上海交通大学
SHANGHAI JIAO TONG UNIVERSITY

SJTU

虚拟化：云计算的核心支撑技术（1）

马汝辉 副教授 博导
计算机科学与工程系
上海交通大学

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

目录

01

虚拟化技术总体介绍

为什么进行系统虚拟化?

- 系统虚拟化是云计算的核心支撑技术

虚拟化带来的优势

服务器整合

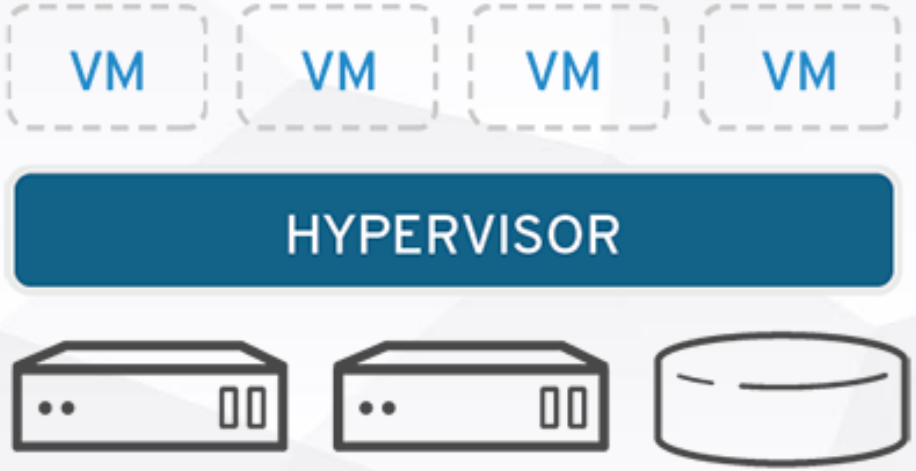
- 单个物理机资源利用率低
- 利用系统虚拟化进行资源整合
- 提升物理机资源利用率
- 降低云服务提供商的成本

方便程序开发

- 调试操作系统
- 测试应用程序的兼容性

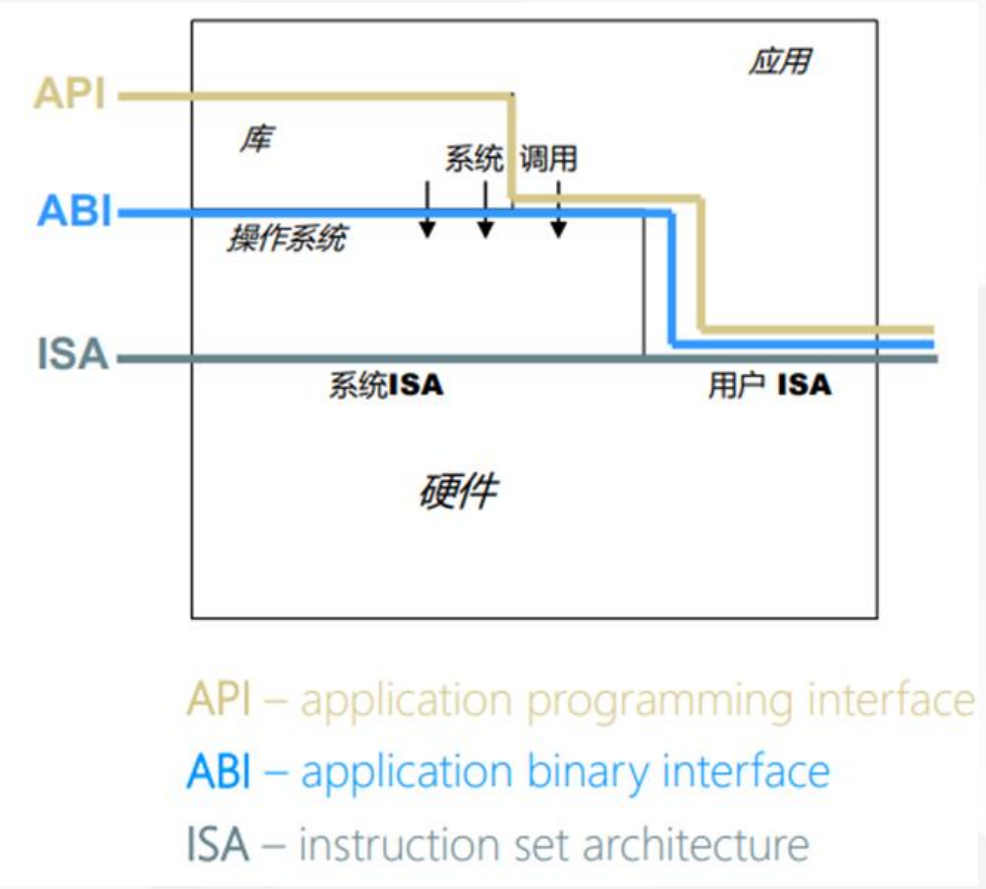
简化服务器管理

- 通过软件接口管理虚拟机
- 虚拟机热迁移



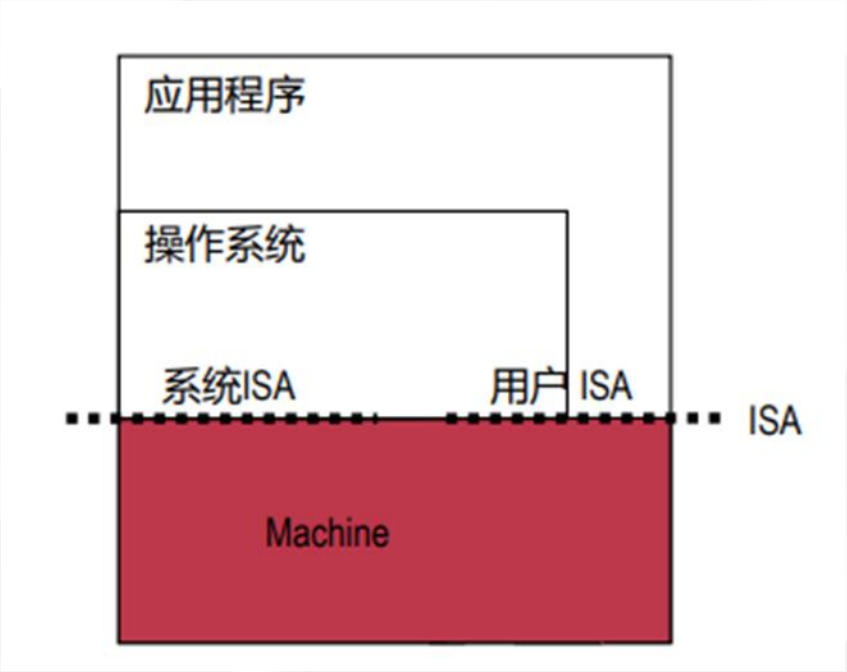
从操作系统中的接口层次看虚拟化的实现

- **ISA（Instruction Set Architecture）**
  - 系统ISA：
    - 特权指令
    - 只有内核态程序以使用
  - 用户ISA：
  - 用户态和内核态程序都可以使用
- **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
- **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



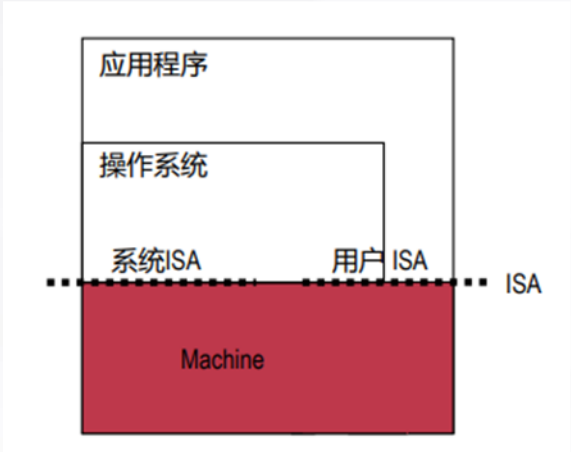
系统软件和物理硬件之间的关系

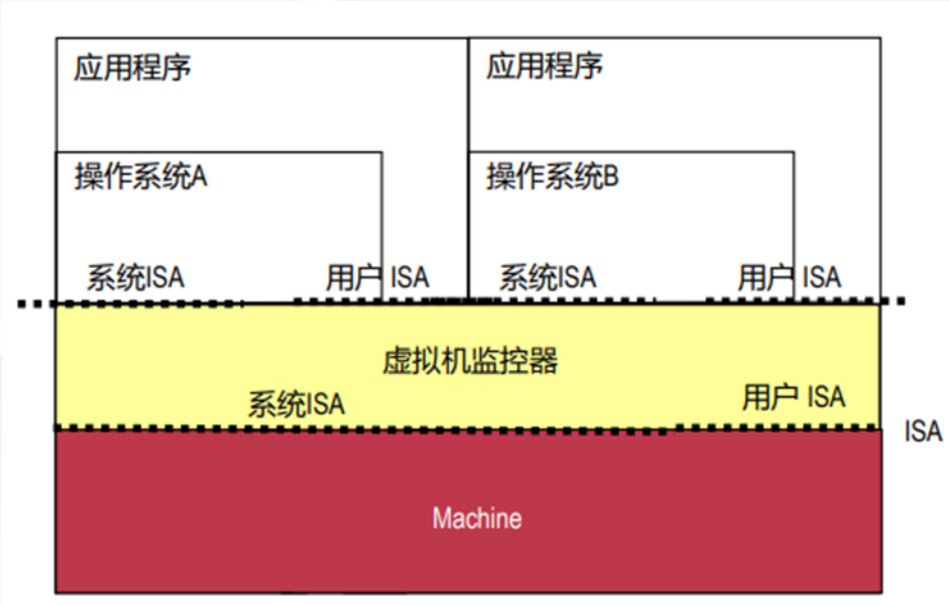
- **操作系统如何看待它管理的一台“机器”呢?**
  - ISA 提供了操作系统和Machine之间的界限
  - 系统软件就是通过ISA与硬件进行交互，也对硬件资源进行隔离
- **那么如何在操作系统的内部隔离出另外的操作系统，让它也可以与物理硬件交互呢?**
  - 虚拟化的具体实现技术就是为了解决这个问题。



虚拟化技术具体实现中最关注的两个部分

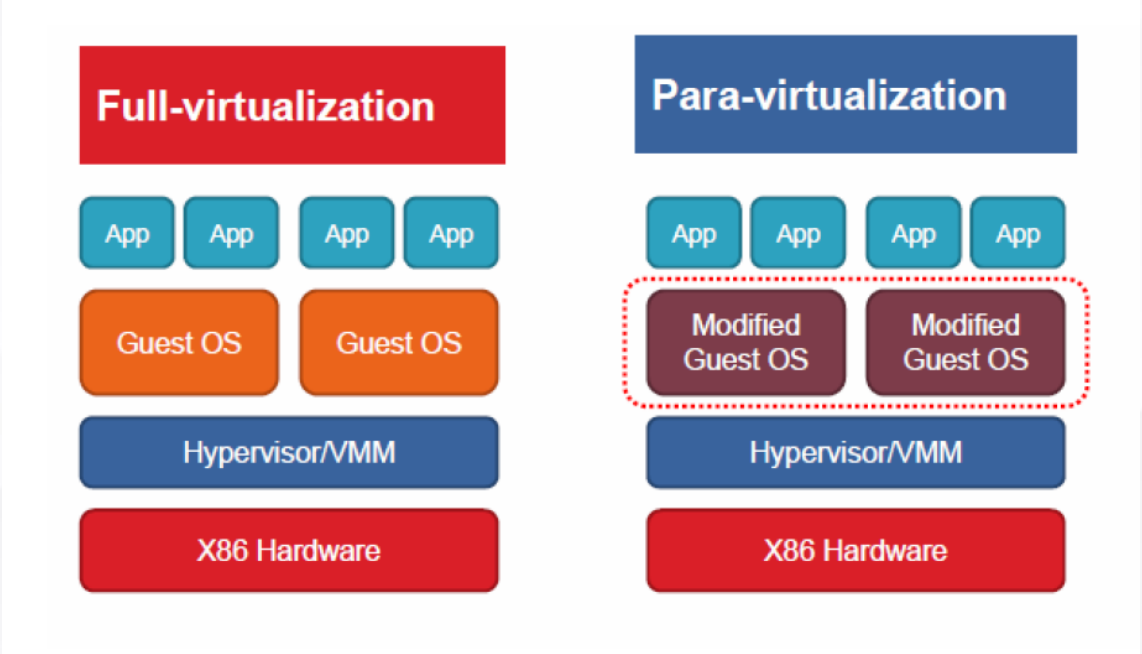
- 虚拟机（Virtual Machine）
- 虚拟机监控器(Virtual Machine Monitor, Hypervisor)





虚拟化技术实现中的分类

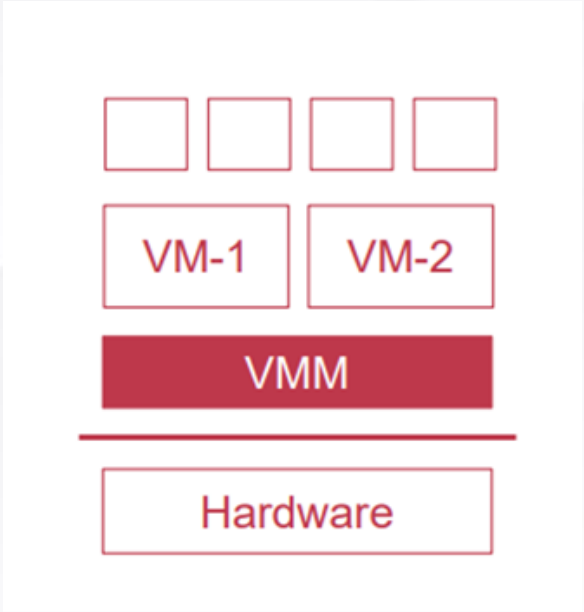
- **全虚拟化（Full Virtualization） vs. 半虚拟化 (Para-Virtualization)**
  - 概念的变化
  - 从虚拟机的角度来分析
    - 虚拟机操作系统是否修改?



虚拟化技术实现中虚拟机监控器的分类

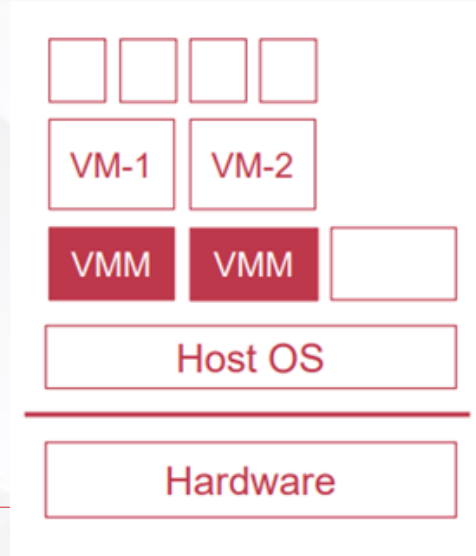
Type-1虚拟机监控器

- 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

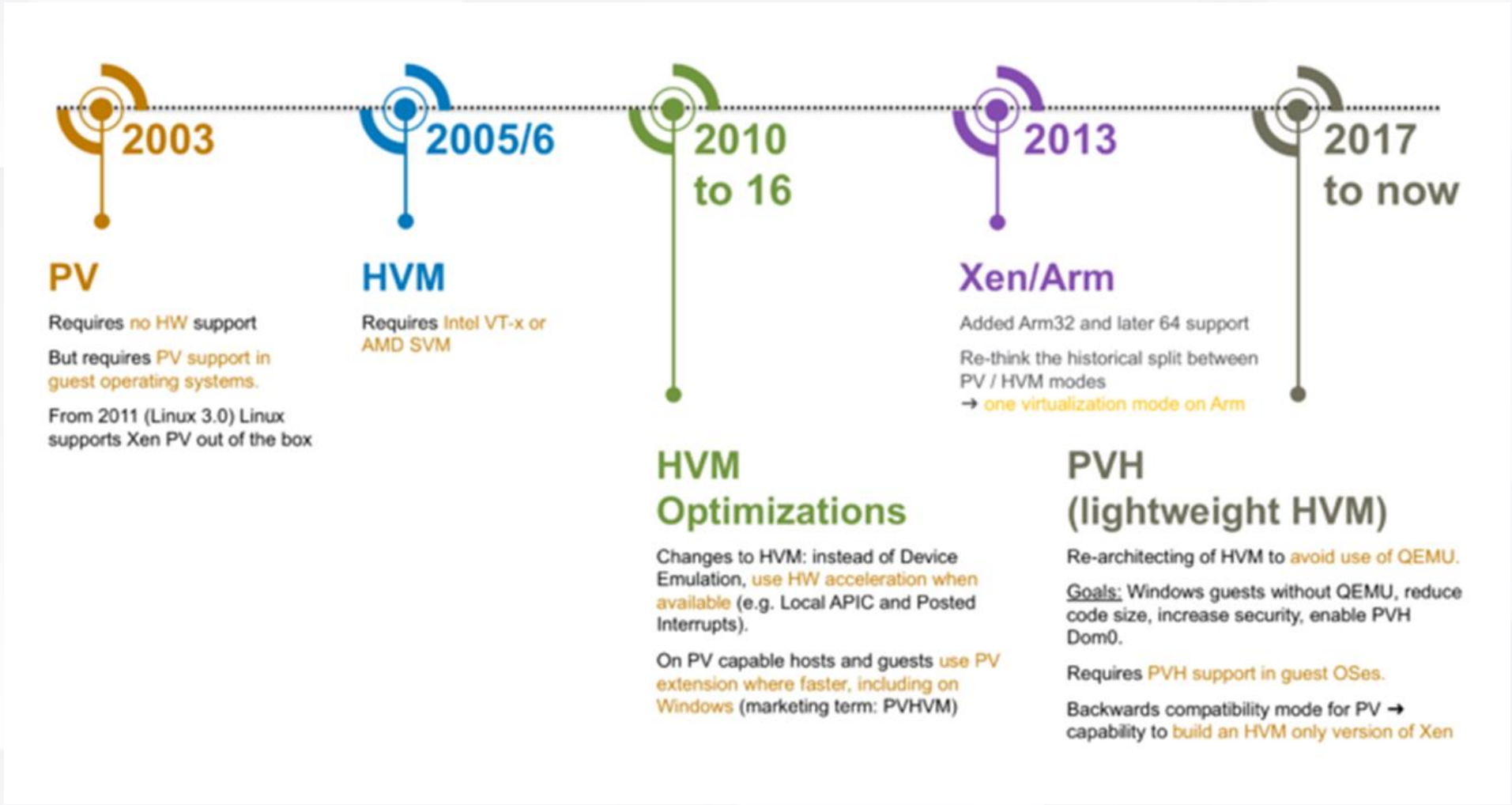


02

Type-1 VMM: Xen

SJTU

Type1虚拟机监控器的典型——Xen



Xen从半虚拟化到硬件辅助虚拟化的技术背景

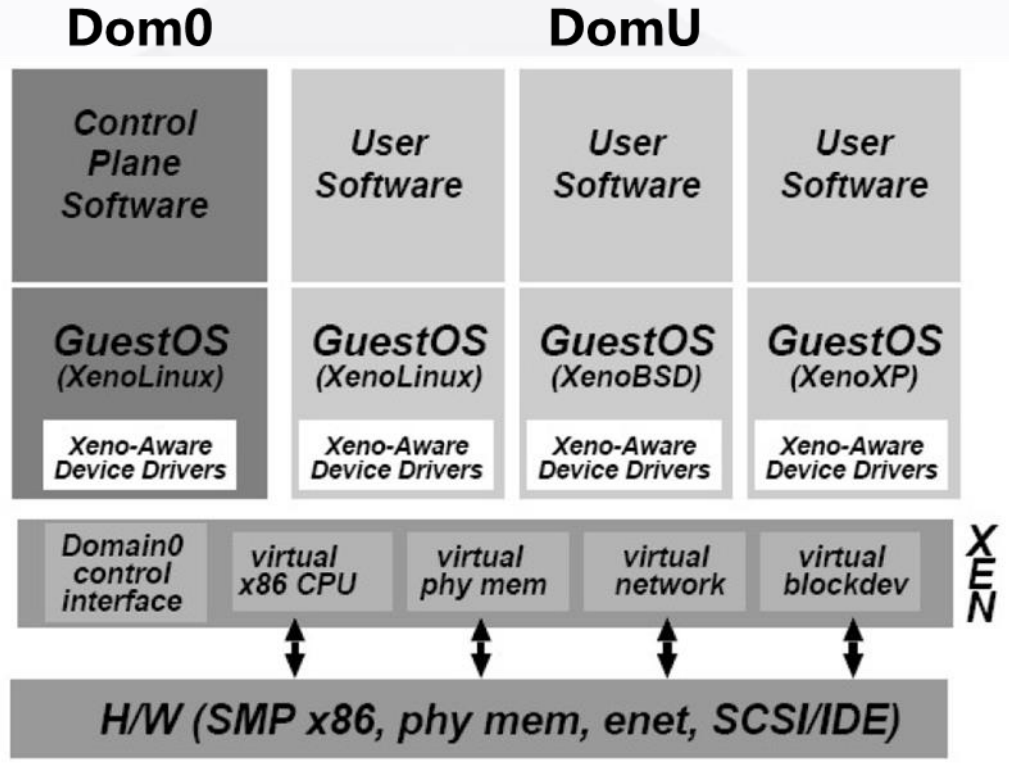
- **CPU虚拟化技术的发展过程**
  - Trap & Emulate
  - Binary Translation
  - Para-Virtualization
  - Hardware Assisted Virtualization （e.g. Intel VT-x or AMD-V, ARM EL2）
- **内存虚拟化发展的过程**
  - Para-Virtualization
  - Shadow Page Table
  - Hardware Assisted Virtualization （e.g. Extended Page Table）
- **IO虚拟化发展的过程**
  - Device Emulation
  - Para-Virtualization
  - Mediated Pass-through
  - Hardware Assisted Virtualization (e.g. SR-IOV)

Xen半虚拟化的设计

- Xen对于x86指令集的使用问题的相关解决方案
  - 不允许所有的Guest OS直接使用和处理敏感指令
  - 将所有那些不会trap到VMM（Xen）中的敏感指令都改成会trap的指令
- Guest OS需要通过“hypercall”来与系统资源交互
  - 同时允许Hypervisor保护虚拟机之间的隔离性
- 所有的Exception会被Xen里面的handler直接处理
  - 一些OS system call的Fast handler可以直接invoke
  - 例如Page Fault的handler就需要针对内存虚拟化进行修改
- Guest Os需要针对架构作出一定的变化
  - 例如编译内核时Compile for ARCH=xen instead of ARCH=i686（x86_64）
  - 物理机上的代码需要有大约1.36%被修改

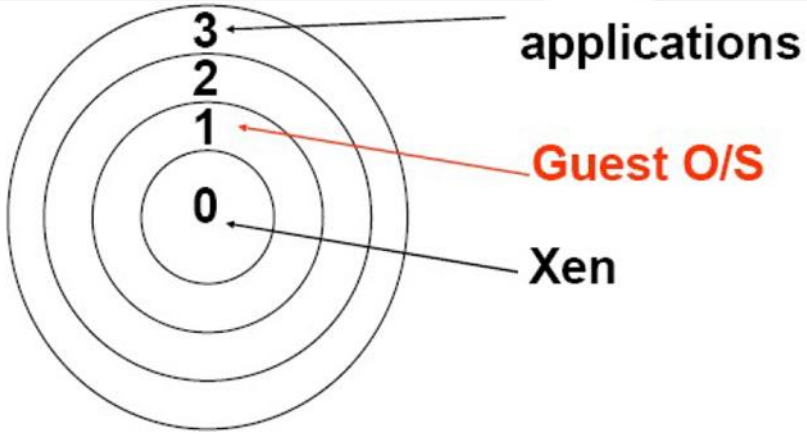
Xen的整体架构（以SOSP'03论文为例）

- 几个重要的概念:
  - Guest Domains/Virtual Machines
  - The Control Domain (or Domain 0)
    - System Services
    - 原生Device Drivers
    - 虚拟Device Drivers (作为DomU的backend)
    - Toolstack
  - Xen Project-enabled operating systems



Xen的CPU虚拟化

- **x86 提供了4个rings (even VAX processor provided 4)**
  - 一般OS只需要使用ring 0 and 3；
  - Guest OS需要运行在ring 1

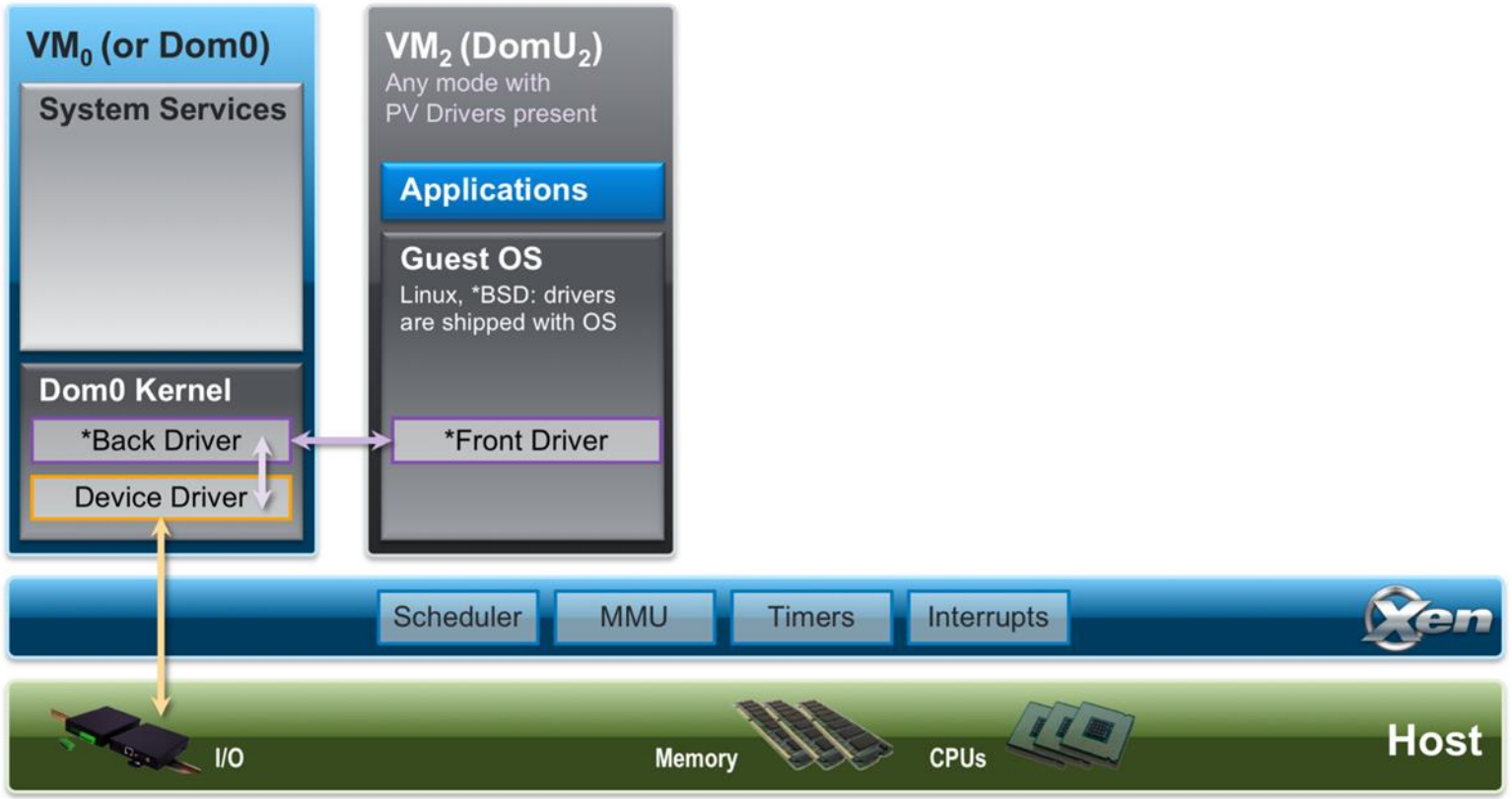


- **设计了一些新的hypercall**
  - #define __HYPERVISOR_set_trap_table 0
  - #define __HYPERVISOR_mmu_update 1
  - #define __HYPERVISOR_sysctl 35
  - #define __HYPERVISOR_domctl 36

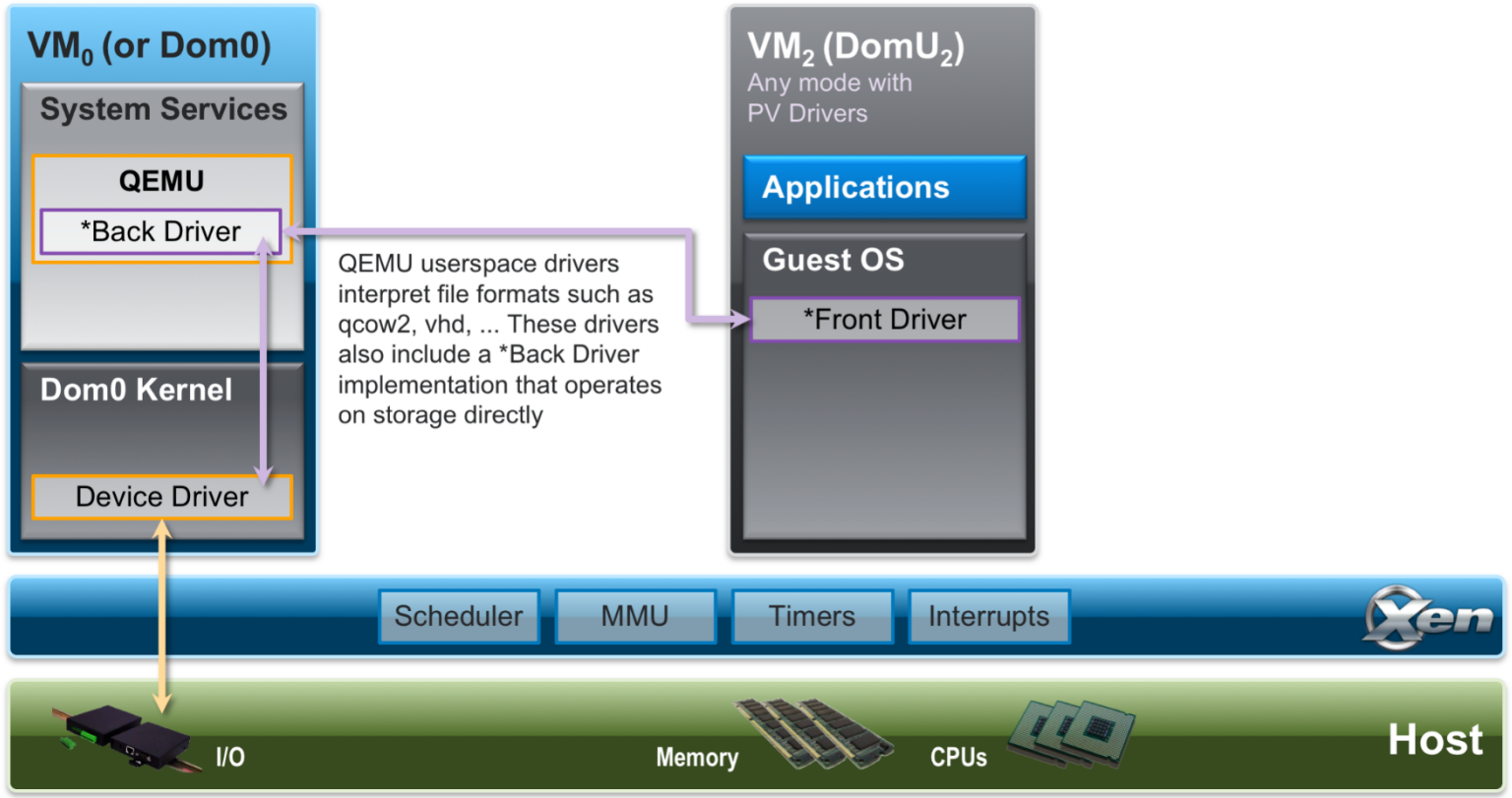
Xen的Memory虚拟化

- Xen半虚拟化（PV）设计下内存的管理
  - 虚拟机可以直接读访问硬件上的页表（不需要通过Trap）
  - 虚拟机对页表的写会被Trap到hypervisor。
  - 每个虚拟机实际使用的内存页是不连续的。
- Xen借助硬件辅助虚拟化（HVM）的内存管理
  - 引入了Shadow Page Table，这里借助了VT技术中的新指令。

Xen的IO虚拟化


VM0 (or Dom0)
System Services
Dom0 Kernel
*Back Driver
Device Driver
VM2 (DomU2)
Any mode with
PV Drivers present
Applications
Guest OS
Linux, *BSD: drivers
are shipped with OS
*Front Driver
Scheduler
MMU
Timers
Interrupts
Xen
I/O
Memory
CPUs
Host

Xen的IO虚拟化


VM0 (or Dom0)
System Services
QEMU
*Back Driver
Dom0 Kernel
Device Driver
QEMU userspace drivers interpret file formats such as qcow2, vhd, ... These drivers also include a *Back Driver implementation that operates on storage directly
VM2 (DomU2)
Any mode with PV Drivers
Applications
Guest OS
*Front Driver
Scheduler
MMU
Timers
Interrupts
Xen
I/O
Memory
CPUs
Host

02

Type-2 VMM: KVM & QEMU

Type2虚拟机监控器的典型——KVM

- KVM 全称是 基于内核的虚拟机（Kernel-based Virtual Machine）
  - 它最早由 Quramnet 开发，该公司于 2008年被 Red Hat 收购。
  - 目前，Red Hat，Intel等公司是KVM最主要的contributor。
  - 它支持 x86 (32 and 64 位), s390, Powerpc 等架构 CPU。
  - 它从 Linux 2.6.20 起就作为一内核模块（kvm.ko）被merge进入 Linux 内核的主干。
  - 它需要支持硬件虚拟化扩展的 CPU，也就是需要有Intel VT或AMD-V。
- 使用KVM，一般最常用的就是QEMU/KVM同时使用
  - 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
  - 随后QEMU支持模拟众多设备，是作为虚拟化技术中IO虚拟化管理重要的一个工具

KVM虚拟化架构介绍

客户机系统（Guest OS）

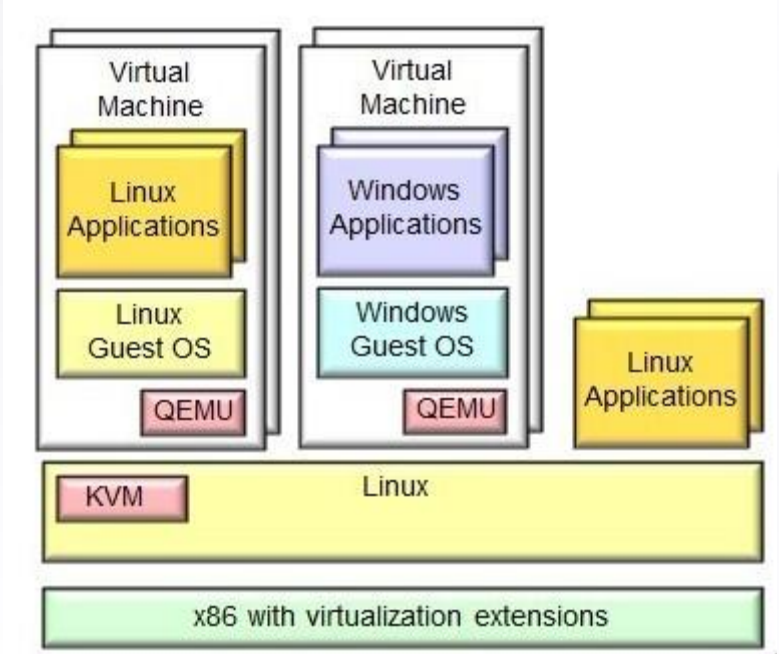
- 包括CPU（vCPU）、内存、驱动（Console、网卡、I/O 设备驱动等），被 KVM 置于一种受限制的 CPU 模式下运行。

KVM内核模块

- 运行在内核空间，提供 CPU 和内存的虚级化，以及客户机的 I/O 拦截。Guest 的 I/O 被 KVM 拦截后，交给 QEMU 处理。

QEMU

- 修改过的被 KVM 虚机使用的 QEMU 代码，运行在用户空间，提供硬件 I/O 虚拟化，通过 IOCTL /dev/kvm 设备和 KVM 交互。



QEMU

- 开源软件，不属于KVM，包含整套虚拟机实现技术。
- 采用纯软件方式实现虚拟机，性能很低。
- 为了简化开发，KVM没有选择从零起步，而是对QEMU进行了修改和利用。

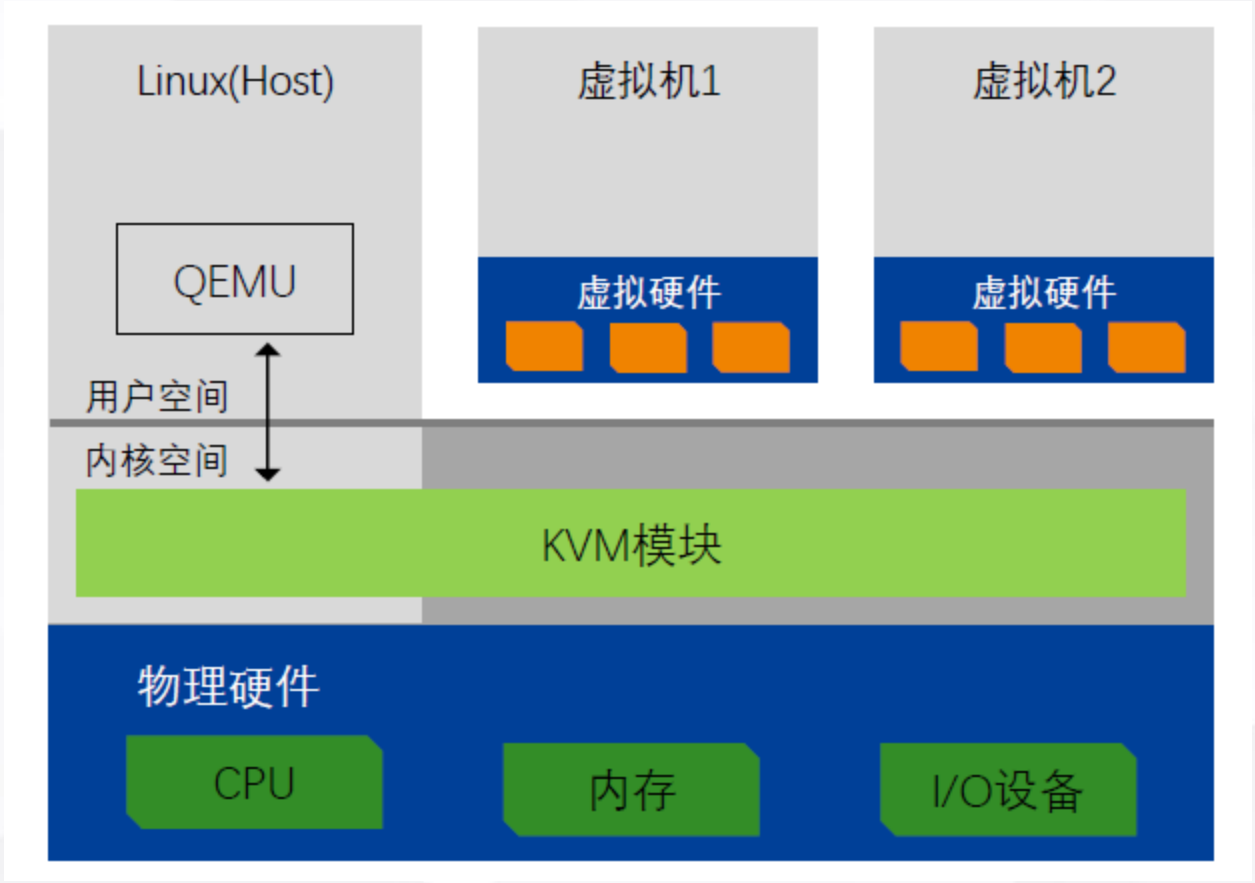
KVM

- 借用了现有的QEMU来完成设备模拟，仅需专注于对性能要求较高的CPU虚拟化、内存虚拟化

QEMU

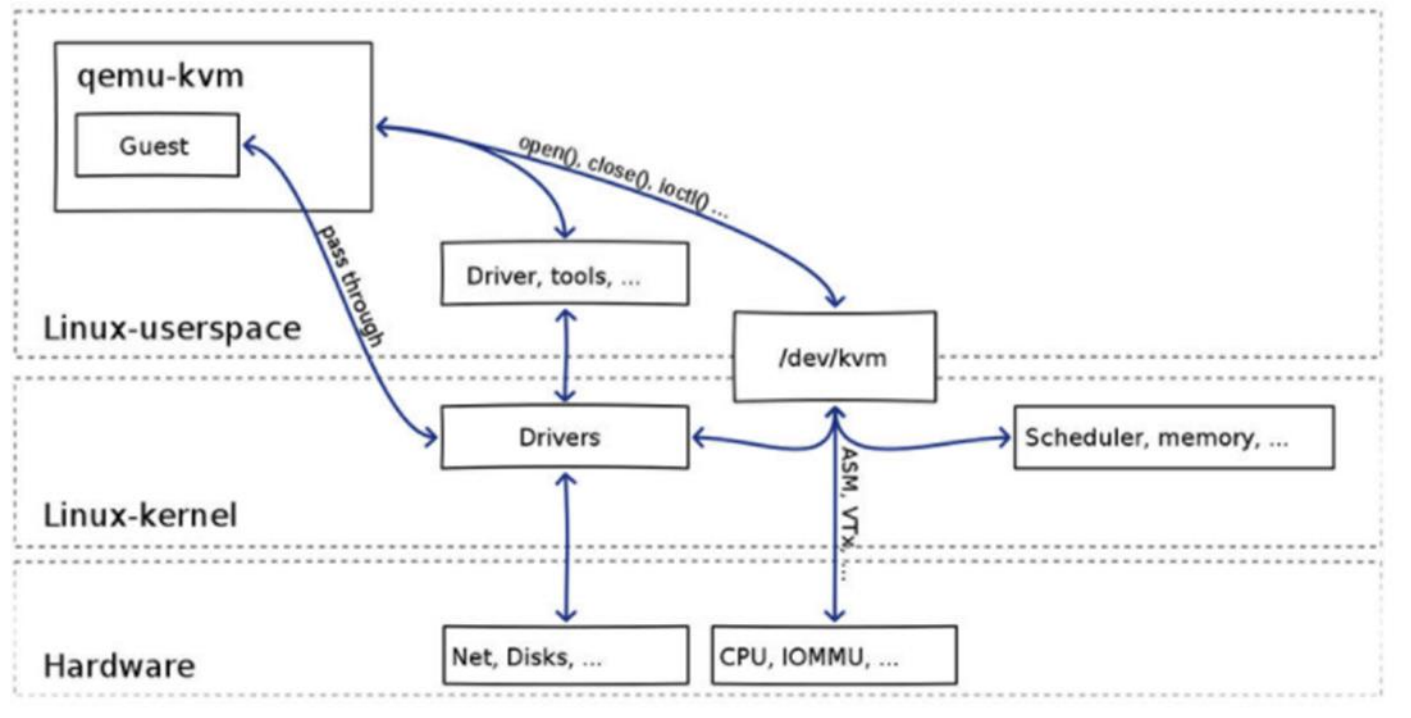
- 使用了KVM的虚拟化技术，为自己的虚拟机提供硬件虚拟化的加速，极大的提高性能

QEMU使用KVM的用户态接口



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传递命令：CREATE_VM, CREATE_VCPU, KVM_RUN等



KVM虚拟化

- KVM的CPU虚拟化
  - 基于有虚拟化扩展的硬件进行虚拟化
  - 以Intel VT-x为例，KVM需要找到每个VCPU对应的VMCS，通过VMCS结构保存VCPU的相关状态
- KVM的内存虚拟化
  - 基于有虚拟化扩展的硬件进行虚拟化
  - 以Intel 的Extended Page Table为例，需要通过EPT进行内存地址翻译
- KVM的设备IO虚拟化
  - 可以通过QEMU完成设备虚拟化的交互

Xen/KVM 对比的优劣

- Xen的主要优势
  - Type 1的hypervisor，或者说裸金属的虚拟化，可以提供**更高的性能。**
  - 可以提供较好的设备驱动的隔离性
  - 可以提供比较好的半虚拟化设备虚拟化，减少DomU的负担
  - 可以运行在一些不支持硬件扩展的机器上。
- KVM的主要优势
  - 其实际实现的过程中，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，设备虚拟化的支持也更加方便。
  - 简单易用，它作为Linux内核的一个模块，可以非常方便的安装，卸载，修改等，而Xen必须重新安装整个操作系统。
  - KVM作为Linux内核的一部分，开源生态更好。

KVM超过Xen 的主要原因：

- KVM 支持自 2.6.20 版开始已自动包含在每个 Linux 内核中。在 Linux 内核 3.0 版之前，将 Xen 支持集成到 Linux 内核中需要应用大量的补丁，并仍然无法保证每个可能硬件设备的每个驱动程序都能在 Xen 环境中正确工作。
- Xen 支持所需的内核源代码补丁仅提供给特定的内核版本，这阻止了 Xen 虚拟化环境利用仅在其他内核版本中可用的新驱动程序、子系统及内核修复和增强。KVM 在 Linux 内核中的集成使它能够自动利用新 Linux 内核版本中的任何改进。
- Xen 要求在物理虚拟机服务器上运行一个特殊配置的 Linux 内核，以用作在该服务器上运行的所有虚拟机的管理域。KVM 可在物理服务器上使用在该物理系统上运行的 Linux VM 中使用的相同内核。
- Xen 的虚拟机管理程序是一段单独的源代码，它自己的潜在缺陷与它所托管的操作系统中的缺陷无关。因为 KVM 是 Linux 内核的一个集成部分，所以只有内核缺陷能够影响它作为 KVM 虚拟机管理程序的用途。

03

CPU虚拟化

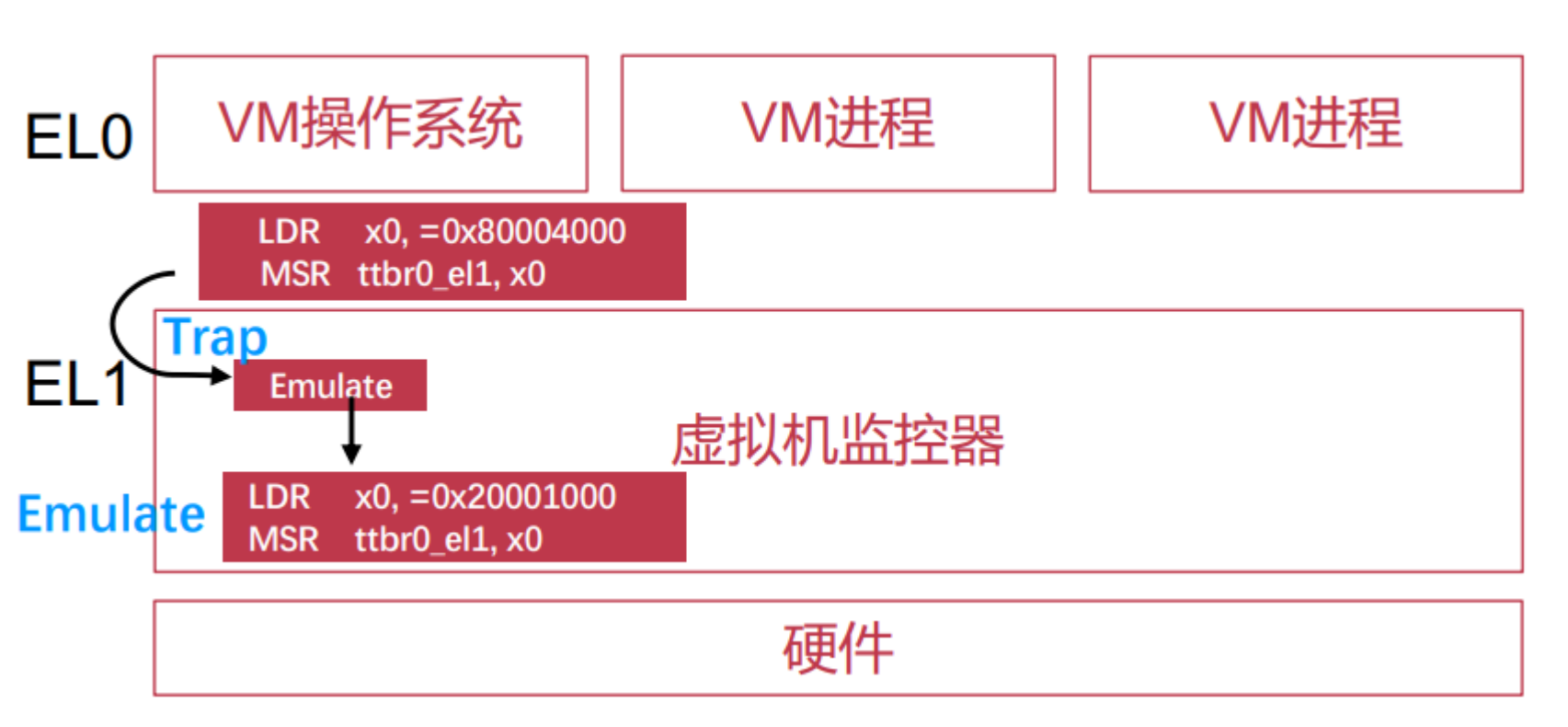
- 方法1：解释执行
- 方法2：二进制翻译
- 方法3：半虚拟化
- 方法4：硬件虚拟化（改硬件）

CPU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



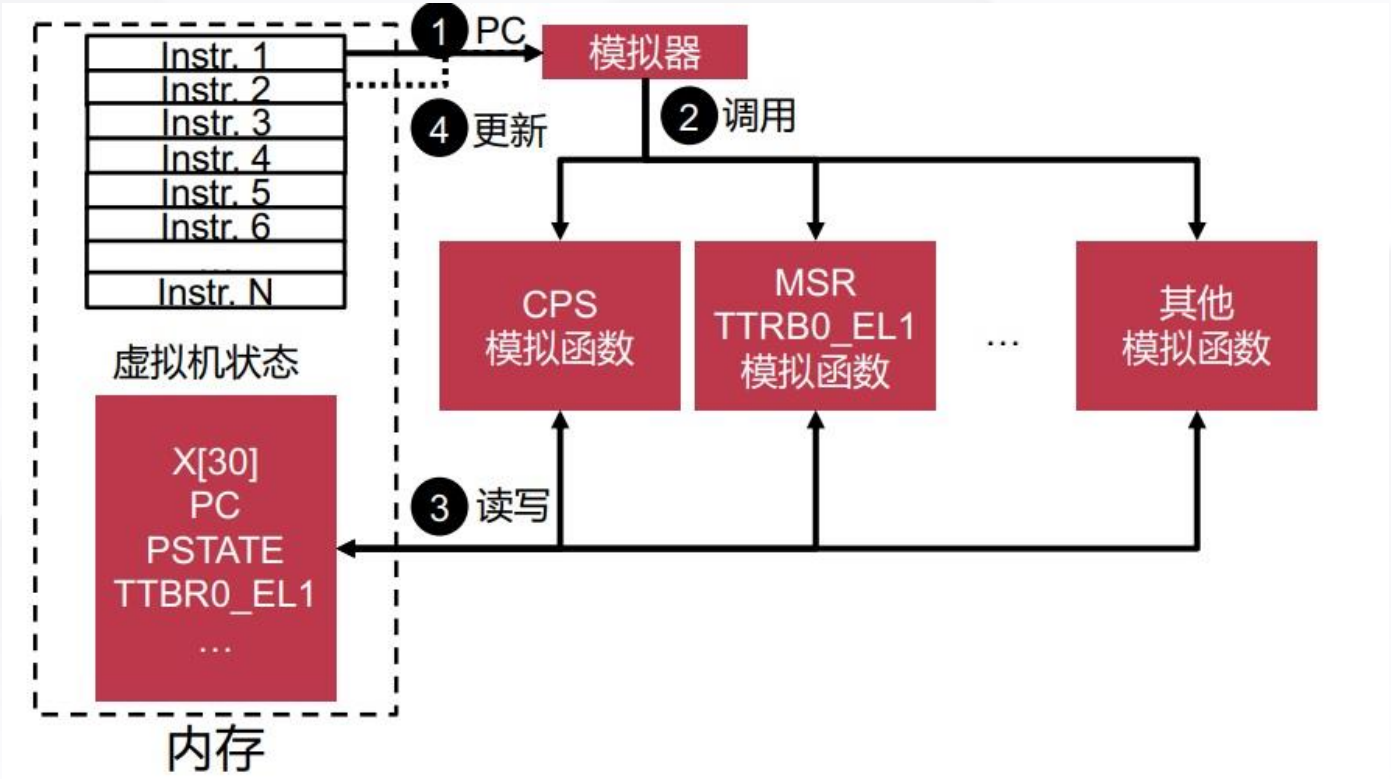
敏感指令处理

- 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）

解释执行

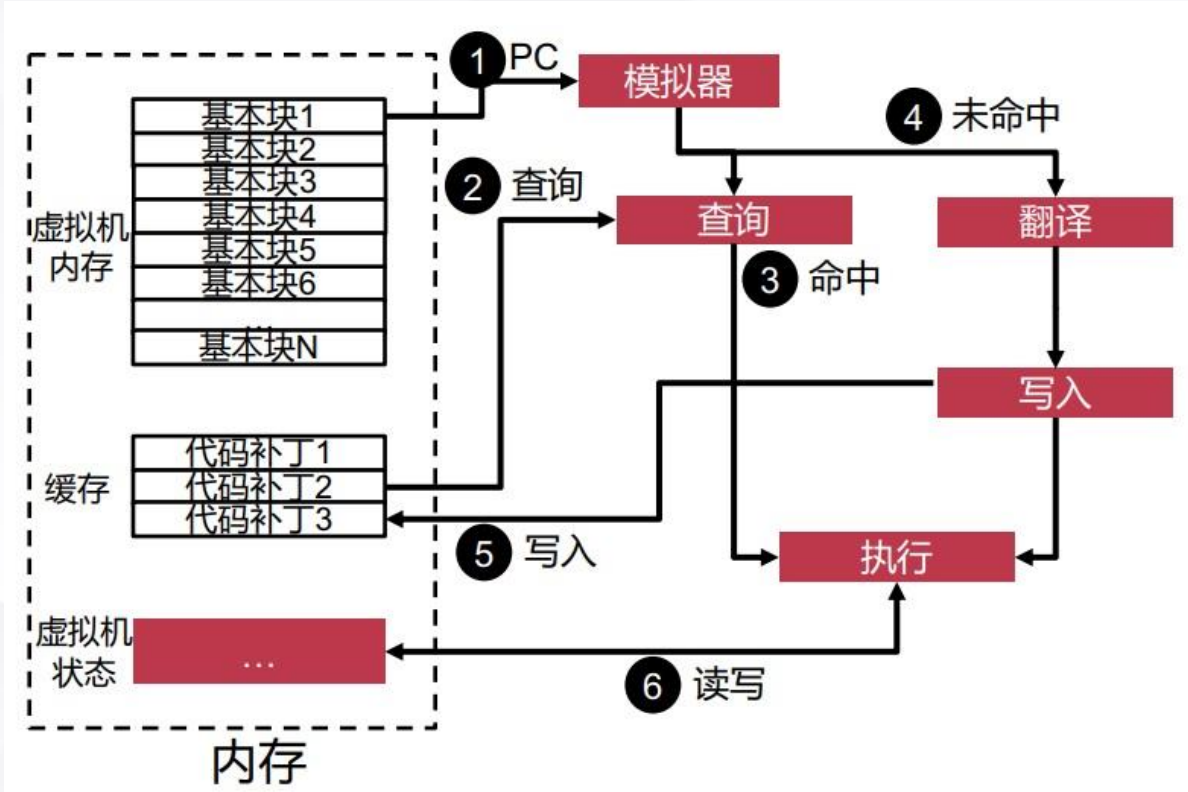
- 解决了敏感函数不下陷的问题
- 可以模拟不同ISA的虚拟机
- 易于实现、复杂度低

- 非常慢：任何一条虚拟机指令都会转换成多条模拟指令



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
- 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



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

硬件虚拟化

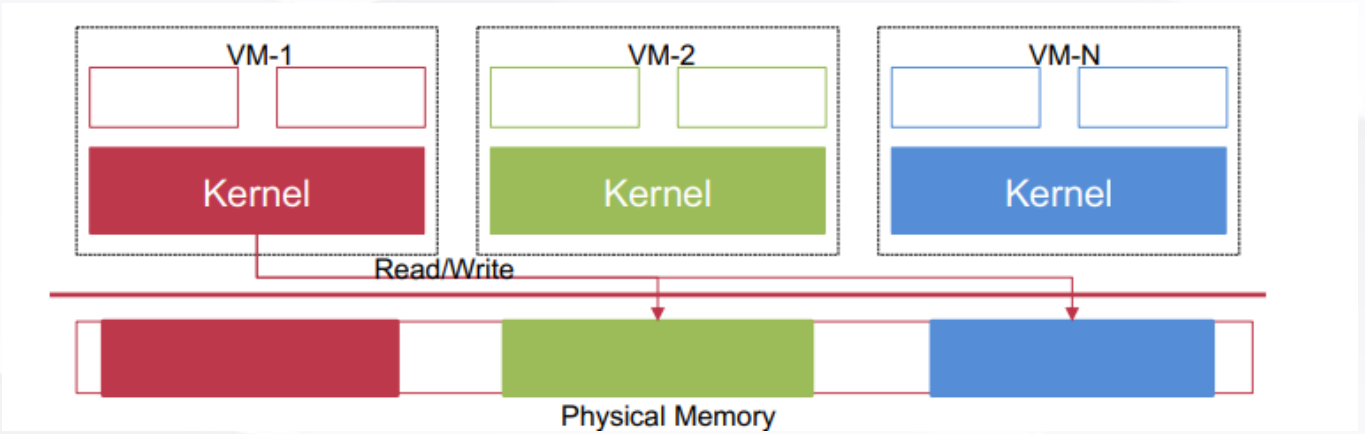
- x86和ARM都引入了全新的虚拟化特权级
- x86引入了root模式和non-root模式
  - Intel推出了VT-x硬件虚拟化扩展
  - Root模式是最高特权级别，控制物理资源
  - VMM运行在root模式，虚拟机运行在non-root模式
  - 两个模式内都有4个特权级别：Ring0~Ring3
- ARM引入了EL2
  - VMM运行在EL2
  - EL2是最高特权级别，控制物理资源
  - VMM的操作系统和应用程序分别运行在EL1和EL0

04

内存虚拟化

为什么需要内存虚拟化?

- 操作系统内核直接管理物理内存
  - 物理地址从0开始连续增长
  - 向上层进程提供虚拟内存的抽象
- 如果VM使用的是真实物理地址
- 为虚拟机提供虚拟的物理地址空间
  - 物理地址从0开始连续增长
  - 隔离不同虚拟机的物理地址空间
  - VM-1无法访问其他VM的内存



怎么实现内存虚拟化?

- 影子页表(Shadow Page Table)
- 直接页表(Direct Page Table)
- 硬件虚拟化

四个地址

- GVA（Guest Virtual Address）
  - 虚拟机程序所访问的地址
- GPA（Guest Physical Address）
  - Guest OS所“认为”的物理地址
- HVA（Host Virtual Address）
  - Host OS上的程序所访问的地址（包括Hypervisor）
- HPA（Host Physical Address）
  - 真正的物理地址，用于索引DRAM上的数据

影子页表（Shadow Page Table）

- Guest OS所维护的页表负责传统的从guest虚拟地址GVA到guest物理地址GPA的转换。如果MMU直接装载guest OS所维护的页表来进行内存访问，那么由于页表中每项所记录的都是GPA，MMU无法实现地址翻译。
  - 解决方案：影子页表 (Shadow Page Table)
  - 作用：GVA直接到HPA的地址翻译,真正被VMM载入到物理MMU中的页表是影子页表；
- Guest OS“认为”它建立了从GVA->GPA的映射，并且硬件会根据该页表寻址；
- 实际上Hypervisor截取了Guest OS对页表的修改，并将真实的页表改为GVA->HPA的映射。

影子页表的特点

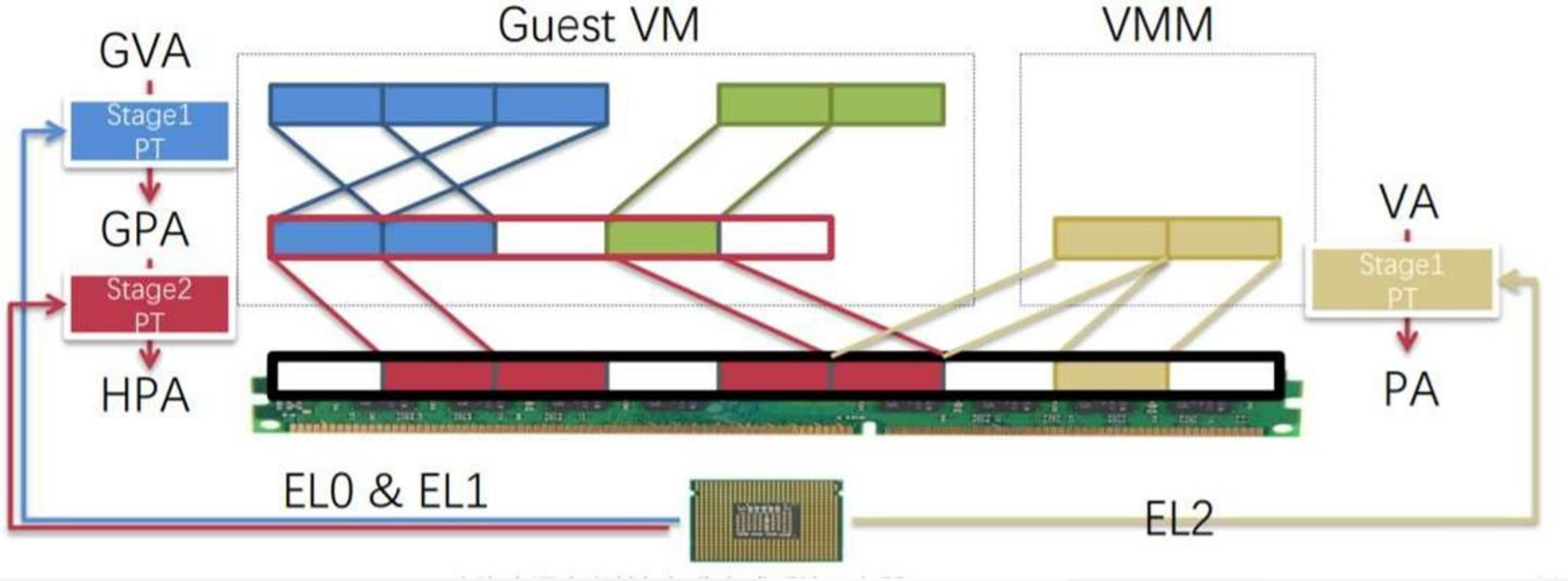
- 优点：
  - 从GVA->HPA一步到位
  - 软件实现：灵活性
- 缺点：
  - 每一个Guest进程都有一个SPT（大量的内存消耗）
    - 为什么每个进程都需要有一个SPT?
  - 每一次页表修改都会导致VMExit和TLB flush

硬件虚拟化对内存翻译的支持

- Intel VT-x和ARM硬件虚拟化都有对应的内存虚拟化
  - Intel Extended Page Table (EPT)
  - ARM Stage-2 Page Table (第二阶段页表)
- 新的页表
  - 将GPA翻译成HPA
  - 此表被VMM直接控制
  - 每一个VM有一个对应的页表

第二阶段页表

- 两阶段翻译
  - 第一阶段：GVA->GPA，硬件根据Guest页表进行翻译，完全不受 Hypervisor控制
  - 第二阶段：GPA->HPA，硬件根据EPT进行翻译，受Hypervisor控制



TLB变得更加重要

- 复习：TLB用来缓存VA->PA的结果
  - 不再需要遍历二级/四级页表
  - 当页表修改后，软件（操作系统或Hypervisor）必须调用TLB Flush来刷新 TLB，否则TLB中将存有旧的映射关系
- 在VT-x中，有四项种LB表项可以存储在TLB中：
  - HVA->HPA (Host PT)
  - GVA->GPA (Guest PT)
  - GPA->HPA (EPT)
  - GVA->HPA (Combined PTE)

VMID

- 假设有多个VM使用TLB，而其中一个VM修改了其EPT
  - 为了保证页表与TLB的一致性，Hypervisor必须刷新TLB
  - 但在大多数时候，只有一个VM的EPT被修改，但刷新操作会让TLB中所 有内容都失效
- VMID
  - 对每一个VM制定一个ID，刷新TLB时不会影响其他VM
  - 提升性能

EPT Violation

- 复习：当CPU发现一个虚拟地址 在页表中没有所对应的项， MMU会向CPU注入一个缺页中断（Page Fault）

```
// 物理内存此时并未分配
void *ptr = malloc(size);
// 触发一个#PF，并在中断处理函 数分配物理内存
memset(ptr, 0, size);
```

- 当VT-x硬件发现GPA没有对应的项，或者CPU违反了EPT表项所指定的规则（如写只读页），它会向CPU注入一种特殊的VMExit：EPT Violation；
- 在大多数情况下，Hypervisor需要分配物理内存并设置EPT
  - Type-I：Hypervisor实现的内存分配系统
  - Type-II：Linux/Windows提供的内存分配系统

谢谢！

饮 水 思 源　爱 国 荣 校