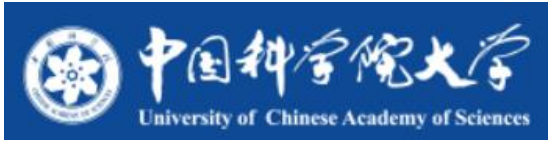
ISCAS
中国科学院软件研究所
Institute of Software,Chinese Academy of Sciences
中国科学院大学
University of Chinese Academy of Sciences

内存虚拟化

改编声明

- 本课程教学及PPT内容基于**上海交通大学并行与分布式系统研究所**发布的操作系统课程修改，原课程官网：
  - https://ipads.se.sjtu.edu.cn/courses/os/index.shtml
- 本课程修改人为**中国科学院软件研究所**，用于国科大操作系统课程教学。



Review: RISC-V是严格的可虚拟化架构

Our goals in defining RISC-V include:

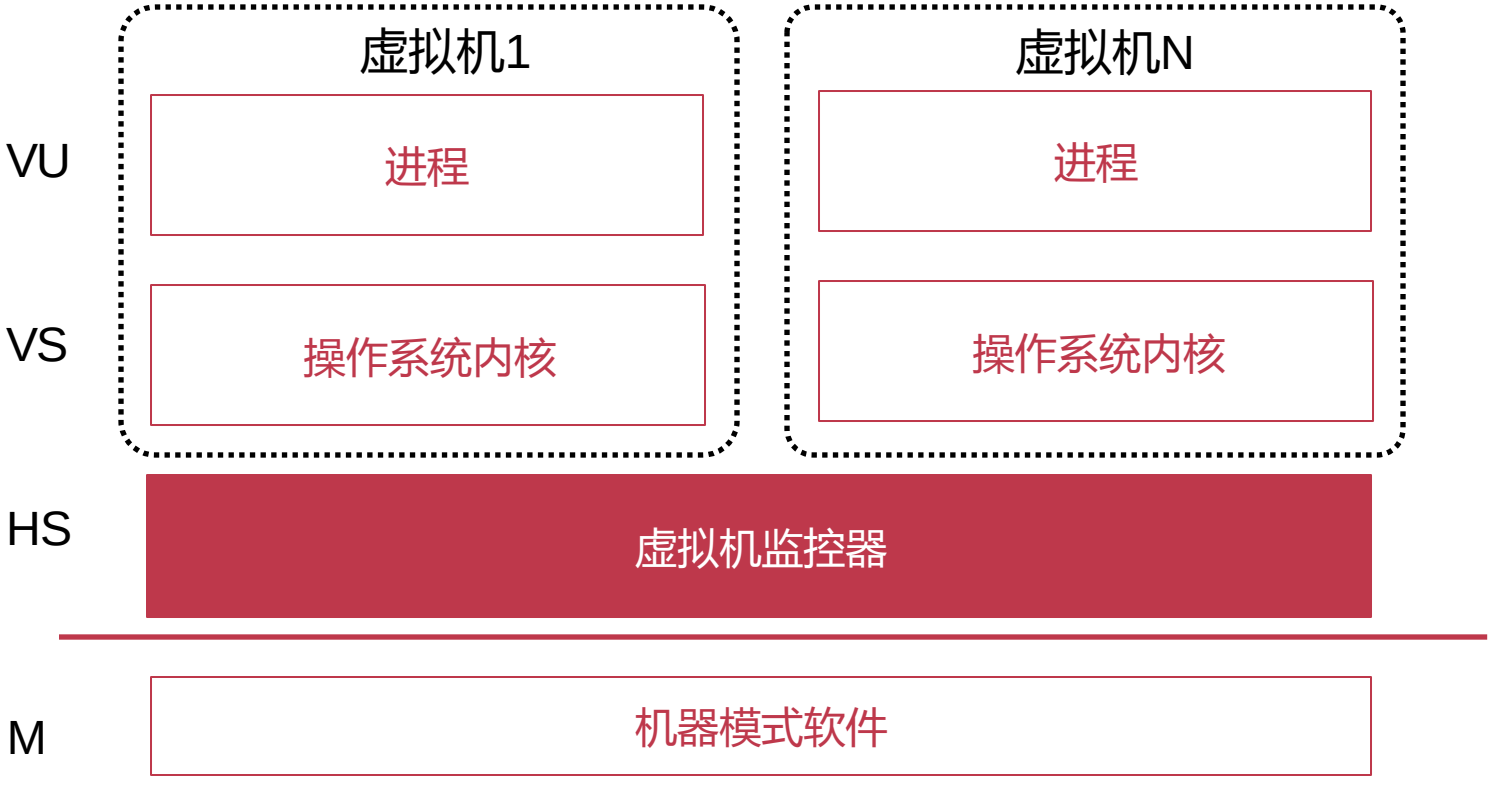
- *[…] A fully virtualizable ISA to ease hypervisor development. […]*

- **高特权级严格控制低特权级的敏感指令**
  - 允许性能计数器访问：由 mcounteren 和 scounteren 控制
  - 虚拟内存，特权级切换：由 mstatus.TVM/TSR 控制
  - ……

RISC-V的虚拟化技术

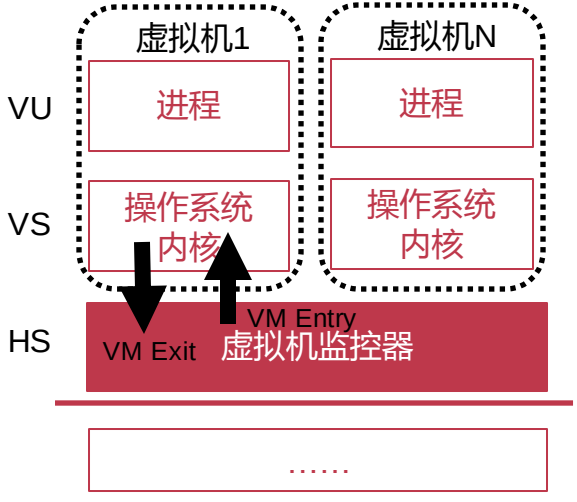
Review: Intel VT-x的处理器虚拟化

RISC-V的处理器虚拟化



RISC-V的VM Entry和VM Exit

- **VM Entry**
  - 使用SRET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、CSR
    - VM的控制状态：hstatus、hgatp等
- **VM Exit**
  - 虚拟机执行敏感指令或收到中断等
  - 以Exception、Interrupt的形式回到VMM
    - 调用VMM记录在stvec中的处理函数
  - 下陷第一步：VMM主动保存所有VM的状态



RISC-V硬件虚拟化的新功能

- **RISC-V中没有VMCS**
- **VM能直接控制VS和VU模式的状态**
  - 可以直接读写satp/sstatus/...
  - 访问satp/sstatus/...实际访问的是vsatp/vsstatus/...，并受这些 vs 寄存器控制
- **VM Exit时VMM可以访问vsatp/vsstatus/...**

- **思考题1：为什么RISC-V中可以不需要VMCS?**
- **思考题2：RISC-V中没有VMCS，对于VMM的设计和实现来说有什么优缺点?**

Hypervisor CSR 简介

- **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**
  - hstatus.TVM：VM执行虚拟内存操作是否下陷，如读写satp，sfence.vma
  - hstatus.TW：执行WFI指令是否下陷
  - hedeleg/hideleg：Exception/Interrupt委托还是下陷
  - hgatp: 控制第二阶段地址翻译

Hypervisor CSR简介

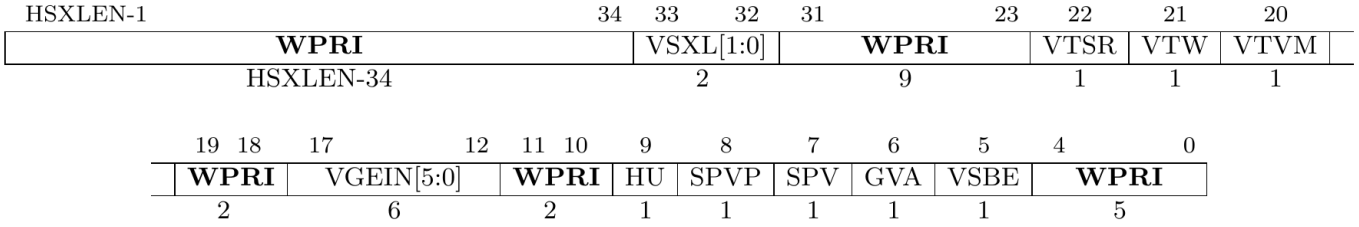


Figure 8.2: Hypervisor status register (`hstatus`) when HSXLEN=64.

Figure 8.3: Hypervisor exception delegation register (`hedeleg`).

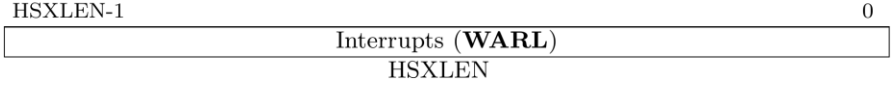
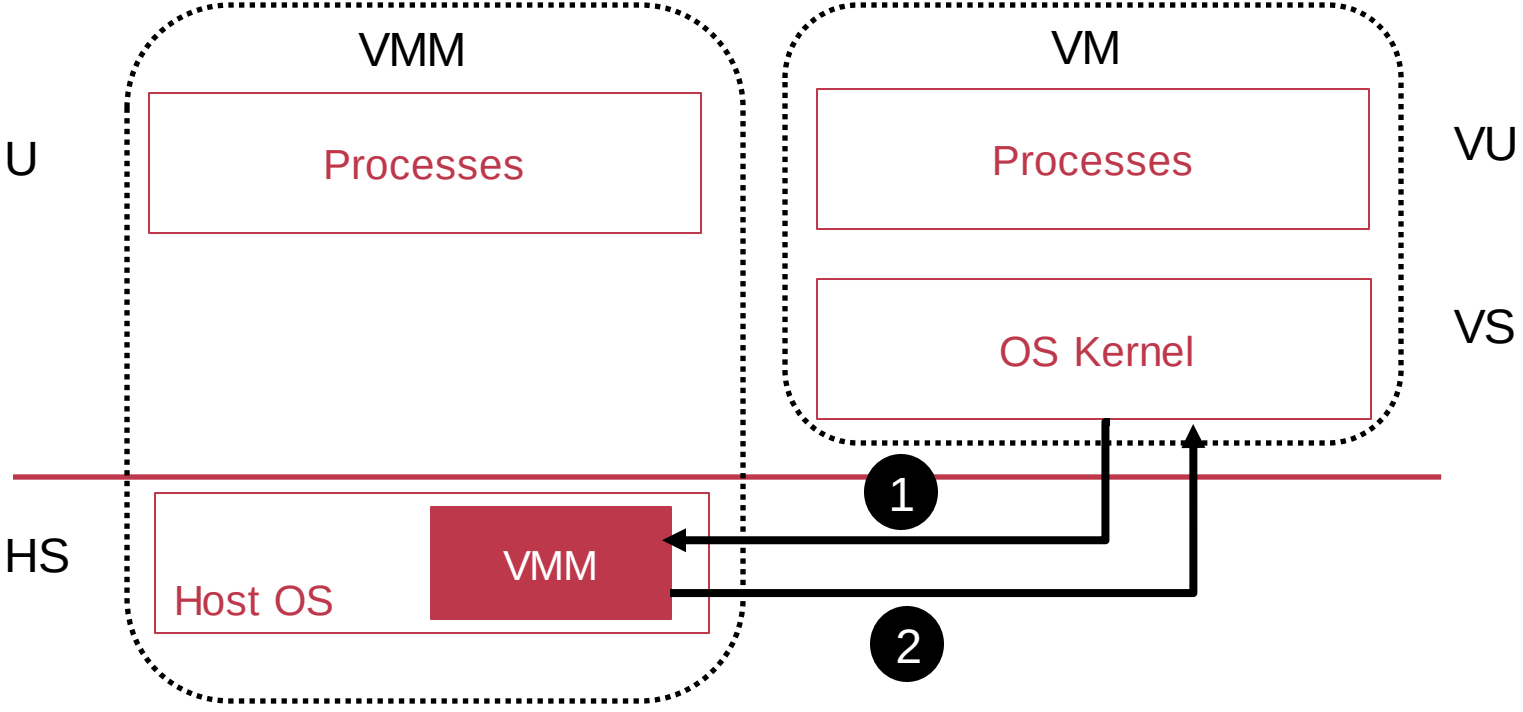


Figure 8.4: Hypervisor interrupt delegation register (`hideleg`).

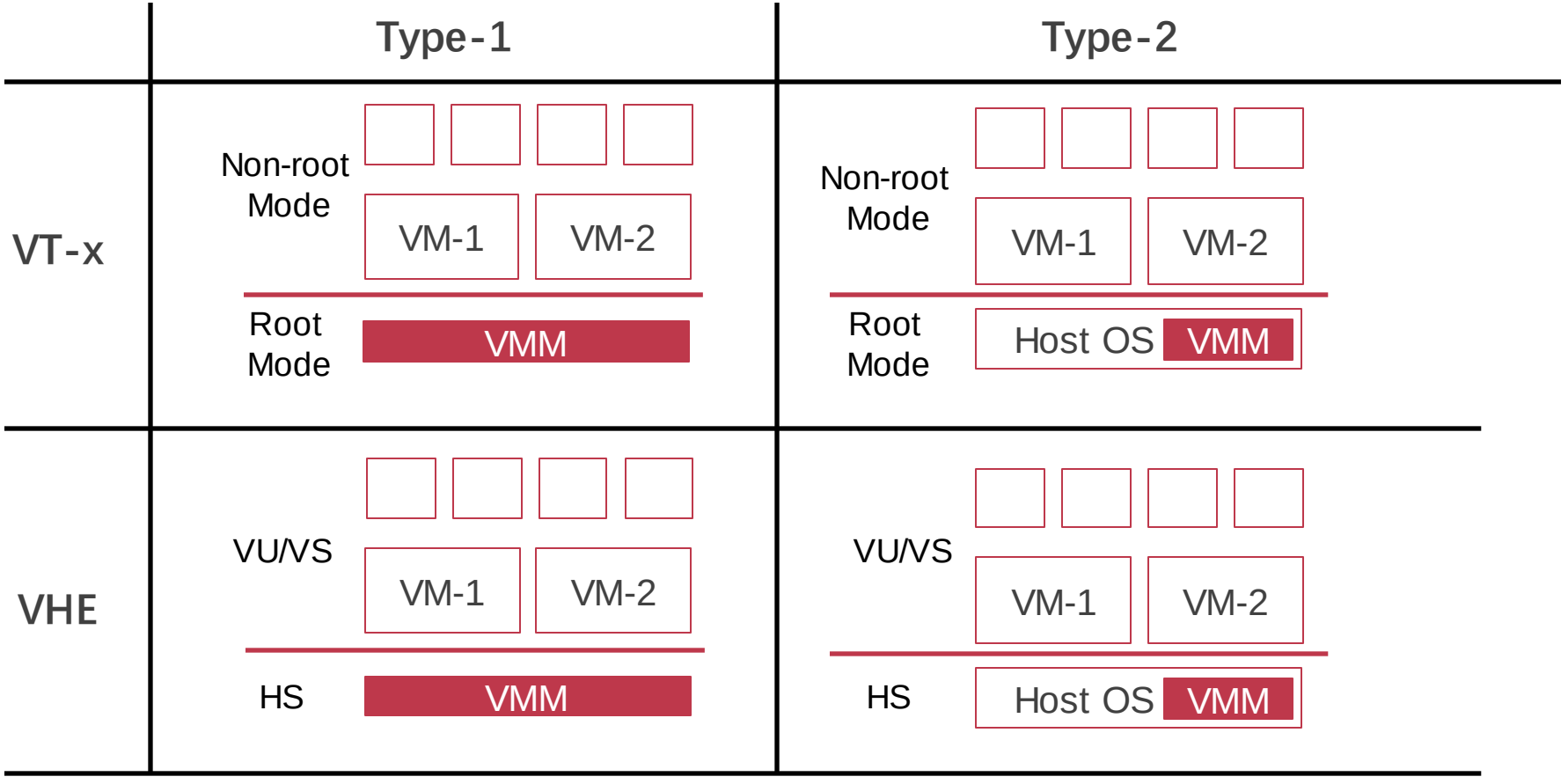
RISC-V中的Type-2 VMM架构



VT-x和RISC-V Hypervisor对比

	VT-x	RVH
新特权级	Root和Non-root	HS,VS,VU
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RVH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
  - 纯软件实现
  - Guest Instruction -> QEMU TCG Instruction -> Host Instruction

QEMU简介：用户模式

- 用户模式（User mode）
  - 支持模拟的操作系统： Linux/BSD
  - Syscall转换
  - POSIX SIGNAL处理
  - 支持POSIX clone syscall

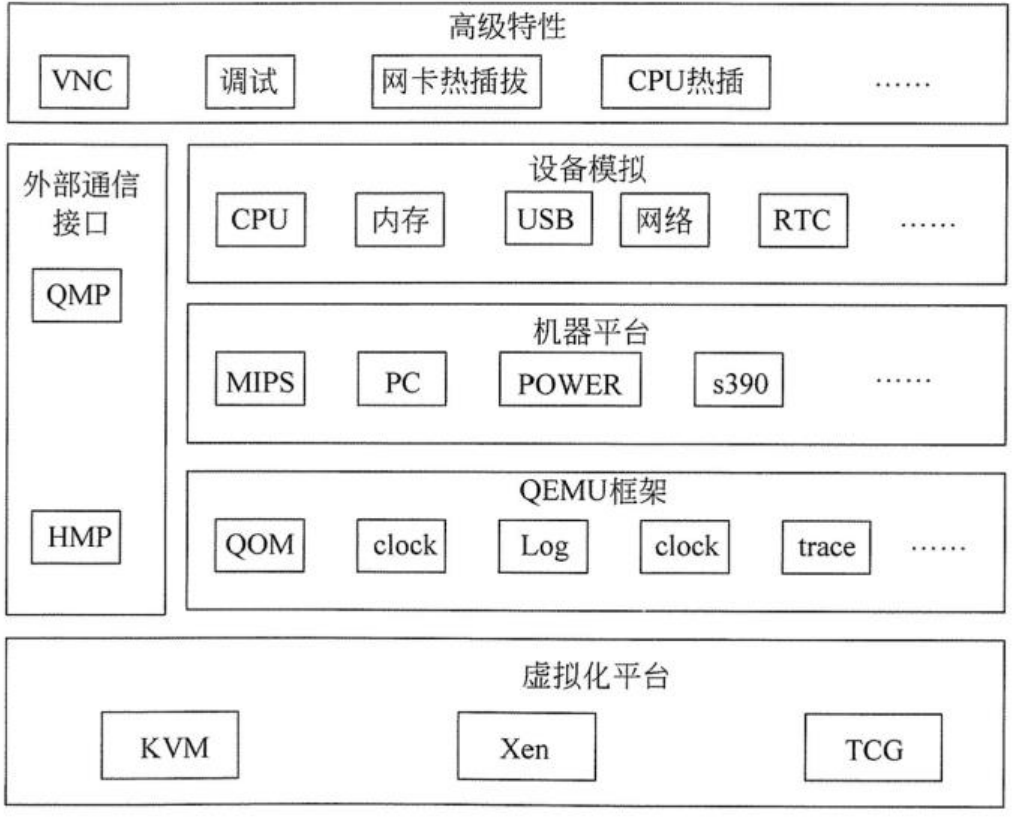
  在当前操作系统上模拟运行目标操作系统的用户程序

QEMU简介：系统模式

- 系统模式（System mode）
  - 模拟运行整个系统
  - 可以配合KVM/Xen等Hypervisor使用， QEMU提供IO设备虚拟化
  - 纯软件模拟时效率低下（指令集翻译）

QEMU基础架构

- CPU模拟器
- 设备模拟器
- 调试器
- 用户调用接口
- QMP
  - QEMU Monitor Protocol
- HMP
  - Human Monitor Protocol



QEMU发展历史

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1亿多美元收购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

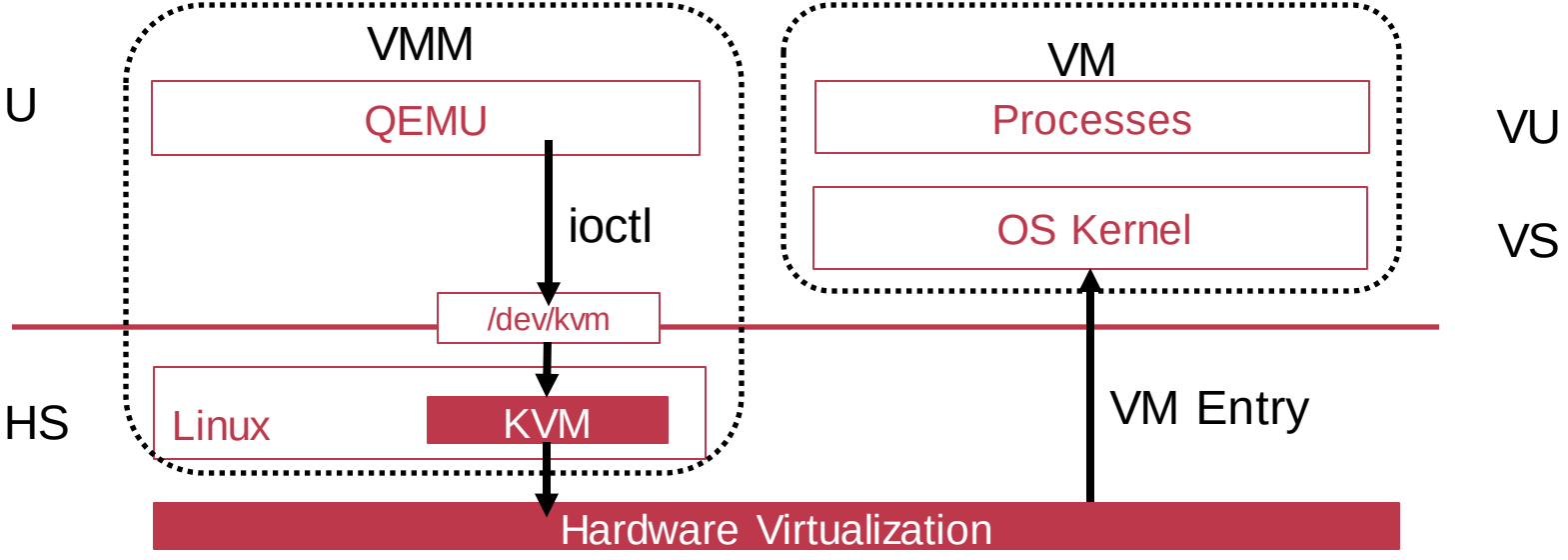
QEMU使用KVM的用户态接口

- **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

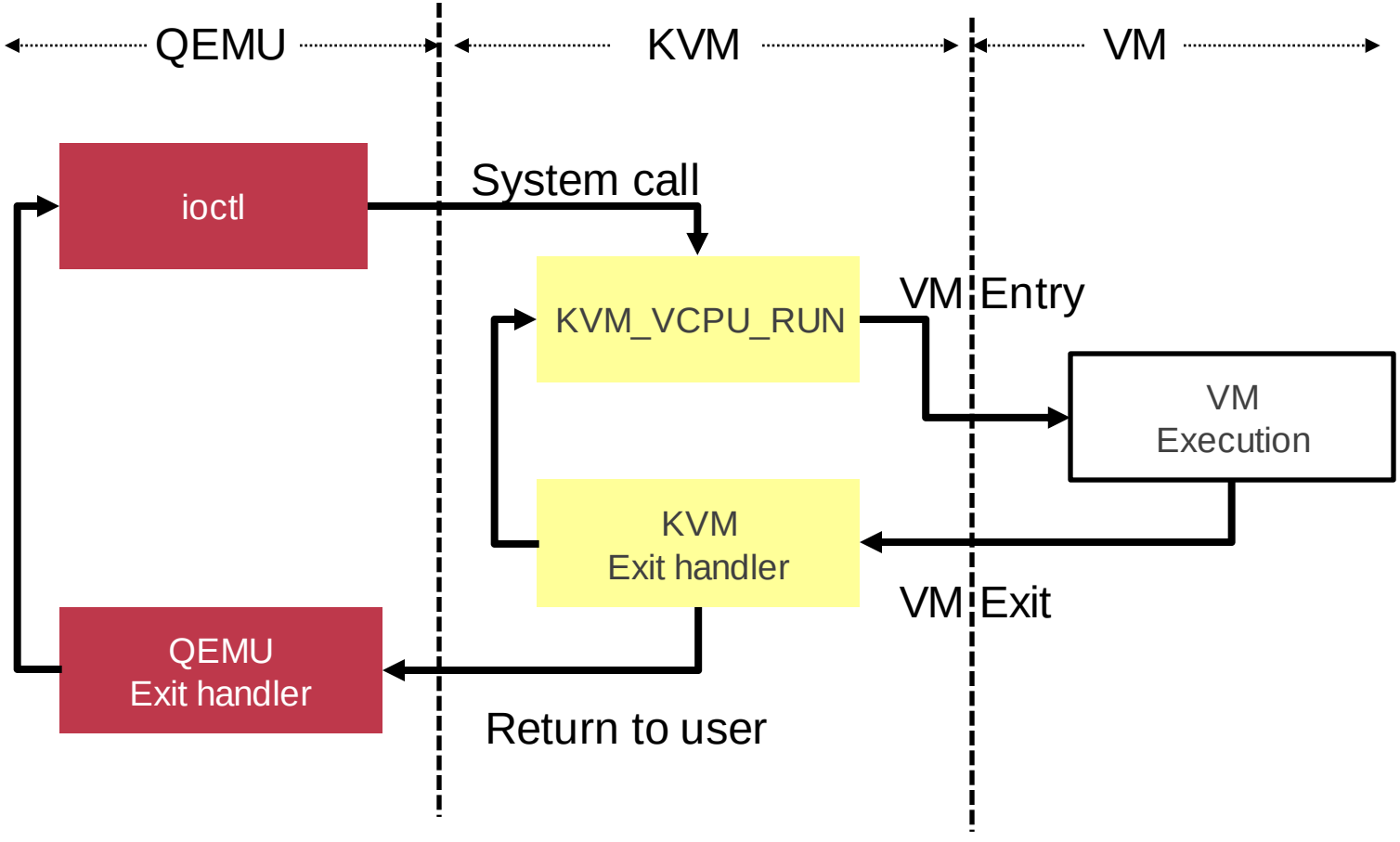
```
open("/dev/kvm")
ioctl(KVM_CREATE_VM)
ioctl(KVM_CREATE_VCPU)
while (true) {
      ioctl(KVM_RUN)
      exit_reason = get_exit_reason();
      switch (exit_reason) {
        case KVM_EXIT_IO:  /* ... */
          break;
        case KVM_EXIT_MMIO: /* ... */
          break;
      }
}
```

ioctl(KVM_RUN)时发生了什么

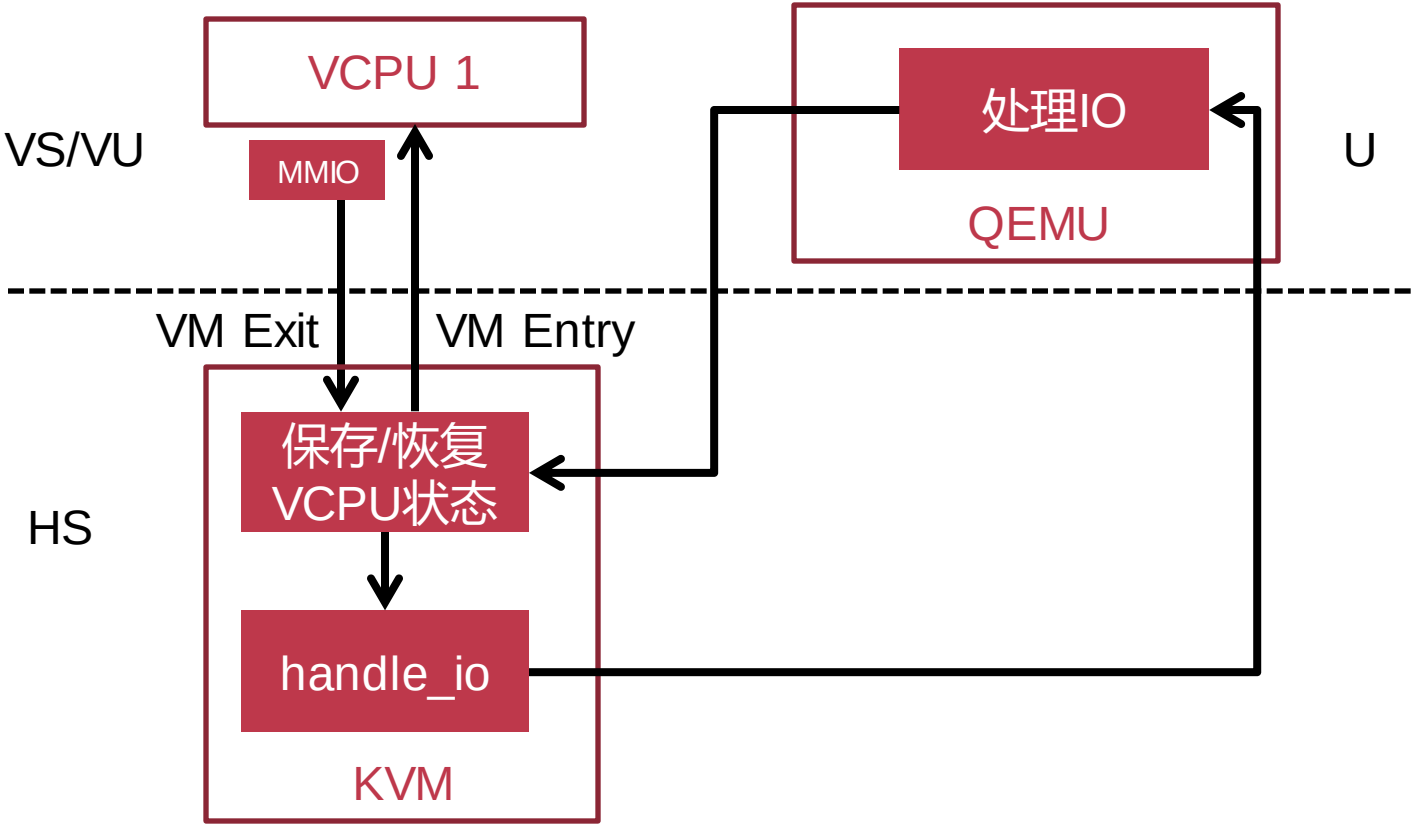
- **x86中**
  - KVM加载vCPU对应的VMCS，VMLAUNCH/VMRESUME进入Non-root模式
- **RISC-V中**
  - KVM主动加载VCPU对应的所有状态，使用eret指令进入虚拟机，开始执行



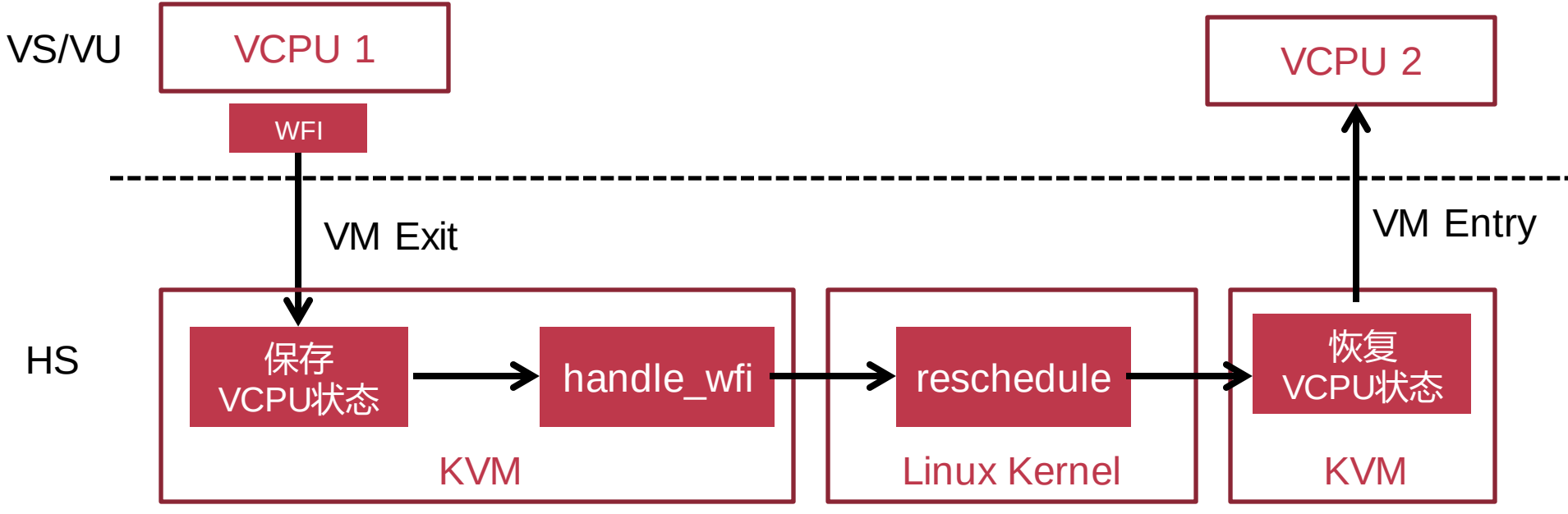
QEMU/KVM的流程


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

例：I/O指令VM Exit的处理流程



例：WFI指令VM Exit的处理流程

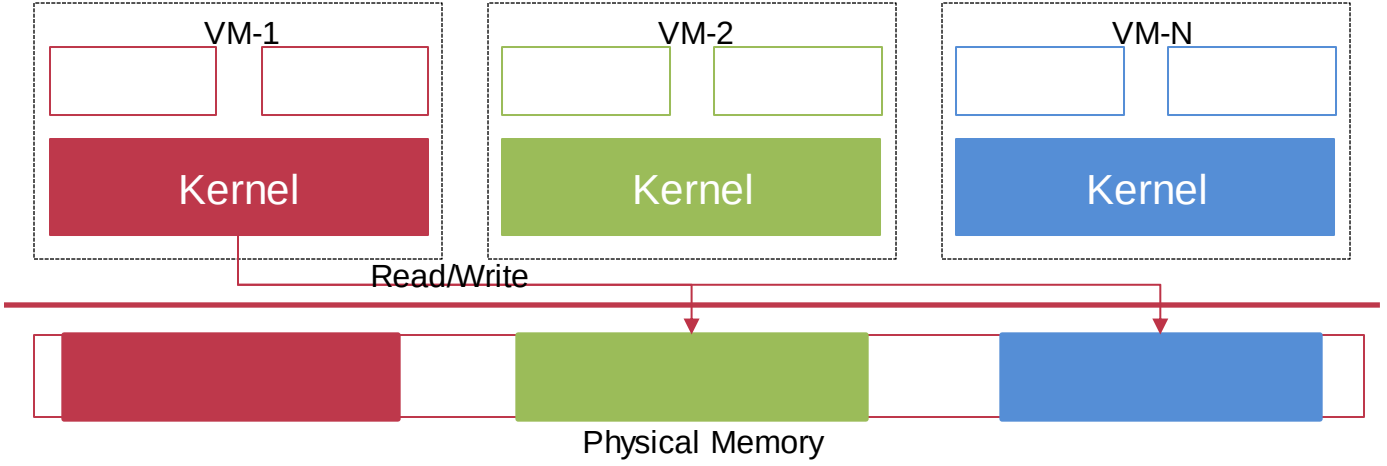


Memory Virtualization

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**



内存虚拟化的目标

- **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无法访问其他的内存

内存虚拟化

- 客户机操作系统通过进程页表将虚拟地址映射到虚拟机定义的“物理”地址
- 而 VMM 进一步将客户机物理地址映射到底层真正的物理地址
- 两大挑战
  - 地址映射
  - 访存截获

三种地址

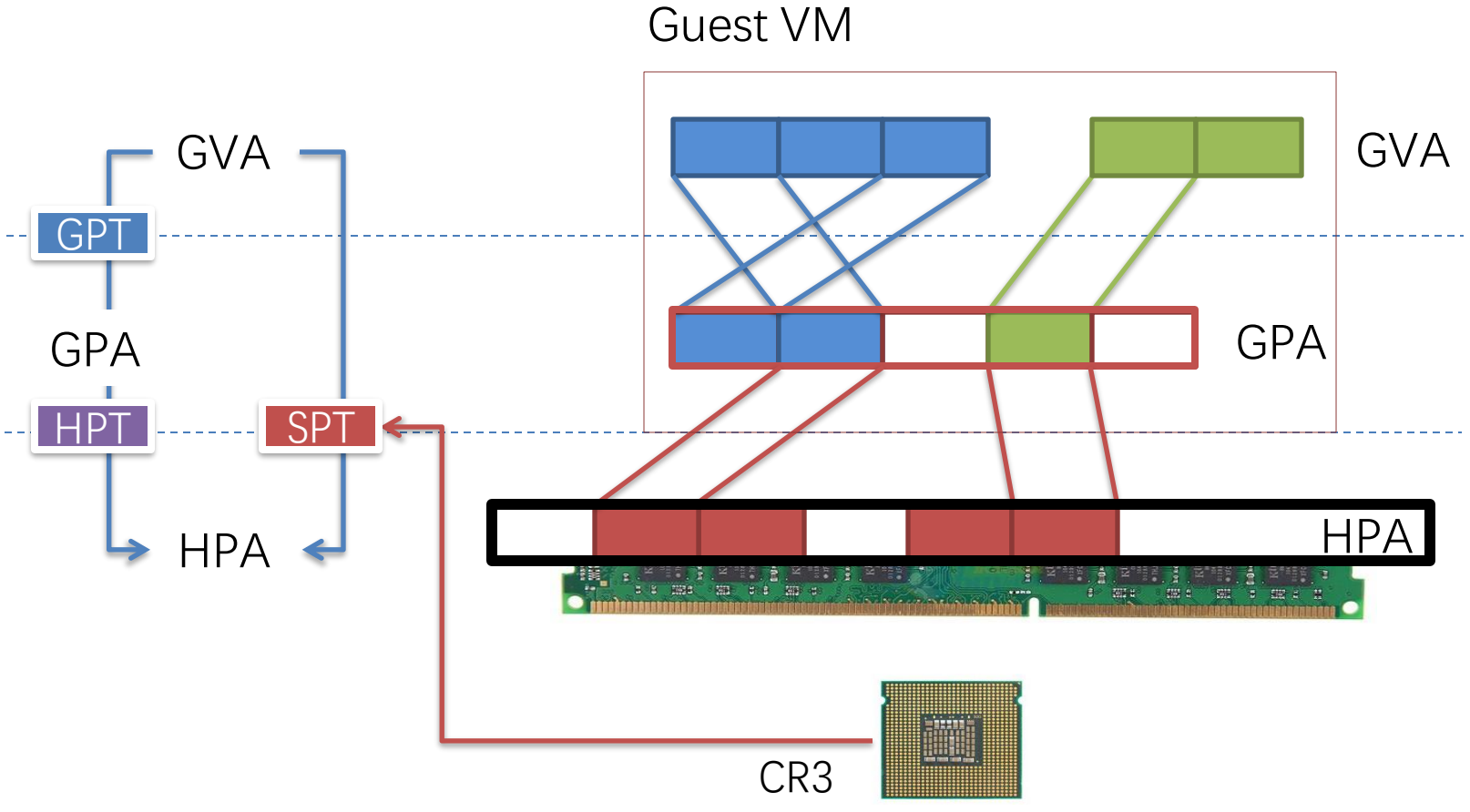
- **客户虚拟地址(Guest Virtual Address, GVA)**
  - 虚拟机内进程使用的虚拟地址
- **客户物理地址(Guest Physical Address, GPA)**
  - 虚拟机内使用的“假”物理地址
- **主机物理地址(Host Physical Address, HPA)**
  - 真实寻址的物理地址
  - GPA需要翻译成HPA才能访存

VMM管理

怎么实现内存虚拟化?

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

1、影子页表



2个页表 -> 1个页表

1. VMM intercepts guest OS setting the virtual CR3 (SATP)
2. VMM iterates over the guest page table, constructs a corresponding shadow page table
3. In shadow PT, every guest physical address is translated into host physical address
4. Finally, VMM loads the host physical address of the shadow page table

Shadow Page Table 设置

```
set_cr3 (guest_page_table):
    for GVA in 0 to 2^20
        if guest_page_table[GVA] & PTE_P:
            GPA = guest_page_table[GVA] >> 12
            HPA = host_page_table[GPA] >> 12
            shadow_page_table[GVA] = (HPA<<12)|PTE_P
        else
            shadow_page_table[GVA] = 0
     CR3 = PHYSICAL_ADDR(shadow_page_table)
```

Guest OS修改页表，如何生效?

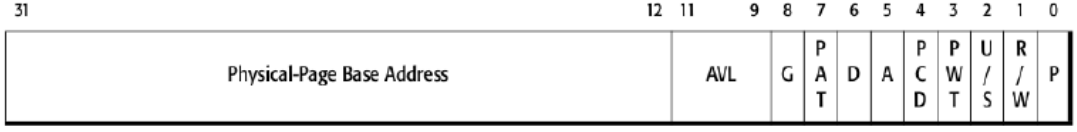
- **Real hardware would start using the new page table's mappings**
  - Virtual machine monitor has a separate shadow page table
- **Goal:**
  - VMM needs to intercept when guest OS modifies page table, update shadow page table accordingly
- **Technique:**
  - Use the read/write bit in the PTE to mark those pages read-only
  - If guest OS tries to modify them, hardware triggers page fault
  - Page fault handled by VMM: update shadow page table & restart guest

Guest内核如何与Guest应用隔离?

- **How do we selectively allow / deny access to kernel-only pages in guest PT?**
  - Hardware doesn't know about the virtual U/K bit
- **Idea:**
  - Generate two shadow page tables, one for U, one for K
  - When guest OS switches to U mode, VMM must invoke set_ptp(current, 0)

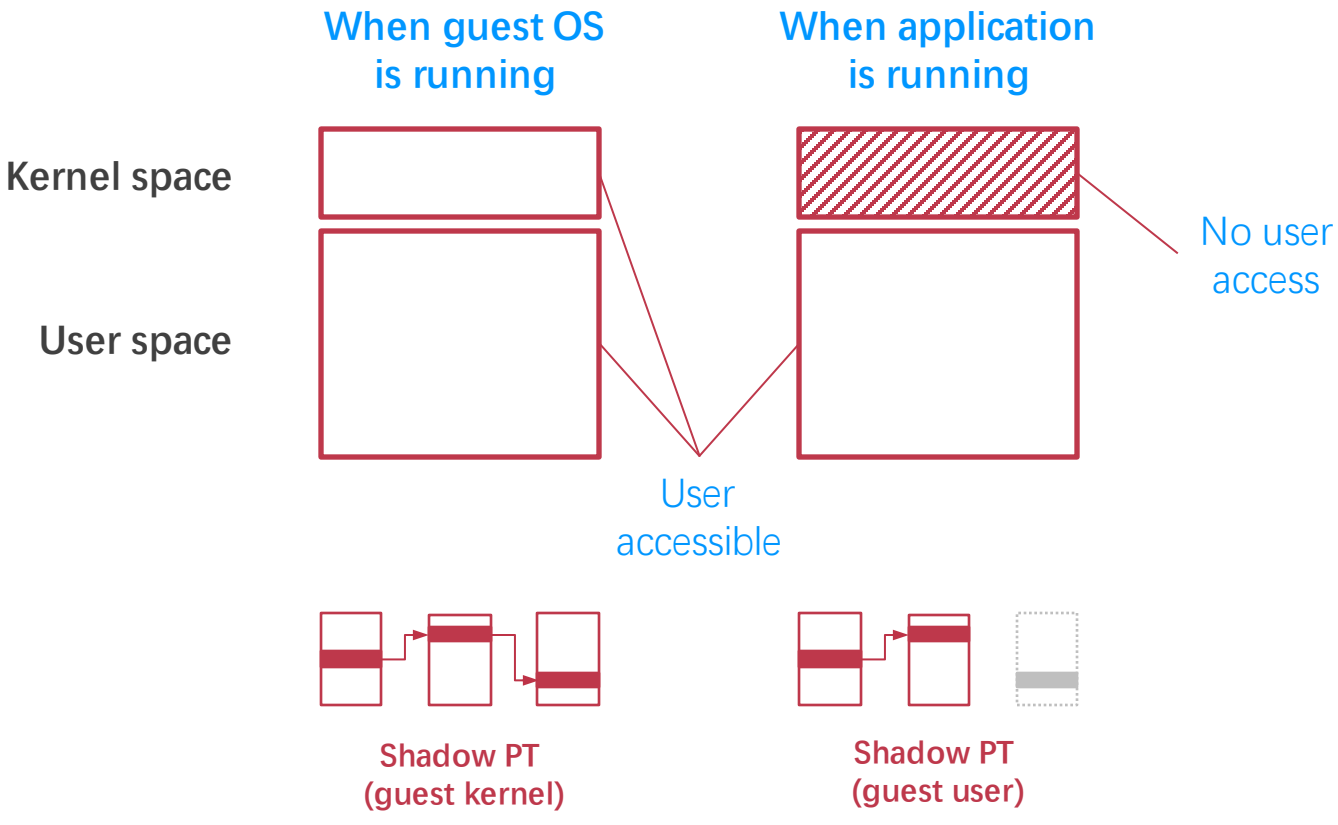
1个页表 -> 2个页表

构建2个不同的Guest Pages




```
set_ptp(guest_pt, kmode):
    for gva in 0 .. 2^20:
        if guest_pt[gva] & PTE_P and
           (kmode or guest_pt[gva] & PTE_U):
            gpa = guest_pt[gva] >> 12
            hpa = host_pt[gpa] >> 12
            shadow_pt[gva] = (hpa << 12) | PTE_P | PTE_U
        else:
            shadow_pt[gva] = 0
    PTP = shadow_pt
```

Two Memory Views of Guest VM



影子页表

- 在VMM中为每一个VM中的进程维护一张影子页表
- VMM纯软件模拟
  - CR3写指令
  - 为页表内容添加写保护， GuestOS试图修改页表内容时触发异常，进入VMM

影子页表

- 每一个VM中的进程对应着两套页表
  - VM看到的页表：页表内容为GPA
  - VMM维护的页表（实际载入到CR3）：页表内容为HPA

- 当VM试图修改页表内容时：
  - 由于写保护，触发异常，控制权移交VMM
  - VMM分配物理页帧，修改影子页表
  - VMM修改VM页表对应内容
  - 异常返回
  - 在VM看来本次修改没有异常发生

影子页表

- 当Guest OS“尚未启用”页表时：
  - 硬件上的MMU是开启状态
  - MMU：GPA->HPA

- VM“载入”页表基地址时：
  - 因执行特权指令发生异常，VMM接管，实际载入cr3的是影子页表
  - VM：载入成功，载入的值为某个GPA
  - VMM：虚拟成功，载入了某个HPA，并维护GPA->HPA关系

- VMM中同步维护关于GPA->HPA的映射关系

影子页表

- 提供GVA->HPA的直接转换
- 为每个进程在VMM中多维护一张影子页表，内存开销大
- 软件模拟进行地址转换，由于写保护将会多次触发异常， 效率较低
- 开发困难， debug难度大

2、Direct Paging (Para-virtualization)

- **Modify the guest OS**
  - No GPA is needed, just GVA and HPA
  - Guest OS directly manages its HPA space
  - Use *hypercall* to let the VMM update the page table
  - The hardware CR3 will point to guest page table
- **VMM will check all the page table operations**
  - The guest page tables are read-only to the guest

2、Direct Paging (Para-virtualization)

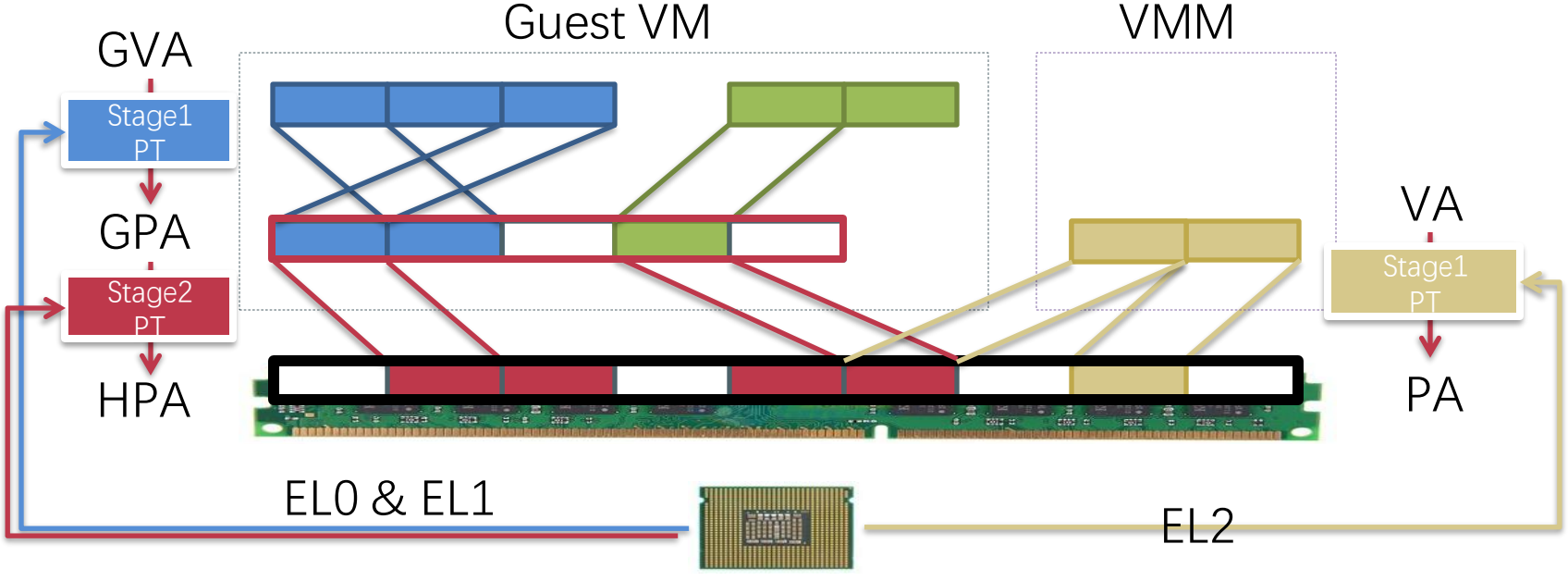
- **Positive**
  - Easy to implement and more clear architecture
  - Better performance: guest can batch to reduce trap
- **Negatives**
  - Not transparent to the guest OS
  - The guest now knows much info, e.g., HPA
    - May use such info to trigger *rowhammer* attacks

3、硬件虚拟化对内存翻译的支持

- **Intel VT-x和RISC-V硬件虚拟化都有对应的内存虚拟化**
  - Intel Extended Page Table (EPT)
  - RISC-V G-stage 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

- 第一阶段页表：虚拟机内虚拟地址翻译（GVA->GPA）
- 第二阶段页表：虚拟机客户物理地址翻译（GPA->HPA）



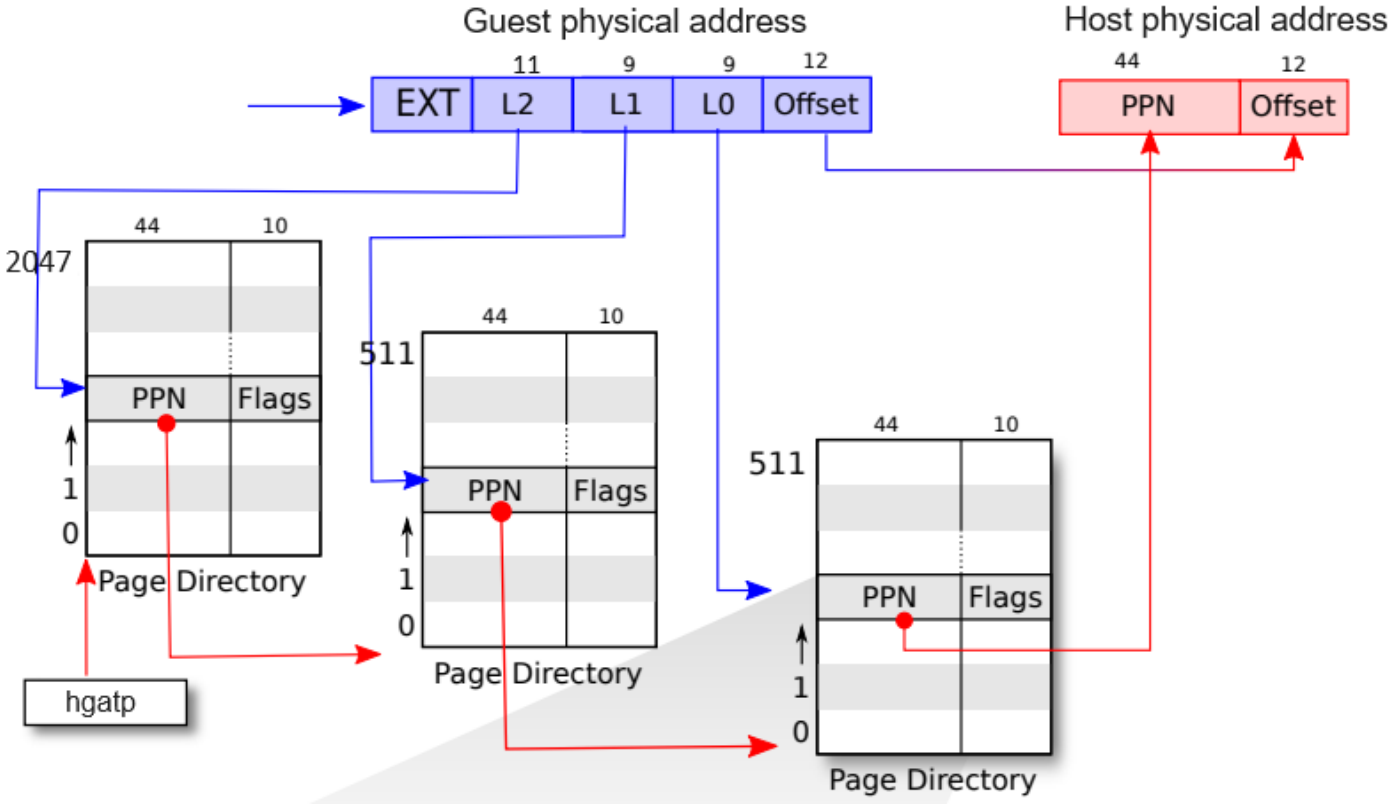
hgatp

- **存储虚拟机第二阶段页表基地址PPN和翻译模式MODE**
  - hgatp 里的 MODE 和 PPN
- **对比第一阶段页表**
  - satp 里的 MODE 和 PPN
- **VMM在调度VM之前需要在hgatp中写入此VM的第二阶段页表基地址和翻译模式**
  - MODE != Bare 启用二级翻译

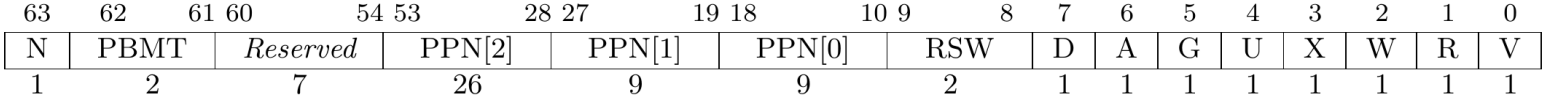
RISC-V第二阶段翻译模式

- **支持的翻译模式**
  - Sv39x4：41 位 GPA
  - Sv48x4：50 位 GPA
  - Sv57x4：59 位 GPA
- **第二阶段页表的根节点是原来 4 倍大小，16KiB**
  - 为了处理 4 倍大的地址空间
- **GPA 零扩展（不是符号扩展）**

第二阶段3级页表为例


Guest physical address
11
9
9
12
EXT
L2
L1
L0
Offset
Host physical address
44
12
PPN
Offset
2047
44
10
PPN
Flags
1
0
Page Directory
511
44
10
PPN
Flags
1
0
Page Directory
511
44
10
PPN
Flags
1
0
Page Directory
hgatp

第二阶段 HGTAP

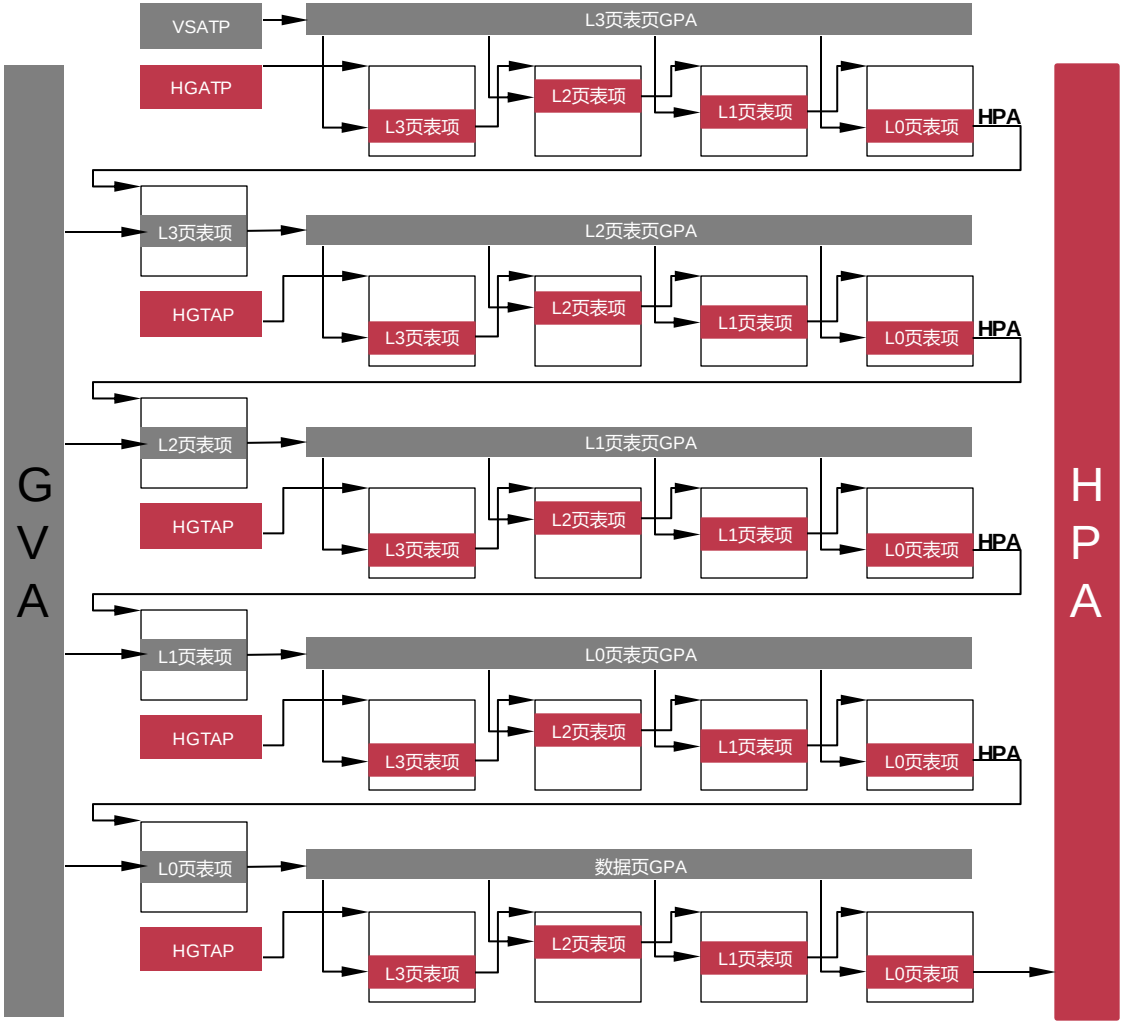


63	62–61	60–54	53–28	27–19	18–10	9–8	7	6	5	4	3	2	1	0
N	PBMT	*Reserved*	PPN[2]	PPN[1]	PPN[0]	RSW	D	A	G	U	X	W	R	V
1	2	7	26	9	9	2	1	1	1	1	1	1	1	1

- **与第一阶段格式大致相同**
  - U = 1，VM 可访问
  - Leaf PTE 的 RWX != 111，限制 VM 物理地址权限

翻译过程

- Sv48x4 + Sv48
- 总共24次内存访问
  - 为什么?
  - 25-1
  - 第一次访问寄存器



TLB：缓存地址翻译结果

- **回顾：TLB不仅可以缓存第一阶段地址翻译结果**
- **TLB也可以第二阶段地址翻译后的结果**
  - 包括第一阶段的翻译结果(GVA->GPA)
  - 包括第二阶段的翻译结果(GPA->HPA)
  - 大大提升GVA->HPA的翻译性能：不需要很多次内存访问
- **切换hgatp时**
  - 理论上应将前一个VM的TLB项全部刷掉

TLB刷新

- **刷TLB相关指令**
  - 清空全部
    - hfence.gvma
  - 清空指定GVA
    - hfence.vvma rs1
  - 清空指定GPA
    - hfence.gvma rs1
- **VMID (Virtual Machine IDentifier)**
  - VMM为不同进程分配VMID，将VMID填写在hgatp.VMID
  - VMID位数由hgatp.VMID可写位数决定
  - 避免刷新上个VM的TLB

如何处理缺页异常

- **两阶段翻译的缺页异常分开处理**
- **第一阶段缺页异常**
  - 直接调用VM的Page fault handler
  - 修改第一阶段页表**不会**引起任何虚拟机下陷
- **第二阶段缺页异常**
  - 虚拟机下陷，直接调用VMM的Page fault handler

第二阶段页表的优缺点

- **优点**
  - VMM实现简单
  - 不需要捕捉Guest Page Table的更新
  - 减少内存开销：每个VM对应一个页表
- **缺点**
  - TLB miss时性能开销较大

扩展页表的问题

- EPT机制的多次访存，会给内存虚拟化带来较大的性能损失
- 假如客户机页表是m级， EPT 有n级
  - 找到每一级客户视页表实际对应的宿主机物理页都需要通过EPT做地址转换
  - 一次虚拟机中的地址转换，最多需要进行 (m+1)*(n+ 1)-1次内存访问
  - 例如，有2级客户机页表与4级 EPT，那么，访存上限(每次 TLB 都未命中)就是14 次。
- 优化方法
  - 增大 EPT TLB
  - 增大 VMM 的页面大小