Lecture21 Memory Virtualization

1. 内存虚拟化

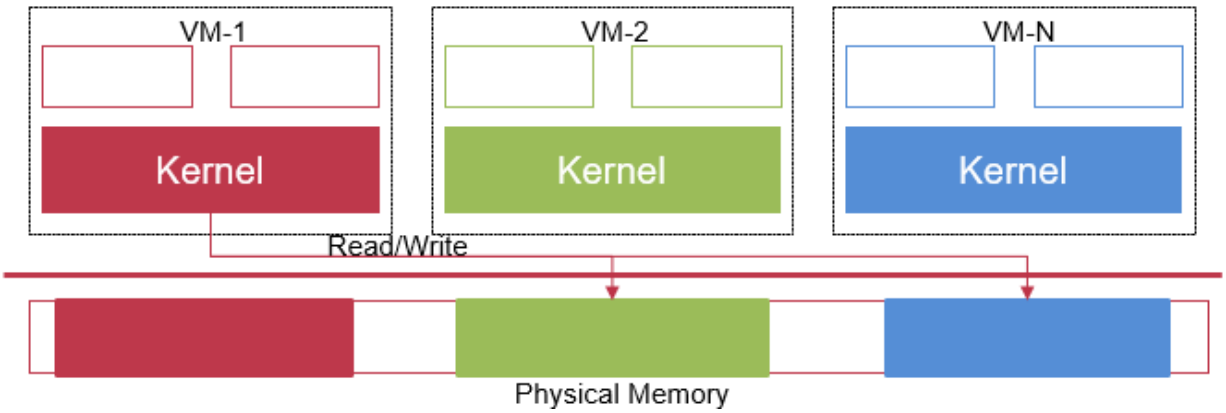
> 依托于CPU虚拟化

为什么需要内存虚拟化?

> OS以为自己独占机器的全部物理内存

- 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

> 正确性：相当于进程看到了全部的物理地址，可能会修改其它"进程"的数据
>
> 安全性：可能会被其它虚拟机访问到本虚拟机的数据



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

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

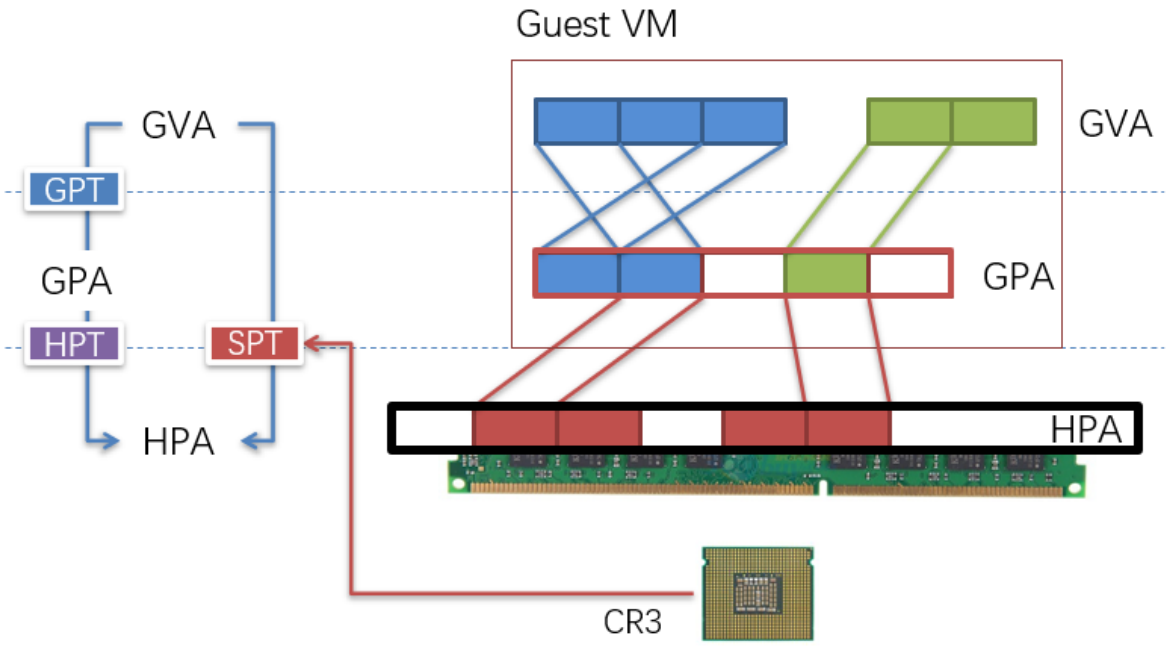
VMM管理

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

方法1：影子页表

> Shadow Page Table



> 硬件限制，只能由一套翻译机制
>
> **SPT：直接将GVA -> HPA**

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)
```

> VMM维护了两个数据结构`guest_page_table`和`host_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

> Trap住对页表进行写操作的指令（将SPT所在的页标记为read-only）

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

> U-SPT和K-SPT

方法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

> Guest直接管页表会带来安全风险，因此不能直接写CR3和页表，必须通过VMM来执行这种操作（hypercall）
>
> 可以通过batch的思想去优化写入的性能

- **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

方法3：硬件虚拟化对内存翻译的支持

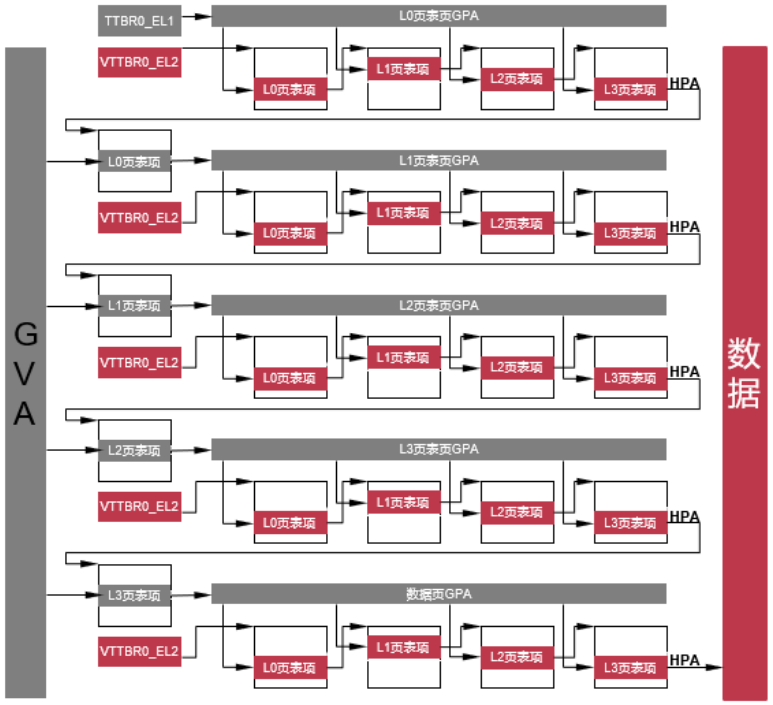
- **Intel VT-x和ARM硬件虚拟化都有对应的内存虚拟化**
  - Intel Extended Page Table (EPT)
  - ARM Stage-2 Page Table (第二阶段页表)
- **新的页表**
  - 将GPA翻译成HPA
  - 此表被VMM直接控制
  - 每一个VM有一个对应的页表

> 影子页表中每一个进程一个SPT，因为创建进程时会创建页表而SPT：GVA->HPA
>
> 本方法中的页表：GPA->HPA，所以每个VM一个这种页表

翻译的过程

翻译过程

- 总共24次内存访问
  - 为什么?
  - 25-1
  - 读TTBR0_EL1无需内存访问



TLB：缓存地址翻译的结果

- 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24次内存访问
- 切换VTTBR_EL2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
    - TLBI VMALLS12E1IS
  - 清空指定GVA
    - TLBI VAE1IS
  - 清空指定GPA
    - TLBI IPAS2E1IS
- **VMID (Virtual Machine IDentifier)**
  - VMM为不同进程分配8/16 VMID，将VMID填写在VTTBR_EL2的高8/16位
  - VMID位数由VTCR_EL2的第19位（VS位）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

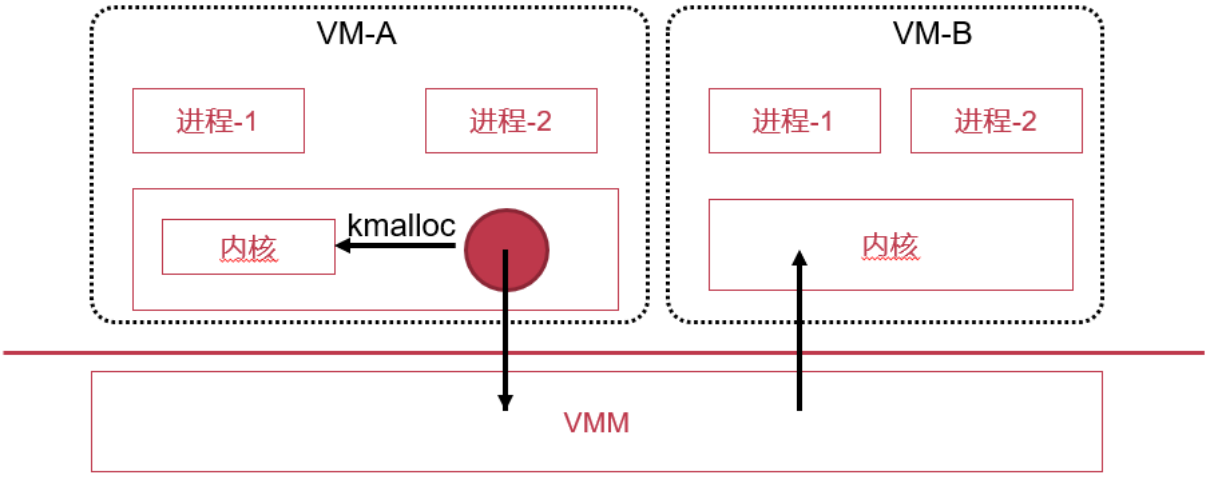
第二阶段页表的优缺点

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

如何实现虚拟机级别的内存换页?

- **具体场景：将虚拟机A的128MB内存转移到虚拟机B中**
  - 虚拟机A对内存的使用较少
  - 虚拟机B对内存需求较大
- **问题**
  - VMM无法识别虚拟机内存的语义
  - 两层内存换页机制
    - VM与VMM的换页机制可能彼此冲突，造成开销。

内存气球机制



> VMM调upcall，此时swap发生在VM的kmalloc中
>
> 红色的圈圈是一个驱动

- 缺点：太慢了
- 优点：修改了少量的代码，解决了两层swap冲突的问题