第二章：虚拟化技术及其实现方法概述

2021.9

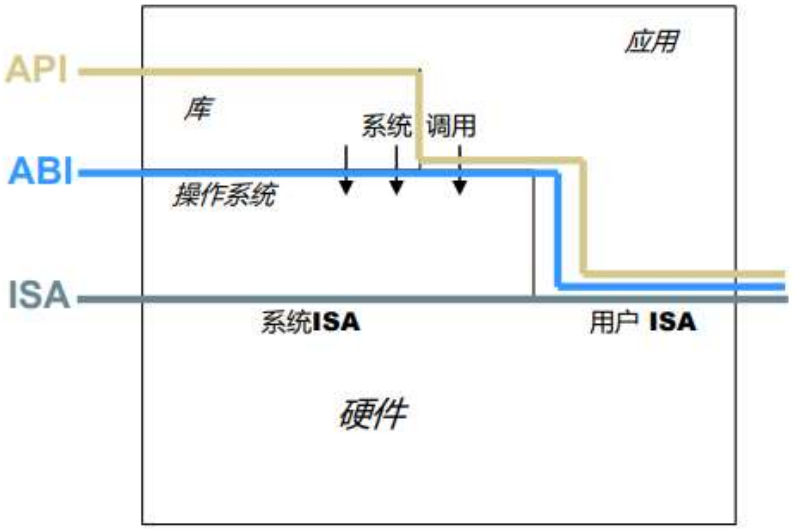
上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Contents

1. 虚拟化技术总体介绍

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

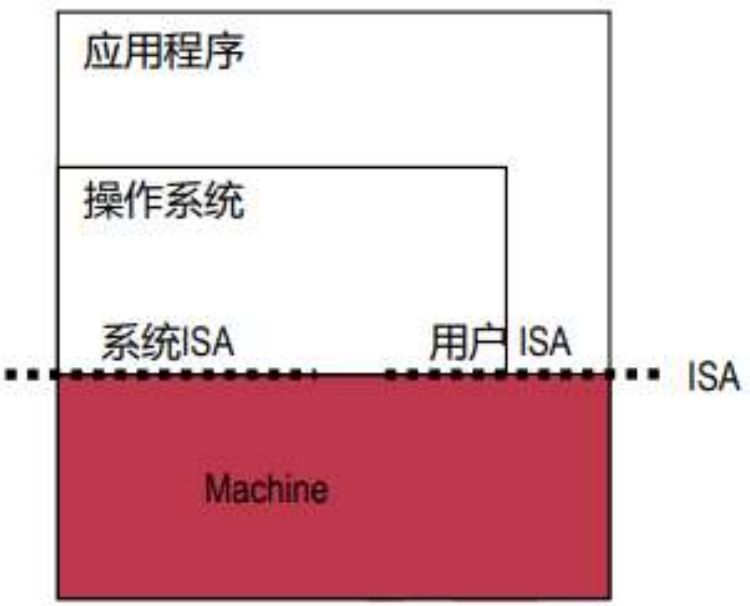
从操作系统中的接口层次看虚拟化的实现



API – application programming interface
ABI – application binary interface
ISA – instruction set architecture

- **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

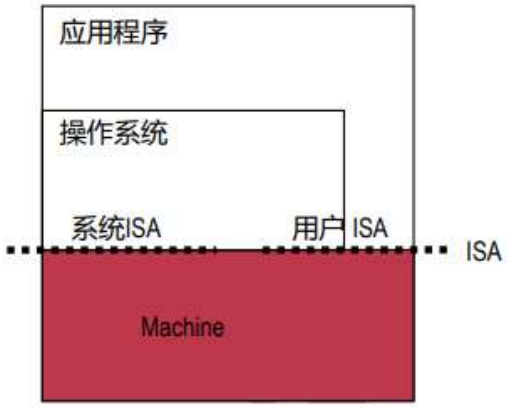
系统软件和物理硬件之间的关系

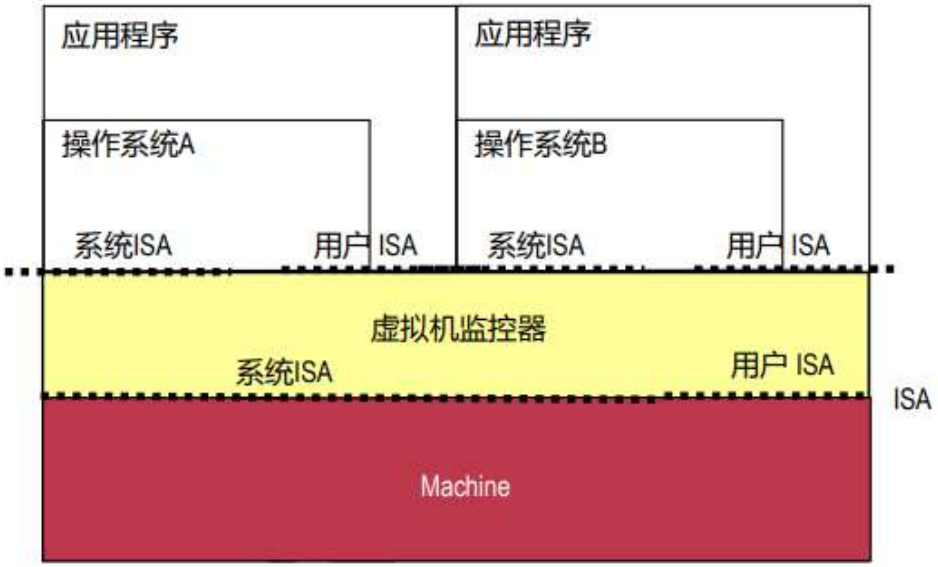


- **操作系统如何看待它管理的一台“机器”呢?**
  - ISA 提供了操作系统和Machine之间的界限
  - 系统软件就是通过ISA与硬件进行交互，也对硬件资源进行隔离
- **那么如何在操作系统的内部隔离出另外的操作系统，让它也可以与物理硬件交互呢?**
  - 虚拟化的具体实现技术就是为了解决这个问题。

虚拟化技术具体实现中最关注的两个部分

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





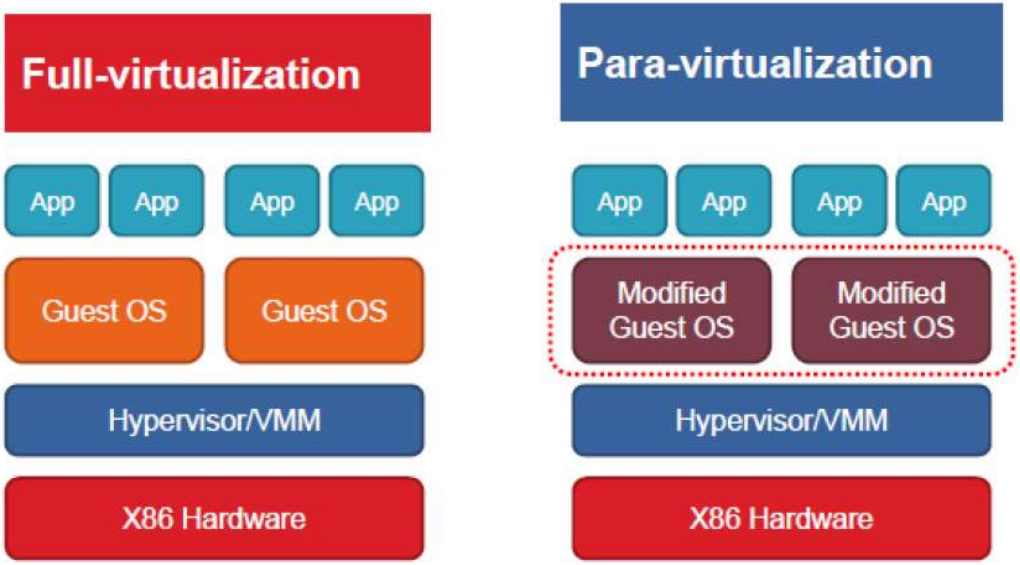
系统虚拟化实现的标准

- **高效系统虚拟化具体实现时，需要考虑三个特性**[1]
  - 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**

[1] Popek & Goldberg, 1974 “Formal Requirements for Virtualizable Third Generation Architectures”

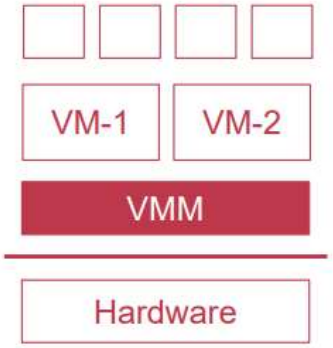
虚拟化技术实现中的分类

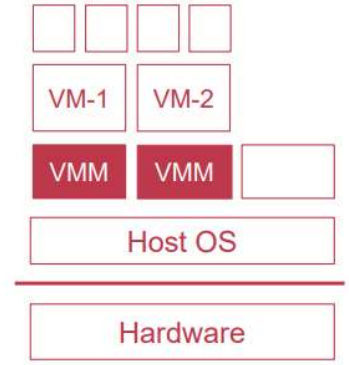
- **全虚拟化（Full Virtualization）** vs. **半虚拟化 (Para-Virtualization)**
  - 概念的变化
  - 从虚拟机的角度来分析
    - 虚拟机操作系统是否修改?



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

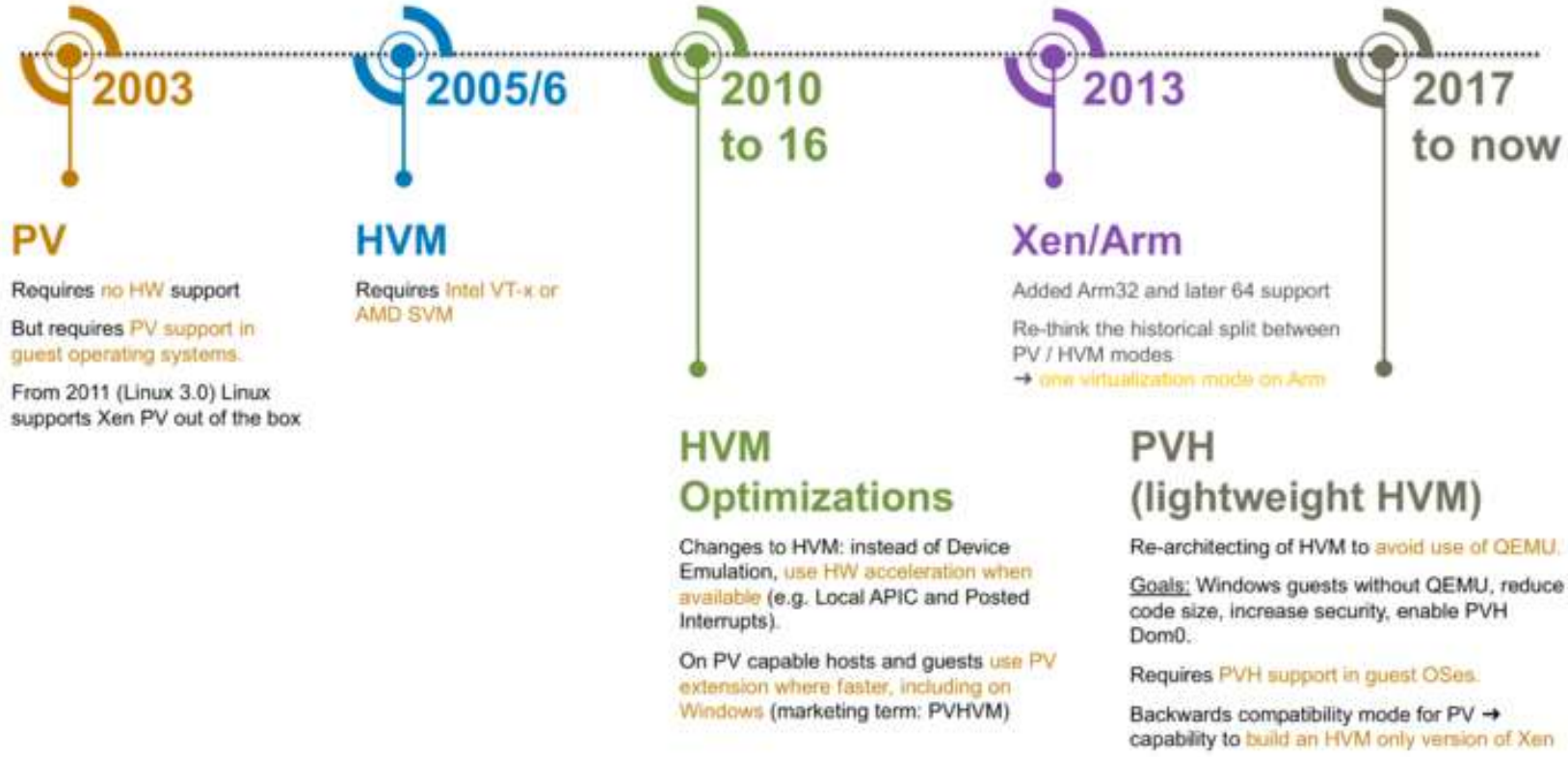




2. Xen/KVM 虚拟化技术架构

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

Type1虚拟机监控器的典型——Xen



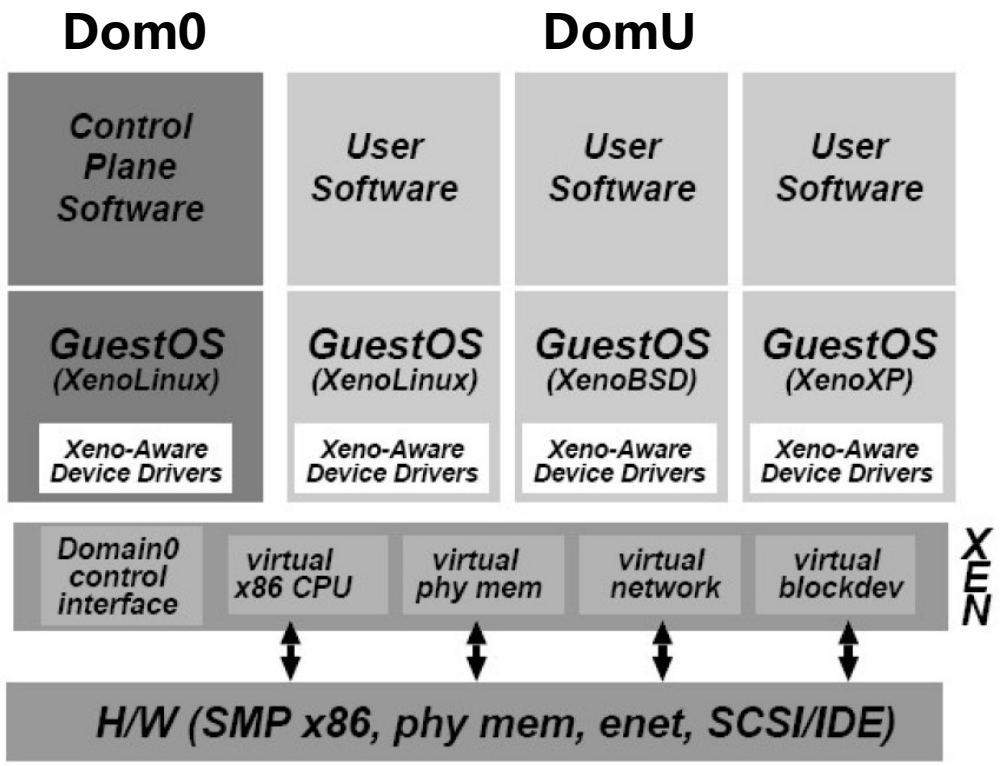
Xen从半虚拟化到硬件辅助虚拟化的技术背景

- **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

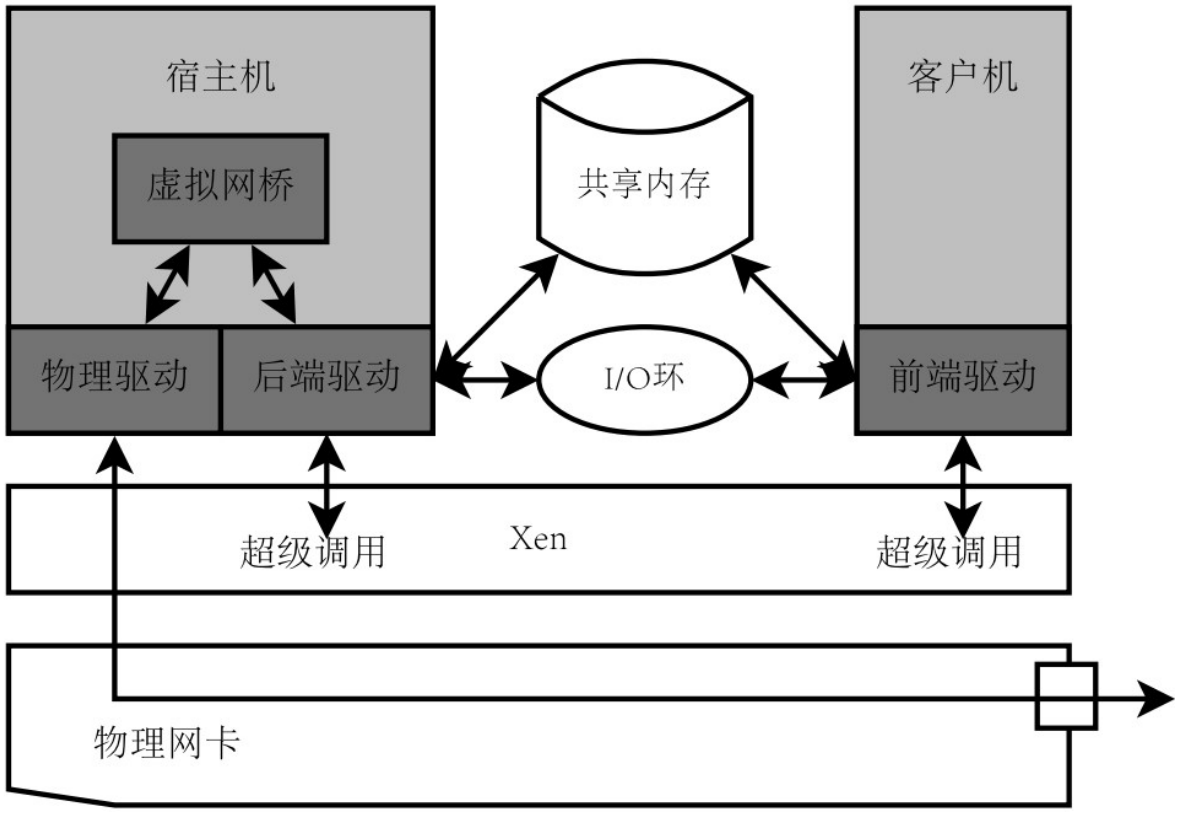
- Xen对于x86指令集的使用问题的相关解决方案
  - 不允许所有的Guest OS直接使用和处理敏感指令
  - 将所有那些不会trap到VMM（Xen）中的敏感指令都改成会trap的指令
- Guest OS需要通过“hypercall”来与系统资源交互
  - 同时允许Hypervisor保护虚拟机之间的隔离性
- 所有的Exception会被Xen里面的handler直接处理
  - 一些OS system call的Fast handler可以直接invoke
  - 例如Page Fault的handler就需要针对内存虚拟化进行修改
- Guest Os需要针对架构作出一定的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



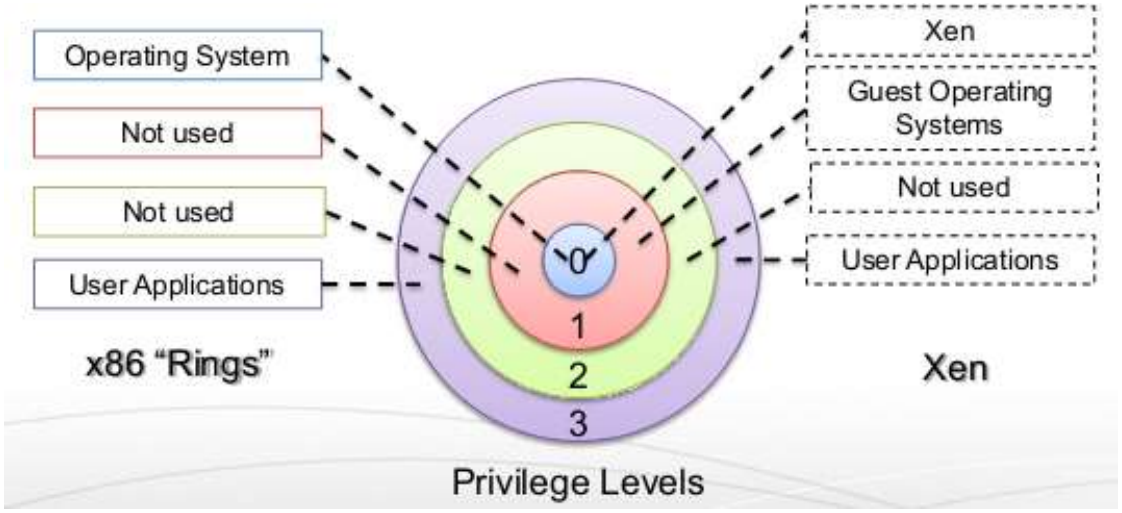
- **几个重要的概念:**
- Guest Domains/Virtual Machines
- The Control Domain (or Domain 0)
  - System Services
  - 原生Device Drivers
  - 虚拟Device Drivers (作为DomU的backend)
  - Toolstack
- Xen Project-enabled operating systems

Xen virtualization model



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

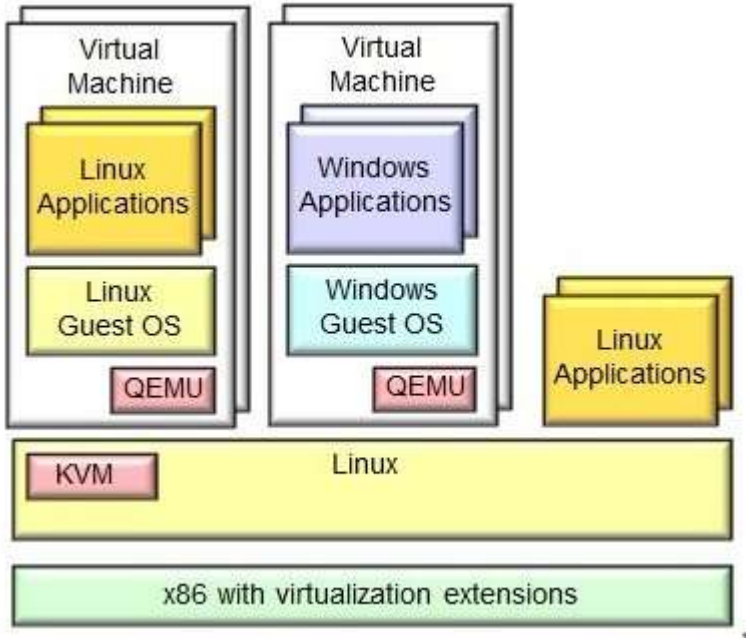
Xen的Memory虚拟化

- 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。

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。

KVM简介

基础知识

什么是KVM

Kernel-based Virtual Machine, 属于类型二VMM，采用完全虚拟化
内核中的一个模块，按需加载，与QEMU协同工作，构建虚拟化环境

KVM和QEMU的分工

KVM负责CPU虚拟化，内存虚拟化，以及一些对性能要求比较高的虚拟设备；大部分外设（网卡、硬盘等）交由用户空间的QEMU来模拟

KVM简介

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。

我是KVM

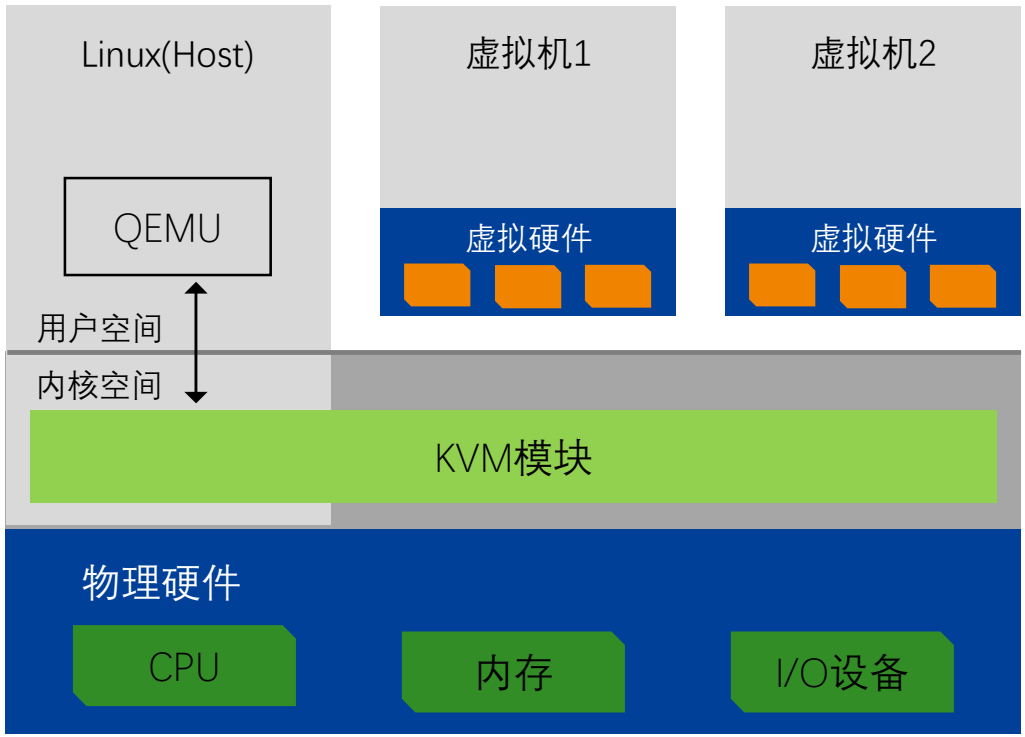
我借用了现有的QEMU来完成设备模拟，仅需专注于对性能要求较高的CPU虚拟化、内存虚拟化

我是QEMU

我使用了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

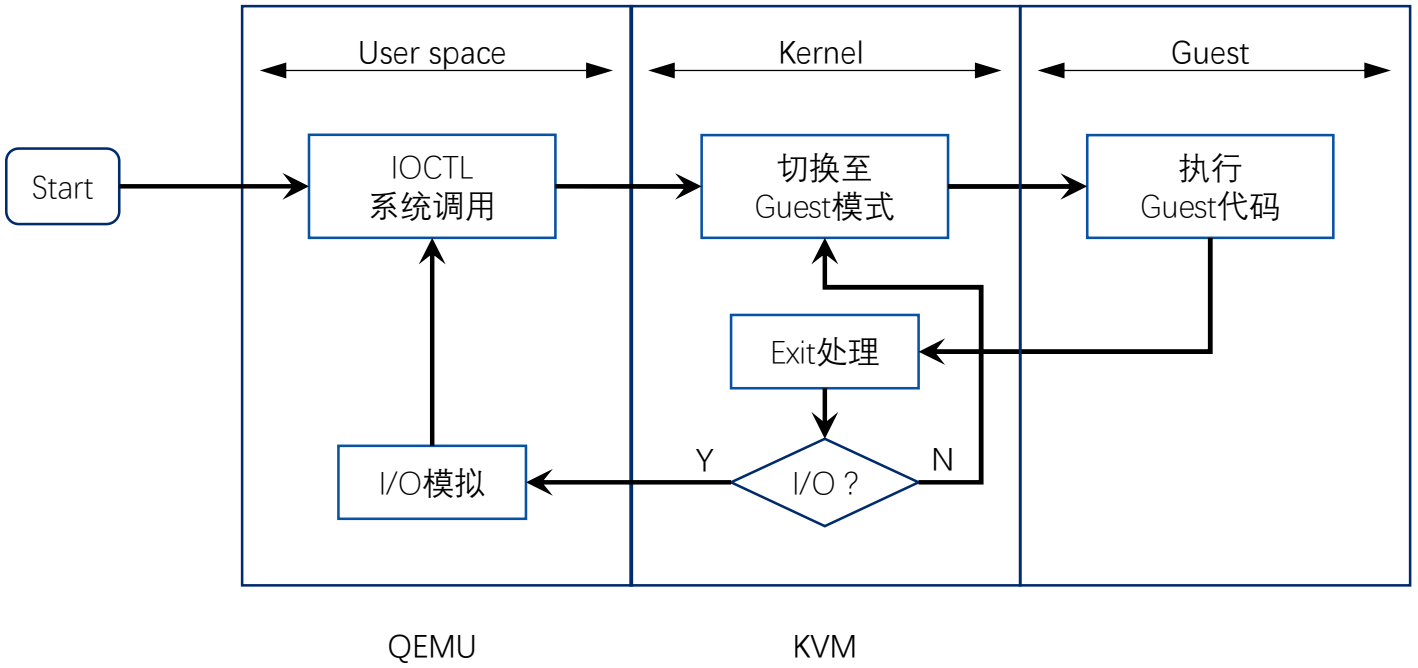
KVM简介

基本架构


Linux(Host)
虚拟机1
虚拟机2
QEMU
虚拟硬件
虚拟硬件
用户空间
内核空间
KVM模块
物理硬件
CPU
内存
I/O设备

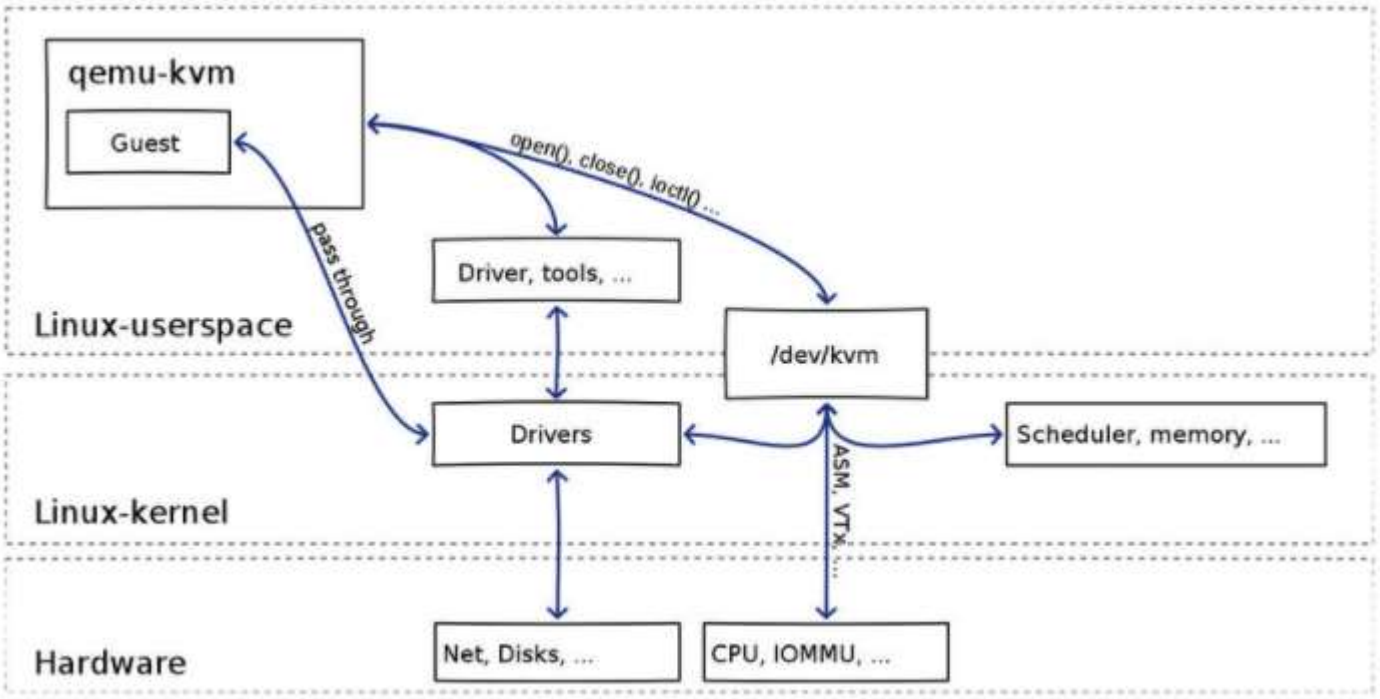
KVM简介

工作原理


User space
Kernel
Guest
Start
IOCTL
系统调用
切换至
Guest模式
执行
Guest代码
Exit处理
I/O模拟
Y
I/O ?
N
QEMU
KVM

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（本章后续将会介绍）
  - 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（本章后续将会介绍）
  - 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（下一章将会介绍）
  - 可以通过QEMU完成设备虚拟化的交互

3. Xen/KVM 的对比与应用

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

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。**

云计算中虚拟化技术

- **云计算中虚拟化的主要优势**
  - 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
- **开源虚拟化技术XEN和KVM等在云计算中的使用情况**
  - 2006年，Xen已经是Amazon Cloud绝对主力的虚拟化技术
  - 近年来，KVM已经超越 Xen，成为大多数企业环境首选的开源虚拟化技术；
  - 容器虚拟化技术也在云计算中扮演重要角色，例如Docker，Kata Container，Firecracker等。
  - 目前，有很多软硬件协同的新技术，正在提供裸金属客户机，提供绝对性能隔离的服务，其很重要的思路就是将虚拟化的软件栈卸载到一些新型高性能计算的硬件上。

云计算中虚拟化技术

- **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。

4. KVM 中的 CPU 虚拟化

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

非敏感/指令

- 非敏感指令

```
add %reg, $imm
div %reg0, %reg1
```

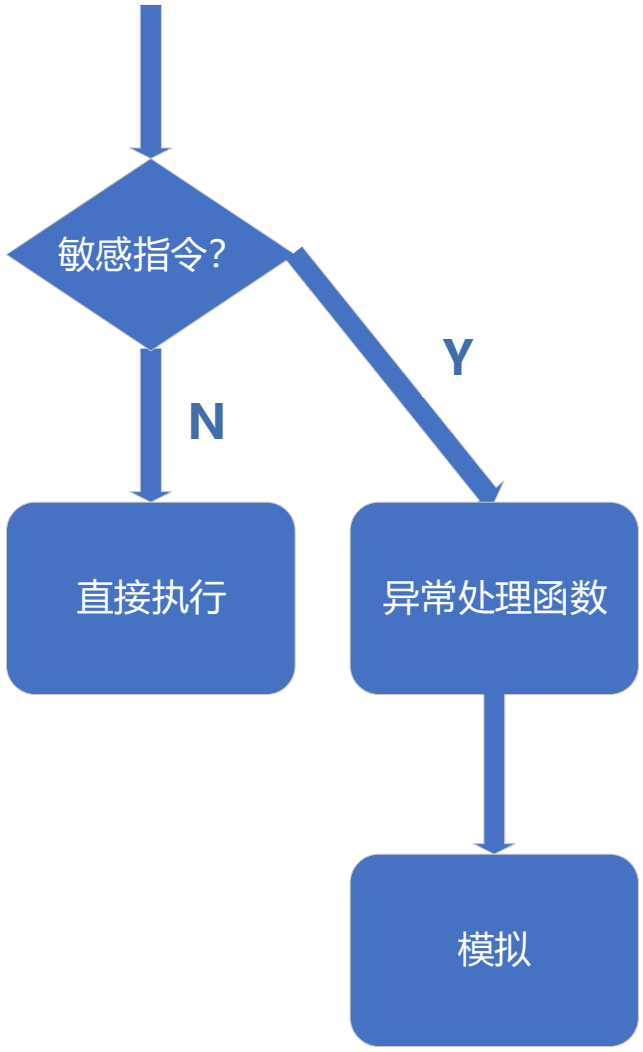
- 可以在低权限（ring3）执行
- 恶意攻击者无法使用这些指令进行攻击

- 敏感指令

```
mov mem, %cr3
rdmsr $imm
```

- **大部分**在高权限（ring0）执行
- 恶意攻击者可能可以利用这些指令进行恶意操作

第一版Hypervisor



问题

- 敏感非特权指令
  - 危险但是无法被模拟
- ARM `CPUSID`
  - 内核态执行可以关中断
  - 用户态执行被当做NOP
- 方法：
  - 解释执行
  - 二进制翻译
  - 半虚拟化
  - 硬件虚拟化

解释执行

```
addl %reg, $imm
```

```
switch (opnum): {
  case addl:
    regs[reg] += imm;
    break;
}
```

```
inb $port
```

```
switch (opnum): {
  case inb:
    simulate_inb(port);
    break;
}
```

二进制翻译

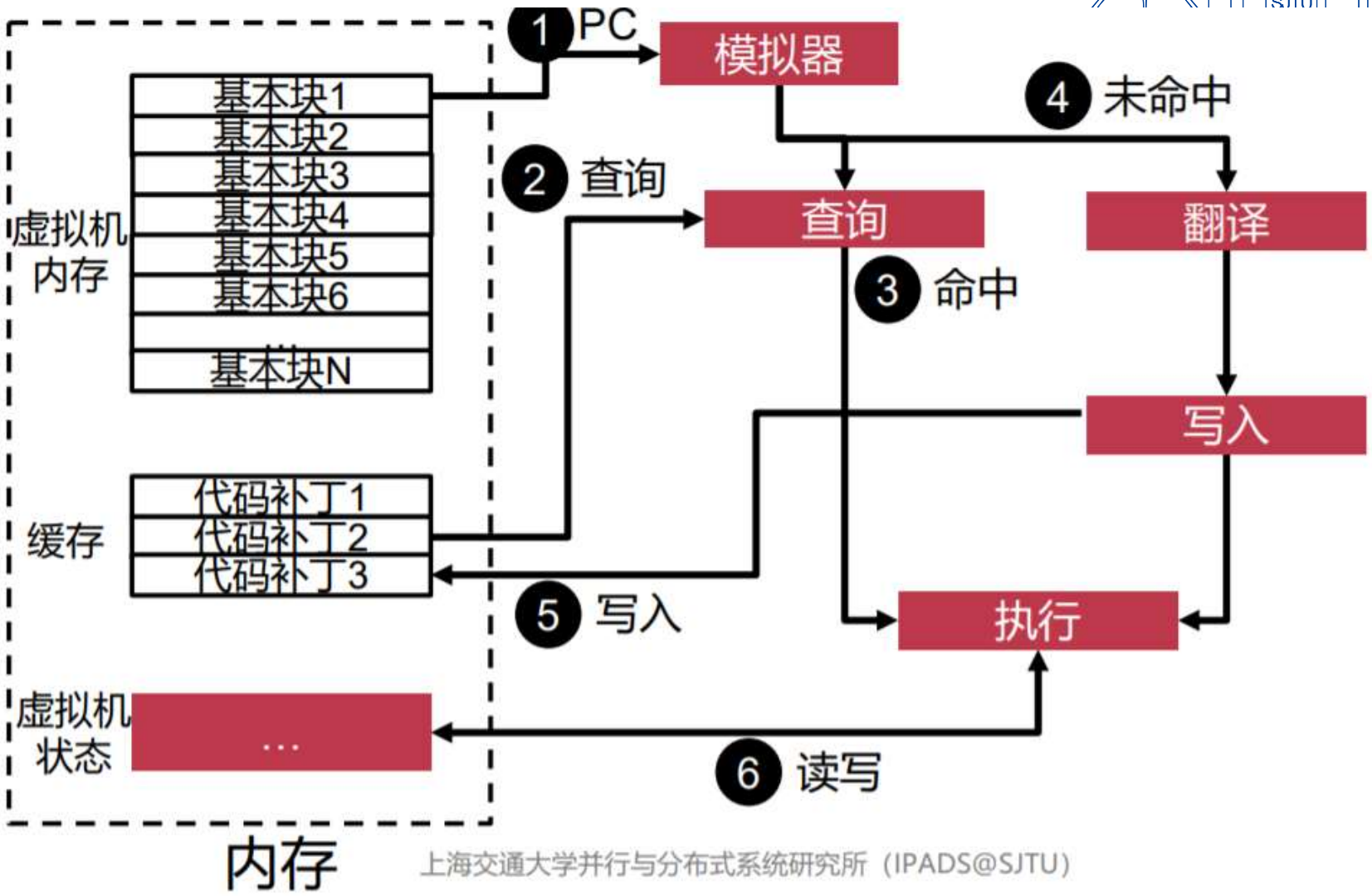
```
addl %reg, $imm
```

```
addl %reg, $imm
```

```
inb $port
```

```
switch (opnum): {
  case inb:
    simulate_inb(port);
    break;
}
```

二进制翻译

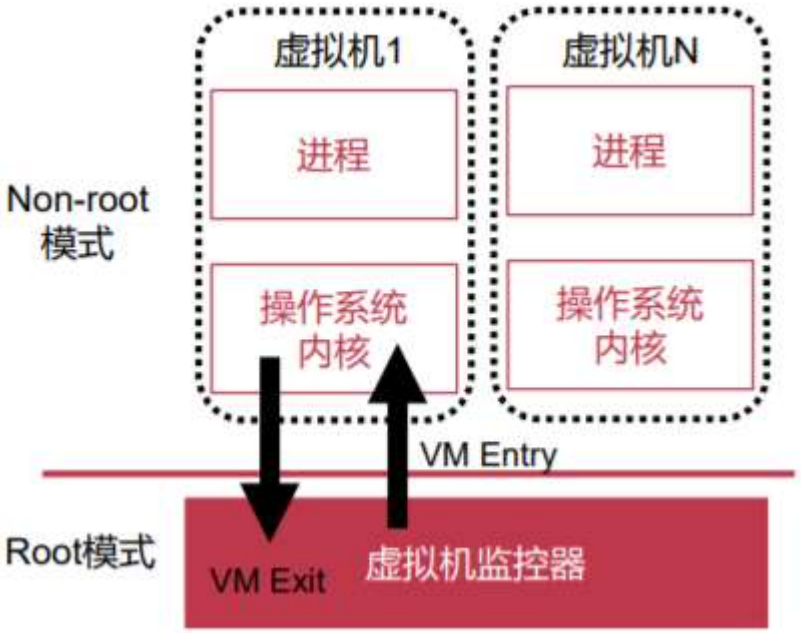

1 PC
模拟器
4 未命中
2 查询
查询
翻译
3 命中
虚拟机
内存
基本块1
基本块2
基本块3
基本块4
基本块5
基本块6
基本块N
写入
缓存
代码补丁1
代码补丁2
代码补丁3
5 写入
执行
虚拟机
状态
…
6 读写
内存

硬件辅助的CPU虚拟化：Intel VT-x

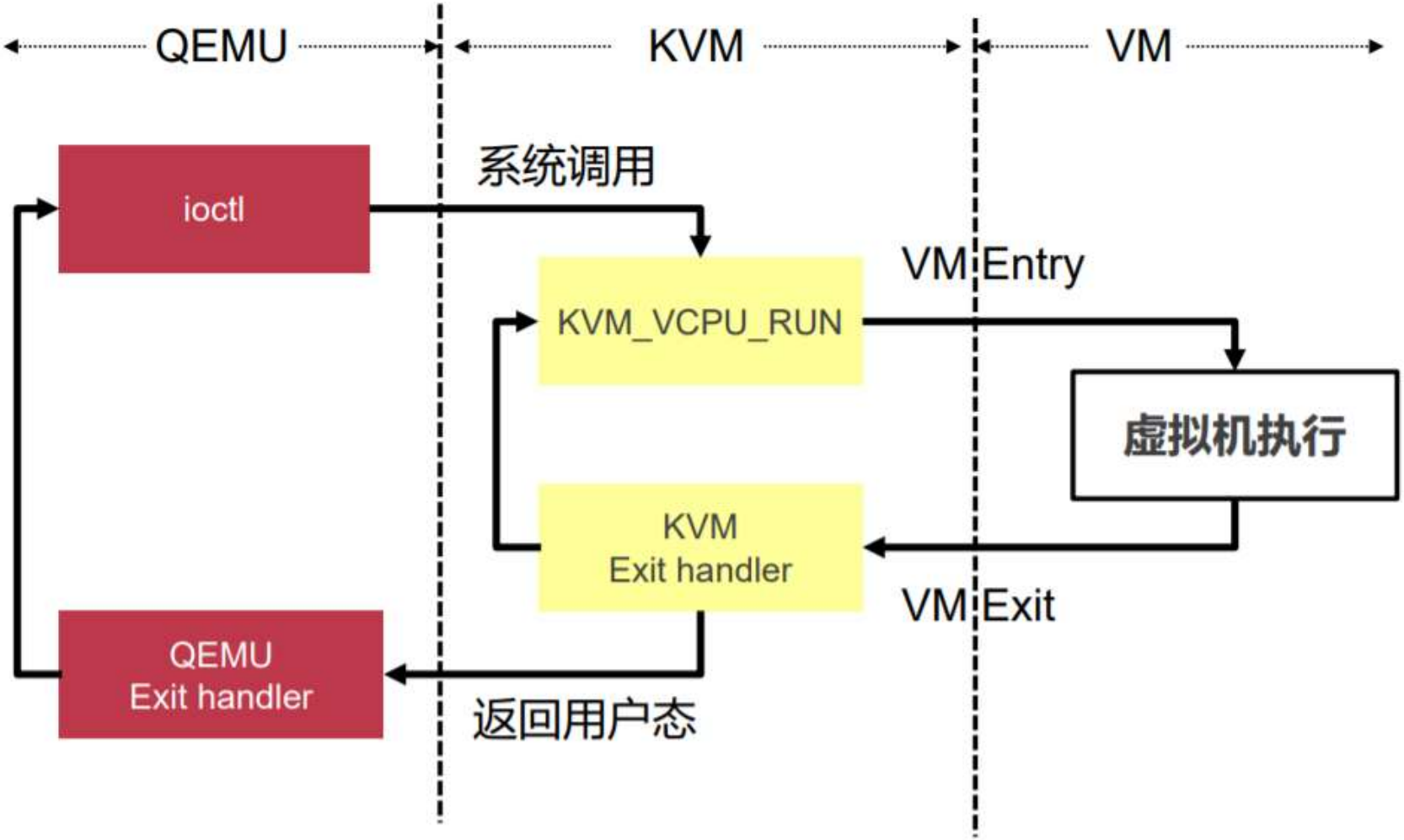
- Root mode: Host OS和Hypervisor
- Non-root mode: Guest OS

事件循环

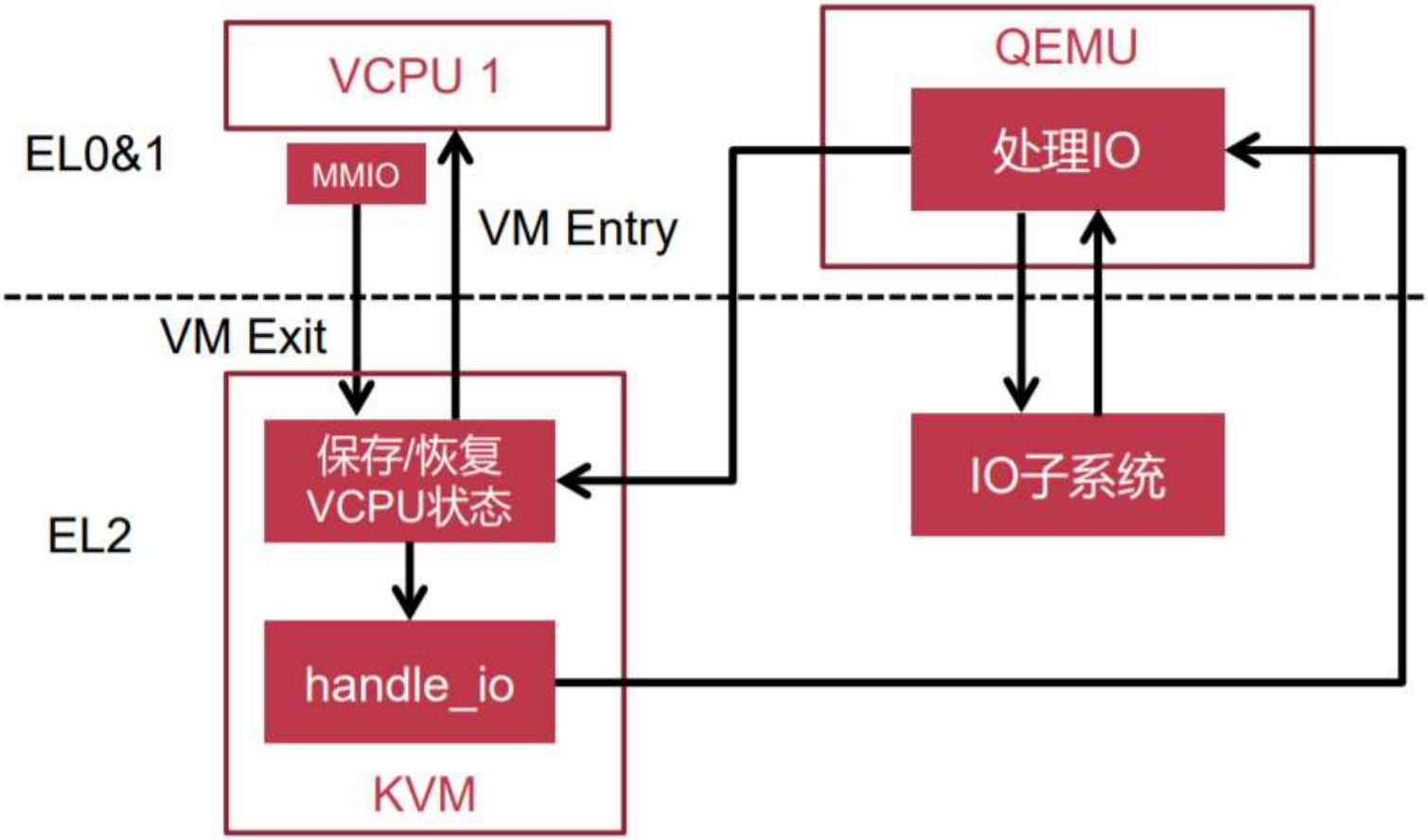
- VM Entry：由Root模式进入Non-Root模式
  - 由Hypervisor进入VM
  - 在第一次执行VM：VM Launch
  - 在VM Exit之后：VM Resume
- VM Exit：由Non-Root模式进入Root模式
  - 外部中断
  - 敏感指令（不再有敏感非特权指令问题）
  - Hypercall



QEMU-KVM的事件循环



I/O指令VM Exit的处理流程



QEMU-KVM的vCPU实现

- 作为pthread的vCPU线程
  - vCPU受到Host调度器（例如CFS）的调度
  - 作为被调度的线程，vCPU线程会与其他Host OS线程/进程竞争

```
/* Leave signal handling to the iothread.  */
sigfillset(&set);
pthread_sigmask(SIG_SETMASK, &set, &oldset);
err = pthread_create(&thread->thread, &attr, start_routine, arg);
if (err)
    error_exit(err, __func__);
```

```
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

VMCS（Virtual Machine Control Structure）

- 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

VMCS示例

```
FIELD64(IO_BITMAP_A, io_bitmap_a),
FIELD64(IO_BITMAP_B, io_bitmap_b),
```

- 指定哪些IO操作会引发VMExit

```
FIELD(HOST_CR3, host_cr3),
```

- Host OS的页表地址

```
FIELD(HOST_RIP, host_rip),
```

- 返回到Hypervisor之后所需要执行的指令地址

```
FIELD(EXIT_QUALIFICATION, exit_qualification)
```

```
FIELD(VM_EXIT_REASON, vm_exit_reason),
FIELD(VM_EXIT_INTR_INFO, vm_exit_intr_info),
FIELD(VM_EXIT_INTR_ERROR_CODE, vm_exit_intr_error_code),
```

- VMExit的原因

```
FIELD64(EPT_POINTER, ept_pointer),
```

- EPT的物理地址

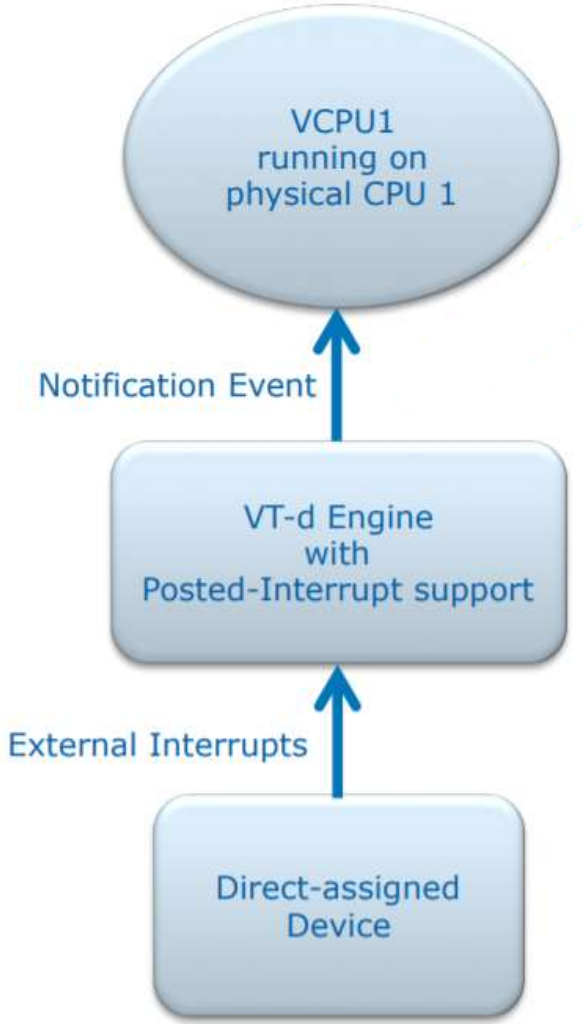
- 超过100个字段

VMExit性能开销

- VMCS保存/恢复开销
- Hypervisor模拟开销
- Cacheline污染开销
- …

- 当一个正在Non-root模式下的CPU收到一个（应该呈递给Guest OS）外部中断时：
  - VMExit
  - Host OS IDT
  - Host OS
  - QEMU I/O thread, `epoll`
  - KVM
  - VM Resume
  - Guest OS IDT
  - …

Posted Interrupt



- 当一个正在Non-root模式下的CPU收到一个（应该呈递给Guest OS）外部中断时：
  - ~~VMExit~~
  - ~~Host OS IDT~~
  - ~~Host OS~~
  - ~~QEMU I/O thread, `epoll`~~
  - ~~KVM~~
  - ~~VM Resume~~
  - Guest OS IDT
  - …

5. KVM 中的内存虚拟化

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

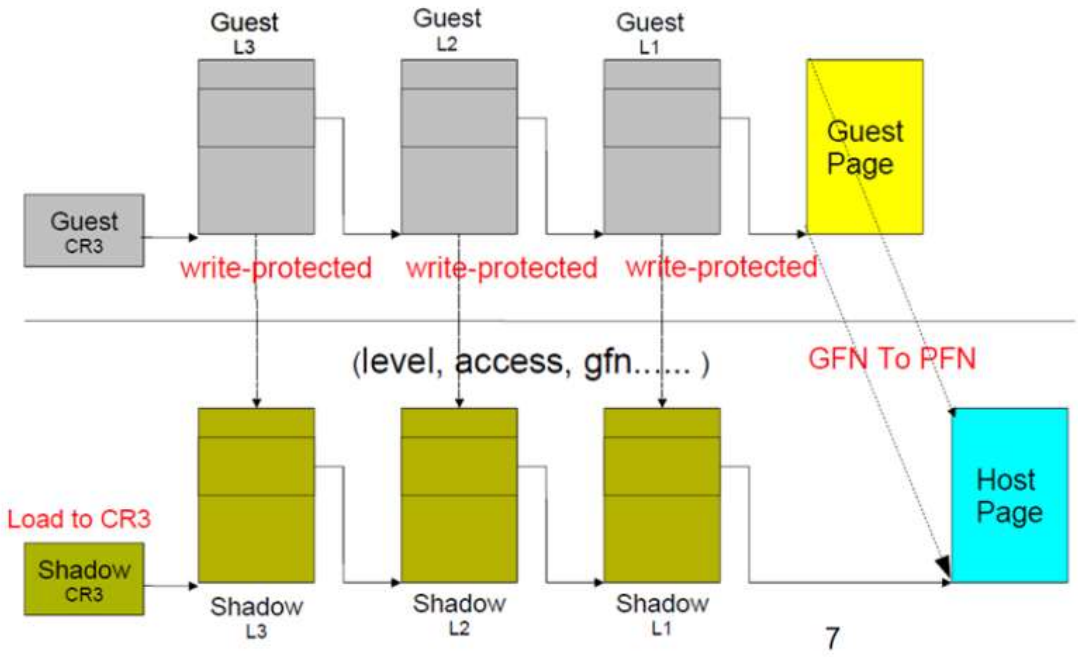
四个地址

- 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

- 硬件可能直接访问GVA->HPA（SPT），或者先访问Guest OS页表，再访问GPA->HPA（EPT）
- 硬件（MMU，TLB）会访问Host OS页表来找到HVA所对应的HPA

影子页表（Shadow Page Table）

- Guest OS“认为”它建立了从GVA->GPA的映射，并且硬件会根据改页表寻址；
- 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。

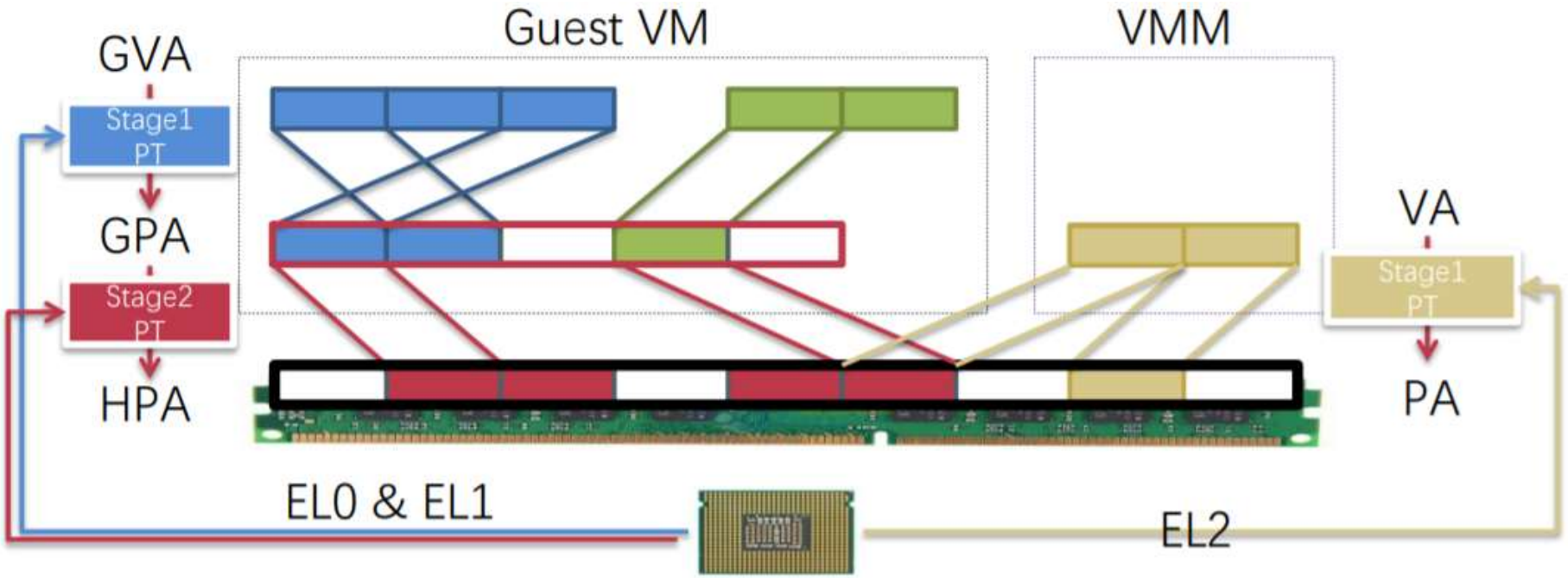


SPT的特点

- 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

EPT（Extended Page Table）

- 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

相关参考

- Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. 2003. Xen and the art of virtualization. In Proceedings of the nineteenth ACM symposium on Operating systems principles (SOSP '03). Association for Computing Machinery, New York, NY, USA, 164–177. DOI:https://doi.org/10.1145/945445.945462
- https://wiki.xen.org/wiki/Xen_Project_Software_Overview
- https://wiki.xen.org/wiki/X86_Paravirtualised_Memory_Management
- https://www.ibm.com/developerworks/cn/linux/l-using-kvm/index.html
- https://ipads.se.sjtu.edu.cn/courses/os/slides/OS-18-Virtualization2.pdf
- https://ipads.se.sjtu.edu.cn/courses/os/slides/OS-17-Virtualization.pdf
- Yabusame: Postcopy Live Migration for Qemu/KVM

谢谢！

附录：QEMU的编译与安装

1. 从Linux发行版公共仓库中安装QEMU二进制文件：

```
$ sudo apt-get install qemu-kvm
```

- Ubuntu的Linux内核默认有KVM模块

2. 源代码编译：

QEMU：

```
$ git clone https://github.com/qemu/qemu.git
$ ../configure --enable-kvm …
$ make
```

附录：QEMU源代码修改

- QEMU是一个用户态程序
  - 可以使用glibc，Linux系统调用和标准C语言库

例如：

`./kvm-all.c`包含了大部分QEMU与KVM交互的代码

附录：QEMU源代码修改

```
1961        do {
1962            MemTxAttrs attrs;
1963
1964            if (cpu->kvm_vcpu_dirty) {
1965                kvm_arch_put_registers(cpu, KVM_PUT_RUNTIME_STATE);
1966                cpu->kvm_vcpu_dirty = false;
1967            }
1968
1969            kvm_arch_pre_run(cpu, run);
1970            if (cpu->exit_request) {
1971                DPRINTF("interrupt exit requested\n");
1972                /*
1973                 * KVM requires us to reenter the kernel after IO exits to complete
1974                 * instruction emulation. This self-signal will ensure that we
1975                 * leave ASAP again.
1976                 */
1977                qemu_cpu_kick_self();
1978            }
1979
1980            run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);
1981
1982            attrs = kvm_arch_post_run(cpu, run);
1983
1984            if (run_ret < 0) {
1985                if (run_ret == -EINTR || run_ret == -EAGAIN) {
1986                    DPRINTF("io window exit\n");
1987                    ret = EXCP_INTERRUPT;
V-LINE    kvm-dsm    kvm-all.c
```

附录：QEMU源代码修改

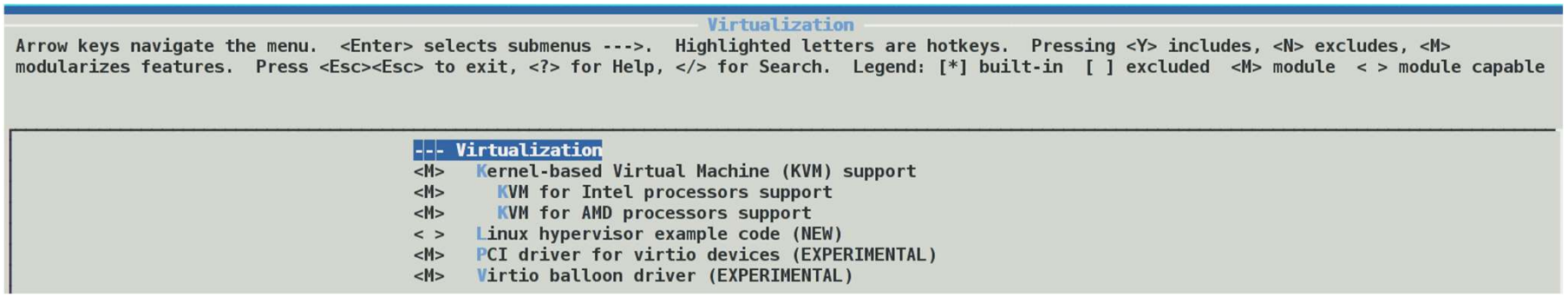
```
1960
1961    do {
1962        MemTxAttrs attrs;
1963
1964        if (cpu->kvm_vcpu_dirty) {
1965            kvm_arch_put_registers(cpu, KVM_PUT_RUNTIME_STATE);
1966            cpu->kvm_vcpu_dirty = false;
1967        }
1968
1969        kvm_arch_pre_run(cpu, run);
1970        if (cpu->exit_request) {
1971            DPRINTF("interrupt exit requested\n");
1972            /*
1973             * KVM requires us to reenter the kernel after IO exits to complete
1974             * instruction emulation. This self-signal will ensure that we
1975             * leave ASAP again.
1976             */
1977            qemu_cpu_kick_self();
1978        }
1979
1980        printf("Enter KVM (ring3 to ring0)");
1981        run_ret = kvm_vcpu_ioctl(cpu, KVM_RUN, 0);
1982        printf("Exit from KVM (ring0 to ring3)");
1983
1984        attrs = kvm_arch_post_run(cpu, run);
1985
1986        if (run_ret < 0) {
1987            if (run_ret == -EINTR || run_ret == -EAGAIN) {
1988                DPRINTF("io window exit\n");
1989                ret = EXCP_INTERRUPT;
1990                break;
1991            }
V-LINE   kvm-dsm   kvm-all.c
```

附录：Linux内核（内含KVM模块）的编译与安装

Linux:

```
$ git clone https://github.com/torvalds/linux.git
```

```
$ make menuconfig
```



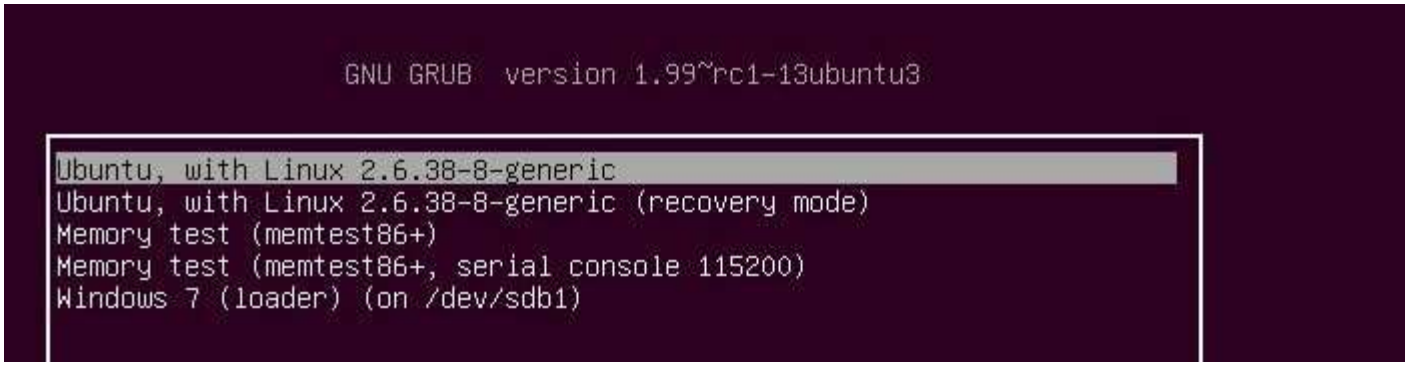

```
$ make
```

```
$ sudo make modules_install
```

```
$ sudo make install
```

附录：Linux内核（内含KVM模块）的编译与安装

- GNU GRUB是一个bootloader，即计算机启动之后执行的第一个软件。它负责选择一个内核，并跳转执行逻辑到该内核



- 编译完成并Install之后，GRUB默认优先选择最新编译的Linux内核
  - 如果该内核有错误，用户可以使用GRUB选择较早的内核启动

附录：Linux源代码修改

- arch/x86/kvm/x86.c:

```
6980 static int vcpu_run(struct kvm_vcpu *vcpu)
6981 {
6982   int r;
6983   struct kvm *kvm = vcpu->kvm;
6984
6985   vcpu->srcu_idx = srcu_read_lock(&kvm->srcu);
6986
6987   for (;;) {
6988     if (kvm_vcpu_running(vcpu)) {
6989       r = vcpu_enter_guest(vcpu);
6990     } else {
6991       r = vcpu_block(kvm, vcpu);
6992     }
6993
6994     if (r <= 0)
6995       break;
6996
6997     clear_bit(KVM_REQ_PENDING_TIMER, &vcpu->requests);
6998     if (kvm_cpu_has_pending_timer(vcpu))
6999       kvm_inject_pending_timer_irqs(vcpu);
7000
7001     if (dm_request_for_irq_injection(vcpu) &&
7002       kvm_vcpu_ready_for_interrupt_injection(vcpu)) {
7003       r = 0;
7004       vcpu->run->exit_reason = KVM_EXIT_IRQ_WINDOW_OPEN;
V-LINE   master   x86.c
```

附录：Linux源代码修改

- arch/x86/kvm/x86.c:

```
6980 static int vcpu_run(struct kvm_vcpu *vcpu)
6981 {
6982   int r;
6983   struct kvm *kvm = vcpu->kvm;
6984
6985   vcpu->srcu_idx = srcu_read_lock(&kvm->srcu);
6986
6987   for (;;) {
6988     if (kvm_vcpu_running(vcpu)) {
6989       printk(KERN_INFO "vCPU %d VM Resume", vcpu->vcpu_id);
6990       r = vcpu_enter_guest(vcpu);
6991       printk(KERN_INFO "vCPU %d VM Exit", vcpu->vcpu_id);
6992     } else {
6993       r = vcpu_block(kvm, vcpu);
6994     }
6995
6996     if (r <= 0)
6997       break;
6998
V-LINE   master   x86.c
```

附录：Linux源代码修改

- 温馨提示：在KVM模块中的错误可能会影响其他Linux子系统，例如文件系统
  - 必须重启
  - 如果在物理机上做实验，请确保重要数据备份
  - 宏内核（Linux/Windows）v. 微内核（HarmonyOS）
- Segmentation Fault

```
int *ptr = NULL; *ptr = 1;
```

```
Unable to handle kernel paging request at virtual
address 0
```

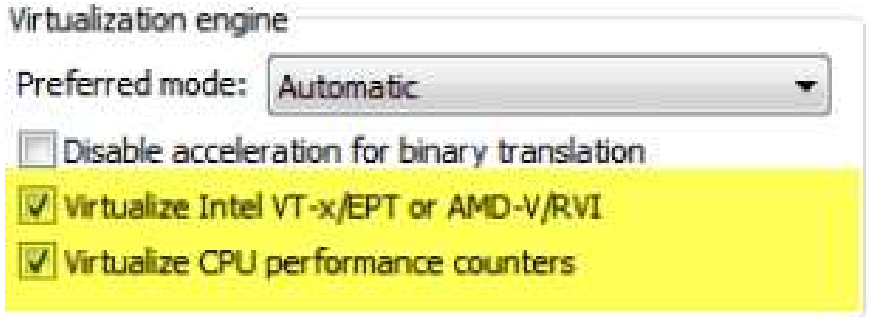
- Deadlock

```
spin_lock(&lck); spin_lock(&lck);
```

```
watchdog: BUG: soft lockup - CPU #1 stuck for 22s!
```

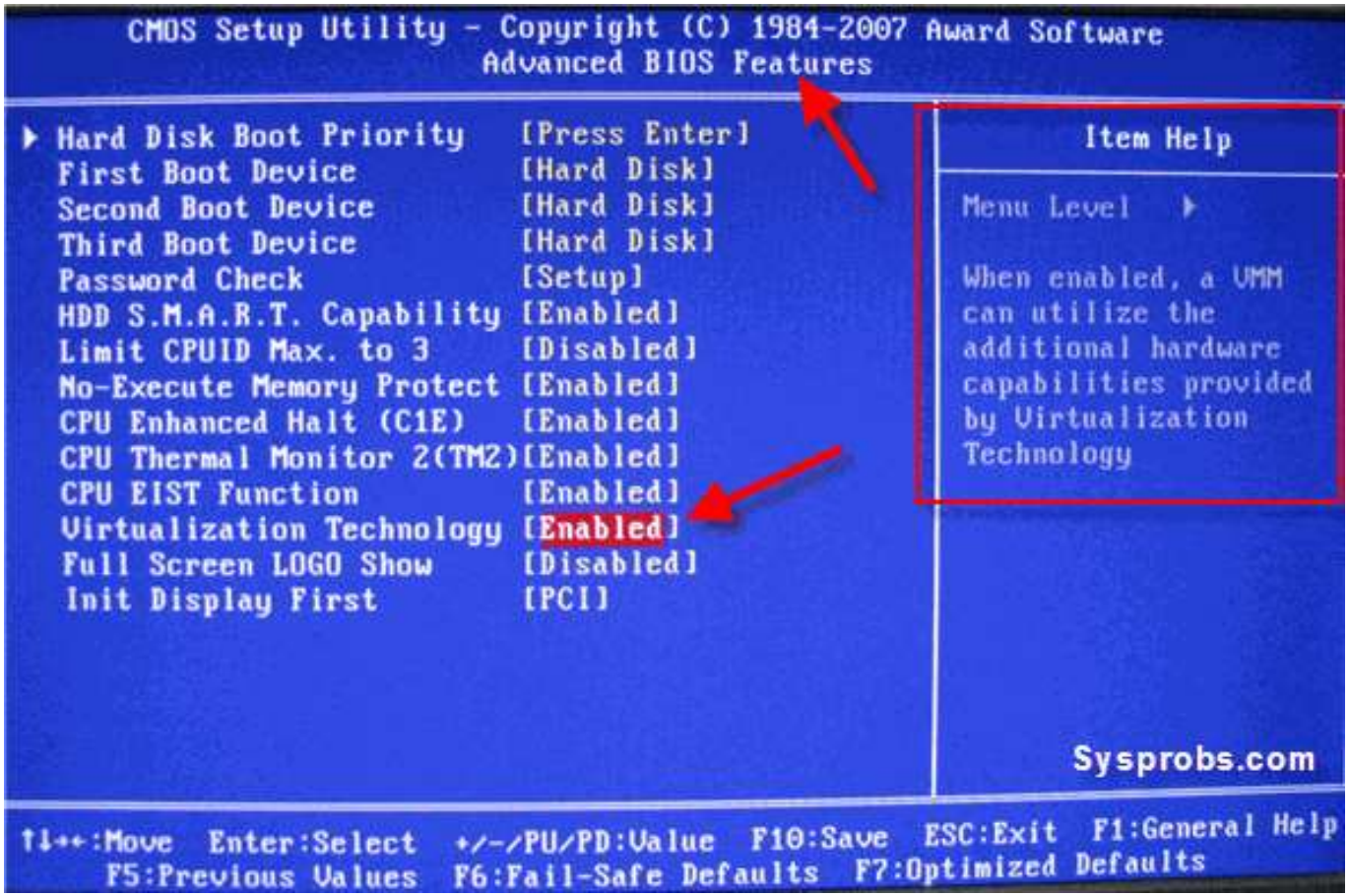
附录：VMWare Workstation

- VMWare Workstation本身也提供了与QEMU-KVM相同原理的虚拟化支持
  - Binary-translation; Hardware-assisted virtualization
- 如果操作系统是Windows，那么需要模拟一个Linux的ABI才能运行QEMU-KVM
  - 在虚拟机中运行虚拟机被称为嵌套虚拟化
  - 嵌套虚拟化性能很差，而且有很多安全漏洞
- 在VMWare中开启嵌套虚拟化选项：



附录：在BIOS中打开VT-x支持

- 默认打开



附录：运行QEMU-KVM

- 在QEMU-KVM中运行Ubuntu步骤概览：

1. 创建磁盘镜像
2. 下载Guest OS镜像，并在磁盘镜像上安装
3. 运行QEMU

- `qemu-system-x86_64 …` 是使用二进制翻译
- `qemu-system-x86_64 --enable-kvm …` 是使用硬件辅助虚拟化