IPADS INSTITUTE OF PARALLEL AND DISTRIBUTED SYSTEMS

上海交通大學 SHANGHAI JIAO TONG UNIVERSITY

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

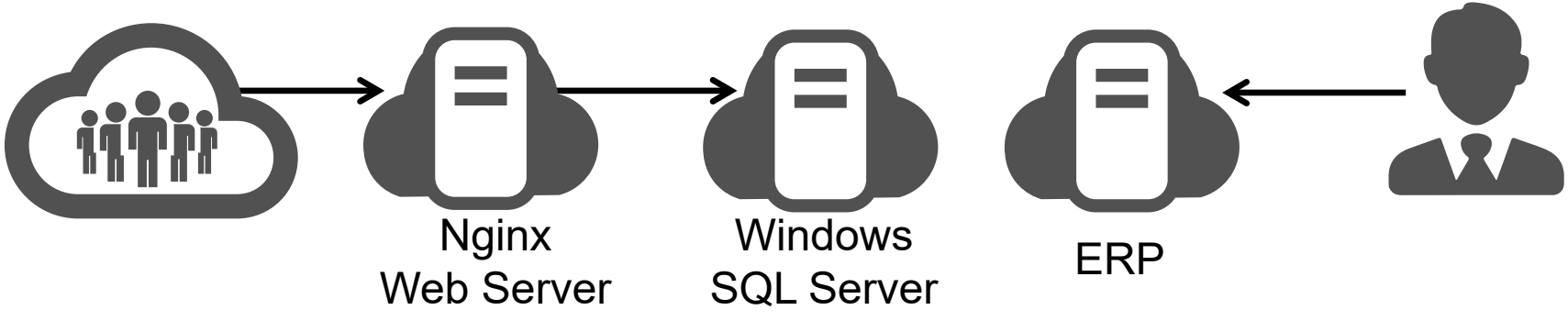
https://ipads.se.sjtu.edu.cn

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

现代IT公司的部署方式：云

- 云服务器代替物理服务器
- 云服务器配置与物理服务器一致
- 所有云服务器维护由服务商提供



云计算为云租户带来的优势

- 按需租赁、无需机房租赁费
- 无需雇佣物理服务器管理人员
- 可以快速低成本地升级服务器
- …

系统虚拟化

- **云计算的核心支撑技术**
- **新引入的一个软件层**
  - 上层是操作系统（虚拟机）
  - 底层是硬件

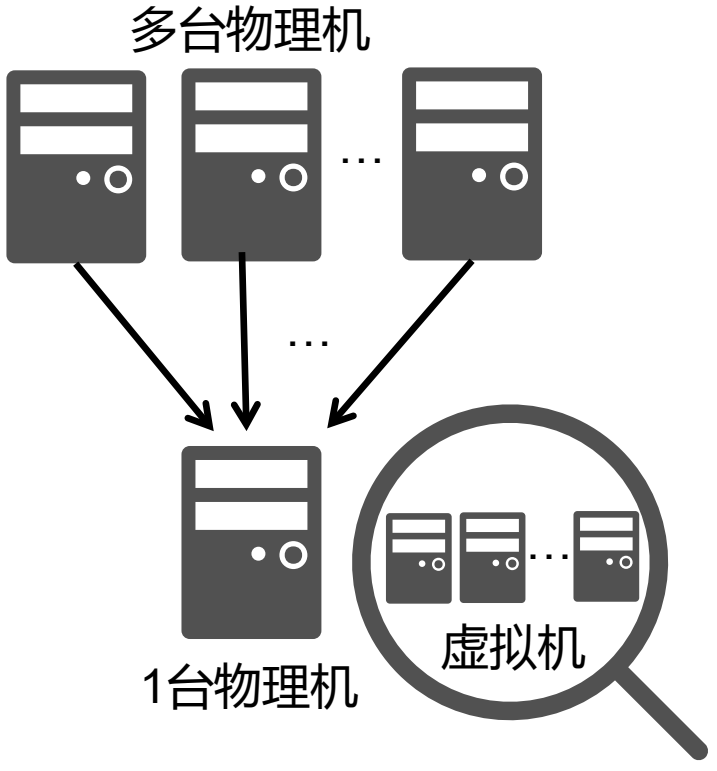
虚拟化带来的优势

- ”*Any problem in computer science can be solved by another level of indirection*" **--- David Wheeler**

- **服务器整合：提高资源利用率**
- **方便程序开发**
- **简化服务器管理**
- **…**

服务器整合

- **单个物理机资源利用率低**
  - CPU利用率通常20%
- **利用系统虚拟化进行资源整合**
  - 一台物理机中同时运行多台虚拟机
- **提升物理机资源利用率**
- **降低云服务提供商的成本**



方便程序开发

- **调试操作系统**
  - 单步调试操作系统
  - 查看当前虚拟硬件的状态
    - 寄存器中的值是否正确
    - 内存映射是否正确
  - 随时修改虚拟硬件的状态
- **测试应用程序的兼容性**
  - 可以在一台物理机上同时运行在不同的操作系统
  - 测试应用程序在不同操作系统上的兼容性

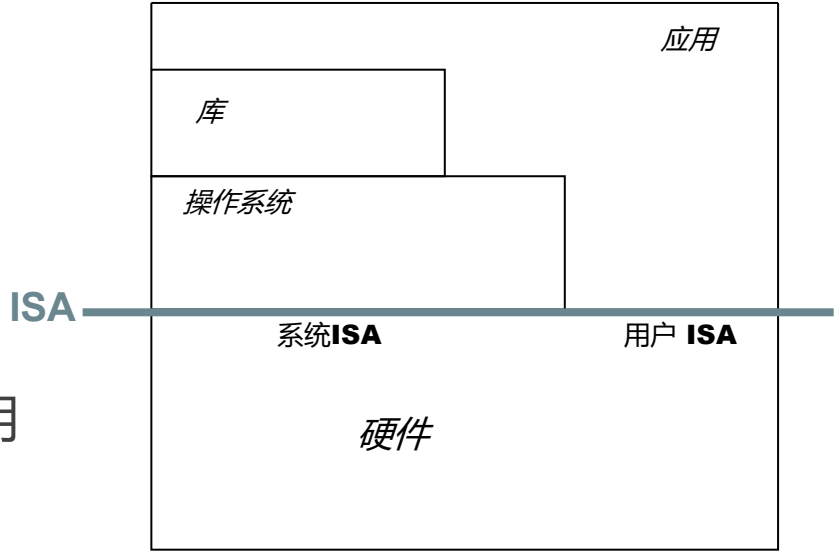
简化服务器管理

- **通过软件接口管理虚拟机**
  - 创建、开机、关机、销毁
  - 方便高效
- **虚拟机热迁移**
  - 方便物理机器的维护和升级

什么是系统虚拟化？

操作系统中的接口层次: ISA

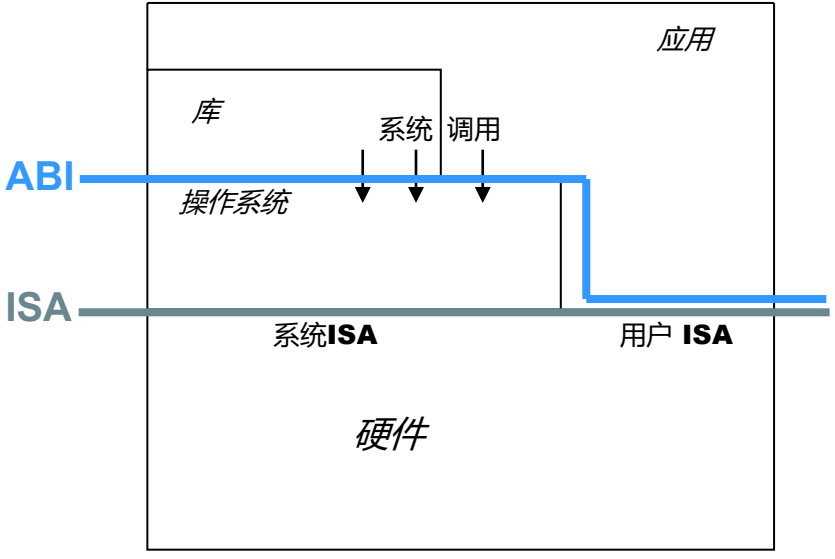
- **ISA层**
  - Instruction Set Architecture
  - 区分硬件和软件
  - 用户ISA
    - 用户态和内核态程序都可以使用
    - mov x0, sp
    - add x0, x0, #1
  - 系统ISA
    - 只有内核态程序可以使用
    - msr vbar_el1, x0



ISA – instruction set architecture

操作系统中的接口层次: ABI

- **ABI**
  - Application Binary Interface
  - 提供操作系统服务或硬件功能
  - 包含用户ISA和系统调用

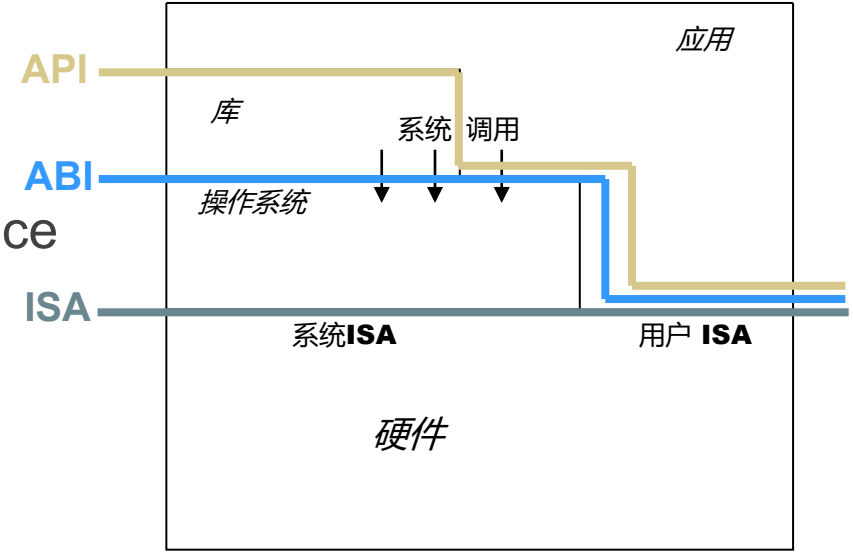


ABI – application binary interface

ISA – instruction set architecture

操作系统中的接口层次: API

- **API**
  - Application Programming Interface
  - 不同用户态库提供的接口
  - 包含库的接口和用户ISA
  - UNIX环境中的**clib**:
    - 支持UNIX/C编程语言



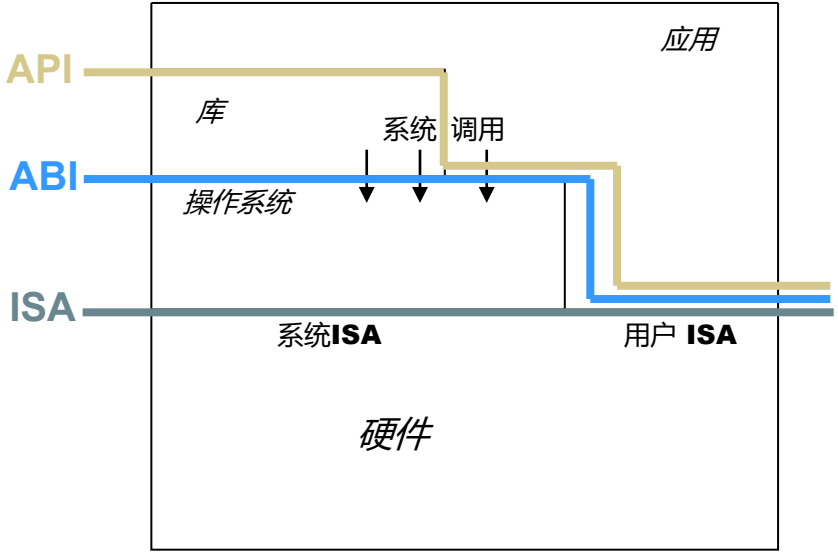
API – application programming interface

ABI – application binary interface

ISA – instruction set architecture

思考：这些程序用了哪层接口？

- Hello world
- Web game
- Dota
- Office 2016
- Windows 10
- Java applications
- Chcore



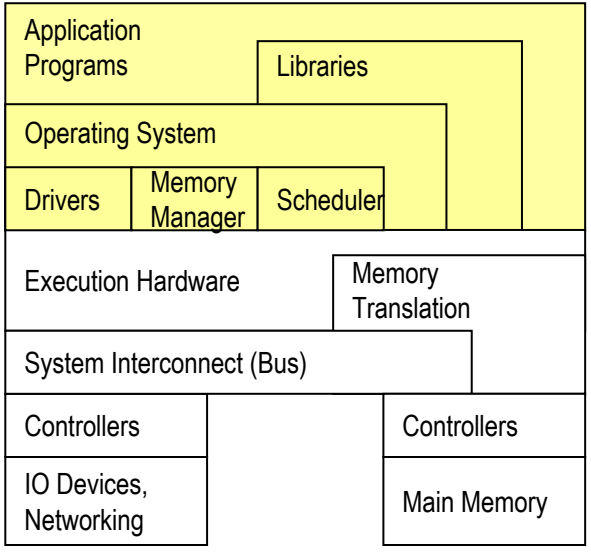
API – application programming interface

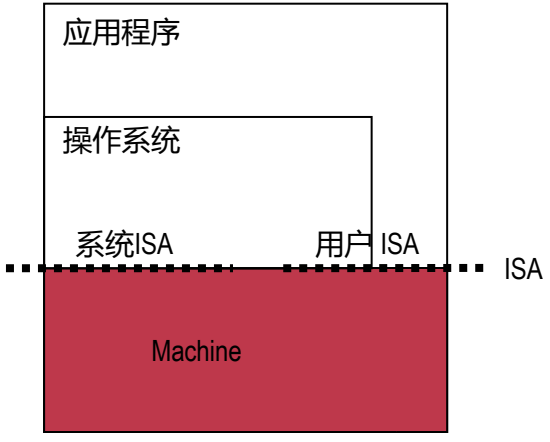
ABI – application binary interface

ISA – instruction set architecture

如何定义虚拟机？

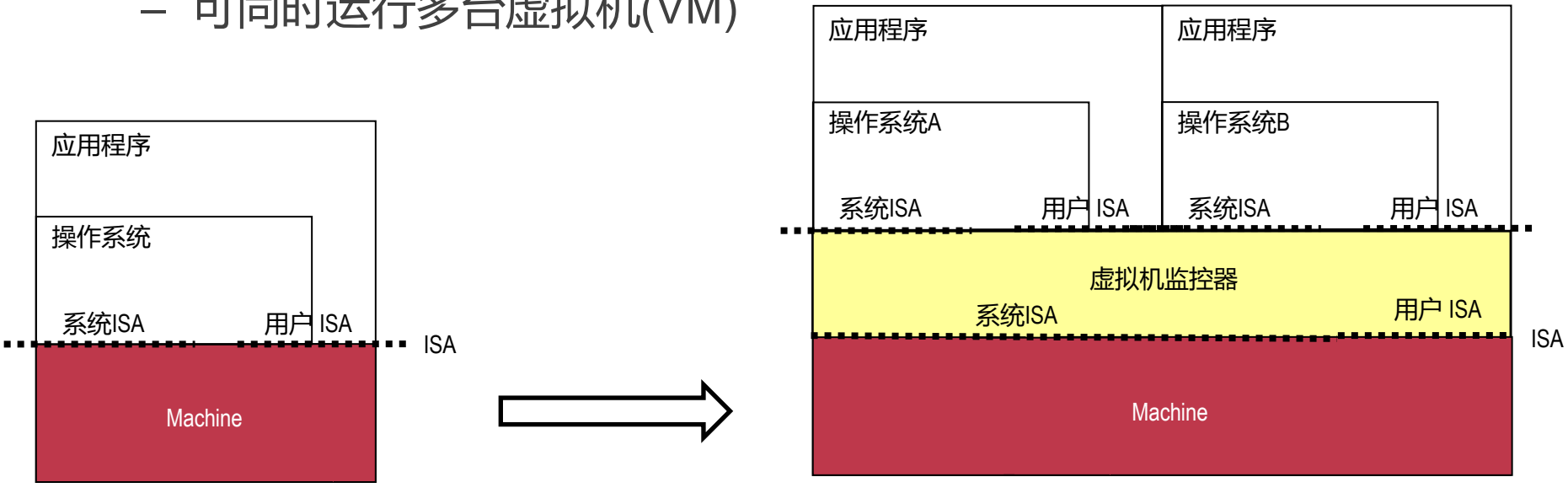
- **从操作系统角度看"Machine"**
  - ISA 提供了操作系统和Machine之间的界限





虚拟机和虚拟机监控器

- **虚拟机监控器(VMM/Hypervisor)**
  - 向上层虚拟机暴露其所需要的ISA
  - 可同时运行多台虚拟机(VM)



系统虚拟化的标准

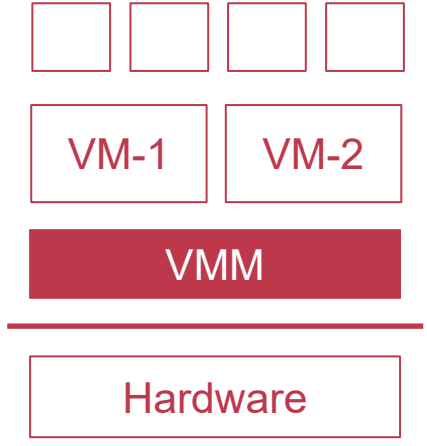
- **Popek & Goldberg, 1974 “Formal Requirements for Virtualizable Third Generation Architectures”**

- **高效系统虚拟化的三个特性**
  - 为虚拟机内程序提供与该程序原先执行的硬件完全一样的接口
  - 虚拟机只比在无虚拟化的情况下性能略差一点
  - 虚拟机监控器控制所有物理资源

虚拟机监控器的分类

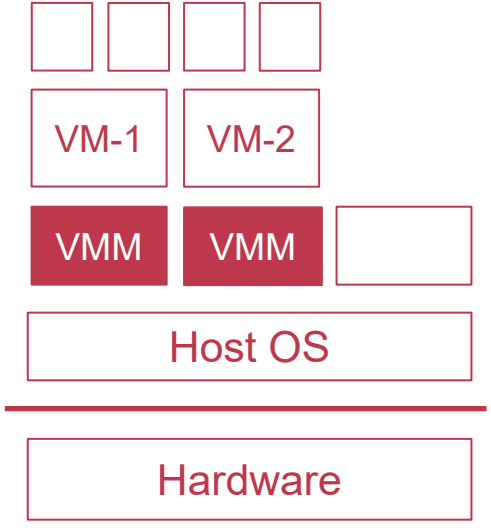
Type-1虚拟机监控器

- **直接运行在硬件之上**
  - 充当操作系统的角色
  - 直接管理所有物理资源
    - 实现调度、内存管理、驱动等功能
- **性能损失较少**
- **例如Xen, VMware ESX Server**



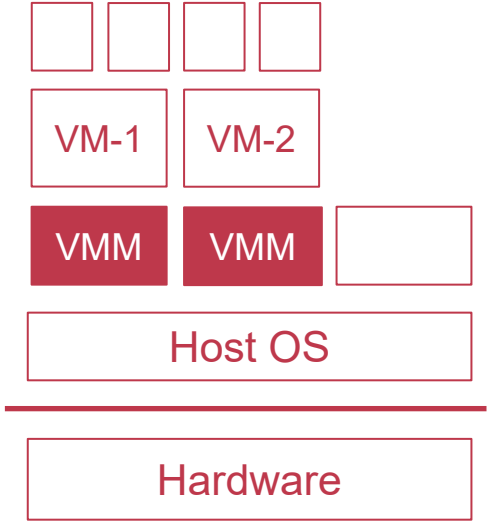
Type-2虚拟机监控器

- **依托于主机操作系统**
  - 主机操作系统管理物理资源
  - 虚拟机监控器以进程/内核模块的形态运行
- **易于实现和安装**
- **例如QEMU/KVM**
- **思考：**
  - Type-2类型有什么优势？



Type-2的优势

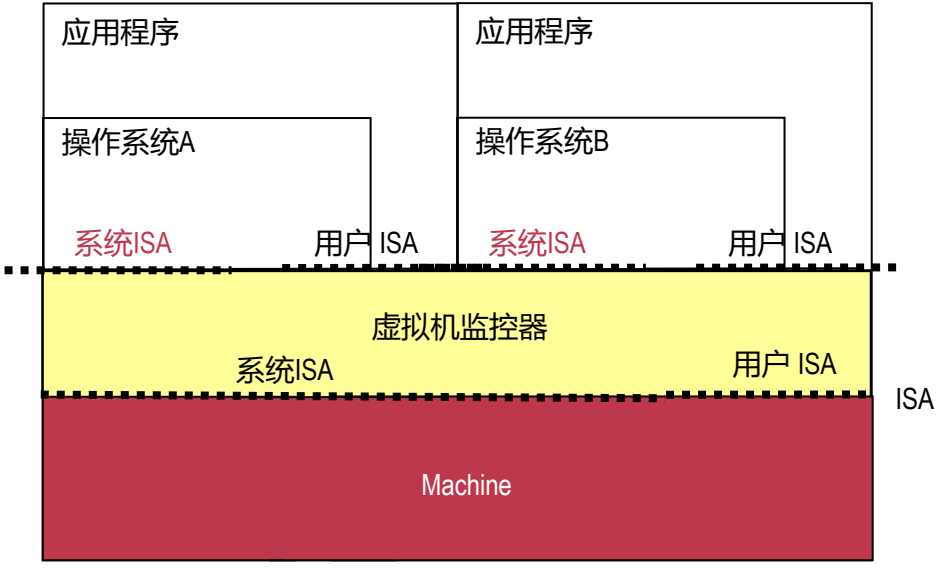
- **在已有的操作系统之上将虚拟机当做应用运行**
- **复用主机操作系统的大部分功能**
  - 文件系统
  - 驱动程序
  - 处理器调度
  - 物理内存管理



如何实现系统虚拟化？

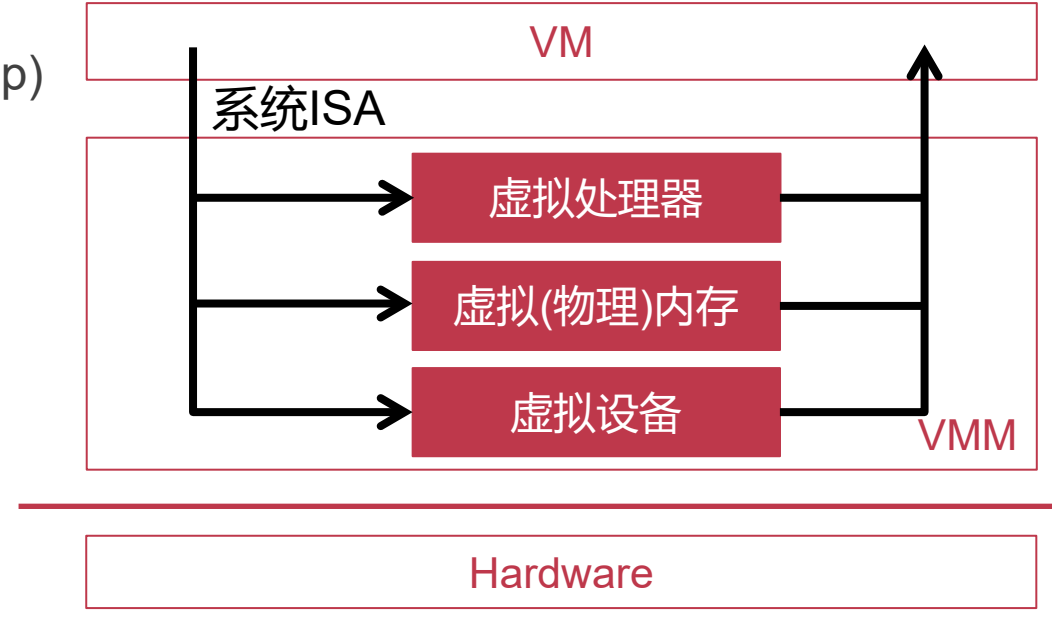
系统ISA

- **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



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



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

CPU Virtualization

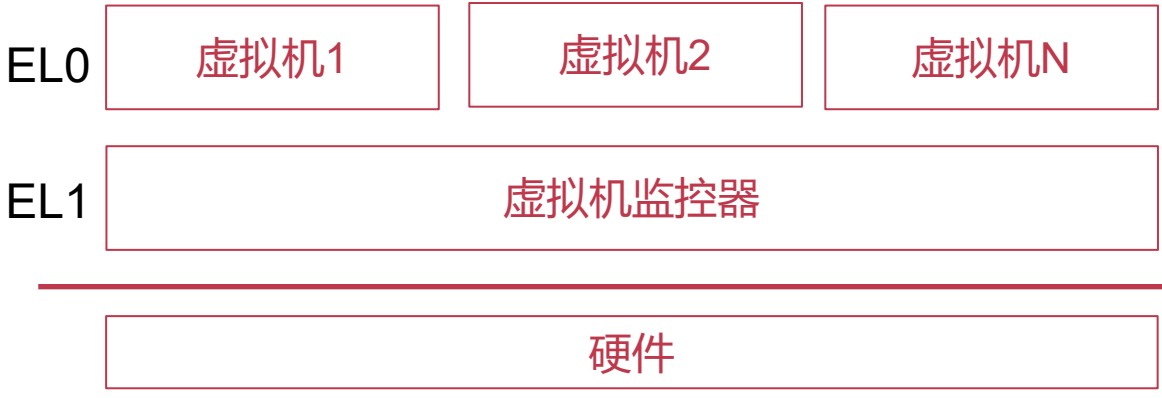
处理器虚拟化

回顾：ARM的特权级

- EL0: 用户态进程
- EL1: 操作系统内核

处理器虚拟化：一种直接的实现方法

- 将虚拟机监控器运行在EL1
- 将客户操作系统和其上的进程都运行在EL0
- 当操作系统执行系统ISA指令时下陷
  - 写入TTBR0_EL1
  - 执行WFI指令
  - …



Trap & Emulate

- Trap: 在用户态EL0执行特权指令将陷入EL1的VMM中
- Emulate：这些指令的功能都由VMM内的函数实现

非可虚拟化架构：Non-virtualizable

- **ARM不是严格的可虚拟化架构**
- **敏感指令**
  - 读写特殊寄存器或更改处理器状态
  - 读写敏感内存：例如访问未映射内存、写入只读内存
  - I/O指令
- **特权指令**
  - 在用户态执行会触发异常，并陷入内核态
- **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）

如何处理这些不会下陷的敏感指令?

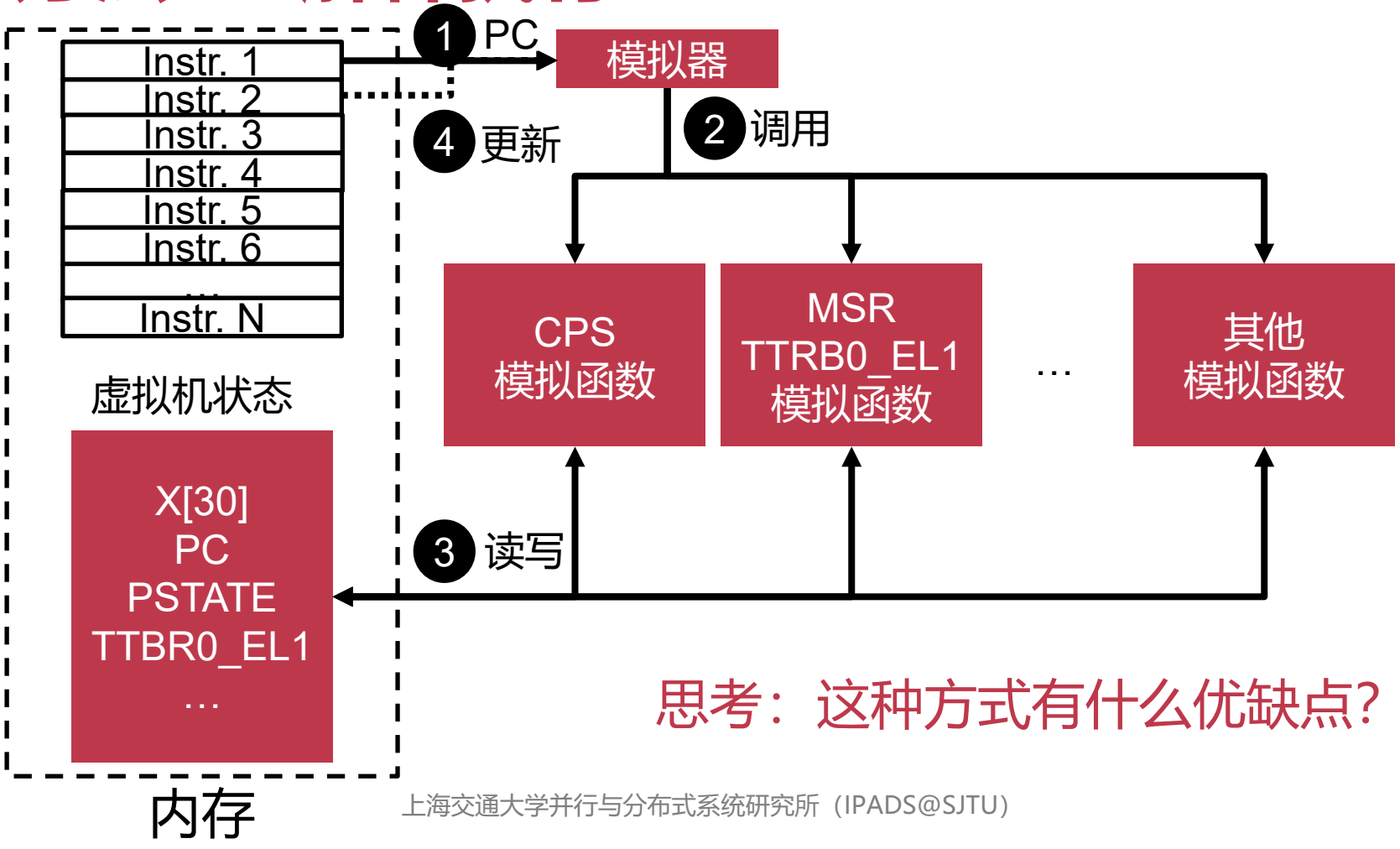
处理这些不会下陷的敏感指令，使得虚拟机中的操作系统能够运行在用户态（EL-0）

- **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）**

方法1：解释执行

- **使用软件方法一条条对虚拟机代码进行模拟**
  - 不区分敏感指令还是其他指令
  - 没有虚拟机指令直接在硬件上执行
- **使用内存维护虚拟机状态**
  - 例如：使用uint64_t x[30]数组保存所有通用寄存器的值

方法1：解释执行



思考：这种方式有什么优缺点？

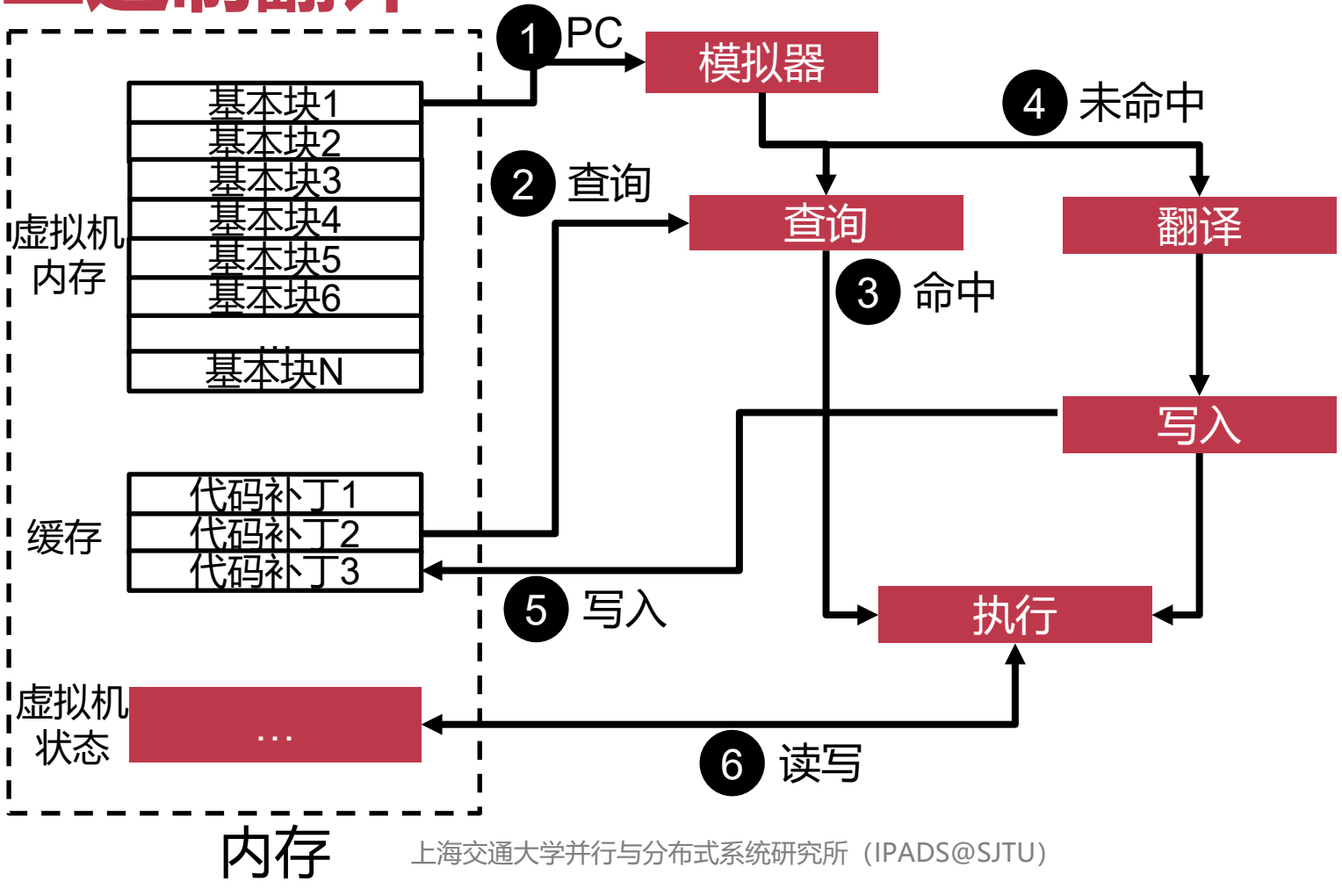
解释执行的优缺点

- **优点：**
  - 解决了敏感函数不下陷的问题
  - 可以模拟不同ISA的虚拟机
  - 易于实现、复杂度低
- **缺点：**
  - 非常慢：任何一条虚拟机指令都会转换成多条模拟指令

方法2：二进制翻译

- **提出两个加速技术**
  - 在执行前批量翻译虚拟机指令
  - 缓存已翻译完成的指令
- **使用基本块(Basic Block)的翻译粒度（为什么?）**
  - 每一个基本块被翻译完后叫代码补丁

二进制翻译


1 PC
模拟器
4 未命中
2 查询
查询
翻译
3 命中
虚拟机
内存
基本块1
基本块2
基本块3
基本块4
基本块5
基本块6
基本块N
写入
缓存
代码补丁1
代码补丁2
代码补丁3
5 写入
执行
虚拟机
状态
…
6 读写
内存

二进制翻译的缺点

- **不能处理自修改的代码(Self-modifying Code)**
- **中断插入粒度变大**
  - 模拟执行可以在任意指令位置插入虚拟中断
  - 二进制翻译时只能在基本块边界插入虚拟中断（为什么？）

方法3：半虚拟化(Para-virtualization)

- **协同设计**
  - 让VMM提供接口给虚拟机，称为**Hypercall**
  - 修改操作系统源码，让其主动调用VMM接口
- **Hypercall可以理解为VMM提供的系统调用**
  - 在ARM中是HVC指令
- **将所有不引起下陷的敏感指令替换成超级调用**
- **思考：这种方式有什么优缺点?**

半虚拟化方法的优缺点

- **优点：**
  - 解决了敏感函数不下陷的问题
  - 协同设计的思想可以提升某些场景下的系统性能
    - I/O等场景
- **缺点：**
  - 需要修改操作系统代码，难以用于闭源系统
  - 即使是开源系统，也难以同时在不同版本中实现

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


虚拟机N
进程
操作系统内核
Non-root 模式
Ring-3
Non-root 模式
Ring-0
Root模式
Ring-3
进程
Root模式
Ring-0
虚拟机监控器

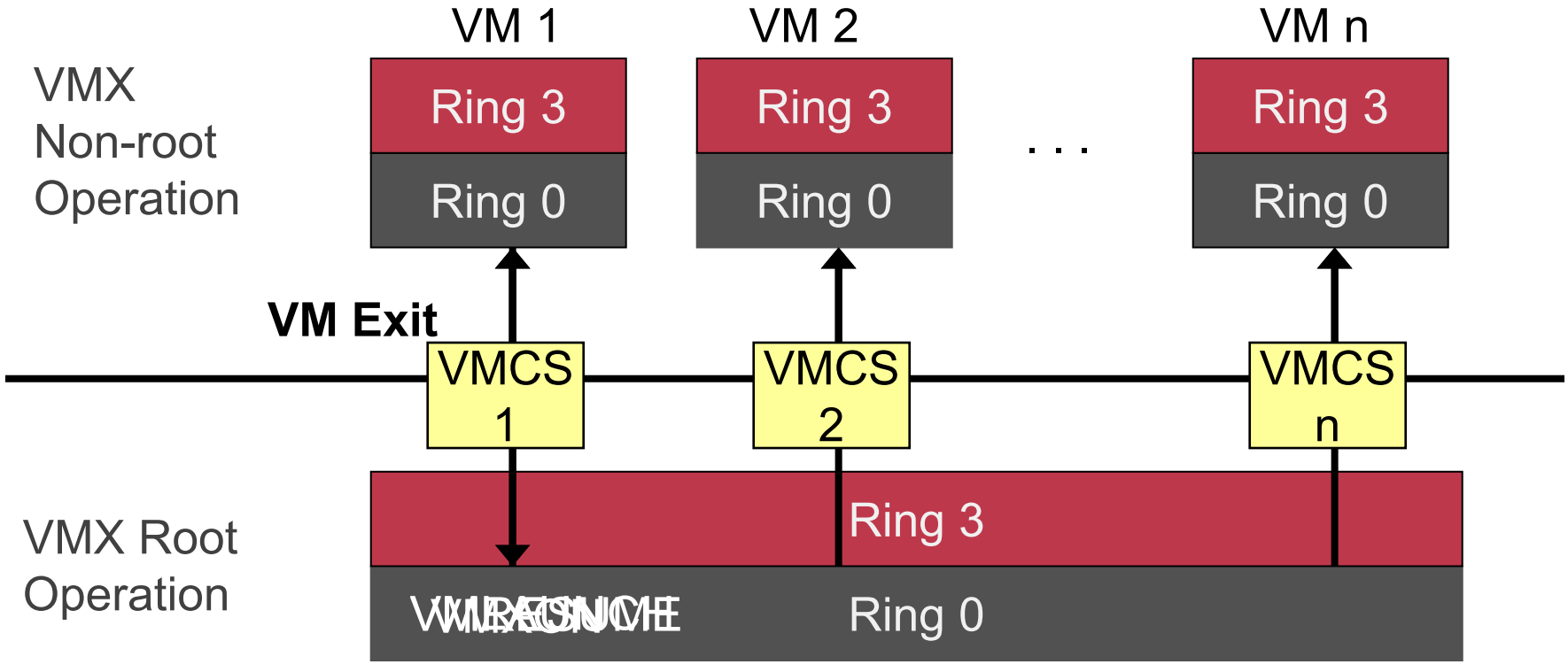
Virtual Machine Control Structure (VMCS)

- **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

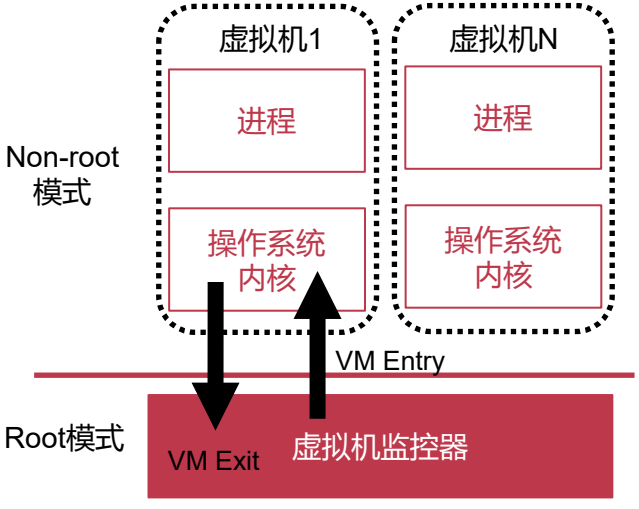
- **包含6个部分**
  - Guest-state area: 发生VM exit时，CPU的状态会被硬件自动保存至该区域；发生VM Entry时，硬件自动从该区域加载状态至CPU中
  - Host-state area：发生VM exit时，硬件自动从该区域加载状态至CPU中；发生VM Entry时，CPU的状态会被自动保存至该区域
  - VM-execution control fields： 控制Non-root模式中虚拟机的行为
  - VM-exit control fields：控制VM exit的行为
  - VM-entry control fields：控制VM entry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


VM 1
VM 2
VM n
VMX
Non-root
Operation
Ring 3
Ring 0
Ring 3
Ring 0
. . .
Ring 3
Ring 0
VM Exit
VMCS
1
VMCS
2
VMCS
n
VMX Root
Operation
Ring 3
Ring 0

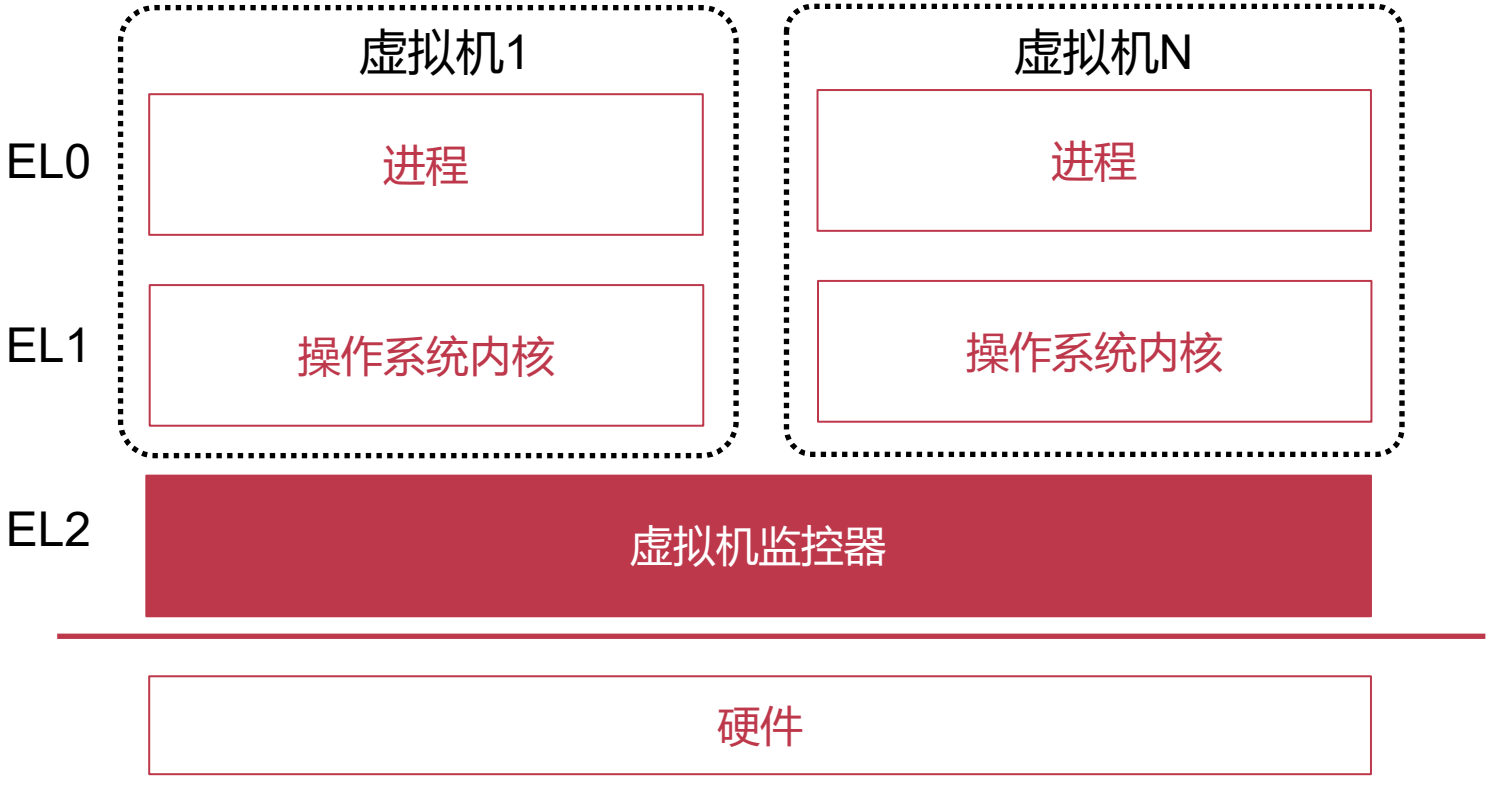
x86中的VM Entry和VM Exit

- **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)



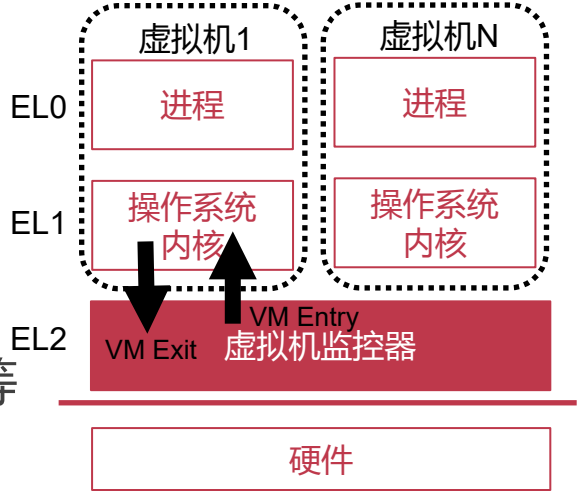
ARM的虚拟化技术

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

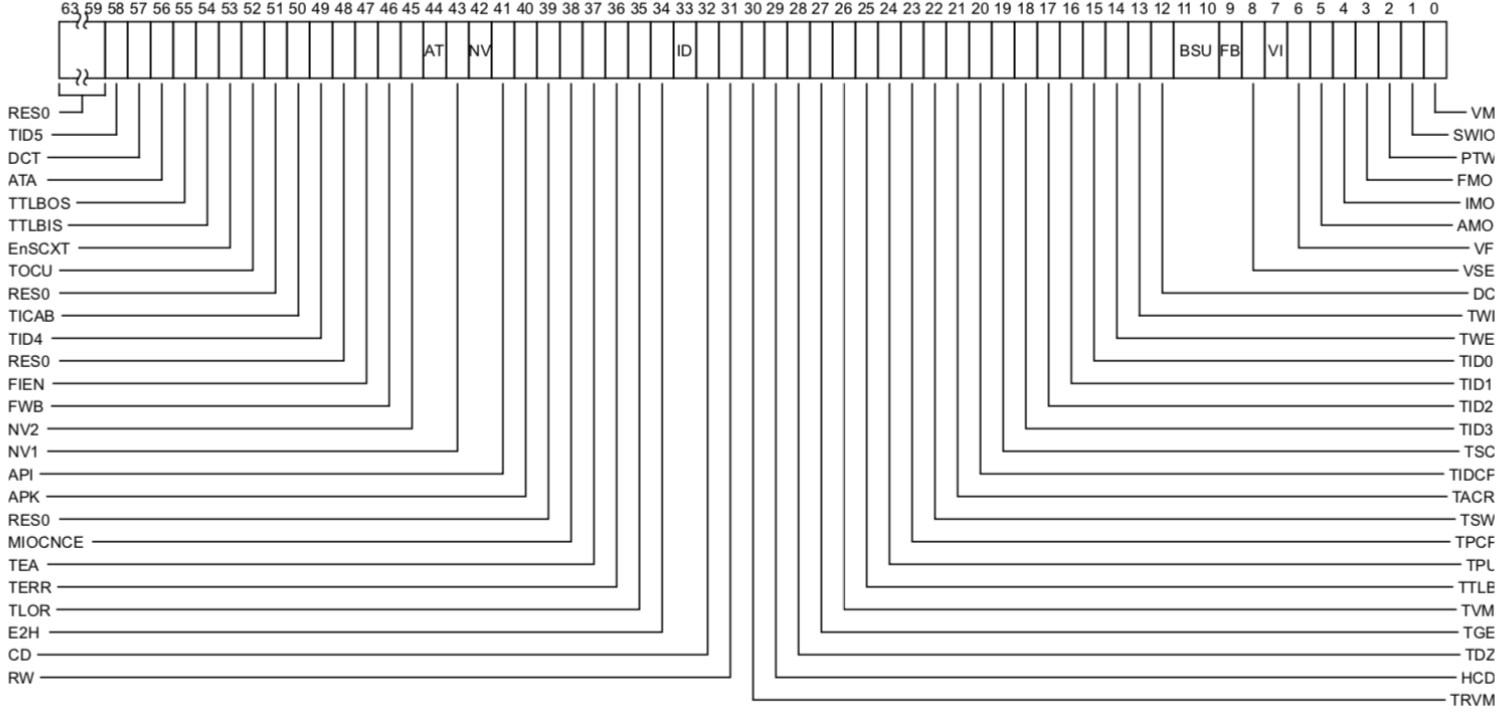
- **ARM中没有VMCS**
- **VM能直接控制EL1和EL0的状态**
  - 自由地修改PSTATE(VMM不需要捕捉CPS指令)
  - 可以读写TTBR0_EL1/SCTRL_EL1/TCR_EL1等寄存器
- **VM Exit时VMM仍然可以直接访问VM的EL0和EL1寄存器**

- **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(下节课内容)

HCR_EL2寄存器简介

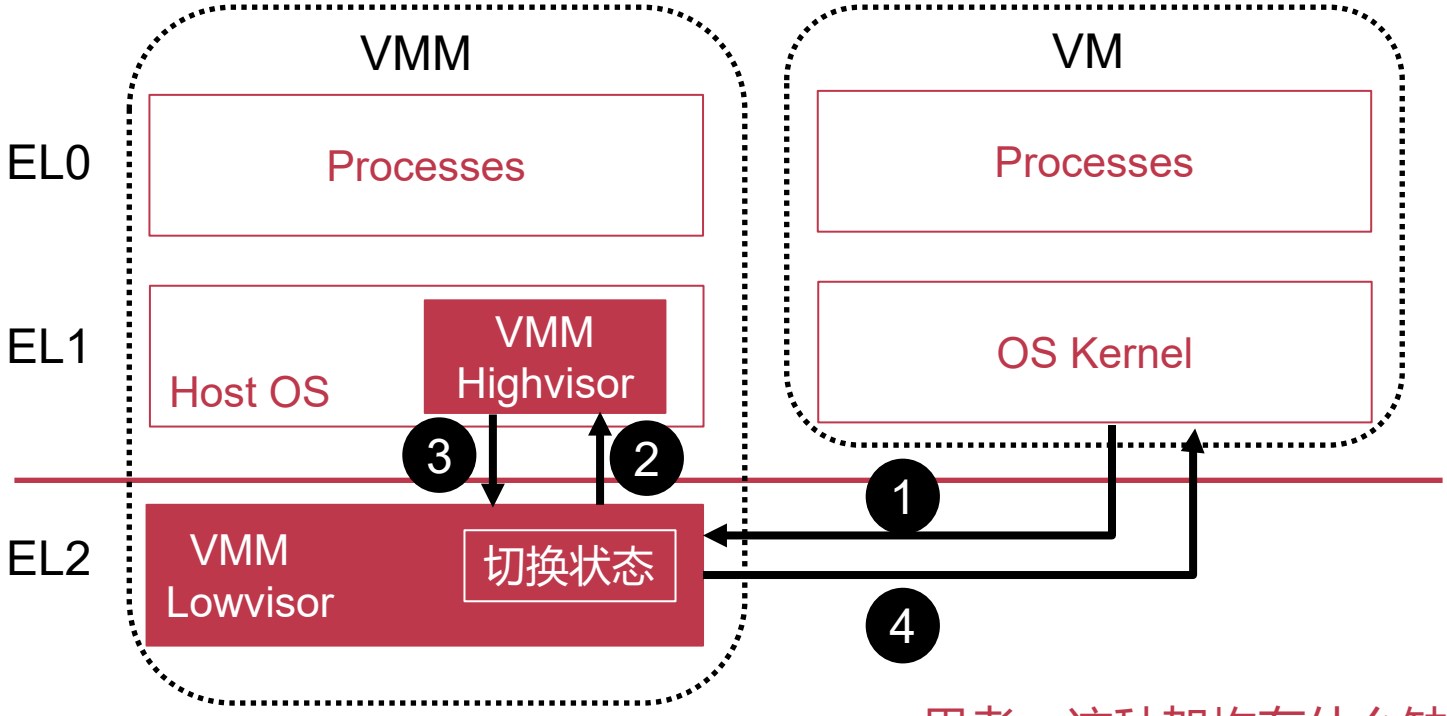


ARM硬件虚拟化发展

- **ARMv8.0**
  - 增加EL2[1]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，例如TTBR0_EL1，在EL2中不存在对应的寄存器
    - 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

1: 实际上名为hyp模式

ARMv8.0中的Type-2 VMM架构

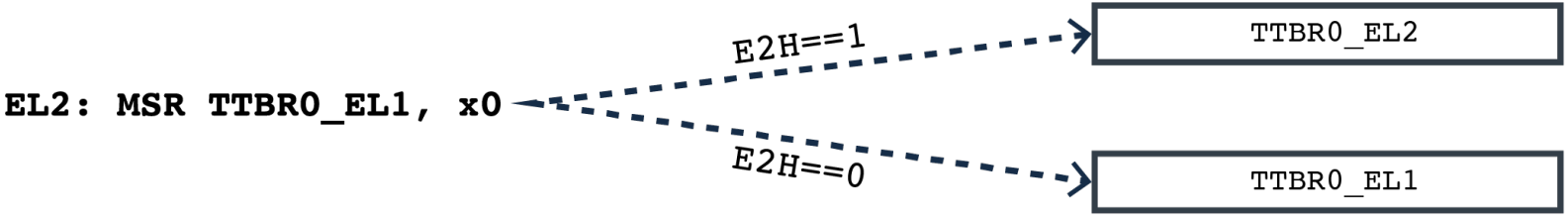


思考：这种架构有什么缺点?

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

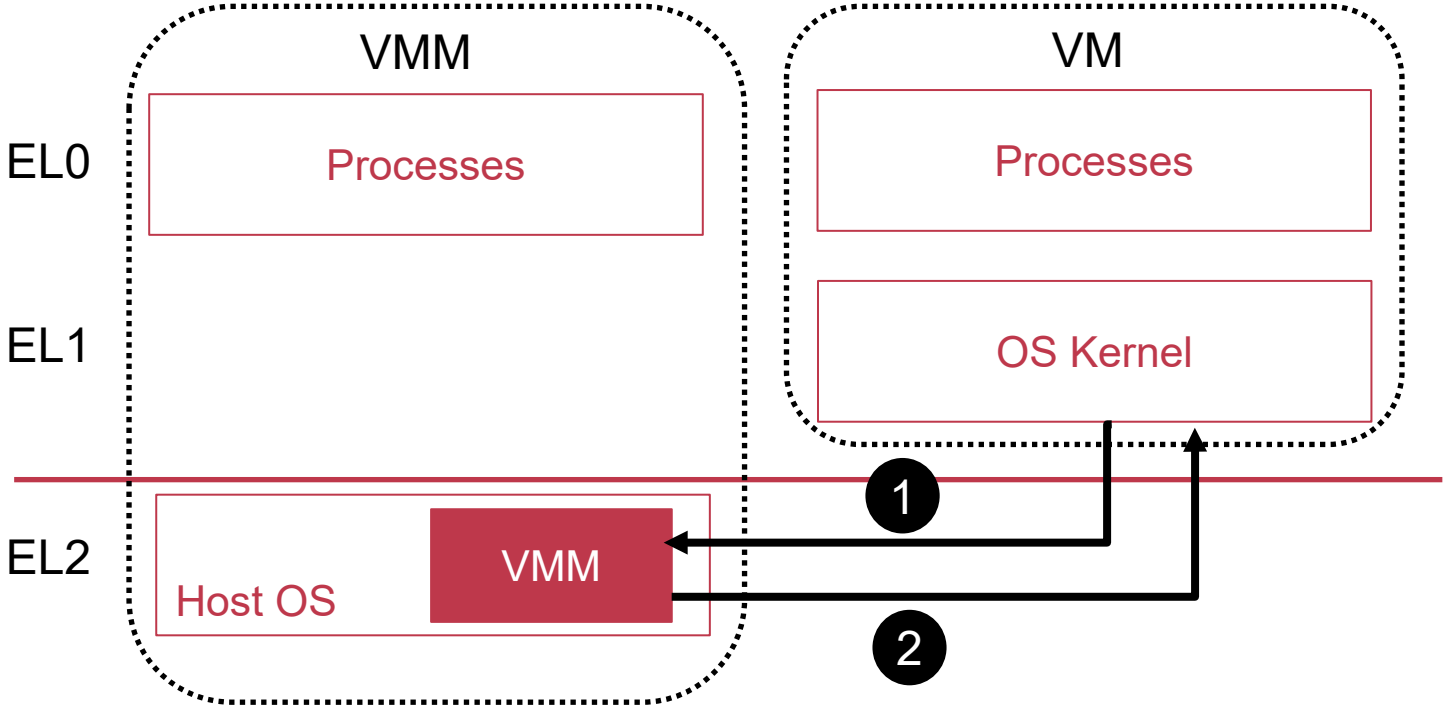
ARMv8.1中的Type-2 VMM架构

- **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
  - EL2中可直接运行未修改的操作系统内核（Host OS）



图片来自于https://developer.arm.com/

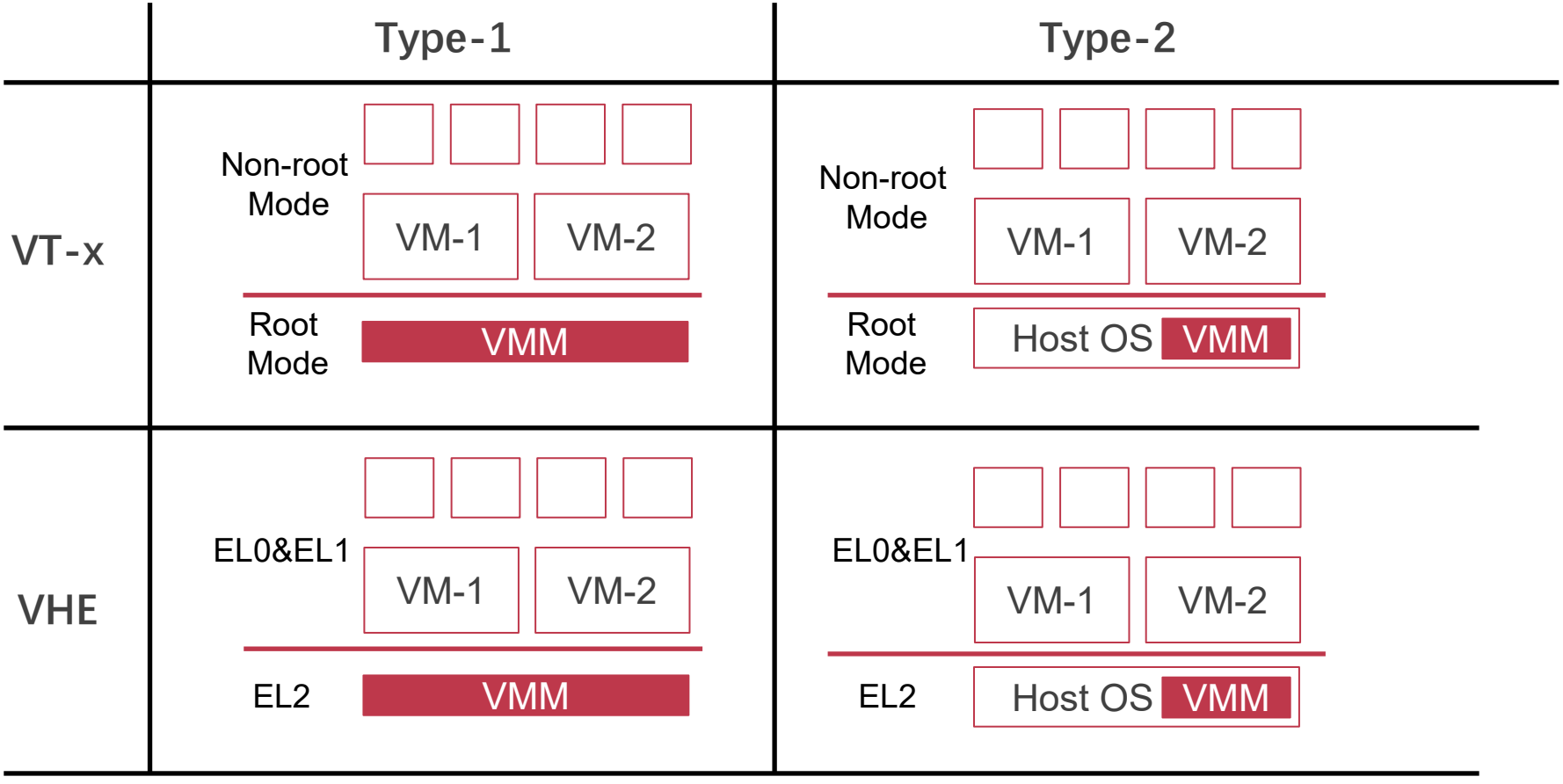
ARMv8.1中的Type-2 VMM架构



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 ... 、SPARC等
  - 在这阶
    - 如

Fabrice Bellard fabrice.bellard at 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/qemu/. 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

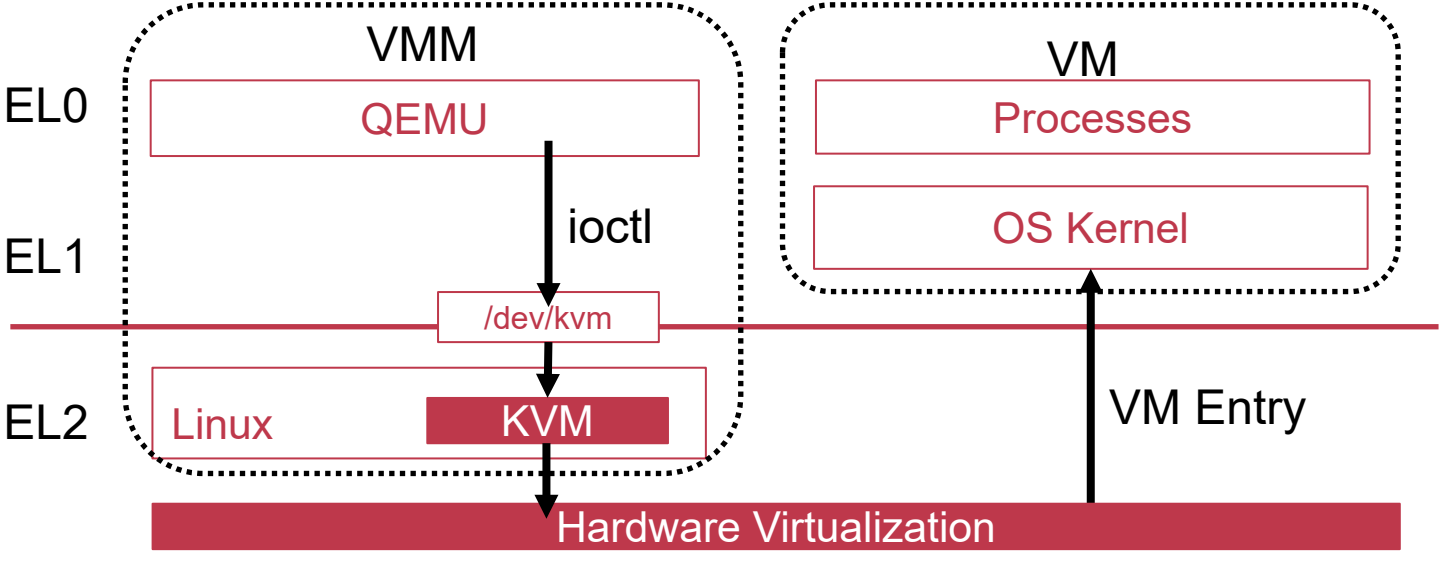
- 2005年11月，Intel发布带有VT-x的两款Pentium 4处理器
- 2006年中期， Qumranet公司在内部开发KVM(Kernel-based Virtual Machine)，并于11月发布
- 2007年，KVM被整合进Linux 2.6.20
- 2008年9月，Redhat出资1亿700美元收购Qumranet
- 2009年，QEMU 0.10.1开始使用KVM，以替代其软件模拟的方案

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

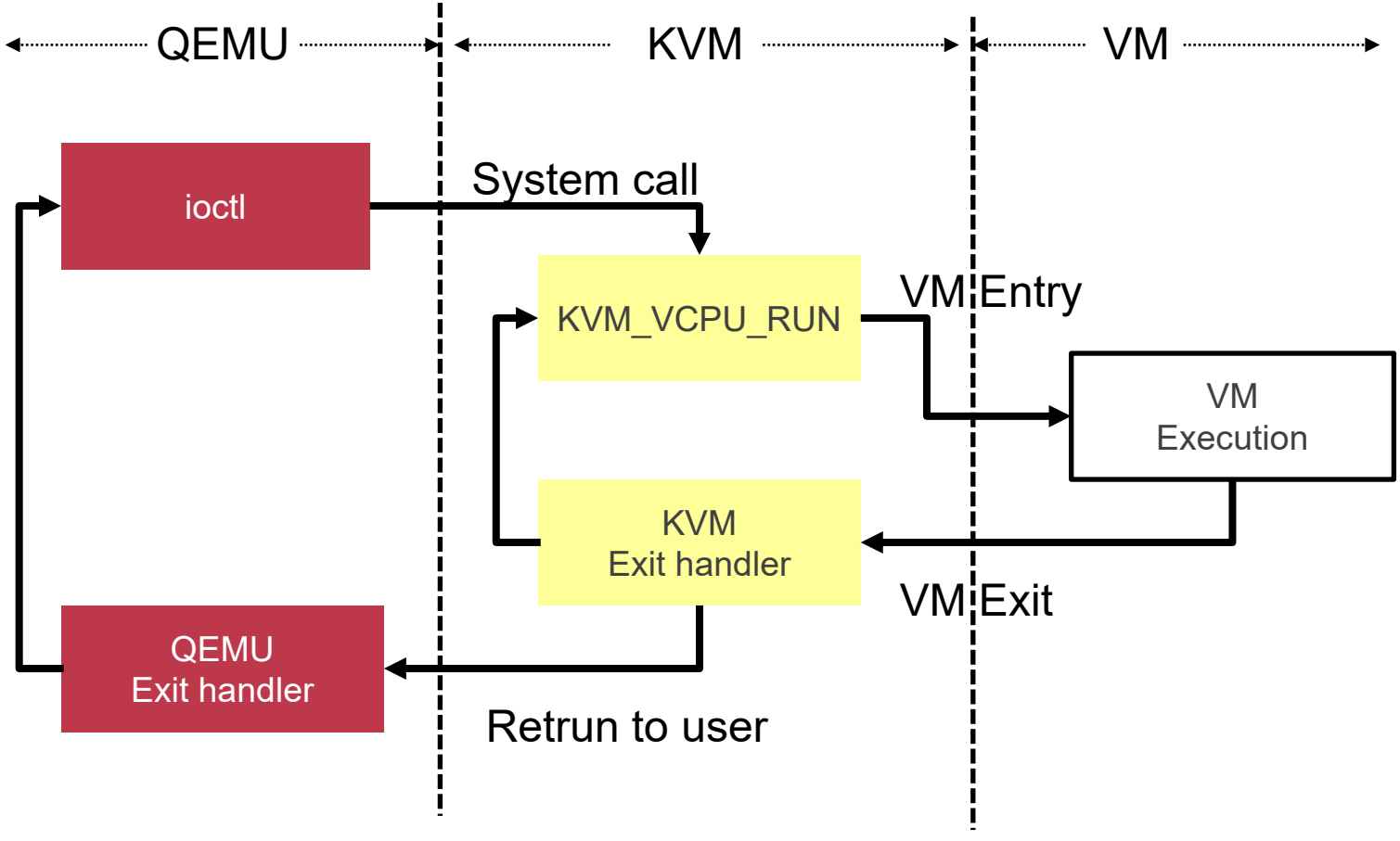
```
open("/dev/kvm")
ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU)
while (true) {
      ioctl(KVM_RUN)
      exit_reason = get_exit_reason();
      switch (exit_reason) {
        case KVM_EXIT_IO:  /* ... */
          break;
        case KVM_EXIT_MMIO: /* ... */
          break;
      }
}
```

Invoke VMENTRY

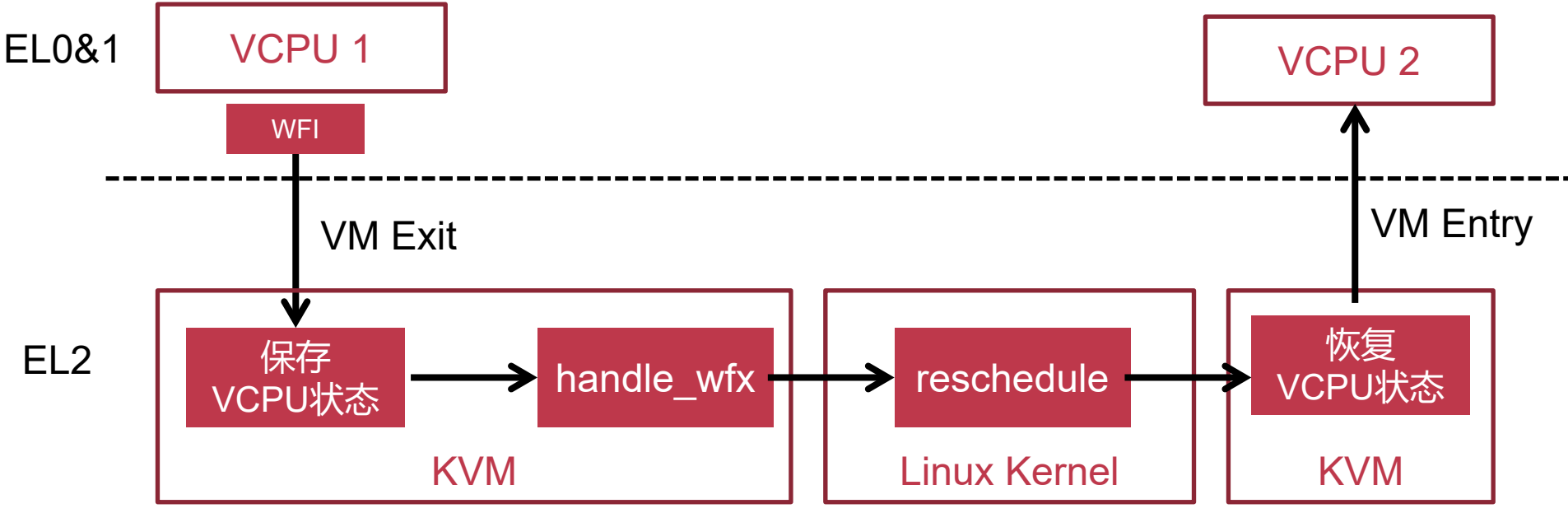
ioctl(KVM_RUN)时发生了什么

- **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EL2
    - PC切换成ELR_EL2的值，开始执行

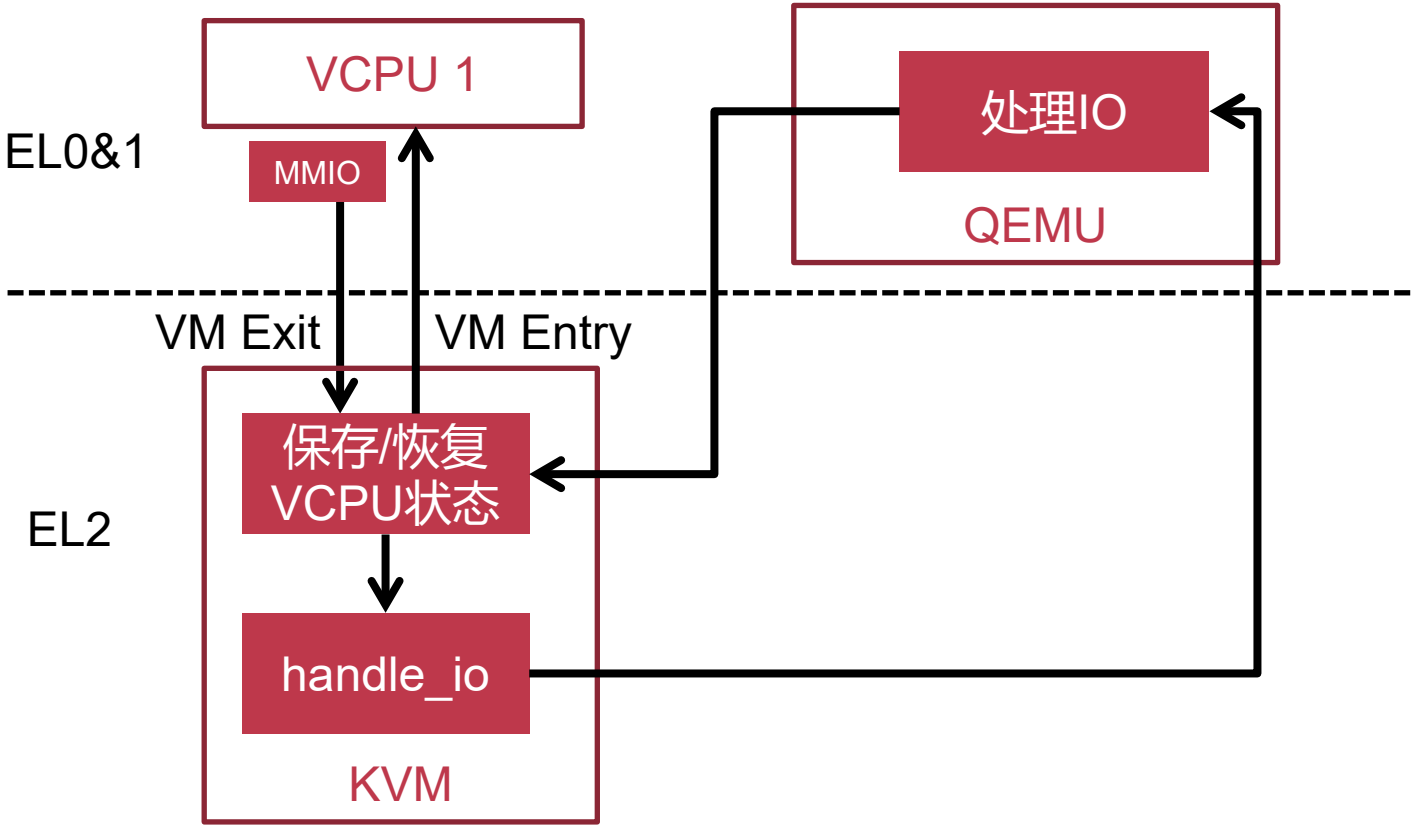
QEMU/KVM的流程


QEMU
KVM
VM
ioctl
System call
KVM_VCPU_RUN
VM Entry
VM
Execution
KVM
Exit handler
VM Exit
QEMU
Exit handler
Retrun to user

例：WFI指令VM Exit的处理流程



例：I/O指令VM Exit的处理流程



下次课内容

- 内存与I/O虚拟化