高级云操作系统

2022年11月

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

CPU虚拟化简介

戚正伟

2022年11月

上海交通大學
SHANGHAI JIAO TONG UNIVERSITY

目录 Contents

目录 Contents

CPU虚拟化的主要任务

- CPU的主要功能
  - **指令执行**，完成各种各样的计算任务
  - **事件响应**
    - 同步事件：指令执行所触发的异常，如除零错误、段错误等；
    - 异步事件：外部环境产生，如外部设备产生的中断等；具有不可预知性，随时都可能发生。

- 虚拟CPU的主要任务
  - **执行虚拟机程序指令**
  - **响应虚拟机内外部事件**

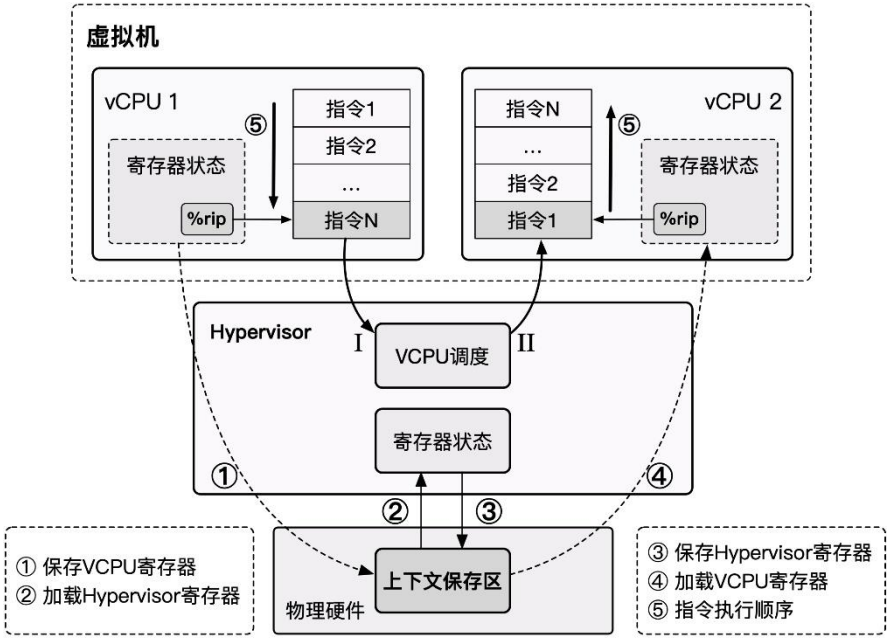
CPU虚拟化面临的挑战——敏感非特权指令

- **可虚拟化架构**
  - Popek 和 Goldberg证明，基于**陷入-模拟机制**的虚拟化架构只能在所有敏感指令都是特权指令的架构中被建立
  - **敏感指令：**操作敏感物理资源的指令，如I/O 指令、 页表基地址切换指令等
  - **特权指令：**必须运行在最高特权级的指令，在非最高特权级中执行这些指令将会触发特权级切换。

Group	Instructions
Access to interrupt flag	pushf, popf, iret
Visibility into segment descriptors	lar, verr, verw, lsl
Segment manipulation instructions	pop <seg>, push <seg>, mov <seg>
Read-only access to privileged state	sgdt, sldt, sidt, smsw
Interrupt and gate instructions	fcall, longjump, retfar, str, int <n>

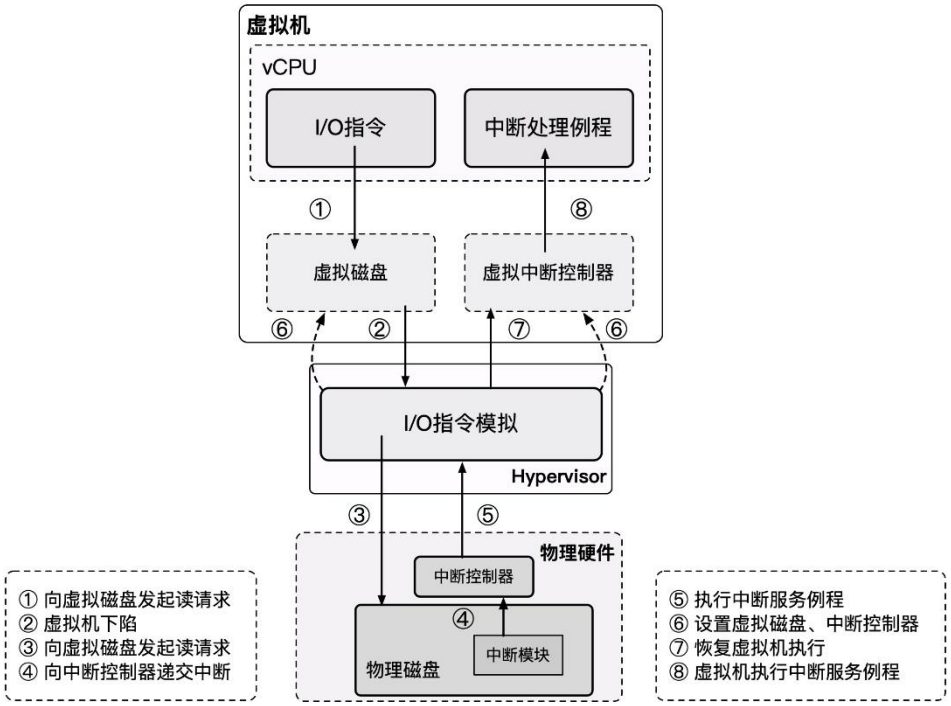
CPU虚拟化面临的挑战——上下文切换

- **虚拟CPU上下文**
  - 类似于进程上下文的概念，当虚拟机发生退出时，需要保存虚拟CPU中各寄存器的状态
  - 发生虚拟CPU调度时，需要保存当前虚拟CPU的上下文并加载待调度虚拟CPU上下文



CPU虚拟化面临的挑战——中断事件处理

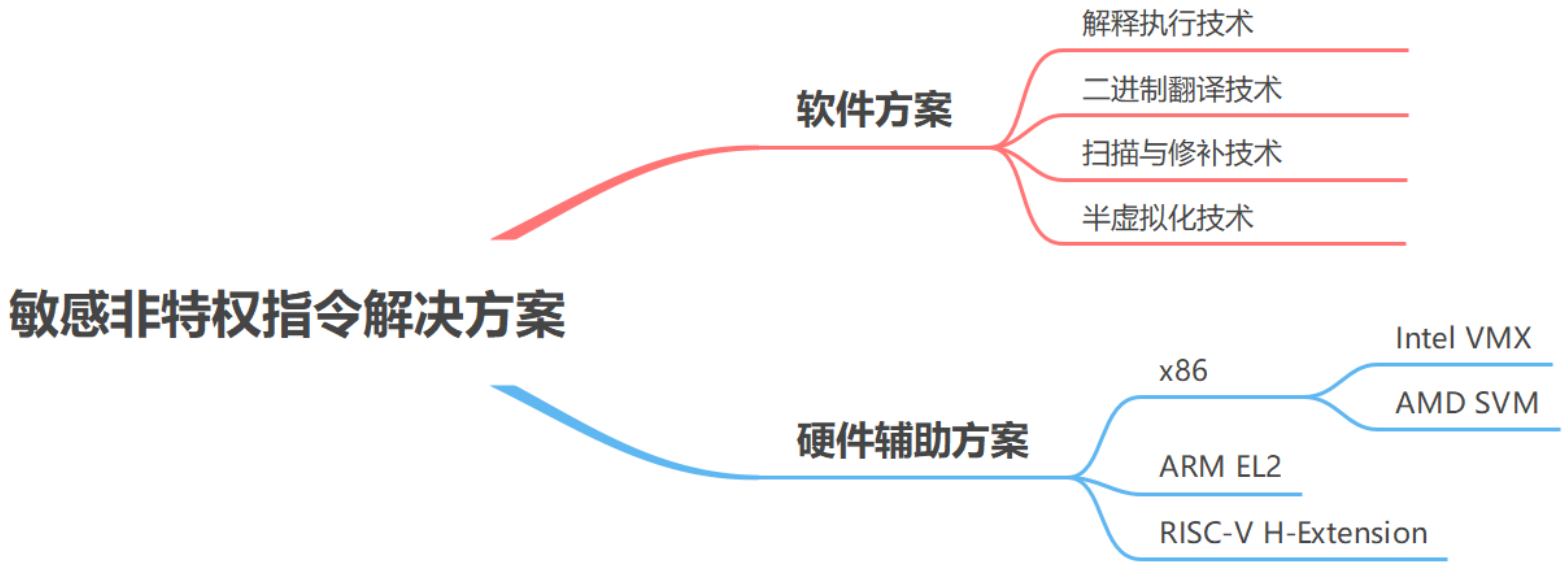
- **中断控制器的模拟**
  - 为每一个虚拟机维护一个虚拟中断控制器
- **物理中断的截获与处理**
- **中断递送流程模拟**
  - 中断产生
  - 中断路由
  - 中断接受
  - 中断确认
  - 中断交付



虚拟磁盘中断处理流程

敏感非特权指令的解决方案

- 软件方案
  - 解释执行、二进制翻译、扫描与修补、半虚拟化
- 硬件辅助方案
  - Intel VT-x、AMD SVM、ARM EL2、RISC-V H-Extension



软件解决方案

- **解释执行**
  - 解释器将程序二进制解码后调用指令相应的模拟函数
  - 采用软件模拟的方式逐条模拟虚拟机指令的执行
- **二进制翻译**
  - 区别于解释执行技术，二进制翻译技术以基本块为单位，将虚拟机指令批量翻译后保存在代码缓存中
  - 基本块中的敏感指令会被替换为一系列其他指令

待翻译的 CPUID 指令

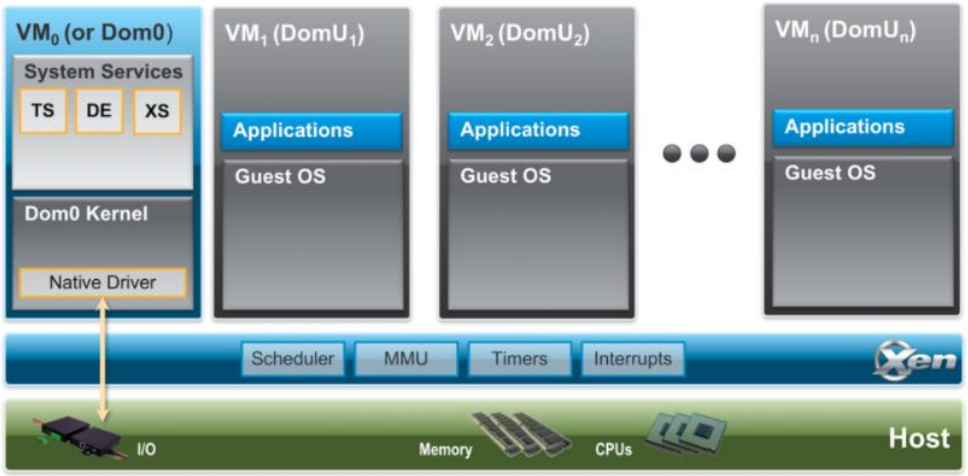
```
1   mov %eax   $0x1
2   cpuid
```

翻译后的目标指令序列

```
1   mov %edi   $0x1
2   call helper_cpuid
```

软件解决方案

- **扫描与修补**
  - 在执行每段代码前对其进行扫描，找到其中的敏感指令，将其替换为特权指令
  - CPU 执行翻译后的代码时，遇到替换后的特权指令便会陷入 Hypervisor 中进行模拟，执行对应的补丁代码
- **半虚拟化技术**
  - 虚拟机需要执行执行敏感操作时主动通过超调用(Hypercall)陷入 Hypervisor 中，避免了扫描程序二进制代码引入的开销

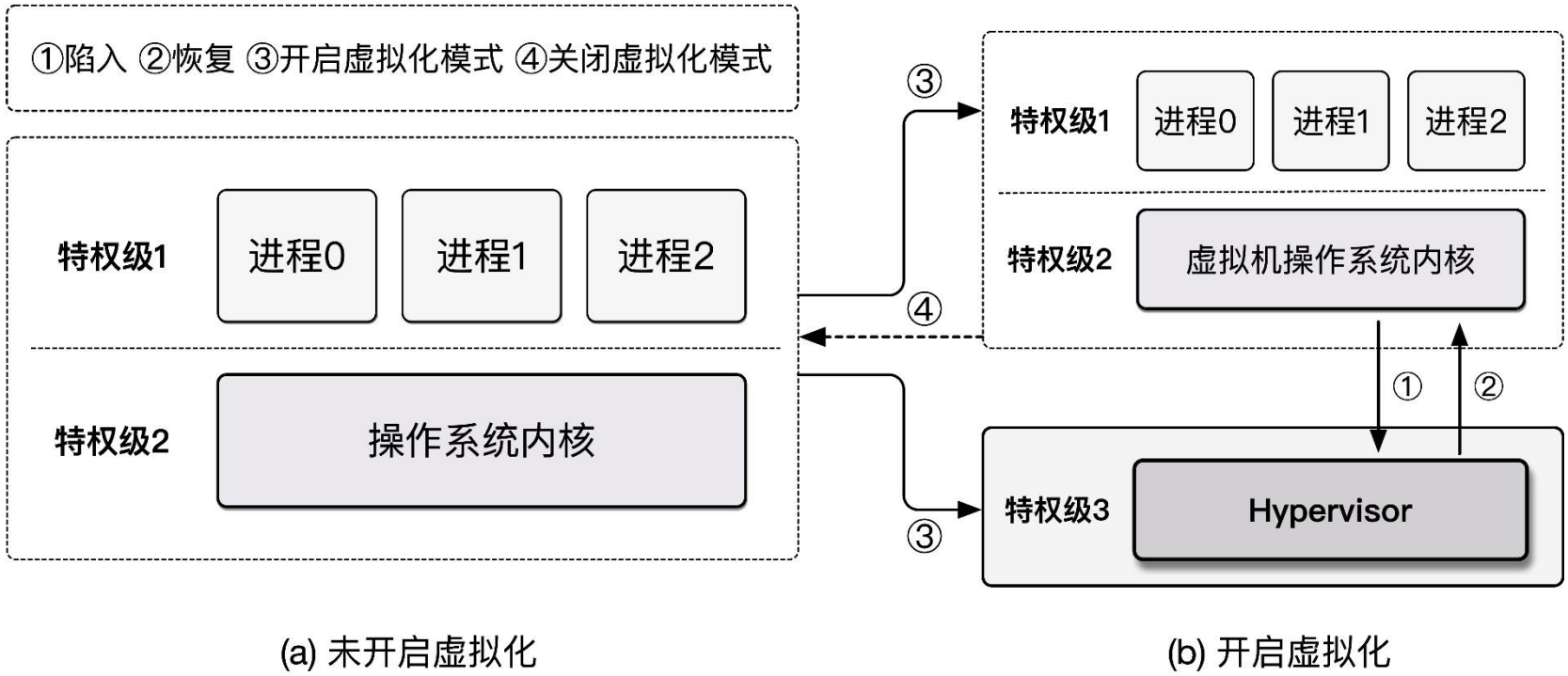


软件解决方案对比

软件解决方案	优点	缺点
解释执行技术	允许虚拟机ISA不同于物理机ISA	不加区分地模拟每一条指令，效率低下
二进制翻译技术	批量翻译与缓存虚拟机指令，代码局部性较高	指令数目通常会增加，占用内存较多，相对寻址指令与跳转指令需要进一步处理
扫描与修补技术	非敏感指令直接运行，无须模拟	要求虚拟机ISA与物理机ISA相同，需要特权级切换，代码局部性较差
半虚拟化技术	无须扫描程序二进制代码，虚拟机主动陷出，性能较好	需要修改客户机操作系统，打破虚拟机与Hypervisor间的界限

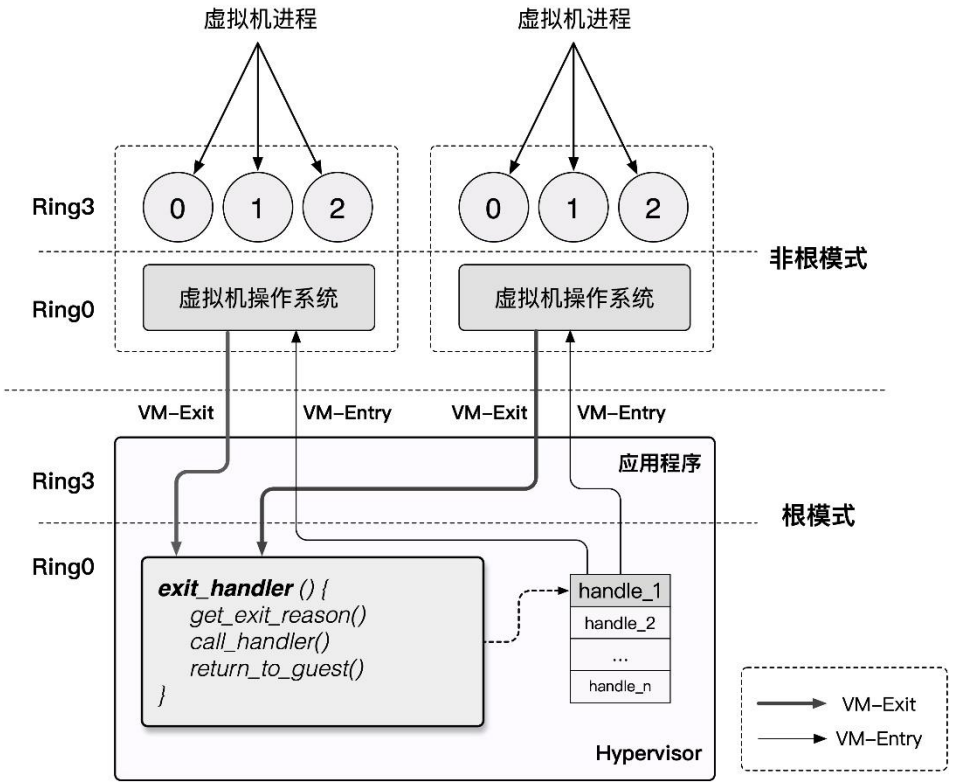
硬件辅助解决方案

- 所有敏感指令 → 特权指令
  - 可能会引起兼容性问题，导致现有应用不可用
- 引入虚拟化模式



Intel VT-x

- VMX操作模式(通过vmxon指令开启)
  - Guest App → 非根模式 ring 3
  - Guest OS → 非根模式 ring0
  - Host App → 根模式 ring3
  - Host OS/Hypervisor → 根模式ring0
- VM-Entry
  - 根模式 → 非根模式
  - Hypervisor → 虚拟机
  - 指令触发(vmlaunch/vmresume)
- VM-Exit
  - 非根模式 → 根模式
  - 虚拟机 → Hypervisor
  - 虚拟机敏感指令/外部中断触发

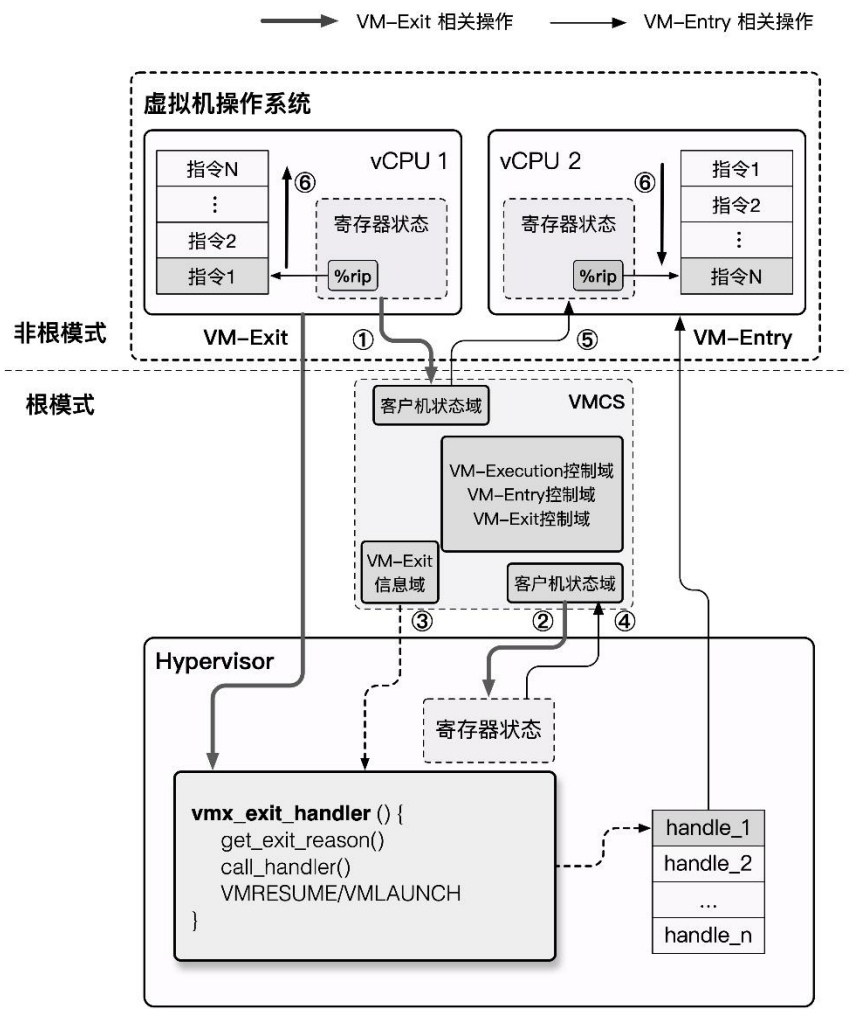


Intel VMCS

- VMCS(Virtual Machine Control Structure, 虚拟机控制结构)包括六部分:
  - **客户机状态域：**保存虚拟机寄存器状态的区域
  - **宿主机状态域：**保存Hypervisor寄存器状态的区域
  - VM-Execution**控制域：**控制客户机在non-root模式下运行时的行为
  - VM-Entry**控制域：**控制VM-Entry过程中的某些行为
  - VM-Exit**控制域：**控制VM-Exit过程中的某些行为
  - VM-Exit**信息域：**保存VM-Exit的基本原因及其他详细信息
- VMCS相关指令
  - vmptrld<**VMCS地址**>：加载VMCS
  - vmclear<**VMCS地址**>：清除VMCS，确保VMCS数据写入内存
  - vmread<**索引**>：读取指定索引处的VMCS字段
  - vmwrite<**索引**><**数据**>：将数据写入指定索引处的VMCS字段

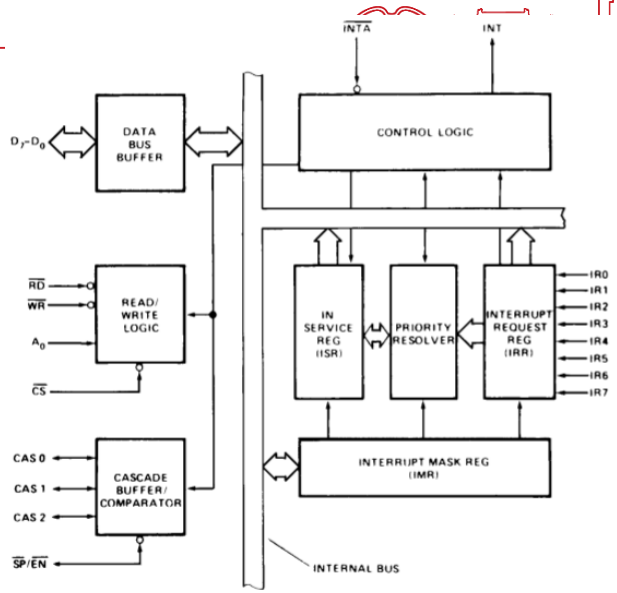
Intel VMCS

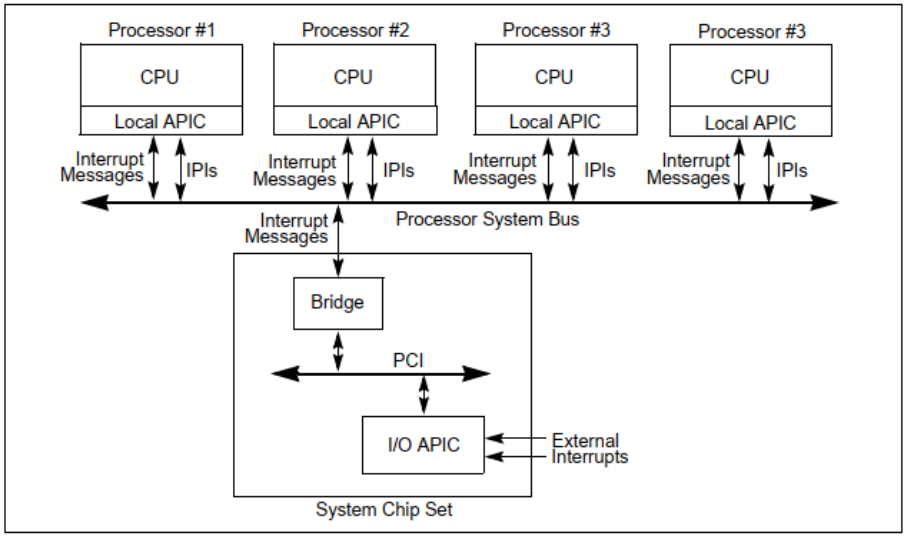
- VM-Entry流程:
  - 检查VMCS中各字段值是否合法
  - 将客户机状态域各字段值加载至相应寄存器中
  - 加载指定MSR
  - 将VMCS状态设为launched
- VM-Exit流程
  - 将虚拟机退出的原因及详细信息写入VM-Exit信息域
  - 将当前CPU寄存器状态保存至客户机状态域
  - 将宿主机状态域各字段值加载至相应寄存器中
  - 加载指定MSR



PIC & APIC

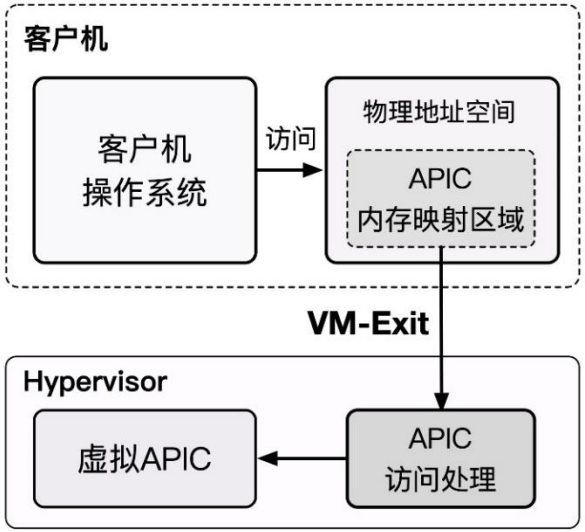
- PIC(Intel 8259A)
  - 单处理器时代流行，包含8个中断引脚
  - **中断屏蔽寄存器IMR：**置1表示相应引脚中断被屏蔽
  - **中断请求寄存器IRR：**置1表示收到相应引脚的中断信号
  - **中断服务寄存器ISR：**置1表示相应引脚的中断信号正在被CPU处理
- APIC
  - 应对多处理器架构，**包括IOAPIC和LAPIC两部分**
  - 每个CPU拥有一个LAPIC，整个机器拥有一个或多个IOAPIC，设备的中断信号先经由IOAPIC汇总，再分发给一个或多个CPU的LAPIC
  - LAPIC中断源
    - **处理器本地中断，如时钟中断**
    - **通过IOAPIC接收的外部中断**
    - IPI(Inter-processor Interrupt，**处理器间中断**)



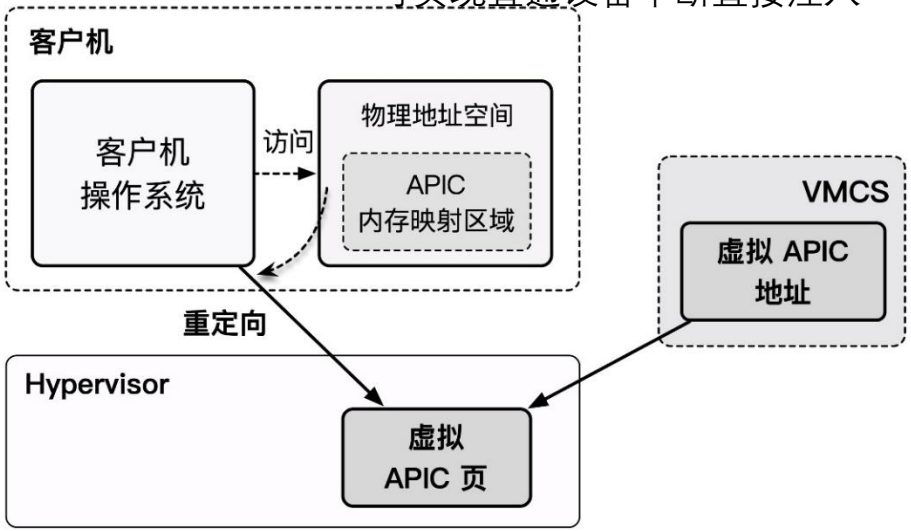


Intel APICv

- 传统中断虚拟化
  - Hypervisor为虚拟机维护虚拟机APIC
  - 虚拟机**通过MMIO访问**虚拟APIC内部寄存器
  - **通过配置影子页表或EPT页表权限截获虚拟机访问**
  - Hypervisor读取虚拟APIC相应寄存器再恢复虚拟机运行
  - 缺点：通过虚拟机陷出来截获虚拟机APIC访问
- APICv
  - **引入虚拟APIC页**
  - **将虚拟机对APIC的访问重定向至虚拟机APIC页**
  - 不会影响真实的APIC
  - APICv还**引入了Posted Interrupt机制**，允许Hypervisor在不触发虚拟机陷出的情况下注入虚拟中断，与Intel VT-d中断重映射功能相结合可实现直通设备中断直接注入



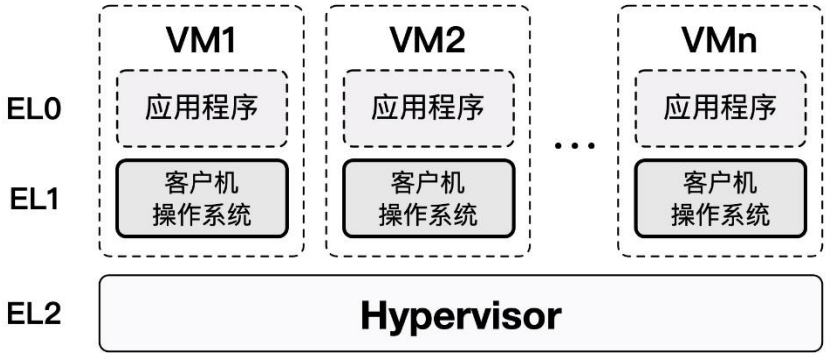
(a) 未使用APICv

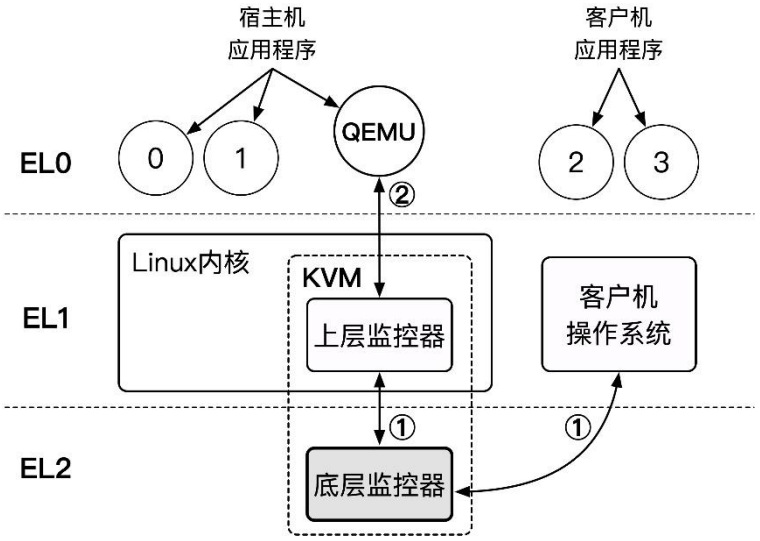


(b) 使用APICv

ARM EL2

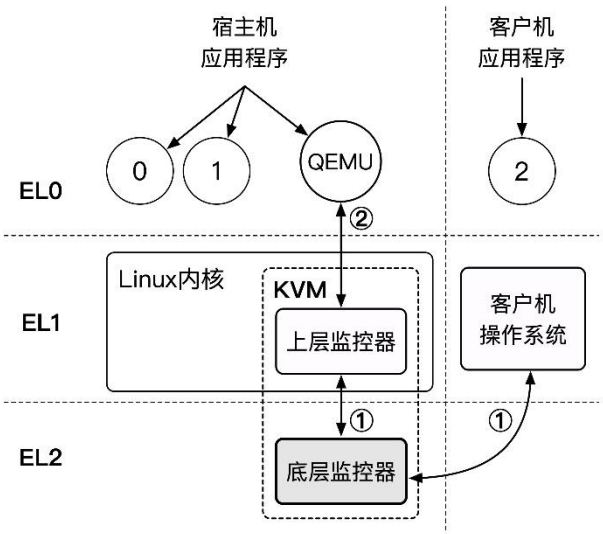
- ARM v8引入EL2
  - Guest App → EL0
  - Guest OS → EL1
  - Hypervisor → EL2
- 模式切换
  - 虚拟机 → Hypervisor
    - EL1 → EL2
    - 敏感指令触发(可通过HCR_EL2寄存器细粒度控制)
  - Hypervisor → 虚拟机
    - EL2 → EL1
    - eret指令触发
- 上下文切换
  - 不同于Intel根模式与非根模式共享一套寄存器，**EL1与EL2各自有一套系统寄存器**
  - **虚拟CPU调度时，需要将原虚拟CPU系统寄存器保存至内存**并从内存中加载目标虚拟CPU寄存器



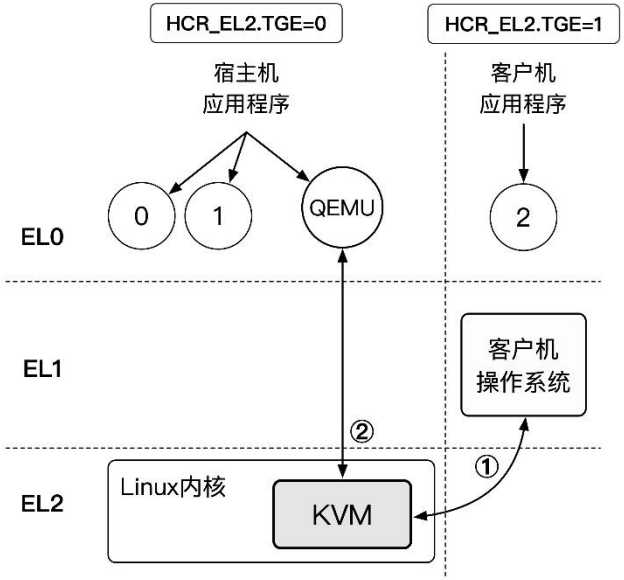


ARM VHE

- Type-II Hypervisor on ARM
  - Host OS → EL1 Hypervisor → EL2
  - Hypervisor分为两层，运行时造成频繁的上下文切换，严重影响性能
- VHE(Virtual Host Extension)
  - 允许Host OS运行在EL2，解决Hypervisor分层问题
  - HCR_EL2.E2H控制是否开启VHE
  - HCR_EL2.TGE用于在使能VHE时区分Guest App和Host App



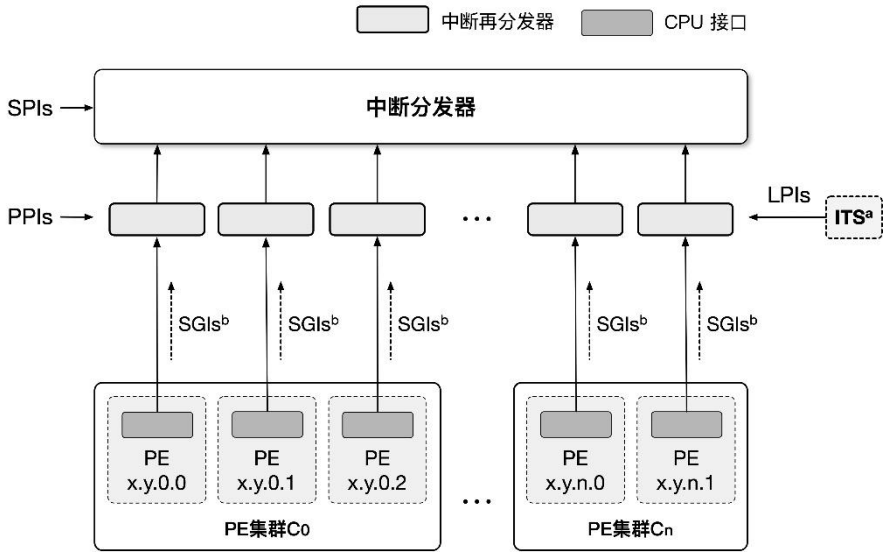
(a) 未引入VHE



(b) 引入VHE

ARM GIC

- GIC中断类型
  - **软件生成中断SGI：**由CPU直接写对应的寄存器触发，用于核间通信
  - **私有设备中断PPI：**为CPU私有的中断源，类似于APIC中的本地中断源
  - **共享设备终端SPI：**外部设备中断，类似于IOAPIC接收的外部中断
  - **特殊设备中断LPI：**GICv3引入的基于消息的中断，兼容PCIe总线MSI/MSI-X机制
- GIC组件
  - **中断分发器Distributor：**维护SPI类型中断的相关信息，负责SPI与SGI类型中断的路由
  - **中断再分发器Redistributor：**维护PPI/SGI中断相关信息
  - **中断翻译服务ITS：**GICv3引入，输出LPI类型中断。GICv4引入Virtual LPI直接注入机制，类似于APICv提供的Posted-Interrupt机制，无需Hypervisor参与便能直接注入虚拟中断。
  - **CPU接口：**负责中断优先级检测、优先级仲裁（选择优先级最高的中断处理）和中断应答

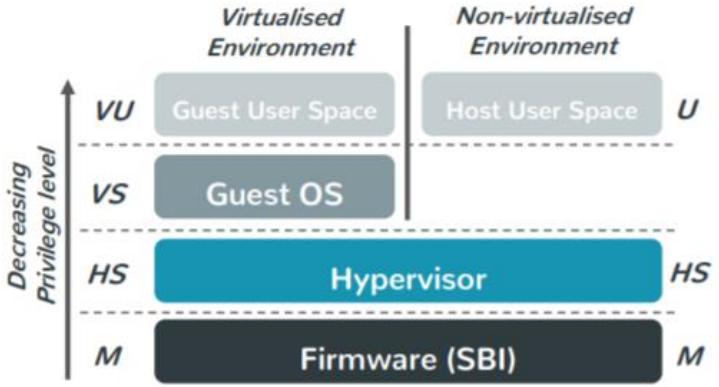


ARM GIC中断虚拟化支持

- **虚拟CPU接口直接访问**
  - 将CPU接口置于CPU内部，通过寄存器访问
  - 为虚拟CPU接口提供专用寄存器(ICV_*)，区别于物理CPU接口
  - HCR_EL2.IMO和HCR_EL2.FMO设置为1时，运行在EL1中的Guest OS对CPU接口系统寄存器（ICC_*）的访问将被重定向到相应的ICV_*寄存器而不会触发虚拟机陷出
- **虚拟中断注入**
  - GICv3配置使得所有的物理中断路由到EL2
  - Hypervisor检查中断目标是否为vCPU
  - 根据ICH_LR<n>_EL2寄存器中保存的虚拟中断信息向虚拟机中注入中断
- **虚拟LPI中断直接注入**
  - 类似于x86的Posted Interrupt机制
  - GICv3引入组件中断翻译服务ITS，该组件通过查询若干内存表将物理中断转化为虚拟LPI中断注入相应的虚拟机

RISC-V H-Extension

- RISC-V H-Extension引入Virtualization Mode(V)控制是否开启虚拟化模式
  - Guest App → VU mode
  - Guest OS → VS mode
  - Hypervisor/Host OS → HS mode
- 模式切换(类似于ARM)
  - 虚拟机 → Hypervisor
    - VS mode → HS mode(先进入M mode，再由M mode转发给HS mode)
    - 敏感指令触发
  - Hypervisor → 虚拟机
    - HS mode → VS mode
    - sret指令触发
- 上下文切换(类似于ARM)
  - 为VS-mode提供VS CSR
  - **虚拟CPU调度时，同样需要从内存中保存和加载相应的寄存器**

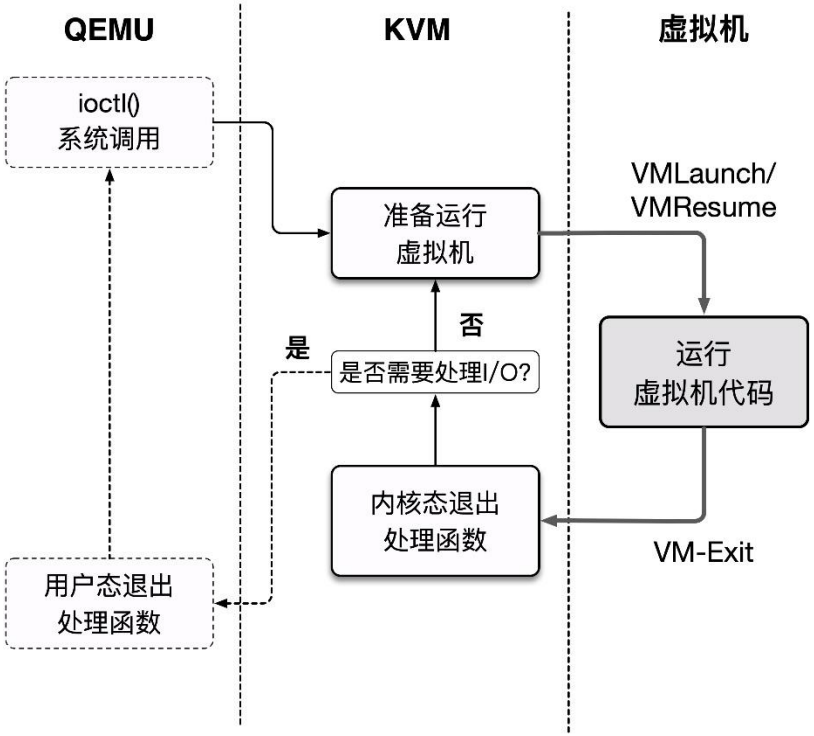


Virtualization Mode (V)	Nominal Privilege	Abbreviation	Name
0	U	U-mode	User mode
0	S	HS-mode	Hypervisor-extended supervisor mode
0	M	M-mode	Machine mode
1	U	VU-mode	Virtual user mode
1	S	VS-mode	Virtual supervisor mode

目录 Contents

QEMU/KVM CPU虚拟化实现简介

- QEMU/KVM CPU虚拟化实现可以概括为“**三个阶段，两种转换**”
  - 三个阶段
    - **初始化阶段：**创建虚拟机、创建vCPU
    - **虚拟机运行阶段：**运行虚拟机指令
    - **异常处理阶段：**陷入Hypervisor，处理相应异常
  - 两种转换
    - 虚拟机陷入
    - 虚拟机陷出

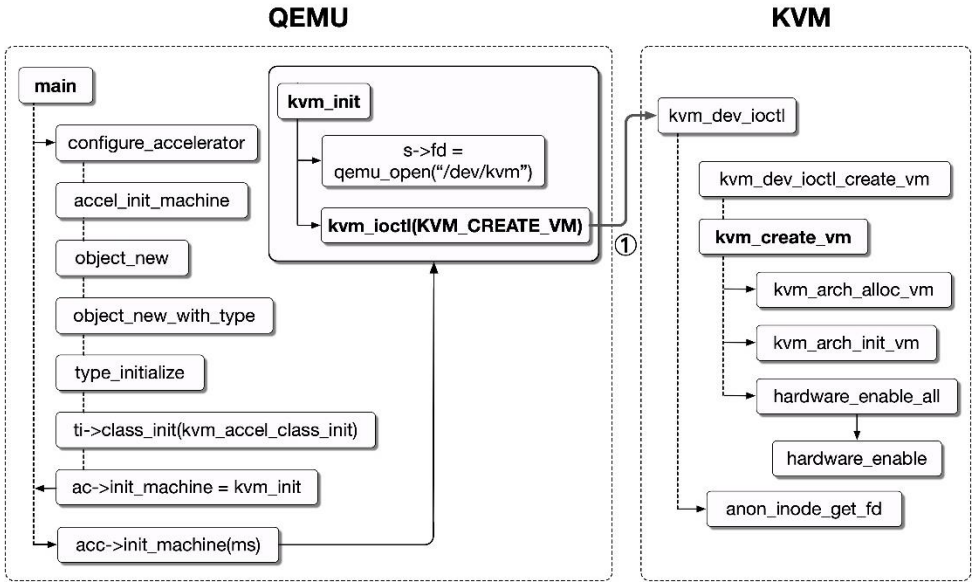


KVM模块初始化

- KVM模块初始化主要工作
  - **创建设备文件**/dev/kvm，并注册设备文件读写回调kvm_chardev_ops；后续QEMU会打开/dev/kvm设备文件调用相应的回调创建虚拟机
  - 根据当前CPU能力**创建全局VMCS配置**vmcs_config，后续根据vmcs_config初始化VMCS

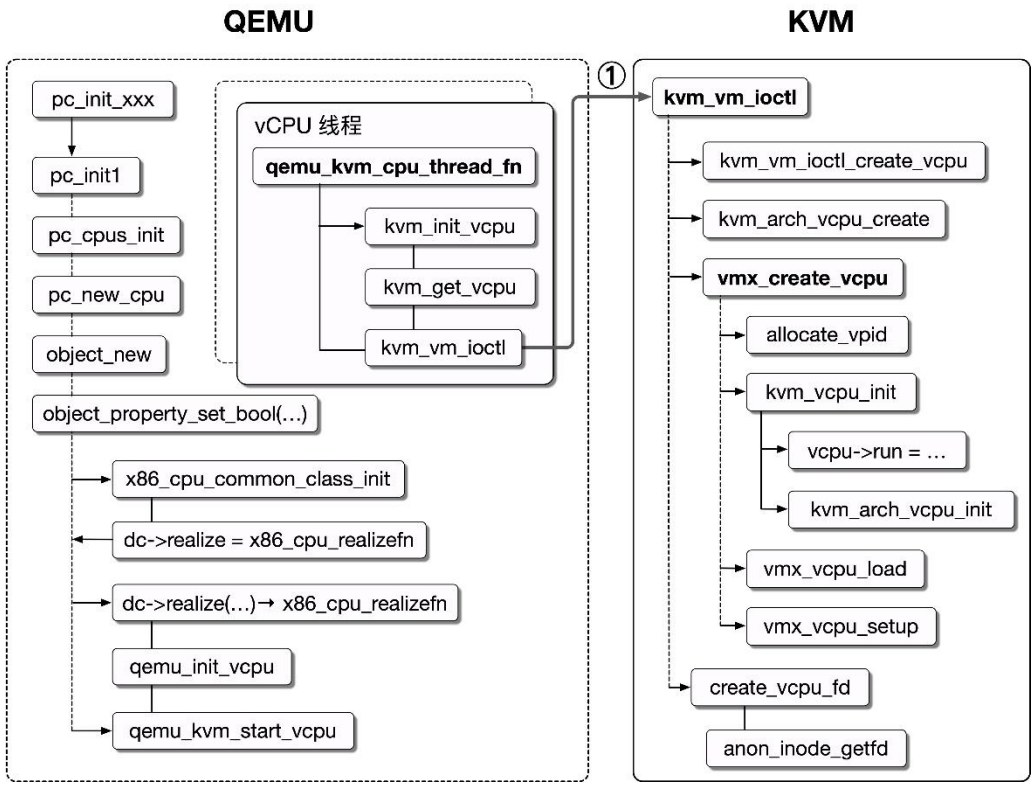
创建虚拟机

- KVM模块初始化后，QEMU即可打开/dev/kvm设备文件发起创建虚拟机的请求。虚拟机创建流程如下：
  - QEMU侧解析命令行参数，创建相应类型的虚拟机结构体(QOM机制)
  - QEMU打开/dev/kvm设备文件，向KVM发起创建虚拟机的请求
  - KVM**创建并初始化虚拟机结构体**
  - 在每个CPU上**执行vmxon指令**，使能VMX操作模式
  - **为虚拟机创建一个匿名文件**，并将文件描述符返回给QEMU
  - QEMU后续可以通过虚拟机文件描述符调用虚拟机相应的接口，如创建vCPU等



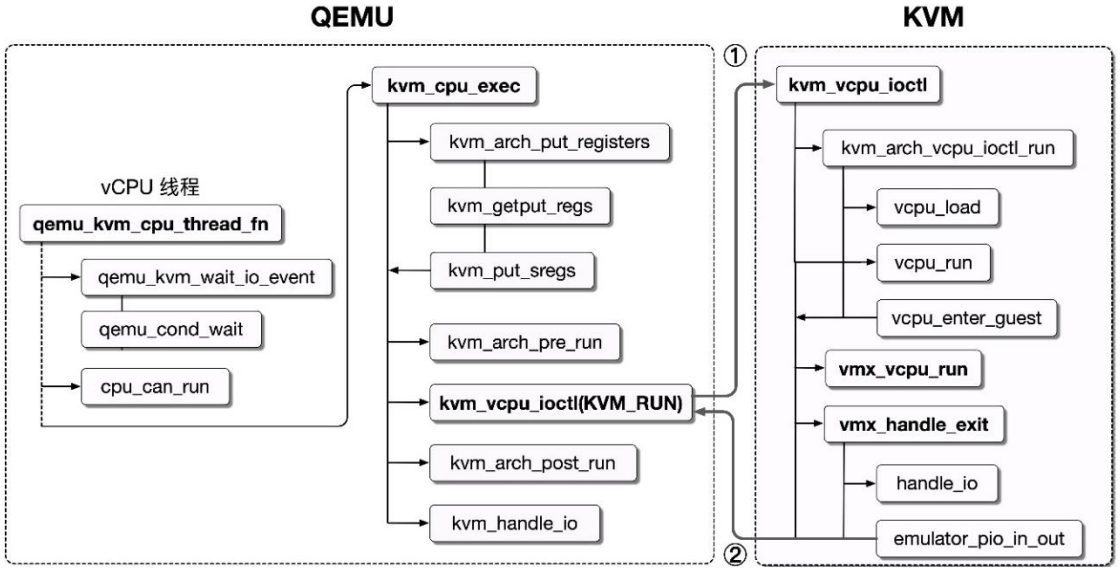
创建vCPU

- 虚拟机创建完成后，QEMU便可以通过前述虚拟机文件描述符发起IOCTL请求KVM创建vCPU
  - QEMU根据命令行传入的CPU类型及CPU数目创建并初始化相应类型的vCPU结构体
  - QEMU**为每个vCPU创建一个线程**并运行
  - vCPU线程通过虚拟机文件描述符**请求KVM创建vCPU结构体**
  - KVM创建并初始化vCPU结构体
  - KVM加载VMCS并**初始化**VMCS中的各个字段
  - KVM**为每个vCPU创建一个匿名文件**并将文件描述符返回给QEMU



vCPU运行

- vCPU创建后，QEMU便可以通过vCPU文件描述符请求KVM运行vCPU，主要流程如下：
  - QEMU通过vCPU文件描述符调用相应KVM接口**设置通用寄存器和段寄存器**的值
  - QEMU通过vCPU文件描述符发起KVM_RUN请求陷入KVM
  - **KVM检查当前是否有未处理的事件，如中断、异常等；若有，处理这些事件**
  - 将宿主机状态保存至VMCS宿主机状态域
  - **调用vmlaunch或vmresume指令**进入非根模式运行虚拟机代码
  - 虚拟机执行敏感指令，触发VM-Exit陷入KVM
  - KVM从VMCS中**读取虚拟机退出原因并调用相应的处理函数**；若KVM无法处理，则退出至QEMU进行处理
  - 处理完成后，回到第2步，如此循环往复



实验：sample-qemu

sample-qemu.c

```
int main(void){
int kvm, vmfd, vcpufd, ret;
    uint8_t *mem;
    struct kvm_sregs sregs;
    size_t mmap_size;
    struct kvm_run *run;
    const uint8_t code[] = {
        0xba, 0xf8, 0x03, /* mov $0x3f8, %dx */
        0xb0, 'H',        /* mov $'H', %al */
        0xee,             /* out %al, (%dx) */
        …
        0xf4,             /* hlt */
    };  /* 写入指定端口0x3f8，输出Hello, World! */
    kvm = open("/dev/kvm", O_RDWR | O_CLOEXEC); /* 打开KVM模块设备文件 */
    ret = ioctl(kvm, KVM_GET_API_VERSION, NULL);/* 获取KVM API版本 */
    /* 创建虚拟机获得虚拟机文件描述符 */
    vmfd = ioctl(kvm, KVM_CREATE_VM, (unsigned long)0);
    /* 分配4KB大小的内存空间存放二进制代码 */
    mem = mmap(NULL, 0x1000, PROT_READ | PROT_WRITE,
               MAP_SHARED | MAP_ANONYMOUS, -1, 0);
```

实验：sample-qemu

sample-qemu.c

```
memcpy(mem, code, sizeof(code)); /* 将二进制代码拷贝至分配的内存页中 */
struct kvm_userspace_memory_region region = {
    .slot = 0,
    .guest_phys_addr = 0x1000,
    .memory_size = 0x1000,
    .userspace_addr = (uint64_t)mem,
};  /* 将该内存页映射至虚拟机物理地址0x1000 (GPA) 处 */
ret = ioctl(vmfd, KVM_SET_USER_MEMORY_REGION, &region);
/* 创建vCPU获得vCPU文件描述符 */
vcpufd = ioctl(vmfd, KVM_CREATE_VCPU, (unsigned long)0);
/* 获取QEMU/KVM共享内存空间大小并映射kvm_run结构体 */
mmap_size = ioctl(kvm, KVM_GET_VCPU_MMAP_SIZE, NULL);
run = mmap(NULL, mmap_size, PROT_READ | PROT_WRITE,
           MAP_SHARED, vcpufd, 0);
/* 设置CS寄存器与RIP寄存器使得vCPU从0x1000处开始执行 */
ret = ioctl(vcpufd, KVM_GET_SREGS, &sregs);
sregs.cs.base = 0;
sregs.cs.selector = 0;
ret = ioctl(vcpufd, KVM_SET_SREGS, &sregs);
```

实验：sample-qemu

sample-qemu.c

```
    struct kvm_regs regs = {
        .rip = 0x1000,
    };
    ret = ioctl(vcpufd, KVM_SET_REGS, &regs);
    while (1) {
        ret = ioctl(vcpufd, KVM_RUN, NULL); /* 运行vCPU */
        switch (run->exit_reason) {          /* 处理VM-Exit*/
        case KVM_EXIT_HLT:
            return 0;                        /* 退出程序 */
        case KVM_EXIT_IO:
            /* 输出写入0x3f8端口的字符*/
            if (run->io.direction == KVM_EXIT_IO_OUT
                && run->io.size == 1 && run->io.port == 0x3f8
                && run->io.count == 1)
                putchar(*(((char *)run) + run->io.data_offset));
        }
    }
}
```

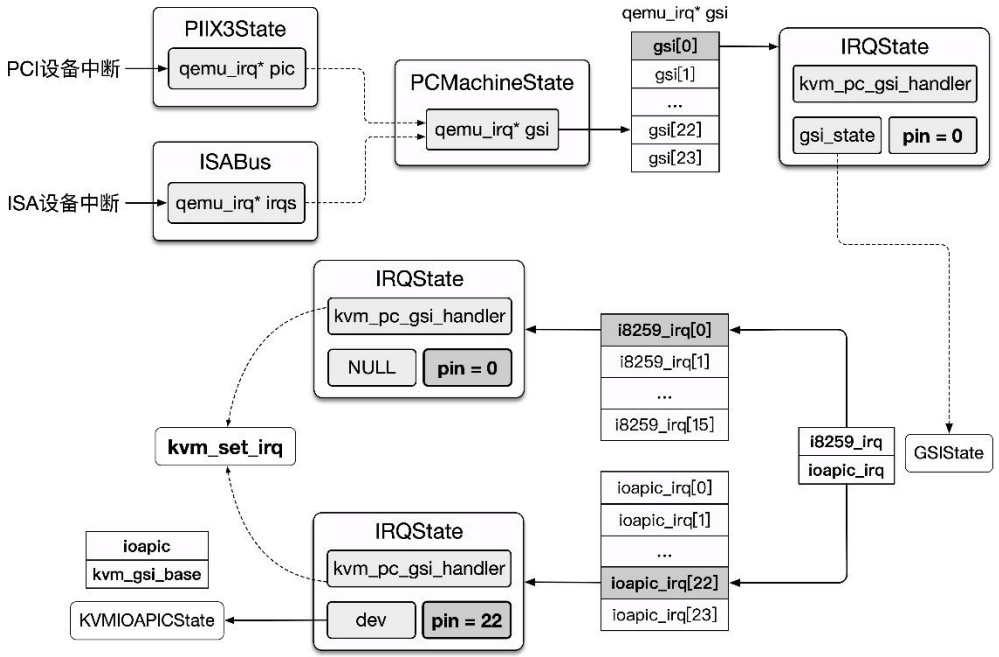
目录 Contents

中断控制器模拟

- QEMU/KVM都实现了对PIC/APIC中断控制器的模拟
- QEMU提供-kernel-irqchip**参数**决定中断控制器的模拟方式
  - kernel-irqchip=on：PIC和APIC中断控制器均由KVM模拟
  - kernel-irqchip=off：PIC和APIC中断控制器均由QEMU模拟
  - kernel-irqchip=split：PIC和IOAPIC由QEMU模拟，LAPIC由KVM模拟

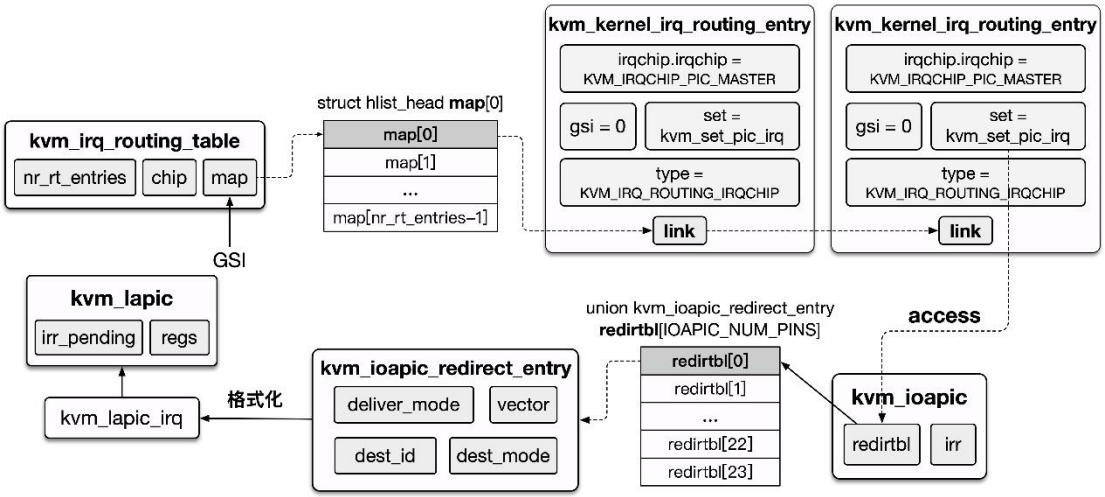
QEMU中断路由

- QEMU中断路由流程
  - QEMU中断路由的**起点位于pcms->gsi**，它本质上是一个qemu_irq数组，为所有的虚拟设备定义了统一的中断传递入口
  - QEMU根据传入的中断控制器引脚号以及系统当前使用的中断控制器类型决定该中断如何进行传递。
  - **若IRQ < 16，调用i8259_irq中的回调函数**(因为PIC最多支持16个中断源)
  - **若IRQ >= 16，调用ioapic_irq中注册的回调函数**(IOAPIC有24个引脚，可以支持24个中断源)
  - 上述回调**最终都会调用**kvm_set_irq将中断信号传给由KVM模拟的LAPIC
  - 实际上当IRQ号小于16时，KVM会将中断信号**同时发送给虚拟PIC和虚拟IOAPIC**，虚拟机通过当前真正使用的中断控制器接收该中断信号。



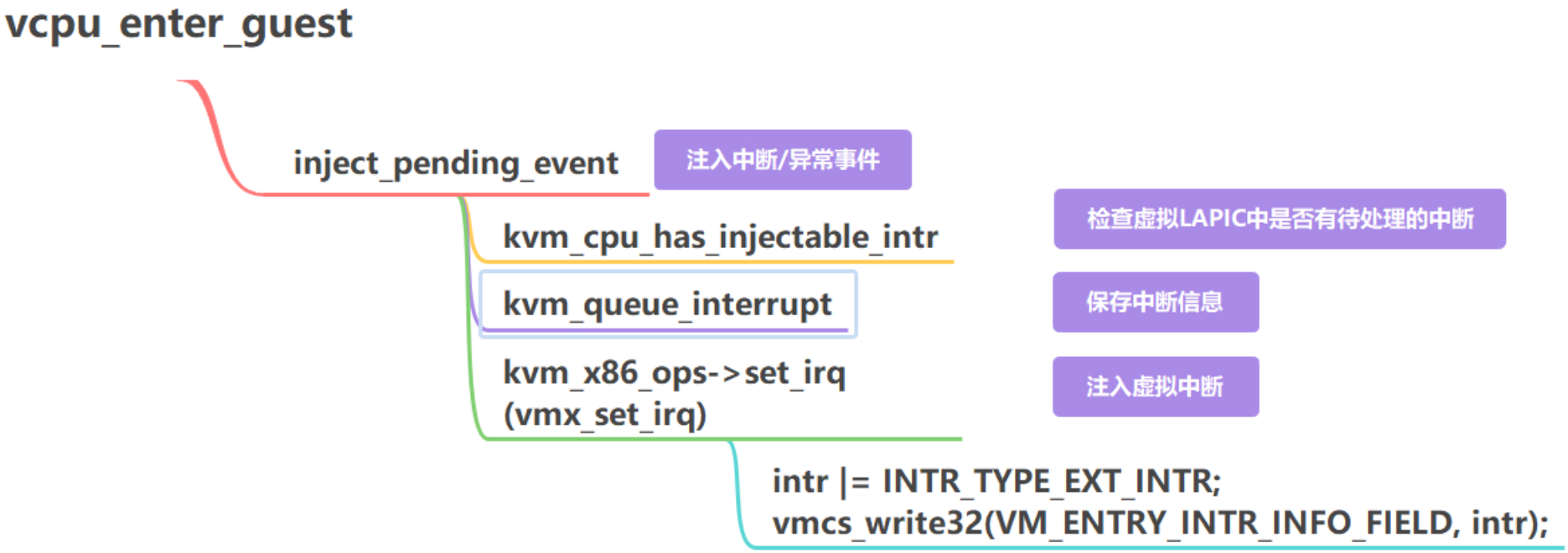
KVM中断路由

- KVM**中断路由由**struct kvm_irq_routing_table**完成**，类似于QEMU中的GSIState，根据传入的IRQ号调用不同的回调函数。具体流程如下：
  - **当**IRQ < 16**时，**KVM**会同时将中断信号发送给虚拟**PIC**和虚拟**IOAPIC，虚拟机会通过当前使用的中断控制器接收该中断信号，其余中断信号将被忽略
  - **当**IRQ >= 16**时，中断信号只会发送给虚拟**IOAPIC
  - 当终端信号到达IOAPIC时，KVM会**模拟IOAPIC中的中断重定向表**(Interrupt Redirection Table)。根据表项中的中断配置信息，KVM模拟一条中断消息传递给目标LAPIC。
  - LAPIC调用相应回调设置IRR寄存器，并在目标vCPU中记录一个中断请求
  - 而后调用kvm_vcpu_kick函数通知目标vCPU，**若目标vCPU处于睡眠状态，唤醒该vCPU。否则，发送IPI使其发生虚拟机陷出**
  - vCPU再次进入非根模式前，会处理前述中断请求，**向虚拟机中注入一个虚拟中断**



虚拟中断注入

- 虚拟机在进入非根模式之前，会**检查虚拟LAPIC中是否有待处理的中断**
- 若有，保存该中断信息并**设置VM-Entry控制域中断信息字段**(Interruption Information)
- 在VM-Entry时，CPU会检查该字段发现有待处理的中断并用中断向量号索引IDT执行相应的处理函数



实验：e1000网卡中断虚拟化

- 实验目标：使用GDB调试工具熟悉e1000网卡中断递送流程
- 实验概述
  - 为了使用GDB调试虚拟中断在KVM中的递送流程，本次实验在嵌套虚拟化环境中进行
  - 本次实验所使用的**内核版本均为**4.19.0
- 实验流程
  - 启动虚拟机，等待远程GDB连接
  - 加载KVM模块调试信息
  - 使用GDB运行QEMU，启动嵌套虚拟机
  - 通过QEMU monitor查询e1000网卡中断相关信息
  - 使用GDB调试中断传递流程

启动虚拟机

- 启动虚拟机。QEMU提供-s和-S选项允许GDB远程连接调试内核，其中-s选项使得QEMU等待来自1234端口的TCP连接，-S选项则会使得QEMU阻塞客户机执行直到远程连接的GDB允许它继续执行

Physical Machine Terminal 1

```
./qemu/x86_64-softmmu/qemu-system-x86_64 -s -S -smp 4 -m 4096 -cpu host \
-kernel linux/arch/x86/boot/bzImage -initrd initrd.img-4.19.0 \
-append "root=/dev/sda1 nokaslr" \
-drive file=desktop.img,format=raw,index=0,media=disk \
-netdev tap,id=mynet0,ifname=tap0,script=no,downscript=no \
-device e1000,netdev=mynet0,mac=52:55:00:d1:55:01 --enable-kvm
```

- 在Terminal 2启动GDB加载Linux 内核调试信息并连接至1234端口，而后开始运行虚拟机。

Physical Machine Terminal 2

```
gdb vmlinux
(gdb) target remote:1234
(gdb) c
```

加载KVM模块调试信息

- 在虚拟机中通过/sys/module/module_name /sections查看查看kvm.ko和kvm-intel.ko模块所在的GPA，并在GDB中**手动引入KVM模块的调试信息**

Virtual Machine Terminal 1

```
sudo cat /sys/module/kvm/sections/.text
0xffffffffc01a4000
sudo cat /sys/module/kvm/sections/.data
0xffffffffc0209040
sudo cat /sys/module/kvm/sections/.bss
0xffffffffc0218900
sudo cat /sys/module/kvm_intel/sections/.text
0xffffffffc039c000
sudo cat /sys/module/kvm_intel/sections/.data
0xffffffffc03c7aa0
sudo cat /sys/module/kvm_intel/sections/.bss
0xffffffffc03c8880
```

Physical Machine Terminal 2

```
(gdb) add-symbol-file kvm.ko 0xffffffffc01a4000 -s .data 0xffffffffc0209040
-s .bss 0xffffffffc0218900
(gdb) add-symbol-file kvm-intel.ko 0xffffffffc039c000 -s .data 0xffffffffc0
3c7aa0 -s .bss 0xffffffffc03c8880
(gdb) c
```

启动嵌套虚拟机

- **通过GDB运行QEMU启动嵌套虚拟机，可以查看中断在QEMU中产生以及路由的流程**
- QEMU提供-monitor参数允许使用者远程连接至QEMU监视器查看虚拟机内部信息

Virtual Machine Terminal 2

```
gdb -arg ./qemu/x86_64-softmmu/qemu-system-x86_64 -smp 2 -m 2048 -cpu host \
-kernel linux/arch/x86/boot/bzImage -initrd initrd.img-4.19.0 \
-append "root=/dev/sda1 nokaslr" -machine kernel-irqchip=on \
-drive file=desktop.img,format=raw,index=0,media=disk \
-netdev tap,id=mynet0,ifname=tap0,script=no,downscript=no \
-device e1000,netdev=mynet0,mac=52:55:00:d1:55:02 \
-monitor telnet:127.0.0.1:4444 --enable-kvm
(gdb) handle SIGUSR1 nostop
(gdb) handle SIGUSR2 nostop
(gdb) start
```

查看e1000网卡中断信息

- 在虚拟机中连接QEMU monitor，查看e1000网卡相关信息

Virtual Machine Terminal 3

```
telnet 127.0.0.1 4444
QEMU 4.1.1 monitor - type 'help' for more information
(qemu) info qtree /* 列出qemu结构树，以下只展示部分输出*/
bus: main-system-bus
dev: kvm-ioapic, id ""
  gpio-in "" 24
  gsi_base = 0 (0x0)
  mmio 00000000fec00000/0000000000001000
dev: i440FX-pcihost, id ""
pci-hole-size = 2147483648 (2 GiB)
short_root_bus = 0 (0x0)
x-pci-hole64-fix = true
bus: pic.0
  type PCI
  dev: e1000, id ""
    mac = "52:55:00:d1:55:02"
    netdev = "mynet1"
    multifunction = false
```

查看e1000网卡中断信息

- 在嵌套虚拟机中启动一个终端并执行lspci -v指令查看e1000网卡具体信息

Nested Virtual Machine Terminal 1

```
lspci -v
00:03.0 Ethernet controller: Intel Corporation 82540EM Gigabit Ethernet
Controller (rev 03)
        Subsystem: Red Hat, Inc. QEMU Virtual Machine
        Physical Slot: 3
        Flags: bus master, fast devsel, latency 0, IRQ 11
        Memory at febc0000 (32-bit, non-prefetchable) [size=128K]
        I/O ports at c000 [size=64]
        Expansion ROM at feb80000 [disabled] [size=256K]
        Kernel driver in use: e1000
```

- 上述信息表明e1000的BDF为00:03.0，即24，而e1000使用的中断IRQ号为11。而后使用QEMU监视器键入info pic查看虚拟机IOAPIC每一个引脚所连接设备信息。

Virtual Machine Terminal 3

```
(qemu) info pic
ioapic0: ver=0x11 id=0x00 sel=0x26 (redir[11])
pin 0  0x0000000000010000 dest=0 vec=0  active-hi edge masked fixed physical
…
pin 11 0x0000000000008026 dest=1 vec=38 active-hi level fixed physical
…
```

GDB调试e1000网卡中断递送流程

- 在做好前面的准备工作后，我们可以在e1000网卡中断递送流程的任意时刻断点查看虚拟中断相关信息
- 如对e1000网卡而言，无论是收包中断还是发包中断，最后**都会调用**set_interrupt_cause**函数**，在此处断点可以查看e1000网卡的设备号

Virtual Machine Terminal 2

```
(gdb) break set_interrupt_cause
(gdb) c
Continuing
Thread 1 "qemu-sysrem-x86" hit Breakpoint 2, set_interrupt_cause(
     s=0x7ffff4ab8010,index=0, val=0) at hw/net/e1000.c:270
270         {
(gdb) n
275             PCIDevice *d = PCI_DEVICE(s);
(gdb) n
271             s->mac_reg[ICR] = val;
(gdb) print d->devfn
$1 = 24 // 与嵌套虚拟机lspci -v打印的设备号相符
```

GDB调试e1000网卡中断递送流程

- 前面讲到QEMU**最终会调用**kvm_set_irq将中断信号传入KVM。在此处设置条件断点截获e1000网卡中断并打印函数栈帧即可看到e1000网卡在QEMU中路由的全流程

Virtual Machine Terminal 2

```
(gdb) break accel/kvm/kvm-all.c:1187 if irq == 11 // lspci -v显示e1000 IRQ=11
(gdb) c
Continuing
Thread 1 "qemu-system-x86" hit Breakpoint 6, kvm_set_irq (s=0x5555568dc770,
irq=11, level=1) at /home/sin/qemu/accel/kvm/kvm-all.c:1190
1190                    assert(kvm_async_interrupts_enabled());
(gdb) backtrace /* 打印当前函数调用栈帧 */
#0 kvm_set_irq (s=0x5555568dc770, irq=11, level=1)
    at /home/sin/qemu/accel/kvm/kvm-all.c:1190
#1 0x00005555559598a6 in kvm_pic_set_irq (opaque=0x0, irq=11, level=1)
    at /home/sin/qemu/hw/i386/kvm/i8259.c:117
#2 0x0000555555a7398a in qemu_set_irq (irq=0x555556e87b10, level=1)
    at hw/core/irq.c:44
#3 0x0000555555959c38 in kvm_pc_gsi_handler (opaque=0x555556a65810, n=11,
level=1) at /home/sin/qemu/i386/kvm/ioapic.c:55
#4 0x0000555555a7398a in qemu_set_irq (irq=0x555556a6d800, level=1)
at hw/core/irq.c:44
...
```

GDB调试e1000网卡中断递送流程

- 中断信号传入KVM后经由前面讲到的内核中断路由表传递给IOAPIC，**相应的回调函数为**kvm_set_ioapic_irq，在此处断点。值得注意的是，此时中断流程位于虚拟机的KVM中，我们需要在物理机的GDB中断点

Physical Machine Terminal 1

```
(gdb) hbreak arch/x86/kvm/irq_comm.c:54 if e->gsi == 11
(gdb) hbreak kvm_ioapic_set_irq
(gdb) c
Continuing
Thread 2 hit Breakpoint 1, kvm_set_ioapic_irq (e=0xffffc900019efc90,
     kvm=0xffffc90001b19000, irq_source_id=0, level=1, line_status=true)
     at arch/x86/kvm/irq_comm.c:54
54        arch/x86/kvm/irq_comm.c: No such file or directory.
(gdb) c
Continuing
Thread 2 hit Breakpoint 2, kvm_ioapic_set_irq (ioapic=0xffff88012cb79800,
     irq=11, irq_source_id=0, level=1, line_status=true)
     at arch/x86/kvm/ioapic.c:386
386       arch/x86/kvm/irq_comm.c: No such file or directory.
```

GDB调试e1000网卡中断递送流程

- kvm_ioapic_set_irq函数**最终调用ioapic_service函数**处理指定引脚的中断请求。ioapic_service函数根据**传入的中断引脚号在PRT（ioapic->redirtbl）中找到相应的RTE（entry）并格式化出一条中断消息（irqe）发送给所有的目标LAPIC**。irqe中包含的供CPU最终使用的中断向量号。打印irqe查看e1000网卡中断向量号可以发现e1000网卡**对应的中断向量号为38，触发方式为水平触发**（trig_mode = 1），与通过QEMU监视器执行info pic输出信息一致。

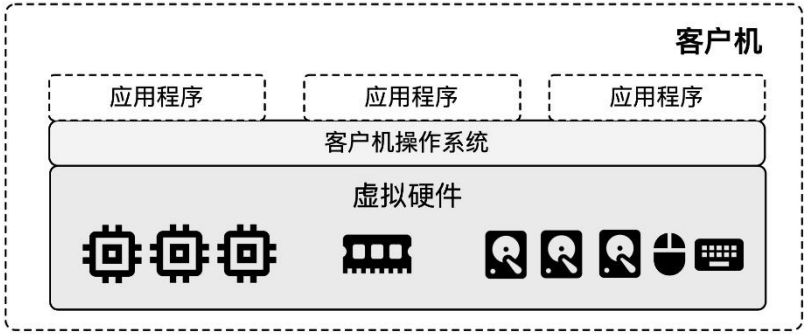
Physical Machine Terminal 1

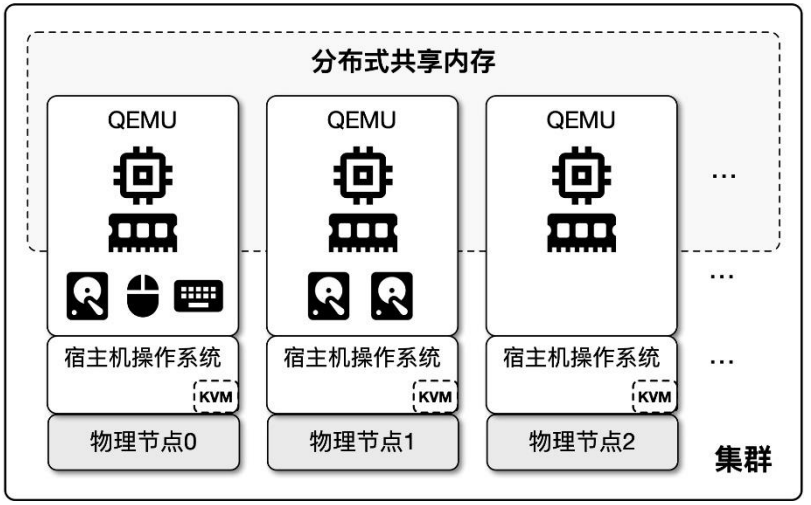
```
(gdb) hbreak arch/x86/kvm/ioapic.c:361 if irq == 11
(gdb) c
Continuing
Thread 2 hit Breakpoint 3,ioapic_service (ioapic=0xffff88007f4df000, irq=11,
     line_status=true) at arch/x86/kvm/ioapic.c:361
361       in arch/x86/kvm/ioapic.c
(gdb) print irqe // 打印中断消息
$1 = {vector = 38, delivery_mode = 65535. dest_mode = 0, level = true,
trig_mode = 1, shorthand = 0, dest_id = 1, msi_redir_hint = false}
```

目录 Contents

GiantVM

- 基于QEMU/KVM的“多虚一”虚拟化架构GiantVM
  - 在多个物理节点上运行一个虚拟机
  - 跨节点CPU/内存/IO资源聚合
  - 分布式Hypervisor
  - 分布式共享内存
  - 跨节点中断转发





分布式vCPU

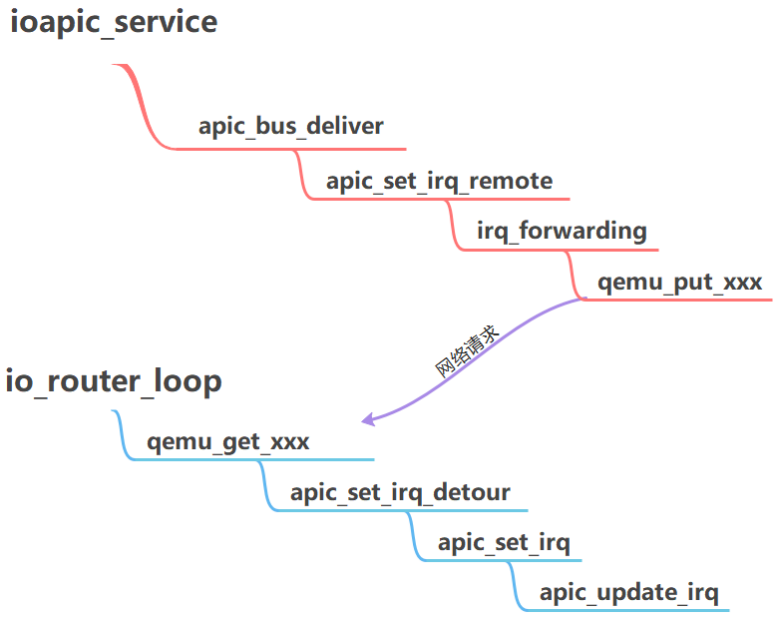
- 每个物理节点上创建等同于虚拟机CPU数目的vCPU线程
- **运行在当前物理节点的vCPU标记为**local，同前述vCPU运行流程一致
- **运行在远端物理节点的vCPU标记为**remote，处于睡眠状态，直至虚拟机关机时被销毁

giantvm-qemu/hw/i386/pc.c

```
static void *qemu_kvm_cpu_thread_fn(void *arg)
{
  if (cpu->local) {
    do {
      if (cpu_can_run(cpu)) {
        r = kvm_cpu_exec(cpu);
        }
        …
    } while (!cpu->unplug || cpu_can_run(cpu));
  } else {
    while (1) {
      qemu_cond_wait(&qemu_remote_cpu_cond,
                     &qemu_remote_mutex);
      wait_remote_cpu_count--;
        if (qemu_shutdown_requested_get()) {
        cpu->stopped = true;
        break;
        }
        if (qemu_reset_requested_get()) {
        cpu->stopped = true;
        }
    }
```

跨节点中断转发

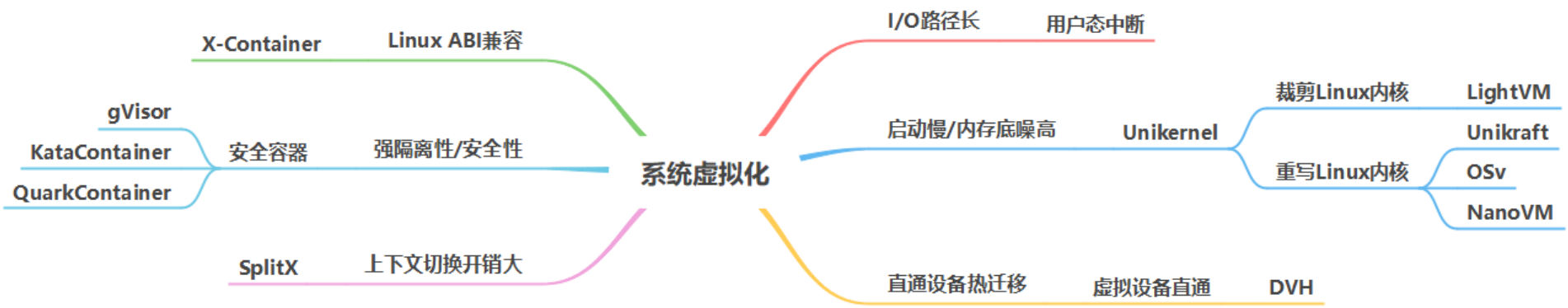
- 以下场景需要进行跨节点中断转发
  - 位于物理机0上的vCPU 0需要发送给位于物理机1上的vCPU 1发送一个IPI
  - 挂载在物理机0的I/O设备需要将中断发送给位于物理机1 的vCPU时
- 解决方案
  - 在当前节点与目标物理节点间**预先建立网络连接**
  - 当虚拟中断控制器根据中断目的CPU标识符识别出发送给远程vCPU的中断时，**根据中断类型进行转发，再由目的节点调用中断控制器的回调函数**进行中断注入



目录 Contents

虚拟化研究现状

- 系统虚拟化特点
  - 优点：**强隔离性/安全性**、Linux **ABI兼容**(运行完整的Guest OS)
  - 缺点：**上下文切换开销大**、**I/O路径过长**、**启动慢**、**内存底噪高**
- CPU虚拟化研究方向
  - **轻量级虚拟化**
  - **安全容器**



轻量级虚拟化

- 基本假设
  - **云上虚拟机通常只运行单一应用**
  - 虚拟机OS，如Linux，过于繁重，具有很多不必要的功能
  - 可以基于特定应用**定制虚拟机内核**
- 虚拟机内核定制方式——“**减或加**”
  - 减：**基于Linux内核进行裁剪**，如LightVM
  - 加：实现一个全新的OS，再根据应用需求慢慢增加功能，如NanoVM/OSv/Unikraft
  - Tradeoff：内核的简易程度与通用性
- 内核定制化的缺点：无法实现ABI完全兼容
  - X-Container：提出基于Xen-PV架构，**将Linux作为LibOS，Xen作为Exokernel**
  - 优点：**将Syscall替换为Hypercall**，相比于容器，减少了底层攻击面
  - 缺点：上层应用可能会受到GPL**协议污染**

安全容器

- Kata Container
  - **将容器运行在虚拟机中**，实现容器之间的强隔离，从而保证安全性
- gVisor
  - ptrace platform：使用ptrace来截获系统调用
  - KVM platform：将sentry作为GuestOS，通过硬件辅助虚拟化能力截获Syscall，提升安全性以及性能
- Quark Container
  - 类似于gVisor KVM架构，但是Guest Space与Host Space之间**通过共享内存的方式进行通信**避免了虚拟机退出

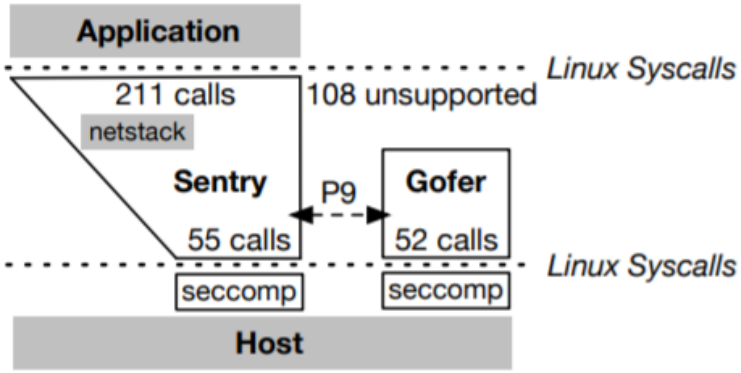
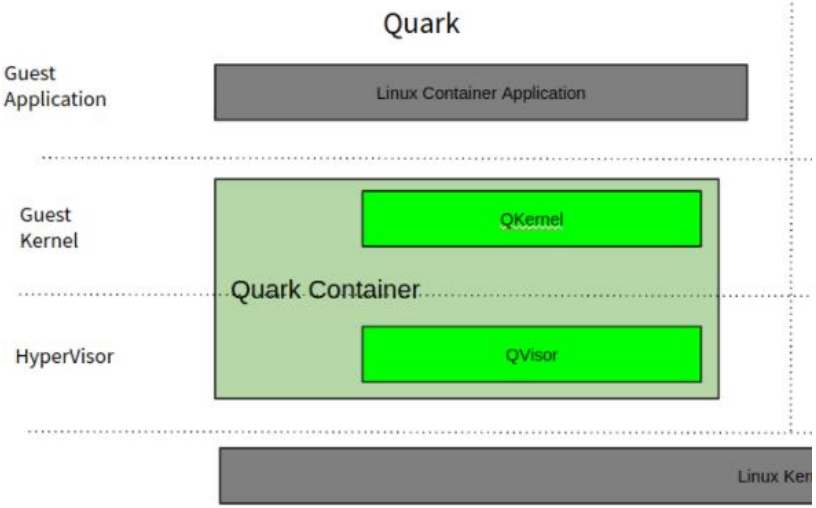


Figure 1: **gVisor Architecture.**



谢 谢!