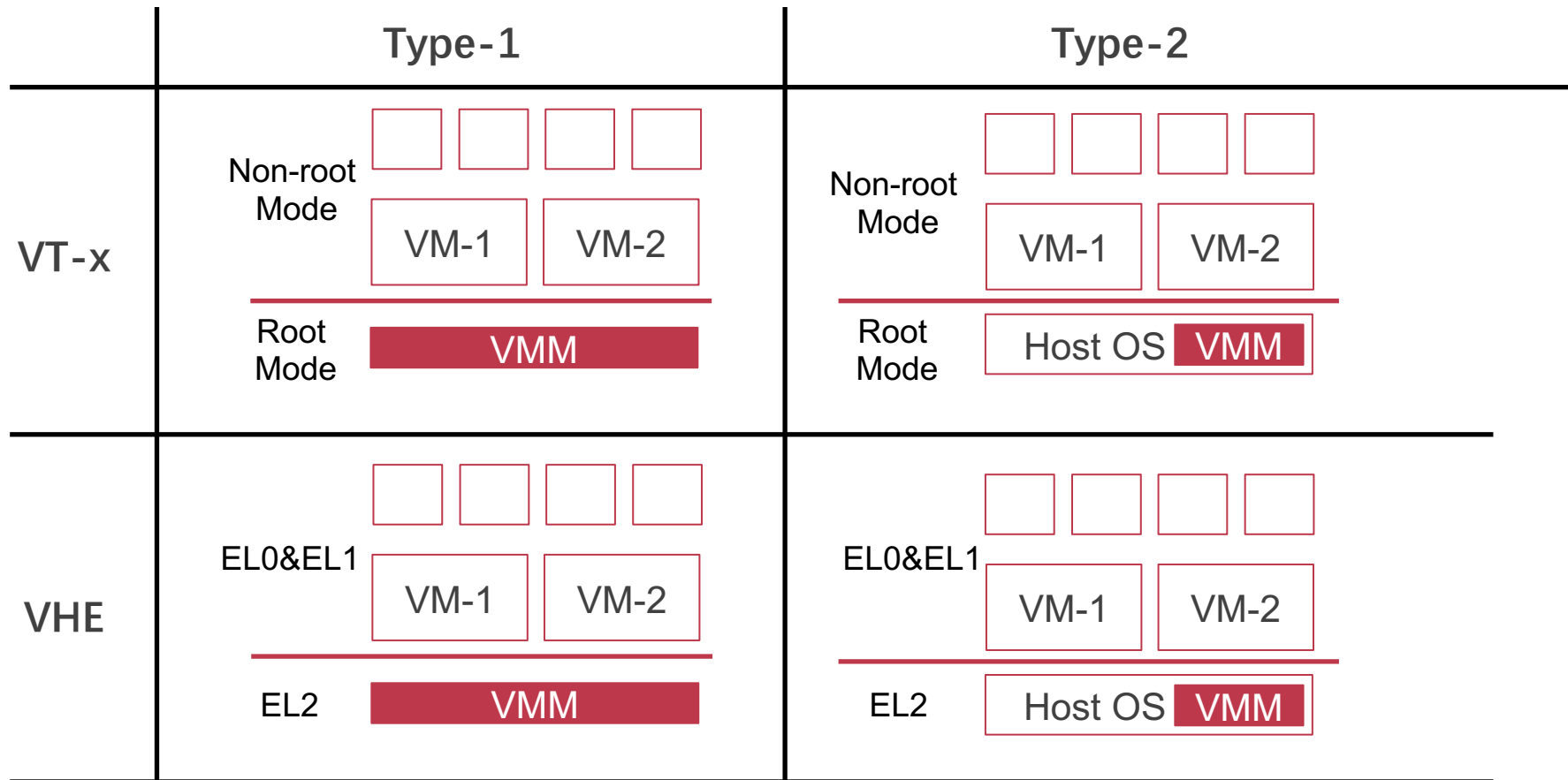
ARMv8.1中的Type-2 VMM架构



VT-x和VHE对比

	VT-x	VHE
新特权级	Root和Non-root	EL2
是否有VMCS ?	是	否
VM Entry/Exit时硬件自动保存状态 ?	是	否
是否引入新的指令 ?	是(多)	是(少)
是否引入新的系统寄存器?	否	是(多)
是否有扩展页表(第二阶段页表)?	是	是

Type-1和Type-2在VT-x和VHE下架构



案例：QEMU/KVM

QEMU发展历史



- **2003年，法国程序员Fabrice Bellard发布了QEMU 0.1版本**
 - 目标是在非x86机器上使用动态二进制翻译技术模拟x86机器
- **2003-2006年**
 - 能模拟出多种不同架构的虚拟机，包括S390、ARM、MIPS、SPARC等
 - 在这阶段，QEMU一直使用**软件方法**进行模拟
 - 如二进制翻译技术

QEMU发展历史

Fabrice Bellard [fabrice.bellard at free.fr](mailto:fabrice.bellard@free.fr)

Sun Mar 23 14:46:47 CST 2003

- Previous message: [SPI_GETGRADIENTCAPTIONS](#)
- Next message: [\[announce\] QEMU x86 emulator version 0.1](#)
- Messages sorted by: [\[_date_\]](#) [\[_thread_\]](#) [\[_subject_\]](#) [\[_author_\]](#)

Hi,

The first release of the QEMU x86 emulator is available at <http://bellard.org/gemu/>. QEMU achieves a fast user space Linux x86 emulation on x86 and PowerPC Linux hosts by using dynamic translation. Its main goal is to be able to run the Wine project on non-x86 architectures.

Fabrice.

KVM发展历史

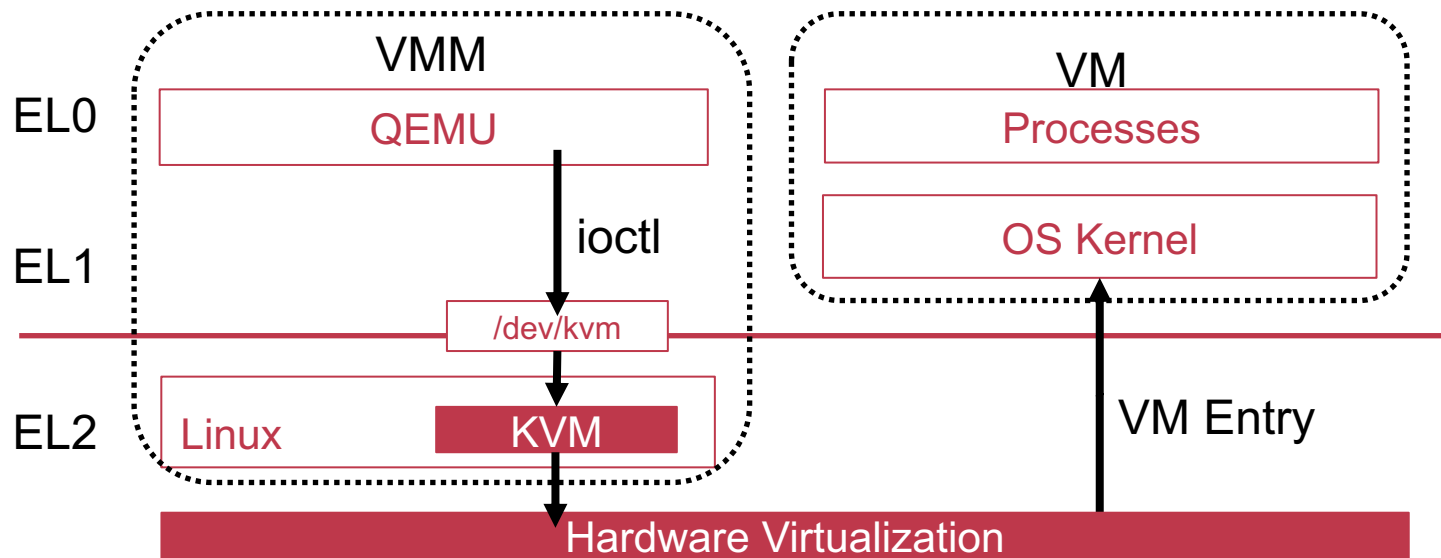
- 2005年11月，Intel发布带有VT-x的两款Pentium 4处理器
- 2006年中期，Qumranet公司在内部开发KVM(Kernel-based Virtual Machine)，并于11月发布
- 2007年，KVM被整合进Linux 2.6.20
- 2008年9月，Redhat出资1亿700万美元收购Qumranet
- 2009年，QEMU 0.10.1开始使用KVM，以替代其软件模拟的方案

QEMU/KVM架构

- **QEMU运行在用户态，负责实现策略**
 - 也提供虚拟设备的支持
- **KVM以Linux内核模块运行，负责实现机制**
 - 可以直接使用Linux的功能
 - 例如内存管理、进程调度
 - 使用硬件虚拟化功能
- **两部分合作**
 - KVM捕捉所有敏感指令和事件，传递给QEMU
 - KVM不提供设备的虚拟化，需要使用QEMU的虚拟设备

QEMU使用KVM的用户态接口

- QEMU使用/dev/kvm与内核态的KVM通信
 - 使用ioctl向KVM传递不同的命令：CREATE_VM, CREATE_VCPU, KVM_RUN等



QEMU使用KVM的用户态接口

```
open("/dev/kvm")
ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU)
while (true) {
    ioctl(KVM_RUN)
    exit_reason = get_exit_reason();
    switch (exit_reason) {
        case KVM_EXIT_IO: /* ... */
            break;
        case KVM_EXIT_MMIO: /* ... */
            break;
    }
}
```

Invoke VMENTRY

ioctl(KVM_RUN)时发生了什么

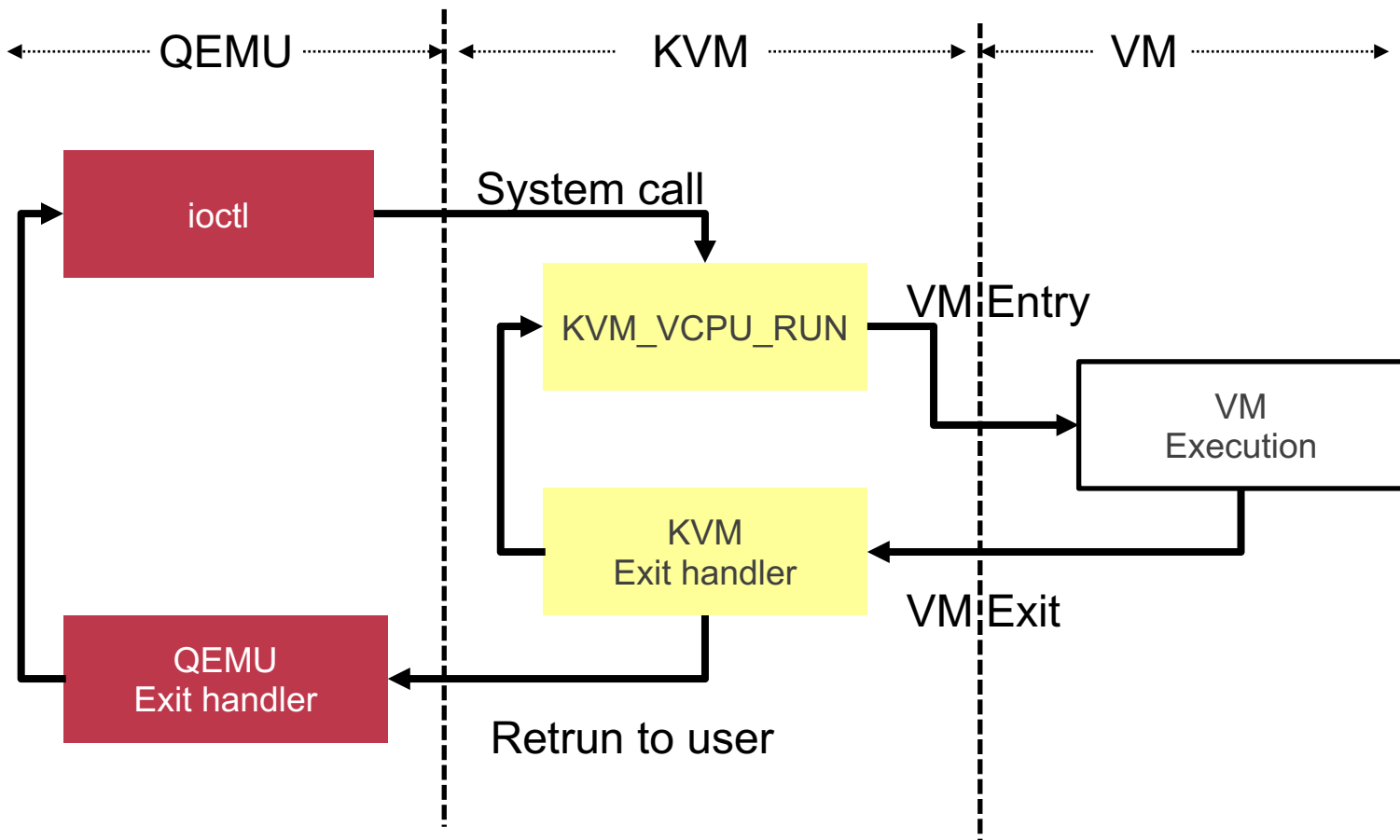
- **x86中**

- KVM找到此VCPU对应的VMCS
- 使用指令加载VMCS
- VMLAUNCH/VMRESUME进入Non-root模式
 - 硬件自动同步状态
 - PC切换成VMCS->GUEST_RIP，开始执行

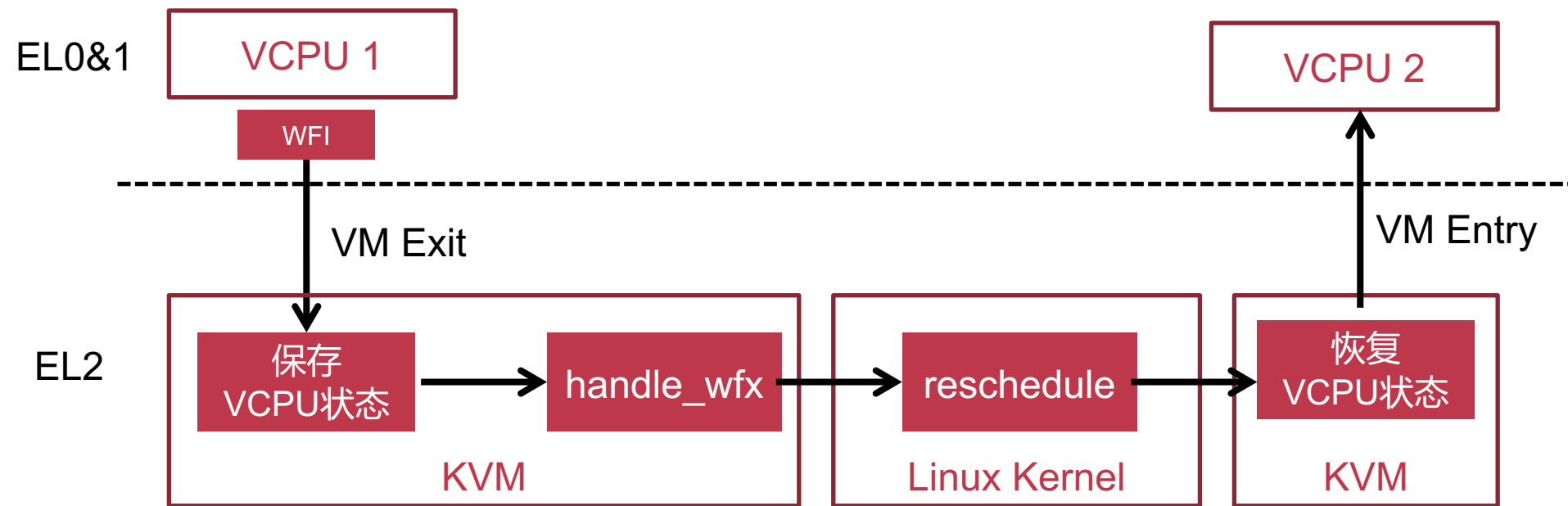
- **ARM中**

- KVM主动加载VCPU对应的所有状态
- 使用eret指令进入EL1
 - PC切换成ELR_EL2的值，开始执行

QEMU/KVM的流程



例：WFI指令VM Exit的处理流程



例：I/O指令VM Exit的处理流程

