**1. 概述**

**1.1. KVM简介**

KVM是一个基于Linux内核的虚拟机，它属于完全虚拟化范畴，从Linux-2.6.20开始被包含在Linux内核中。KVM基于x86硬件虚拟化技术，它的运行要求Intel VT-x或AMD SVM的支持。

一般认为，虚拟机监控的实现模型有两类：监控模型（Hypervisor）和宿主机模型（Host-based）。由于监控模型需要进行处理器调度，还需要实现各种驱动程序，以支撑运行其上的虚拟机，因此实现难度上一般要大于宿主机模型。KVM的实现采用宿主机模型（Host-based），由于KVM是集成在Linux内核中的，因此可以自然地使用Linux内核提供的内存管理、多处理器支持等功能，易于实现，而且还可以随着Linux内核的发展而发展。另外，目前KVM的所有I/O虚拟化工作是借助Qemu完成的，也显著地降低了实现的工作量。以上可以说是KVM的优势所在。

本文仅分析KVM中与Intel VT-x相关的实现，不考虑KVM中与AMD SVM相关的实现，因此有关术语的使用与Intel VT-x保持一致。

**2. 处理器虚拟化**

**2.1. VT-x技术**

我们知道处理器一般存在应用编程接口和系统编程接口。对于x86处理器来说，应用编程接口仅向应用程序暴露了通用寄存器、RFLAGS、RIP和一组非特权指令，而系统编程接口向操作系统暴露了全部的ISA（Instruction Set Architecture）。传统的进程/线程模型也是对处理器的一种虚拟化，但只是对处理器的应用编程接口的虚拟化，而所谓的系统虚拟化（system virtualization）是要实现处理器系统编程接口的虚拟化。从这个角度讲，系统虚拟化与进程/线程模型相比并无本质的区别。

处理器虚拟化的本质是分时共享。实现虚拟化需要两个必要条件，第一是能够读取和恢复处理器的当前状态，第二是有某种机制防止虚拟机对系统全局状态进行修改。

第一个必要条件没有必要一定由硬件来实现，虽然硬件实现可能比软件实现更为简单。例如，x86处理器对多任务，也就是应用编程接口虚拟化，提供了硬件的支持，软件通常只需要执行一条指令，就可以实现任务切换，处理器硬件负责保存当前应用编程接口的状态，并为目标任务恢复应用编程接口的状态。但操作系统并不一定要使用处理器提供的这种虚拟化机制，完全可以使用软件来完成应用接口状态的切换。例如，Linux就没有使用x86处理器提提供多任务机制，完全依赖软件实现任务切换。

第二个必要条件一定要由硬件来实现，通常处理器采用多模式操作（multi-mode operation）来确保这一点。在传统x86处理器上，共有4种模式的操作，也就是常说的4个特权级。虚拟机（这里指进程/线程）通常运行在特权级3上，而虚拟机监控器（这里指操作系统）运行于特权级0上，进程/线程的所有访问全局的操作，如访问共享的操作系统所在的地址空间，访问I/O等等，均会导致异常的发生，被操作系统所截获并处理，使操作系统有机会向进程/线程提供一个虚拟的世界。

系统虚拟化与进程/线程模型相比并无本质的区别。x86处理器完全有机会以较小的代价提供对系统虚拟化的支持，但很可惜Intel没有考虑那么长远。x86的4个特权级对于实现系统虚拟化已经足够了，但传统的x86处理器上，许多特权指令要求必须在特权级0上执行，如LGDT，因此通常操作系统都占用了特权级0，也就没有特权级供虚拟机监控器使用了。为此，许多基于传统x86处理器的虚拟化软件不得不采用ring deprivileging方法，让操作系统运行于特权级1，而由虚拟机监控器使用特权级0。ring deprivileging方法带来了许多问题，包括：ring aliasing、address space compression、nonfaulting accessing to privileged state、adverse impact on guest transitions、interrupt virtualization、access to hidden state等问题，通常将以上问题统称为x86平台的虚拟化漏洞。

ring aliasing问题是指，采用ring deprivileging方法时，由于处理器的CPL保存在CS的低两位，所以操作系统通过执行PUSH CS指令和一条POP EAX指令可以很容易发现其目前不在特权级0上执行，这违背了虚拟化对操作系统透明的原则。

address space compression问题是指，操作系统通常期望能够访问整个4GB线性地址空间，但虚拟机监控器可能也需要占用操作系统的一部分线性地址空间，以便其能够方便地访问操作系统的地址空间。但如果操作系统是运行于特权级1，那么操作系统也同样可以访问虚拟机监控器的存储空间，对虚拟机监控器造成威胁。

nonfaulting accessing to privileged state问题是指，Intel的特权级机制不能确保所有的访问处理器状态的指令在低特权级状态下执行时都产生故障（Fault），这使得操作系统在访问某些处理器状态时虚拟机监控器无法获得控制，也就无法对这些指令进行仿真。例如，IA-32的GDTR, LDTR, IDTR, TR包含了控制处理器状态的指针，对这些寄存器的修改只能在特权级0进行，但IA-32允许在所有的特权级中读取这些寄存器的值。操作系统可以读取这些寄存器的值，如果与真实的计算机上的值不同，操作系统就可以认为自己正运行在虚拟机环境中。

adverse impact on guest transitions问题是指，为加快系统调用的速度，Intel引入了SYSENTER和SYSEXIT指令，但SYSENTER指令总是将特权级切换到0，且从0以外的特权级执行SYSEXIT指令将导致故障。因此，在采用ring deprivileging方法实现虚拟化时，SYSENTER和SYSEXIT指令总是先陷入到虚拟机监控器，经后者仿真后再交给操作系统，这使系统调用的速度减慢。

interrupt virtualization问题是指，IA-32使用EFLAGS.IF位来控制中断的屏蔽，修改IF位需要在CPL<=IOPL的情况下进行，否则将产生故障。操作系统可能需要频繁地修改IF位，会频繁地导致虚拟机监控器的陷入，影响系统性能。而且，有些情况下，虚拟机监控器需要向虚拟机注入事件，但如果虚拟机正处于中断屏蔽状态，虚拟机监控器就必须等待，直到虚拟机打开中断。虚拟机监控器为了及时得知虚拟机已打开中断，也必须截获操作系统对EFLAGS.IF位的修改。

access to hidden state问题是指，IA-32处理器的某些状态，例如段描述符高速缓存，是无法通过指令访问的。当虚拟机切换时，IA-32没有提供保存和恢复段描述符高速缓存的手段。也就是说，上文所述的实现虚拟化的第一个必要条件，能够读取和恢复处理器的当前状态，并不完全具备。

总之，虽然采用ring deprivileging方法可能实现系统虚拟化，但具有很多缺陷，且软件上比较复杂。为此，Intel提出了VT-x技术来解决系统虚拟化问题，其主要思路是增加一个新的比0还高的特权级，通常称之为特权级-1，并在硬件上支持系统编程接口状态的保存和恢复。

首先，VT-x提供了一套称作VMX（Virtual Machine eXtension）的新的工作模式，工作在该模式下的处理器又具有两类操作模式：VMX root operation和VMX non-root operation。通常，虚拟机监控器运行在VMX root operation模式下，即所谓的特权级-1，客户操作系统运行在VMX non-root operation模式下。VMX non-root operation模式仍保留4个特权级，对操作系统来说，VMX non-root operation模式与传统的x86处理器兼容，最大的差别在于当虚拟机执行一些访问全局资源的指令时将导致虚拟机退出操作（VM exit），从而使虚拟机监控器获得控制权，以便对访问全局资源的指令进行模拟。以后，虚拟机监控器可以通过虚拟机进入操作（VM entry）使虚拟机重新获得控制权。

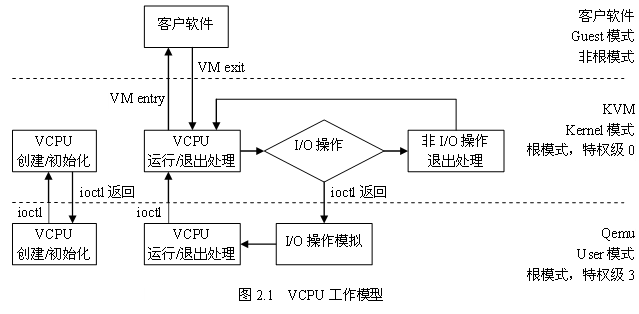
其次，VT-x为系统编程接口状态的切换提供硬件支持。VT-x为每个虚拟机维护至少一个VMCS（Virtual Machine Control Structure）结构，其中保存了虚拟机和虚拟机监控器的系统编程接口状态。当执行VM exit和VM entry操作时，VT-x自动根据VMCS中的内容完成虚拟机和虚拟机监控器间的系统编程接口状态切换。为系统编程接口状态的切换提供硬件支持是必要的，因为x86处理器的系统编程接口相比应用编程接口要复杂的多，且在不停的变化，如较新的处理器可能增加一些MSR（Model Specific Register），这使得单独依靠软件来实现系统编程接口的保存和恢复工作变得十分复杂。另外，VT-x还提供了一组指令，使得虚拟机监控器通过一条指令就可以完成虚拟机间的切换。

VT-x解决了ring deprivileging方法的一系列问题，从硬件上堵住了所谓的x86平台的虚拟化漏洞。由于操作系统所在的VMX non-root operation模式仍具有4个特权级，使得ring aliasing问题不存在了。同时，由于SYSENTER和SYSEXIT指令所引起的adverse impact on guest transitions问题也不存在了；由于VT-x在VM exit和VM entry时完成系统编程接口的切换，也就是说虚拟机和虚拟机监控器拥有各自的GDT，也就拥有了各自的地址空间，解决了address space compression问题。同时，虚拟机和虚拟机监控器拥有各自的GDTR/IDTR等寄存器，在虚拟机中访问这些寄存器无需陷入，解决了nonfaulting accessing to privileged state问题，再者，VMCS中保存了虚拟机的段描述符高速缓存，因此在虚拟机切换时不会出现access to hidden state问题；通过对VMCS进行设置，可以使处理器在VMX non-root operation模式时的EFLAGS.IF失效，即该标志位不再对中断屏蔽产生影响，因此操作系统对EFLAGS.IF的频繁操作不会导致频繁的VM exit，解决了interrupt virtualization的问题。

VT-x提供了完备的处理器虚拟化机制，利用VT-x可以在单个硬件平台上虚拟出任意数量的虚拟处理器VCPU。VT-x除了解决了处理器虚拟化的问题之外，还为内存虚拟化和I/O虚拟化提供了支撑。在内存虚拟化方面，VT-x为影子页表的实现提供了支撑，并且在较新的处理器中还提供了EPT机制，进一步提高了内存虚拟化的效率。在I/O虚拟化方面，通过I/O位图机制可以方便地实现对Programmed I/O的虚拟化，除此之外，VT-x还提供了中断事件退出机制和中断事件注入机制，方便对设备中断进行虚拟化。

**2.2. KVM实现**

作为VMM，KVM分为两部分，分别是运行于Kernel模式的KVM内核模块和运行于User模式的Qemu模块。这里的Kernel模式和User模式，实际上指的是VMX根模式下的特权级0和特权级3。另外，KVM将虚拟机所在的运行模式称为Guest模式。所谓Guest模式，实际上指的是VMX的非根模式。



利用VT-x技术的支持，KVM中的每个虚拟机可具有多个虚拟处理器VCPU，每个VCPU对应一个Qemu线程，VCPU的**创建**、**初始化**、**运行**以及**退出处理**都在Qemu线程上下文中进行，需要Kernel、User和Guest三种模式相互配合，其工作模型如图2.1所示。Qemu线程与KVM内核模块间以ioctl的方式进行交互，而KVM内核模块与客户软件之间通过VM Exit和VM entry操作进行切换。

Qemu线程以ioctl的方式指示KVM内核模块进行VCPU的创建和初始化等操作，主要指VMM创建VCPU运行所需的各种数据结构并初始化。其中很重要的一个数据结构就是VMCS，其初始化配置见附2。

初始化工作完成之后，Qemu线程以ioctl的方式向KVM内核模块发出运行VCPU的指示，后者执行VM entry操作，将处理器由kernel模式切换到Guest模式，中止宿主机软件，转而运行客户软件。注意，宿主机软件被中止时，正处于Qemu线程上下文，且正在执行ioctl系统调用的kernel模式处理程序。客户软件在运行过程中，如发生异常或外部中断等事件，或执行I/O操作，可能导致VM exit，将处理器状态由Guest模式切换回Kernel模式。KVM内核模块检查发生VM exit的原因，如果VM exit由于I/O操作导致，则执行系统调用返回操作，将I/O操作交给处于User模式的Qemu线程来处理，Qemu线程在处理完I/O操作后再次执行ioctl，指示KVM切换处理器到Guest模式，恢复客户软件的运行；如果VM exit由于其它原因导致，则由KVM内核模块负责处理，并在处理后切换处理器到Guest模式，恢复客户机的运行。

**3. 内存虚拟化**

**3.1. VT-x内存虚拟化介绍**

**3.1.1. 基于VTLB的内存虚拟化**

对于x86平台来说，处理器在加电或复位后处于实模式，此后在操作系统的操纵下一般会先切换到非分页保护模式，最后在切换到分页保护模式。VMM必须能够满足以上各种模式下的内存虚拟化要求，完全控制客户机对物理地址的访问，实现虚拟机间物理内存的隔离，并对客户机保持透明。客户机绝大多数情况下将工作在分页保护模式下，因此我们首先介绍客户机工作在分页保护模式时的内存虚拟化。

我们知道，当处理器未启动VMX模式时，页表机制确定了线性地址到物理地址的对应关系。为了加快地址转换的效率，减少地址转换过程中的页表访问次数，处理器使用TLB（Translation Lookaside Buffer）来缓存线性地址到物理地址的映射关系。实际的地址转换过程中，处理器首先根据线性地址查找TLB，如果未发现该线性地址到物理地址的映射关系（TLB miss），将根据页表中的映射关系填充TLB（TLB fill），然后再进行地址转换。因此，页表虽然确定了线性地址到物理地址的对应关系，但它并不直接控制线性地址到物理地址的转换，这种转换是由TLB控制的。在不会导致地址转换错误的前提下，处理器不要求TLB的内容一定与页表保持一致，例如一次进程切换后，TLB的内容可能都是无效的，但进程的页表却是有效的，此时页表和TLB是不一致的，以后访问内存时会引发TLB miss，并通过TLB fill令某个页表项与某个TLB项保持一致。

VT-x中可采用一种称为Virtual TLB（VTLB）的技术来实现内存虚拟化，其思想就源于上述多级页表与TLB间的交互机制，其工作机制如图3.1所示。VM的页表并不直接控制线性地址到物理地址的转换，即CR3寄存器并不指向VM的页目录，而是指向VMM维护的影子页表（Shadow Page Table），因此线性地址到物理地址的转换是由影子页表和TLB来控制的，影子页表和TLB共同构成了所谓的Virtual TLB。VM可以任意修改其页表，而不会导致VM exit，由此可能产生两种VM页表与VTLB之间的不一致情况。

第一种不一致是指VM的页表拥有比影子页表更多的访问能力，可以利用页面故障来修补这种不一致性。例如VM页表有某个线性地址到物理的对应关系而影子页表中却没有，那么VM对该线性地址的访问将导致页面故障。VMM可以捕获该异常，进而根据VM的页表来修改影子页表。

第二种不一致是指VM的页表的访问能力不及影子页表，那么一定是由于VM修改了自己的页表，并且把页表的访问权限降低了，例如页面交换将页表项的P位置0，但该操作并不会马上反映到TLB中。此时，VM需要执行INVLPG指令或重装CR3寄存器来刷新TLB，VM在执行INVLPG指令或重装CR3寄存器时将导致VM exit，使VMM有机会根据VM的页表来修改影子页表，从而修补这种不一致性。

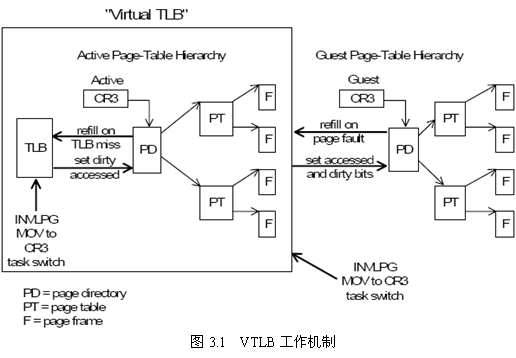
另外，处理器在进行内存访问时会自动修改页表项的A位或D位。A位在所有级别的页表项中都存在，但在TLB entry中并不缓存A位，原因是本次访存操作涉及的各个级别的页表项都允许本次访问，才会在TLB中生成一个entry，因此只要TLB中存在一个entry，那么一定已经将entry相关的各级页表项都访问过了，各级页表项的A位都是1。处理器在初次访问一个页表项时，将其A位置1，但由软件负责将A位清0。如果软件将A位由1变成0，它应该使用INVLPG指令使对应的TLB entry失效。如果不这样做，那么在访存时，总是使用TLB中缓存的entry，处理器认为页表项中的A位已经是1了，从而不会将A位由0改为1。在软件看来，好象该页自将A位置0以来未曾被访问过，而实际情况并不是这样。

D位仅在最后一级页表中存在，而且在TLB entry中也缓存D位。当首次写一个页时，如果TLB entry不存在或其D位为0，则将页表项中的D位置1，同A位相同，由软件负责将D位置1。如果软件将PTE的D位由1变成0，应该使用INVLPG指令使对应的TLB entry失效。如果不这样做，那么再次写该页时，总是优先使用缓存的TLB entry，并且TLB entry的D位为1，这时处理器认为PTE的D位已经是1了，它不会再将其置1。在软件看来，好象该页自将D位置0以来未曾被写过，而实际情况并不是这样。

由于VM进行内存访问实际使用的是影子页表，因此这种修改不能自动反映到客户页表中，从而导致客户页表和影子页表的不一致。应该有某种机制使VMM得以捕获处理器对影子页表的A位和D位的初次修改，并更新VM页表，从而解决这种不一致性。

对于A位，只要VMM不先于VM建立线性地址到物理地址的映射关系，就可以确保捕获客户对内存的访问，进而有机会确保影子页表项和客户页表项在A位上的一致。因为，当客户首次访问某个线性地址时，由于影子页表中没有该线性地址到物理地址的对应关系，将导致一次页面故障，VMM可以捕获该故障，在影子页表中建立线性地址到物理地址的对应关系，并将客户页表中相应项的A位置1。

对于D位，VMM在影子页表中建立线性地址到物理地址间的映射之初，可以将页置为只读，这样当VM对该页进行写操作时，将导致页面故障，使VMM获得控制，进而有机会确保影子页表项和VM页表项在D位上的一致。



上面提到，VMM必须能够满足以上各种处理器模式下的内存虚拟化要求，但由于VMX模式要求处理器必须工作在分页保护模式下，因此VMM不可能允许客户软件将处理器切换到实模式或非分页保护模式，而只能在分页保护模式下为客户软件模拟出类似实模式或非分页保护模式的环境。

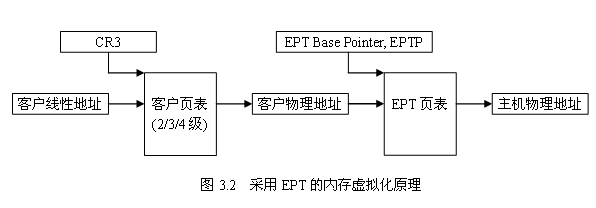
可采用一个v86任务来模拟客户机所需的实模式环境。我们知道，v86软件在执行时，其逻辑地址经16位分段机制转换为20位线性地址，其后再经过分页机制转换为物理地址。因此，在客户机引导之初，VMM可以另客户操作系统的实模式代码执行于一个v86任务中，并为其构造执行所需的页表结构，从而接管客户软件对物理内存访问。

客户的非分页保护模式代码实际上是执行在分页保护模式之下，只不过通过影子CR0等机制使客户察觉不到而已。客户逻辑地址经32位分段机制转换后为32位线性地址，而后经VMM构造的页表转换为物理地址，从而避免了客户软件对物理内存的直接控制。

**3.1.2. 基于EPT的内存虚拟化**

基于VTLB进行内存虚拟化时，因为客户在读写CR3、执行INVLPG指令或客户页表不完整等情况下均会导致VM exit，这导致了内存虚拟化效率很低。较新的Intel处理器中引入了EPT（Extended Page Table）技术，用于提高内存虚拟化的效率。

EPT引入了额外的一套页表结构，后者定义了客户物理地址到主机物理地址之间的映射关系，所有的客户物理地址（如CR3给出的页目录基址、PDE给出的页表基址、PTE给出的页基址等）须经由EPT页表结构转换后用于访存，其原理如图3.2所示。



EPT由VMM控制，并且仅在处理器工作于非根模式时才参与地址转换。采用EPT后，客户在读写CR3和执行INVLPG指令时不会导致VM exit，并且由于客户页表结构自身导致的页故障也不会导致VM exit，因此极大地提高了内存虚拟化的效率，简化了内存虚拟化的实现。

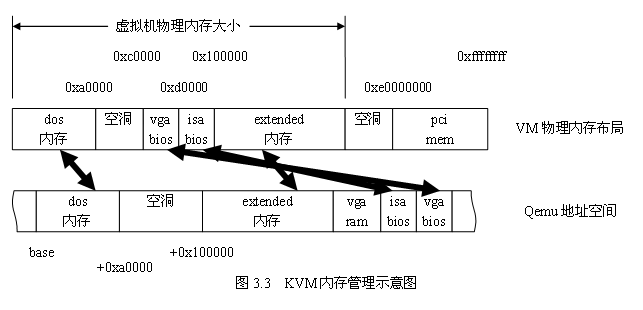
**3.2. KVM实现**

**3.2.1. 客户物理内存管理**

每个虚拟机都需要拥有一定数量的物理内存。物理内存是宝贵的资源，为了提高物理内存的利用率，也为了在一台计算机上运行尽可能多的虚拟机，不能将一块物理内存固定划分给某个虚拟机使用，而应该采用按需分配的方式。下面简述KVM的内存管理原理，参见图3.3。

PC机的物理内存通常是不连续的，例如地址0xA0000至0xFFFFF、0xE0000000至0xFFFFFFFF等通常留给BIOS ROM和MMIO而不是物理内存。设虚拟机包括n块物理内存，分别记做P1, P2, …, Pn，每块物理内存的起始地址分别记做PB1, PB2, …, PBn，每块物理内存的大小分别为PS1, PS2, …, PSn。

在虚拟机创建之初，Qemu使用malloc()从其进程地址空间中申请了一块与虚拟机的物理内存大小相等的区域，设该区域的基地址为B。



接下来，Qemu根据虚拟机的物理内存布局，将该区域划分成n个子区域，分别记做V1, V2, …, Vn，第i个子区域与第i块物理内存对应，每个子区域的起始线性地址记做VB1, VB2, …, VBn，每个子区域的大小等于对应的物理内存块的大小，仍是PS1, PS2, …, PSn。

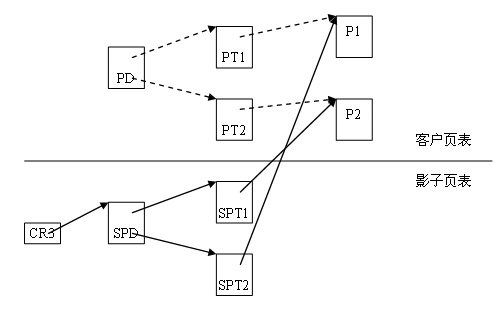
然后，Qemu向KVM内核模块通告虚拟机的物理内存布局。KVM内核模块中使用slot结构来记录虚拟机的物理内存布局，每一个物理内存块对应一个slot，其中记录着该物理内存块的起始物理地址PBi、大小PSi等信息，还记录了该物理内存块对应的在Qemu线性地址空间中的子区域的起始地址VBi。

当发生由于页故障引发的VM exit时，VMM首先搜索客户页表，如果客户页表中本身就不存在客户线性地址GVA到客户物理地址GPA的映射，则将该异常事件回注给虚拟机，由客户软件处理该页故障。如果在客户页表中存在GVA到GPA的映射，则从客户页表中得到该GPA，然后根据GPA得到其所属的slot，进而得到该GPA对应的Qemu地址空间中的主机线性地址HVA，然后通过linux内核函数get\_user\_pages()确定HVA所对应的主机物理地址HPA，如果HVA到HPA的映射不存在，get\_user\_pages()会分配物理内存，然后再建立HVA到HPA的映射。之后，VMM可以使用该HPA来构建影子页表，即建立GVA到HPA的映射。因此，KVM系统中的虚拟机所使用的物理内存是最终还是由Linux内核来分配的。

**3.2.2. VTLB实现**

我们先来看一下VTLB的基本操作。客户真正的访存是通过影子页表进行的，如果影子页表中存在客户线性地址到物理地址的映射，那么访存操作就正常进行了。如果影子页表中不存在客户线性地址到物理地址的映射，那么将引发一次页故障，从而导致一次VM exit。VMM获得控制后，将首先根据引发异常的客户线性地址去查找客户页表，如果客户页表本身限制这次访问，如到物理地址的映射不存在、违反页级保护规则等，VMM将把异常事件回注给客户，由客户操作系统处理该页故障。如果客户页表允许本次访问，那么通常本次页故障是由于影子页表中不存在客户线性地址到物理地址的映射引起的，此时就需要根据客户页表的内容来构建相应的影子页表，或称为对客户页表进行影射（Shadowing）。

如图所示，SPD是PD的影子页表，SPT1/SPT2是PT1/PT2的影子页表。由于客户PDE和PTE给出的页表基址和页基址并不是真正的物理地址，所以我们采用虚线表示PDE到客户页表以及PTE到普通客户页的映射关系。



VMM中用于影子页表的内存是受限的，因此当内存紧张时，VMM可能回收一部分影子页表。例如，可能回收图中的影子页表SPT2，以后客户访问P1时将导致页故障，VMM将再次分配影子页表，查询客户页表，并修补客户线性地址到P1的映射。

如果完全模拟物理TLB的行为，客户机在切换CR3时，VMM需要清空整个VTLB，使所有影子页表的内容无效。在多进程客户操作系统中，CR3将被频繁地切换，某些影子页表的内容可能很快就会被再次用到，而重建影子页表是一项十分耗时的工作。因此，采用完全模拟物理TLB行为的方法构建VTLB在效率上是较差的。

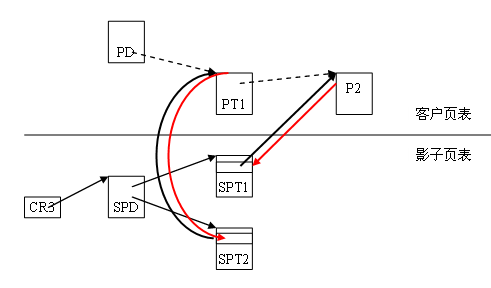
提高效率的主要做法就是缓存影子页表，即客户切换CR3时不清空影子页表。例如，假设客户机上有两个进程A和B，参见图3，在T1时刻之前A正在运行，此时CR3指向进程A的影子页表。在T1至T2时刻进程B运行，此时CR3指向进程B的影子页表，但并不丢弃进程A的影子页表。以后在T3时刻再次切换到进程A时，原来A的影子页表还可以重用，这就避免了全部重新构建A的影子页表，提高了效率。

为了实现缓存影子页表的做法，必须意识到以下问题的存在：客户可能在不通知VMM的情况下，象修改普通内存一样修改影子页表。例如，在进程B运行时，客户OS可能由于内存的紧张，将属于进程A的内存换出，并将相应的页表项的P位置0，而由于A不是当前进程，所以客户OS不会使用INVLPG指令刷新TLB，VMM也就无从得知客户修改了进程A的页表。以后，当进程A恢复运行时，由于影子页表与客户页表不一致，将导致错误。

因此，在采用缓存影子页表的做法时，必须有某种机制保持客户页表与影子页表间的一致性，这可通过为客户页表所在的页设置写保护来实现。

首先必须区别普通客户内存和客户页表，因为效率上的考虑，不能对所有的客户页面进行写保护。当一个页表没有用于访存时，VMM是无从知道该页的身份的。例如，客户操作系统在初始化某张页表时，VMM不能确定该页是普通客户内存还是客户页表，只有以后该表页用于访存时，由于在VTLB中没有影射，将导致一次VTLB Fill，并触发VMM搜索客户页表结构，从而得知与引起页面故障的客户线性地址相关的客户页面的真实身份。

VTLB Fill操作实际上在客户页表和影子页表之间进行了一次同步，为了跟踪客户页表的后续变化，应该对客户页表进行写保护。注意，客户页表也是通过影子页表来访问的，为了设置写保护就必须知道影子页表中访问客户页表所使用的PTE，为了做到这一点，KVM在影子页表中建立客户线性地址到物理地址的映射关系的同时，还维护了物理地址到末级页表PTE间的逆向映射，即给定客户页面，能够方便地得到访问该客户页面的末级页表PTE。图中，红色箭头表示逆向映射。同时，给定一个客户页面，如果其逆向映射存在，那么正向映射一定存在，即该客户页面可以通过影子页表被访问到。



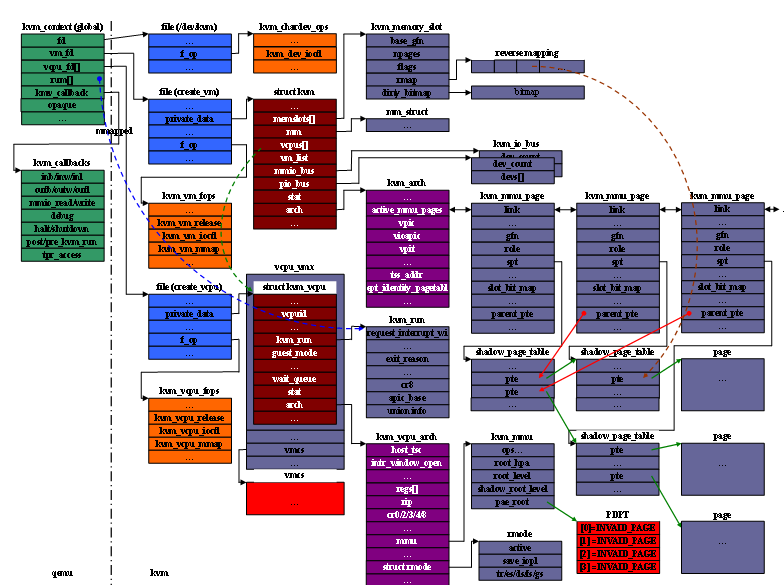
当VMM在VTLB Fill操作过程中识别一个客户页表，如PT1，就会通过逆向映射找到访问其所需的影子页表项，如SPT2中的某个PTE，将PTE的WP位置1。以后，客户对该客户页表的修改将导致VM exit，从而使VMM有机会与客户页表保持同步。

需注意的问题是，内存的紧张可能导致SPT2被回收，因此在识别到一个客户页表时，影子页表中不一定总存在到客户页表的映射，也就不能为其设置写保护。但以后客户要修改该页表时，总要首先在影子页表中建立到它的映射，在建立映射时，VMM检查该客户页面是否是客户页表且已被影射[[1]](#footnote-1)\*，如果是则置页表项的WP位为1。

还有一种可能是，影子页表中存在到PT1的映射，但PT1并没有被影射。这时，VMM不对PT1进行写保护，客户可随意修改PT1，以后当客户使用PT1进行访存时，必然会引起一次VTLB Fill，从而使VMM有机会影射该客户页表，与其同步，并在SPT2没有被回收的情况下设置对该客户页表的写保护。

由以上分析中我们还可看出，客户页表可以很大，但VMM没有必要对它们全部进行影射，VMM仅需影射那些真正用于访存的客户页表。

另外，当客户机切换到分页保护模式之后，可选择采用多种分页机制，包括普通IA-32分页机制、PAE分页机制、IA-32e分页机制和PSE-36分页机制。为了简化实现，KVM的影子页表采用PAE分页机制或IA-32e分页机制实现，客户的普通IA-32分页机制和PSE-36分页机制可以通过采用PAE分页机制的影子页表来模拟。



1. [↑](#footnote-ref-1)