CSE后两节课

虚拟机，虚拟CPU、内存与设备

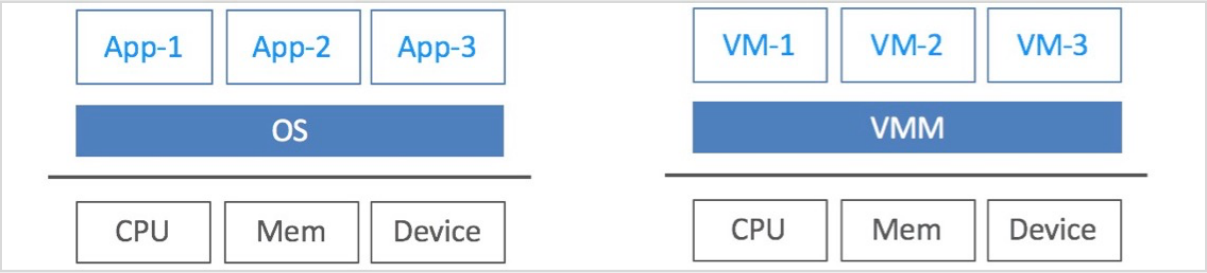
虚拟机

每一个virtual machine都是一个guest，VMM（virtual machine monitor）相当于kernel，VMM被称作host

应用程序远比虚拟化所需要信任的代码行数多
consolidation：在一个机器上跑多个系统
isolation：container模式，fault tolerant
maintenance：安装、backup支持、复制、转移
security：更安全

- 系统和虚拟机的区别
  memory一部分是load and store读和写
  但没有访问cr3（页表）的权利
  App可以访问cpu的部分功能，但CPU不提供给app CR3和开关中断的指令
  Cpu提供给VMM一切功能

  OS调processor、VMM调VMs



- 虚拟化
  CPU的虚拟化：让每个VM有自己的内核和user模式
  Memory的虚拟化：让guest VM有自己的虚拟的MMU改页表
  不同VM彼此隔离
  I/O的虚拟化：每个VM有自己的虚拟化设备

CPU的虚拟化

传统CPU：分成time-slice
为每个process schedule

- 把os当作一个app在os上运行：
  第一行指令：cli（关中断）kernel指令，无法在user mode运行

类似的还有改cr3等，都产生矛盾：不同的VM要改的内容不一样，产生问题

解决：

trap and emulate模仿

在user mode运行mv $100 CR3的指令 会产生**trap**，trap到kernel 态，运行host os，运行一段代码去**emulate**模拟mv $100 CR3的事，然后有一些代码做检查：看100个是不是合法的、安全检查，把页表哦生成新的页表，让真正的CR3指向新的页表

运行cli时，触发trap，host OS 改变IF（interrupt flag） bit进行模拟，把IF bit清了就关了中断。虚拟机的IF bit是在内存中的，当给虚拟机送中断，看是否置了内存中的虚拟机的IFbit
deliver中断的过程类似于signal handler

有17条指令的行为不能被trap，如：
popf：从一个stack中取一个word，放进flag中，iF flag会被覆盖掉
但在用户态下也能跑，不会触发trap，但在用户态下跑，IF flag不会被更新
SGDT, SIDT, SLDT, SMSW, PUSHF, POPF, LAR, LSL, VERR, VERW, POP, PUSH, CALL, JMP, INT n, RET, STR, MOV
如何解决：

1. instruction interpretation解释执行

用软件的方法模拟emulate所有指令,在内存中模拟cpu
用软件写CPU，写一个switch case：

```
void CPU_Run(void)
{
   while (1) {
      inst = Fetch(CPUState.PC);
      CPUState.PC += 4;
      switch (inst) {
      case ADD:
         CPUState.GPR[rd] = GPR[rn] + GPR[rm];
         break;
      …
      case CLI:
         CPU_CLI(); break;
      case STI:
         CPU_STI(); break;
      }
      if (CPUState.IRQ && CPUState.IE) {
         CPUState.IE = 0;
```

```
            CPU_Vector(EXC_INT);
        }
    }
}
void CPU_CLI(void)
{
    CPUState.IE = 0;
}
void CPU_STI(void)
{
    CPUState.IE = 1;
}
void CPU_Vector(int exc)
{
    CPUState.LR = CPUState.PC;
    CPUState.PC = disTab[exc];
}
```

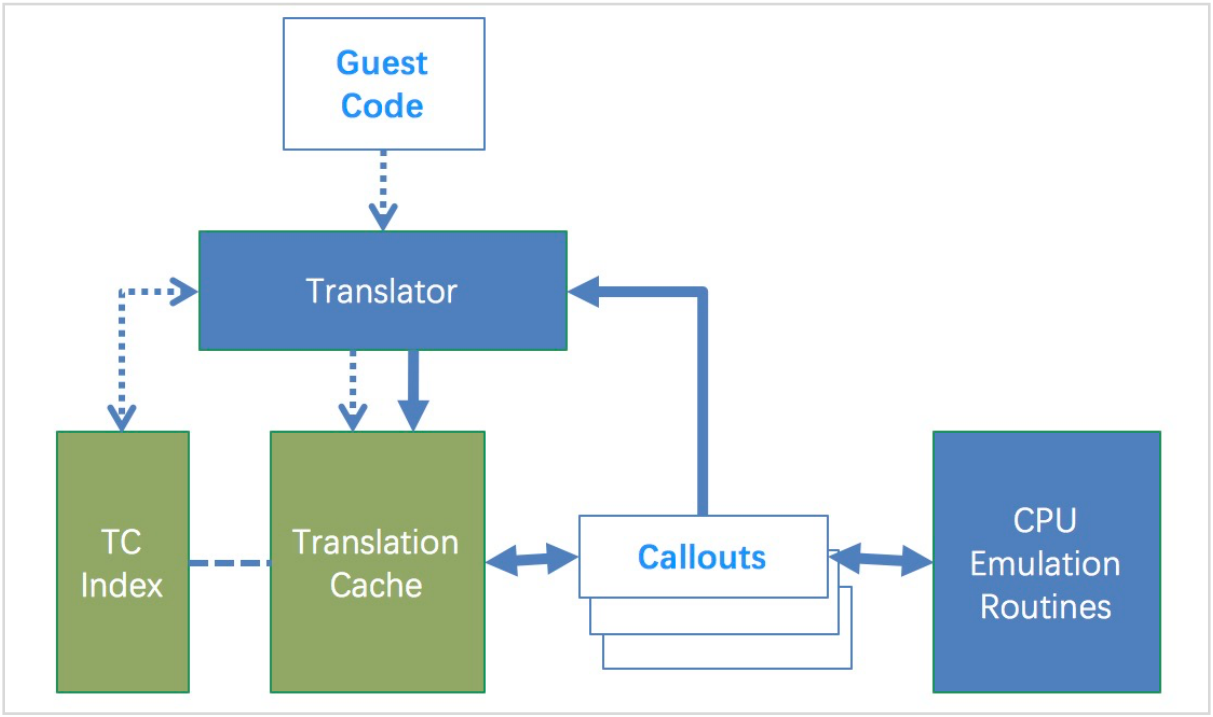
缺点：特别特别慢，一条指令变很多指令

软件Bochs就是解释执行

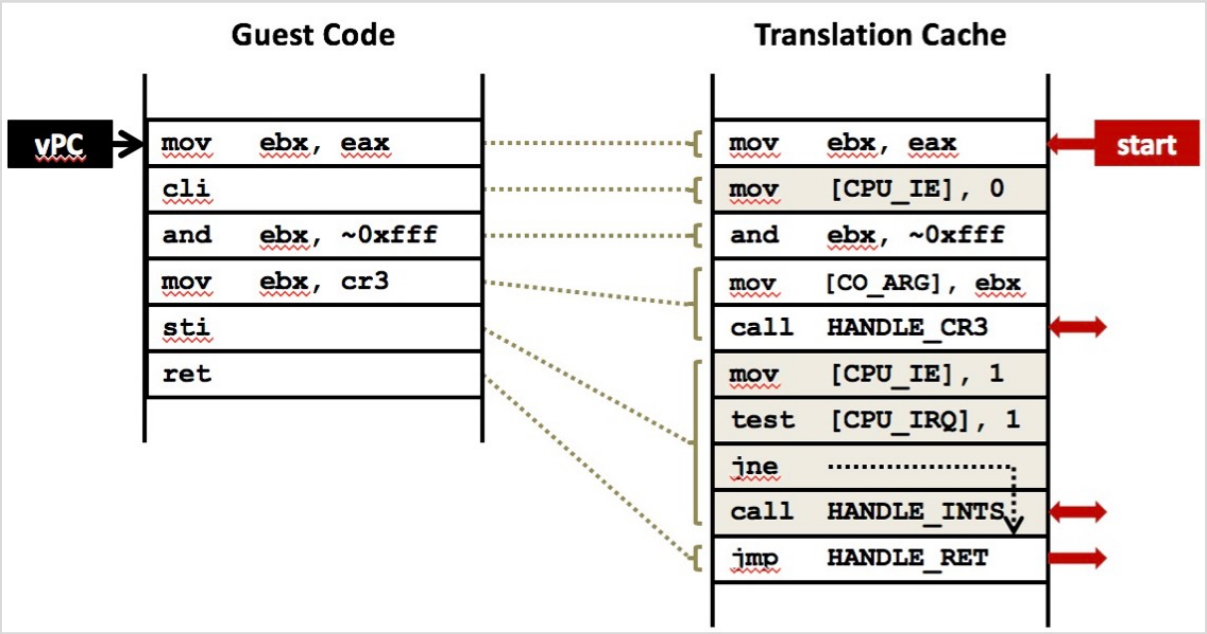
2. binary translator二进制翻译

   指令中如果有需要trap但没trap的，给他翻译出来，变成一个functoin call

   VMWARE是代表



inline优化（把call的代码拉过来）：

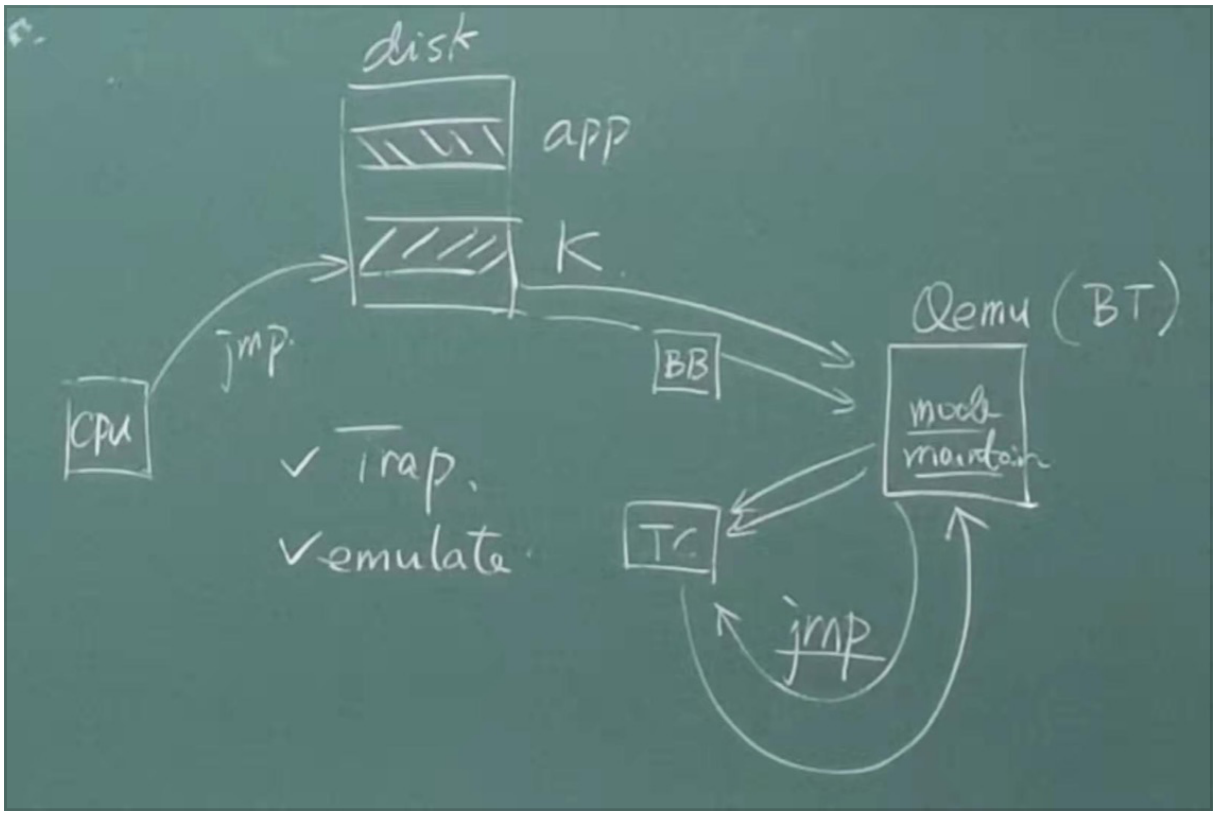



```
void BT_Run(void)
{
   CPUState.PC = _start;
   BT_Continue();
}

void BT_Continue(void)
{
   void *tcpc;

   tcpc = BTFindBB(CPUState.PC);

   if (!tcpc) {
      tcpc = BTTranslate(CPUState.PC);
   }

   RestoreRegsAndJump(tcpc);
}
void *BTTranslate(uint32 pc)
{
   void *start = TCTop;
   uint32 TCPC = pc;

   while (1) {
```

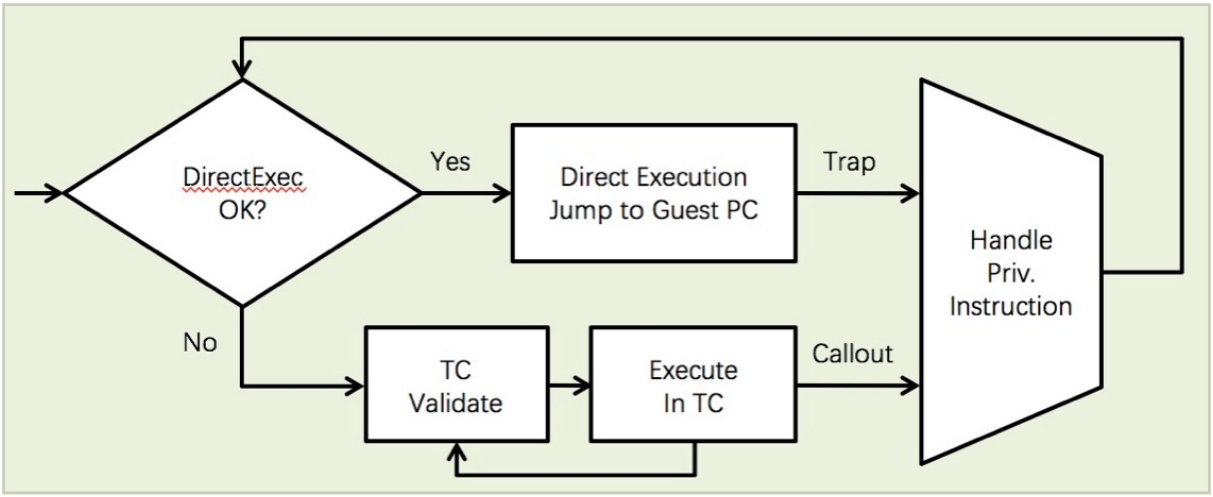
```
      inst = Fetch(TCPC);
      TCPC += 4;

      if (IsPrivileged(inst)) {
         EmitCallout();
      } else if (IsControlFlow(inst)) {
         EmitEndBB();
         break;
      } else {
         /* ident translation */
         EmitInst(inst);
      }
   }

   return start;
}
void BT_CalloutSTI (BTSavedRegs regs)
{
   CPUState.PC = BTFindPC(regs.tcpc);
   CPUState.GPR[] = regs.GPR[];

   CPU_STI();

   CPUState.PC += 4;

   if (CPUState.IRQ
         && CPUState.IE) {
      CPUVector();
      BT_Continue();
      /* NOT_REACHED */
   }

   return;
}
```

kernel调iret进user， user靠中断、syscall、int0x80都是

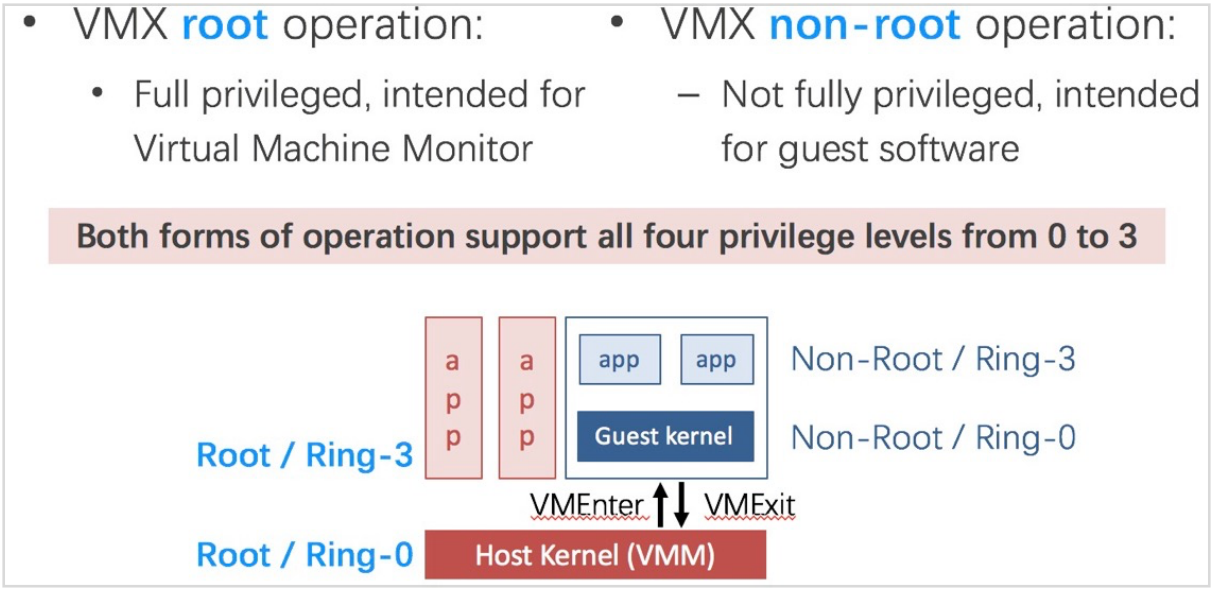
K还是u在Qemu中的mode里看



BT的好处和缺点： interrupt是有问题的，虚拟机大部分时间执行TC（translated code），执行完一个TC，返回在下次跳过去之前检查一下是否有interrupt，导致问题：执行一个basic block的时候，程序永远不会被打断掉。



3. para-virtualization
   修改os的源代码，不让他产生17条指令，把本来需要trap的指令都变成function call，hypercall（是VMMS 提供的相当于system call的接口）
4. hardware supported CPU virtualizaiton
   硬件实现CPU的虚拟化
   VMM要做的： switch case
   处理完kernel指令之后再回去



memory virtualization

3种地址术语：

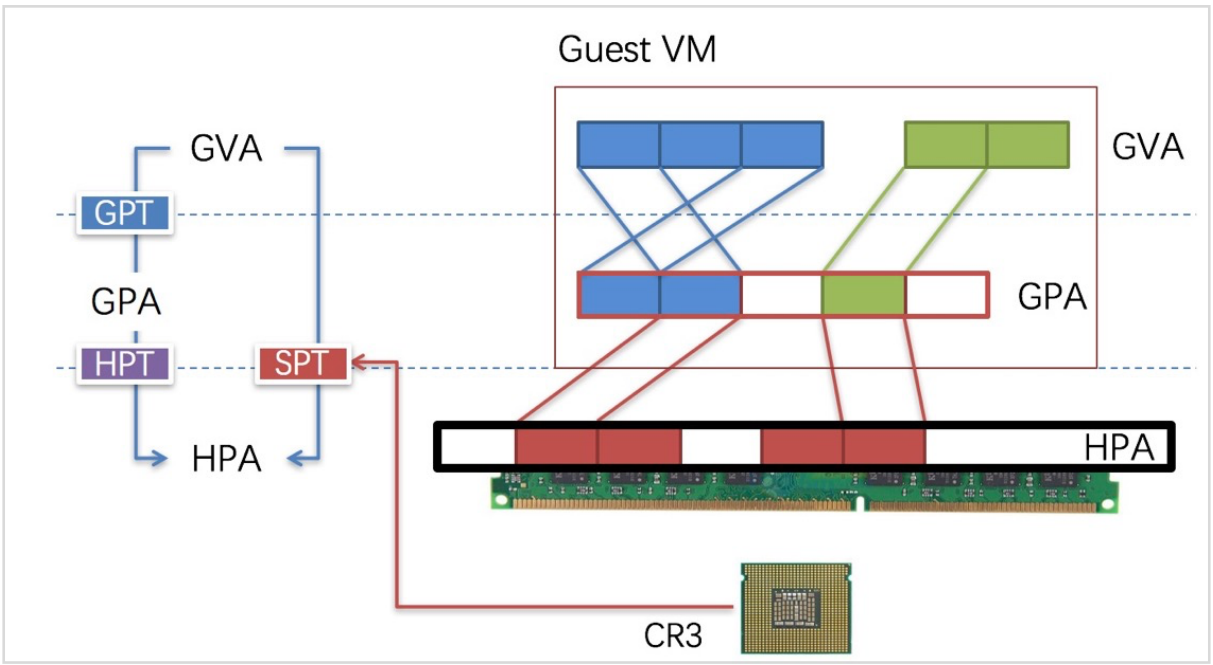
GVA（guest virtual memory）

GPA（guest physical memory）

HPA （Host physical memory）

不能让CR3指向guest page table：VM的一个进程可能访问HPA的0-1Gb，可能并不属于虚拟机

1. shadow pages



Guest看到的内容是GVA，他需要读取GVA映射到GPA上来读取HPA，这个过程是两个页表，但是CR3只有一个，把这两个页表映射成一个页表SPT

- 当guest OS 更新自己的页表时：

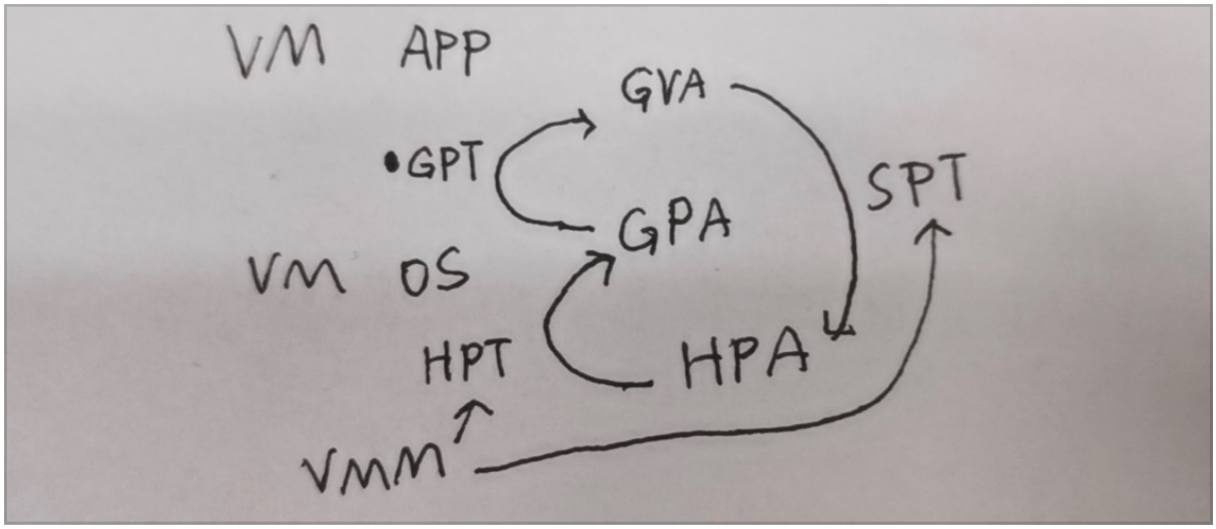
无效，因为CR3现在指向SPT

因此需要对SPT与GPT同步

解决方案：

Guest改了页表是没有用的，不能接受一个新一个旧，所以需要在guest改的时候 也更新一下：

知道被改了的方法：readonly，一旦去写就是pagefault，就知道你要写页表 直接修改了但不能直接改成readonly，否则会被guest看到，因此在shadow page table把gust page table的页所在的page table改成read only，guest自己却不知道被改成了readonly，当guest OS要修改页表的时候，出现page fault，VMM handle这个page fault，更新SPT



- 两张页表合二为一，变成GVA到HPA的映射：

  1.VMM拦截 guest OS 设置虚拟CR3的地址

  2.VMM遍历guest page table，建立shadow page table

  3.在SPT中，每一个GPA被翻译为一个HPA

  4.最后，VMM 加载每一个SPT的HPA地址

```
set_cr3 (guest_page_table):
    for GVA in 0 to 220
        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 = 0
     CR3 = PHYSICAL_ADDR(shadow_page_table
```

SPT是每个application有一个

GPT是每个application有一个

HPT是每个VM有一个

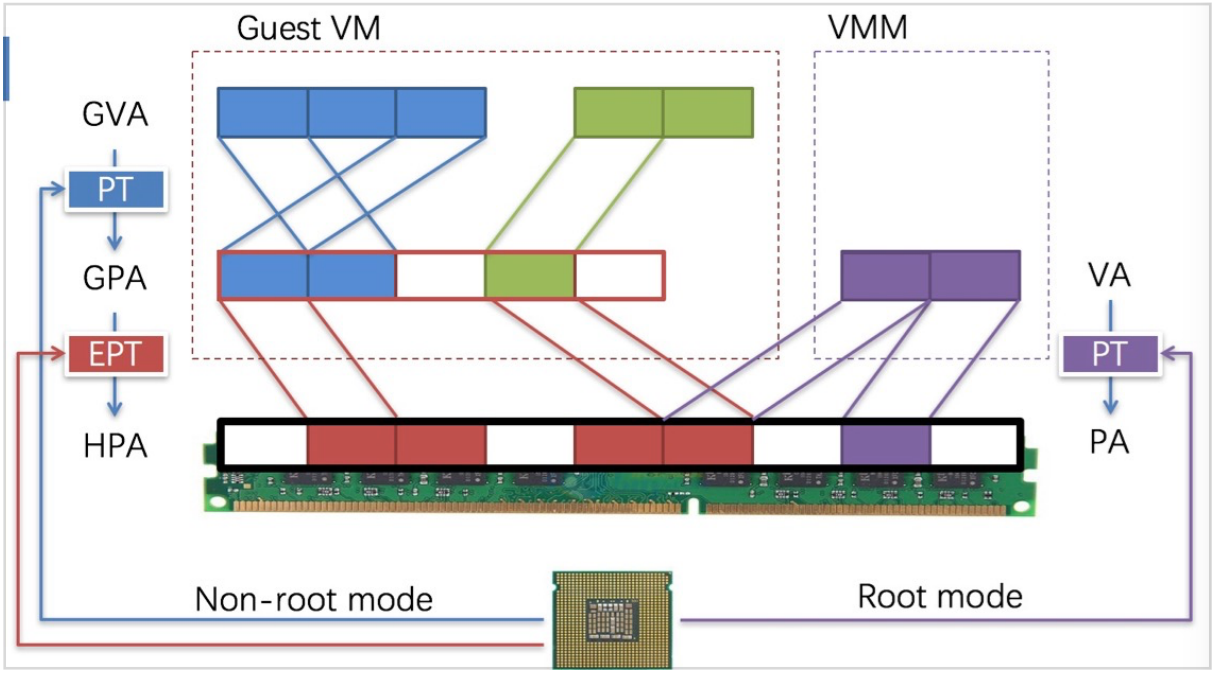
例如：有10个VM，每个VM有10个APP
那么有100个SPT，100个GPT，10个HPT

- 问题Guest APP想访问guest kernel，此时在user态，app理论上可以直接访问，如何隔离开?
  guest APP 访问kernel memory怎么办
  怎么允许/拒绝 kernel only pages：
  解决: 把SPT分为两张表，一个给user 一个给kernel

  例如：有10个VM，每个VM有10个APP
  那么有200个SPT，100个GPT，10个HPT

2. Direct Paging (Para-virtualization)
   不需要GPA，只需要GVA 和HPA
   guest os直接操控HPA
   用hypercall让VMM更新页表
   CR3指向GPT
   好处：容易操作、更好的performance
   坏处：guest OS知道了更多的信息

3. hardware supported memory visualization
   intel：extended page table
   nested page table



硬件：
CPU先看CR3，得到第一级页表，从某个entry得到第二级页表，再得到第三极页表，四次访存是GPA，一个GPA需要四次访存，现在需要20次访存4*4+4（最坏情况）

两个页表，访存最坏的情况要 访guest访问CR3拿到GVA，guset page table，5次访存

hypervisor可信吗

Hypervisor存在bug
在漏洞中：hypervisor漏洞与熟悉的漏洞有什么区别

内存、CPU虚拟化
绝大部分的攻击都是Dos攻击： 让服务器崩掉

- 方法1:嵌套虚拟化
  hypervisor降权，变成guest mode，当一个运行在root mode下变成guest root中，自己并不知道他做的事会引发exit（），传入cloudvisor，当出现VM exit的时候运行到cloud visor，cloud visor把vm exit发给hypervisor，

  原来：hypervisor有自己的EPT（英特尔叫EPT，又叫HPT），是真实的页表，下面就是EPT。只需要控制EPT即可。

  现在以为是HPA，其实是GPA，cloud visor让hypervisor以为自己有hPA，这一层gPA是cloudvisor为其定制的页表，虚拟化。防止hypervisor读到vm的内存，cloudvisor可以限制hypervisor对应的EPT所能访问的HPA。

  Cloudvisor对hypervisor一一映射，但VM的内存的部分被删除。页表因此被用于隔离。

- 让内核的高权限区域只有一个地方可以改页表

  ### enclave and tee
  对密文的数据放在内存里，不解密就能用
  做一个内存加密的安全处理器

- 内存加密Counter-Mode Encryption
  Counter 每写一个+1，cpu写两次一样的数据也被认为不一样的

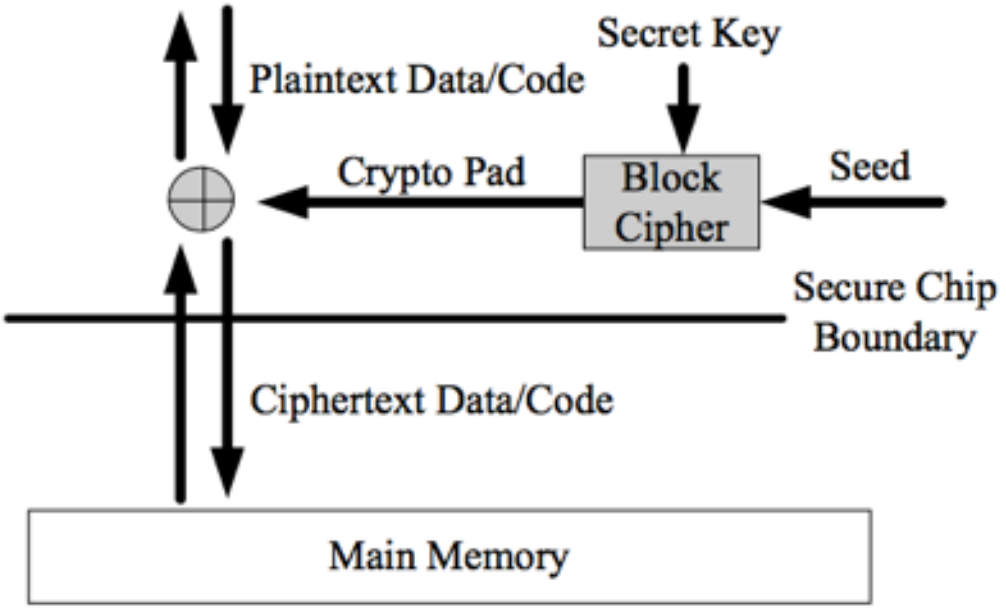


Figure 1. Counter-mode based memory encryption.

replay attack： 安全补丁、拉黑与roll back信息泄漏

- AISE
  The main idea is that instead of encrypt/decrypt the data directly, the processor encrypt counters into pad, and then XOR the pad with data for encryption and decryption. Meanwhile, there is a dedicated cache on-chip for the counters to improvement the performance.

- SGX
  SGX allows part of application code to run in isolation inside an enclave. The enclave region of the main memory is encrypted. The content is only decrypted inside the CPU package using processor specific keys. Thus, even if a malicious adversary has full control over the hardware, it cannot access/modify the enclave. Also, the enclave is protected from other software running in the host, including the OS and hypervisor. In summary, the TCB is the CPU package and the application code running inside and enclave.